NTNU
Kunnskap for en bedre verden

Department of Chemical Engineering

TKP4580 - Specialization Project

Developing an MPC Package with Julia Programming Language

Author:
Amirreza Zamani Meighani
Supervisors:
Johannes Jäschke
Mandar Thombre

Autumn, 2020

Acknowledgement

I would like to express my special thanks to professor Johannes Jäschke who introduced this project to me and my supervisor, Mandar Thombre who helped me a lot along the project. This project is really important for me because it opened a new window in mt mind to create so many ideas about continuing my study in NTNU. Secondly, I would like to thank Sandeep Prakash for all the efforts did for helping me in this project.

Contents

List of Figures

List of Tables

1 Introduction

Motivation: Why is this work important?

Working with complex cases needs to have powerful tools! Considering the huge models for industrial problems or even an experimental model which has complexity, one can think of implementing Model Predictive Controller(*MPC*). The model based controllers needs a very good model to be able to perform an acceptable performance for control. These controllers solve a dynamic optimization problem to control the plant. But sometimes increasing the number of variables in the optimization problem results in failing the solvers to solve the problem or might cause failure because of using a inappropriate method. Introducing Julia programming language as a smooth and fast high level language made some people to think about doing scientific computation problems with julia. The provided features by julia make the producing a software for MPC easier. Features like multiple dispatch or creating packages.

This work is important because it will base a background for a bigger projects in julia such that different features can be added to the package once it is created even for a very basic problem. This project considers creating a package for MPC which is able to work with different models and situations.

Abstract
Creating a package in julia for controlling with MPC is considered. To build the package, first a bioreactor has been taken as case study to write the MPC code based on that. Afterwards the scripts were transformed to take the function form and the package was created. An introduction for installing the package was presented and different commands were introduced for user to use the package. In the end the package was tested with different situations which was successful. Finally the package is published on GitHub as *PkgMPC* and is available to public.

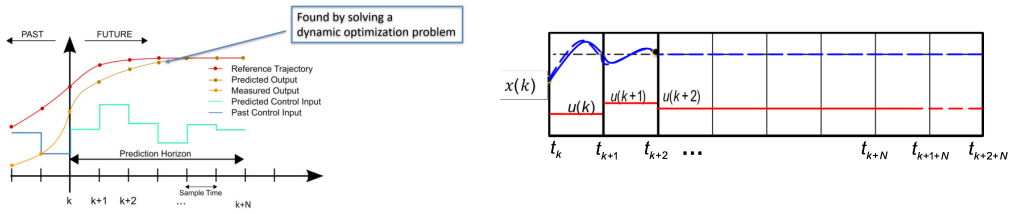
2 An introduction to MPC and Orthogonal Collocation

The work is considered as creating a package in julia to control with MPC. So before starting the work an introduction to MPC will be presented to make sure that everything is covered. Also, There will be an introduction to Orthogonal Collocation method as the main method which is used in MPC code. Note that both of these two topics are highly extensive and it cannot be a full lecture on them which is out of scope of this report. In the end a brief introduction of the case study which was used for setting up the MPC will be given.

2.1 Model Predictive Controller (MPC)

2.1.1 How does it work?

As it can be guessed by the name, the Model Predictive Controller (MPC) uses a model to predict future in order to make decision for manipulated variables. Also it should be noted that for having a reliable MPC, one needs not a perfect, but a very good model such that the outputs can be extracted somehow from the input.



A schematic of MPC Prediction Horizon

Figure 1: Two figures for the prediction horizon and the procedure of MPC working

Source: Professor Jäschke's Slides in TKP4555,as Nonlinear MPC module

As it can be seen from the 1, the MPC sees the past for model and predicting future bu solving a dynamic optimization problem. The MPC does that by changing the available input(s) to the plant. If it was a perfect model, MPC could solve the optimization problem only once and find the best input trajectory, but it is not. So, it would not be wise if we optimize the plant for the whole time once, instead there is a idea called prediction horizon which resolve that problem. The idea is to break simulation period by picking the first NFE parts pf simulation and solve the optimization problem with them. Once the optimal input vector is obtained, one can use the first element of it as the input to the plant and then it just needs to shift the window one time forward in time (finite element) and repeat these steps until the simulation horizon finished.

2.1.2 What Does It Control?

Considering all of these concepts, one might ask "So beside all of these theories, What does a MPC control?".The answer is quite obvious: It controls the objective

function! The objective function can be chosen from a range of option which makes different types of MPCs such as set point tracker MPC or Economic MPC which controls the costs directly. An example of a set point tracker is shown in2. It also has a term called *Input Usage Penalty* which is used to prevent MPC to have big jumps in input usage in order to have a more robust MPC. THis might happen when input needs to be changed considerably and heer is the place that one should prevent MPC to make too big changes in input.

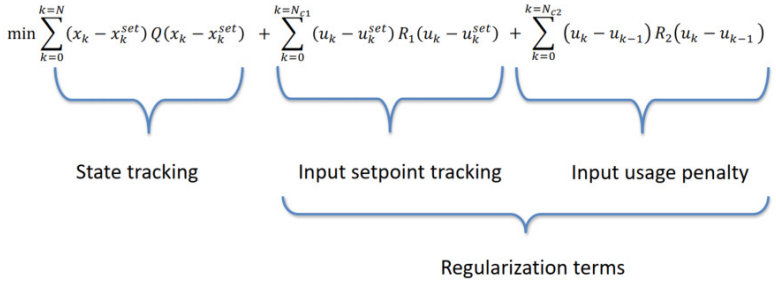


Figure 2: An example of the objective function for MPC, the sum function are for State tracking, input setpoint tracking and input usage penalty respectfully

Source: Professor Jäschke's Slides in TKP4555,as Nonlinear MPC module

The regularization terms made for MPC make the answer of MPC unique and help MPC to find the answer while it is stable. The parameters Q, R_1 and R_2 are called tuning parameters. The order of magnitude that they have defines the priorities for MPC in satisfying the objective function. For example if Q is considerably greater than R_i, it means that tracking the set points is much more important for MPC than the other two terms, but it does not mean that MPC will forget those terms. There will be a vast area of information about MPC which are expanding everyday! But considering the purpose of this project, it will be enough until now but it is advised to read them by readers themselves.

2.2 Orthogonal Collocation on Finite Elements

As it was mentioned earlier, the MPC should solve a dynamic optimization problem which means that there are some differential equations in the problem which need to be translated in a way that the solver can handle. So there should be a method for translating those equation for solver. There are a range of different methods for that such as Single Shooting, Multiple Shooting or Orthogonal Collocation.

As a brief introduction to Orthogonal collocation, one can keep the fact in mind that in shooting based methods there is an integrator which integrates the differential equations and feed them into optimizer. But it can be also done by optimizer itself. The idea of letting the optimizer do the integration also is the philosophy behind Orthogonal Collocation method.(The concept is taken from Biegler 2010)

2.2.1 Philosophy

As it was mentioned in previous part, in the collocation method the optimizer also handle the integration of differential equations beside optimizing. This is done by approximating the solution by a polynomial of order K. Considering the differential equation as what is shown below:

$$\dot{Z} = f(Z) \tag{1}$$

One can approximate the solution by:

$$Z(t) \approx A + Bt + Ct^2 + \frac{1}{3}Dt^3 \tag{2}$$

Which will create some other points between t_0 and t_f based on the order of polynomial. It can be shown graphically as what is shown in Figure 3.

