NTNU
Norwegian University of
Science and Technology

Masters Specialization Project TKP4580

Sensitivity-Based Economic NMPC with a Path-Following Approach in Python

Brittany Hall

Norwegian University of Science and Technology
Department of Chemical Engineering
Process-Systems Engineering Group

Supervised by
Johannes Jäschke and Eka Suwartadi

December 19, 2017

Contents

List of Figures

List of Tables

List of Abbreviations

ADMM Alternative Direction Method of Multipliers

asNMPC Advanced step nonlinear model predictive control

CSTR Continuous stirred tank reactor

DAEs Differential algebraic equations

eMPC Economic model predictive control

iNMPC Ideal nonlinear model predictive control

KKT Karush-Kuhn-Tucker

LICQ Linear independence constraint qualification

MPC Model predictive control

NLP Nonlinear programming

NMPC Nonlinear model predictive control

OSI Open source initiative

pfNMPC Path following model predictive control

QP Quadratic programming

SC Strict complimentary

SSOSC Second-order sufficient condition

List of Symbols

Sign	Description	Unit
$\mathbf{A}$	Equality Constraint Matrix	
$\mathbf{A}$	Matrix	
A	Chemical component	
a_{ij}	Runge Kutta coefficient	
$\mathbf{a_{ij}}$	Matrix elements for an $i \times j$ matrix	
α_1	Path-following weight used to shorten step	
α	Relative volatility	
B	Bottoms flow rate	kmol/min
B	Chemical component	
$\mathbf{b}$	Constraint vector	
b_i	Runge Kutta coefficient	
$\boldsymbol{\chi}$	Decision variables (state variables + control input)	
$\Delta\boldsymbol{\chi}$	Change in $\boldsymbol{\chi}$	
$\boldsymbol{\chi}_f$	Terminal region	
$\boldsymbol{\chi}^*$	Optimal $\boldsymbol{\chi}$	
D	Distillate/Recycle flow rate	kmol/min
$\mathbf{d}$	Second-order sufficient condition variable	
$\frac{dM_i}{dt}$	Derivative of liquid molar holdup on stage i with respect to time	kmol/min
$\frac{d(M_i x_i)}{dt}$	Derivative of material on stage i with respect to time	kmol/min
$\frac{\partial}{\partial t}$	Partial derivative with respect to t	
$\frac{dx_i}{dt}$	Derivative of component on stage i with respect to time	min^{-1}
$\frac{dz}{dt}$	Derivative of z with respect to t	
F	Feed flow rate to distillation column	kmol/min
F_0	Feed flow rate to CSTR	kmol/min
$\mathbf{G}$	Inequality Constraint Matrix	
$\mathbf{H}$	$n_x \times n_x$-dimensional real symmetric matrix	
$\mathbf{h}$	Constraint vector	
J	Objective function	
J_m	Objective function for regularized stage	
K	Active constraint set	
$\mathscr{K}$	Order of polynomial	

Sign	Description	Unit
k	Current sample	
K_0	Weakly active constraint set	
$k+1$	Next sample	
$\boldsymbol{\kappa}$	Implicit feedback law	
K_+	Strongly active constraint set	
$\mathbf{lb}$	Constraint lower bounds	
$\boldsymbol{\lambda}$	Vector of Lagrange multipliers (equality constraint)	
$\Delta\boldsymbol{\lambda}$	Change in $\boldsymbol{\lambda}$	
λ_i	Eigenvalues of matrix $\mathbf{A}$	
$\boldsymbol{\lambda}^*$	Optimal $\boldsymbol{\lambda}$	
L_i	Liquid flow rate on stage i	kmol/min
L_{i+1}	Liquid flow rate on stage $i+1$	kmol/min
L_i^*	Nominal liquid flow rate on stage i	kmol/min
L_T	Reflux flow rate	kmol/min
$\mathbf{M}$	Collocation matrix	
M_B	Molar holdup on bottom stage	kmol
M_i	Molar holdup on stage i	kmol
M_i^*	Nominal molar holdup on stage i	kmol
$\boldsymbol{\mu}$	Vector of Lagrange multipliers (inequality constraint)	
$\Delta\boldsymbol{\mu}$	Change in $\boldsymbol{\mu}$	
μ_i	Lagrange multiplier for constraint i	
μ_i^*	Optimal μ of constraint i	
μ_j	Lagrange multiplier for constraint j	
$\boldsymbol{\mu}^*$	Optimal $\boldsymbol{\mu}$	
$\mathbb{N}$	Number of MPC/NMPC iterations	
N	Number of steps in path-following algorithm	
n_χ	Number of decision variables	
n_c	Number of equality constraints	
n_g	Number of inequality constraints	
n_p	Number of parameter variables	
n_u	Number of control inputs	
n_x	Number of states	
NF	Feed stage	
$\mathbf{n_k}$	Noise at samples k	
NT	Total condenser stage	
$*$	Optimal value	
$\not{p}$	Variables independent of t	
p	Parameter	
$\mathbf{p}$	Parameter vector	
$\mathbf{p_0}$	Initial parameter vector	

Sign	Description	Unit
p_1	Element 1 of $\mathbf{p}$	
p_2	Element 2 of $\mathbf{p}$	
p_B	Product price	\$/kg
$\bar{\mathbf{p}}$	Updated parameter vector	
p_D	Distillate price	\$/kg
$\Delta\mathbf{p}$	Change in parameter vector	
p_F	Feed cost	\$/kg
$\mathbf{p_f}$	Final parameter vector	
$\boldsymbol{\Psi}$	Terminal cost	
$\boldsymbol{\psi}$	Stage cost	
p_V	Steam cost	\$/kg
$\mathbf{Q}$	Gershgorin weight	
$\mathbf{q}$	Real valued, n_x-dimensional vector	
$\mathbf{Q_1}$	Gershgorin weight on states	
$\mathbf{Q_2}$	Gershgorin weight on inputs	
q_F	Liquid fraction of feed	
R	Recycle stream	
$\mathbb{R}$	Real numbers	
t	Time	min
τ_L	Time constant for liquid dynamics	min
Δt	Step size	min
$\boldsymbol{\theta}$	Collocation parameters	
t_k	Time at sample k	min
t_{k+1}	Time at sample $k+1$	min
$\mathbf{ub}$	Constraint upper bounds	
$\mathbf{u}$	Control input	
$\mathbf{u_k}$	Control input at sample k	
$\mathbf{u_{k+1}}$	Control input at sample $k+1$	
$\mathbf{u_{ss}}$	Steady state optimal input	
V	Vapor flow rate	kmol/min
$\mathbf{v}$	Predicted control input	
V_0	Nominal vapor flow rate	kmol/min
V_B	Bottom vapor flow rate	kmol/min
V_i	Vapor flow rate on stage i	kmol/min
V_{i+1}	Vapor flow rate on stage $i+1$	kmol/min
V_{i-1}	Vapor flow rate on stage $i-1$	kmol/min
V_T	Boilup vapor flow rate	kmol/min
w	Collocation NLP decision variables	

Sign	Description	Unit
$\mathbf{x_0}$	Initial solution of $\mathbf{x}$	
x_1	Element 1 of $\mathbf{x}$	
x_2	Element 2 of $\mathbf{x}$	
$\mathbf{x}$	State variable	
x_B	Bottoms liquid composition	
x_i	Liquid composition on stage i	
x_{i+1}	Liquid composition on stage $i+1$	
$\mathbf{x_k}$	State variable at sample k	
$\mathbf{x_{k+1}}$	State variable at sample $k+1$	
$\mathbf{y_0}$	Initial solution of $\mathbf{y}$	
y_D	Distillate vapor composition	
y_i	Vapor composition on stage i	
y_{i-1}	Vapor composition on stage $i-1$	
$\mathscr{Z}$	Path constraints	
$\mathbb{Z}$	Set of all integers	
$\mathbf{z}$	Predicted state variable	
z_F	Feed composition	

List of Functions

Sign	Description	Unit
c	Equality constraint function	
$c(\boldsymbol{\chi}, \mathbf{p})$	Equality constraints function	
$c_i(\boldsymbol{\chi}, \mathbf{p})$	Equality constraint i function	
$c_i(\boldsymbol{\chi}^*, \mathbf{p_0})$	Equality constraint i function evaluated at optimal point and initial parameter	
$c_i(\boldsymbol{\chi}^*, \mathbf{p_0} + \boldsymbol{\Delta p})$	Equality constraint i function evaluated at optimal point and parameter value	
$c(\boldsymbol{\chi}^*, \mathbf{p_0})$	Equality constraint functions evaluated at optimal point and initial parameter	
$\dot{\mathbf{x}}(\boldsymbol{\theta}_k, \mathbf{t})$	Derivative of state variable function	
F	Scalar objective function	
f	Continuous model function	
$F\left(x, \frac{dx}{dt}, u(t), \not{p}, t\right)$	Generic differential algebraic function	
$f(z(t), y(t), u(t), \not{p})$	Semi-explicit differential algebraic equation function	
$f(z(t), t)$	Generic system function	
g	Inequality constraint function	
$g_A(\boldsymbol{\chi}^*, \mathbf{p_0})$	Active inequality constraint function evaluated at optimal point and initial parameter	
$g(\boldsymbol{\chi}, \mathbf{p})$	Inequality constraints function	
$g_i(\boldsymbol{\chi}, \mathbf{p})$	Inequality constraint i function	
$g_i(\boldsymbol{\chi}^*, \mathbf{p_0})$	Inquality constraint i function evaluated at optimal point and initial parameter	
$g_j(\boldsymbol{\chi}^*, \mathbf{p_0})$	Inequality constraint j function evaluated at optimal point and initial parameter	
$g_j(\boldsymbol{\chi}^*, \mathbf{p_0} + \boldsymbol{\Delta p})$	Inequality constraint j function evaluated at optimal point and parameter value	
$\nabla_{\boldsymbol{\chi}}$	Gradient function (w.r.t $\boldsymbol{\chi}$)	
$\nabla_{\mathbf{p}}$	Gradient function (w.r.t $\mathbf{p}$)	
$g(w)$	Collocation NLP constraints	
$g(z(t), y(t), u(t), \not{p})$	Semi-explicit differential algebraic equation function	
$h(x(0))$	Initial value	
$\nabla^2_{\boldsymbol{\chi}\boldsymbol{\chi}}$	Hessian function (w.r.t $\boldsymbol{\chi}$)	
$\nabla^2_{\mathbf{p}\boldsymbol{\chi}}$	Hessian function (w.r.t $\mathbf{p}$ and $\boldsymbol{\chi}$)	
$\nabla^2_{\mathbf{pp}}$	Hessian function (w.r.t $\mathbf{p}$)	

Sign	**Description**	**Unit**
$\mathcal{L}(\boldsymbol{\chi}, \mathbf{p}, \boldsymbol{\lambda}, \boldsymbol{\mu})$	Lagrangian function	
$\mathcal{L}(\boldsymbol{\chi}^*, \mathbf{p_0} + \boldsymbol{\Delta}\mathbf{p}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$	Lagrangian function evaluated at optimal points and parameter value	
$\mathcal{L}(\boldsymbol{\chi}^*, \mathbf{p_0}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$	Lagrangian function evaluated at optimal point and initial parameter	
$P_{k,i}(t)$	Lagrange polynomial	
$P_{k,i}(t_{k,l})$	Lagrange polynomial function property	
$\dot{P}_{k,i}(t)$	Derivative of Lagrange polynomial	
$\Phi(w)$	Collocation NLP objective function	
$\sigma(\mathbf{p})$	Locally unique minimum of general parameteric NLP problem	
$u(t)$	Control variable function	
$x(t)$	State variable function	
$\mathbf{x}(\boldsymbol{\theta_k}, \mathbf{t})$	State variable function	
$\mathbf{x}(\boldsymbol{\theta_k}, \mathbf{t_k})$	State variable function	
$\mathbf{x}(\boldsymbol{\theta_k}, \mathbf{t_{k,j}})$	State variable function	
$y(t)$	Differential variable function	
$z(t)$	Algebraic variable function	
$z(0) = z_0$	Initial value	

Summary

In this project, a sensitivity-based predictor-corrector path-following method for advanced-step nonlinear model predictive control (asNMPC) is presented. NMPC is an advanced control strategy where an optimization problem is solved for a defined horizon and the solution is the optimized manipulated variable. Solving the full nonlinear programming (NLP) problem at every time step can be computationally expensive; this can cause delays that can lead to increasingly worse performance and even result in instability in the process. One approach to reduce the computational delay is to use sensitivity-based methods to solve the NLP; these exploit the fact that NMPC optimization problems are identical at each sample time except for the initial state. One such method is advanced-step NMPC (asNMPC); the full NLP is solved at every sample time but it is done in advance for a predicted initial state. When a new state measurement is available from the actual process, the NLP solution is updated by solving the sensitivity equation such that the solution matches the measured state. This correction technique is known as an improved path-following method.

This project focused on implementing both NMPC and path-following asNMPC on a system comprised of a CSTR and distillation column in Python; [26] has previously implemented this same system successfully in MATLAB. The NMPC was treated as an ideal system that could be solved instantly and was intended to be used as a comparison point for the asNMPC solution. In [26], it is shown that the asNMPC path-following algorithm traces the exact solution. The iNMPC was successfully implemented in Python and was verified by comparison with the MATLAB results from [26]. Unfortunately, due to difficulties in finding an open-source QP solver that could solve a system of this size, the asNMPC algorithm has not been successfully implemented in Python during the time period of this project. However, it should be possible to find an open source QP solver that handle large problems. Several potential solvers were identified and are discussed in more detail in this report.

Chapter 1

Introduction

Model predictive control (MPC) and non-linear model predictive control (NMPC) are advanced control strategies that involve solving an optimization problem for a set horizon to determine the optimal value of the manipulated variables at each sampling interval. Historically, this control strategy was only widely used in the chemical industry for processes with large time constants (i.e., slow dynamics) since the computations required are large. However, due to modern computation capabilities and algorithm development, this type of control has expanded to a variety of system types (even fast dynamics) [26]. MPC has a growing interest in both research and industry due to its performance in a variety of processes, in addition to its ability to handle constraints and perform optimization all while considering economics and nonlinearities of the process. The current areas of interest are: development of algorithms for rapid optimization, development of better modeling strategies, and new alternatives/variations that lead to improved closed-loop performance or reduce the computation time of the optimization problem [26]. In this project, the focus is on the reduction of the computation time of the optimization problem.

Since maximizing the profitability of the plant/process is often the ultimate goal, another type of MPC, known as economic MPC (eMPC), was developed. This allows for the integration of the economic optimization and the control layer into a single dynamic optimization layer [26]. Economic MPC works by adjusting the inputs such that the economic cost of the operation is directly minimized; thus allowing for the optimization of the cost during operation of the plant. When an optimization-based controller such as MPC is used, the economic criterion can be included directly in the cost function of the controller [16]. It is common to use nonlinear process models for this style of optimization. Therefore, one drawback of economic MPC is the requirement of solving a large nonlinear optimization problem (NLP) with the NMPC problem at every sample time for larger plant models. This computation can take a significant amount of time, lead to increasingly worse performance and even instability of the process [26].

One idea to reduce the effect of computational delay in NMPC is to use sensitivity-based methods which exploit the fact that the NMPC optimization problems are identical at each sample time with the exception of one changing parameter: the initial state. Using sensitivity-based methods, the full nonlinear optimization problem is no longer solved, thus reducing the computational delay. Instead, the sensitivity of the NLP solution at the previously-computed iteration is used to obtain an approximate solution to the new NMPC problem [26]. One such method is the advanced-step NMPC (asNMPC) which still involves solving the full NLP at every sample time, but it is computed in advance for a predicted

initial state. When the new state measurement is available from the process, the NLP solution is corrected using a fast sensitivity update to make the solution match the measured state. To update the solution, a path-following method can be utilized. This is referred to as advanced step NMPC using path-following or pfNMPC for short.

The focus of this project was the implementation of both the NMPC and pfNMPC in Python on a continuous stirred tank reactor (CSTR) and distillation column system. The work done here supplements the work conducted by Suwartadi, Kungurtsev and Jäschke [26]; the code was developed in MATLAB and utilized CasADi [3] and TOMLAB optimization software [14] to create the model and solve the optimization problem. The aim of implementing this same code in Python is to make a more widely available version of the path-following advanced-step NMPC implementation that uses only open-source code (see Appendix A for a discussion). The ultimate goal is to make the pfNMPC algorithm into a Python module that is generic and can handle any model.

Chapter 2

Background

2.1 NMPC Problem Formulations

2.1.1 The NMPC Problem

Consider a nonlinear discrete-time dynamic system expressed as:

$$\mathbf{x_{k+1}} = f(\mathbf{x_k}, \mathbf{u_k}) + \mathbf{n_k} \tag{2.1}$$

where $\mathbf{x}_k \in \mathbb{R}^{n_x}$ denotes the state variable, $\mathbf{u}_k \in \mathbb{R}^{n_u}$ is the control input and $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x}$ is a continuous model function, which calculates the next state $\mathbf{x_{k+1}}$ from the previous state $\mathbf{x_k}$ and control input $\mathbf{u_k}$, where $k \in N$ [26]. This system can be optimized by a nonlinear model predictive controller that solves the problem

$$\begin{aligned}
(\mathscr{P}_{NMPC}): \min_{\mathbf{z}_l,\, \mathbf{v}_l} \quad & \boldsymbol{\Psi}(\mathbf{z}_N) + \sum_{l=0}^{N-1} \boldsymbol{\psi}(\mathbf{z}_l, \mathbf{v}_l) \\
\text{s.t.} \quad & \mathbf{z}_{l+1} = f(\mathbf{z}_l, \mathbf{v}_l), \qquad l = 0, \ldots, N-1, \\
& \mathbf{z}_0 = \mathbf{x}_k, \\
& (\mathbf{z}_l, \mathbf{v}_l) \in \mathscr{Z} \qquad l = 0, \ldots, N-1, \\
& \mathbf{z}_N \in \boldsymbol{\chi}_f
\end{aligned} \tag{2.2}$$

at each sample time. Here $\mathbf{z}_l \in \mathbb{R}^{n_x}$ is the predicted state variable; $\mathbf{v}_l \in \mathbb{R}^{n_u}$ is the predicted control input; and $\mathbf{z}_n \in \boldsymbol{\chi}_f$ is the final predicted state variable restricted to the terminal region $\boldsymbol{\chi}_f \in \mathbb{R}^{n_x}$. The stage cost is denoted by $\boldsymbol{\psi} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}$ and the terminal cost by $\boldsymbol{\Psi} : \boldsymbol{\chi}_f \to \mathbb{R}$. $\mathscr{Z}$ denotes the path constraints where $\mathscr{Z} = \{(\mathbf{z}, \mathbf{v}) \mid q(\mathbf{z}, \mathbf{v}) \leq 0\}$, where $q : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_q}$. The solution to this problem is denoted as $\{\mathbf{x}_0^*, \ldots, \mathbf{z}_N^*, \mathbf{v}_0^*, \ldots, \mathbf{v}_{N-1}^*\}$.

The idea is that at sample time k, an estimate or measurement of the state $\mathbf{x_k}$ is obtained and the problem $\mathscr{P}_{NMPC}$ is solved, The first part of the optimal control sequence becomes the plant input such that $\mathbf{u_k} = \mathbf{v}_0^*$. This part of the solution defines an implicit feedback law $\mathbf{u_k} = \boldsymbol{\kappa}(\mathbf{x_k})$, and the system evolves according to Equation 2.1. At the next sample time $k + 1$, when the measurement of the new state is obtained, the procedure is repeated. Algorithm 2.1 summarizes the generic NMPC algorithm.

Algorithm 2.1: General NMPC algorithm.

```
set k ← 0;
while MPC is running do
    Measure or estimate x_k.
    Assign the initial state: set z_0 = x_k.
    Solve the optimization problem P_NMPC to find v_0^*.
    Assign the plant input u_k = v_0^*.
    Inject u_k to the plant 2.1.
    Set k ← k + 1.
```

2.1.2 Ideal NMPC and Advanced-Step NMPC Framework

To achieve optimal economic performance and good stability properties, the problem shown in $\mathscr{P}_{NMPC}$ needs to be solved instantaneously, such that the optimal input can be injected into the process immediately. This is known as ideal NMPC. However, in reality, there will always be some time delay between obtaining the updated values of the states and injecting them into the plant. The main cause of the delay is the time required to solve the optimization problem $\mathscr{P}_{NMPC}$. As the process models grow, so too does the computation time. With sufficiently large systems, this computational delay cannot be neglected. One approach to decrease this delay is the advanced-step NMPC (asNMPC) which is based on the following steps:

1. Solve the NMPC problem at time k with a predicted state value of $k+1$
2. When the measurement $\mathbf{x_{k+1}}$ becomes available at time $k+1$, compute an approximation of the NLP solution using fast sensitivity methods
3. Update $k \leftarrow k+1$, and repeat from Step 1

There are different fast sensitivity methods that can be employed but this project focuses on the application of the sensitivity-based path-following algorithm.

2.2 Sensitivity-Based Path-Following NMPC

Sensitivity results from other works are outlined in the following sections. These results are utilized in a path-following scheme for obtaining fast approximate solutions to the NLP problem.

2.2.1 Sensitivity Properties of NLP

The dynamic optimization problem shown in Equation 2.2 can be written as a generic NLP problem:

$$\begin{aligned} (\mathscr{P}_{NLP}): \min_{\boldsymbol{\chi}} \quad & F(\boldsymbol{\chi}, \mathbf{p}) \\ \text{s.t.} \quad & c(\boldsymbol{\chi}, \mathbf{p}) = 0, \\ & g(\boldsymbol{\chi}, \mathbf{p}) \leq 0 \end{aligned} \tag{2.3}$$

where $\boldsymbol{\chi} \in \mathbb{R}^{n_\chi}$ are the decision variables (typically the state variables and the control input) and $\mathbf{p} \in \mathbb{R}^{n_p}$ is the parameter (typically the initial state variable). $F : \mathbb{R}^{n_\chi} \times \mathbb{R}^{n_p} \to \mathbb{R}$ is the scalar objective function, $c : \mathbb{R}^{n_\chi} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_c}$ denotes the equality constraints, and $g : \mathbb{R}^{n_\chi} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_g}$ denotes the inequality constraints. Each instance of the general parameteric NLP, shown in Equation 2.3, that is solved for each sample time differs only in the parameter $\mathbf{p}$.

The Lagrangian function of this problem is defined as

$$\mathcal{L}(\boldsymbol{\chi}, \mathbf{p}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = F(\boldsymbol{\chi}, \mathbf{p}) + \boldsymbol{\lambda}^T c(\boldsymbol{\chi}, \mathbf{p}) + \boldsymbol{\mu}^T g(\boldsymbol{\chi}, \mathbf{p}) \tag{2.4}$$

and the Karush-Kuhn-Tucker (KKT), first order optimality, conditions are written as [26]:

$$\begin{aligned} c(\boldsymbol{\chi}, \mathbf{p}) = 0, \quad g(\boldsymbol{\chi}, \mathbf{p}) \leq 0, \quad & \textit{(primal feasibility)} \\ \boldsymbol{\mu} \geq 0, \quad & \textit{(dual feasibility)} \\ \nabla_{\boldsymbol{\chi}} \mathcal{L}(\boldsymbol{\chi}, \mathbf{p}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = 0, \quad & \textit{(stationary condition)} \\ \boldsymbol{\mu}^T g(\boldsymbol{\chi}, \mathbf{p}) = 0, \quad & \textit{(complementary slackness)} \end{aligned} \tag{2.5}$$

For the KKT conditions to be a necessary condition of optimality, it is assumed that the linear independence constraint qualification (LICQ) holds. The LICQ states

Definition 2.1 *(LICQ) Given a vector* $\mathbf{p}$ *and a point* $\boldsymbol{\chi}$*, the LICQ holds at* $\boldsymbol{\chi}$ *if the set of vectors* $\left\{ \{\nabla_{\boldsymbol{\chi}} c_i(\boldsymbol{\chi}, \mathbf{p})\}_{i \in \{1,\dots,n_c\}} \cup \{\nabla_{\boldsymbol{\chi}} g_i(\boldsymbol{\chi}, \mathbf{p})_{i:\, g_i(\boldsymbol{\chi}, \mathbf{p})=0}\} \right\}$ *is linearly independent.*

This implies that the Lagrange multipliers $(\boldsymbol{\lambda}, \boldsymbol{\mu})$ satisfying the KKT conditions are unique. If a second-order condition also holds, then a unique local minimum is guaranteed. The second-order condition states that the Hessian matrix must be positive definite in a set of appropriate directions defined in the following property [26]:

Definition 2.2 *(SSOSC) The strong second-order sufficient condition (SSOSC) holds at* $\boldsymbol{\chi}$ *with multipliers* $\boldsymbol{\lambda}$ *and* $\boldsymbol{\mu}$ *if* $\mathbf{d}^T \nabla_{\boldsymbol{\chi}}{}^2 \mathcal{L}(\boldsymbol{\chi}, \mathbf{p}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \mathbf{d} > \mathbf{0}$ *for all* $\mathbf{d} \neq 0$*, such that* $\nabla_{\boldsymbol{\chi}} c(\boldsymbol{\chi}, \mathbf{p})^T \mathbf{d} = 0$ *and* $\nabla_{\boldsymbol{\chi}} g_i(\boldsymbol{\chi}, \mathbf{p})^T \mathbf{d} = \mathbf{0}$ *for* i*, such that* $g_i(\boldsymbol{\chi}, \mathbf{p}) = 0$ *and* $\mu_i > 0$*.*

Before sensitivity results can be discussed, one more definition must be presented.

