Semester Project Report

Embedded Real Time Optimization of a Fuel Cell System

Author :
Thammisetty Devakumar

Supervisors :
Petr Listov, Tafarel De Avilla
Professors :
Colin Jones, Jan Van Herle

January 20, 2020

Acknowledgement

I would like to thank everyone who helped me throughout this semester project. First and foremost, I would like to thank *Petr Listov* for being an exceptional assistant, for giving me an opportunity to work on numerical optimization, for always being around to answer my queries or doubts and constantly guiding me throughout this challenging project.

I would also like to thank *Tafarel De Avilla* for introducing me to the SOFC problem and providing valuable resources regarding the SOFC real time optimization including MATLAB code

Finally, I would like to thank Prof. Colin Jones, for supervising the project and for his valuable guidance.

Contents

List of Figures

1 Introduction

Scope of the project includes familiarizing with the numerical methods and the software[1] for embedded optimization & optimal control developed at the Automatic Control laboratory. Subsequently, a real time optimization (RTO) of a Solid Oxide Fuel Cell system (SOFC) is formulated and solved using the software. During this process, a sub-module for the preconditioning of the Quadratic Problems (QP) is added to the software. Benchmarking of the QP solver, with and without QP preconditioning is carried out using random QPs of various dimensions. SQP solver is updated so that the non-linear optimization problem can be solved with QP preconditioning in optimized manner. While solving the SOFC system dynamics, it is found that the dynamics are highly complex and time consuming to be re-programmed in C++ within the duration of the project. Hence, a simplified sub-problem of SOFC which includes the Reformer and SOFC dynamics is programmed in C++. Further, a simplified SOFC problem which includes 6 variables, 2 inequality constraints, 6 box constraints in addition to complex non-linear system dynamics for the reformer (1 equality constraint) is solved using the non-linear program solver included in the PolyMPC package[1].

2 Nonlinear optimization with constraints

Three classes of methods are presented in literature[2] for solving constrained non-linear optimization problems.

1. *Penalty and augmented Lagrangian methods* : Constrained non-linear problem is translated into a sequence of unconstrained problems using penalty or Lagrangian functions which are then solved using unconstrained solvers such as Newtons method.
2. *Interior point or Barrier methods* : These methods introduce barrier functions in the objective function to ensure constraints satisfaction and drive the solution towards optimal value when the barrier parameter is updated iteratively.
3. *Sequential Quadratic Programming (SQP)* : Non-linear constrained problem is modelled as a sequence of Quadratic Programming(QP) problems by linearizing the original problem. Solution of QPs define the search direction at every iteration.

The embedded optimization and optimal control software (PolyMPC) developed at Automatic Laboratory implements SQP for solving non-linear optimization problems. Hence, SQP method and its implementation in PolyMPC are discussed briefly in the next section.

3 SQP solver: PolyMPC

A generic non-linear optimization problem formulation is given in equations (8). The formulation includes objective function(Φ), decision variables($\mathbf{x}$) and set of constraints (H, G). Further, the constraints are of three types including equality(H), inequality(G) and box constraints.

Nonlinear optimization problem

$$\begin{aligned}
&\min_{\mathbf{x}} && \Phi(\mathbf{x}) && && \textit{Objective} && \text{(1a)}\\
&\text{s.t} && H_e(\mathbf{x}) = 0, && e = 1, \dots, m && \textit{Equality Constraints} && \text{(1b)}\\
& && G_i(\mathbf{x}) \leq 0, && i = 1, \dots, p && \textit{Inequality Constraints} && \text{(1c)}\\
& && x_b^L \leq x_b \leq x_b^U, && b = 1, \dots, n && \textit{Box Constraints} && \text{(1d)}
\end{aligned}$$

SQP method solves the problem (Equations 8) iteratively by first linearizing the problem to formulate an appropriate QP whose solution gives optimal direction of descent. The linearized problem (QP) formulation where the original problem is linearized around x_k with descent direction given by (Δx) is given in Equations (2).

SQP : Linearized formulation

$$\begin{aligned}
&\min_{\Delta x} && \Delta x^T P_k \Delta x + q_k^T \Delta x && \textit{Objective} && \text{(2a)}\\
&\text{s.t} && H(x_k) + A_e \Delta x = 0, && \textit{Equality Constraints} && \text{(2b)}\\
& && G(x_k) + A_i \Delta x \leq 0, && \textit{Inequality Constraints} && \text{(2c)}\\
& && x^L \leq x_k + \Delta x \leq x^U, && \textit{Box Constraints} && \text{(2d)}
\end{aligned}$$

Where A_e, A_i, P_k, q_k are linearized parameters around x_k, given by Equations (3).

$$\begin{aligned}
P &= \nabla^2_{xx}\mathcal{L}(x_k, \lambda_k) && \textit{Hessian of augmented Lagrangian} && \text{(3a)}\\
q &= \nabla\Phi(x_k) && \textit{Gradient of cost function} && \text{(3b)}\\
A_e &= [\nabla H_1(x_k), \dots, \nabla H_m(x_k)]^T && \textit{Jacobian of equality constraints} && \text{(3c)}\\
A_i &= [\nabla G_1(x_k), \dots, \nabla G_p(x_k)]^T && \textit{Jacobian of inequality constraints} && \text{(3d)}
\end{aligned}$$

The Hessian(P) in the formulation (2) is not evaluated directly but approximated using Broyden-Fletcher-Goldfarb-Shanno (BFGS) method which uses the step size and gradient information in successive iterations to produce a positive definite hessian so that the QP formulation in Equations (2) can be solved using QP solvers.

SQP algorithm

The optimization problem formulated in Equations (8) is solved iteratively using the linearized problem formulation given in equation (2) using the SQP algorithm 1, given below.

Algorithm 1 SQP algorithm

```
1: procedure SQP($x_0, \lambda_0$)
2:     while Convergence criterion not reached do
3:         Linearize the optimization problem around $(x_k, \lambda_k)$        ▷ Formulate QP (Equation 2)
4:         $(\Delta x_k, \hat{\lambda}) \leftarrow$ Solve QP problem(2) using selected QP solver    ▷ QP Solver (Algorithm 2)
5:         $\alpha \leftarrow$ Linesearch($\Delta x_k$)        ▷ Linesearch algorithm [3]
6:         $x^{k+1} \leftarrow x^k + \alpha \Delta x_k$
7:         $\lambda^{k+1} \leftarrow \lambda_k + \alpha(\hat{\lambda} - \lambda_k)$
8:     return $x^{k+1}$
```

PolyMPC implementes the SQP algorithm given in Algorithm 1. Further, following improvements in the SQP implementation are suggested in Master thesis[3].

- Memory optimization in Hessian initialization to store only lower triangular entries.
- Handling indefinite Hessian during the BFGS update.
- Better termination criteria (ϵ)
- Handling inconsistent linearization of optimization problem

4 QP solver : PolyMPC

SQP algorithm in PolyMPC is an independent module which formulates a QP at each iteration, calls the QP solver. For solving the QP problem, an independent module based on Alternating Direction Method of Multipliers(ADMM) is implemented in PolyMPC[3][4]. Brief outline of generic QP problem formulation and its solution method using ADMM based QP solver is given in this section.

QP formulation

$$\min_{x} \quad x^T P x + q^T x \qquad \textit{Objective} \tag{4a}$$

$$l \leq Ax \leq u, \qquad \textit{Constraints} \tag{4b}$$

In the QP formulation given in equations (4), equality constrains are represented using $l = u$. Further, box constraints are represented by selecting appropriate A.

Solution procedure using ADMM based QP solver

ADMM based QP solver reformulates the given problem (Equations 4) by introducing additional auxiliary variable aliases ($\tilde{x}$, $\tilde{z}$) to (x, z), so that the optimization is performed alternatively between objective function and constraints. For detailed description of the solution procedure one can refer OSQP paper[4]. A brief outline of the various steps in the ADMM based qp solver are given in Algorithm 2, given below for reference.

Algorithm 2 ADMM based QP algorithm[5]

```
1: procedure QP($x^0, z^0, \lambda^0$)
2:     $(\rho, \sigma, \alpha) \leftarrow$ Select parameters $\rho > 0, \sigma > 0$ and $\alpha \in (0, 2)$
3:     $M \leftarrow \begin{bmatrix} P + \sigma I & A^T \\ A & -\rho^{-1} I \end{bmatrix}$        ▷ Construct KKT matrix
4:     while Convergence criterion not reached do
5:         $(\tilde{x}^{k+1}, \tilde{\nu}^{k+1}) \leftarrow$ Solve linear system $M \begin{bmatrix} \tilde{x}^{k+1} \\ \tilde{\nu}^{k+1} \end{bmatrix} = \begin{bmatrix} \sigma x^k - q \\ z^k - \rho^{-1} y^k \end{bmatrix}$
6:         $\tilde{z}^{k+1} \leftarrow z^k + \rho^{-1}(\tilde{\nu}^{k+1} - y^k)$
7:         $x^{k+1} \leftarrow x^k + \alpha(\tilde{x}^{k+1} - x^k)$
8:         $z^{k+1} \leftarrow \Pi\left(z^k + \alpha(\tilde{z}^{k+1} - z^k) + \rho^{-1} y^k\right)$
9:         $y^{k+1} \leftarrow y^k + \rho\left(z^k + \alpha(\tilde{z}^{k+1} - z^k) - z^{k+1}\right)$
10:    return $x^{k+1}, y^{k+1}$
```

In the existing implementation of ADMM based QP solver in PolyMPC, following improvements are suggested [3].

- Preconditioning KKT matrix in order to speed up the convergence
- Infeasibility detection and handling

As part of the project, preconditioning of the QP formulation is implemented in the PolyMPC framework. The following sections give details of QP preconditioning implementation, tests and benchmarking studies of QP preconditioning.

4.1 QP Preconditioning

Precondition of the QP solver improves the stability of the linear solver in addition to improving the quality of the solution. Further, number of iterations taken by the QP solver and computational costs are expected to be lower with the preconditioning. Hence, existing implementation of QP solver is amended with an independent preconditioning module as an optional setting.

Preconditioning of QP solver

Matrix equilibration based scaling of the QP formulation is found to be simple and computationally efficient[6]. Given a matrix M, Matrix equilibration finds diagonal matrices D and E such that the matrix DME has all columns with equal l_p norm and all rows also have equal lp-norm. Three different methods namely, Sinkhorn-Knopp equilibration algorithm, Ruiz equilibration algorithm and Matrix free algorithms are presented in Article[7]. Further, a modified version of the Ruiz equilibration is proposed in original OSQP paper[4] in order to scale both the KKT matrix and the cost function in QP. Since, the modified version of Ruiz-equilibration is specifically suited for ADMM based QP solver, it is implemented in existing PolyMPC framework, a brief description of the algorithm is given below.

Preconditioning scales the QP matrices P, q, A, l, u presented in QP formulation (Equations 4) so that the condition number of the KKT matrix is equal to 1. The resulting scaled matrices are represented as $\bar{P}, \bar{q}, \bar{A}, \bar{l}, \bar{u}$.