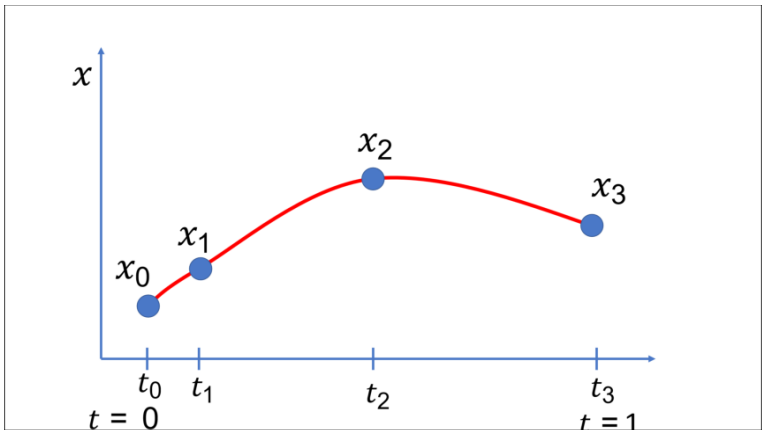


Figure 3: The arrangement of collocation points to approximate the solution by e set of polynomials

Source: Professor Biegler's slides in TKP4555 Nonlinear MPC module

Neglecting the proof, one can show that using this method would result in having the solution in the form that Equation 3 shows.

$$\begin{bmatrix} Z_1 \\ Z_2 \\ Z_3 \\ \dots \end{bmatrix} = \begin{bmatrix} Z_0 \\ Z_0 \\ Z_0 \\ \dots \end{bmatrix} + M \begin{bmatrix} \dot{Z}_1 \\ \dot{Z}_2 \\ \dot{Z}_3 \\ \dots \end{bmatrix} \tag{3}$$

Where M is a matrix of constants which depends on the chosen Lagrangian polynomial like Radau, Legendre or so on. Once one have the matrix M, it would be possible to "translate" the differential equations into an acceptable form for optimizer. Note that this method will make the size of optimization problem considerably

greater than the one in shooting methods for example. In other words one can trade non-linearity with the optimization problem size by using collocation method.

As the other numerical methods, the orthogonal collocation method will have a big error if it is applied on a long period, say from the beginning to the end of simulation horizon. Instead, one can consider applying this method on finite elements which improve the accuracy very much. There is always the option of having more finite element than collocation method or having more collocation points with less finite elements where the first idea seems to be more popular in academia.

2.3 Case Study: A Biochemical Reactor

It would be beneficial if one starts writing the code aiming for solving a problem. In that way one can write and test the written code with the case as toy problem and generalized the code afterwards. The case study for this project is a simple model of a bioreactor which has one input and one output and has two components named x_1 and x_2. A schematic of the reactor is shown in Figure 4.

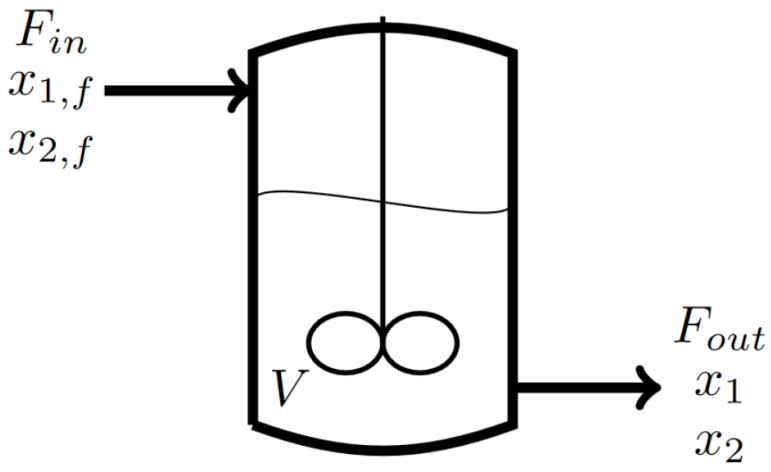


Figure 4: The bio reactor schematic with the input and output

Source: Kuure-Kinsey et al. 2005

The model for the reactor can be summarized what is shown below:

$$\mu_{monod} = \frac{\mu_{max} x_2}{K_m + x_2} \tag{4}$$

$$\frac{\mathrm{dx_1}}{\mathrm{d}t} = -Dx_1 + \mu x_1 = (\mu - D)x_1 \tag{5}$$

$$\frac{\mathrm{dx_2}}{\mathrm{d}t} = D(x_{2,f} - x_2) - \frac{\mu x_1}{Y} \tag{6}$$

and the control problem can be shown as:

$$\min_{x,u} \frac{1}{2}\left(\sum_{i=1}^{n_p} (y_i - y_{SP,i})^T Q (y_i - y_{SP,i}) + \sum_{i=1}^{n_m} \Delta u_i^T R \Delta u_i \right)$$

$$\begin{aligned}
&x_0 = x(0) && \\
&x_{i+1} = F(x_i, u_i) && i = 0, \ldots, n_p - 1 \\
&y_i = H(x_i) && i = 1, \ldots, n_p \\
&u_{\min} \leq u_i \leq u_{\max} && i = 1, \ldots, n_p \\
&-\Delta u_{\max} \leq \Delta u_i \leq \Delta u_{\max} && i = 1, \ldots, n_m \\
&\Delta u_i = 0 && i = n_m + 1, \ldots, n_p
\end{aligned}$$

Figure 5: The setpoint tracking MPC problem

Source: Professor Jäschke's Slides in TKP4555,as Nonlinear MPC module

Considering the model and the MPC problem, one can comprehend that there is a dynamic optimization problem which needs to be solved. For the rest of the report there is going to be the explanation of developing a package in julia which is able to solve this problem.

3 Calculation Procedure

3.1 Package Structure

The package is constructed with three main modules with two other complementary files where the parameters are loaded and sent to the main files. The main modules are *Main.jl*, *NMPC.jl* and *Plant.jl* in the following parts, each of them will be illustrated in case of the code arrangement and how they are connected.

3.2 Main.jl

This file contains information about simulation of the process, plotting and exporting data outside of the program. Starting *main.jl*, all the parameters which are needed for simulation procedure is loaded from the *Parameters.jl* which contains all the parameters. The command *include(File.jl)* loads the files that will be used in rest of the program. This can be done in *julia* like what is shown below.

```
include("Plant.jl")
include("NMPC.jl")
include("Parameters.jl")

using Plots

## Defining Simulation Parameters



sim_par = Simulation_Data()
    T0_sim  = sim_par[:T0_sim]
    Tf_sim  = sim_par[:Tf_sim]
    dt_sim  = sim_par[:dt_sim]
    Tf_MPC  = sim_par[:Tf_MPC]
    dt_MPC  = sim_par[:dt_MPC]
    NCP     = sim_par[:NCP]
    NFE_sim = sim_par[:NFE_sim]
    NFE_MPC = sim_par[:NFE_MPC]
    uk      = sim_par[:uk]
    xsp     = sim_par[:xsp]
    xk      = sim_par[:xk]
    U_Plot = [];
    X1_Plot = [];
    X2_Plot = [];
    # push the starting point into the final vectors
    push!(U_Plot, uk[1])
    push!(X1_Plot, xk[1])
    push!(X2_Plot, xk[2])
```

After initializing the parameters, it is possible to turn some features on/off like displaying the final figure, save that as a file or exporting the results in *.CSV* format.

```
## Options for plotting, saving the figure and Eporting the data
Plotting    = true;
```

```
Fig_Save    = true;
Export_Vars = true;
```

Now the simulation part can begin to start, This part is nothing than a *for* loop which first simulates the optimization of the plant in order to set the optimal input also then feeding the optimal input to the plant to achieve the new states. At the end of each iteration, the achieved date will be pushed into some vectors for the plotting part. The loop for simulating the process is shown below:

```
## Simulation of the plant
println("Start Simulation!")
for k in 1:NFE_sim
    global xk, uk, dt_sim
    x_sp_MPC = x_sp[k:k - 1 + NFE_MPC]
    uk = Solve_MPC(xk, uk, x_sp_MPC,NFE_MPC, NCP)
    xk = Plant(xk, uk, dt_sim);
    push!(U_Plot, uk[1])
    push!(X1_Plot, xk[1])
    push!(X2_Plot, xk[2])
end
println("The objective function is satisfied successfully!")
```

After running the for loop successfully, the results of optimization are available and can be used to be plotted or saved as a file for further usage. This is done like what is shown below.