Definition 2.3 *(SC) Given a vector* $\mathbf{p}$ *and a solution* $\boldsymbol{\chi}^*$ *with vectors of multipliers* $\boldsymbol{\lambda}^*$ *and* $\boldsymbol{\mu}^*$*, strict complimentary (SC) holds if* $\mu_i^* - g_i(\boldsymbol{\chi}^*, \mathbf{p_0}) > 0$ *for each* $i = 1, \dots, n_g$*.*

It has been shown in [9] that the following holds:

Theorem 2.1 *(Implicit function theorem applied to optimality conditions) Let* $\boldsymbol{\chi}^*(\mathbf{p})$ *be a KKT point that satisfies Equation 2.5, and assumed that LICQ, SSOSC, and SC all hold at* $\boldsymbol{\chi}^*$*. Further, let the function* F, c, g *be at least* $(k+1)$*-times differentiable in* $\boldsymbol{\chi}$ *and* k*-times differentiable in* $\mathbf{p}$*. Then:*

- $\boldsymbol{\chi}^*$ *is an isolated minimizer and the associated multipliers* $\boldsymbol{\lambda}$ *and* $\boldsymbol{\mu}$ *are unique*
- *for* $\mathbf{p}$ *in a neighborhood of* $\mathbf{p_0}$*, the set of active constraints remains unchanged*
- *for* $\mathbf{p}$ *in a neighborhood of* $\mathbf{p_0}$*, there exists a k-times differentiable function* $\sigma(\mathbf{p}) = \begin{bmatrix}\boldsymbol{\chi}^*(\mathbf{p})^T & \boldsymbol{\mu}^*(\mathbf{p})^T & \boldsymbol{\lambda}(\mathbf{p})^T\end{bmatrix}$*, that corresponds to a locally unique minimum for Equation 2.3*

Using these results, the sensitivity of the optimal solution $(\boldsymbol{\chi}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ in a small neighborhood of $\mathbf{p_0}$ can be found by solving the system of linear equations that arises from applying the implicit function theorem to the KKT conditions of Equation 2.3.

$$\begin{bmatrix} \nabla^2_{\boldsymbol{\chi}\boldsymbol{\chi}}\mathcal{L}(\boldsymbol{\chi}^*, \mathbf{p_0}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) & \nabla_{\boldsymbol{\chi}} c(\boldsymbol{\chi}^*, \mathbf{p_0}) & \nabla_{\boldsymbol{\chi}} g_A(\boldsymbol{\chi}^*, \mathbf{p_0}) \\ \nabla_{\boldsymbol{\chi}} c(\boldsymbol{\chi}^*, \mathbf{p_0})^T & 0 & 0 \\ \nabla_{\boldsymbol{\chi}} g_A(\boldsymbol{\chi}^*, \mathbf{p_0})^T & 0 & 0 \end{bmatrix} \begin{bmatrix} \nabla_{\mathbf{p}}\boldsymbol{\chi} \\ \nabla_{\mathbf{p}}\boldsymbol{\lambda} \\ \nabla_{\mathbf{p}}\boldsymbol{\mu} \end{bmatrix} = - \begin{bmatrix} \nabla^2_{\mathbf{p}\boldsymbol{\chi}}\mathcal{L}(\boldsymbol{\chi}^*, \mathbf{p_0}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) \\ \nabla_{\mathbf{p}} c(\boldsymbol{\chi}^*, \mathbf{p_0}) \\ \nabla_{\mathbf{p}} g_A(\boldsymbol{\chi}^*, \mathbf{p_0}) \end{bmatrix} \tag{2.6}$$

where $g_A(\boldsymbol{\chi}^*, \mathbf{p_0})$ indicates that only the vectors and components of the Jacobian corresponding to the active inequality constraints at $\boldsymbol{\chi}$ are included; in other words, where $i \in A$ if $g_i(\boldsymbol{\chi}, \mathbf{p}) = 0$.

The solution to the system of the linear equations is written as $\begin{bmatrix}\nabla_{\mathbf{p}}\boldsymbol{\chi} & \nabla_{\mathbf{p}}\boldsymbol{\lambda} & \nabla_{\mathbf{p}}\boldsymbol{\mu}\end{bmatrix}^T$. It is possible to obtain a good estimate of the solution to the NLP problem for small $\Delta\mathbf{p}$ at the parameter value $\mathbf{p_0} + \Delta\mathbf{p}$:

$$\boldsymbol{\chi}(\mathbf{p_0} + \Delta\mathbf{p}) = \boldsymbol{\chi}^* + \nabla_{\mathbf{p}}\boldsymbol{\chi}\Delta\mathbf{p} \tag{2.7}$$
$$\boldsymbol{\lambda}(\mathbf{p_0} + \Delta\mathbf{p}) = \boldsymbol{\lambda}^* + \nabla_{\mathbf{p}}\boldsymbol{\lambda}\Delta\mathbf{p} \tag{2.8}$$
$$\boldsymbol{\mu}(\mathbf{p_0} + \Delta\mathbf{p}) = \boldsymbol{\mu}^* + \nabla_{\mathbf{p}}\boldsymbol{\mu}\Delta\mathbf{p} \tag{2.9}$$

However, if $\Delta\mathbf{p}$ becomes large, the approximate solution may no longer be sufficiently accurate due to the fact that strict complementary requires that the active set cannot change; a large $\Delta\mathbf{p}$ can result in active set changes. The above condition thus only holds for small perturbations in $\Delta\mathbf{p}$.

Note that the sensitivity system of linear equations corresponds to the stationary conditions for a particular quadratic programming (QP) problem [26]. It can be proven that for $\Delta\mathbf{p}$ sufficiently small, the set $\{i \ : \ \boldsymbol{\mu}(\bar{\mathbf{p}})_i > 0\}$ is constant for $\bar{\mathbf{p}} = \mathbf{p_0} + \Delta\mathbf{p}$. A QP can then be formed where weakly-active constraints are moved off of and strongly-active ones are remained on. The primal-dual solution of this QP will then be the directional derivative of the primal-dual solution path $\boldsymbol{\chi}^*(\mathbf{p}), \boldsymbol{\lambda}^*(\mathbf{p}), \boldsymbol{\mu}^*(\mathbf{p})$.

It has been shown that the solution of the perturbed NLP can be found by solving a QP problem of the form [4]:

$$\begin{aligned} \min_{\Delta\boldsymbol{\chi}} \quad & \frac{1}{2}\Delta\boldsymbol{\chi}^T\nabla^2_{\boldsymbol{\chi}\boldsymbol{\chi}}\mathcal{L}(\boldsymbol{\chi}^*, \mathbf{p_0}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)\Delta\boldsymbol{\chi} + \Delta\boldsymbol{\chi}^T\nabla^2_{\mathbf{p}\boldsymbol{\chi}}\mathcal{L}(\boldsymbol{\chi}^*, \mathbf{p_0}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)\Delta\mathbf{p} \\ \text{s.t.} \quad & \nabla_{\boldsymbol{\chi}} c_i(\boldsymbol{\chi}^*, \mathbf{p_0})^T\Delta\boldsymbol{\chi} + \nabla_{\mathbf{p}} c_i(\boldsymbol{\chi}^*, \mathbf{p_0})^T\Delta\mathbf{p} = 0, \quad i = 1, \ldots, n_c, \\ & \nabla_{\boldsymbol{\chi}} g_j(\boldsymbol{\chi}^*, \mathbf{p_0})^T\Delta\boldsymbol{\chi} + \nabla_{\mathbf{p}} g_j(\boldsymbol{\chi}^*, \mathbf{p_0})^T\Delta\mathbf{p} = 0, \quad j \in K_+, \\ & \nabla_{\boldsymbol{\chi}} g_j(\boldsymbol{\chi}^*, \mathbf{p_0})^T\Delta\boldsymbol{\chi} + \nabla_{\mathbf{p}} g_j(\boldsymbol{\chi}^*, \mathbf{p_0})^T\Delta\mathbf{p} \leq 0, \quad j \in K_0 \end{aligned} \tag{2.10}$$

where $K_+ = \{j \in \mathbb{Z} : \mu_j > 0\}$ is the strongly-active set and $K_0 = \{j \in \mathbb{Z} : \mu_j = 0,\ g_j(\boldsymbol{\chi}^*, \mathbf{p_0}) = 0\}$ denotes the weakly active set. Note that the solution to this QP is the directional derivative of the primal-dual solution of the NLP; thus it is a predictor step and Equation (2.10) is referred to as a pure-predictor. Obtaining the sensitivity via Equation (2.10) instead of Equation (2.6) is advantageous in that changes in the active set are accounted for and strict complementarity is not required. In the case that SC does hold, then Equation (2.6) and Equation (2.10) are equivalent.

2.2.2 Path-Following Based on Sensitivity Properties

It is important to recognize that Equation (2.6) and the QP in Equation (2.10) are only able to produce the optimal solution accurately for small perturbations and cannot be guaranteed to work for larger perturbations. This is due to the curvature in the solution path and active set changes that may happen further away from the linearization point. One way of handling cases where this is true, is to divide the perturbation into several smaller intervals and to iteratively use the sensitivity to track the path of optimal solutions [26]; this is known as a path-following method.

The core idea of the path-following method is to reach the solution of the problem at a final parameter value $\mathbf{p_f}$ by tracing a sequence of solutions $(\boldsymbol{\chi}_k, \boldsymbol{\lambda}_k, \boldsymbol{\mu}_k)$ for a series of parameter values given by $\mathbf{p}(t_k) = (1 - t_k)\mathbf{p_0} + t_k\mathbf{p_f}$ where $0 = t_0 < t_1 < \ldots < t_k < \ldots < t_N = 1$. The new direction is found by evaluating the sensitivity at the current point. Note that this is similar to applying Euler integration for ordinary differential equations [26].

A path-following algorithm that is based only on the pure-predictor QP may fail to track the solution accurately enough and thus lead to poor solutions. To address this problem, elements are introduced that are similar to a Newton step, which will force the path-following algorithm towards the true solution. A corrector element can be introduced into a QP that results in a QP similar to the predictor QP (2.10). If Equation 2.3 is approximated by a QP, linearized with respect to both $\boldsymbol{\chi}$ and $\mathbf{p}$, and the equality of the strongly-active constraints is enforced, the NLP can be written as a QP of the form:

$$
\begin{aligned}
\min_{\Delta\boldsymbol{\chi},\,\Delta\mathbf{p}} \quad & \frac{1}{2}\Delta\boldsymbol{\chi}^T\nabla^2_{\boldsymbol{\chi}\boldsymbol{\chi}}\mathcal{L}(\boldsymbol{\chi}^*, \mathbf{p_0}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)^T\Delta\boldsymbol{\chi} + \Delta\boldsymbol{\chi}^T\nabla^2_{\mathbf{p}\boldsymbol{\chi}}\mathcal{L}(\boldsymbol{\chi}^*, \mathbf{p_0}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)\Delta\mathbf{p} \\
& + \nabla_{\mathbf{p}}F^T\Delta\boldsymbol{\chi} + \nabla_{\mathbf{p}}F\Delta\mathbf{p} + \frac{1}{2}\Delta\mathbf{p}^T\nabla^2_{\mathbf{pp}}\mathcal{L}(\boldsymbol{\chi}^*, \mathbf{p_0}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)\Delta\mathbf{p} \\
\text{s.t.} \quad & c_i(\boldsymbol{\chi}^*, \mathbf{p_0}) + \nabla_{\boldsymbol{\chi}}c_i(\boldsymbol{\chi}^*, \mathbf{p_0})^T\Delta\boldsymbol{\chi} + \nabla_{\mathbf{p}}c_i(\boldsymbol{\chi}^*, \mathbf{p_0})^T\Delta\mathbf{p} = 0, \quad i = 1, \ldots n_c, \\
& g_j(\boldsymbol{\chi}^*, \mathbf{p_0}) + \nabla_{\boldsymbol{\chi}}g_j(\boldsymbol{\chi}^*, \mathbf{p_0})^T\Delta\boldsymbol{\chi} + \nabla_{\mathbf{p}}g_j(\boldsymbol{\chi}^*, \mathbf{p_0})\Delta\mathbf{p} = 0, \quad j \in K_+, \\
& g_j(\boldsymbol{\chi}^*, \mathbf{p_0}) + \nabla_{\boldsymbol{\chi}}g_j(\boldsymbol{\chi}^*, \mathbf{p_0})^T\Delta\boldsymbol{\chi} + \nabla_{\mathbf{p}}g_j(\boldsymbol{\chi}^*, \mathbf{p_0})^T\Delta\mathbf{p} \leq 0, \quad j \in \{1, \ldots, n_g\} \setminus K_+
\end{aligned}
$$

For the NMPC problem $\mathscr{P}_{NMPC}$, the parameter $\mathbf{p}$ corresponds to the current "initial" state ($\mathbf{x_k}$). The cost function is independent of $\mathbf{p}$ which means that $\nabla_{\mathbf{p}}F(\boldsymbol{\chi}, \mathbf{p}) = 0$. In addition, the parameter is linear in the constraints meaning that $\nabla_{\mathbf{p}}c(\boldsymbol{\chi}, \mathbf{p})$ and $\nabla_{\mathbf{p}}g(\boldsymbol{\chi}, \mathbf{p})$ are constants. Applying these simplifications, the

above QP can be written as:

$$\begin{aligned}
\min_{\Delta\boldsymbol{\chi}} \quad & \frac{1}{2}\Delta\boldsymbol{\chi}^T\nabla^2_{\boldsymbol{\chi}\boldsymbol{\chi}}\mathcal{L}(\boldsymbol{\chi}^*, \mathbf{p_0}+\boldsymbol{\Delta p}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)\Delta\boldsymbol{\chi} + \nabla_{\boldsymbol{\chi}}F^T\Delta\boldsymbol{\chi} \\
\text{s.t.} \quad & c_i(\boldsymbol{\chi}^*, \mathbf{p_0}+\boldsymbol{\Delta p}) + \nabla_{\boldsymbol{\chi}}c_i(\boldsymbol{\chi}^*, \mathbf{p_0}+\boldsymbol{\Delta p})^T\Delta\boldsymbol{\chi} = 0 \qquad i = 0,\dots,n_c, \\
& g_j(\boldsymbol{\chi}^*, \mathbf{p_0}+\boldsymbol{\Delta p}) + \nabla_{\boldsymbol{\chi}}g_j(\boldsymbol{\chi}^*, \mathbf{p_0}+\boldsymbol{\Delta p})^T\Delta\boldsymbol{\chi} = 0 \qquad j \in K_+, \\
& g_j(\boldsymbol{\chi}^*, \mathbf{p_0}+\boldsymbol{\Delta p}) + \nabla_{\boldsymbol{\chi}}g_j(\boldsymbol{\chi}^*, \mathbf{p_0}+\boldsymbol{\Delta p})^T\Delta\boldsymbol{\chi} \leq 0 \qquad j \in \{1,\dots,n_g\} \setminus K_+
\end{aligned} \tag{2.11}$$

This formulation is known as the predictor-corrector form. This QP tries to estimate how the NLP solution changes as the parameter does in the predictor component and refines the estimate, as the corrector, so that the KKT conditions are more closely satisfied at the new parameter.

The predictor-corrector QP is well suited for use in a path-following algorithm. Recall the parameter equation: $\mathbf{p}(t_k) = (1-t_k)\mathbf{p_0} + t_k\mathbf{p_f}$. At each point $\mathbf{p}(t_k)$, the QP is solved and the primal-dual solutions are updated using:

$$\boldsymbol{\chi}(t_{k+1}) = \boldsymbol{\chi}(t_k) + \Delta\boldsymbol{\chi} \tag{2.12}$$
$$\boldsymbol{\lambda}(t_{k+1}) = \Delta\boldsymbol{\lambda} \tag{2.13}$$
$$\boldsymbol{\mu}(t_{k+1}) = \Delta\boldsymbol{\mu} \tag{2.14}$$

where $\Delta\boldsymbol{\chi}$ is obtained from the primal solution of the QP (2.11); $\Delta\boldsymbol{\lambda}$ and $\Delta\boldsymbol{\mu}$ correspond to the Lagrange multipliers of the QP.

This QP formulation is able to detect changes in the active set along the path. If a constraint becomes inactive, the corresponding multiplier μ_j will first become weakly active, meaning that it is added to the set K_0. If a new constraint becomes active, the corresponding linearized inequality constraint in the QP will be active and tracked at the next iteration.

The path-following algorithm is summarized with its main steps in Algorithm 2.2. This algorithm is used to find a fast approximation of the optimal NLP solution corresponding to the new available state measurement; this is done by following the optimal solution path from the predicted state to the measured state. The use of the path following algorithm should result in faster computation time in comparison to solving the full NMPC problem.

Algorithm 2.2: Path-following algorithm

Input: initial variables from NLP $\boldsymbol{\chi}^*(\mathbf{p_0})$,$\boldsymbol{\lambda}^*(\mathbf{p_0})$,$\boldsymbol{\mu}^*(\mathbf{p_0})$

Fix stepsize Δt, and set $N = \frac{1}{\Delta t}$;
Set initial parameter value $\mathbf{p_0}$,;
Set final parameter value $\mathbf{p_f}$,;
Set $t = 0$;
for $k \leftarrow 1$ **to** N **do**
    Compute step $\Delta \mathbf{p} = \mathbf{p}_k - \mathbf{p}_{k-1}$;
    Solve QP problem;
    if *QP is feasible* **then**
        $\boldsymbol{\chi} \leftarrow \boldsymbol{\chi} + \Delta\boldsymbol{\chi}$;
        Update dual variables appropriately using either the pure-predictor method or the predictor-corrector method;
        $t \leftarrow t + \Delta t$;
        $k \leftarrow k + 1$;
    else
        $\Delta t \leftarrow \alpha_1 \Delta t$;
        $t \leftarrow t - \alpha_1 \Delta t$;

2.2.3 Path-Following asNMPC Approach

The asNMPC approach solves the full NLP at every time step for a predicted state; when a new measurement is available, the precomputed NLP solution is updated by tracking the optimal solution curve from the predicted initial state to the new measured state. The update is done by solving a linearized version of the NLP, which becomes a QP problem, until a set criteria is met; either a predictor or a predictor-corrector method can be used to update the solution. This correction method is known as path-following. Note that the solution of the last QP along the path corresponds to the updated NLP solution and only the inputs from the last QP become inputs to the plant.

One unique quality of this method is that strong and weakly active inequality constraints are differentiated between. Strongly-active inequalities are linearized and included as equality constraints in the QP, but weakly active constraints are linearized and included as inequality constraints in the QP. This helps to ensure that the true solution path is tracked more accurately, particularly in the case that the full Hessian of the optimization problem is non-convex [26]. The pfNMPC method outlined in 2.2 is illustrated with an example below.

Example 2.1 *Consider the following parametric NLP [17]:*

$$\begin{aligned} \min_{\mathbf{x} \in \mathbb{R}^2} \quad & p_1 {x_1}^3 + {x_2}^2 \\ \text{s.t.} \quad & x_2 - e^{-x_1} \geq 0, \\ & x_1 \geq p_2 \end{aligned} \tag{2.15}$$

Start at the approximate solution to Equation 2.15 $(\mathbf{x_0}, \mathbf{y_0}) = ((0.5, 0.6), 1.2)$ *with* $\mathbf{p} = (1, -4)$ *and trace a path to generate an approximate solution for* $\mathbf{p} = (8, 1)$. *Note that the starting point* $\mathbf{p} = (1, -4)$ *is referred to as* $\mathbf{p_0}$ *and the final point* $\mathbf{p} = (8, 1)$ *as* $\mathbf{p_f}$.

Figure 2.1 shows the contour plots and constraints for the approximate solution at $\mathbf{p_0}$ *and at* $\mathbf{p_f}$ *respectively. The contours of the objective function are given in black, the constraints plotted in red, and the current point is a blue star. Note that as plotted the contour plot for* $\mathbf{p_0}$ *does not show the second constraint since* $x_2 = -4$ *is out of range for the axis.*

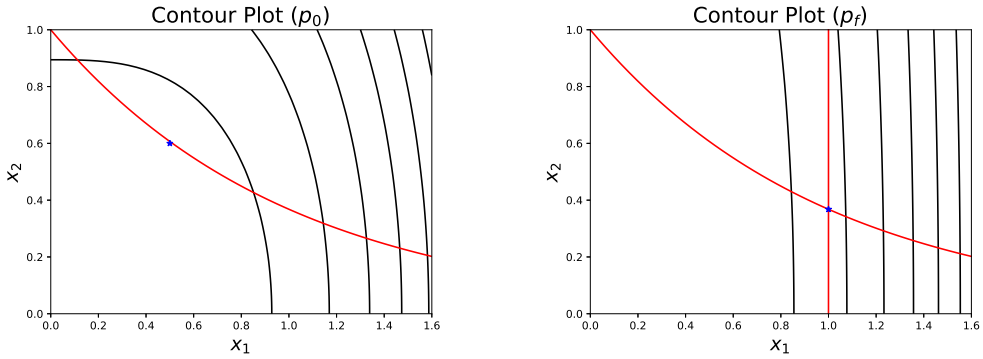

Figure 2.1: Plot of the problem at $t = 0$ and $t = 1$

This problem has two inequality constraints ($n_g = 2$) *and zero equality constraints* ($n_c = 0$). *Algorithm 2.2 is applied to this problem. The full NLP be solved to find the initial variables using the predicted solution:* $\boldsymbol{\chi}(\mathbf{p_0}), \boldsymbol{\lambda}^*(\mathbf{p_0}), \boldsymbol{\mu}^*(\mathbf{p_0})$. *The NLP solution is then fed to a QP solver where the linearized NLP is solved as a QP problem. Either the pure-predictor QP (2.10) or the predictor-corrector QP (2.11) formulation can be used; here the predictor-corrector formulation was utilized. If the QP is feasible, the primal variables* $\boldsymbol{\chi}$ *and the dual variables* ($\boldsymbol{\mu}$,$\boldsymbol{\lambda}$) *are updated either using the pure-predictor method or the predictor-corrector method depending on which QP formulation was solved. The update method should be selected based on the problem to be solved; stiff problems should not use predictor-corrector methods. Next the step size is updated using the path following equation given previously. If the QP is infeasible, then the step size is reduced and the QP is solved again.*

Figure 2.2 illustrates how x_1 *changes with respect to* t *when* $k = 100$ *iterations are used* ($\Delta t = 0.01$). *Note how* x_1 *changes steeply as the constraints become active.*

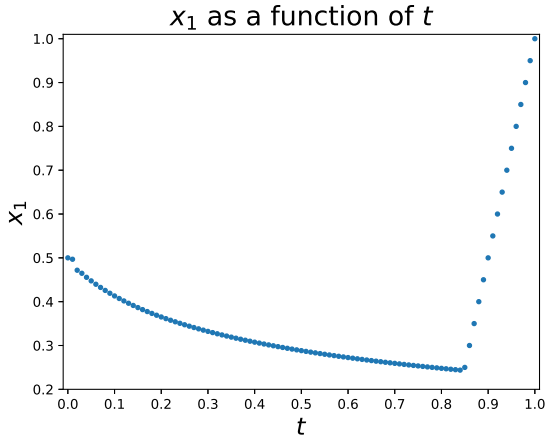

Figure 2.2: Plot of x_1 as a function of t, 100 iterations

If less iterations are used, the final solution is still approximately the same. Figure 2.3 illustrates x_1 versus time for $k = 10$ iterations ($\Delta t = 0.1$). Notice that the shape of both the plots of x_1 versus time are the same and the final solution is still approximately the same.