Algorithm 3 QP Preconditioning

```
1:  procedure
2:      $(\bar{P}, \bar{q}, \bar{A}) \leftarrow (P^{n\times n}, q, A^{m\times n})$        ▷ Initialize QP matrices from QP formulation
3:      $(D, E, c) \leftarrow (I^{n\times n}, I^{m\times m}, 1)$        ▷ Initialize preconditioning solution
4:      $(\delta, \epsilon_{equil}) \leftarrow (0^{(n+m)\times 1}, 1e^{-3})$
5:      while $\|1-\delta\|_\infty \geq \epsilon_{equil}$ do
6:          $M \quad \leftarrow \begin{bmatrix} \bar{P} & \bar{A}^T \\ \bar{A} & 0 \end{bmatrix}$        ▷ Construct KKT matrix
7:          for Each column in M with index $i$ do
8:              if $\|M_i\|_\infty \approx$ zero then
9:                  $\delta_i \leftarrow 1$        ▷ Adapted from Scaling in original OSQP code
10:             else
11:                 $\delta_i \leftarrow 1/\sqrt{\|M_i\|_\infty}$        ▷ Scaling parameter for $i^{th}$ column
12:         $D_t \leftarrow$ Diagonal matrix with first $n$ entries of $\delta$        ▷ $n$ is the size of Matrix P
13:         $E_t \leftarrow$ Diagonal matrix with last $m$ entries of $\delta$        ▷ $m$ is number of rows in A
14:         $(\bar{P}, \bar{q}, \bar{A}) \leftarrow (cD_t\bar{P}D_t, cD_t\bar{q}, E_t\bar{A}D_t)$
15:         $\gamma \leftarrow$1/max(mean($\|\bar{P}_i\|_\infty$), $\|\bar{q}_i\|_\infty$)        ▷ Cost scaling parameter
16:         $(\bar{P}, \bar{q}, c) \leftarrow (\gamma\bar{P}, \gamma\bar{q}, \gamma c)$
17:         $(D, E) \leftarrow (D_tD, E_tE)$
18:     return $c, D, E$
```

Scaling matrices(D, E) and cost scaling parameter c are obtained using the preconditioning algorithm described in Algorithm 3. Further, scaling can be applied to QP problem or reverted at any time using the following set of transformations, where (x, y) is the solution to the original QP problem characterized by (P, q, A, l, u), while $(\bar{x}, \bar{y})$ is the solution to the scaled problem characterized by matrices $(\bar{P}, \bar{q}, \bar{A}, \bar{l}, \bar{u})$.

$$\bar{P} = cDPD; \qquad \bar{q} = cDq; \qquad \bar{A} = EAD; \qquad \bar{l} = El; \qquad \bar{u} = Eu;$$
$$\bar{x}^{primal} = D^{-1}x; \qquad \bar{y}^{dual} = cE^{-1}y;$$

In case of QP preconditioning, the residuals and termination criteria also need to be updated with Equations (6) in the existing QP solver implementation.

$$r^k_{primal} = E^{-1}\bar{r}^k_{primal} \tag{6a}$$
$$r^{dual} = c^{-1}D^{-1}\bar{r}^{dual} \tag{6b}$$
$$\epsilon_{prim} = \epsilon_{abs} + \epsilon_{rel}\max\left\{\|E^{-1}\bar{A}\bar{x}^k\|_\infty, \|E^{-1}\bar{z}^k\|_\infty\right\} \tag{6c}$$
$$\epsilon_{dual} = \epsilon_{dual} + \epsilon_{rel}c^{-1}\max\left\{\|D^{-1}\bar{P}\bar{x}^k\|_\infty, \|D^{-1}\bar{A}^T\bar{y}^k\|_\infty, \|D^{-1}\bar{q}\|_\infty\right\} \tag{6d}$$

4.2 Benchmarking of QP preconditioning

In order to benchmark the performance of the QP solver with preconditioning, various instances of random QPs with varying dimensions are generated and solved using PolyMPC. The QP matrices given in Equations (3) are generated randomly using normal distribution. Further, in each of the QP instance, number of constraints are considered to be double that of the number of variables (n).

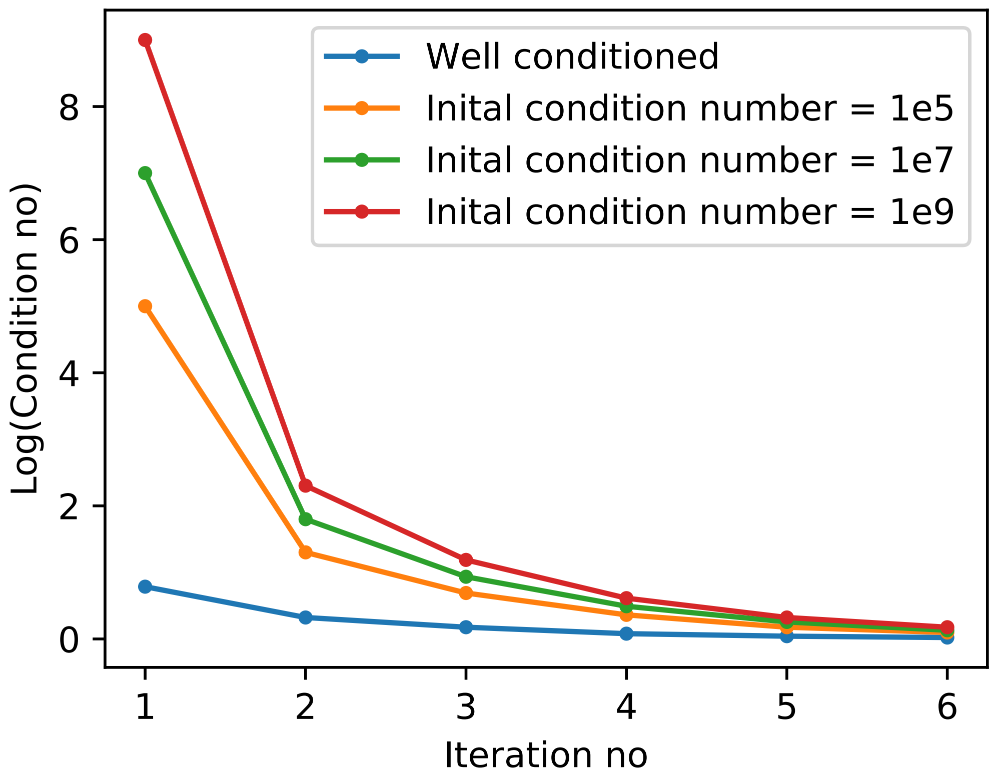

Figure 1: Matrix equilibration: Condition number variation w.r.t number of preconditioning iterations

The convergence criteria for the preconditioning algorithm given in Algorithm 3 requires the KKT matrix norm to be close to 1. However, the number of required iterations to reach this convergence level may be indeterministic in some cases[7]. Hence, it is recommended to limit the maximum number of preconditioning iterations in Algorithm 3. Figure 1 shows the variation of condition number w.r.t number of preconditioning iterations for various instances of QPs with different initial condition number. It is observed that the preconditioning algorithm is very effective in scaling ill-conditioned QPs within few iterations (≈ 6). Hence, the max number of iterations recommended is 6 to have condition number between 1 and 2 for given QPs initial condition numbers ranging from 1 to 1e12.

Preconditioning algorithm profiling[1]

In order to optimize the performance of QP preconditioning implementation, a comprehensive profiling exercise is carried out using random instances of QPs. Table 1 summarizes the time taken for different steps in preconditioning algorithm 3.

Figure 2 shows the time taken by various steps in QP preconditioning algorithm for QPs of varying dimensions. Based on the profiling results, scaling costs of QP and objective are tuned in the implementation. Further, initialization of KKT matrix in each of the preconditioning algorithm is found to be most time consuming followed by the matrix column estimation norm operation.

[1]All the profiling results in this report are generated using the Machine: Mac, 3.0 GHz x 4 Intel Core i7. Ubuntu O.S

Steps	Description	Computation time w.r.t QP size (n, m = 2n)				
		[μs], n=1	[μs], n=10	[μs], n=50	[μs], n=100	[μs], n=200
06-06: Alg 3	KKT matrix	0.03	0.3	6.4	29.0	459.0
07-11: Alg 3	Column norm	0.03	0.3	6.6	26.0	320.0
12-14: Alg 3	Scaling QP	0.02	0.1	3.3	13.0	116.0
15-17: Alg 3	Cost scaling	0.02	0.07	1.1	6.0	75.0
06-17: Total	Iteration time	0.1	0.77	17.4	74.0	970.0

Table 1: QP preconditioning time for randomly generated QPs. Machine: Mac, 3.0 GHz x 4 Intel Core i7. Ubuntu O.S

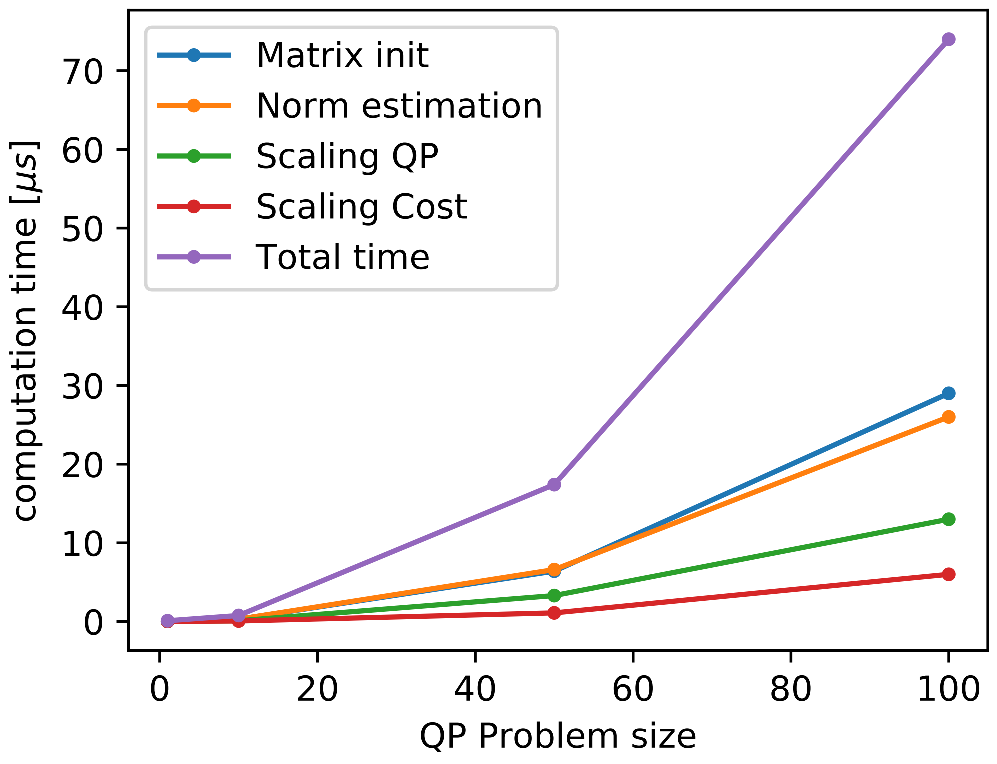

Figure 2: Time taken by various steps in QP preconditioning for different QP dimensions

