Nonlinear MPC Implementation with google-OSQP

Ali BOYALI

June 2021

1 Introduction

In this repository, we present an implementation of Nonlinear Model Predictive Control (NMPC) using a quadratic solver OSQP [8] and C++ interface google-OSQP [1]. The implementation is the bare minimum implementation and only a single Quadratic Problem (QP) solved at the each time step in the MPC loop. There is no Sequential Quadratic Programming (SQP) iteration in the minimal implementation. This is due to the fact that the longitudinal speed trajectory and a reference path are always available to the NMPC algorithm and there is no nonlinear constraint linearized within the loop. Although we use a highly nonlinear vehicle model, due to the linearization of motion equations, we do not need to use SQP.

The repository contains the following classes and modules organized under the folders.

- MPC algorithm (mpc_algorithm_scv folder): This class implements the main MPC loop and provides an interface to vehicle model, discretization, simulation, and map handling methods.
- Spline Methods (Spline folder)
- MPC SQP Problem Folder (mpc_sqp_problem)
  - Discretization class
  - LPV initialization class
  - MPC SQP Problem class
  - Simulation class
  - Smoothing splines - BSpline smoothers

  - Interpolating splines - BSpline interpolators (Can be used to compute the curvature and from the x and y trajectory information),
  - Pre-Conditioned Conjugate Gradient based interpolating splines with line (Linear) and spline interpolation methods. This spline can also be used to interpolate a single point.

- Vehicle Base Class (Vehicle folder)
  - Kinematic vehicle model with sideslip angle,
  - cppAD codegen codes [6],
- Utils folder: Contains the helper functions for the Eigen matrix library and some other small-scale helper files such as norm computation, tangent and normal vector, etc. We implemented numpy like functionalities for the Eigen library under this folder.

In the following section, we give the details of the classes used to implement a NMPC algoritm.

2 Model Equations and Initialization

2.1 Vehicle Model - Kinematic Vehicle Model with Side-Slip Angle

The kinematic vehicle model with the side-slip angle is used for the computations. This model approximates the dynamical vehicle equations. The details of the model and its comparison with the dynamical vehicle models are presented in [5]. The authors recommend to use the kinematic vehicle model with the side-slip formulation in MPC predictions.

The kinematic vehicle model equations have two input as the controls are given in Eq. (1);

$$\begin{aligned} \dot{x} &= v\cos(\psi+\beta) \\ \dot{y} &= v\sin(\psi+\beta) \\ \dot{\psi} &= \frac{v}{l_r}\sin(\beta) \\ \dot{v} &= a \\ \beta &= \tan^{-1}\left(\frac{l_r}{l_f+l_r}\tan\left(\delta_f\right)\right) \end{aligned} \tag{1}$$

where $[x,\, y,\, \Psi]$ are the global coordinates and the heading angle of the vehicle respectively, v represents the longitudinal velocity. Using this model, we can derive the differential equations for the error model described in the Frenet frame.

$$
\begin{aligned}
\dot{x} &= v\cos(\psi + \beta) \\
\dot{y} &= v\sin(\psi + \beta) \\
\dot{\psi} &= \frac{v}{l_r}\sin(\beta) \\
\dot{s} &= \frac{v}{1-\kappa e_y}\cos(e_\psi + \beta) \\
\dot{e_\psi} &= \dot{\psi} - \kappa\dot{s} \\
\dot{e_y} &= v\sin(e_\psi + \beta) \\
\dot{v} &= a \\
\dot{\delta} &= u_\delta, \leftarrow \text{direct input, if we use a steering model;} \rightarrow (-\frac{1}{\tau}\delta + \frac{1}{\tau}u_\delta) \\
\beta &= \tan^{-1}\left(\frac{l_r}{l_f + l_r}\tan\left(\delta_f\right)\right)
\end{aligned}
\tag{2}
$$

The controls in the models are the acceleration a and steering rate $u_\delta = \dot{\delta}$ inputs. Although the world coordinates, $[x$ and $y]$ are not used in the optimization computations, we keep these states in the equations for debugging and visualization purpose. The optimization computations are based on reducing the error states to zero. The curvature κ is the only reference entering in the model. In the equations, the steering model is not included. However, it is easy to add a first order model to the steering differential equation by changing u_δ to $-\frac{1}{\tau}\delta + \frac{1}{\tau}u_\delta$.