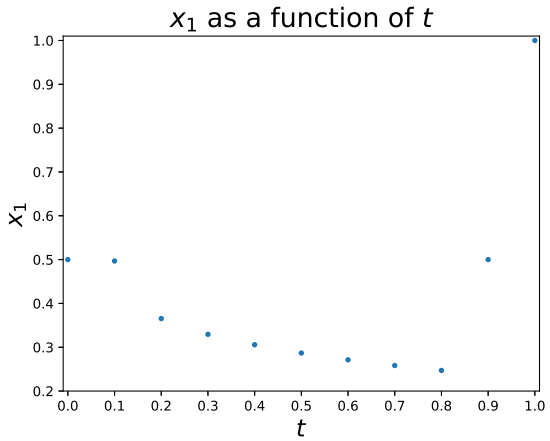

Figure 2.3: Plot of x_1 as a function of t, 10 iterations

While this is a relatively simple problem, it is a good test for Algorithm 2.2 since the problem changes substantially both in the nature of the active constraints and the slope of the objective function from $\mathbf{p_0}$ to $\mathbf{p_f}$ [17].

2.3 Introduction to Dynamic Process Optimization

Given that most optimization problems in chemical processes are dynamic optimization problems, further discossion on dynamic optimization is required. A

dynamic optimization problem is one that has a dynamic process model, meaning that time dependent balances are used to construct a model of the process. Dynamic models are given by an implicit set of differential-algebraic equations (DAEs) and ordinary differential-equations (ODEs).

DAEs are expressed with respect to an independent variable (often t), representing time or distance. In process engineering, DAEs are often written as initial value problems:

$$F\left(x, \frac{dx}{dt}, u(t), \not{p}, t\right), \qquad h(x(0)) = 0 \tag{2.16}$$

where $x(t) \in \mathbb{R}^{n_x}$ are the state variables, $u(t) \in \mathbb{R}^{n_u}$ are control variables, and $\not{p} \in \mathbb{R}^{n_p}$ are variables that are independent of t.

The fully implicit DAEs (Equation 2.16) are difficult to analyze so it is common to consider a simpler form where we partition the state variables into differential variables $z(t)$ and algebraic variables $y(t)$ which leads to the semi-explicit form:

$$\begin{aligned} &\frac{dz}{dt} = f(z(t), y(t), u(t), \not{p}), \qquad z(0) = z_0 \\ &g(z(t), y(t), u(t), \not{p}) = 0 \end{aligned} \tag{2.17}$$

where it is assumed that $y(t)$ can be solved uniquely from $g(z(t), y(t), u(t), \not{p}) = 0$ once $z(t), u(t)$, and $\not{p}$ are specified. DAEs of the form in Equation 2.17 are common in many areas of process engineering where the differential equations come from conservation laws and the algebraic equations from constitutive equations and equilibrium conditions.

Dynamic optimization strategies often have to solve problems in infinite dimensions and provide reasonable levels of approximation even for poorly conditioned or unstable systems. In the following sections, a brief introduction to one of the methods of solving dynamic optimization problems known as direct collocation is conducted.

2.3.1 Direct methods for solving dynamic optimization problems

There are three main methods of solving a dynamic optimization problem: dynamic programming, direct methods, and indirect methods. There are two subcategories of direct methods: sequential methods and simultaneous methods. In this project, simultaneous methods are utilized; specifically the method known as direct collocation. Therefore, no discussion of the other methods is given in this report.

The basic principle of collocation methods is the discretization of both the control and the state variables [1]. Collocation methods are based on Runge-Kutta methods where the a_{ij} and b_i coefficients are constructed in a specific way and are of order at least $\mathscr{K}$ [23].

Direct Collocation

Direct collocation is a fully simultaneous approach since integration and optimization are performed together in the NLP solver [12]. The following properties of this method should be noted:

- The differential constraint is only fulfilled at discrete points (the collocation points)
- Increasing the number of elements increases the accuracy but also the size of the NLP
- Numerical stability properties for one-step methods are inherited

Looking at a generic dynamic system given by

$$\frac{dz}{dt} = f(z(t), t), \qquad z(0) = z_0 \tag{2.18}$$

from which a collocation method can be derived by solving the differential equation at selected points in time. The state variable $\mathbf{x}$ can be approximated using a polynomial approximation of order $\mathscr{K}$ over a single finite element. Figure 2.4 illustrates this polynomial interpolation.

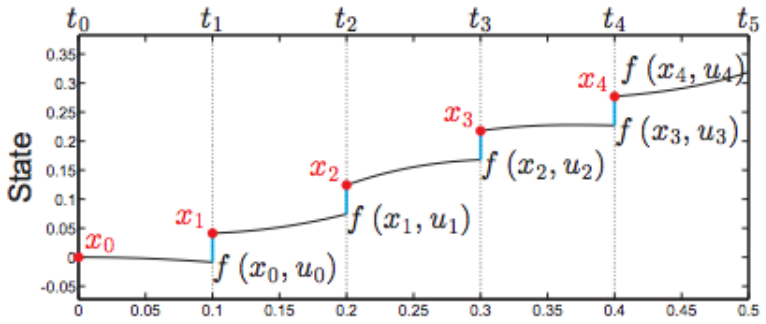

Figure 2.4: Polynomial interpolation of finite elements [12]

Lagrange polynomials are commonly used for the polynomial approximation:

$$P_{k,i}(t) = \prod_{j=0, j\neq i}^{\mathscr{K}} \frac{t - t_{k,j}}{t_{k,i} - t_{k,j}} \in \mathbb{R} \tag{2.19}$$

of order $\mathscr{K}$ and has the following property:

$$P_{k,i}(t_{k,l}) = \begin{cases} 1 & \text{if} \quad l = i \\ 0 & \text{if} \quad l \neq i \end{cases} \tag{2.20}$$

The states $\mathbf{x}$ can then be approximated by interpolating on each time interval

$$\mathbf{x}(\boldsymbol{\theta}_\mathbf{k}, \mathbf{t}) = \sum_{i=0}^{\mathscr{K}} \underbrace{\boldsymbol{\theta}_{k,i}}_{\text{parameters}} \underbrace{P_{k,i}(t)}_{\text{polynomials}} \tag{2.21}$$

where $\mathbf{x}(\boldsymbol{\theta}_\mathbf{k}, \mathbf{t}_{\mathbf{k,j}}) = \boldsymbol{\theta}_{k,j}$. This idea is illustrated in Figure 2.5.

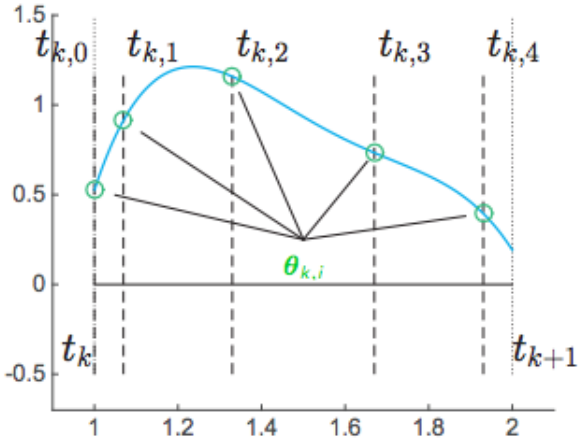

Figure 2.5: Parameter values of polynomial interpolation estimate [12]

The parameters $\boldsymbol{\theta}_{k,i}$ are adjusted to approximate the dynamics $\dot{\mathbf{x}}(\boldsymbol{\theta}_k, \mathbf{t}) = \boldsymbol{F}(\mathbf{x}, \mathbf{u})$. On each interval $[t_k, t_{k+1}]$, the derivative is approximated using Equation 2.21. Collocation uses the constraints

$$\mathbf{x}(\boldsymbol{\theta}_\mathbf{k}, \mathbf{t}_\mathbf{k}) = \boldsymbol{\theta}_{k,0} = \mathbf{x}_\mathbf{k}$$
$$\frac{\partial}{\partial t}\mathbf{x}(\boldsymbol{\theta}_\mathbf{k}, \mathbf{t}_{\mathbf{k,j}}) = \boldsymbol{F}(\mathbf{x}(\boldsymbol{\theta}_\mathbf{k}, \mathbf{t}_{\mathbf{k,j}}), \mathbf{u}_\mathbf{k}), \qquad j = 1, \ldots, \mathscr{K}$$

where $\mathbf{x}_\mathbf{k}$ and $\mathbf{u}_\mathbf{k}$ are coming from the NLP. This can also be written in the form

$$\boldsymbol{\theta}_{k,0} = \mathbf{x}_\mathbf{k} \tag{2.22}$$
$$\sum_{i=0}^{\mathscr{K}} \boldsymbol{\theta}_{k,i}\dot{P}_{k,i}(t)(t_{k,j}) = \boldsymbol{F}(\boldsymbol{\theta}_{k,j}, \mathbf{u}_\mathbf{k}) \tag{2.23}$$

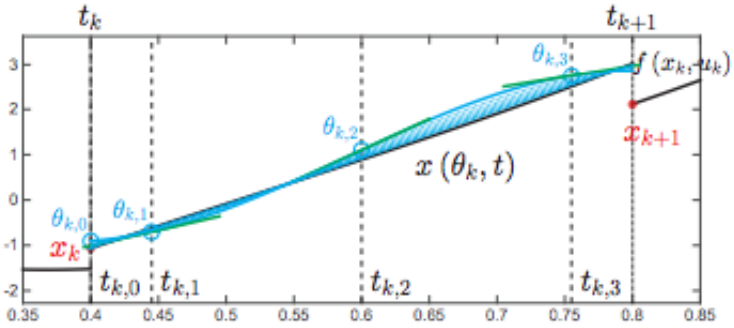

Figure 2.6: Illustration of the direct collocation method [12]

In direct collocation, all constraints are given to the NLP solver. Thus, the NLP formulation becomes:

$$\begin{aligned} \min_{\boldsymbol{w}} \quad & \Phi(w) \\ \text{s.t.} \quad & g(w) = \mathbf{M} \end{aligned} \tag{2.24}$$

where

$$\mathbf{M} = \begin{bmatrix} \boldsymbol{\theta}_{0,0} - \bar{\mathbf{x}}_0 \\ \mathbf{x}(\boldsymbol{\theta}_0, t_1) - \boldsymbol{\theta}_{1,0} \\ \boldsymbol{F}(\boldsymbol{\theta}_{0,i}, \mathbf{u}_0) - \sum_{j=0}^{K} \boldsymbol{\theta}_{0,j} \dot{P}_{0,j}(t_{0,i}) \\ \vdots \\ \mathbf{x}(\boldsymbol{\theta}_k, t_{k+1}) - \boldsymbol{\theta}_{k+1,0} \\ \boldsymbol{F}(\boldsymbol{\theta}_{k,i}, \mathbf{u_k}) - \sum_{j=0}^{K} \boldsymbol{\theta}_{k,j} \dot{P}_{k,j}(t_{k,i}) \\ \vdots \end{bmatrix}$$

The constraints are made up of the initial conditions $\bar{\mathbf{x}}_0$, the continuity constraints, and the integration constraints for $k = 0, \ldots, N-1$. The decision variables w are defined as $w = \{\boldsymbol{\theta}_{0,0}, \ldots, \boldsymbol{\theta}_{0,K}, \boldsymbol{u}_0, \ldots, \boldsymbol{\theta}_{N-1,0}, \ldots, \boldsymbol{\theta}_{N-1,K}, \mathbf{u}_{N-1}\}$. This problem is then solved using a NLP solver.

Chapter 3

Numerical Case Study

3.1 Process Description

The ideal NMPC method and path-following asNMPC method are both applied to an isothermal reactor and separator process shown in Figure 3.1. The continuously-stirred tank reactor (CSTR) receives a stream of pure component A and a recycle stream R from the distillation column. A first-order reaction (A $\longrightarrow$ B) takes place in the CSTR, where B is the desired product. The product is then fed with a flow rate F to the distillation column where the unreacted raw material A is then separated from the product B and recycled to the reactor. The bottom product B must meet a certain specified purity. Table 3.1 summarizes the reaction kinetic parameters for the CSTR. The distillation column model is taken from [24] and is outlined in 3.1.1. The parameters used for the distillation column are summarized in Table 3.2.

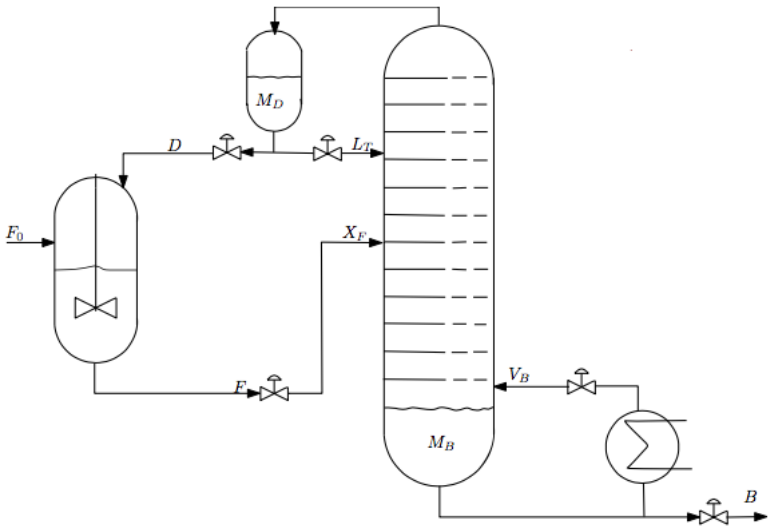

Figure 3.1: Diagram of a CSTR and distillation column system [26]

Table 3.1: Reaction kinetic parameters

Reaction	**Reaction Rate Constant** (min^{-1})	**Activation Energy** ($\mathrm{J\,mol}^{-1}$)
$A \longrightarrow B$	1×10^8	6×10^4

Table 3.2: Distillation column parameters

Parameter	**Value**
α_{AB}	1.5
τ_L	0.063
number of stages	41
feed stage location	21

The distillation column is comprised of 41 theoretical stages: 39 trays, a reboiler, and a total condenser. The feed is an equimolar liquid mixture of components A and B with a relative volatility of 1.5. The pressure is assumed constant due to perfect control using V_T as an input. The reflux and boilup rates are such that nominally there is a 99% purity for each product (y_D and x_B). The nominal holdup is $M_i^*/F = 0.5$ min for all stages, including the reboiler and condenser. A simple linear relationship $L_i(t) = L_i^* + (M_i(t) - M_i^*)/\tau_L$, where $\tau_L = 0.063$ min, is used to model the liquid flow dynamics on all trays.

The following assumptions are used in the construct of the model: binary separation, constant relative volatility, no vapor holdup, one feed and two products, constant molar flows, and a total condenser. Actuator and measurement dynamics are not included in the model. The system (CSTR and distillation column) has a total of 84 state variables: 82 from the distillation column (mole fractions and liquid holdups from each stage) and two from the CSTR (concentration and liquid holdup).

3.1.1 Model Equations

The equations that make up the process model of the CSTR and distillation column system are outlined below.

i) Total balance on stage i:

$$\frac{dM_i}{dt} = L_{i+1} - L_i + V_{i+1} - V_i \tag{3.1}$$

ii) Material balance for light component on each stage i:

$$\frac{d(M_i x_i)}{dt} = L_{i+1}x_{i+1} + V_{i-1}y_{i-1} - L_i x_i - V_i y_i \tag{3.2}$$

which also gives the following expression for the derivative of the liquid mole fraction:

$$\frac{dx_i}{dt} = \frac{\frac{d(M_i x_i)}{dt} - x_i \frac{dM_i}{dt}}{M_i} \tag{3.3}$$

iii) Algebraic equations (applies to all stages except condenser, feed and reboiler):

- Vapor-liquid equilibrium:

$$y_i = \frac{\alpha x_i}{1 + (\alpha - 1)x_i} \tag{3.4}$$

- From assumption of constant molar flows and no vapor dynamics (except if feed is partially vaporized):

$$V_i = V_{i-1} \tag{3.5}$$

- Linearized liquid flow:

$$L_i = L_i^* + \frac{(M_i - M_i^*)}{\tau_L} + (V - V_0)_{i-1} \tag{3.6}$$

where L_i^* kmol min^{-1} and M_i^* kmol are the nominal values for the liquid flow and holdup on stage i.

iv) Feed stage ($i = NF$):

$$\frac{dM_i}{dt} = L_{i+1} - L_i + V_{i-1} - V_i + F \tag{3.7}$$

$$\frac{d(M_i x_i)}{dt} = L_{i+1}x_{i+1} + V_{i-1}y_{i-1} - L_i x_i - V_i y_i + F z_F \tag{3.8}$$

v) Total condenser ($i = NT$):

$$\frac{dM_i}{dt} = V_{i-1} - L_i - D \tag{3.9}$$

$$\frac{d(M_i x_i)}{dt} = V_{i-1}y_{i-1} - L_i x_i - D x_i \tag{3.10}$$

vi) Reboiler ($i = 1$):

$$M_i = M_B \tag{3.11}$$

$$V_i = V_B = V \tag{3.12}$$

$$\frac{dM_i}{dt} = L_{i+1} - V_i - B \tag{3.13}$$

$$\frac{d(M_i x_i)}{dt} = L_{i+1}x_{i+1} - V_i y_i - B x_i \tag{3.14}$$

3.1.2 Column data

The column has 41 stages including the reboiler and total condenser; the feed stage is located at stage 21. The nominal steady state conditions for this column are summarized in Table 3.3; these values were found by performing a steady state optimization on the system with a 1% Gaussian distributed measurement noise added to the states.

Table 3.3: Column data

Parameter	Value	Units
Feed rate F	1	$\mathrm{kmol\,min^{-1}}$
Feed composition z_F	0.5	mole fraction unit
Feed liquid fraction q_F	1	saturated liquid
Reflux flow L_T	2.706	$\mathrm{kmol\,min^{-1}}$
Boilup V	3.206	$\mathrm{kmol\,min^{-1}}$
Liquid holdup M_i^*	0.5	kmol
Time constant for liquid dynamics τ_L	0.063	min
Distillate D	0.5	$\mathrm{kmol\,min^{-1}}$
Distillate composition $y_D = x_{NT}$	0.99	mole fraction units
Bottoms B	0.5	$\mathrm{kmol\,min^{-1}}$
Bottoms composition $x_B = x_1$	0.01	mole fraction units

This steady state data can easily be recalculated to simulate different operating conditions or column setups (number of stages, feed composition, flows, relative volatility, holdups) by changing values in `params.py`, `col_model.py`, and `col_LV.py`.

3.2 Objective Function and Constraints

The economic objective function for this system, which is to be optimized under operation, is given by:

$$J = p_F F_0 + p_V V_B - p_B B - p_D D \tag{3.15}$$

where p_F is the feed cost, p_V is the steam cost, p_D is the distillate price, and p_B is the product price. The following prices are used in this case study: p_F=1 \$/kmol, p_V=0.02 \$/kmol, p_D=0 \$/kmol and p_B=2 \$/kmol. The constraints are the concentration of the bottom product ($x_B \leq 0.1$), the liquid holdup at the bottom and the top of the distillation column and in the CSTR ($0.3 \leq M_{(B,D,CSTR)} \leq 0.7\,\mathrm{kmol}$). The control inputs are the reflux flow (L_T), boil-up flow (V_B), feed rate to the distillation column (F), distillate flow rate (D) and bottom product flow

rate (B). These control inputs have the following bounds:

$$\begin{bmatrix} 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \end{bmatrix} \leq \begin{bmatrix} L_T \\ V_B \\ F \\ D \\ B \end{bmatrix} \leq \begin{bmatrix} 10 \\ 4.008 \\ 10 \\ 1.0 \\ 1.0 \end{bmatrix} \tag{3.16}$$

To solve this problem, the optimal steady-state values must first be calculated to get the optimal values for the control inputs and state variables; a feed rate of $F_0 = 0.3\,\mathrm{kmol\,min^{-1}}$ is selected (see Table 3.3). The optimal steady state input values are found to be $\mathbf{u_{ss}} = \begin{bmatrix} 1.18 & 1.92 & 1.03 & 0.74 & 0.29 \end{bmatrix}^T$.

The optimal steady-state state and control inputs are then used to construct a regularization term that is added to the objective function. Regularization terms are often used in optimization problems because they introduce more information to the function which helps solve an ill-posed problem or prevent overfitting. The regularization term also helps to regulate the different goals of the objective function. The new objective function for the regularized stage is written as:

$$J_m = p_F F_0 + p_V V_B - p_B B - p_D D + (\mathbf{z} - \mathbf{x}_s)^T \mathbf{Q_1} (\mathbf{z} - \mathbf{x}_s) + (\mathbf{v} - \mathbf{u}_s)^T \mathbf{Q_2} (\mathbf{v} - \mathbf{u}_s) \tag{3.17}$$

The weights ($\mathbf{Q_1}$ and $\mathbf{Q_2}$) are selected to make the rotated stage cost of the steady state problem strongly convex. To find a valid diagonal regularization matrix $\mathbf{Q}$, the Gershgorin property for a matrix is applied. This states that for a matrix $\mathbf{A} = (\mathbf{a_{ij}})$:

$$a_{ii} - \sum_{i \neq j} |a_{ij}| \leq \mu_i \leq a_{ii} + \sum_{i \neq j} |a_{ij}| \tag{3.18}$$

where λ_i are the eigenvalues of $\mathbf{A}$ [16]. This property can be utilized to systematically find the regularization terms such that the rotated stage cost will be strongly convex and thus a stable economic NMPC controller can be obtained using this method. For further details on this method, see [16].

Next, the NLP is set up to calculate the predicted state variables $\mathbf{z}$ and the predicted control inputs $\mathbf{v}$. A direct collocation approach is used on finite elements; specifically, Lagrange collocation is utilized to discretize the dynamics and then three collocation points are used in each finite element. Using this approach means that the state variables and the control inputs are actually optimization variables. See 2.3 for further discussion on the use of direct collocation to discretize the dynamic optimization problem.

The economic NMPC case study is initialized using the steady states values for a rate of $F_0 = 0.29\,\mathrm{kmol\,min^{-1}}$ meaning that the economic NMPC controller is essentially controlling for a throughput change from $F_0 = 0.29\,\mathrm{kmol\,min^{-1}}$ to $F_0 = 0.30\ \mathrm{kmol\,min^{-1}}$. The simulation is run for 150 NMPC iterations with a sample time of 1 min. The prediction horizon of the NMPC controller is set to 30 minutes. This results in an NLP with 10,314 optimization variables [26]. To solve the NLP, CasADi [3] with IPOPT [27] is used. To solve the QP, CasADi with qpOASES [8], Gurobi [13], and IPOPT [27] were all tried. Unfortunately, none of the solvers was unable to find a solution for even one NMPC iteration. Further discussion on this issue is conducted in Chapter 5.

Chapter 4

Results

4.1 Closed-Loop Optimization Results

The "true" solution of the dynamic optimization problem $\mathscr{P}_{NMPC}$ versus the steady-state solution is now discussed. First, the distillation column results are analyzed. Figure 4.1 compares the steady-state optimal solution to the dynamic iNMPC solution for closed loop process operation; the Python results are also compared to the MATLAB results from [26]. A disturbance of 0.01 kmol min^{-1} in the feed to the CSTR column is used.