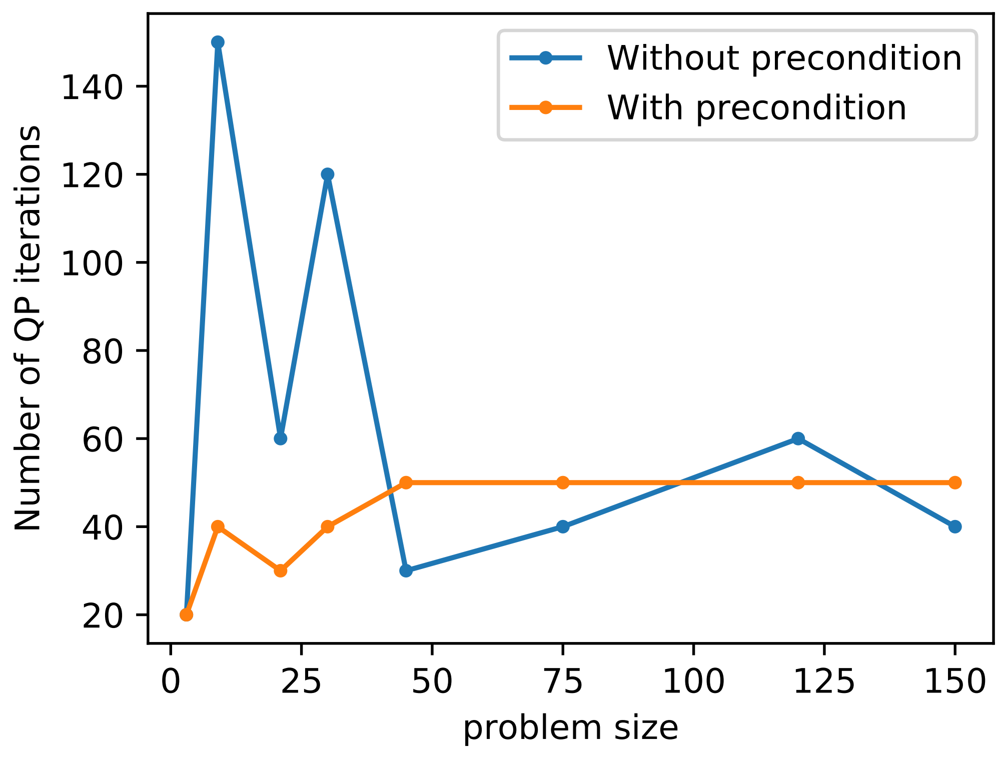

Figure 3: Number of QP solver iterations for well conditioned QP with preconditioning and without preconditioning

4.3 Profiling QP solver

In order to estimate the computational costs of preconditioning and its effect on the overall solving time, various QPs of varying dimensions are solved with and without preconditioning, the number of iterations as well as the computational time are compared in this section. Figure 3 shows the comparison of number of iterations while solving random QP instances with and without precondition option. It is observed that the number of QP solve iterations are lower for problem of dimension less than 50. However, for larger problem dimensions preconditioning increases the computational costs due to higher time complexity of preconditioning algorithm in higher dimensions. Figure 4 shows the QP solver iterations for an ill-conditioned QP with and without preconditioning. It is observed that the number of iterations remain almost same with respect to QP preconditioning.

Figure 3 shows the comparison of solve time for random QP instances with and without precondition option. It is observed that the overall solve time is comparable in both cases for QPs of dimension upto 150. However, for larger problem dimensions preconditioning increases the computational costs due to higher time complexity of preconditioning algorithm in higher dimensions. Figure 6 shows the solver iterations for ill-conditioned QPs with and without preconditioning. It is observed that the QP solve time remain almost same with respect to preconditioning.

In conclusion, we observe that the number of QP solver iterations and QP solve time remain invariant w.r.t QP preconditioning. The results remain same irrespective of the initial conditioning of the corresponding KKT system.

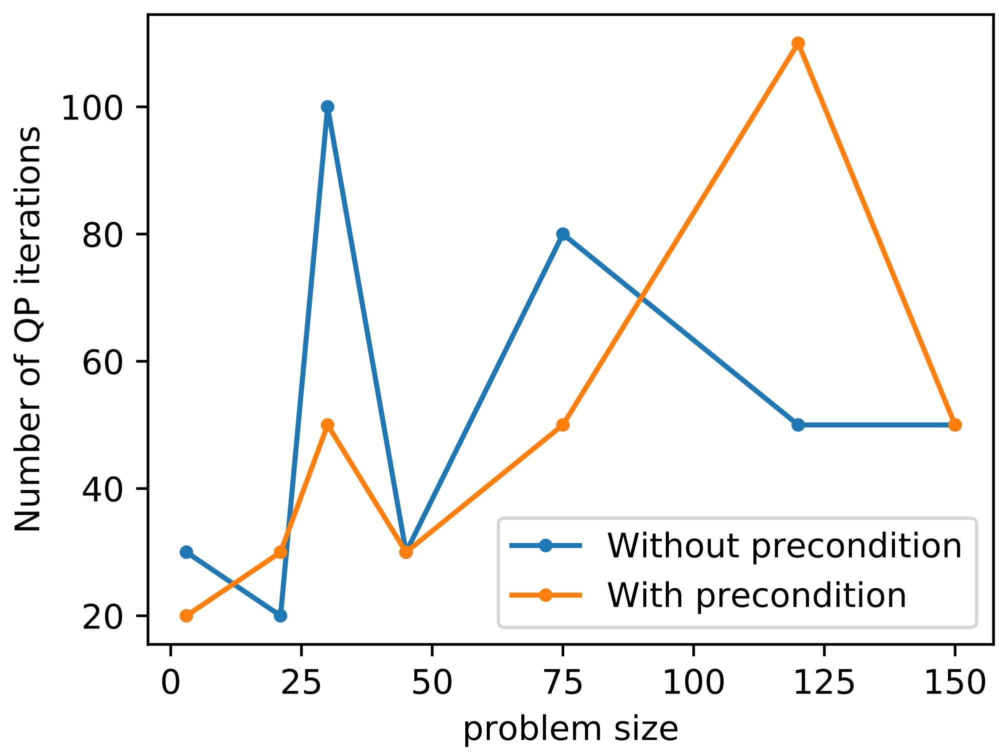

Figure 4: Number of QP solver iterations for ill-conditioned QP(condition no: 1e9) with preconditioning and without preconditioning

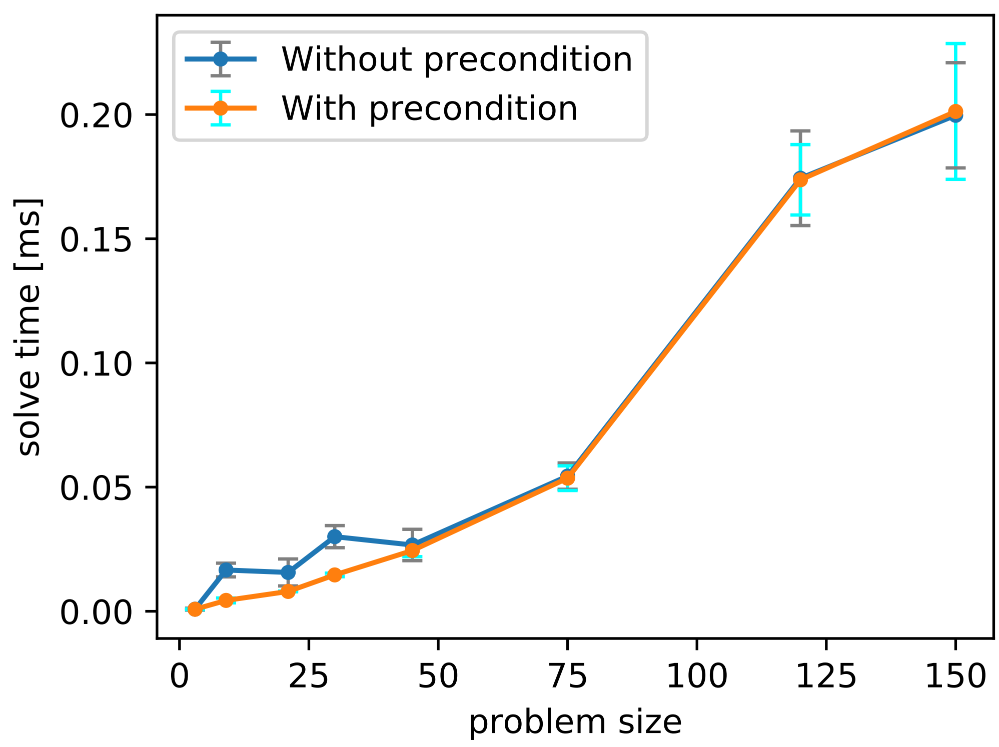

Figure 5: QP solve time for randomly generated well-conditioned QPs with and without QP preconditioning

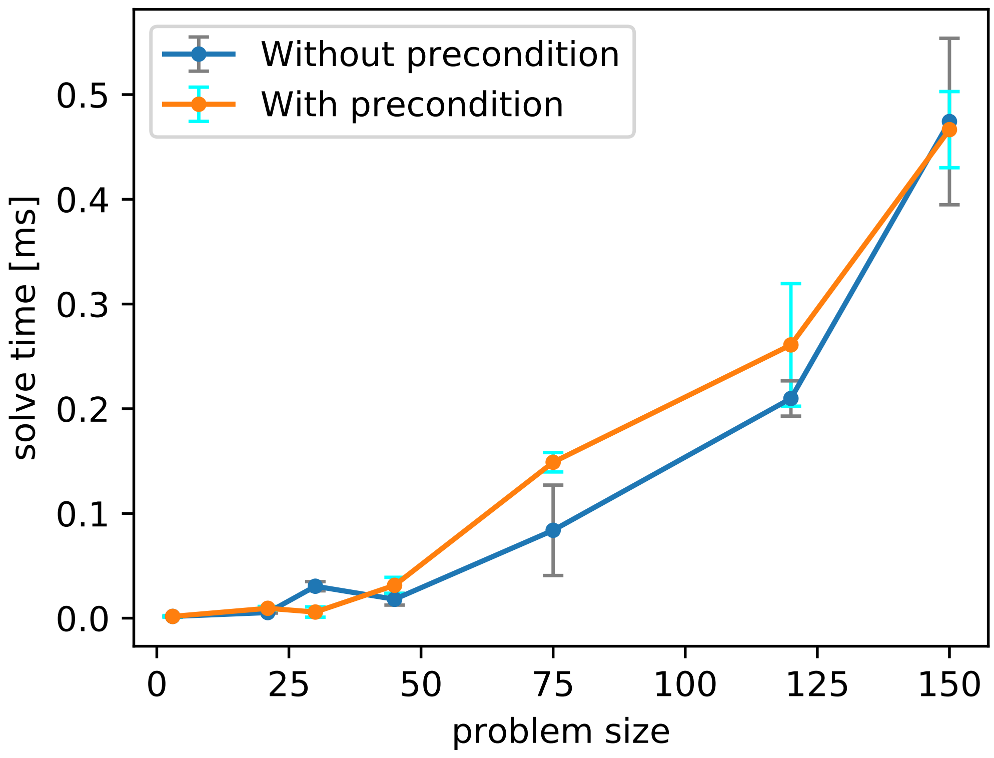

Figure 6: QP solve time for randomly generated ill-conditioned QPs(Condition no:1e9) with and without QP preconditioning

5 Benchmarking SQP with QP preconditioning

SQP algorithm uses the QP solver in every iteration to solve for the optimal descent direction. In order to profile and benchmark SQP solver, a d-dimensional Rosenbrock function[8] is taken as a test case for optimization. Figure 7 shows the time taken for various steps in SQP algorithm (Algorithm 1) in order to solve Rosenbrock problem of various dimensions without QP preconditioning. It is observed that the QP solver constitutes more than 90% of the SQP solver time in each SQP iteration. Further, QP setup/update time is found to be most costly step with respect to the computation time followed by QP solve time itself. SQP algorithm steps such as linearization, linesearch and BFGS update are found to be relatively inexpensive operations in SQP iterations.