For Plotting

```
## Plotting with the option to show/not show the figure. Also different backends can be
↪  choose!

t_plot = collect(T0_sim:dt_sim:Tf_sim)
p11 = plot( t_plot,[X1_Plot[nIter]  for nIter in 1:NFE_sim+1],label="x1")
p11 = plot!(t_plot,[X2_Plot[nIter]  for nIter in 1:NFE_sim+1],label="x2")
p11 = plot!(t_plot, x_sp[1:NFE_sim + 1],                      label="y_sp",
↪  linetype=:steppost, linestyle=:dash)
p12 = plot(t_plot[1:end], [U_Plot[nIter]  for nIter in 1:NFE_sim+1],    label="u",
↪  linetype=:steppost)
fig1 = plot(p11, p12, layout=(2, 1));
#Checking the choosed Options for plotting
if Fig_Save == true
    savefig(fig1,"MPC_Results.eps")
end

if Plotting == true
        plotlyjs()
        #gr()
        println("Start Plotting!")
        display(fig1)
        println("Finish Plotting!")
end
```

For exporting data

```
## Exporting Data
if Export_Vars == true
    println("Start Exporting Data!")
    using CSV
    using DataFrames
    df = DataFrame(Name = ["x1","x2","U","SetPoint","Time"],
                   Value = [X1_Plot, X2_Plot2, U_Plot, x_sp[1:NFE_sim+1], t_plot]
                   )
    CSV.write("MPC_Results.csv", df)
    println("Finish Exporting Data!")
end
```

In the following parts,two other files which have been used in *main.jl* are explained.

3.3 NMPC.jl

Starting the simulation, the input to the plant might not be the optimal value regarding the starting point for the states. So it is better to start the simulation with MPC to control even the first input to the plant. As it was explained before the method used for translating the dynamic optimization into the form that the the optimizer can read is using the orthogonal collocation on finite elements. The general procedure of work in *NMPC.jl* is to start with loading the required parameters from *Parameters.jl* after that setting the constraints and the bounds for optimization variables. Stating the objective function and applying the orthogonal collocation as a constraint, optimization for the whole problem is done by the command *optimize!(Model)*. In the end the results can be extracted from the *IPopt* outputs.

In each iteration of the simulation, the MPC takes *NFE_MPC* (e.g. 8) parts of the simulation finite elements as prediction horizon with the same length of set point vector to calculate optimal trajectory for the plant. In the end, the first element of MPC optimal trajectory is used as the output of MPC function to feed the plant with. This function uses Orthogonal Collocation method to solve the differential equation of each finite elements such as all of the equations for all *NFE_MPC* elements are written as constraints ans create a system of equations which can be solved by optimizer. Now the explanation of each part of the *NMPC.jl* will be illustrated. A line-by-line description of the *NMPC.jl* is written for the rest of this part.

Starting the code, one should define what is going to be used as packages for calculation. This is done by putting *using* command in julia. For the optimization the packages Ipopt and JuMP are used to define the dynamic optimization problem and other requirements for that such as constraints or bounds and so on. Also the code uses the data from *CollMat.jl* function which provides the M matrix for orthogonal collocation. This function will be illustrated later in subsection 3.5

```
using Ipopt, JuMP
include("CollMat.jl")
```

Then the function *Solve_MPC* is introduced to set the optimizer up by taking 5

inputs which are the states X0, the input to the plant U0, a vector a setpoint, the number of finite elements (prediction horizon) and the number of collocation points NCP. After that the model is defined using IPopt feature and some of the required parameters for starting the work are loaded from *Parameters.jl*. Also the matrix M is built with given NCP and using *CollMat.jl*. Until this part can be done like what is shown below.

```
function Solve_MPC(x0, u0, x_sp, NFE, NCP)

m1 = Model(Ipopt.Optimizer)
Nx  = size(x0, 1);
Nu  = size(u0, 1);
dx0 = 0*copy(x0)
q0  = 0;
dq0 = 0;

model_par = Model_par()
    sf  = model_par[:x2_f]
    km    = model_par[:km]
    k1    = model_par[:k1]
    Y     = model_par[:Y]
    μmax = model_par[:μmax]

M = Collocation_Matrix(NCP)
```

Now it is required to define the variables for optimization and their bound and starting points (if needed). In this part the variables are created for the whole prediction horizon not only for one element and using a *for* loop for building the problem like what is used in MATLAB when the problem is solved by CasADi. This makes the optimizer much faster than the same problem solved by MATLAB because once a big system of variable is created, all of the constraint and other equations can be defined and the optimizer need to solve the problem for only one time instead of solving it for *NFE* times. The way which constraints and other equations are defined is illustrated below:

```
##Defining the variables for the whole NFE_MPC horizon
@variable(m1, x[1:Nx, 1:NFE, 1:NCP+1]);
@variable(m1, dx[1:Nx, 1:NFE, 1:NCP]);
@variable(m1,  q[   1,       1:NFE, 1:NCP])
@variable(m1, dq[   1,       1:NFE, 1:NCP])
@variable(m1, u[1:Nu, 1:NFE])
##Setting up the bounds for variables (in case of need)
for nx in 1:Nx, nu in 1:Nu, nfe in 1:NFE, ncp in 1:NCP
    set_lower_bound(x[nx, nfe, ncp], 0)
    set_upper_bound(x[1 , nfe, ncp], 4.5)
    #set_lower_bound(dx[nx, nfe, ncp], 0)
    #set_upper_bound(dx[nx, nfe, ncp], 999)
    set_lower_bound(u[nu, nfe], 0)
    set_upper_bound(u[nu, nfe], 1)
end
##Setting up the starting points for variables
for nx in 1:Nx, nu in 1:Nu, nfe in 1:NFE, ncp in 1:NCP
    set_start_value(x[nx, nfe, ncp],    x0[nx])
    set_start_value(dx[nx, nfe, ncp],dx0[nx])
    set_start_value(u[nu, nfe],         u0[nu])
```

```
    set_start_value(q[1, nfe, ncp],     q0)
    set_start_value(dq[1, nfe, ncp],    dq0)
end
##Rename some of the variables to write the equations easier
@NLexpressions(m1, begin
    x1[nfe in 1:NFE, ncp in 1:NCP],          x[1, nfe, ncp]
    x2[nfe in 1:NFE, ncp in 1:NCP],          x[2, nfe, ncp]
    D[nfe in 1:NFE],                         u[1, nfe]
end)
##Constraints defining region!


#Here is where you should defining the model ODEs in each line
@NLconstraints(m1, begin
Constr_ODE1[nfe in 1:NFE, ncp in 1:NCP], dx[1, nfe, ncp]      == (((μmax * x2[nfe, ncp]) /
  ↪  (km + x2[nfe, ncp] + k1 * x2[nfe, ncp]^2)) - D[nfe]) * x1[nfe, ncp];
Constr_ODE2[nfe in 1:NFE, ncp in 1:NCP], dx[2, nfe, ncp]      == (sf - x2[nfe, ncp]) *
  ↪  D[nfe] - (((μmax * x2[nfe, ncp]) / (km + x2[nfe, ncp] + k1 * x2[nfe, ncp]^2))/Y) *
  ↪  x1[nfe, ncp];
#For more ODEs:
#Constr_ODEi[nfe in 1:NFE, ncp in 1:NCP], dx[i, nfe, ncp] = Equation!
end)


#Defining the Quadrature Equations
@NLconstraints(m1, begin
      Constr_dq0[nfe = 1    , ncp in 1:NCP], dq[1, nfe, ncp]  == (x1[nfe,ncp] -
      ↪  x_sp[nfe])^2 + 0.5*(D[nfe]  - u0[1]    )^2
      Constr_dq[nfe in 2:NFE, ncp in 1:NCP], dq[1, nfe, ncp]  == (x1[nfe,ncp] -
      ↪  x_sp[nfe])^2 + 0.5*(D[nfe] - D[nfe-1])^2
end)


#Constraints on input usage (To avoid big jumps on input usage)
@NLconstraints(m1, begin
      #Defining Inequality Constraints in each line
      Constr_Ineq1[nfe in 1:1   ],   -0.08 <=   D[nfe] - u0[1]       <=  0.08
      Constr_Ineq2[nfe in 2:NFE-3],   -0.08 <=   D[nfe] - D[nfe-1]    <=  0.08
      Constr_Ineq3[nfe in NFE-2:NFE ],     D[nfe] - D[nfe-1]    == 0.0
end)
```

Where the three numbers in the notations for variables define the number of variable, number of finite element and number of collocation points. For instance, the variable X_1 in the first finite element can be illustrated as what is shown in the figure below:

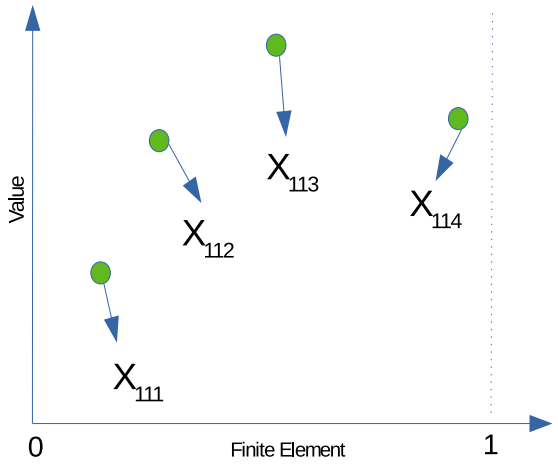


Figure 6: Notation of different variables which are created from X_1 with 4 collocation points in the first finite element

Now the only things needed are to apply orthogonal collocation and state the objective function. In order to apply collocation equation as constraint for optimizer, one can follow this procedure:

As it can be seen in "variable defining" part, there are *NCP+1* points for each finite element. The reason is that following this idea, it would be possible to write the collocation equations for each finite element and then fix the continuity with saying that the last point in $(\text{F.N.})_i$ is equal to the first point in $(\text{F.N.})_{i+1}$. The only exceptional finite element are the first element which is compared with the starting point (*X0*). For example with 3 collocation points, one can write these equations for the points in thew second finite elements:

$$\begin{aligned} x_{122} &= x_{121} + M[1,:] * dx[1,2,:] \\ x_{123} &= x_{121} + M[2,:] * dx[1,2,:] \\ x_{124} &= x_{121} + M[3,:] * dx[1,2,:] \\ x_{121} &= x_{114} \end{aligned} \tag{7}$$

and this would be the same for all finite elements and variables that need to be translated by orthogonal collocation which are the states and quadrature(s). The julia version of 7 for all finite elements results in what is written in *NMPC.jl*:

```
#Collocation Equations for states and quadratures as Constraints
@NLconstraints(m1, begin
        Coll_Eq_Diff[nx in 1:Nx,  nfe = 1:NFE, ncp in 1:NCP],     x[nx, nfe, ncp+1] ==
        ↪  x[nx, nfe, 1] + sum(M[ncp, i] * dx[nx, nfe, i] for i in 1:NCP)
        Cont_Eq_First[nx in 1:Nx],                                 x[nx, 1, 1] == x0[nx]
        Cont_Eq_rest[nx in 1:Nx,  nfe = 2:NFE],                    x[nx, nfe, 1] == x[nx,
        ↪  nfe-1, end]
        Coll_Eq_Quad0[             nfe = 1, ncp in 1:NCP],         q[1, nfe, ncp] == q0  +
        ↪  sum(M[ncp, i] * dq[1, nfe, i] for i in 1:NCP)
        Coll_Eq_Quad[       nfe in 2:NFE, ncp in 1:NCP],           q[1, nfe, ncp] == q[1,
        ↪  nfe-1, NCP]  + sum(M[ncp, i] * dq[1, nfe, i] for i in 1:NCP)
        end)
##And finally Defining the Objective Function!
@NLobjective(m1, Min,  sum( (x1[nfe,NCP] - x_sp[nfe])^2   for nfe in 1:NFE )  + 0.05*(D[1]
↪  - u0[1])^2 + sum( 0.05*(D[nfe] - D[nfe-1])^2 for nfe in 2:NFE)  )
```

Now, one can simply optimize the whole problem using the command *optimize!(m1)*. The results from optimization can be extracted and transformed to an acceptable Julia form. In the end the first item of the optimal input vector will be the output of the *NMPC* function.

```
## Calling the optimizer to optimize the whole prediction horizon
optimize!(m1)
JuMP.termination_status(m1)
JuMP.solve_time(m1::Model)
## Extarcting the results and reshape them into julia form

star_x = JuMP.value.(x[:, :, NCP])
star_x = cat(x0, star_x, dims = 2)
star_u = JuMP.value.(u)
star_x1 = JuMP.value.(x1[:, NCP])
star_x2 = JuMP.value.(x2[:, NCP])
star_x1 = cat(x0[1], star_x1, dims = 1)
star_x2 = cat(x0[2], star_x2, dims = 1)
star_D = JuMP.value.(D)
star_MPC = star_u[:,1]
##The output of the Function!
#Export the first input among NFE_MPC inputs as the output to be fed into the plant
return star_MPC

end
```

3.4 Plant.jl

Having obtained the optimal input for the plant, one should design another function which plays the role on behalf of the real plant to be able to take the input and generate new states. This can be done by implementing an ODE solver to solve the sets of differential equations to calculate the new states. The package *DifferentialEquation.jl* is used to solve the ODEs. The *Plant.jl* file uses the data coming from *Parameters.jl* and put them as parameters for ODEs. The function *Plant* takes 3 inputs which are the *states*, *Optimal input to the plant* and dt for integration. It has also another function inside itself to generate the differential equations. Generating the equations, one can substitute the inputs as initial values for equations and define the time of integration. Finally a solver such as Tsit5() can solve the problem and the result (new states) can be exported as the output of the function. This is done in julia like what is shown below:

```
include("Parameters.jl")
using DifferentialEquations

function Plant(xkold,ukold,dt)
    function Sgen(dx,x,p,t)
                model_par = Model_par()
                        x2_f  = model_par[:x2_f]
                        km    = model_par[:km]
                        k1    = model_par[:k1]
```

```
                        Y     = model_par[:Y]
                        μmax  = model_par[:μmax]
        μ = (μmax * x[2]) / (km + x[2] + k1 * x[2]^2);
        D = x[3];
        dx[1] = (μ - D) * x[1];
        dx[2] = (x2_f - x[2]) * D - (μ/Y) * x[1];
    end
    x0 = [xkold[1],xkold[2]]
    D0 = [ukold[1]]
    U0 = vcat(x0,D0)
    tspan = (0.0, dt)
    prob = ODEProblem(Sgen,U0,tspan)
    sol = DifferentialEquations.solve(prob, Tsit5())
    xf = sol.u[end][1:2];
return xf
end
```

3.5 Supplementary files

There are also two other Supplementary files named *Parameters.jl* and *CollMat.jl* which are providing the data for required parameters and the M matrix which is used in collocation equations. These files are listed in appendices E and F respectively.

4 How to use the package

For making a package-form out of the code, one should do some transformation on the scripts and change them into functions which can take some date and generate the results. For example the *main.jl* was only a script which was able to solve the problem by running the script by user, but that needs to be in a function form in julia. There were 3 main file and 2 other supplementary files which are broken into some function each of which doing an specific part of the program such as solving the optimization problem, plotting and so on. All the functions will be explained later in 4.2.

4.1 Install the Package

The package which is called *PkgMPC* is available on GitHub (Address) and can be downloaded by everyone who has julia installed on their computers. PkgMPC is not registered yet which mean one should put the address of the GitHub repository instead of putting the name the package. For installing the package, one can follow this procedure in julia REPL:

] add "https://github.com/Amirrezz94/PkgMPC.jl"

This will install the package and all dependencies which are required such as Ipopt, JuMP, Plots and so on. It is recommended to have these packages installed before using PkgMPC:

- Reexport
- DifferentialEquations
- JuMP
- Ipopt
- Plots
- PlotlyJS
- CSV
- DataFrames

After installation, one can check the status by typing what is shown below in julia REPL:

]st PkgMPC

which should give the result like this:

```
(@v1.4) pkg> st PkgMPC
Status `~/.julia/environments/v1.4/Project.toml`
```

```
  [aa087d6f] PkgMPC v0.1.0 #master (https://github.com/Amirrezz94/PkgMPC.jl)

(@v1.4) pkg>
```

Also for updating the package, one can use:

]up PkgMPC

Now the PkgMPC is installed on julia and ready to action!

4.2 Available Commands

As it was mentioned in previous parts, the scripts has been changed and updated to be in a *Function* form in julia so each function can be one of the commands in the package. There are plenty of commands in PkgMPC which can be used to solve the optimization problem, plot the result and so on. Now a brief introduction for each command is considered.