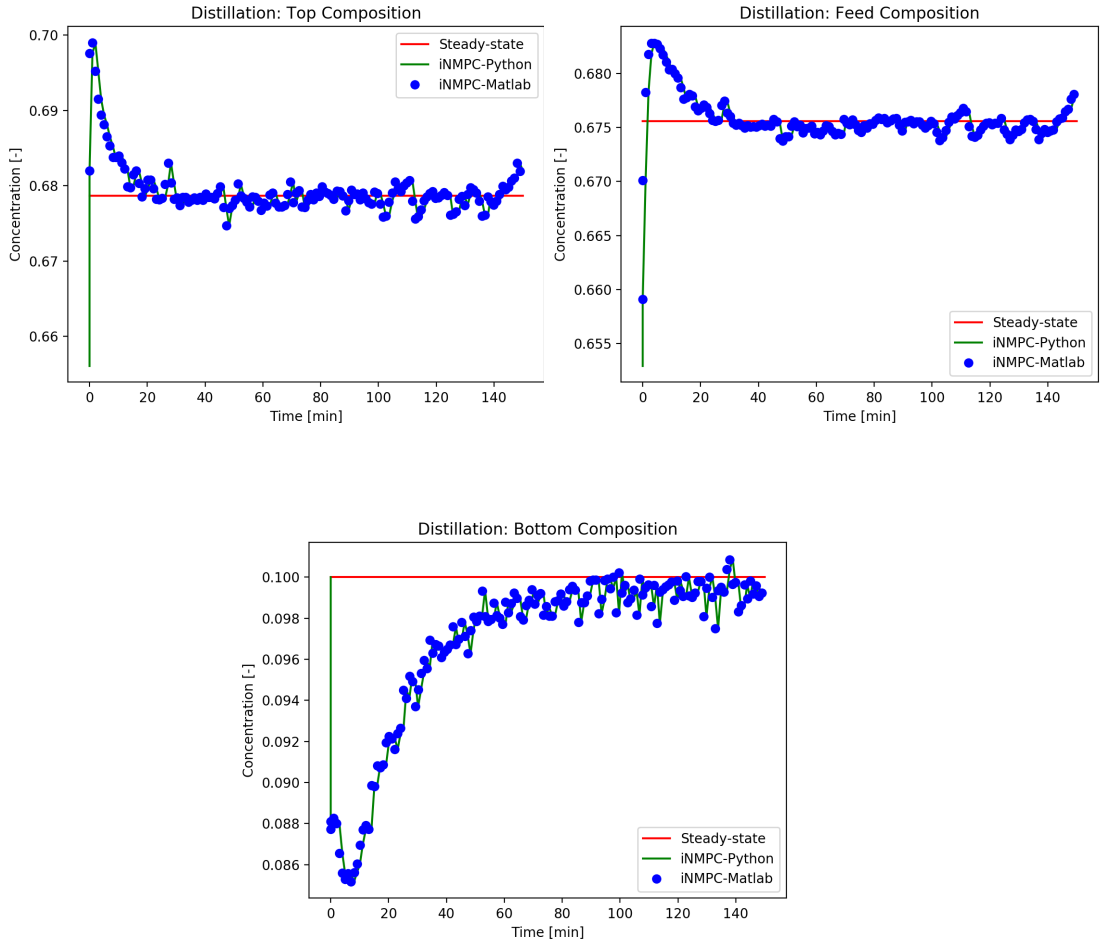

Figure 4.1: Distillation column results

The dynamic optimal solution is controlled to the steady state solution well

in each of the trays (top, feed, and bottom). The fluctuations are a result of the 1% Gaussian distributed noise that was added to the state variables in the simulation. The top composition and distillation column feed composition reach the steady-state value after approximately 25 minutes but the bottom composition does not reach the steady-state value until after approximately 100 minutes. Further illustrated in Figure 4.1 is the match between the Python and the MATLAB implementation. This verifies that the two codes provide the same output for iNMPC.

The CSTR results are shown in Figure 4.2, which compares the steady-state solution to the dynamic iNMPC solution. The concentration has larger fluctuations around the steady-state value in comparison to the distillation columns stages; despite these fluctuations it only requires one iteration to be near the steady-state value. The fluctuations are caused by a combination of the added noise and the changes in the recycle flow rate to the CSTR. It takes about 50 minutes before the CSTR holdup reaches the steady state value. Figure 4.2 shows the match between the two implementations serving as a verification of the outputs.

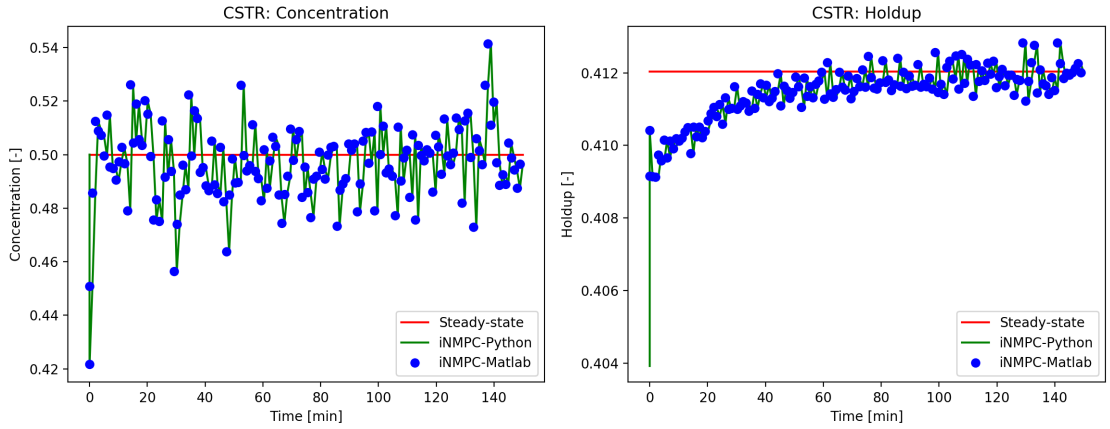

Figure 4.2: CSTR results

The run time for the ideal NMPC method in Python was approximately 9 minutes. In comparison, the ideal NMPC method in MATLAB had a run time of approximately 13 minutes. Both codes were run on the same computer: Lenovo Ideapad with an Intel Core i7 processor and 8GB of RAM. The time difference is likely due to the fact that MATLAB graphical user interface requires utilization of a significant portion of the RAM thus, slowing the solver down. In addition, MATLAB performance slows down considerable with the use of `for` loops, which are utilized in the NMPC method several times.

Chapter 5

Discussion

5.1 MATLAB to Python Conversion

The aim of this project was to convert the work done in [26] into an equivalent Python code. First, a steady state and dynamic model for a CSTR and distillation column system were developed utilizing CasADi [3]. CasADi was selected because it is a "symbolic framework for algorithmic differentiation and numeric optimization" [7]. This allowed for the construction of a symbolic model which could then be evaluated for different operating conditions to produce numerical values. CasADi provides built-in capabilities for the differentiation of thes symbolic equations and thus the construction of the Jacobian and Hessian, which are beneficial to use in the optimization problem. Further, CasADi is open-source under the LGPL license (see Appendix A) and written in C++ code, which can be used in Python "with little to no difference in performance" [7].

The ideal NMPC case was then implemented. As previously mentioned, `IPOPT` was used to solve the NLP problem [27]. `IPOPT` uses a primal-dual interior point method and was selected because it was designed to handle large-scale nonlinear optimization. Further motivation to use this solver came from the fact that an interface to the solver is available in CasADi; therefore, it was trivial to couple the model and the solver. This solver was excellent for this problem since it was able to quickly and accurately solve the NLP problem. Since `IPOPT` was successful, no further discussion is given to NLP solvers. Comparison of the ideal NMPC results from Python to the iNMPC results from MATLAB was used as validation of the model and the code for the iNMPC method (see Chapter 4).

Next the aim was to construct the pfNMPC algorithm in Python using the same system model used for the iNMPC method. The implementation proved problematic as a result of the challenge of finding an appropriate QP solver for this particular problem. Further examination of this issue is given in Section 5.2.

5.2 QP Solver Issues

Despite the numerical case problem being constructed such that the H matrix and A matrix are sparse, neither of the two QP solvers evaluated or the NLP solver tested were able to solve the problem; constructing the problem with sparse matrices was intended to help make a large problem easier for solvers to handle. In [26], a TOMLAB Optimization solver is used but this is not available in an open source form [14]; specifically, `MINOS` (qp-minos), which solves sparse quadratic problems, was utilized [22]. `MINOS` solves linear programs works by using the

primal simplex method [21]. Even though a student version of this solver could be obtained through a free student license of AMPL (which contains MINOS and has an API to Python), the aim is to make this project totally open source (see Appendix A) this solver was not tested for use in Python. Thus, an alternative solver had to be found.

It was proposed to first try `qpOASES` since it is described as a "software package [that] implements a parameteric active-set method for solving convex quadratic programming QP problems", which is exactly the problem type being considered in this project [8]. In addition, CasADi provides a interface and installation of `qpOASES` and, as previously mentioned, CasADi was employed for the model construction. However, this proved unable to solve the problem for even one asNMPC iteration. Next Gurobi's QP solver was tried since CasADi offers an interface to this solver as well; therefore, no problem reformulation is required to use this solver. However, this solver was also unsuccessful at finding a solution. As a last quick fix, `IPOPT` was tried to solve the QP since it had been able to solve the full NLP. This required some minor code changes since `IPOPT` requires a format different than that of the QP solvers. `IPOPT` was able to handle the problem but was unsuccessful in finding a feasible solution even if the step size was decreased using the path-following algorithm. It is possible there was an error in the implementation of the QP to work for the NLP solver and further investigating should done to confirm that the problem was being passed to `IPOPT` correctly. Regardless, it is preferred that a QP solver is found, since it is not efficent to use a NLP solver.

Due to the time constraint, unfortunately, a successful QP solver was unable to be identified; therefore, the asNMPC results for a Python implementation are not provided. Further discussion on the two QP solvers tested is conducted in Sections 5.2.1 and 5.2.2. Other solvers that should be evaluated as part of future work are discussed in 5.3.

5.2.1 qpOASES

`qpOASES` was the first QP solver used but it failed to converge for even one asNMPC iteration and it took a long time to run for one iteration [8]. `qpOASES` was selected because the algorithm uses the QP form known as the primal-dual parameteric quadratic programming method, which is exactly what was desired. While numerical tests have shown that `qpOASES` can outperform other popular academic commercial solvers for small to medium scale convex test examples, this problem proved too large for it to solve [8]. Further investigation into `qpOASES` revealed that the "current implementation can be expected to show satisfactory performance for problems with up to about 1000 unknowns and constraints" [8]; this suggests that the selected numerical case study is far too large for this solver. Even if `qpOASES` was able to find a solution, it appeared to be a slow solver for a problem of this size anyway.

It was difficult to identify the exact reasons why the solver failed because there was insufficient documentation on `qpOASES`'s output in CasADi. The output was of the form: iteration number, step length, information, nFX, nAC. While the contents of column one and two were obvious, the contents of columns three,

four, and five were less so. nFX likely stands for the number of the function being solved; nAC likely stands for the number of active constraints. This made it seem like `qpOASES` solves the optimization row by row which seemed strange. Unfortunately, as stated, it was difficult to find much information on the exact solver `qpOASES` solver used by CasADi so the details of how the solver works are not well understood. Figure 5.1 gives a snapshot of the output format to the terminal.

```
####################   qpOASES  --  QP NO.   1   ####################

   Iter   |    StepLength    |       Info       |   nFX   |   nAC
----------+------------------+------------------+---------+---------
      0   |   7.514823e-02   |   REM BND   85   |  10313  |      0
      1   |   5.500068e-04   |   REM BND  426   |  10312  |      0
      2   |   4.042891e-04   |   REM BND  767   |  10311  |      0
      3   |   3.018922e-04   |   REM BND 1108   |  10310  |      0
      4   |   2.309359e-04   |   REM BND 1449   |  10309  |      0
      5   |   1.913387e-04   |   REM BND 1790   |  10308  |      0
      6   |   1.740671e-04   |   REM BND 2131   |  10307  |      0
      7   |   1.685425e-04   |   REM BND 2472   |  10306  |      0
      8   |   1.682721e-04   |   REM BND 2813   |  10305  |      0
      9   |   1.695698e-04   |   REM BND 3154   |  10304  |      0
     10   |   1.705443e-04   |   REM BND 3495   |  10303  |      0
     11   |   1.704155e-04   |   REM BND 3836   |  10302  |      0
     12   |   1.689908e-04   |   REM BND 4177   |  10301  |      0
     13   |   1.663160e-04   |   REM BND 4518   |  10300  |      0
     14   |   1.624699e-04   |   REM BND 4859   |  10299  |      0
     15   |   1.574629e-04   |   REM BND 5200   |  10298  |      0
     16   |   1.512051e-04   |   REM BND 5541   |  10297  |      0
     17   |   1.435335e-04   |   REM BND 5882   |  10296  |      0
```

Figure 5.1: `qpOASES` output using CasADi wrapper

5.2.2 Gurobi

Next `Gurobi` was tried; while it is a commercial solver, it has a free academic license available to students [13]. This solver was only tried because CasADi offers an interface to it and thus its use does not require any problem reformulation. The Gurobi Optimizer supports all common problem types and states that it is a robust code [13]. Gurobi uses simplex LP solvers.

With the current problem formulation, `gurobi` determined that the model was infeasible and thus could not find a solution. Gurobi gave the warning that the model contained a large quadratic objective coefficient range; it suggested to reformulate the model or set the `NumericFocus` parameter to avoid numerical issues. Setting the `NumericFocus` controls the degree to which the code detects and manages numerical issues; for higher values, the code spends more time focus on being careful in numerical computations. It proved difficult to pass any Gurobi options through the CasADi interface so it was not possible to see if setting the `NumericFocus` would improve performance. Even after adjusting the step size (i.e., applying the path-following algorithm), the solver could not find a solution.

```
Academic license - for non-commercial use only
Warning for adding constraints: zero or small (< 1e-13) coefficients, ignored
Optimize a model with 10164 rows, 10314 columns and 65874 nonzeros
Model has 32910 quadratic objective terms
Coefficient statistics:
  Matrix range     [3e-07, 2e+01]
  Objective range  [2e-08, 3e+00]
  QObjective range [3e-09, 2e+02]
  Bounds range     [2e-03, 9e+00]
  RHS range        [2e-19, 2e+00]
Warning: Model contains large quadratic objective coefficient range
         Consider reformulating model or setting NumericFocus parameter
         to avoid numerical issues.
Presolve removed 0 rows and 8 columns
Presolve time: 0.01s

Barrier solved model in 0 iterations and 0.01 seconds
Model is infeasible
{'x': DM([nan, nan, nan, ..., nan, nan, nan]), 'cost': DM(nan), 'lam_x': DM([0, 0, 0,
 ..., 0, 0, 0]), 'lam_a': DM([0, 0, 0, ..., 0, 0, 0])}
```

Figure 5.2: Gurobi output using Casadi wrapper

5.3 Potential Candidate Solvers

After the two above mentioned QP solvers proved unsuccessful, research was conducted on what other solvers were interfaced with CasADi or interfaced to Python and open-source. In the following sections, the solvers are discussed in more detail. Some options were quickly discarded due to not being open-source and others were discarded due to size constraints or other issues. However, a few solvers worth further investigation were identified.

5.3.1 Other CasADi Interfaced Solvers

CasADi offers interfaces to the following additional QP solvers: `CPLEX`, `HPMPC`, `OOQP`, and `SQIC`. The use of `CPLEX` requires a commercial license so this solver was eliminated from the possibilities. The `HPMPC` solver is meant for Model Predictive Control and requires that the decision variables are only be state and control and that the variable ordering is $[x0\, u0\, x1\, u1]$; it also requires the constraints to be in order. Thus, the use of this solver requires some reformatting of the problem to test. The `SQIC` solver is an implementation of an active-set method utilizing inertial control [29]; however, it is a commercial software and thus was not considered further.

`OOQP` solver is based on the primal-dual interior-point method that can be used for solving convex quadratic programming problems [5]. This solver is not included in the standard installation of CasADi and requires a copy of MA27. MA27 can be downloaded for free from the HSL archive and provides either a personal license or incorporate license as desired. To get a copy of `OOQP` requires filling out a request form [5]. A copy was received but there was insufficient time to test it since a new installation of CasADi would have to be compiled that included the interface to this solver as well as the installation of the solver itself plus the MA27 software.

In summary, only the OOQP solver appears to be a potential option from the list of solvers that CasADi provides an interface to.

5.3.2 Other QP Solvers

Investigating other potential QP solvers lead to the discovery of the following solvers: `quadprog`, `CVXOPT`, `CVXPY` and `MOSEK` [6]. `quadprog` and `CVXOPT`, like `qpOASES`, are numerical solvers and the other solvers are symbolic. `MOSEK` is a commercially licensed solver so it was not considered. Since the problem at hand is a numerical optimization, the numerical solvers are focused on and consequently `CVXPY` was not looked into further. From [6], it appeared that `quadprog` was able to solve problems of any size the fastest with `CVXOPT` being the second best option.

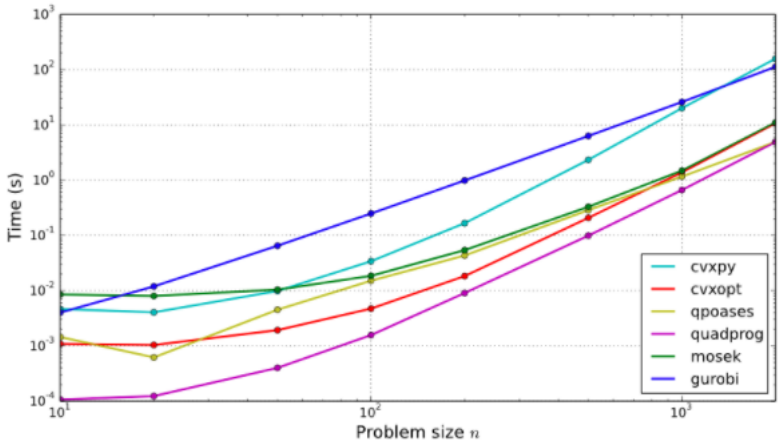

Figure 5.3: Solver time versus problem size [6]

Based on this information, it is most prudent to investigate `quadprog` and `CVXOPT` further. `CVXOPT` requires the use of its own matrix types and thus would require the problem as used with `qpOASES` to be reformulated. `CVXOPT` also it requires that the H matrix is symmetric. The `quadprog` module works directly on NumPy arrays so type conversion is not required. However, there was not much documentation available on how to use this solver.

Wrappers for all the QP solvers shown in Figure 5.3 have been found [6]. This should help decrease the amount of restructuring required to utilize these solvers. However, users should also be wary of using so many wrappers as this may lead to the code having decreased speed. More details on the `quadprog` and `CVXOPT` solvers are provided in Sections 5.3.3 and 5.3.4, respectively.

Other potential QP solvers available in Python that warrant further investigation are: FortMP (supported in AMPL), LOQO (supported in AMPL), MOSEK and pySLEQP [18]. Note that of those listed here only pySLEQP are fully open-source. The other solvers require a license; a trial or student license is available in each instance.

After this project work was submitted, further work was done to find a QP solver. A decision tree that discussed what optimization software to use based on the problem type was discovered [20]. This lead to the discover of the `OSQP` solver [25] which is meant for large problems and already had a Python interface.

5.3.3 Quadprog

The `quadprog` [19] solver minimizes the standard QP form using the Goldfarb/Idnani dual algorithm [11]. It solves quadratic programming problems of the format

$$\begin{aligned} \min_{\mathbf{x}} \quad & \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} - \mathbf{q}^T\mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \end{aligned} \tag{5.1}$$

This solver only works with strictly convex quadratic program problems and requires that the H matrix be symmetric. The documentation for this solver is poor making it difficult to figure out how to use; the wrapper found in [6] may help with this issue. Due to the lack of documentation it could not be determined if `quadprog` is able to handle problems of this size.

5.3.4 CVXOPT

`CVXOPT` is a free software package for convex optimization in Python. It extends built-in Python objects with two matrix objects: `matrix` for dense matrices and `spmatrix` for sparse matrices. `CVXOPT` provides interfaces to several libraries for dense and sparse matrix computations; these include convex optimization solvers written in Python and interfaces to a few other optimization libraries [2]. The function `qp` is considered because it is an interface to the various solvers: `coneqp` and `MOSEK`. `coneqp` uses an interior-point algorithm to solve quadratic programming problems. The QP formulation is

$$\begin{aligned} \min_{\mathbf{x}} \quad & \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{q}^T\mathbf{x} \\ \text{s.t.} \quad & \mathbf{G}\mathbf{x} \leq \mathbf{h}, \\ & \mathbf{A}\mathbf{x} = \mathbf{b} \end{aligned} \tag{5.2}$$

There exists significant documentation on this software making it easier to use than `quadprog`; however, as mentioned, it would require the problem to be redefined. The documentation did not provide any information on what size problems `CVXOPT` is able to handle so it is unknown if this solver would prove sufficient for this problem.

5.3.5 OSQP

`OSQP` solves convex quadratic programs (QP) of the form:

$$\begin{aligned} \min_{\mathbf{x}} \quad & \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{q}^T\mathbf{x} \\ \text{s.t.} \quad & \mathbf{lb} \leq \mathbf{A}\mathbf{x} \leq \mathbf{ub} \end{aligned} \tag{5.3}$$

where $\mathbf{x} \in \mathbb{R}^{n_x}$ is the optimization variable [25]. The objective function consists of a positive semidefinite matrix $\mathbf{H}$ and the vector $\mathbf{q}$. The linear constraints are given by the matrix $\mathbf{A} \in \mathbb{R}^{n_c \times n_x}$ and the vectors $\mathbf{lb} \in \mathbb{R}^{n_c} \cup \{-\infty\}^{n_c}$ and $\mathbf{ub} \in \mathbb{R}^{n_c} \cup \{\infty\}^{n_c}$.

The solver works by using an Alternative Direction Method of Multipliers (ADMM) algorithm [10]. The ADMM algorithm is well suited for distributed convex optimization and in particular, large-scale problems.

Chapter 6

Conclusion

First, a steady-state optimization of the CSTR and distillation column system was implemented. The steady-state results were used as an initial starting point to solve the dynamic optimization problem. The dynamic optimization problem was discretized utilizing collocation. The dynamic problem was then solved using two different methods: "ideal" NMPC and path-following NMPC. The ideal NMPC method works by solving the full problem for every iteration, where the NMPC is constructed as an NLP problem. In comparison, the path-following NMPC utilizes the sensitivity of the NLP solution at the previous iteration to obtain a fast approximate solution to the next iterate of the NMPC problem. The particular approach used in this project solves the full NLP at every sample time but this is done in advance for a predicted initial state. When a new measurement is available, the NLP solution is corrected using the path-following method so that it matches the measured or estimated initial state. The idea is that by pre-solving the full problem at each time-step for a predicted value, the computational time will be shorter and thus, decrease the delay.

The ideal nonlinear model predictive control (iNMPC) method was successfully implemented in Python utilizing `IPOPT` [27] to solve the full NLP. As seen in Chapter 4, the ideal NMPC dynamic optimization results from Python matched the results from MATLAB exactly. The dynamic results are able to be controlled to the steady-state results well for a disturbance of 0.01 $\mathrm{kmol\,min^{-1}}$ in the CSTR feed.

The aim was then to implement the path-following advanced-step nonlinear model predictive control (pfNMPC) algorithm in Python and compare the results to that of the ideal NMPC. However, it proved challenging to find a quadratic programming solver that could solve a problem of this size. While the path-following advanced-step nonlinear model predictive control algorithm has proven to be a valuable alternative to solving the full nonlinear model predictive control problem in [26], it was more problematic to implement in Python than in MATLAB. The next steps are then to test the `quadprog`,`CVXOPT`, and `OSQP` solvers. After a QP solver is found, the pfNMPC algorithm and associated code needs to be verified. The pfNMPC results should then be verified with the MATLAB results. Finally the iNMPC and pfNMPC results and runtimes should be compared to one another.

Bibliography

[1] Johan Åkesson. *Overview of Direct Methods for*. http://www.control.lth.se/media/Education/DoctorateProgram/2011/OptimizationWithCasadi/lecture4b_short_slides.pdf. Accessed: 2017-11-08.

[2] M.S. Andersen, J. Dahl, and L. Vandenberghe. *CVXOPT: A Python package for convex optimization*. Version 1.1.9. URL: http://cvxopt.org/.

[3] Joel Andersson. "A General-Purpose Software Framework for Dynamic Optimization". PhD thesis. Department of Electrical Engineering (ESAT/SCD) and Optimization in Engineering Center, Kasteelpark Arenberg 10 3001-Heverlee Belgium: Arenberg Doctoral School- KU Leuven, 2013.

[4] J. Frèdèric Bonnans and Alexander Shapiro. "Optimization Problems with Perturbations: A Guided Tour". In: *SIAM Rev* 40 (1998), pp. 228–264.

[5] Stephen P. Boyd. *Alternating Direction of Method of Multipliers (ADMM)*. URL: http://web.stanford.edu/~boyd/admm.html (visited on 12/13/2017).

[6] Stephane Caron. *Quadratic Programming in Python*. URL: https://scaron.info/blog/quadratic-programming-in-python.html (visited on 11/28/2017).

[7] *CasADi*. URL: https://github.com/casadi/casadi/wiki (visited on 12/13/2017).