Figure 8 shows the time taken for various steps in SQP algorithm (1 to solve Rosenbrock problem of various dimensions with QP preconditioning. It is observed that preconditioning is a costly operation in SQP iteration compared to other steps. Since, preconditioning is an additional operation w.r.t normal SQP steps, the overall time taken for SQP solution is higher for the SQP with QP preconditioning.

Figure 9 shows the number iterations for solving Rosenbrock problem w.r.t QP preconditioning. We note that the overall number of SQP iterations remain invariant w.r.t QP preconditioning.

Figure 10 shows the time taken for solving Rosenbrock problem w.r.t QP preconditioning. For problems of low dimension, solve time is comparable since the QP scaling time is less. However, preconditioning for problems of large dimensions results in higher SQP computational costs.

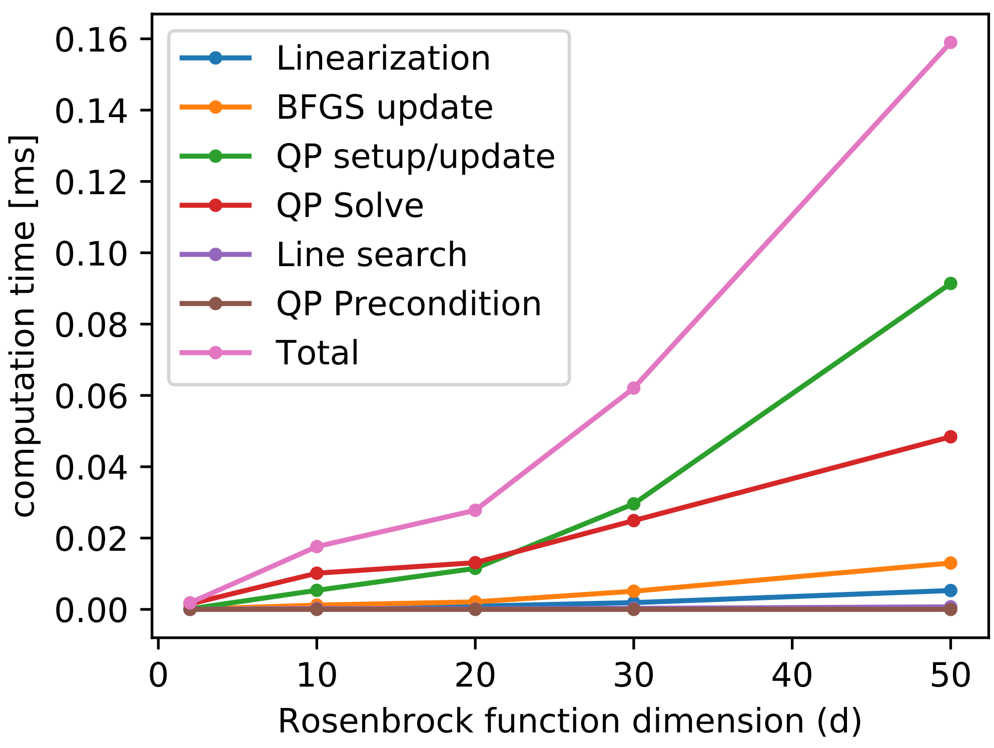

Figure 7: SQP solver profiling without QP preconditioning: Time taken by various parts of SQP algorithm

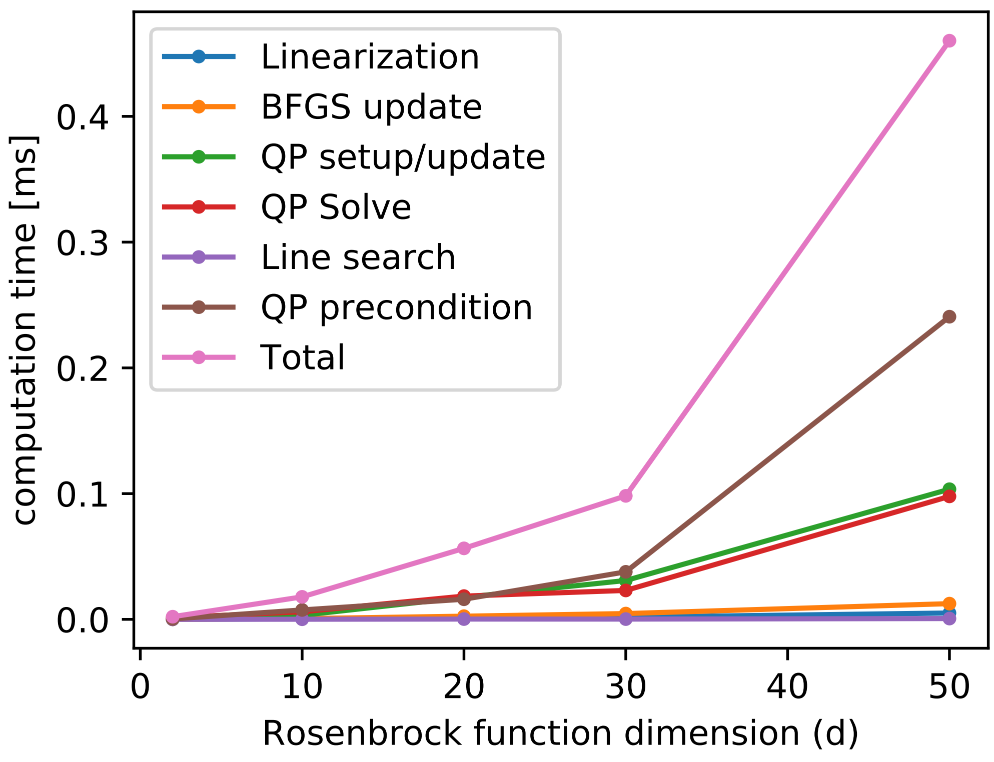

Figure 8: SQP solver profiling with QP preconditioning: Time taken by various parts of SQP algorithm

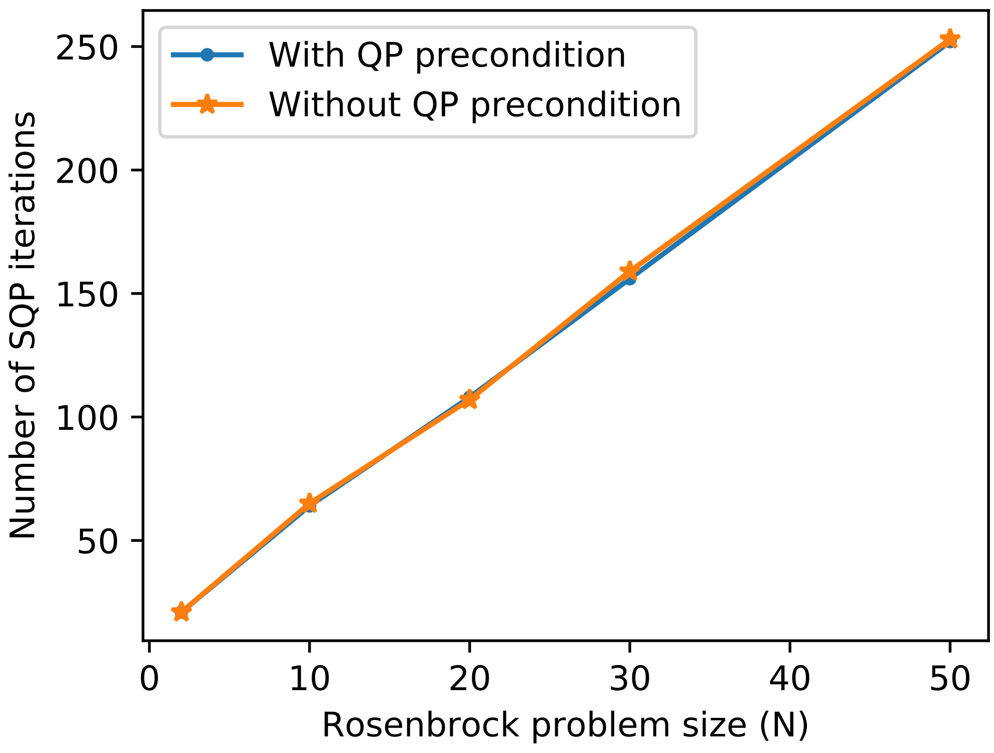

Figure 9: Number of SQP iterations when QP is preconditioned vs QP without preconditioning as a function of Rosenbrock problem dimension

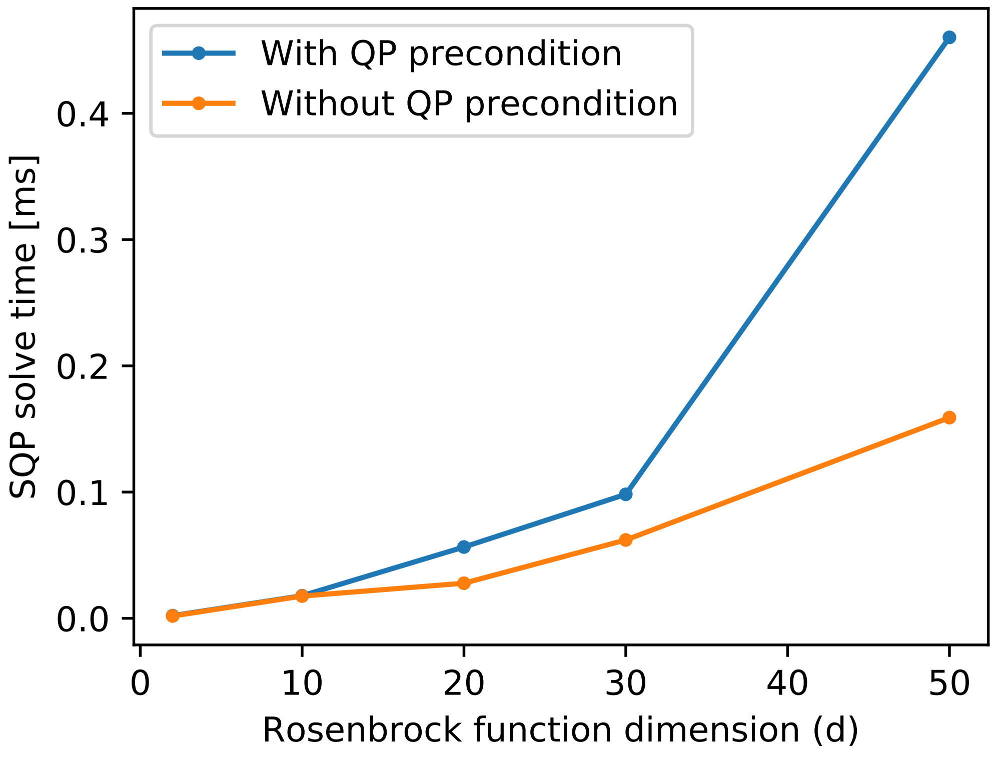

Figure 10: SQP solve time when QP is preconditioned vs QP without preconditioning as a function of Rosenbrock problem dimension

6 Non-linear optimization of Solid Oxide Fuel Cell Dynamics using PolyMPC

The Solid Oxide Fuel Cell (SOFC) system consists of four main subsystems namely reformer, SOFC stack, after-burner, heat-exchangers. The schematic of the SOFC system is shown in Figure 11.