2.2 Linear Parameter Varying (LPV) Model and using for the trajectory initialization.

The most important aspect in the NMPC and nonlinear optimization is the initial solution. If an adequate initial feasible trajectory solution is not provided to the nonlinear solvers, there is no way to solve the resulting problem. In the literature, an initial trajectory is computed by interpolating the terminal points at the interim discretization points. This approach provides a rough estimate to an initial solution. A better approach to compute the initial trajectories is the use of feedback solutions that we used in the implementation.

To be able to compute a feedback solution for a nonlinear model, we can use the LPV parametrization for the model. In the kinematic vehicle models (Eqs.

1, 2) we have only one varying parameter; longitudinal speed (v).

If we linearize the error model using only the error states [e_y, e_ψ, v and δ] of the equations, we can obtain the state transition matrix $A = \frac{df}{dx}$ and the input matrix B. The computation of B is trivial and contains only ones, therefore we omit its representation here.

$$\frac{df}{dx} = \begin{bmatrix} 0 & v\cos(e_\psi + \beta) & \sin(e_\psi + \beta) & v\cos(e_\psi + \beta)\frac{d\beta}{d\delta} \\ -\frac{\kappa^2}{(1-\kappa)^2} v\cos(e_\psi + \beta) & \frac{\kappa}{1-\kappa} v\sin(e_\psi + \beta) & \frac{\sin\beta}{l_r} - \frac{\kappa}{1-\kappa} v\cos(e_\psi + \beta) & \frac{v\cos\beta}{l_r} + \frac{\kappa}{1-\kappa} v\sin(e_\psi + \beta)\frac{d\beta}{d\delta} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{\tau} \end{bmatrix}$$

If we can parametrize the transition matrix $A = A(\theta)$, we can compute the parametrized Lyapunov matrices $P(\theta)$ on a state grid. Using the parameterized Lyapunov functions, it is easy to obtain stabilizing controllers given the measured vehicle states. We first start with defining parameters of the state transition matrix $A(\theta)$. As seen in the transition matrix, we can choose each nonlinear term in the state transition matrix as a parameter. As an example, the first parameter would be $\theta_1 = v\cos(e_\psi + \beta)$. There are seven parameters in the state transition matrix.

We can use continuous differential equations for continuous time state feedback coefficient computation, which is easier than the discrete version [3, 4]. The Lyapunov equations for the stability and performance of the continuous time model equations can be expressed as;

$$\begin{aligned} &V(x(\theta(t))) = x(\theta(t))^T P(\theta) x(\theta(t)) \\ &\dot{V}(x(\theta(t))) <= -\rho_c V(x(\theta(t))) - x(\theta(t))^T Q x(\theta(t)) - u(\theta(t))^T R u(\theta(t)) \end{aligned} \tag{3}$$

where Q and R are the performance weight matrices and ρ_c is the contraction rate of the trajectories. The parameter variations, $\dot{\theta}$ is taken into account in the Lyapunov energy-like function. We create the grid of error states along with the curvature with a low resolution and computed the parametrized Lyapunov matrices $P(\theta)$. The controllers can be computed online by first evaluating the $\theta(x)$ equations than computing the results of the following expression;