[8] H.J. Ferreau et al. "qpOASES: A parametric active-set algorithm for quadratic programming". In: *Mathematical Programming Computation* 6.4 (2014), pp. 327–363.

[9] Anthony V. Fiacco. *Introduction to Sensitivity and Stability Analysis in Nonlinear Programming*. 1st. Vol. 165. Academic Press, 1983.

[10] Mike Gertz and Steve Wright. *OOQP: Object-oriented software for quadratic programming*. URL: http://pages.cs.wisc.edu/~swright/ooqp/ (visited on 12/13/2017).

[11] D. Goldfarb and A. Idnani. "A numerically stable dual method for solving strictly convex quadratic programs". In: *Mathematical Programming* 27 (1983), pp. 1–33.

[12] Sèbastien Gros. *Numerical Optimal Control with DAES: Lecture 8 Direct Collocation*. https://www.syscop.de/files/2015ws/noc-dae/lecture%20slides/08-Collocation.pdf. (Visited on 11/08/2017).

[13] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2016. URL: http://www.gurobi.com.

[14] Kenneth Holmström and Marcus M. Edvall. "The TOMLAB Optimization Environment". In: *Modeling Languages in Mathematical Optimization*. Ed. by Josef Kallrath. Boston, MA: Springer US, 2004, pp. 369–376. ISBN: 978-1-4613-0215-5. DOI: 10.1007/978-1-4613-0215-5_19. URL: https://doi.org/10.1007/978-1-4613-0215-5_19.

[15] Open Source Initiative. *Licenses and Standards*. URL: `https://opensource.org/licenses` (visited on 12/13/2017).

[16] Johannes Jäschke, Xue Yang, and Lorenz T. Biegler. "Fast Economic Model Predictive Control Based on NLP-Sensitivities". In: *Journal of Process Control* (2014).

[17] Vyacheslav Kungurtsev and Moritz Diehl. "Sequential quadratic programming methods for parametric nonlinear optimization". In: *Computational Optimization and Applications* 59 (2014), pp. 475–509.

[18] Felix Lenders, Christian Kirches, and Hans Georg Bock. *pySLEQP: A Sequential Linear Quadratic Programming Method Implemented in Python*. 2016. URL: `http://mo.tu-bs.de/static/preprints/Lenders2016.pdf`.

[19] Robert T. McGibbon. *Quadratic Programming Solver*. URL: `https://pypi.python.org/pypi/quadprog/` (visited on 11/29/2017).

[20] H.D. Mittelmann. *Decision Tree for Optimization Software*. URL: `http://plato.asu.edu/sub/nlores.html#QP-problem` (visited on 01/15/2018).

[21] Bruce Murtagh and Michael Saunders. *MINOS 5.5*. URL: `http://www.sbsi-sol-optimize.com/asp/sol_product_minos.htm`.

[22] Bruce Murtagh and Michael Saunders. *MINOS 5.5 User's Guide*. URL: `http://www.sbsi-sol-optimize.com/manuals/Minos%20Manual.pdf`.

[23] *Note on Numerical solution of ordinary differential equations initial value problems*. `https://www.math.ntnu.no/emner/TMA4215/2015h/notes/akodenote.pdf`. (Visited on 11/09/2017).

[24] Sigurd Skogestand and Ian Postlethwaite. "Multivariable Feedback Control: Analysis and Design". In: (2005).

[25] B. Stellato et al. "OSQP: An Operator Splitting Solver for Quadratic Programs". In: *ArXiv e-prints* (Nov. 2017). arXiv: `1711.08013 [math.OC]`.

[26] Eka Suwartadi, Vyacheslav Kungurtsev, and Johannes Jäscke. "Sensitivity-Based Economic NMPC with a Path-Following Approach". In: *Processes* 5 (2017), pp. 8–35.

[27] A. Wächter and L.T. Biegler. "On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming". In: *Mathematical Programming* 106 (1 2006), pp. 25–27.

[28] Wikipedia. *Open-source software*. URL: `https://en.wikipedia.org/wiki/Open-source_software` (visited on 12/13/2017).

[29] Elizabeth Wong. "Active-Set Methods for Quadratic Programming". PhD thesis. Department of Mathematics, University of California, San Diego, 2011.

Appendices

Appendix A

Open Source

Open-source software is defined as "computer software with its source code made available with a license in which the copyright holder provides the rights to study, change, and distribute the software to anyone and for any purpose" [28]. Another definition is that open source "describes a broad general type of software license that makes source code available to the general public with relaxed or non-existent restrictions on the use and modification of the code."

The idea behind open-source software is that it leads to more collaborative development which yields a more diverse scope of design. Open-source software is not equal to free software, which is considered a subset of open-source.

A.1 Open-source software licensing

Open-source licenses are licenses that comply with the Open Source Definition; meaning that the license must allow the software to be freely used, modified and shared [15]. The Open Source Initiative (OSI) reviews these licenses and determines if it meets these criteria. The following OSI-approved licenses are widely used [15]:

- Apache License 2.0
- BSD 3-Clause "New" or "Revised" license
- BSD 2-Clause "Simplified" or "FreeBSD" license
- GNU General Public License (GPL)
- GNU Library or "Lesser" General Public License (LGPL)
- MIT license
- Mozilla Public License 2.0
- Common Development and Distribution License
- Eclipse Public License

Each license has different caveats under which its software can be used. It is the user's responsibility to make sure that they compile with these rules.

When using open-source software, it is important to notice what license the software is distributed under. For example, if a software is distributed under the MIT license, any code/software generated utilizing this software can still be sold

commercially. However, if the software is distributed under the GPL license, under no conditions can anything using it be sold as commercial software. Therefore, when developing software it is important to think about the desired market before selecting other softwares to use. Further explanation of all the license types is beyond the scope of this discussion but details can be found online. This short discussion was simply to highlight the importance of selecting a software that uses a license that matches your needs.

Appendix B

Python Code

All of the code created for this project can be downloaded at Github:
`https://github.com/brittanh/masters-project`

B.1 Example Code

This is the code to solve the example problem; the `main.py` is the main file and is the only one that needs to be executed by the user. The main file is where the initial values are defined and then passed to the NLP solver. The NLP solver is used to solve the NLP at the initial parameter and is then used as an initial guess for the QP solver in the path following algorithm.

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: Solving an NLP problem using a path following algorithm
    @author: Brittany Hall
    @date: 18.09.2017
    @version: 0.1
    @updates:
"""
from numpy import array, zeros, linspace, meshgrid, arange, exp
import matplotlib.pyplot as plt
from problem import prob, obj
from nlp_solve import *
from pathfollowing import *

#Initial Values
p_init = array([1,-4])                                  #initial
    parameter value
p_final = array([8,1])                                  #final
    parameter value
x_init = array([[0.5],[0.6]])                             #initial
    primal variable
y_init = array([1.2])                                   #initial
    dual variable


"""
Solving the problem
"""
#Solving NLP at p0 to get initial values
x_opt, lam_opt, mu_opt, con = nlp_solve(prob, obj, p_init, x_init,
    y_init)
```

```
#define method to use (predictor or predictor-corrector)
case = 'predictor-corrector'

#Solving the NLP to get optimal parameters using path-following
    algorithm
x_init, y_init, t_list, x_list_0, x_list_1, lam_list, mu_list, p =
    pathfollowing(p_init, p_final, x_init, x_opt, y_init, lam_opt,
    mu_opt, case)

print(x_list_0)
print(x_list_1)
print(t_list)
print(p)
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
@purpose: Path-following algorithm (algorithm 2 from Suwartadi et al
    2016)
@author: Brittany Hall
@date: 20.09.2017
@version: 0.1
@updates:
"""
from numpy import array, append, zeros
from nlp_solve import *
from qp_solve import *

def pathfollowing(p_init, p_final, x_init, x_opt, y_init, lam_opt,
    mu_opt, case):
    """
    Applying a path following algorithm to an NLP
    """

    #defining empty arrays
    t = 0.0
    t_list = array([])
    x_list_0 = array([])
    x_list_1 = array([])
    y_list = array([])
    lam_list = array([])
    mu_list = array([])
    iter = 1

    #appending initial values
    t_list = append(t_list, t)
    x_list_0 = append(x_list_0,x_init[0])
    x_list_1 = append(x_list_1,x_init[1])
    lam_list = append(lam_list,lam_opt)
    mu_list = append(mu_list,mu_opt)

    #initial algorithm parameters

    delta_t = 0.1                                    #step size
    N = int(1/delta_t)                    #number of iterations
```

```
    alpha1 = 0.25
    p = zeros((N+1,2))
    p[0,:] = (1-t)*p_init + t*p_final
    for k in range(1,N+1):
        print "-----------------------------------------------------"
        print "Iteration number: %d \n" %(iter)

        #calculate step for p
        p[k,:] = (1-t)*p_init + t*p_final
        step = p[k,:] - p[k-1,:]
        if case == 'pure-predictor':
            param = p[k,:]
        elif case == 'predictor-corrector':
            param = p[k,:] + step
        #Solve QP problem
        qp_exit, optimal, x_qpopt, lam_qpopt, mu_qpopt = qp_solve(prob, obj,
                                    param, x_opt, y_init, lam_opt, mu_opt, case)
        print 'QP x:', x_qpopt

        #redefining variables
        del_x= x_qpopt
        del_lam = lam_qpopt
        del_mu = mu_qpopt

        if (qp_exit == 'optimal'):
            x_opt = x_opt + del_x
            if case == 'pure-predictor':
                lam_opt = lam_opt + del_lam * step
                mu_opt = mu_opt + del_mu * step
                lam_list = append(lam_list, lam_opt)
                mu_list = append(mu_list, mu_opt)
            elif case == 'predictor-corrector':
                lam_opt = del_lam
                mu_opt = del_mu
                lam_list = append(lam_list, lam_opt)
                mu_list = append(mu_list, mu_opt)
            t = t + delta_t
            t_list = append(t_list, t)
            x_list_0 = append(x_list_0, x_opt[0])
            x_list_1 = append(x_list_1, x_opt[1])
        else:
            delta_t = alpha1*delta_t
            t = t-alpha1*delta_t
        iter += 1

    return x_opt, y_init, t_list, x_list_0, x_list_1, lam_list, mu_list, p
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: NLP solver
    @author: Brittany Hall
```

```
    @date: 18.09.2017
    @version: 0.1
    @updates:
"""
from casadi import nlpsol

def nlp_solve(prob, obj, p_init, x_init, y_init):
    """
    NLP solver for initial conditions to path-following algorithm
    """
    nx, np, neq, niq, name = prob()
    if niq >0:
        x, p, f, f_fun, con, conf, ubx, lbx, ubg, lbg = obj(x_init,
    y_init, p_init, neq, niq, nx, np)

        #Formulating NLP to solve
        #All constraints must be formatted as inequality constraints
    for this solver
        nlp = {'x':x, 'p':p, 'f':f, 'g':con}
        solver = nlpsol('solver', 'ipopt', nlp)
        sol = solver(x0 = x_init, p = p_init,
                     lbg = lbg, ubg = ubg, ubx = ubx, lbx = lbx)
        x_opt = sol['x']                                        #
    Solving for x
        lagmul = sol['lam_g']
        #Determining active constraints
        #(necessary to determine which multipliers are a lambda and
    which are a mu)
        con_vals = conf(x_opt,p_init)
        tol = 1e-6
        for k in range(0,len(con_vals)):
            if con_vals[k] >= 0 + tol or con_vals[k] >= 0 - tol: #
    active constraint
                lam_opt = lagmul[k]
            else: #inactive constraint
                mu_opt = lagmul[k]
        #print('x_opt:',x_opt,'lambda:',lam_opt,'mu:',mu_opt)
        return x_opt, lam_opt, mu_opt, con
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: Solving a QP
    @author: Brittany Hall
    @date: 20.09.2017
    @version: 0.1
    @updates:
"""
from numpy import array, append, zeros
from casadi import vertcat, gradient, jacobian, hessian, Function,
    conic, SX, mtimes
from problem import prob, obj

#QP solver
def qp_solve(prob, obj, p_init, x_init, y_init, lam_opt, mu_opt, case
```

```
    ):
    """
    QP solver for path-following algorithm
    inputs: prob - problem description
            obj - problem equations
            p_init - initial parameter
            x_init - initial primal variable
            y_init - initial dual variable
            lam_opt - Lagrange multipliers of equality and active constraints
            mu_opt - Lagrange multipliers of inequality constraints
    outputs: y - solution primal variable
            qp_val - objective function value
            qp_exit - return status of QP solver
            deriv - derivatives of the problem
            k_zero_tilde - active set index
            k_plus_tilde - inactive set index
            grad - gradient of objective function
    """
    print 'Current point x:', x_init
    #Importing problem to be solved
    nx, np, neq, niq, name = prob()
    x, p, f, f_fun, con, conf, ubx, lbx, ubg, lbg = obj(x_init, y_init,
                                                        p_init, neq, niq, nx, np)

    #Deteriming constraint types
    eq_con_ind = array([]) #indices of equality constraints
    iq_con_ind = array([]) #indices of inequality constraints
    eq_con = array([]) #equality constraints
    iq_con = array([]) #inequality constraints

    for i in range(0,len(lbg[0])):
        if lbg[0,i] == 0:
            eq_con = vertcat(eq_con,con[i])
            eq_con_ind = append(eq_con_ind,i)
        elif lbg[0,i] < 0:
            iq_con = vertcat(iq_con,con[i])
            iq_con_ind = append(iq_con_ind,i)

    #Evaluating constraints at current iteration point
    con_vals = conf(x_init,p_init)
    #Determining which inequality constraints are active
    k_plus_tilde = array([])                    #active constraint
    k_zero_tilde = array([])                   #inactive constraint
    tol = 10e-5 #tolerance
    for i in range(0, len(iq_con_ind)):
        if ubg[0,i] - tol <= con_vals[i] and  con_vals[i] <= ubg[0,i]+tol:
            k_plus_tilde = append(k_plus_tilde,i)
        else:
            k_zero_tilde = append(k_zero_tilde,i)

```

```
66      nk_pt = len(k_plus_tilde)          #number of active constraints
        nk_zt = len(k_zero_tilde)        #number of inactive constraints
68
    #Calculating Lagrangian
70      lam = SX.sym('lam',neq)           #Lagrangian multiplier (eq)
        mu = SX.sym('mu',niq)           #Lagrangian multiplier(iq)
72      lag_f = f + mtimes(lam.T,eq_con)+ mtimes(mu.T,iq_con)

74      #Calculating derivatives
    g = gradient(f, x)#Derivative of objective function
76      g_fun = Function('g_fun',[x,p], [gradient(f, x)])
        H = 2*jacobian(gradient(lag_f,x),x)     #Second derivative of the
        Lagrangian
78      H_fun = Function('H_fun',[x,p,lam,mu],[jacobian(jacobian(lag_f,x)
        ,x)])

80      if len(eq_con_ind)>0:
            deq = jacobian(eq_con,x)#Derivative of equality constraints
82      else:
            deq = array([])
84      if len(iq_con_ind)>0:
            diq = jacobian(iq_con,x)#Derivative of inequality constraints
86      else:
            diq = array([])
88      #Creating constraint matrices
        nc = niq + neq                        #Total number of constraints
90      if (niq>0) and (neq>0):    #Equality and inequality constraints
            #this part needs to be tested
92          if (nk_zt >0):                    #Inactive constraints exist
                A = SX.zeros((nc,nx))
94              A[0,:] = deq                                    #A matrix
                lba = -1e16*SX.zeros((nc,1))
96              lba[0,:] = -eq_con                        #lower bound of A
                uba = 1e16*SX.zeros((nc,1))
98              uba[0,:]= -eq_con                         #upper bound of A
                for j in range(0,nk_pt): #adding active constraints
100                 A[neq+j+1,:] = diq[int(k_plus_tilde[j]),:]
                    lba[neq+j+1] = -iq_con[int(k_plus_tilde[j])]
102                 uba[neq+j+1] = -iq_con[int(k_plus_tilde[i])]
                for i in range(0,nk_zt): #adding inactive constraints
104                 A[neq+nk_pt+i+1,:] = diq[int(k_zero_tilde[i]),:]
                    uba[neq+nk_pt+i+1] = -iq_con[int(k_zero_tilde[i])]
106                     #inactive constraints don't have lower bounds
            else:                               #Active constraints only
108             A = vertcat(deq,diq)
                lba = vertcat(-eq_con,-iq_con)
110             uba = vertcat(-eq_con,-iq_con)
        elif (niq>0) and (neq==0):                    #Inquality constraints
112         if (nk_zt >0):                    #Inactive constraints exist
                A = SX.zeros((nc,nx))
114             lba = -1e16*SX.ones((nc,1))
                uba =  1e16*SX.ones((nc,1))
116             for j in range(0,nk_pt): #adding active constraints
                    A[j,:] = diq[int(k_plus_tilde[j]),:]
118                 lba[j] = -iq_con[int(k_plus_tilde[j])]
```

```
                uba[j] = -iq_con[int(k_plus_tilde[j])]
            for i in range(0,nk_zt): #adding inactive constraints
                A[nk_pt+i,:] = diq[int(k_zero_tilde[i]),:]
                uba[nk_pt+i] = -iq_con[int(k_zero_tilde[i])]
                    #inactive constraints don't have lower bounds
        else:
            A = vertcat(deq,diq)
            lba = -iq_con
            uba = -iq_con
    elif (niq==0) and (neq>0):                  #Equality constriants
            A = deq
            lba = -eq_con
            uba = -eq_con
    A_fun = Function('A_fun',[x,p],[A])
    lba_fun = Function('lba_fun',[x,p],[lba])
    uba_fun = Function('uba_fun',[x,p],[uba])
    #Checking that matrices are correct sizes and types
    if (H.size1() != nx) or (H.size2() != nx) or (H.is_dense()=='
    False'):
        #H matrix should be a sparse (nxn) and symmetrical
        print('WARNING: H matrix is not the correct dimensions or
    matrix type')
    if (g.size1() != nx) or (g.size2() != 1) or g.is_dense()=='True':
        #g matrix should be a dense (nx1)
        print('WARNING: g matrix is not the correct dimensions or
    matrix type')
    if (A.size1() !=(neq+niq)) or (A.size2() != nx) or (A.is_dense()
    =='False'):
        #A should be a sparse (nc x n)
        print('WARNING: A matrix is not the correct dimensions or
    matrix type')
    if lba.size1() !=(neq+niq) or (lba.size2() !=1) or lba.is_dense()
    =='False':
        print('WARNING: lba matrix is not the correct dimensions or
    matrix type')
    if uba.size1() !=(neq+niq) or (uba.size2() !=1) or uba.is_dense()
    =='False':
        print('WARNING: uba matrix is not the correct dimensions or
    matrix type')

    #Evaluating QP matrices at optimal points
    H_opt = H_fun(x_init,p_init,lam_opt,mu_opt)
    g_opt = g_fun(x_init, p_init)
    A_opt = A_fun(x_init,p_init)
    lba_opt = lba_fun(x_init,p_init)
    uba_opt = uba_fun(x_init,p_init)

    #Defining QP structure
    qp = {}
    qp['h'] = H_opt.sparsity()
    qp['a'] = A_opt.sparsity()
    optimize = conic('optimize','qpoases',qp)
    optimal = optimize(h=H_opt, g=g_opt, a=A_opt,
                       lba=lba_opt, uba=uba_opt, x0=x_init)
    x_qpopt = optimal['x']
```

```
    if x_qpopt.shape == x_init.shape:
166         qp_exit = 'optimal'
    else:
168         qp_exit = ''
    lag_qpopt = optimal['lam_a']
170
    #Determing Lagrangian multipliers (lambda and mu)
172 lam_qpopt = zeros((nk_pt,1))     #Lagrange multiplier of active
    constraints
    mu_qpopt = zeros((nk_zt,1))    #Lagrange multiplier of inactive
    constraints
174 if nk_pt > 0:
        for j in range(0,len(k_plus_tilde)):
176         lam_qpopt[j] = lag_qpopt[int(k_plus_tilde[j])]
    if nk_zt > 0:
178     for k in range(0,len(k_zero_tilde)):
            print lag_qpopt[int(k_zero_tilde[k])]
180 return qp_exit, optimal, x_qpopt, lam_qpopt, mu_qpopt
```

```
#!/opt/local/bin/python
2 # -*- encoding: ascii -*-
"""
4     @purpose: Defining the problem to be solved
      @author: Brittany Hall
6     @date: 18.09.2017
      @version: 0.1
8     @updates:
"""
10
from casadi import SX, Function, vertcat
12 from numpy import array, ones, zeros, exp


14
#Defining the problem
16 def prob():
      """
18    Information on the problem to be solved
      """
20    nx = 2                                          #number of variables
      np = 2                                         #number of parameters
22    neq = 0                          #number of equality constraints
      niq = 2                        #number of inequality constraints
24    name = "Problem 1"
      return nx, np, neq, niq, name
26
def obj(x, y, p, neq, niq, nx, np):
28    """
      Problem to be solved
30    """
      x = SX.sym('x',nx)                                        #Variable
32    p = SX.sym('p',np)                                       #Parameters
      f = p[0]*x[0]**3 + x[1]**2                             #Objective fxn
34    f_fun = Function('f_fun',[x,p],[p[0]*x[0]**3+x[1]**2])

36    con = vertcat(exp(-x[0])-x[1],p[1]-x[0])                 #Constraints
```

```
    conf = Function('conf',[x,p],[exp(-x[0])-x[1],p[1]-x[0]])
38
40  #Specifying Bounds
    ubx = 1e16*ones([1,nx])                    #Variable upper bound
42  lbx = -1e16*ones([1,nx])                   #Variable lower bound
    ubg = zeros([1,niq+neq])                  #Constraint upper bound
44  lbg= -1e16*ones([1,niq+neq])              #Constraint lower bound
    return x, p, f, f_fun, con, conf, ubx, lbx, ubg, lbg
```

B.2 Numerical Case Study Code

This is the code for both the ideal NMPC case and the path-following NMPC case. Both cases utilize all the same code with the exception that the ideal NMPC case uses: `iNMPC.py` and `itPredHorizon.py`; pfNMPC uses:`pfNMPC.py` and `itPredHorizon_pf.py`.

First a steady state optimization is performed. These results are saved (`CstrDistXinit.mat` and `LambdaCstrDist.mat`) and then loaded into the dynamic optimization problem (iNMPC and pfNMPC); the steady state optimal results are used as the initial guess for the dynamic optimization problem.

B.2.1 Steady State Optimization

Run the `ColCSTR_SS.py` file.