6.1 Plant model

SOFC plant model which is required for the real time optimization is given by a non-linear function (f) which depends on inputs(u) and parameters(θ). The governing plant dynamics are given in equations 7, where, the parameters u, θ are given in (9a), (9b).

$$[d\theta_1, m_1] = \textbf{Reformer}\left(u, \theta, T_{in}^1\right) \tag{7a}$$
$$[d\theta_2^5, m_2, U_{cell}, \nu, \lambda_{air}] = \textbf{SOFC}\left(u, \theta, m_1\right) \tag{7b}$$
$$[d\theta_6, m_3] = \textbf{Burner}\left(u, \theta, m_2\right) \tag{7c}$$
$$[d\theta_7^{15}] = \textbf{HeatEX}\left(u, \theta, m_1, m_3, T_{in}^2, T_{in}^3\right) \tag{7d}$$

Each of the subsystem in SOFC is modelled independently with required interconnections from the remaining units as represented in the equations (7). $T_{in}^1, T_{in}^2, T_{in}^3$ are pre-defined external inputs.

6.2 Optimization problem formulation

In order to find the optimal operating point of the Solid Oxide Fuel Cell (SOFC) system in real time, an optimization problem is formulated. Outline of the optimization formulation is given in equations (8) and (9).

$$\min_u \quad \Phi(u) := \phi(u, y(u, \theta)) \tag{8a}$$
$$\text{subject to} \quad H(u) := h(u, y(u, \theta)) = 0, \tag{8b}$$
$$G(u) := g(u, y(u, \theta)) \leq 0, \tag{8c}$$
$$u^L \leq u \leq u^U, \tag{8d}$$

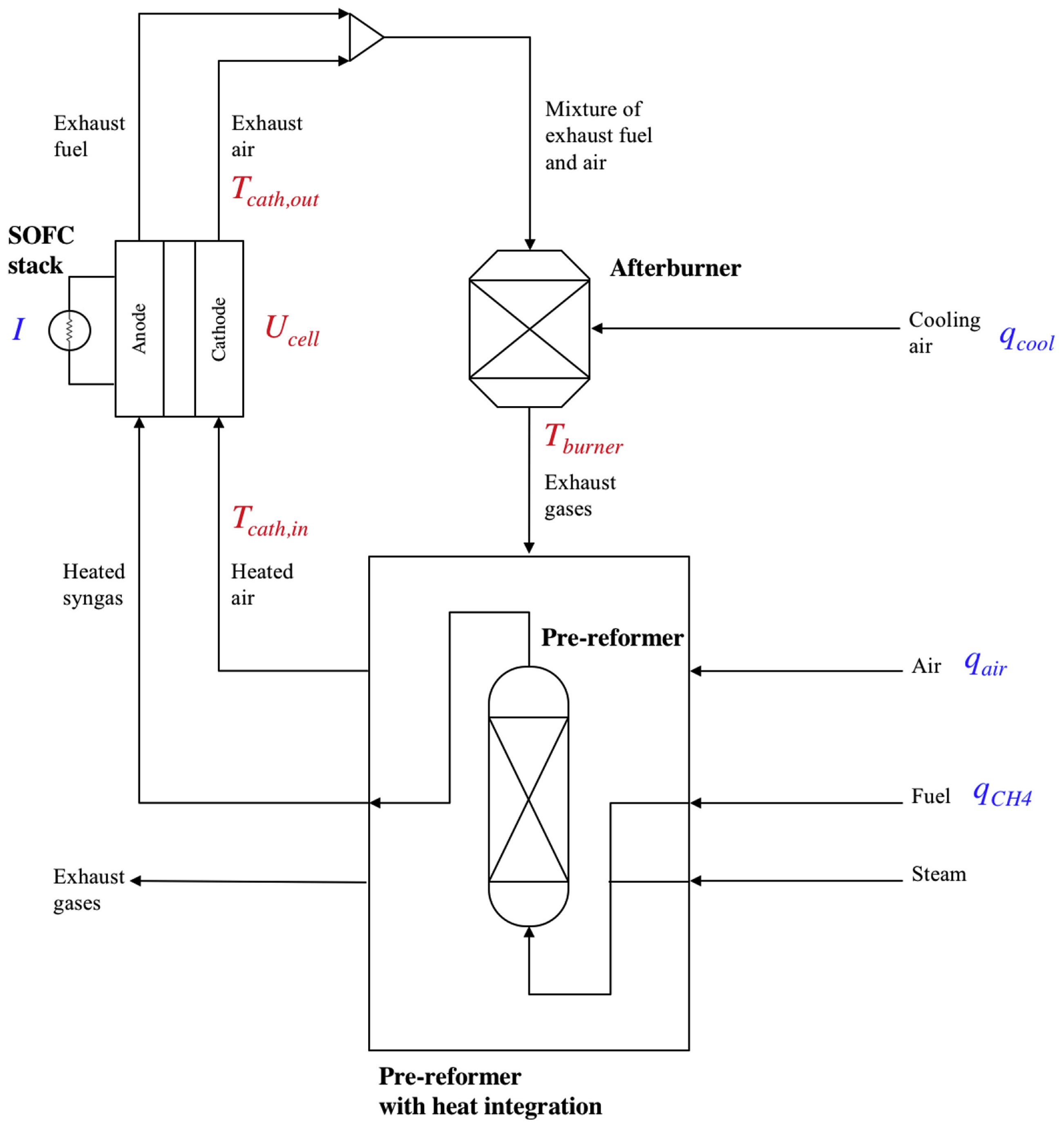

Figure 11: Schematic of the SOFC plant [9]

where

$$u = [q_{CH4}, q_{air}, I, q_{cool}] \tag{9a}$$

$$\theta = [T_i], \qquad i = 1, \dots 15 \tag{9b}$$

$$U_{cell} = y(u, \theta) \tag{9c}$$

$$\Phi = \frac{u_3 U_{cell}}{u_1 k} - \delta_{air} u_2^2 - \delta_{cool} u_4^2 \tag{9d}$$

$$H = \begin{pmatrix} P_{el}^{ref} - u_3 U_{cell} N_{cell} \\ f(u, \theta)_{15\times1} \end{pmatrix} \tag{9e}$$

$$G = \begin{pmatrix} 0.7 - U_{cell} \\ \dfrac{u_3 N_{cell}}{u_1 k} - 0.8 \\ 4 - \dfrac{u_2}{2u_1} \\ 650 - \theta_1 \\ \theta_1 - 750 \\ 650 - \theta_5 \\ \theta_5 - 790 \\ \theta_6 - 890 \end{pmatrix} \tag{9f}$$

$$u^L = \begin{pmatrix} 1 \\ 85 \\ 0 \\ 0 \end{pmatrix} \tag{9g}$$

$$u^U = \begin{pmatrix} 7 \\ 200 \\ 50 \\ 40 \end{pmatrix} \tag{9h}$$

and $N_{cell}, k, \delta_{air}, \delta_{cool}, P_{el}^{ref}$ are predefined constants.

6.3 Implementation of SOFC dynamics in PolyMPC

The dynamics of the SOFC problem is available as a collection of MATLAB functions. However, in order to solve the problem using PolyMPC, we are interested in converting them to C++ code. For this purpose, the experiments were carried out with Matlab C-Code generation tool. This turned out to be a very complex exercise due to following reasons. Automatic Differentiation of PolyMPC is not compatible with the variable type declarations in auto-generated code. Hence, the autogenerated C++ code would have to be manually edited before it could be used by PolyMPC. This would have been possible, but the auto generated C++ code for SOFC happened to be too big to be manually edited(5000 lines of code for the dynamics). In addition, there were additional routines generated by MATLAB which were not entirely compatible with the PolyMPC C++ implementation. Moreover, the variable naming in MATLAB C++ code generation is found to be ambiguous because the same variable names are used multiple times. Therefore, all of the code needed to be re-written in C++ if we were to solve the entire SOFC dynamics. Since, this is a very complex exercise to be completed within the score of the semester project, it was decided to reduce the SOFC problem dimension so that it could be implemented in C++.

Simplified SOFC dynamics

Reformer dynamics and SOFC dynamics are re-implemented in C++ which included complex a set of expressions as part of equality constraints in optimization problem formulation using the auto-generated C++-code. The remaining parameters of the original SOFC problem were assumed to be constant, corresponding to a steady solution obtained using MATLAB. Reduced dimension SOFC problem is given in equations 10.

$$\min_{[u_1,u_2,u_3,u_4,U_{cell},\theta]} \quad -u_3U_{cell} + \left(\delta_{air}u_2^2 + \delta_{cool}u_4^2\right)u_1k \qquad Objective \tag{10a}$$

$$\text{s.t} \quad H(u) := \begin{pmatrix} P_{el}^{ref} - u_3U_{cell}N_{cell} \\ Reformer(u,\theta)_{1\times1} \end{pmatrix} = 0 \quad Eq.\ Constraints \tag{10b}$$

$$G(u) := \begin{pmatrix} u_3N_{cell} - 0.8u_1k \\ 8u_1 - u_2 \end{pmatrix} \leq 0 \quad Ineq.\ Constraints \tag{10c}$$

$$\begin{pmatrix} 1 \\ 85 \\ 0 \\ 0 \\ 0.7 \\ 650 \end{pmatrix} \leq \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ U_{cell} \\ \theta_1 \end{pmatrix} \leq \begin{pmatrix} 7 \\ 200 \\ 50 \\ 40 \\ \infty \\ \infty \end{pmatrix}; \quad \theta\ Box\ Constraints \tag{10d}$$

6.4 SOFC solution with SQP

Solution without QP preconditioning

Simplified problem formulated in Equations 10 is solved using the SQP solver. Figure 12 shows the convergence pattern of SQP residuals and objective value where QPs are not preconditioned. During the SQP iterations, primal and dual residuals are found to converge very fast but the constraint violation norm is found to be saturated at 1.26. As a result, SQP solution doesn't converge within 100 SQP iterations and 10100 cumulative QP iterations.