$$K(\theta) = Y(\theta) X^{-1}(\theta). \tag{4}$$

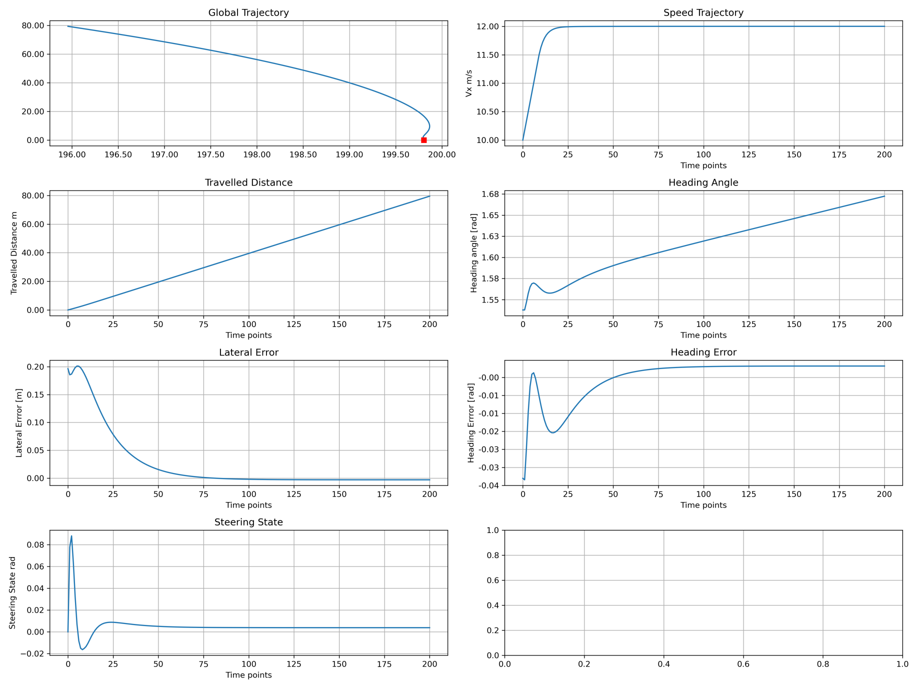
where the parametric matrices X and Y can be computed using the following LMI;

$$
\begin{aligned}
&\min_{x(r),Y(r),X_{\text{minin}}} \quad -\log\det X_{\min} \\
&\text{s.t.} \begin{pmatrix} A(r)X(r)+B(r)Y(r)+(A(r)X(r)+B(r)Y(r))^\top - \frac{d}{dt}[X(r)] & \left((Q+\epsilon)^{1/2}X(r)\right)^\top & \left(R^{1/2}Y(r)\right)^\top \\ * & -I & 0 \\ & * & -I \end{pmatrix} \leq \ 0 \\
&X_{\min} \leq X(r), \\
&\forall r \in \mathcal{Z}_r, \dot{r} \in \mathcal{R}(r).
\end{aligned} \tag{5}
$$

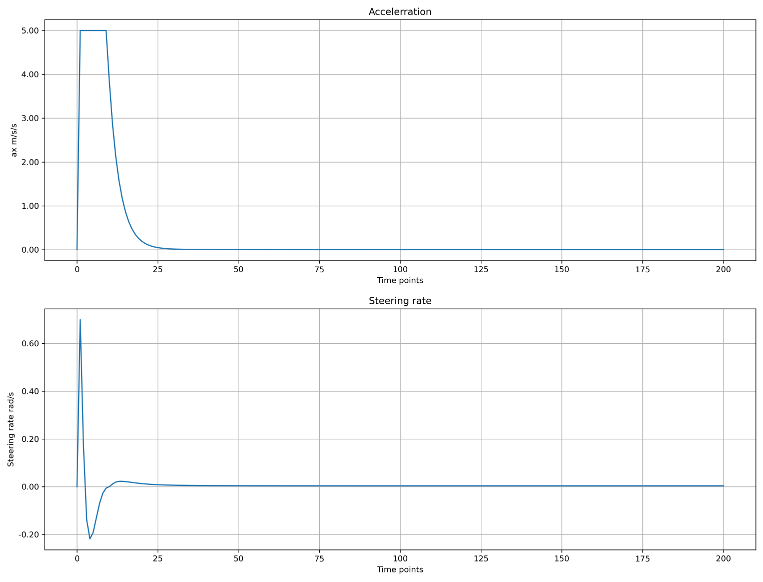
These matrices can be efficiently computed for each point on the state grid Using Matlab Yalmip toolbox, or CVXPY in Python, and LMI can be solved efficiently. The reference grid is represented by r in the LMI equations. The parametrized Lyapunov matrices $P(\theta)$ can also be used to compute the terminal weights for the Robust NMPC applications.