```
1  #!/opt/local/bin/python
   # -*- encoding: ascii -*-
3  """
       @purpose: Steady state optimization for CSTR and distillation
       column A
5      Creates 'CstrDistXinit.mat', 'LambdaCstrDist.mat' and 'Qmax.mat'
       @author: Brittany Hall
7      @date: 11.10.2017
       @version: 0.1
9      @updates:
   """
11
   from scipy.io import savemat
13 from casadi import *
   from numpy import append, ones, transpose, shape, abs, size,
       concatenate, array, savetxt
15 from scipy.linalg import eigvals
   from buildmodel import *
17 from params import * #imports cstr and distillation column parameters
   from nlp_solve import *
19 import time

21 #Unpacking parameter values
   NT = params['dist']['NT']
23 LT = params['dist']['LT']
   VB = params['dist']['VB']
```

```
F = params['dist']['F']
D = params['dist']['D']
B = params['dist']['B']


#Symbolic
x = SX.zeros(2*NT+2,1)
l = SX.zeros(2*NT+2,1)
for i in range(0,2*NT+2):
    x[i] = SX.sym('x_'+ str(i+1))
    l[i] = SX.sym('l_'+ str(i+1))

u1 = SX.sym('u1')
            #LT
u2 = SX.sym('u2')
            #VB
u3 = SX.sym('u3')
             #F
u4 = SX.sym('u4')
             #D
u5 = SX.sym('u5')
             #B

#Collecting states and inputs
x = vertcat(x[:])
x = vertcat(x,u1)
x = vertcat(x,u2)
x = vertcat(x,u3)
x = vertcat(x,u4)
x = vertcat(x,u5)

#Decision variables (states and controls)
Xinit = 0.5*ones((2*NT+2,1))
Uinit = vertcat(Xinit, LT)
Uinit = vertcat(Uinit, VB)
Uinit = vertcat(Uinit, F)
Uinit = vertcat(Uinit, D)
Uinit = vertcat(Uinit, B)

#Define the dynamics as equality constraints and additional
    inequality constraints
obj, eq, lbx, ubx, lbg, ubg = buildmodel(x,params)
prob = {'f': obj, 'x': x, 'g': eq}
options = {}
tic = time.time()
startnlp = tic
w0 = Uinit
lbw = lbx
ubw = ubx
sol = nlp_solve(prob, options, w0, lbw, ubw, lbg, ubg)
toc = time.time() - tic
elapsednlp = toc
print('IPOPT solver runtime = %f\n', elapsednlp)

u = sol['x']
```

```
lam = sol['lam_g']
lam[NT+1:-1] = -1*lam[NT+1:-1]
lam = lam.full().flatten()
Xinit = u.full().flatten()


#Saving steady state data to be used in dynamic optimization (.mat
    and .csv)
savemat('CstrDistXinit.mat', {'Xinit': Xinit}, do_compression=True)
savemat('LamdaCstrDist.mat', {'lambda':lam}, do_compression=True)
savetxt('CstrDistXinit.csv', Xinit, delimiter=',')
savetxt('LambdaCstrDist.csv', lam, delimiter=',')

"""
Compute Hessian and perform Greshgorin convexification
"""
xsol = u
lamda = {}
lamda['eqnonlin'] = lam

L = obj + l*eq # Lagrangian

Lagr = Function('Lagr', [x, l], [L], ['x','l'], ['Lagr'])
H = Function('H',[hessian(Lagr),['x','Lagr']])
cons = Function('Const', [x], [eq],['x'], ['cons'])
Jcon = Function(cons.jacobian('x','cons'))

eqVal = cons(xsol)
Hx = H(xsol, lamda['eqnonlin'])
Hx = Hx.full()

Jac = Jcon(xsol)
Jac = Jac.full()

# Nullspace of the constraints and its eigenvalue
rH = transpose(Jac.nullspace())*Hx*Jac.nullspace()
eigen_RH = eigvals(rH)

#Calculating the Greshgorin convexification
def Gershgorin(H):
    numH = H.shape[0]
    Q = zeros((numH,numH))
    delta = 2.5 #with measurement noise of 1 percent
    for i in range(0,numH): #iterate every row of the Hessian
        sumRow = 0
        for j in range(0,numH):
            if j != i:
                sumRow += abs(H[i,j])
        if H[i,i] <= sumRow: #include equality
            Q[i,i] = sumRow - H[i,i] + delta
    Q = diag(Q)
    return H, Q

Hxxl, Qmax = Gershgorin(Hx)
savemat('Qmax', Qmax)
```

```
129 #Check at some initial point for optimization
    xstat = Xinit[0:2*NT+2]
131 u0 = array([[2.5],[3.5],[0.6],[0.5],[0.5]])
    xeval = concatenate((xstat,u0))
133 Jeval = Jcon(xeval)
    Jeval = full(Jeval)
135 Hxxl = H(xeval, lam['eqnonlin'])
    Hxxl = full(Hxxl)
137 Hconv = Hxxl + diag(Qmax)
    rHe = transpose(Jeval.nullspace())*Hconv*Jeval.nullspace()
```

```
    #!/opt/local/bin/python
  2 # -*- encoding: ascii -*-
    """
  4     @purpose: Creates objective function and constraints for
        Distillation column
        A model and CSTR
  6     @author: Brittany Hall
        @date: 11.10.2017
  8     @version: 0.1
        @updates:
 10 """
    from casadi import *
 12 from numpy import divide, multiply, zeros, array

 14 def buildmodel(u, params):
        #Unpacking model parameters
 16     NT = params['dist']['NT']                                    #number
         of stages
        NF = params['dist']['NF']                                  #stage where
        feed enters
 18     alpha = params['dist']['alpha'] #relative volatility
        Muw = params['dist']['Muw'] #nominal liquid hold ups
 20     taul = params['dist']['taul'] #time constant for liquid dynamics
        F = params['dist']['F'] #nominal distillation feed flowrate
 22     qF = params['dist']['qF'] #nominal distillation feed liquid
        fraction
        L0 = params['dist']['L0'] #nominal reflux flow
 24     L0b = params['dist']['L0b'] #nominal liquid flow below feed
        F_0 = params['dist']['F_0'] #nominal CSTR feed flowrate
 26     zF = params['dist']['zF'] #nominal feed composition

 28     #Inputs and disturbances
        LT = u[2*NT+2]
           #Reflux
 30     VB = u[2*NT+3]
           #Boilup
        F = u[2*NT+4]
        #Feedrate
 32     D = u[2*NT+5]                                                #
        Distillate
        B = u[2*NT+6]
          #Bottoms
 34
```

```
    """
    The Model
    """
    #Objective function
    pf = params['price']['pf']
    pV = params['price']['pV']
    pB = params['price']['pB']
    pD = params['price']['pD']
    J = pf*F_0 + pV*VB - pB*B - pD*D

    #Vapor and Liquid flowrates, composition, and holdups
    y = SX.zeros(NT-1)
    V = SX.zeros(NT-1)
    L = SX.zeros(NT)
    dMdt = SX.zeros(NT+1)
    dMxdt = SX.zeros(NT+1)
    for i in range(0,NT-1):
        y[i] = SX.sym('y_'+str(i+1),1)
        V[i] = SX.sym('V_'+str(i+1),1)
        L[i] = SX.sym('L_'+str(i+1),1)
        dMdt[i] = SX.sym('dMdt_'+str(i+1),1)
        dMxdt[i] = SX.sym('dMxdt_'+str(i+1),1)
    L[NT-1] = SX.sym('L_'+str(NT),1)
    dMdt[NT-1] = SX.sym('dMdt_'+str(NT),1)
    dMxdt[NT-1] = SX.sym('dMxdt_'+str(NT),1)
    dMdt[NT] = SX.sym('dMdt_'+str(NT+1),1)
    dMxdt[NT] = SX.sym('dMxdt_'+str(NT+1),1)

    #Vapor-liquid equilibria
    for i in range(0,NT-1): #don't calculate value for last stage NT
        y[i] = alpha*u[i]/(1+(alpha-1)*u[i])

    #Vapor flows (constant molar flows assumed)
    for i in range(0,NT-1):#don't calculate value for last stage NT
        if i >= NF-1:
            V[i] = VB + (1-qF)*F
        else:
            V[i] = VB

    #Liquid flows
    L[NT-1] = LT #last stage liquid
    for i in range(0,NT-1):#don't calculate value for last stage NT
        if i <= NF-1:
            L[i] = L0b + divide((u[NT+1+i] - Muw),taul)
        else:
            L[i] = L0 + divide((u[NT+1+i] - Muw),taul)

    #Time derivatives for material balances for total holdup and
    component
    for i in range(1,NT-1):
        dMdt[i] = L[i+1] - L[i] + V[i-1] - V[i]
        dMxdt[i] = multiply(L[i+1], u[i+1,0]) - multiply(L[i], u[i
    ,0]) + multiply(V[i-1], y[i-1]) - multiply(V[i],y[i])

    #Correction for feed stage
```

```
88      dMdt[NF-1] = dMdt[NF-1] + F
        dMxdt[NF-1] = dMxdt[NF-1] + F*u[NT]
90
        #Reboiler (assumed to be an equilibrium stage)
92      dMdt[0] = L[1] - V[0] - B
        dMxdt[0] = L[1]*u[1] - V[0]*y[0] - B*u[0]
94
        #Total condenser (not an equilbrium stage)
96      dMdt[NT-1] = V[NT-2] - LT - D
        dMxdt[NT-1] = V[NT-2]*y[NT-2] - LT*u[NT-1] - D*u[NT-1]
98
        #Compute the derivative for the mole fractions d(Mx) = xdM + Mdx
100     ceq = SX.zeros(2*NT+2)
        for i in range(0,2*NT+2):
102         ceq[i] = SX.sym('ceq_'+str(i+1),1)

104     #CSTR model
        k1 = params['cstr']['k1']
106     dMdt[NT] = F_0 + D - F
        dMxdt[NT] = F_0*zF + D*u[NT-1] - F*u[NT] - k1*u[2*NT+1]*u[NT]
108
        for i in range(0,NT+1):
110         ceq[i] = dMxdt[i]

112     for i in range(0,NT+1):
            ceq[NT+1+i] = dMdt[i]
114
        #Constraint bounds
116     lbx = params['bounds']['lbx']
        ubx = params['bounds']['ubx']
118     lbg = params['bounds']['lbg']
        ubg = params['bounds']['ubg']
120
        return J, ceq, lbx, ubx, lbg, ubg
```

```
1   #!/opt/local/bin/python
    # -*- encoding: ascii -*-
3   """
        @purpose: NLP solver
5       @author: Brittany Hall
        @date: 18.09.2017
7       @version: 0.1
        @updates:
9   """
    from casadi import *
11
    def nlp_solve(prob, options, w0, lbw, ubw, lbg, ubg):
13      """
        NLP solver for initial conditions to path-following algorithm
15      """
        #Formulating NLP to solve
17      solver = nlpsol('solver', 'ipopt', prob, options)
        sol = solver(x0 = w0, lbx = lbw, ubx = ubw, lbg=lbg, ubg=ubg)
19      return sol
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: Generates noise for states
    @author: Brittany Hall
    @date: 23.10.2017
    @version: 0.1
    @updates:
"""
import scipy.io as spio
from numpy import zeros, array, append, random

mpc_iter = 500
noiselevel = 0.1 # 1 percent noise

#Load in steady state data
data = spio.loadmat('CstrDistXinit.mat', squeeze_me = True)
Xinit = data['Xinit']
xf = Xinit[0:2*NT+2]
xholdup = xf[NT+1:-1]

noise = array([])
for i in range(0,mpc_iter):
    noise = append(noise, noiselevel*xholdup*random.randn(NT+1,1))