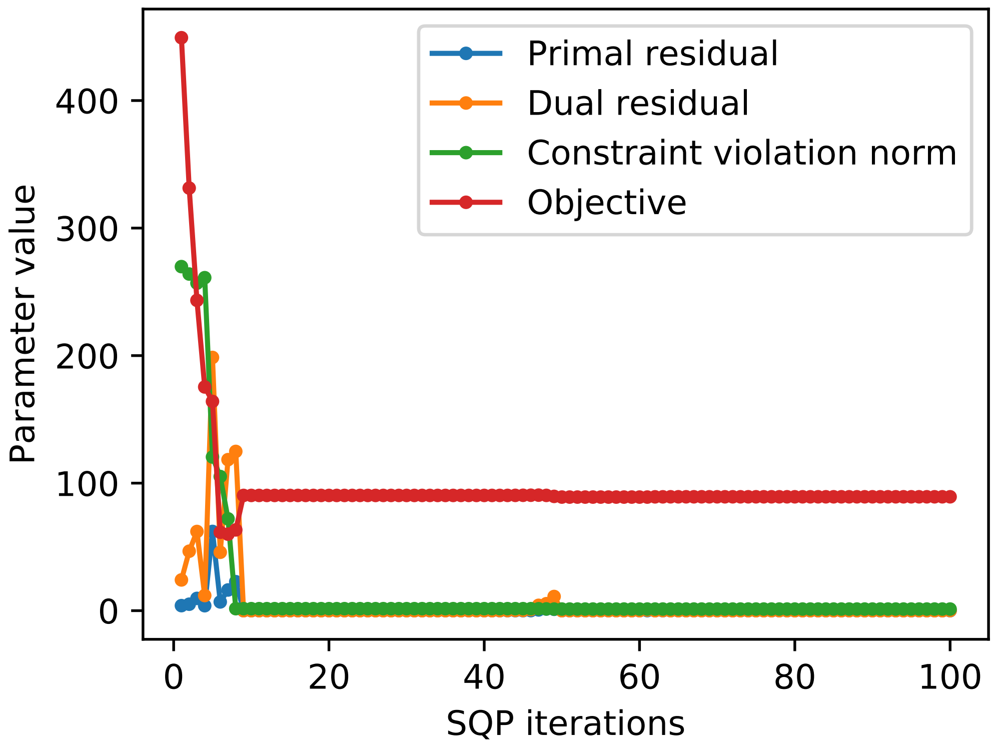

Figure 12: Convergence pattern of Simplified SOFC model without QP precondition : Solution doesn't converge in 100 iterations

Solution with QP preconditioning

Figure 13 shows the convergence pattern for simplified SOFC problem with QP precondition. During the SQP iterations, primal and dual residuals are found to converge very fast but the constraint violation norm reduction is very slow. However, the constraint violation norm reduces to the required residual level in this case. As a result, SQP solution converges within 20 SQP iterations and 876 cumulative QP iterations.

Results and discussion

Evolution of residuals and constraint violation norms for the problem with and without precondition are plotted in Figures (14, 15, 16). It is observed that convergence pattern is better and faster in SQP with QP precondition. Though the time taken for the QP preconditioning operation is higher, the fewer number of SQP iterations makes QP preconditioning option computationally better compared to SQP solution without QP preconditioning.

7 Discussion and Further improvements

With respect to the SQP solver in PolyMPC, the QP solver module is found to be robust, solves convex QPs efficiently. However, several issues w.r.t SQP solver are persistent. Firstly, the line search algorithm doesn't update the variables when the optimal search direction is too small and constraint violation norm is too high. This results in QPs that are almost same at every SQP iteration, leading to no convergence. Hence, a second order line search algorithm may be explored. Secondly, the BFGS update produces non-positive definite hessian in multiple scenarios including the case when

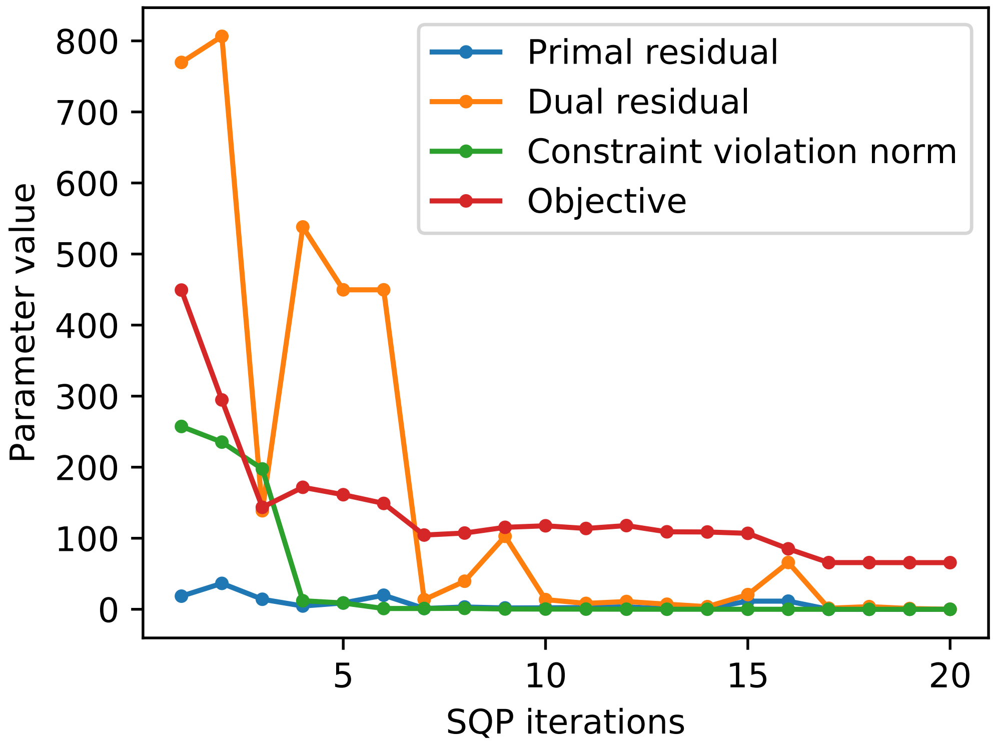

Figure 13: Convergence pattern of Simplified SOFC model with QP precondition : Solution converges within 20 SQP iterations

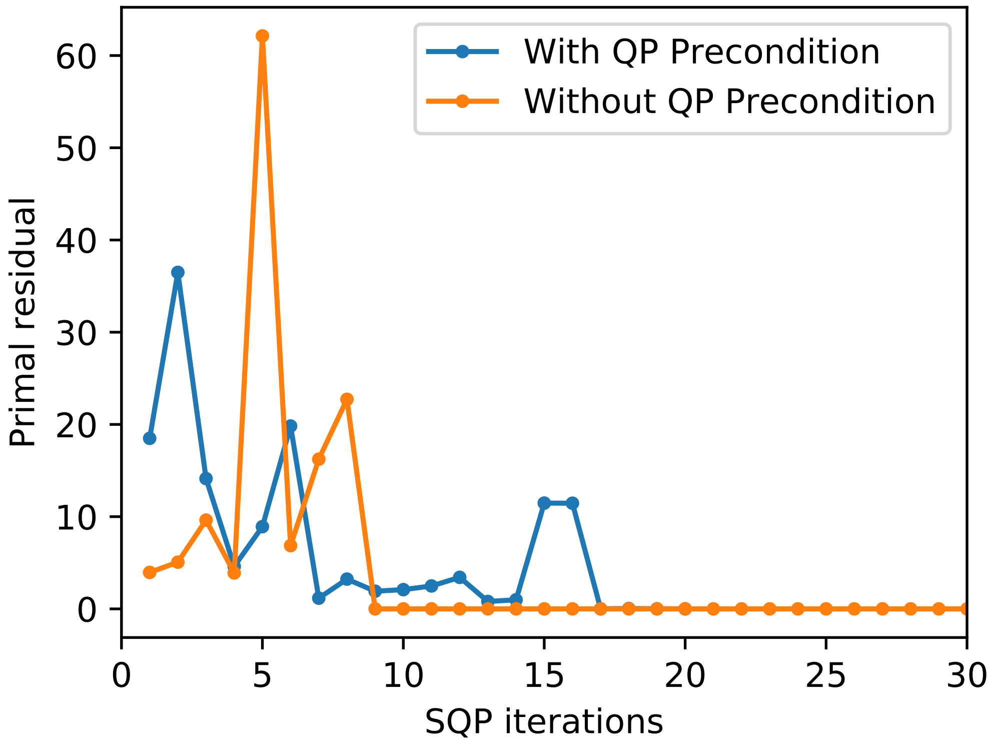

Figure 14: SQP primal residual variation w.r.t SQP iterations

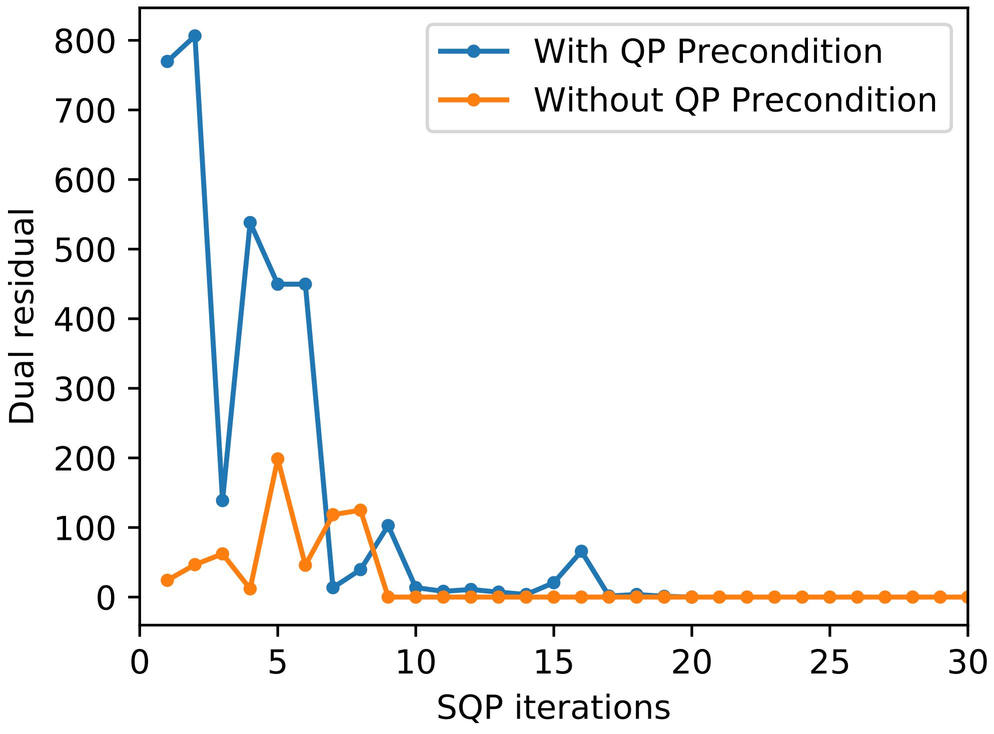

Figure 15: SQP dual residual variation w.r.t SQP iterations

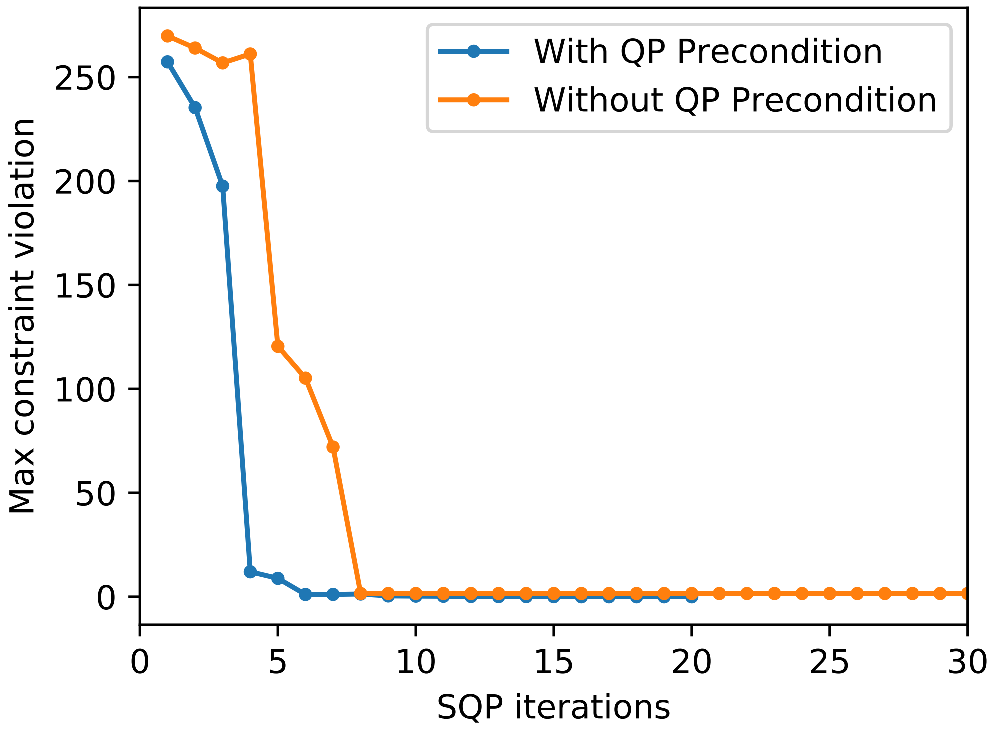

Figure 16: SQP max. constraint violation norm variation w.r.t SQP iterations

successive QPs remain similar. Thirdly, the residuals are found to oscillate close to zero leading to slower convergence or no convergence in some cases. With respect to the SOFC real time optimization problem, it is observed that the real dynamics are highly non-linear. Since the Reformer and SOFC dynamics are already formulated as as SQP problem in PolyMPC, solving a full scale SOFC problem could be attempted as a next logical step.