4.2.1 Model_Par

This is a function which sets the required parameters for the ODEs and creates a dictionary as output which can be sent to ODE solver or other places if needed. For instance, in the bio-reactor case study, one can use these information to create the dictionary:

```
julia> using PkgMPC

julia> Data_Plant = Model_Par(4.0, 0.12, 0.4545, 0.4, 0.4)
Dict{Symbol,Float64} with 5 entries:
  :μmax => 0.4
  :km   => 0.12
  :Y    => 0.4
  :x2_f => 4.0
  :k1   => 0.4545

julia>
```

4.2.2 Simulation_Data

This function is almost the same function with previous one but it sets the information which are needed for setting up the optimization problem or the plant. The inputs for this function are the length of simulation horizon (Tfsim), the the length of prediction horizon of MPC (TfMPC), number of collocation points which can be selected from 1 to 5 (Ncp), starting point for states and starting point for input(s). It can be summerized as :

Output_Dict = Simulation_Data(Tfsim, TfMPC, Ncp, states, inputs)

One can use these information to create the dictionary:

```
julia> using PkgMPC

julia> Data_MPC   = Simulation_Data(60.0, 8.0, 3, [1.0; 1.0], 0.35)
Dict{Symbol,Any} with 11 entries:
  :x_sp     => [1.5032 1.5032 ... 0.0 0.0]
  :dt_MPC   => 1
  :uk       => 0.35
  :xk       => [1.0, 1.0]
  :NFE_MPC => 8
  :dt_sim   => 1
  :Tf_sim   => 60.0
  :NCP      => 3
  :T0_sim   => 0.0
  :Tf_MPC   => 8.0
  :NFE_sim => 60

julia>
```

4.2.3 Simulate_Plant

This is the main function of the main package where the MPC starts to work on the created plant considering the information which are stored in two mentioned dictionary. The general form of this function can be summerised as:

Simulate_Plant(Data_MPC, Data_Plant)

where Data_MPC and Data_Plant are two dictionaries obtained from 4.2.2 and 4.2.1 respectively. Once it is executed, the data will be sent to specific places in Package which results in executing the for loop mentioned here with specified information and controlling the plant with MPC to satisfy the objective function (which can be set point tracking). The function provides some messages to let user know in which stage the optimizer is such as the Ipopt output messages, starting the simulation and messages if the simulation finished successfully. It also export one dictionary as the result containing the states during the simulation and a vector (matrix) of inputs that MPC applied to the plant.It can be used in the way which is shown below:

Results = Simulate_Plant(Data_MPC, Data_Plant)

4.2.4 Plot_Plant

Plotting the result can be considerably beneficial for better understanding of the MPC. This can be done using the function *Plot_Plant*. This function takes the result of 4.2.3 and the information obtained from 4.2.2 as inputs and create a figure showing the results graphically. It can be used in the way which is shown below:

Figure = Plot_Plant(Results, Data_MPC)

4.2.5 Save_Plant

Exporting the results to outside of the program is a key to keep the important data for further analysis or comparing with other cases. The figure made in 4.2.4 can be saved in a range of formats which can be picked by user. It also takes an address for saving the file and finally save the figure in specified directory. An example of using the plotting command in PkgMPC is shown below:

Save_Plant(Figure,"//home//amir//Documents//Package_Results//MPC_Results.svg")

This will save a file named *MPC_Results.svg* in the directory *...//Package_Results*. A range of other formats can be used instead of .svg such as *.eps* or non_vectorized fromat: *.jpg* or *.png* .

4.2.6 Export_Plant

This is anothe options of PkgMPC for exporting not the figures but the data as a file. The command *Export_Plant* will take the data obtained from MPC and the data obtained from 4.2.2 and save the outputs of the program (states and optimal inputs and setpoint) in the specified directory. This function can be used in the way which is stated below:

Export_Plant(Results,Data_MPC, "//home//amir//MPC_Results.csv")

Where the *Results* and *Data_MPC* are obtained from 4.2.3 and 4.2.2 respectively.

5 Improvements and Performance

5.1 Improvements

This package has another version in MATLAB which can perform optimization and control a plant with MPC using Orthogonal Collocation method which makes it a good case for comparing different factors in MATLAB version of the program and Julia version of it. These two program can be compared from different points of view such as execution time, the price of buying the license or other features in MATLAB and Julia. for the rest of this part, there is going to be a comparison between these factor. Note that the MATLAB version of the program is available upon the request from the author (*Postdoctoral Fellow* 2020)

5.2 Execution Time

Speed is one of the most important factors when it comes to analysing the performance of a program. It might not be the case in small scale problems, but in industrial cases the execution speed might make a considerable difference. So the first factor for making comparison is the execution time of running the same problem in MATLAB and julia. It has been done with considering the bio-reactor case study with the same parameter and simulation horizon in MATLAB and julia. It turned out that the julia version is considerably faster than MATLAB. In MATLAB there is a function called *tic-toc* which calculates the execution time between two lines of the code. For the simulator For-Loop (Without Plotting) it takes 23.567436 seconds to run the plant for NFE_Sim = 60 while it's just 2.559795929 seconds in julia to run the *SimulatePlant* command. In julia, the command *@elapsed* is used to calculate the execution time. The elapse time for executing each of commands in PkgMPC is shown in Table 1. Note that the Precompiling time is excluded from the numbers because that time is just for one time for loading the dependencies.

Name	Elapsed Time (sec)
Simulation_Data	9.316e-5
Model_Par	3.907e-6
Simulate_Plant	2.521711482
Plot_Plant	4.941620962
Save_Plant	0.602635045
Export_Plant	0.000587409

Table 1: Execution Time for different commands in PkgMPC

5.3 Privileges of Having Package Form

Having compared a single code in any language with a program which can handle different problems and situation, one can comprehend that the second case might be a better case to be used by user. Because in a package form of a code, user can change the properties of the problem without changing the scripts directly and so can be more convenient. For instance, in the PkgMPC package, user can change the simulation horizon or the number of collocation point without knowing how the

different functions are managed and working, it is just a change in a parameter. So, in the author's opinion it can be more convenient and helpful to have a package form for the codes so everyone can use it easier! The other matter is the opportunity for developing the work can be more than the situation that only a script is written for a specific problem. For instance, people can contribute in different parts of the project to develop them in a better way than when they should know almost everything about the code in a single script, because in a package there are different functions which can be connected together. All in all, while writing the code in MATLAB and using the features that MATLAB provides are beneficial, it could be done in a more convenient and faster way in julia when the scripts are written in a package form.

5.4 Plotting Options

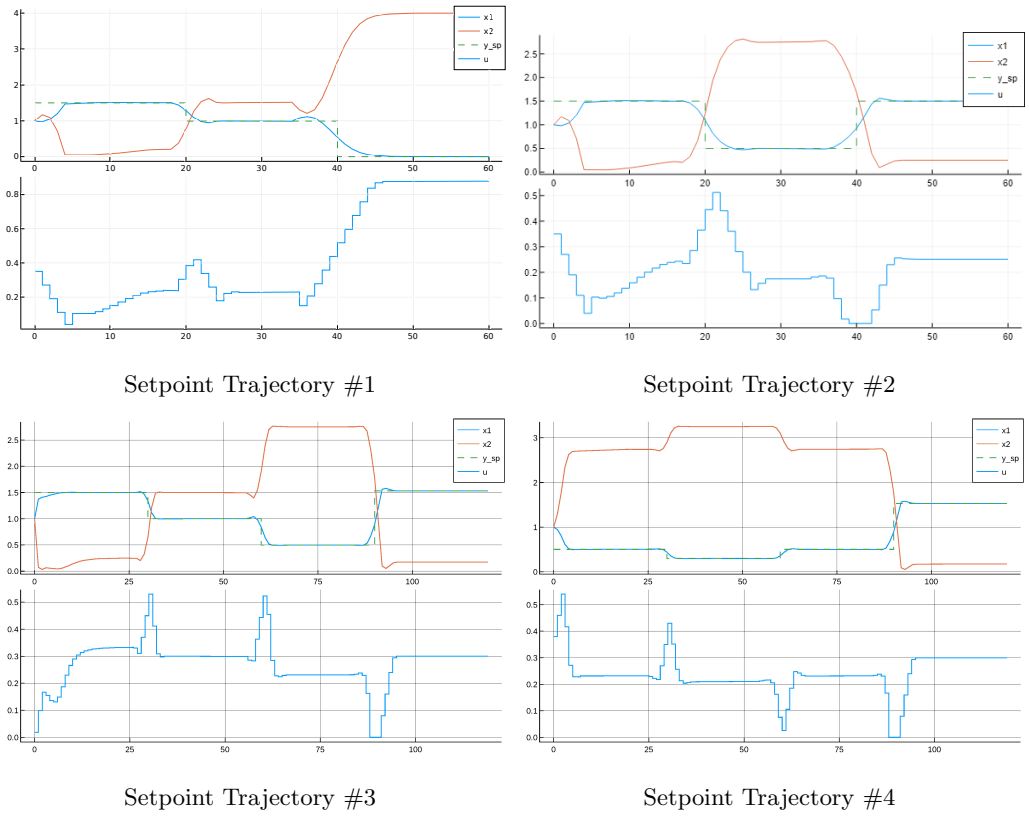
When it comes to analysing the data, visualizing the result plays a key role. In julia language there is a feature which user can choose between different backends for plotting. This will make a huge difference between the plots created in MATLAB and the same in julia. It might be possible to create the same quality plots in both languages, both considering the time and needed effort for making them, julia will occupy a better ranking. Considering different backends in julia, it is possible to change them just by choosing, for example, *PlotlyJS()* or *GR()*. Some of the most popular backends in julia can be found in julia's documentation for backends