print noise
raw_input()
spio.savemat('noise1pct',noise)
```

B.2.2 Dynamic Optimization

Run the `process_main.py` file.

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: Main file to run iNMPC and pfNMPC
    @author: Brittany Hall
    @date: 06.10.2017
    @version: 0.1
    @updates:
"""
from numpy import reshape, tile
import scipy.io as spio
#user made functions
from optProblem import *
from system import *
from pfNMPC import *
from iNMPC import *
from params import *
from plotting import *

#MPC iterations
```

```
MPCit = 150
#Prediction Horizon
N = 30
#Sampling time
T = 1                                                    #[min]

#Loading in initial data (different initial conditions)
data = spio.loadmat('Xinit29.mat', squeeze_me = True)
Xinit = data['Xinit29']
u0 = Xinit[84:89]                                    #Initial inputs
u0 = u0.reshape(len(u0),1)
u0 = tile(u0,N)
tmeasure = 0.0                                          #Start time
xmeasure = Xinit[0:84]                                #Initial states
Uf = 0.3                                        #Feed rate to CSTR (F_0)
params['dist']['F_0'] = Uf

#Applying ideal NMPC
#_, xmeasureAll, uAll,_, _, _, runtime = iNMPC(optProblem, system,
    MPCit, N, T, tmeasure, xmeasure, u0, params)

#print "iNMPC finished \n"

#Applying path-following NMPC
_, xmeasureAll_pf, uAll_pf, _, _, _, runtime_pf = pfNMPC(optProblem,
    system, MPCit, N, T, tmeasure, xmeasure, u0, params)

print "pfNMPC finished in %f\n" %runtime_pf

#Plotting results
#plotting(u0, xmeasure, MPCit, T)
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: CSTR model (stage NT+1) with a first order
    reaction (A-> B) plus nonlinear distillation column model
    with NT-1 theoretical stages including a reboiler (stage 1)
    plus a total condenser (stage NT).
    The model is based on column A in Skogestad and Postlethwaite
    (1996).
    @author: Brittany Hall
    @date: 05.10.2017
    @version: 0.1
    @updates:
"""
from casadi import SX
from numpy import *
from params import *

def col_cstr_model(t, X, U):
    #Column Information
    """
        Inputs: t - time [min]
                X - States, the first 41 states are compositions
```

```
                of light component A with reboiler/bottom
                stage as X(0) and condenser as X(40). X(41) is
                the holdup in the reboiler/bottom stage and X(81)
                is the hold-up in condenser
            U[0] - reflux L
            U[1] - boilup V
            U[2] - top or distillate product flow D
            U[3] - bottom product flow B
            U[4] - feed rate F
            U[5] - feed composition zF
            U[6] - feed rate F0

        Outputs: xprime - vector of time derivative all states
    """
    #Unpacking model parameters
    #==============Column Dependent Properties================#
    NC = params['dist']['NC']
    NF = params['dist']['NF']
    NT = params['dist']['NT']
    qF = params['dist']['qF']
    alpha = params['dist']['alpha']
    zF0 = params['dist']['zF']
    M0 = params['dist']['M0']
    F_0 = U

    #Data for linearized Liquid flow dynamics
    #(does not apply to reboiler and condenser)
    taul = params['dist']['taul']
    F0 = params['dist']['F0']
    qF0 = params['dist']['qF0']
    L0 = params['dist']['L0']
    L0b = L0 + qF0*F0
    lam = params['dist']['lam']
    V0 = params['dist']['V0']
    V0t = V0 + (1-qF0)*F0

    #=========================================================#

    #Dividing the states
    #Liquid composition of column plus composition in tank
    x = X[0:NT+1]

    #Liquid hold up from btm to top of col plus hold up in tank
    M = X[NT+1:]

    #Inputs and Disturbances
    LT = U[0]                                   #Reflux flowrate
    VB = U[1]                                   #Boilup flowrate
    D = U[2]                                 #Distillate flowrate
    B = U[3]                                    #Bottoms flowrate
    F = U[4]                               #Distillation feed flowrate
    zF_0 = U[5]                                 #CSTR Feed composition
    qF = params['dist']['qF']                    #Feed liquid fraction
    F_0 = U[6]                                          #CSTR flowrate
```

```
    """
    Model Development
    """
    #Vapor-liquid equilibria
    y = []
    for i in range(0,NT-1):
        y.append(alpha*x[i]/(1+(alpha-1)*x[i]))

    #Vapor flows (assuming constant molar flow)
    V = []
    for i in range(0,NT-1):
        V.append(VB)
    for i in range(NF,NT-1):
        V[i] = V[i] + (1-qF)*F

    #Liquid flows (assuming linearized tray hydraulics)
    L = []
    L.append(0)
    for i in range(1,NF):
        L.append(L0b +  (M[i]-M0[i])/taul)

    for i in range(NF,NT-1):
        L.append(L0 + (M[i]-M0[i])/taul)

    L.append(LT)

    """
    Time Derivatives of material balances for total
    holdup and component holdup
    """
    #Column
    dMdt = []
    dMdt.append(0)
    dMxdt = []
    dMxdt.append(0)
    for i in range(1,NT-1):
        dMdt.append (L[i+1] - L[i] + V[i-1] - V[i])
        dMxdt.append(L[i+1]*x[i+1]-L[i]*x[i]+V[i-1]*y[i-1]
                     -V[i]*y[i])

    #Correction for feed at feed stage
    dMdt[NF-1] = dMdt[NF-1] + F
    dMxdt[NF-1] = dMxdt[NF-1] + x[NT]*F

    #Reboiler (assumed to be an equilibrium stage)
    dMdt[0] = L[1]-V[0]-B
    dMxdt[0] = L[1]*x[1]-V[0]*y[0] -B*x[0]

    #Total condensor (not an equilibrium stage)
    dMdt.append(V[NT-2] - LT - D)
    dMxdt.append(V[NT-2]*y[NT-2]-LT*x[NT-1]-D*x[NT-1])

    #CSTR Model (inputs F_0 z_F0)
    k1 = params['cstr']['k1']
    dMdt.append(F_0 + D - F)
```

```
    dMxdt.append(F_0*zF0[0,0] + D*x[NT-1] - F*x[NT]
134                 -k1*M[NT]*x[NT])

136     #Calculating the derivative of the mole fractions
    dxdt = []
138     for i in range(0,NT+1):
        dxdt.append((dMxdt[i]-x[i]*dMdt[i])/M[i])
140
    xprime = append(dxdt,dMdt)
142     return xprime
```

```
#!/opt/local/bin/python
2 # -*- encoding: ascii -*-
"""
4     @purpose: Ideal Nonlinear Model Predictive Control (iNMPC)
    @author: Brittany Hall
6     @date: 11.10.2017
    @version: 0.1
8     @updates:
"""
10 from numpy import size, zeros, append, hstack, savetxt, reshape
from compObjFn import *
12 from solveOpt import *
from plotStates import *
14 from scipy.io import savemat, loadmat

16 def iNMPC(optProblem, system, MPCit, N, T, tmeasure, xmeasure, u0,
    params):

18     #Unpacking required parameters
    NT = params['dist']['NT']
20
    #Constructing empty arrays for later use
22     Tall = []
    Xall = zeros((MPCit, size(xmeasure, axis = 0)))
24     Uall = zeros((MPCit, size(u0, axis = 0)))
    ObjVal = {}
26     ObjVal['econ'] = []
    ObjVal['reg'] = []
28     xmeasureAll = []
    uAll = []
30     xAll = []
    runtime = []
32     u_nlp_opt = []
    x_nlp_opt = []
34
    #NMPC iteration
36     iter = 1

38     #Load in noise data
    data = loadmat('noise1pct.mat', squeeze_me = True)
40     noise = data['noise']

42     while (iter <= MPCit):
        print "--------------------------------------------------\n"
```

```
        print "MPC iteration: %d \n" %(iter)

        #Obtaining new initial value
        def measureInitVal(tmeasure, xmeasure):
            t0 = tmeasure
            x0 = xmeasure
            return t0, x0
        t0,x0 = measureInitVal(tmeasure, xmeasure)

        #Measurement noise
        n_M = noise[:,iter-1]                    #Holdup noise
        n_X = zeros((NT+1,1))              #Concentration noise
        measure_noise = append(n_X, n_M)
        x0_measure = x0 + measure_noise#Add measmt noise to states

        #Solving NLP
        primalNLP,_,lb,ub,_,params,_=solveOpt(optProblem,x0,
                                              u0,N,x0_measure,params)

        #Re-arrange NLP solutions
        #(turning vectors into matrices to make easier to plot)
        u_nlp_opt, x_nlp_opt = plotStates(primalNLP, lb, ub, N,
    params)

        #Save open loop solution for error computation
        z1 = x_nlp_opt[0:nx,4]

        #Record information
        Tall = append(Tall, t0)
        Xall[iter-1,:] = transpose(x0)
        Uall[iter-1,:] = u0[:,0]

        #Applying control to process with optimized control
        def dynamic(system, T, t0, x0, u0):
            x = system(t0, x0, u0, T)
            x_intermediate = append(x0, x)
            t_intermediate = hstack([t0, t0+T])
            return x, t_intermediate, x_intermediate

        def applyControl(system, T, t0, x0, u0):
            xapplied, _, _ = dynamic(system, T, t0, x0, u0[:,0])
            tapplied = t0 + T
            return tapplied, xapplied

        #Apply control to process with optimized
        #control from path-following algorithm
        x0 = xmeasure                            #From online step
        tmeasure,xmeasure=applyControl(system,T,t0,x0,u_nlp_opt)

        #Using actual state
        Jobj = compObjFn(u_nlp_opt[:,0], xmeasure)

        #Storing Output Variables
        ObjVal['econ'].append(float(Jobj['econ'][0]))
        ObjVal['reg'].append(float(Jobj['reg'][0]))
```

```
98
            xmeasureAll = append(xmeasureAll, xmeasure)
100         uAll = append(uAll, u_nlp_opt[:,0])
            runtime = append(runtime, elapsedtime)
102
            def shiftHorizon(u):
104             u0 = hstack((u[:,1:u.shape[1]], u[:,u.shape[1]-1]))
                return u0
106
            u0 = shiftHorizon(u_nlp_opt)
108
            iter += 1
110
        xmeasureAll = reshape(xmeasureAll,(xmeasureAll.shape[0],1))
112     xmeasureAll = reshape(xmeasureAll, (2*NT+2, MPCit))
        xmeasureAll = array(xmeasureAll)
114
        ObjReg = array(ObjVal['reg'])
116     ObjEcon = array(ObjVal['econ'])

118     ideal = {
                'ideal':{
120                 'xmeasureAll': xmeasureAll,
                    'uAll': uAll,
122                 'ObjReg': ObjReg,
                    'ObjEcon': ObjEcon,
124                 'T': T,
                    'mpciterations': MPCit
126                 }
        }
128
        savemat('iNMPC.mat',ideal)              #saving iNMPC results
130
        return Tall, xmeasureAll, uAll, ObjVal,primalNLP,params,runtime
```

```
1  #!/opt/local/bin/python
   # -*- encoding: ascii -*-
3  """
       @purpose: Path- following based Nonlinear Model Predictive
       Control (pfNMPC)
5      @author: Brittany Hall
       @date: 07.10.2017
7      @version: 0.1
       @updates:
9  """
   from solveOpt import solveOpt
11 from scipy.io import savemat, loadmat
   from plotStates import plotStates
13 from ColCSTR_pf import ColCSTR_pf
   from predictor_corrector import predictor_corrector
15 from numpy import size, zeros, append, array, hstack, vstack
   from compObjFn import *
17
   def pfNMPC(optProblem, system, MPCit, N, T, tmeasure, xmeasure, u0,
       params):
```

```
19
    NT = params['dist']['NT']
21  #Dimension of state and input
    nx = size(xmeasure) #Elements in state
23  nu = size(u0, axis = 0) #Size of inputs
    #Constructing empty arrays for later use
25  Tall = []
    Xall = zeros((MPCit, xmeasure.shape[0]))
27  Uall = zeros((MPCit, u0.shape[0]))
    ObjVal = {}
29  ObjVal['econ'] = []
    ObjVal['reg'] = []
31  xmeasureAll = []
    uAll = []
33  runtimepf = []
    u_pf_opt = []
35  x_pf_opt = []

37  #starting NMPC iteration
    iter = 1
39  z1 = xmeasure
    #loading in noise data
41  data = loadmat('noise1pct.mat', squeeze_me = True)
    noise = data['noise']
43  runtime_pf = 0

45  while (iter <= MPCit):
        print("-----------------------------------------\n")
47      print("MPC iteration: %d\n" %iter)

49      #Obtaining new initial value
        def measureInitVal(tmeasure, xmeasure):
51          t0 = tmeasure
            x0 = xmeasure
53          return t0, x0
        t0,x0 = measureInitVal(tmeasure, xmeasure)
55
        #adding measurement noise
57      n_M = noise[:,iter-1]                    #Holdup noise
        n_X = zeros((NT+1,1))              #Concentration noise
59      measure_noise = append(n_X, n_M)
        x0_measure = x0 + measure_noise  #Add measmt noise to states
61
        #advanced-step NMPC
63      primalNLP, dualNLP, lb, ub, objVal, params,_ = solveOpt(
    optProblem, x0, u0, N, z1, params)

65      #re-arrange NLP solutions
        _, x_nlp_opt = plotStates(primalNLP, lb, ub, N, params)
67
        p_init = primalNLP[0:nx]
69      p_final = x0_measure
        xstart = primalNLP
71      ystart = dualNLP
```

```
        delta_t = 0.5                                                    #
    Step size
        lb_init = lb
        ub_init = ub

        #NLP sensitivity (predictor-corrector)
        primalPF, _, elapsedqp = predictor_corrector(lambda p:
    ColCSTR_pf(p),
            p_init, p_final, xstart, ystart, delta_t, lb_init,
    ub_init, 0, N)

        #Formatting variables for plotting
        u_pf_opt, x_pf_opt = plotStates(primalPF, lb, ub, N, params)
        z1 = x_pf_opt[0:nx,5]

        #Store output variables
        Tall = append(Tall, t0)
        Xall[iter-1,:] = transpose(x0)
        Uall[iter-1,:] = u0[:,0]

        #Apply control to process with optimized control from pf-
    algorithm
        x0 = xmeasure #from online step

        def dynamic(system, T, t0, x0, u0):
            x = system(t0, x0, u0, T)
            x_intermediate = vstack((x0, x))
            t_intermediate = hstack((t0, t0+T))
            return x, t_intermediate, x_intermediate

        def applyControl(system, T, t0, x0, u0):
            xapplied, _, _ = dynamic(system, T, t0, x0, u0[:,0])
            tapplied = t0 + T
            return tapplied, xapplied

        tmeasure, xmeasure = applyControl(system, T, t0, x0, u_pf_opt
    )

        #Using actual states to compute the objective function values
        Jobj = compObjFn(u_pf_opt[:,0],xmeasure)

        #Storing Output Variables
        ObjVal['econ'].append(float(Jobj['econ'][0]))
        ObjVal['reg'].append(float(Jobj['reg'][0]))

        #Collect variables
        xmeasureAll = append(xmeasureAll, xmeasure)
        uAll = append(uAll, u_pf_opt[:,0])
        runtime_pf = append(runtime_pf, elapsedqp)

        #Prepare restart
        def shiftHorizon(u):
            u0 = hstack((u[:,1:u.shape[1]], u[:,u.shape[1]-1]))
            return u0
```

```
123         u0 = shiftHorizon(u_pf_opt)

125         iter += 1

127     xmeasureAll = reshape(xmeasureAll,(xmeasureAll.shape[0],1))
        xmeasureAll = reshape(xmeasureAll, (2*NT+2, MPCit))
129     xmeasureAll = array(xmeasureAll)

131     ObjReg = array(ObjVal['reg'])
        ObjEcon = array(ObjVal['econ'])
133
        pathfollowing = {
135                 'pf':{
                        'xmeasureAll': xmeasureAll,
137                     'uAll': uAll,
                        'ObjReg': ObjReg,
139                     'ObjEcon': ObjEcon,
                        'T': T,
141                     'mpciterations': MPCit
                        }
143                 }

145     savemat('pfNMPC.mat',pathfollowing)             #saving pfNMPC
        results

147     return Tall, xmeasureAll, uAll, ObjVal, primalPF, params,
        runtime_pf
```

```
1  #!/opt/local/bin/python
   # -*- encoding: ascii -*-
3  """
       @purpose: Distillation Column A and CSTR model parameters
5      @author: Brittany Hall
       @date: 11.10.2017
7      @version: 0.1
       @updates:
9  """
   from numpy import zeros, ones, concatenate, array
11 params = {}
   #--------------Distillation column parameters---------------#
13 NC = 2                                    #Number of components
   NT = 41                                      #Number of stages
15 NF = 21                                  #Location of feed stage
   LT = 2.827                                             #Reflux
17 VB = 3.454                                             #Boilup
   F = 1.0                                               #Feedrate
19 zF = array([[1.0],[0.0]])  #Feed composition (# components)
   D = 0.5                                  #Distillate flowrate
21 B = 0.5                                      #Bottoms flowrate
   qF = 1.0                                 #Feed liquid fraction
23 F_0 = 0.3                                       #CSTR Feed rate
   F0 = F                             #Nominal feed rate to column
25 qF0 = qF
   alpha = 1.5                              #Relative volatility
27 #Nominal liquid holdups
```

```
Muw = 0.5
MO = zeros(NT+1)
MO[0] = 0.5                    #Nominal reboiler holdup [kmol]
MO[1:NT-1] = 0.5             #Nominal stage (tray) holdup [kmol]
MO[NT-1] = 0.5                 #Nominal condenser holdup [kmol]
MO[NT] = 0.5                      #Nominal CSTR hold up [kmol]
#Linearized flow dynamics (NA to reboiler and condenser)
taul = 0.063          #Time constant for liquid dynamics [min]
L0 = 2.70629
L0b = L0 + qF*F0  #Nominal liquid flow below feed [kmol/min]
lam = 0
V0 = 3.206
VB_max = 4.008
#----------------------CSTR parameters-------------------------#
#Reaction
k1 = 34.1/60.0
#---------------Objective Function & Constraints--------------#
#Prices
pf = 1
pV = 0.02
pB = 2
pD = 0
#Gains
KcB = 10
KcD = 10
#Nominal holdup values
MDs = 0.5
MBs = 0.5
#Nominal flow rates
Ds = 0.5
Bs = 0.5
#Constraint bounds
u_min = array([[0.1], [0.1], [0.1], [0.1], [0.1]])
u_max = array([[10],[VB_max],[10],[1.0],[1.0]])
#State bounds
x_min = zeros((2*NT+2,1))
x_max = ones((2*NT+2,1))

lbx = concatenate((x_min, u_min))
ubx = concatenate((x_max, u_max))
lbg = zeros((2*NT+2,1))
ubg = zeros((2*NT+2,1))
#Problem Dimensions
nx = 2*NT+2     #Number of states (CSTR + Distillation Column)
nu = 5                        #Number of inputs (LT, VB, F, D, B)
nk = 1
tf = 1
h = tf/nk
ns = 0
#Collecting all parameters into a dictionary
params = {}
params['dist'] = {'NC':NC, 'F_0': F_0, 'NT': NT, 'zF': zF,
    'qF': qF, 'NF': NF, 'VB': VB, 'LT': LT, 'F': F, 'alpha': alpha,
    'B': B, 'D': D, 'zF': zF, 'Muw': Muw, 'L0': L0, 'L0b': L0b,
    'qF0': qF0, 'F0': F0, 'taul': taul, 'V0':V0, 'lam':lam, 'MO': MO}
```

```
83 params['cstr'] = {'k1': k1}
   params['price'] = {'pf': pf, 'pV': pV, 'pB': pB, 'pD': pD}
85 params['bounds'] = {'x_min':x_min, 'x_max':x_max, 'u_min': u_min,
       'u_max': u_max, 'lbx': lbx, 'ubx': ubx, 'ubg': ubg, 'lbg': lbg}
87 params['gain'] = {'MDs':MDs,'MBs':MBs,'Ds':Ds,'Bs':Bs,
                    'KcD':KcD, 'KcB':KcB}
89 params['prob'] = {'nx':nx, 'nu':nu, 'nk':nk, 'tf': tf,
                    'h': h, 'ns':ns}
```

```
1  #!/opt/local/bin/python
   # -*- encoding: ascii -*-
3  """
       @purpose: solving optimal control problem
5      @author: Brittany Hall
       @date: 07.10.2017
7      @version: 0.1
       @updates:
9  """
   from casadi import *
11 from numpy import transpose, shape, zeros, savetxt
   import numpy
13 numpy.set_printoptions(threshold=numpy.nan)
   from optProblem import *
15 import time
   from nlp_solve import *
17 from collections import *

19 def solveOpt(optProblem, x0, u0, N, z1, params):

21     x0_measure = z1
       x = zeros((N+1,84))
23     x[0,:] = transpose(x0)
       for k in range(0,N):
25         x[k+1,:] = transpose(x0)

27     J, g, w0, w, lbg, ubg, lbw, ubw, params = optProblem(x, u0,
      x0_measure, N, params)

29     #Solving the NLP
       NLP = {'x': w, 'f': J, 'g': g}
31     options = {}
       tic = time.clock()
33     startnlp = tic
       sol = nlp_solve(NLP, options, w0, lbw, ubw, lbg, ubg)
35     toc = time.clock()
       elapsednlp = toc - tic
37     print "IPOPT solver run time = %f\n" %elapsednlp

39     u = sol['x']
       lam = {}
41     lam['lam_g'] = sol['lam_g']
       lam['lam_x'] = sol['lam_x']
43     objVal = sol['f']

45     return u, lam, lbw, ubw, objVal, params, elapsednlp
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: Solving the optimal control problem
    @author: Brittany Hall
    @date: 07.10.2017
    @version: 0.1
    @updates:
"""
from casadi import Function, MX, SX, vertcat
from collocationSetup import collocationSetup
from ColCSTR_model import ColCSTR_model
from numpy import zeros, ones, array, transpose, matlib, tile,
    reshape, shape, savetxt
import scipy.io as spio
from itPredHorizon import itPredHorizon

def optProblem(x, u, x0_measure, N, params):

    NT = params['dist']['NT']
    Uf = params['dist']['F_0']

    #Modeling the system
    _, state, xdot, inputs = ColCSTR_model(Uf,params)
    f = Function('f', [state, inputs], [xdot])

    #Unpacking parameters
    x_min = params['bounds']['x_min']
    x_max = params['bounds']['x_max']

    #Loading steady state data
    data = spio.loadmat('CstrDistXinit.mat', squeeze_me = True)
    Xinit = data['Xinit']
    xf = Xinit[0:84]
    u_opt = Xinit[84:89]

    #Problem dimensions
    nx = params['prob']['nx']           #Number of states
    nu = params['prob']['nu']           #Number of inputs
    nk = params['prob']['nk']
    tf = params['prob']['tf']
    h =  params['prob']['h']
    ns = params['prob']['ns']

    #Collecting model variables
    u = tile(u,nk)
    model = {'NT': NT, 'f': f, 'xdot_val_rf_ss': xf,
             'x': x, 'u_opt': u_opt, 'u':u}
    params['model'] = model

    #Preparing collocation matrices
    _, C, D, d = collocationSetup()
    params['prob']['d'] = d
```

```
53
    #Collecting collocation variables
55  colloc = {'C': C, 'D': D, 'h': h}
    params['colloc'] = colloc
57
    #Empty NLP
59  w = MX()                    #Decision variables (control + state)
    w0 = []                                          #Initial guess
61  lbw = []                       #Lower bound for decision variable
    ubw = []                       #Upper bound for decision variable
63  g = MX()                                    #Nonlinear constraint
    lbg = []                    #Lower bound for nonlinear constraint
65  ubg = []                    #Upper bound for nonlinear constraint
    J = 0                              #Initialize objective function
67
    #Weight variables
69  delta_t = 1
    alpha = 1
71  beta = 1
    gamma = 1
73  weight = {'delta_t': delta_t, 'alpha': alpha,
        'beta': beta, 'gamma': gamma}
75  params['weight'] = weight

77  #Initial conditions
    X0 = MX.sym('X0', nx)
79  w =  vertcat(w,X0)
    w0 = [i for i in x[0,0:nx]]
81  lbw = [i for i in x_min]
    ubw = [i for i in x_max]
83  g = vertcat(g, X0-x0_measure)
    lbg = params['bounds']['lbg']
85  ubg = params['bounds']['ubg']

87  Xk = X0
    data = spio.loadmat('Qmax.mat', squeeze_me = True)
89  Qmax = data['Qmax']
    params['Qmax'] = Qmax
91
    count = 2                                 #Counter for state variable
93  ssoftc = 0
    for iter in range(0,N):
95      J, g, w0, w, lbg, ubg, lbw, ubw, Xk, params, count, ssoftc =
    itPredHorizon(Xk, w, w0, lbw, ubw, lbg, ubg, g, J, params, iter,
    count, ssoftc, d)

97  return J, g, w0, w, lbg, ubg, lbw, ubw, params
```

```
1 #!/opt/local/bin/python
  # -*- encoding: ascii -*-
3 """
      @purpose: Setting up collocation
5     @author: Brittany Hall (based on Joel Anderson's Matlab script)
      @date: 18.10.2017
7     @version: 0.1
```

```
    @updates:
"""
from casadi import *
from numpy import zeros, convolve, polyval, polyder, polyint, array,
    append
def collocationSetup():
    #Degree of interpolating polynomial
    d = 3
    #Get collocation points
    tau_root = collocation_points(d, 'legendre')
    tau_root = append(0,tau_root)
    #Coefficients of the collocation equation
    C = zeros((d+1, d+1))
    #Coefficients of the continuity equation
    D = zeros((d+1, 1))
    #Coefficients of the quadrature function
    B = zeros((d+1, 1))

    #Construct polynomial basis
    for j in range(0,d+1):
        #Lagrange poly to get poly basis at the colloc point
        coeff = 1
        for r in range(0,d+1):
            if r != j:
                coeff = convolve(coeff, [1, -tau_root[r]])
                coeff = coeff/(tau_root[j]-tau_root[r])
        #Evaluate the polynomial at the final time to get
        # the coefficients of the continuity equation
        D[j] = polyval(coeff,1.0)
        #Evaluate the time derivative of the polynomial at
        #all collocation points to get the coefficients of the
        #continuity equation
        pder = polyder(coeff)
        for r in range(0,d+1):
            C[j,r] = polyval(pder,tau_root[r])
        #Evaluate the integral of the polynomial to get
        #the coefficients of the quadrature function
        pint = polyint(coeff)
        B[j] = polyval(pint, 1.0)
    return B,C,D,d
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: CSTR model (stage NT+1) with a first order reaction (A
    -> B) plus
    nonlinear distillation column model with NT-1 theoretical stages
    including
    a reboiler (stage 1) plus a total condenser (stage NT).
    The model is based on column A in Skogestad and Postlethwaite
    (1996).
    @author: Brittany Hall
    @date: 31.10.2017
    @version: 0.2
    @updates: Fixed bug errors on index assignments
```

```
"""
from casadi import *
from numpy import array, Infinity

def ColCSTR_model(U,params):

    #Unpacking model parameters
    #================Column Dependent Properties=================#
    NC = params['dist']['NC']
    NF = params['dist']['NF']
    NT = params['dist']['NT']
    qF = params['dist']['qF']
    alpha = params['dist']['alpha']
    zF0 = params['dist']['zF']
    Muw = params['dist']['Muw']
    F_0 = U

    #Data for linearized Liquid flow dynamics
    #(does not apply to reboiler and condenser)
    taul = params['dist']['taul']
    F0 = params['dist']['F0']
    qF0 = params['dist']['qF0']
    L0 = params['dist']['L0']
    L0b = L0 + qF0*F0
    lam = params['dist']['lam']
    V0 = params['dist']['V0']
    V0t = V0 + (1-qF0)*F0

    #============================================================#

    #States and Control Inputs
    x = SX.sym('x', NT+1, NC-1)                    #Composition
    M = SX.sym('M',NT+1, 1)                           #Holdup
    states = vertcat(x, M)
    L_T = SX.sym('L_T')                            #Liquid flow
    V_B = SX.sym('V_B')                             #Vapor flow
    F = SX.sym('F')                               #Feed to column
    D = SX.sym('D')                                 #Distillate
    B = SX.sym('B')                                    #Bottom
    inputs = vertcat(L_T,V_B)
    inputs = vertcat(inputs,F)
    inputs = vertcat(inputs,D)
    inputs = vertcat(inputs,B)

    t = SX.sym('t')                                          #Time
    y = SX.sym('y', NT-1, NC-1)                 #Vapor composition
    Li = SX.sym('Li', NT, 1)                  #Liquid flow on stages
    Vi = SX.sym('Vi', NT, 1)                   #Vapor flow on stages

    dMdt = SX.sym('dMdt', NT+1, 1)               #Total Molar holdup
    dMxdt = SX.sym('dMxdt', NT+1, NC-1)       #Component wise holdup
    dxdt = SX.sym('dxdt', NT+1, NC-1)        #Rate of change of comp

    #Vapor flows (assumed constant, no dynamics)
    for i in range(1,NT):
```

```
        Vi[i-1] = V_B
        if i-1 >= NF:
            Vi[i-1] = Vi[i-1] + (1-qF)*F
    Vi[NT-1] = float('Inf')

    #Liquid flows (Wier formula)
    Li[0] = float('Inf')
    for i in range(1,NT):
        if i <= NF-1:
            Li[i] = L0b + (M[i]-Muw)/taul
        else:
            Li[i] = L0 + (M[i]-Muw)/taul

    #Top tray liquid
    Li[NT-1] = L_T

    #Vapor Liquid equilibrium
    for i in range(0,NT-1):
        for j in range(0,NC-1):
            y[i,j] = (x[i,j]*alpha)/(1+(alpha-1)*x[i,j])

    #Partial Reboiler
    dMdt[0] = Li[1] - Vi[0] - B
    for i in range(0,NC-1):
        dMxdt[0,i] = Li[1]*x[1,i] - Vi[0]*y[0,i] - B*x[0,j]

    #Stripping and Enrichment sections
    for i in range(1,NT-1):
        dMdt[i] = Li[i+1] - Li[i] + Vi[i-1]-Vi[i]
        for j in range(0,NC-1):
            dMxdt[i,j] = Li[i+1]*x[i+1,j] - Li[i]*x[i,j] + Vi[i-1]*y[
    i-1,j] -Vi[i]*y[i,j]
    #Correction for feed stage
    dMdt[NF-1] = dMdt[NF-1] + F
    for j in range(0, NC-1):
        dMxdt[NF-1,j] = dMxdt[NF-1, j] + F*x[NT]

    #Total Condenser
    dMdt[NT-1] = Vi[NT-2] - Li[NT-1] - D