8 Conclusion

During this project, an additional feature called QP preconditioning is added to the existing QP solver in PolyMPC. Benchmarking of the preconditioning algorithm is carried out to optimize the preconditioning iterations and computational time. Further, profiling of the QP solver and SQP solver in PolyMPC is carried out. It is noticed that the QP setup/update time is the most time consuming operation in the optimization solution procedure. Further, the QP preconditioning time is found to be significant in SQP iterations. For this reason, it is recommended to use the preconditioning solution for multiple iterations once it is computed. Though the QP preconditioning operation is computationally expensive, the SQP convergence pattern and the number of SQP iterations for solving SOFC optimization problem are found to be better in case of preconditioned QPs suggesting that the higher computational time can be offset by the faster SQP convergence.

References

[1] P. Listov and C. Jones, "PolyMPC: An efficient and extensible tool for real-time nonlinear model predictive tracking and path following for fast mechatronic systems", en, *Optimal Control Applications and Methods*, vol. n/a, no. n/a, ISSN: 1099-1514. DOI: 10.1002/oca.2566. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/oca.2566 (visited on 01/13/2020).

[2] J. Nocedal and S. Wright, *Numerical Optimization*, en, 2nd ed., ser. Springer Series in Operations Research and Financial Engineering. New York: Springer-Verlag, 2006, ISBN: 9780387303031. DOI: 10.1007/978-0-387-40065-5. [Online]. Available: https://www.springer.com/de/book/9780387303031 (visited on 11/09/2019).

[3] M. SPIELER, "Master thesis: First-order optimization methods in embedded nonlinear model predictive control, epfl, la3", Jun. 2019.

[4] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "OSQP: An operator splitting solver for quadratic programs", *ArXiv e-prints*, Nov. 2017. arXiv: 1711.08013 [math.OC].

[5] *Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers.* [Online]. Available: https://web.stanford.edu/~boyd/papers/admm_distr_stats.html (visited on 01/19/2020).

[6] P. A. Knight, D. Ruiz, and B. Uçar, "A Symmetry Preserving Algorithm for Matrix Scaling", en, *SIAM Journal on Matrix Analysis and Applications*, vol. 35, no. 3, p. 25, 2014. DOI: 10.1137/110825753. [Online]. Available: https://hal.inria.fr/inria-00569250 (visited on 01/13/2020).

[7] R. Takapoui and H. Javadi, "Preconditioning via Diagonal Scaling", *arXiv:1610.03871 [math]*, Oct. 2016, arXiv: 1610.03871 version: 1. [Online]. Available: http://arxiv.org/abs/1610.03871 (visited on 01/10/2020).

[8] *Rosenbrock Function.* [Online]. Available: https://www.sfu.ca/~ssurjano/rosen.html (visited on 01/11/2020).

[9] T. de Avila Ferreira, Z. Wuillemin, A. G. Marchetti, C. Salzmann, J. Van Herle, and D. Bonvin, “Real-time optimization of an experimental solid-oxide fuel-cell system”, en, *Journal of Power Sources*, vol. 429, pp. 168–179, Jul. 2019, ISSN: 0378-7753. DOI: 10.1016/j.jpowsour.2019.03.025. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378775319302642 (visited on 11/04/2019).

9 Annexure: Program listing

9.1 QP Precondition implementation

```
#ifndef PRECONDITION_QP_HPP
#define PRECONDITION_QP_HPP

#include <Eigen/Dense>
#ifdef QP_SOLVER_USE_SPARSE
#include <Eigen/Sparse>
#endif
#include <limits>

/** Precondition of QP
 * Implements "Procedure from OSQP paper: http://adsabs.harvard.edu/abs/2017
   arXiv171108013S.
 *
 * @param[in, out]      P
 * @param[in, out]      q
 * @param[in, out]      A
 * @param[in, out]      l
 * @param[in, out]      u
 * @param[out]          D
 * @param[out]          E
 * @param[out]          c
 */


#ifdef QP_SOLVER_USE_SPARSE
    template <typename SpMat, typename Mat>
    void sparse_insert(SpMat &dst, int row, int col, const Mat &src)
    {
        for (int k = 0; k < dst.outerSize(); ++k) {
            for (typename SpMat::InnerIterator it(dst, k); it; ++it) {
                int row, col;
                row = it.row();
                col = it.col();
                it.valueRef() = src(row, col);
            }
        }
    }
#endif


template <typename MatP, typename Matq, typename MatA,
          typename Matl, typename MatM,
          typename MatD, typename MatE, typename Matdelta,
          typename MatAt, typename c_t>
void precondition_qp(const MatP& P, const Matq& q, const MatA& A, const Matl& l,
                     MatM& M, MatD& D, MatE& E, Matdelta& delta, MatAt& At, c_t&
    c)
{
    using Scalar = typename MatP::Scalar;
    const int n = P.rows();
    const int m = A.rows();
    const int max_iter = 6;

    c = 1;
    Scalar gamma_scaling;

```

```cpp
    D.setIdentity();
    E.setIdentity();

    Matq Dt_diag;
    Matl Et_diag;

    Matq D_diag = D.diagonal();
    Matl E_diag = E.diagonal();

    MatD Pt = P*D;
    At = A*D;
    delta.setZero();

    Matq qt = q;

    M.setZero();
    delta.setZero();

    Scalar _approx_zero = 1e-6;
    for (int iter=1; iter <= max_iter; iter++)
    {
        M.topLeftCorner(n, n) = Pt;
        M.topRightCorner(n, m) = At.transpose();
        M.bottomLeftCorner(m, n) = At;

        // Find column norm for the matrix M
        delta = M.colwise().template lpNorm<Eigen::Infinity>().transpose();
        // Replace the zero norms with 1
        if (delta.minCoeff() < _approx_zero)
        {
            for (int r=0; r < n+m; r++) {if (abs(delta(r)) < _approx_zero) delta(r) = 1.0;}
        }//
        delta = delta.cwiseSqrt().cwiseInverse();
        Dt_diag = delta.template head(n);
        Et_diag = delta.template tail(m);

        // Apply M equilibration
        Pt =c *  Pt.cwiseProduct(Dt_diag.replicate(1, n)).transpose().cwiseProduct(Dt_diag.replicate(1, n)).transpose();
        qt = c * qt.cwiseProduct(Dt_diag);
        At = At.cwiseProduct(Et_diag.replicate(1, n)).transpose().cwiseProduct(Dt_diag.replicate(1, m)).transpose();
        gamma_scaling = 1/std::max(Pt.colwise().template lpNorm<Eigen::Infinity>().mean(), qt.template lpNorm<Eigen::Infinity>());
        Pt = gamma_scaling * Pt;
        qt = gamma_scaling * qt;
        c = gamma_scaling * c;

        D_diag = D_diag.cwiseProduct(Dt_diag);
        E_diag = E_diag.cwiseProduct(Et_diag);
    }
    D = D_diag.asDiagonal();
    E = E_diag.asDiagonal();
}

#endif /* PRECONDITION_QP_HPP */
```

9.2 SOFC Reformer dynamics implementation

```
template <typename A, typename B, typename C>
void main_ref_rate_Tr(const A& x, B& dTheta, const double Tin[3], C& outflow
)
{
  const Scalar Tflow1 = Tin[0];
  const Scalar Tflow2 = theta[10];

  auto nCH4inp = (P*x(0))/((R*T_st)*(6e+4)); // [mol/s]
  auto inflow1 = nCH4inp;
  auto inflow2 = 2.0*nCH4inp;

  auto x_1 = (inflow1 * R*T_st * 60000.0 / 101325.0 - 5.5) / 2.627;
  auto y_1 = (x(5) - 673.1) / 176.1;
  auto X_data_idx_0 = 0.02664 - 0.02357*x_1 + 0.02388*y_1 + 0.02455*x_1*x_1
- 0.0457*x_1*y_1 + 0.06014*y_1*y_1 + 0.0225*x_1*x_1*y_1 -
0.0264*x_1*y_1*y_1 + 0.0231*y_1*y_1*y_1;
  auto X_data_idx_1 = 0.9996 - 0.0001299*x_1 - 0.0003088*y_1 +
7.618e-6*x_1*x_1 - 7.341e-5*x_1*y_1 + 8.7e-5*y_1*y_1 +
1.304e-7*x_1*x_1*y_1 + 2.236e-5*x_1*y_1*y_1 + 8.649e-5*y_1*y_1*y_1;

   if (inflow1 <= 0.0001175) {
     auto X_data_idx_0 = 0.99999;
     auto X_data_idx_1 = 1.0;
  }

  auto rR = X_data_idx_0 * inflow1;
  auto ksi = rR * (1.0 - X_data_idx_1);
  auto nCH4out = inflow1 - rR;
  auto nH2Oout = (inflow2 - rR) - ksi;
  auto nCOout = rR - ksi;
  auto nH2out = 3.0 * rR + ksi;
  const Scalar DHr = 206.1e3;
  const Scalar DHs = -41.15e3;
  const Scalar ref_eps = 0.8;
  const Scalar cst_sigma = 5.670373e-8;
  auto Q_rad = ref_eps*cst_sigma*0.0001*(x(5)*x(5)*x(5)*x(5) - std::pow(T_ht
, 4.0));

auto dTr =
(inflow1*((1515+R*1000)*(Tflow1-T_ref)+83.47/2*(Tflow1*Tflow1-T_ref*T_ref)
- 0.02053/3*(Tflow1*Tflow1*Tflow1-std::pow(T_ref, 3))) + inflow2 *
((20750+R*1000)*(Tflow2-T_ref)+12.15/2*(Tflow2*Tflow2-T_ref*T_ref)) -
(nCH4out * ((1515+R*1000)*(x(5)-T_ref)+83.47/2*(x(5)*x(5)-T_ref*T_ref) -
0.02053/3*(x(5)*x(5)*x(5)-std::pow(T_ref, 3))) +
nH2Oout*((20750+R*1000)*(x(5)-T_ref)+12.15/2*(x(5)*x(5)-T_ref*T_ref)) +
nH2out*((18290+R*1000)*(x(5)-T_ref)+3.719/2*(x(5)*x(5)-T_ref*T_ref)) +
ksi*((23690+R*1000)*(x(5)-T_ref)+31.01/2*(x(5)*x(5)-T_ref*T_ref) -
(8.875e-3)/3*(x(5)*x(5)*x(5)-std::pow(T_ref, 3))) +
nCOout*((19660+R*1000)*(x(5)-T_ref)+5.019/2*(x(5)*x(5)-T_ref*T_ref))))
/1000.0-rR*DHr
- ksi*DHs - Q_rad;

  outflow << nH2out, nH2Oout, nCH4out, nCOout, ksi;

  dTheta(1) = dTr;
}
```