5.5 Open Source or Licensed

In almost every kind of project, the cost of using the software might be very high. So, using the open-source software or languages could be beneficial considering the economic points of view. MATLAB counts as an expensive software to work with while julia language is a open-source language. Julia might not have the best developed toolboxes compared to MATLAB; but, considering the license cost and the open source packages in that, one can consider it as a good tool to work with.

5.6 Performance of the Package for Different Situations

Considering all the features and properties explained in previous parts, one can ask that how can the package be tested? The answer is quite obvious which is to test the package in different situations such as set point trajectory changing or changing the parameters, In this part the package is tested in four different types of set point trajectory to see the performance of it. The results are shown in 7. As it can be seen from the plots, one can figured out that the controller controlled the plant well and the state X_1 follows the setpoint smoothly. Also the controller seem to be smooth and the input changing does not have big jumps. In other words, the controller seems to be robust.



Setpoint Trajectory #1 Setpoint Trajectory #2

Setpoint Trajectory #3 Setpoint Trajectory #4

Figure 7: Testing the package with 4 different setpoint trajectories. Results are obtained with the bioreactor case study.

5.7 Opportunities for Future Work

One of the best thing about this project is that it is not about to finish something but to start to create a new environment which opens a range of new ideas to work on. The package itself might be simple but it sets a background for considerable number of ideas to be developed. For example the package PkgMPC only accepts ODEs which can be updated with DAEs which is more complicated or the method used for translating the dynamic optimization was Orthogonal collocation which can be supported by other methods such as shooting methods. In this part, some future work which can be added to this project are presented.

Considering developing the package one can starts with developing the methods inside the package or work on the software engineering aspects of the package such as how to make it more user friendly. Some of the methods which might be added to the package can be listed as:

- Adding other solvers for solving the differential equation system which can be chosen by user such as the wide range of methods *DifferentialEquations.jl* can support. (Rackauckas 2020)
- Adding new optimization methods to widen the range of supported problems.
- Importing Machine Learning to the package so that it would help a lot in complex cases which finding the model is not "Easy" or faces difficulties to use conventional methods.

- the orthogonal collocation method, the package uses Radau Lagrangian polynomials which can be updated by other polynomials such as Legendre which is higher in order of accuracy
- Different types of MPC can be applied to the package so the user can just "choose" the objective function and sets the parameter. Also robust MPC can be considered.
- and **SO MANY MORE IDEAS**!

And for the software aspects of the project, one can consider these items:

- Adding different properties for the package which make it more "dynamic" such as *Continues Integration (CI)* which will test the package continuously.
- Managing the function and connection between inputs and outputs to make it more *User-Friendly*. For instance some of the functions are fixed with the first version of the package which can be transformed to be a "Command" in the package.
- Enabling multiple dispatch which is a key to make the package much more faster than the other versions in MATLAB or other languages.
- Enabling the functionality for the package to make it able to decide about considered method based on the user input. For example for solving the MPC problem, only one function is defined which can have different methods based on the inputs users give to it.
- and **SO MANY MORE IDEAS**!

6 Conclusion

Different aspects of the project have been discussed throughout the report such as development of the code, method and results. Keeping all those in mind, one can conclude that the available pack is able to perform an acceptable performance on controlling the plant with different setpoint while it might not be as effective as it was for bioreactor case when it comes to handling different ODE systems. But the work opened a door to a range of new ideas for developing the package to be able to show a better performance.

Also the differences between the julia version and the MATLAB version of the code were illustrated which can be summarized as: while MATLAB has very strong performance on matrix calculation, it might not be the best case for handling the project because of some limitations in coding when the project is handled by CaSADi in MATLAB. The limitations are that the problem must be built in a *for loop* which makes it slower than julia in case of execution time.

Bibliography

Biegler, Lorenz T. (2010). *Nonlinear Programming*. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898719383. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719383. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898719383.

Kuure-Kinsey, M. et al. (July 2005). 'Modeling and predictive control of a rotating disk bioreactor'. In: vol. 5, 3259–3264 vol. 5. DOI: 10.1109/ACC.2005.1470474.

Postdoctoral Fellow (2020). Jose Otavio Assumpcao Matias.

Rackauckas, Christopher (2020). *DifferentialEquations.jl*. URL: https://github.com/SciML/DifferentialEquations.jl (visited on 18th Dec. 2020).

Appendix

For the most updated versions of files please go to : PkgMPC GitHub Page

A PkgMPC.jl

```
module PkgMPC

# Write your package code here.
using Reexport
@reexport using Plots
@reexport using CSV
@reexport using JuMP
@reexport using Ipopt
@reexport using DataFrames
@reexport using DifferentialEquations
@reexport using PlotlyJS

include("Extra.jl")
include("main.jl")
include("NMPC.jl")
include("Plant.jl")


export F, Model_Par, Simulation_Data, Simulate_Plant
export Simulate_Plant, Plot_Plant, Save_Plant, Export_Plant

end
```