    for j in range(0,NC-1):
        dMxdt[NT-1,j] = Vi[NT-2]*y[NT-2,j] - Li[NT-1]*x[NT-1,j]- D*x[
    NT-1,j]

    #CSTR Model
    k1 = params['cstr']['k1']
    dMdt[NT] = F_0 + D - F
    for j in range(0,NC-1):
        dMxdt[NT,j] = F_0*zF0[j] + D*x[NT-1,j] - F*x[NT,j]- k1*M[NT]*
    x[NT,j]

    for i in range(0, NT+1):
        for j in range(0, NC-1):
            dxdt[i,j] = (dMxdt[i,j]-x[i,j]*dMdt[i])/M[i]

    xdot = vertcat(dxdt,dMdt)
```

```
120     return t, states, xdot, inputs
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: Distillation column and CSTR model to be used in
    pathfollowing
              method
    @author: Brittany Hall
    @date: 09.11.2017
    @version: 0.1
    @updates:
"""
from numpy import zeros
from objective import *

def ColCSTR_pf(p):
    prob = {'neq': 0, 'niq': 0, 'cin': 0, 'ceq': 0,
            'dp_in':0, 'dp_eq':0,'hess':0, 'lxp':0,
                'x':0, 'name':0}

    prob['neq'] = 2000          #Number of equality constraints
    prob['niq'] = 0           #Number of inequality constraints
    prob['name'] = 'Distillation Column A + CSTR Model'
    prob['x'] = zeros((2,1))
    prob['obj'] = lambda x,y,p,N: objective(x,y,p,N)

    return prob
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: solving optimal control problem
    @author: Brittany Hall
    @date: 07.10.2017
    @version: 0.1
    @updates:
"""
from casadi import *
from numpy import ones, zeros, multiply, append
import scipy.io as spio

def itPredHorizon(Xk, w, w0, lbw, ubw, lbg, ubg, g, J, params, iter,
    count, ssoftc, d):

    #extracting parameter variables
    nx = params['prob']['nx']           #Number of states
    nu = params['prob']['nu']           #Number of inputs
    nk = params['prob']['nk']
    tf = params['prob']['tf']
    h =  params['prob']['h']
    ns = params['prob']['ns']

    x_min = params['bounds']['x_min']
```

```
    x_max = params['bounds']['x_max']
    u_min = params['bounds']['u_min']
    u_max = params['bounds']['u_max']

    NT = params['model']['NT']
    f = params['model']['f']
    xdot_val_rf_ss = params['model']['xdot_val_rf_ss']
    x = params['model']['x']
    u = params['model']['u']
    u_opt = params['model']['u_opt']

    pf = params['price']['pf']
    pV = params['price']['pV']
    pB = params['price']['pB']
    pD = params['price']['pD']

    F_0 = params['dist']['F_0']

    MDs = params['gain']['MDs']
    MBs = params['gain']['MBs']
    Ds = params['gain']['Ds']
    Bs = params['gain']['Bs']

    C = params['colloc']['C']
    D = params['colloc']['D']
    h = params['colloc']['h']

    delta_t = params['weight']['delta_t']
    alpha = params['weight']['alpha']
    beta = params['weight']['beta']
    gamma = params['weight']['gamma']
    Qmax = params['Qmax']

    for k in range(0,nk):
        #New NLP variable for control
        Uk = MX.sym('U_'+str((iter)*nk+k),nu)
        w = vertcat(w,Uk)
        lbw = append(lbw,u_min)
        ubw = append(ubw,u_max)

        indexU = iter*nk + k
        w0 = append(w0,u[:,indexU])
        Jcontrol = mtimes(transpose(multiply(Qmax[nx:nx+nu],
                                    Uk - u_opt)), (Uk - u_opt))

        #State at collocation points
        SumX1 = 0
        Xkj = {}
        for j in range(0,d):
            Xkj[str(j)] = MX.sym('X_' + str((iter)*nk + k)
                                 +'_'+str(j+1), nx)
            w = vertcat(w, Xkj[str(j)])
            lbw = append(lbw, x_min)
            ubw = append(ubw, x_max)
            w0 = append(w0, x[iter+1,:])
```

```
            count += 1

        #Loop over collocation points
        Xk_end = D[0] * Xk
        for j in range(0,d):
            xp = C[0,j+1] * Xk
            for r in range(0,d):
                xp = xp + C[r+1,j+1] * Xkj[str(r)]

            #Append collocation equations
            fj = f(Xkj[str(j)],Uk)
            g = vertcat(g, h*fj-xp)
            lbg = append(lbg, zeros((nx,1)))
            ubg = append(ubg, zeros((nx,1)))
            #Add contribution to the end state
            Xk_end = Xk_end + D[j+1]*Xkj[str(j)]

        #New NLP variable for state at end of interval
        Xk = MX.sym('X_'+ str((iter)*nk + k), nx)
        w = vertcat(w, Xk)
        lbw = append(lbw, x_min)
        x_maxEnd = ones((2*NT+2,1))
        x_maxEnd[0,0] = 0.1
        x_maxEnd[2*NT+1,0] = 0.7
        ubw = append(ubw, x_maxEnd)
        w0 = append(w0, x[iter+1,:])
        w0 = w0.reshape(len(w0),1)
        count += 1

        #Add equality constraint
        g = vertcat(g, Xk_end-Xk)
        lbg = append(lbg, zeros((nx,1)))
        ubg = append(ubg, zeros((nx,1)))

        Jecon = (pf*F_0 + pV*Uk[1] - pB*Uk[4]
                 - pD*Uk[3]) * delta_t
        Jstate = mtimes(transpose(multiply(Qmax[0:nx],
                (Xk -xdot_val_rf_ss))),(Xk - xdot_val_rf_ss))*delta_t

        J = J + alpha*Jcontrol + gamma*Jstate + beta*Jecon

    return J, g, w0, w, lbg, ubg, lbw, ubw, Xk, params, count, ssoftc
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: Solving optimal control problem
    @author: Brittany Hall
    @date: 10.11.2017
    @version: 0.1
    @updates:
    """
from casadi import *
from numpy import ones, zeros, multiply, append
import scipy.io as spio
```

```
13
def itPredHorizon_pf(Xk, V, cons, obj, params, iter, ssoftc):
15
    #Extracting parameters
17  NT = params['dist']['NT']
    sf = params['model']['sf']
19  xdot_val_rf_ss = params['model']['xdot_val_rf_ss']
    u_opt = params['model']['u_opt']
21
    pf = params['price']['pf']
23  pV = params['price']['pV']
    pB = params['price']['pB']
25  pD = params['price']['pD']
    F_0 = params['dist']['F_0']
27
    C = params['colloc']['C']
29  D = params['colloc']['D']
    h = params['colloc']['h']
31
    delta_t = params['weight']['delta_time']
33  Qmax = params['Qmax']

35  nx = params['prob']['nx']
    nu = params['prob']['nu']
37  nk = params['prob']['nk']
    d = params['prob']['d']
39  ns = params['prob']['ns']

41  count = 0
    for k in range(0,nk):
43      #New NLP variable for control
        Uk = MX.sym('U_'+str((iter)*nk+k), nu)
45      V = vertcat(V,Uk)
        Jcontrol = mtimes(transpose(multiply(Qmax[nx:nx+nu],
47                                  Uk - u_opt)), (Uk - u_opt))

49      #State at collocation points
        SumX1 = 0
51      Xkj = {}
        for j in range(0,d):
53          Xkj[str(j)] = MX.sym('X_' + str((iter)*nk + k)
                                 +'_'+str(j+1), nx)
55          V = vertcat(V,Xkj[str(j)])
            count += 1
57
        #Loop over collocation points
59      Xk_end = D[0] * Xk
        for j in range(0,d):
61          xp = C[0,j+1] * Xk
            for r in range(0,d):
63              xp = xp + C[r+1,j+1] * Xkj[str(r)]
            #Append collocation equations
65          fj = sf(Xkj[str(j)],Uk)
            cons = vertcat(cons, h*fj - xp)
67
```

```
            #Add contribution to the end state
            Xk_end = Xk_end + D[j+1]*Xkj[str(j)]

        #New NLP variable for state at end of interval
        Xk = MX.sym('X_'+ str((iter)*nk + k), nx)
        V = vertcat(V, Xk)
        #Add equality constraint
        cons = vertcat(cons, Xk_end-Xk)

        Jecon = (pf*F_0 + pV*Uk[1] - pB*Uk[4] -
                    pD*Uk[3])*delta_t
        Jstate =  mtimes(transpose(multiply(Qmax[0:nx],
                (Xk -xdot_val_rf_ss))),(Xk - xdot_val_rf_ss))*delta_t

        #Compute rotate cost function
        fm = sf(Xk, Uk)
        alpha = 1
        beta = 1
        gamma = 1

        obj = obj + alpha*Jcontrol + gamma*Jstate + beta*Jecon

    return obj, cons, V, Xk, params, ssoftc
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: NLP solver
    @author: Brittany Hall
    @date: 18.09.2017
    @version: 0.1
    @updates:
"""
from casadi import *

def nlp_solve(problem, options, x0, lbx, ubx, lbg, ubg):
    """
    NLP solver for initial conditions to path-following algorithm
    """
    #Formulating NLP to solve
    solver = nlpsol('solver', 'ipopt', problem, options)
    sol = solver(x0=x0, lbx=lbx, ubx=ubx, lbg=lbg, ubg=ubg)
    return sol
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: Used to reshape the data to make it easier for plotting
    @author: Brittany Hall
    @date: 08.10.2017
    @version: 0.1
    @updates:
"""
from numpy import array, zeros, reshape, delete, size
from casadi import *
```

```
13 def plotStates(data, lb, ub, N, params):
    #unpacking params
15  nu = params['prob']['nu']
    nx = params['prob']['nx']
17  ns = params['prob']['ns']
    nk = params['prob']['nk']
19  d = params['prob']['d']

21  #Optimized initial state
    x0_opt = data[0:nx]
23  index = range(0,nx)
    data = delete(data,index)
25  data = reshape(data, ((nu + (nx+ns)*d + (nx+ns)),N*nk))
    u_nlp_opt = data[0:nu,0:N*nk]
27  data = data[nu:,:]

29  lb0 = lb[0:nx+ns]
    lb = delete(lb,range(0,nx))
31  lb = reshape(lb,(nu+(nx+ns)*d+(nx+ns),N*nk))
    lbU = lb[0:nu, 0:N*nk]
33  lb = lb[nu:,:]
    ub0 = ub[0:nx+ns]
35  ub = ub[nx:]
    ub = reshape(ub, (nu+(nx+ns)*d+(nx+ns),N*nk))
37  ubU = ub[0:nu,0:N*nk]
    ub = ub[nu:,:]
39
    #Preparing matrix for plotting
41  nState = (nx+ns) + N*nk*(d+1)*(nx+ns)
    nPoint = nState/(nx+ns)
43  plotState = zeros((nx+ns,nPoint))
    for i in range(0,nx):
45      plotState[i,0] = x0_opt[i]
    plotLb = zeros((nx+ns, nPoint))
47  plotLb[:,0] = lb0
    plotUb = zeros((nx+ns,nPoint))
49  plotUb[:,0] = ub0

51  #Extract states from each colloc point at each time horizon
    sInd = 1 #initial index row
53  for i in range(0,N*nk-1):
        temp = data[:,i]
55      numCol = size(temp,axis=0)
        numRow = numCol/(nx+ns)
57      temp = reshape(temp,(nx+ns,numRow))
        plotState[:,sInd:(numRow+sInd)] = temp
59      tempLb = lb[:,i]
        tempLb = reshape(tempLb, (nx+ns,numRow))
61      plotLb[:,sInd:(numRow+sInd)] = tempLb
        tempUb = ub[:,i]
63      tempUb = reshape(tempUb, (nx+ns,numRow))
        plotUb[:,sInd:(numRow+sInd)] = tempUb
65
        sInd += numRow
```

```
67
    return u_nlp_opt, plotState
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: Predictor corrector
    @author: Brittany Hall
    @date: 08.10.2017
    @version: 0.1
    @updates:
"""
from casadi import *
from qp_solve import *
from numpy import zeros, shape

def predictor_corrector(problem, p_init, p_final, x_init, y_init,
    delta_t, lb_init, ub_init, verbose_level, N):

    p = p_init
    pp = SX.sym('pp')
    theprob = lambda p: problem(pp)
    prob = theprob(p)
    t = 0
    alpha_1 = 0.5
    iter = 1 #iteration number
    elapsedqp = 0
    numX = shape(x_init)[0]
    x0 = zeros(numX)

    if verbose_level:
        print('Solving problem %s \n', prob['name'])
        print('Iteration delta_t   t   Success\n')

    p_0 = p_init
    while t<=1:
        #Calculating the step
        tk = t + delta_t
        p_t = (1-tk)*p_0 + tk*p_final
        step = p_t - p_init

        #Updating bound constraints
        if lb_init.any():
            lb = lb_init-x_init
            ub = ub_init-x_init
        elif not lp_init:
            lb = array([])
            ub = array([])

        #Solve QP problem
        y, qp_val, qp_exit, lam_qpopt, mu_qpopt, qptime = qp_solve(
    prob, p, x_init, y_init, step, lb, ub, N, x0, lb_init, ub_init)
        elapsedqp += qptime

        if qp_exit == 'infeasible':#QP infeasible
```

```
            print "
    ----------------------------------------------------------------------\n"
52          print "The QP is infeasible"
            delta_t = alpha_1*t                         #shorten step
54          t = t - delta_t

56          #Print out iteration number and failure

58          success = 0
            if verbose_level:
60              print '%f    %f  %f   %d' %(iter, delta_t, t, success
    )
            iter += 1
62      elif qp_exit == 'feasible':#QP feasible
            #Update states, multipliers, parameter and time step
64          x_init = x_init + y
            y_init['lam_x'] = y_init['lam_x'] + mu_qpopt
66          t = t + delta_t
            p_init = p_t
68
            #Print out iteration number and success
70          print "
    ----------------------------------------------------------------------\n"
            print "The QP is feasible"
72          print "iteration number: %d\n" %iter
            success = 1
74          if verbose_level:
                print '%f    %f  %f   %d' %(iter, delta_t, t, success
    )
76
            #Fix Steplength
78          print "delta_t: %f\n" %delta_t
            print "t: %f\n" %t
80          iter += 1

82      if (1-t) <= 1e-5:
            break
84
    return x_init, y_init, elapsedqp
```

```
1 #!/opt/local/bin/python
  # -*- encoding: ascii -*-
3 """
      @purpose: Solving a QP
5     @author: Brittany Hall
      @date: 08.10.2017
7     @version: 0.1
      @updates:
9 """
  from casadi import *
11 from numpy import where, multiply, shape, all, isnan, array, vstack,
      hstack, ones, reshape, zeros
  from params import params
13 from objective import objective
  import time
```

```
15 import osqp
   import scipy.sparse as sparse
17
   def qp_solve(prob, p_init, x_init, y_init, step, lb, ub, N, x0,
       lb_init, ub_init):
19     """
       QP solver for path-following algorithm
21     inputs: prob - problem description
               p_init - initial parameters
23             x_init - initial primal variable
               y_init - initial dual variable
25             step - step to be taken (in p)
               lb -
27             ub -
               N - iteration number
29             x0 - initial guess for primal variable
               lb_init - lower bounds
31             ub_init - upper bounds
       outputs: y - solution primal variable
33             qp_val - objective function value
               qp_exit - return status of QP solver
35
       """
37
       ##================Importing problem to be solved
       ======================##
39
       neq = prob['neq']                                #Number of equality
       constraints
41     niq = prob['niq']                               #Number of inequality
       constraints
       name = prob['name']                                              #Name
       of problem
43
       _, g, H, Lxp, cst, _, _, Jeq, dpe, _ = objective(x_init, y_init,
       p_init, N, params)
45
       #Setting up QP
47     f = mtimes(Lxp,step) + g

49     #Constraints
       ceq = cst
51     Aeq = Jeq
       beq = mtimes(dpe,step) + ceq
53
       #Check Lagrange multipliers from bound constraints
55     lamC = fabs(y_init['lam_x'])

57     #Setting limits to determine if constraint is active
       BAC = where(lamC >= 1e-3)
59     BAC = BAC[0]

61     #Finding active constraints
       numBAC = len(BAC)
63     for i in range(0,numBAC):
```

```
        #Placing strongly active constraint on boundary
65      indB = BAC[i]
        #Keeping upper bound on boundary
67      ub[indB] = 0
        lb[indB] = 0
69
    ##==============Solving the QP using OSQP
    ================================##
71  #Setting up proper format for solver
    nx = params['prob']['nx']
73  nu = params['prob']['nu']
    Ax = sparse.eye(shape(Aeq)[1])
75  leq = reshape(beq, shape(beq)[0])
    lb = reshape(lb, shape(lb)[0])
77  ueq = reshape(beq, shape(beq)[0])
    ub = reshape(ub, shape(ub)[0])
79  l = hstack([leq, lb])
    u = hstack([ueq, ub])
81  A = sparse.vstack([Aeq, Ax])
    A = sparse.csc_matrix(A)
83  P = sparse.csc_matrix(H)
    q = array(f)
85
    #Starting solver
87  prob = osqp.OSQP()
    #ftol = 1e-7
89  #xtol = 1e-7
    prob.setup(P, q, A, l, u, eps_rel = 1e-7)
91  results = prob.solve()
    if results.info.status != 'solved':
93      print "OSQP did not solve the problem!\n"
        qp_exit = 'infeasible'
95  else:
        print "OSQP solver runtime: %f\n" %results.info.solve_time
97      qp_exit = 'feasible'
    #Results
99  y = results.x
    lam_qpopt = results.y[0:shape(Aeq)[0]]
101 mu_qpopt = results.y[shape(Aeq)[0]:]
    qp_val = results.info.obj_val
103 elapsedqp = results.info.solve_time

105

    ##==============Solving QP using QPOASES/GUROBI
    ========================##
107 #   qp = {}
    #   qp['h'] = H.sparsity()
109 #   qp['a'] = Aeq.sparsity()
    #   #optimize = conic('optimize','qpoases',qp,{'sparse':True})
111 #   startqp = time.time()
    #   optimal = optimize(h=H, g=f, a=Aeq, lba=beq, uba=beq, lbx=lb,
    ubx=ub, x0=x0)
113 #   elapsedqp = time.time()-startqp
    #   x_qpopt = optimal['x']                                    #
    primal solution
```

```
115 #     y = x_qpopt
    #     qp_val = optimal['cost']                                        #
    optimal cost
117 #     lam_qpopt = optimal['lam_a']                    #dual solution-
    linear bounds
    #     mu_qpopt = optimal['lam_x']                     #dual solution-
    simple bounds
119 #
    #     if isnan(array(x_qpopt[0])):
121 #         qp_exit = 'infeasible'
    #     else:
123 #         qp_exit = 'optimal'
    #     print qp_val
125 #     print y
    #     raw_input()
127      ##=====================Solving QP using IPOPT
    ============================##
    #     w = MX()
129 #     nx = params['prob']['nx']
    #     nu = params['prob']['nu']           #Number of states
131 #     nk = params['prob']['nk']
    #     X0 = MX.sym('X0', nx)
133 #     w =  vertcat(w,X0)
    #     _, C, D, d = collocationSetup()
135 #     for iter in range(0,N):
    #         for k in range(0,nk):
137 #             Xkj = {}
    #             for j in range(0,d):
139 #                 Xkj[str(j)] = MX.sym('X_' + str((iter)*nk + k)+'_'+
    str(j+1), nx)
    #             #New NLP variable for control
141 #             Uk = MX.sym('U_'+str((iter)*nk+k),nu)
    #             w = vertcat(w,Uk)
143 #             for j in range(0,d):
    #                 w = vertcat(w, Xkj[str(j)])
145 #                 Xk = MX.sym('X_'+ str((iter)*nk + k), nx)
    #             w = vertcat(w, Xk)
147 #
    #     F = 0.5*mtimes(w.T,mtimes(H,w)) + mtimes(f.T,w)
149 #     startqp = time.time()
    #     qp = {'x':w , 'f':F , 'g': mtimes(Aeq,w)}
151 #     optimize = nlpsol('optimize', 'ipopt', qp)
    #     sol = optimize(lbx=lb, ubx=ub, lbg = beq, ubg=beq)
153 #     elapsedqp = time.time()-startqp
    #     y = sol['x']
155 #     qp_val = sol['f']
    #     lam_x = sol['lam_x']
157 #     lam_g = sol['lam_g']

159
         return y, qp_val, qp_exit, lam_qpopt, mu_qpopt, elapsedqp
```

```
#!/opt/local/bin/python
2 # -*- encoding: ascii -*-
"""
```

```
    @purpose: Computing objective function values
    @author: Brittany Hall
    @date: 11.10.2017
    @version: 0.1
    @updates:
"""
from numpy import size, transpose, multiply
import scipy.io as spio
from itPredHorizon import *
from params import *

def compObjFn(uOpt,xActual):

    #prices
    pf = params['price']['pf']
    pV = params['price']['pV']
    pB = params['price']['pB']
    pD = params['price']['pD']

    #Setpoints
    F_0 = params['dist']['F_0']

    #Steady-state values
    data = spio.loadmat('CstrDistXinit.mat', squeeze_me=True)
    Xinit = data['Xinit']

    xs = Xinit[0:84]
    us = Xinit[84:]
    nx = size(xs, axis = 0)
    nu = size(us, axis = 0)

    #Loading in objective function weights
    data = spio.loadmat('Q.mat', squeeze_me = True)
    Qmax = data['Q']
    c1 = -0.05 #noise
    lss = -0.256905910000000 + c1 #ss obj fxn value

    #Defining objective function
    Jecon = pf*F_0 + pV*uOpt[1] - pB*uOpt[4] - pD*uOpt[3]
    Jcontrol = mtimes(transpose(multiply(Qmax[nx:nx+nu],
                                uOpt - us)), (uOpt - us))
    Jstate = mtimes(transpose(multiply(Qmax[0:nx],
                        (xActual -xs))),(xActual - xs))

    J = Jecon + Jcontrol + Jstate -lss

    print('-----------------------------------------------\n')
    print("Jecon: %f,\n Jcontrol: %f, \n Jstate: %f, \n"
                %(Jecon, Jcontrol, Jstate))
    Jobj = {}
    Jobj['reg'] = J
    Jobj['econ'] = Jecon

    return Jobj
```

```
#!/opt/local/bin/python
# -*- encoding: ascii -*-
"""
    @purpose: Plots the results (iNMPC vs pfNMPC and MATLAB vs Python)
    @author: Brittany Hall
    @date: 08.11.2017
    @version: 0.1
    @updates:
"""
import matplotlib.pyplot as plt
import scipy.io as spio
from numpy import reshape, append, hstack, linspace, ones, transpose, vstack
from params import params

def plotting(u0, xmeasure, MPCit, T):

    NT = params['dist']['NT']
    NF = params['dist']['NF']

    ##================Loading in Results
    ================================##

    #Steady state data
    data = spio.loadmat('CstrDistXinit.mat', squeeze_me = True, struct_as_record=False)
    Xinit = data['Xinit']
    xf = Xinit[0:84]
    u_opt = Xinit[84:]


    #Loading in iNMPC Python results
    data_ideal = spio.loadmat('iNMPC_Python.mat',squeeze_me = False)
    uAll = data_ideal['ideal']['uAll']
    uAll = uAll[0,0]
    xmeasureAll = data_ideal['ideal']['xmeasureAll']
    xmeasureAll = xmeasureAll[0,0]
    ObjReg = data_ideal['ideal']['ObjReg']
    ObjReg = transpose(ObjReg[0,0])
    ObjEcon = data_ideal['ideal']['ObjEcon']
    ObjEcon = transpose(ObjEcon[0,0])
    T = data_ideal['ideal']['T']
    mpcit = data_ideal['ideal']['mpciterations']

    #Loading in pfNMPC Python results
    data_pf = spio.loadmat('pfNMPC_Python.mat',squeeze_me = False)
    uAll_pf = data_pf['pf']['uAll']
    uAll_pf = uAll_pf[0,0]
    xmeasureAll_pf = data_pf['pf']['xmeasureAll']
    xmeasureAll_pf = xmeasureAll_pf[0,0]
    ObjReg_pf = data_pf['pf']['ObjReg']
    ObjReg_pf = transpose(ObjReg_pf[0,0])
    ObjEcon_pf = data_pf['pf']['ObjEcon']
```

```
51      ObjEcon_pf = transpose(ObjEcon_pf[0,0])
        T_pf = data_pf['pf']['T']
53      mpcit_pf = data_pf['pf']['mpciterations']

55      #Loading in iNMPC MATLAB results
        data_iMat = spio.loadmat('iNMPC_MATLAB.mat',squeeze_me = False)
57      xmeasureAll_mat = data_iMat['xmeasureAll']
        uAll_mat = data_iMat['uAll']
59      ObjReg_mat = transpose(data_iMat['ObjReg'])
        ObjEcon_mat = transpose(data_iMat['ObjEcon'])
61
        #Loading in pfNMPC MATLAB results
63 #     data_pfmat = spio.loadmat('pfNMPC_MATLAB.mat', squeeze_me = True
       )
   #     uAll_pfmat = data_pfmat['pfNMPC']['uAll']
65 #     xmeasureAll_pfmat = data_pfmat['pfNMPC']['xmeasureAll']
   #     ObjReg_pfmat = data_pfmat['pfNMPC']['ObjReg']
67 #     ObjEcon_pfmat = data_pfmat['pfNMPC']['ObjEcon']

69      #Reshaping values
        nu = u0.shape[0]
71      uAll= uAll.reshape(nu, MPCit,order='F').copy()
        uAll_mat = uAll_mat.reshape(nu, MPCit,order='F').copy()
73      uAll_pf = uAll_pf.reshape(nu, MPCit,order='F').copy()
        #uAll_pfmat = uAll_pfmat.reshape(nu, MPCit,order='F').copy()
75
        #Add initial control
77      u0_0 = reshape(u0[:,0],(nu,1))
        uAll = hstack((u0_0, uAll))
79      uAll_mat = hstack((u0_0, uAll_mat))
        uAll_pf = hstack((u0_0, uAll_pf))
81      # uAll_pfmat = hstack((u0_0, uAll_pfmat))

83      #Add initial states
        xmeasure = reshape(xmeasure,(xmeasure.shape[0],1))
85      xmeasureAll = hstack((xmeasure,xmeasureAll))
        xmeasureAll_pf = hstack(xmeasure,xmeasureAll_pf)
87      # xmeasureAll_pfmat = hstack(xmeasure,xmeasureAll_pfmat)

89      x = linspace(0, MPCit, MPCit/T)
        xi = append(0,x)
91
        ##================================Plotting
       ==============================##
93      #Figure: Objective function comparison
        plt.plot(x,ObjReg, 'g', x, ObjEcon, 'b', x, ObjReg_mat, 'ro', x,
       ObjEcon_mat, 'k*', x, ObjReg_pf,'bo', x, ObjEcon_pf, 'r-')
95      plt.title('Objective function')
        plt.xlabel('Number of MPC iteration [-]')
97      plt.ylabel('Objective function [-]')
        plt.legend(['iNMPC:Full-Python', 'iNMPC:Economic-Python', 'iNMPC:
       Full-Matlab', 'iNMPC:Economic-Matlab', 'pfNMPC:Full-Python', '
       pfNMPC:Economic-Python'])
99      plt.show()
```

```
101     #Figure: Concentration at stage 1 (reboiler)
        plt.plot(xi,xf[0]*ones(MPCit+1),'r', xi, xmeasureAll[0,],'g', xi
        [0:150], xmeasureAll_mat[0,],'bo', xi[0:150], xmeasureAll_pf[0,],
        'k*')
103     plt.ylabel('Concentration [-]')
        plt.xlabel('Time [min]')
105     plt.title('Distillation: Bottom Composition')
        plt.legend(['Steady-state','iNMPC-Python', 'iNMPC-Matlab', '
        pfNMPC-Python'])
107     plt.show()

109     #Figure: Concentration at feed stage
        plt.plot(xi,xf[NF]*ones(MPCit+1),'r',xi,xmeasureAll[NF,],'g', xi
        [0:150], xmeasureAll_mat[NF,],'bo', xi[0:150], xmeasureAll_pf[NF
        ,],'k*')
111     plt.ylabel('Concentration [-]')
        plt.xlabel('Time [min]')
113     plt.title('Distillation: Feed Composition')
        plt.legend(['Steady-state','iNMPC-Python','iNMPC-Matlab', 'pfNMPC
        -Python'])
115     plt.show()

117     #Figure: Concentration at stage NT (top)
        plt.plot(xi,xf[NT]*ones(MPCit+1),'r',xi,xmeasureAll[NT,],'g',xi
        [0:150],xmeasureAll_mat[NT,],'bo', xi[0:150], xmeasureAll_pf[NT,],
        'k*')
119     plt.ylabel('Concentration [-]')
        plt.xlabel('Time [min]')
121     plt.title('Distillation: Top Composition')
        plt.legend(['Steady-state','iNMPC-Python', 'iNMPC-Matlab', '
        pfNMPC-Python'])
123     plt.show()

125     #Figure: Concentration in CSTR
        plt.plot(xi,xf[NT+1]*ones(MPCit+1),'r',xi,xmeasureAll[NT+1,],'g',
        xi[0:150], xmeasureAll_mat[NT+1,],'bo', xi[0:150], xmeasureAll_pf
        [NT+1,],'k*')
127     plt.ylabel('Concentration [-]')
        plt.xlabel('Time [min]')
129     plt.title('CSTR: Concentration')
        plt.legend(['Steady-state','iNMPC-Python', 'iNMPC-Matlab', '
        pfNMPC-Python'])
131     plt.show()

133     #Figure: Holdup in CSTR
        plt.plot(xi,xf[2*NT-1]*ones(MPCit+1),'r',xi,xmeasureAll[2*NT-1,],
        'g', xi[0:150], xmeasureAll_mat[2*NT-1,],'bo', xi[0:150],
        xmeasureAll_pf[2*NT-1,], 'k*')
135     plt.ylabel('Holdup [-]')
        plt.xlabel('Time [min]')
137     plt.title('CSTR: Holdup')
        plt.legend(['Steady-state','iNMPC-Python', 'iNMPC-Matlab', '
        pfNMPC-Python'])
139     plt.show()
```

```
141     #Figure: u[0] LT control input
        plt.plot(xi, u_opt[0]*ones(MPCit+1), 'r', xi, uAll[0,], 'g', xi,
        uAll_mat[0,], 'bo', xi, uAll_pf[0,], 'k*')
143     plt.ylabel('LT [m^3/min]')
        plt.xlabel('Time [min]')
145     plt.title('Control input for LT')
        plt.legend(['Steady-state','iNMPC-Python', 'iNMPC-Matlab', '
        pfNMPC-Python'])
147     plt.show()

149     #Figure: u[1] VB control input
        plt.plot(xi, u_opt[1]*ones(MPCit+1), 'r', xi, uAll[1,], 'g',xi,
        uAll_mat[1,], 'bo', xi, uAll_pf[1,], 'k*')
151     plt.ylabel('VB [m^3/min]')
        plt.xlabel('Time [min]')
153     plt.title('Control input for VB')
        plt.legend(['Steady-state', 'iNMPC-Python', 'iNMPC-Matlab', '
        pfNMPC-Python'])
155     plt.show()

157     #Figure: u[2] F control input
        plt.plot(xi, u_opt[2]*ones(MPCit+1), 'r', xi, uAll[2,], 'g', xi,
        uAll_mat[2,], 'bo', xi, uAll_pf[2,], 'k*')
159     plt.ylabel('F [kmol/min]')
        plt.xlabel('Time [min]')
161     plt.title('Control input for F')
        plt.legend(['Steady-state', 'iNMPC-Python', 'iNMPC-Matlab', '
        pfNMPC-Python'])
163     plt.show()

165     #Figure: u[3] D control input
        plt.plot(xi, u_opt[3]*ones(MPCit+1), 'r', xi, uAll[3,], 'g',xi,
        uAll_mat[3,], 'bo', xi, uAll_pf[3,], 'k*')
167     plt.ylabel('D [kmol/min]')
        plt.xlabel('Time [min]')
169     plt.title('Control input for D')
        plt.legend(['Steady-state', 'iNMPC-Python', 'iNMPC-Matlab', '
        pfNMPC-Python'])
171     plt.show()

173     #Figure: u[4] B control input
        plt.plot(xi, u_opt[4]*ones(MPCit+1), 'r', xi, uAll[4,], 'g', xi,
        uAll_mat[4,], 'bo', xi, uAll_pf[4,], 'k*')
175     plt.ylabel('B [kmol/min]')
        plt.xlabel('Time [min]')
177     plt.title('Control input for B')
        plt.legend(['Steady-state', 'iNMPC-Python', 'iNMPC-Matlab', '
        pfNMPC-Python'])
179     plt.show()
```