9.3 SOFC dynamics implementation

```
template <typename A, typename B, typename C>
void main_SOFC_Dynamics(const A& x, B& dTheta, const double Tin[3], double inflow[5], C& outflow)
{
    Scalar Tin1 = theta[13];
    Scalar Tin2 = theta[7];

    auto nCH4inp    = (P*x(0))/((R*T_st)*(6e+4)); // [mol/s]      methane flow rate
    auto nO2cathinp = (P*x(1)*0.21)/((R*T_st)*(6e+4)); // [mol/s]      oxygen flow rate
    auto u0 = nCH4inp;
    auto u1 = nO2cathinp;
    auto u2 = x(2);

    const Scalar CH_r_N2  = 3.76190444967126;
    auto b_inflow6 = u1 * CH_r_N2;
    auto A_el0 = 0.98 * inflow[2];
    auto nH2in = inflow[0] + 4.0 * A_el0;
    auto nH2Oin = inflow[1] - 2.0 * A_el0;
    auto QrF0 = N_cell * u2;
    auto dxt_idx_1 = QrF0 / (2.0 * cst_F);
    auto nH2r = dxt_idx_1;
    if (dxt_idx_1 > nH2in) {
        auto nH2r = 0.9999 * nH2in;
    }

    auto nO2r = QrF0 / (4.0 * cst_F);
    if (nO2r > u1) {
        auto nO2r = 0.9999 * (nH2in / 2.0);
    }

    auto nH2_idx_2_tmp = nH2in - nH2r;
    auto nH2O_idx_2_tmp = nH2Oin + dxt_idx_1;
    auto nO2_idx_2_tmp = u1 - nO2r;
    outflow << nH2_idx_2_tmp, nH2O_idx_2_tmp, inflow[2] - A_el0, inflow[3],
    inflow[4] + A_el0, nO2_idx_2_tmp, b_inflow6;
    auto FU = QrF0 / 8.0 / cst_F / u0;

    //  Fuel utilisation [-]
    auto L_air = u1 / 2.0 / u0;
    auto T_elchem = Tin1 + (x(8) - Tin1) * 0.7479;

    auto A_el = pow(x(8), 0.25) - 4.1553604643151418;
    auto QrF = log(x(8) / 298.15);
    auto i = nH2_idx_2_tmp + nH2O_idx_2_tmp;
    auto log_temp1 = nH2O_idx_2_tmp / i;
    auto log_temp2 = nH2_idx_2_tmp / i;
    auto log_temp3 = nO2_idx_2_tmp /(nO2_idx_2_tmp + b_inflow6);

    auto U_Nernst =  -(((((((143.05 * (x(8) - 298.15) + -241826.0) - 46.432 *
    (pow(x(8), 1.25) - 1238.9207224355596)) + 5.5167333333333337 *
    (pow(x(8), 1.5) - 5148.1621884294782)) - 0.0184945 * (x(8) * x(8) -
    88893.422499999986)) - x(8) * (((((143.05 * QrF + 188.83) - 232.16 *
    A_el) + 16.5502 * (sqrt(x(8)) - 17.267020588393354)) - 0.036989 * (x(8)
    - 298.15)) - 8.31451 * log(log_temp1))) - ((((56.505 * (x(8) - 298.15) -
    88890.4 * A_el) + 116500.0 * QrF) + 1.1214E+6 * (pow(x(8), -0.5) -
    0.05791387083143839)) - x(8) * (((((56.505 * QrF + 130.59) +
    29630.133333333331 * (pow(x(8), -0.75) - 0.013937147289334706)) -
    116500.0 * (1.0 / x(8) - 0.0033540164346805303)) + 373800.0 * (pow(x(8),
    -1.5) - 0.00019424407456460974)) - 8.31451 * log(log_temp2)))) - 0.5 *
    ((((37.432 * (x(9) - 298.15) + 8.0408E-6 *(pow(x(9), 2.5) -
    1.5349245564802489E+6)) + 357140.0 *(pow(x(9), -0.5) -
    0.057913870831 43839)) - 2.3688E+6 * (1.0 / x(9) -
    0.0033540164346805303)) - x(9) * (((((37.432 * log(x(9)/298.15) +
    205.14) + 1.3401333333333335E-5 * (pow(x(9), 1.5) - 5148.1621884294782))
    + 119046.66666666667 * (pow(x(9), -1.5) - 0.00019424407456460974)) -
    1.1844E+6 * (pow(x(9), -2.0) - 1.1249426244107095E-5)) - 8.31451 *
    log(log_temp3)))) / 2.0 / cst_F;

    auto i1 = u2 / CE_A_act;

    auto A_el1 = R * T_elchem;
    auto QrF_arg = i1 / (2.0 * (A_el1 * 2.0 / cst_F * kinetic_k0 *
    exp(-kinetic_E_actcath /R/ T_elchem)));
    auto QrF1 = log(QrF_arg + sqrt(1 + QrF_arg*QrF_arg)); // Inverse sinh
    auto n_loss0 = A_el1 / cst_F * QrF1;
    auto n_loss1 = i1 * kinetic_R0 * 2.1821789023599241 * exp(kinetic_E_disscath
    / A_el1);
    auto n_loss2 = i1 * (EL_h / (kinetic_sig0_el * exp(-kinetic_E_el / R /
   T_elchem)));
    auto A_el2 = -R * T_elchem / 2.0 / cst_F;
    auto n_loss3 = A_el2 * log(1.0 - (FU + CE_FU_adj));
    auto n_loss4 = A_el2 * log(1.0 - FU / L_air);
    auto n_loss5 = IC_R_MIC * i1 * 0.0001;

    auto y_Ucell = ((U_Nernst - (n_loss0 + n_loss1 + n_loss2 + n_loss3 + n_loss4
    + n_loss5)));
    auto A_el3 = AN_l_x * AN_l_y;

    auto A_i = IC_l_x * IC_l_y;

    auto QdFin_tmp_tmp = x(8) - T_ref;
    auto QdFin_tmp_tmp_tmp = T_ref * T_ref;
    auto b_QdFin_tmp_tmp_tmp = x(8) * x(8);
    auto b_QdFin_tmp_tmp = b_QdFin_tmp_tmp_tmp - QdFin_tmp_tmp_tmp;
    auto QdFin_tmp = (26604.4621 * QdFin_tmp_tmp + 1.8595 * b_QdFin_tmp_tmp) /
   1000.0;

    auto QdFin = QdFin_tmp * nH2r;

    auto QdFout_tmp = (29064.4621 * QdFin_tmp_tmp + 6.075 * b_QdFin_tmp_tmp) /
   1000.0;

    auto QdFout = QdFout_tmp * dxt_idx_1;

    auto QdAin_tmp_tmp = x(9) - T_ref;
    auto b_QdAin_tmp_tmp = x(9) * x(9) - QdFin_tmp_tmp_tmp;
    auto QdAin_tmp = (30324.4621 * QdAin_tmp_tmp + 2.468 * b_QdAin_tmp_tmp) /
   1000.0;

    auto QdAin = QdAin_tmp * nO2r;

    auto QhFe = (x(6) - x(8)) * CE_hFe * A_el3;
    auto QhAe = (x(6) - x(9)) * CE_hAe * A_el3;

    //  Radiative losses with air and fuel
    auto i2 = 1.0 / IC_e_i;
    auto dxt_idx_2 = pow(x(7), 4.0);
    auto U_Nernst1 = cst_sigma * A_el3 * (pow(x(6), 4.0) - dxt_idx_2);
    auto nO2r1 = U_Nernst1 / ((1.0 / AN_e_a + i2) - 1.0);
    auto QrF2 = U_Nernst1 / ((1.0 / CA_e_c + i2) - 1.0);
    auto i3 = nH2r * (((QdFout_tmp -241827.0) - QdFin_tmp) - (30324.4621 *
                 QdFin_tmp_tmp + 2.468 * b_QdFin_tmp_tmp) / 1000.0 / 2.0);

    //  Electrical power
    auto U_Nernst2 = u2 * y_Ucell * N_cell;

    auto dxt_idx_0 = ((((QdFin - QdFout) + QdAin) - (((QhFe + QrF2) + QhAe)
    + nO2r1))- i3) - U_Nernst2;
    auto nH2r1 = (x(8) - x(7)) * CE_hFi * A_i;
    auto QhAi = (x(9) - x(7)) * CE_hAi * A_i;

    //  Radiative heat transfert
    auto i4 = CE_e_VF * cst_sigma * (dxt_idx_2 - std::pow(CE_T_f, 4.0)) *
   CE_A_loss;
    auto dxt_idx_3 = (((nH2r1 + QhAi) + nO2r1) + QrF2) - i4;

    auto i5 = Tin1 - T_ref;
    auto U_Nernst3 = Tin1 * Tin1 - QdFin_tmp_tmp_tmp;
    auto U_Nernst4 = nH2in * ((26604.4621 * i5 + 1.8595 * U_Nernst3) /
    1000.0) + nH2Oin *((29064.4621 * i5 + 6.075 * U_Nernst3) / 1000.0);

    auto QrF3 = (nH2_idx_2_tmp * QdFin_tmp) + nH2O_idx_2_tmp * QdFout_tmp;

    auto U_Nernst5 = ((U_Nernst4 - QrF3) + (QdFout - QdFin)) + (QhFe - nH2r1);


    auto QrF4 = Tin2 - T_ref;
    auto A_el7 = Tin2 * Tin2 - QdFin_tmp_tmp_tmp;
    auto i7 = u1 * ((30324.4621 * QrF4 + 2.468 * A_el7) / 1000.0) +
    b_inflow6 *((27414.4621 * QrF4 + 2.563 * A_el7) / 1000.0);

    //  Heat out
    auto A_el8 = nO2_idx_2_tmp * QdAin_tmp + b_inflow6 * ((27414.4621 *
    QdAin_tmp_tmp + 2.563 * b_QdAin_tmp_tmp) / 1000.0);

    auto A_el9 = ((i7 - A_el8) + (0.0 - QdAin)) + (QhAe - QhAi);

    auto dx_data0 = 1.25 * dxt_idx_0;
    auto dx_data1 = 1.25 * dxt_idx_3;
    auto dx_data2 = 1.25 * U_Nernst5;
    auto dx_data3 = 1.25 * A_el9;

    dTheta(2) = dx_data0;
    dTheta(3) = dx_data1;
    dTheta(4) = dx_data2;
    dTheta(5) = dx_data3;
    dTheta(6) = x(4) - y_Ucell;
}
```