In the Nonlinear MPC implementation, we first compute a the steering and acceleration inputs using the state feedback coefficients, and integrate the model equations one-step. This process are repeated for a number of prediction steps to obtain the initial state and control trajectories. As can be seen from the vehicle model equations, at each time step integration, the curvature at the current reference is needed. We implemented a scalar interpolator class, which takes the reference arc-length and curvature vectors as an argument with the current arc-length distance s_0 and interpolates the curvature at this location.

We provide examples of the state and control initialization in the figures (Fig. 1a, Fig. 1b).



(a)



(b)

Figure 1: (a) State trajectories. (b) Control trajectories.

As we can see from the state trajectories, the initial lateral and heading errors go zero while the vehicle increases its initial speed from 10 [m/s] to 12 [m/s].

3 Linearization and Discretization

We use the fundamental matrix solution approach for the discretization of the linearized differential equation of the form given by;

$$\dot{\boldsymbol{x}}(t)=A(t)\boldsymbol{x}(t)+B(t)\boldsymbol{u}(t)+\boldsymbol{z}(t), \quad \boldsymbol{x}(0)=\boldsymbol{x}_0, \quad t\in[0,t_f] \tag{6}$$

where $z(t)=f_0(x_0,u_0)-A(t)x_0-B(t)u_0$ represents the residuals in the Taylor expansion. The flow (solution) curves of the linear differential equations can be obtained using the fundamental matrix solution;

$$\boldsymbol{x}(t)=\Phi\left(t,t_k\right)\boldsymbol{x}\left(t_k\right)+\int_{t_k}^{t}\Phi(t,\xi)B(\xi)\boldsymbol{u}(\xi)\mathrm{d}\xi+\int_{t_k}^{t}\Phi(t,\xi)\boldsymbol{z}(\xi)\mathrm{d}\xi \tag{7}$$

where $\Phi(t,\xi)$ is the fundamental matrix of the solution with the following properties;

$$\frac{\mathrm{d}}{\mathrm{d}t}\Phi\left(t,t_0\right)=A(t)\Phi\left(t,t_0\right), \quad \Phi\left(t_0,t_0\right)=I$$

and,

$$\text{For all } t,t_0,t_1\in\mathbb{R}, \Phi\left(t,t_0\right)=\Phi\left(t,t_1\right)\Phi\left(t_1,t_0\right).$$

For the discrete system;

$$\boldsymbol{x}_{(k+1)}=A_{|k}\boldsymbol{x}_{(k)}+B_{|k}\boldsymbol{u}_{(k)}+\boldsymbol{z}_{|k}, \quad k=0,\ldots,N-2$$

We can compute the system matrices by the following integral equations;

$$
\begin{aligned}
A_{|k} &= \Phi\left(t_{k+1}, t_k\right), \\
B_{|k} &= A_{|k} \int_{t_k}^{t_{k+1}} \Phi\left(\xi, t_k\right)^{-1} B(\xi) \mathrm{d}\xi \\
\boldsymbol{z}_{|k} &= A_{|k} \int_{t_k}^{t_{k+1}} \Phi\left(\xi, t_k\right)^{-1} \boldsymbol{z}(\xi) \mathrm{d}\xi
\end{aligned}
\tag{8}
$$

We compute these integrals using Runge-Kutta-45 method which is available under the Boost integration library.