B main.jl

```
include("Plant.jl")
include("NMPC.jl")
include("Parameters.jl")
using Plots

## Defining Simulation Parameters



## Options for plotting, saving the figure and Eporting the data
#Plotting    = false;
#Fig_Save    = false;
#Export_Vars = false;

## Simulation of the plant
function Simulate_Plant(Data_MPC, Data_Plant)

    #sim_par = Data_MPC;
        T0_sim  = Data_MPC[:T0_sim]
        Tf_sim  = Data_MPC[:Tf_sim]
        dt_sim  = Data_MPC[:dt_sim]
        Tf_MPC  = Data_MPC[:Tf_MPC]
        dt_MPC  = Data_MPC[:dt_MPC]
```

```
        NCP     = Data_MPC[:NCP]
        NFE_sim = Data_MPC[:NFE_sim]
        NFE_MPC = Data_MPC[:NFE_MPC]
        uk      = Data_MPC[:uk]
        x_sp    = Data_MPC[:x_sp]
        xk      = Data_MPC[:xk]
        U_Plot = [];
        X1_Plot = [];
        X2_Plot = [];

        # push the starting point into the final vectors
        push!(U_Plot, uk[1])
        push!(X1_Plot, xk[1])
        push!(X2_Plot, xk[2])
        global dt_sim, uk, xk


    println("Start Simulation!")
    for k in 1:NFE_sim
        global xk, uk, dt_sim
        x_sp_MPC = x_sp[k:k - 1 + NFE_MPC]
        uk = Solve_MPC(xk, uk, x_sp_MPC,NFE_MPC, NCP, Data_Plant)
        xk = Plant(xk, uk, dt_sim, Data_Plant);
        push!(U_Plot, uk[1])
        push!(X1_Plot, xk[1])
        push!(X2_Plot, xk[2])
    end
    println("The objective function is satisfied successfully!")
    myDict = Dict(
        :X1_Plot   => X1_Plot,
        :X2_Plot     => X2_Plot,
        :U_Plot     => U_Plot
    )
    return myDict
end

## Plotting with the option to show/not show the figure. Also different backends can be
  ↪  choose!
function Plot_Plant(MPC_Results, Data_MPC)
    T0_sim  = Data_MPC[:T0_sim];
    dt_sim  = Data_MPC[:dt_sim];
    Tf_sim  = Data_MPC[:Tf_sim];
    NFE_sim = Data_MPC[:NFE_sim];
    x_sp     = Data_MPC[:x_sp];
    X1_Plot = MPC_Results[:X1_Plot];
    X2_Plot = MPC_Results[:X2_Plot];
    U_Plot  = MPC_Results[:U_Plot];
    t_plot = collect(T0_sim:dt_sim:Tf_sim)
    p11 = plot( t_plot,[X1_Plot[nIter]  for nIter in 1:NFE_sim+1],label="x1")
    p11 = plot!(t_plot,[X2_Plot[nIter]  for nIter in 1:NFE_sim+1],label="x2")
    p11 = plot!(t_plot, x_sp[1:NFE_sim + 1],                      label="y_sp",
    ↪  linetype=:steppost, linestyle=:dash)
    p12 = plot(t_plot[1:end], [U_Plot[nIter]  for nIter in 1:NFE_sim+1],    label="u",
    ↪  linetype=:steppost)
    fig1 = plot(p11, p12, layout=(2, 1));
    plotlyjs()
    #gr()
    println("Start Plotting!")
    display(fig1)
    println("Finish Plotting!")
    return fig1
end
```

```
##Checking the Option for saving
function Save_Plant(fig1, Address)
    savefig(fig1, Address)
end
## Exporting Data


function Export_Plant(MPC_Results, Data_MPC, Address)
    X1_Plot = MPC_Results[:X1_Plot];
    X2_Plot = MPC_Results[:X2_Plot];
    U_Plot  = MPC_Results[:U_Plot];
    x_sp     = Data_MPC[:x_sp];
    NFE_sim = Data_MPC[:NFE_sim];
    T0_sim  = Data_MPC[:T0_sim];
    dt_sim  = Data_MPC[:dt_sim];
    Tf_sim  = Data_MPC[:Tf_sim];
    t_plot  = collect(T0_sim:dt_sim:Tf_sim);
    println("Start Exporting Data!")
    #using CSV
    #using DataFrames
    df = DataFrame(Name = ["x1","x2","U","SetPoint","Time"],
                   Value = [X1_Plot, X2_Plot, U_Plot, x_sp[1:NFE_sim+1], t_plot]
                   )
    CSV.write(Address, df)
    println("Finish Exporting Data!")
end
```

C NMPC.jl

```
using Ipopt, JuMP

include("CollMat.jl")


function Solve_MPC(x0, u0, x_sp, NFE, NCP, DataMPC)

m1 = Model(Ipopt.Optimizer)
Nx  = size(x0, 1);
Nu  = size(u0, 1);
dx0 = 0*copy(x0)
q0  = 0;
dq0 = 0;

model_par = DataMPC
    sf  = model_par[:x2_f]
    km    = model_par[:km]
    k1    = model_par[:k1]
    Y     = model_par[:Y]
    μmax = model_par[:μmax]

M = Collocation_Matrix(NCP)

##Defining the variables for the whole NFE_MPC horizon
@variable(m1, x[1:Nx, 1:NFE, 1:NCP+1]);
@variable(m1, dx[1:Nx, 1:NFE, 1:NCP]);
@variable(m1,  q[   1,       1:NFE, 1:NCP])
@variable(m1, dq[   1,       1:NFE, 1:NCP])
@variable(m1, u[1:Nu, 1:NFE])
##Setting up the bounds for variables (in case of need)
```

```
for nx in 1:Nx, nu in 1:Nu, nfe in 1:NFE, ncp in 1:NCP
    set_lower_bound(x[nx, nfe, ncp], 0)
    set_upper_bound(x[1 , nfe, ncp], 4.5)
    #set_lower_bound(dx[nx, nfe, ncp], 0)
    #set_upper_bound(dx[nx, nfe, ncp], 999)
    set_lower_bound(u[nu, nfe], 0)
    set_upper_bound(u[nu, nfe], 1)
end
##Setting up the starting points for variables
for nx in 1:Nx, nu in 1:Nu, nfe in 1:NFE, ncp in 1:NCP
    set_start_value(x[nx, nfe, ncp],    x0[nx])
    set_start_value(dx[nx, nfe, ncp],dx0[nx])
    set_start_value(u[nu, nfe],         u0[nu])
    set_start_value(q[1, nfe, ncp],     q0)
    set_start_value(dq[1, nfe, ncp],    dq0)
end
##Rename some of the variables to write the equations easier
@NLexpressions(m1, begin
    x1[nfe in 1:NFE, ncp in 1:NCP],           x[1, nfe, ncp]
    x2[nfe in 1:NFE, ncp in 1:NCP],           x[2, nfe, ncp]
    D[nfe in 1:NFE],                          u[1, nfe]
end)
##Constraints defining region!


#Here is where you should defining the model ODEs in each line
@NLconstraints(m1, begin
Constr_ODE1[nfe in 1:NFE, ncp in 1:NCP], dx[1, nfe, ncp]      == (((μmax * x2[nfe, ncp]) /
  ↪  (km + x2[nfe, ncp] + k1 * x2[nfe, ncp]^2)) - D[nfe]) * x1[nfe, ncp];
Constr_ODE2[nfe in 1:NFE, ncp in 1:NCP], dx[2, nfe, ncp]      == (sf - x2[nfe, ncp]) *
  ↪  D[nfe] - (((μmax * x2[nfe, ncp]) / (km + x2[nfe, ncp] + k1 * x2[nfe, ncp]^2))/Y) *
  ↪  x1[nfe, ncp];
#For more ODEs:
# Constr_ODEi[nfe in 1:NFE, ncp in 1:NCP], dx[i, nfe, ncp] = Equation!
end)


#Defining the Quadrature Equations
@NLconstraints(m1, begin
      Constr_dq0[nfe = 1    , ncp in 1:NCP], dq[1, nfe, ncp]  == (x1[nfe,ncp] -
      ↪  x_sp[nfe])^2 + 0.5*(D[nfe]  - u0[1]    )^2
      Constr_dq[nfe in 2:NFE, ncp in 1:NCP], dq[1, nfe, ncp]  == (x1[nfe,ncp] -
      ↪  x_sp[nfe])^2 + 0.5*(D[nfe] - D[nfe-1])^2
end)


#Constraints on input usage (To avoid big jumps on input usage)
@NLconstraints(m1, begin
      #Defining Inequality Constraints in each line
      Constr_Ineq1[nfe in 1:1   ],   -0.08 <=   D[nfe] - u0[1]       <=  0.08
      Constr_Ineq2[nfe in 2:NFE-3],   -0.08 <=   D[nfe] - D[nfe-1]    <=  0.08
      Constr_Ineq3[nfe in NFE-2:NFE ],     D[nfe] - D[nfe-1]    == 0.0
end)


#Collocation Equations for states and quadratures as Constraints
@NLconstraints(m1, begin
        Coll_Eq_Diff[nx in 1:Nx,  nfe = 1:NFE, ncp in 1:NCP],     x[nx, nfe, ncp+1] ==
        ↪  x[nx, nfe, 1] + sum(M[ncp, i] * dx[nx, nfe, i] for i in 1:NCP)
        Cont_Eq_First[nx in 1:Nx],                                x[nx, 1, 1] == x0[nx]
        Cont_Eq_rest[nx in 1:Nx,  nfe = 2:NFE],                   x[nx, nfe, 1] == x[nx,
        ↪  nfe-1, end]
```

```
        Coll_Eq_Quad0[             nfe = 1, ncp in 1:NCP],        q[1, nfe, ncp] == q0  +
        ↪  sum(M[ncp, i] * dq[1, nfe, i] for i in 1:NCP)
        Coll_Eq_Quad[       nfe in 2:NFE, ncp in 1:NCP],          q[1, nfe, ncp] == q[1,
        ↪  nfe-1, NCP]  + sum(M[ncp, i] * dq[1, nfe, i] for i in 1:NCP)
        end)
##And finally Defining the Objective Function!
@NLobjective(m1, Min,  sum( (x1[nfe,NCP] - x_sp[nfe])^2   for nfe in 1:NFE )  + 0.05*(D[1]
↪  - u0[1])^2 + sum( 0.05*(D[nfe] - D[nfe-1])^2 for nfe in 2:NFE)  )


## Calling the optimizer to optimize the whole prediction horizon
optimize!(m1)
JuMP.termination_status(m1)
JuMP.solve_time(m1::Model)
## Extarcting the results and reshape them into julia form

star_x = JuMP.value.(x[:, :, NCP])
star_x = cat(x0, star_x, dims = 2)
star_u = JuMP.value.(u)
star_x1 = JuMP.value.(x1[:, NCP])
star_x2 = JuMP.value.(x2[:, NCP])
star_x1 = cat(x0[1], star_x1, dims = 1)
star_x2 = cat(x0[2], star_x2, dims = 1)
star_D = JuMP.value.(D)
star_MPC = star_u[:,1]
##The output of the Function!
#Export the first input among NFE_MPC inputs as the output to be fed into the plant
return star_MPC

end
```

D Plant.jl

```
include("Parameters.jl")
using DifferentialEquations



function Plant(xkold,ukold,dt, DataPlant)
    function Sgen(dx,x,p,t)
                model_par = DataPlant;
                        x2_f  = model_par[:x2_f]
                        km    = model_par[:km]
                        k1    = model_par[:k1]
                        Y     = model_par[:Y]
                        μmax  = model_par[:μmax]
        μ = (μmax * x[2]) / (km + x[2] + k1 * x[2]^2);
        D = x[3];
        dx[1] = (μ - D) * x[1];
        dx[2] = (x2_f - x[2]) * D - (μ/Y) * x[1];
    end
    x0 = [xkold[1],xkold[2]]
    D0 = [ukold[1]]
    U0 = vcat(x0,D0)
    tspan = (0.0, dt)
    prob = ODEProblem(Sgen,U0,tspan)
    sol = DifferentialEquations.solve(prob, Tsit5())
    xf = sol.u[end][1:2];
```

```
return xf
end
```

E Parameters.jl

```
#This is the place tu put all the variables needed for the project
#All the variable will be sent from here to required places in the programme

##Parameters which are needed for setting up the plant and Differential Equations
function Model_Par(m1,m2,m3,m4,m5)
    myDict = Dict(
        :x2_f   => m1,
        :km     => m2,
        :k1     => m3,
        :Y      => m4,
        :μmax   => m5
    )
    return myDict
end

## Parameters which are needed to set up the simulation of the plant.

function Simulation_Data(Tfsim, TfMPC, Ncp, states, inputs)
    T0_sim  = 0.0;
    Tf_sim  = Tfsim;
    dt_sim  = 1;
    Tf_MPC  = TfMPC;
    dt_MPC  = 1;
    NCP     = Ncp;
    xk      = states;
    uk      = inputs;

    NFE_sim = convert(Int, (Tf_sim - T0_sim) / dt_sim);
    NFE_MPC = convert(Int, (Tf_MPC) / dt_MPC);
    x_sp     = hcat(1.5032 * ones(1, convert(Int, NFE_sim / 3)),     0.9951 *
    ↪  ones(1,convert(Int, NFE_sim / 3)),    0 * ones(1,convert(Int, NFE_sim / 3)), 0 *
    ↪  ones(1,convert(Int, NFE_MPC)));
    myDict = Dict(
        :T0_sim  => T0_sim,
        :Tf_sim  => Tf_sim,
        :dt_sim  => dt_sim,
        :Tf_MPC  => Tf_MPC,
        :dt_MPC  => dt_MPC,
        :NCP     => NCP,
        :NFE_sim => NFE_sim,
        :NFE_MPC => NFE_MPC,
        :x_sp     => x_sp,
        :xk      => xk,
        :uk      => uk
    )
    return myDict
end
```

F CollMat.jl

```
function Collocation_Matrix(N)

      #Radau
      if  N == 2

      t1 = 0.3333333
      t2 = 1.0

      M1 = [
            t1 1 / 2 * t1^2
            t2 1 / 2 * t2^2
            ]
      M2 = [
            1 t1
            1 t2
            ]

      M = M1 * inv(M2)

      elseif  N == 3

            t1 = 0.155051
            t2 = 0.644949
            t3 = 1.0

            M1 = [
                  t1 1 / 2 * t1^2 1 / 3 * t1^3
                  t2 1 / 2 * t2^2 1 / 3 * t2^3
                  t3 1 / 2 * t3^2 1 / 3 * t3^3
                  ]
            M2 = [
                  1 t1 t1^2
                  1 t2 t2^2
                  1 t3 t3^2
                  ]

            M = M1 * inv(M2)

      elseif N == 4

            t1 = 0.088588;
            t2 = 0.409467;
            t3 = 0.787659;
            t4 = 1;


            M1 = [
                  t1 1 / 2 * t1^2 1 / 3 * t1^3 1 / 4 * t1^4
                  t2 1 / 2 * t2^2 1 / 3 * t2^3 1 / 4 * t2^4
                  t3 1 / 2 * t3^2 1 / 3 * t3^3 1 / 4 * t3^4
                  t4 1 / 2 * t4^2 1 / 3 * t4^3 1 / 4 * t4^4
                  ]
            M2 = [
                  1 t1 t1^2 t1^3
                  1 t2 t2^2 t2^3
                  1 t3 t3^2 t3^3
                  1 t4 t4^2 t4^3
                  ]

            M = M1 * inv(M2)
```

```
61
62      elseif N == 5
63
64            t1 = 0.057104;
65            t2 = 0.276843;
66            t3 = 0.583590;
67            t4 = 0.860240;
68            t5 = 1;
69
70            M1 = [
71                  t1 1 / 2 * t1^2 1 / 3 * t1^3 1 / 4 * t1^4  1 / 5 * t1^5
72                  t2 1 / 2 * t2^2 1 / 3 * t2^3 1 / 4 * t2^4  1 / 5 * t2^5
73                  t3 1 / 2 * t3^2 1 / 3 * t3^3 1 / 4 * t3^4  1 / 5 * t3^5
74                  t4 1 / 2 * t4^2 1 / 3 * t4^3 1 / 4 * t4^4  1 / 5 * t4^5
75                  t4 1 / 2 * t4^2 1 / 3 * t4^3 1 / 4 * t4^4  1 / 5 * t5^5
76                  ]
77            M2 = [
78                  1 t1 t1^2 t1^3 t1^4
79                  1 t2 t2^2 t2^3 t2^4
80                  1 t3 t3^2 t3^3 t3^4
81                  1 t4 t4^2 t4^3 t4^4
82                  1 t4 t4^2 t4^3 t5^4
83                  ]
84
85            M = M1 * inv(M2)
86
87      end
88
89
90      return M
91 end
```

G runtests.jl

```
1 using PkgMPC
2 using Test
3
4
5
6
7 F(3,5)
8
9 ##For Testing
10 Data_MPC   = Simulation_Data(60.0, 8.0, 3, [1.0; 1.0], 0.35)
11 Data_Plant = Model_Par(4.0, 0.12, 0.4545, 0.4, 0.4)
12 Results = Simulate_Plant(Data_MPC, Data_Plant)
13 Figure = Plot_Plant(Results, Data_MPC)
14 Save_Plant(Figure,"//home//amir//Documents//Package_Results//MPC_Results.svg")
15 Export_Plant(Results,Data_MPC,
   ↪  "//home//amir//Documents//Package_Results//MPC_Results.csv")
16 ##for timing
17 Data_MPC   = @elapsed Simulation_Data(60.0, 8.0, 3, [1.0; 1.0], 0.35)
18 Data_Plant = @elapsed Model_Par(4.0, 0.12, 0.4545, 0.4, 0.4)
19 result = @elapsed Simulate_Plant(Data_MPC, Data_Plant)
20 Figure = @elapsed Plot_Plant(Results, Data_MPC)
21 time_save = @elapsed
   ↪  Save_Plant(Figure,"//home//amir//Documents//Package_Results//MPC_Results.svg")
22 time_export = @elapsed Export_Plant(Results,Data_MPC,
   ↪  "//home//amir//Documents//Package_Results//MPC_Results.csv")
```

```
23
24
25
26 @testset "PkgMPC.jl" begin
27     # Write your tests here.
28     @test F(2,3) == 13
29     @test F(3,0) == 6
30 end
```