4 QP and SQP Formulation of NMPC

4.1 Motion Equations

The nonlinear MPC problems are solved either by interior point or sequential quadratic optimization methods. In the latter one, the system and constraint equations are linearized at each iteration and solved by quadratic programming sequentially. Since we already have linearized system equations, and we have access to open-source fast quadratic problem solvers, we implemented the NMPC algorithm using SQP method with a quadratic program solver. However, in its bare minimum implementation, we do not use nonlinear constraints in our formulation. The only nonlinearity comes with the model equations which are already linearized. In this section, we first describe the QP formulation in this section and give some details about the SQP formulation in the subsequent sections.

At the heart of QP problem lie the state and input concatenation for a given time horizon in discrete settings. This form of the formulation is called as the "dense-form" QP. After defining concatenated state and control vectors,

$$
\boldsymbol{X} := \begin{bmatrix} \boldsymbol{x}_{(0)} \\ \boldsymbol{x}_{(1)} \\ \vdots \\ \boldsymbol{x}_{(N-1)} \end{bmatrix} \in \mathbb{R}^{nN}, \quad \boldsymbol{U} := \begin{bmatrix} 0 \\ \boldsymbol{u}_{(0)} \\ \vdots \\ \boldsymbol{u}_{(N-2)} \end{bmatrix} \in \mathbb{R}^{mN}
$$

yields the prediction model of the equations as;

$$\boldsymbol{X} = \begin{bmatrix} I & 0 & 0 & \cdots & 0 \\ A_{|0} & 0 & 0 & \cdots & 0 \\ 0 & A_{|1} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & A_{|N-2} & 0 \end{bmatrix} \boldsymbol{X} + \begin{bmatrix} 0 & \cdots & 0 & 0 \\ B_{|0} & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & 0 \\ 0 & & 0 & 0 \\ 0 & \cdots & B_{|N-2} & 0 \end{bmatrix} \boldsymbol{U} + \begin{bmatrix} 0 \\ \boldsymbol{z}_{|0} \\ \boldsymbol{z}_{|1} \\ \vdots \\ z_{|N-2} \end{bmatrix}$$

where N represents the number of time-steps in the prediction horizon.

Optimization solvers use matrix iterations, therefore, the matrices formed by the solvers should not be ill-conditioned. One way to prevent the ill-conditioned matrices is to bring all the states and controls to the same range by scaling. The last equation given above can be expressed in a more compact form as;

$$(A - I)X + BU + Z = 0.$$

Choosing a scaling interval as [-1, 1], we can apply a linear scaling equation for each state and control in this equation. Define;

$$x = S_x\hat{x} + C_x$$

and

$$u = S_u\hat{u} + C_u$$

so that the scaled equations can take the form;

$$\begin{bmatrix} -S_x & 0 & 0 & \cdots & 0 \\ A_{|0}S_x & -S_x & 0 & \cdots & 0 \\ 0 & A_{|1}S_x & -S_x & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & A_{|N-2}S_x & -S_x \end{bmatrix} \hat{\boldsymbol{X}} + \begin{bmatrix} 0 & \cdots & 0 & 0 \\ B_{|0}S_u & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & 0 \\ 0 & & 0 & 0 \\ 0 & \cdots & B_{|N-2}S_u & 0 \end{bmatrix} \hat{\boldsymbol{U}} + \hat{R} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where the expression for the residuals takes the form

$$\hat{R} = \begin{bmatrix} S_x\hat{x_0} \\ z_{|0} + A_{|0}C_x + B_{|0}C_u - C_x \\ \\ z_{|N-2} + A_{|N-2}C_x + B_{|N-2}C_u - C_x \end{bmatrix}$$

In the implementation, the control vector U is treated as constant within the integration interval which is called as Zero-Order Hold (ZOH) integration. In the Python twin of this repository applies First Order Hold (FOH). We did not observe any tangible difference between these implementations. The ZOH is simpler and less complicated then the FOH implementation.

The discretisized compact form of state dynamics are used to formulate the motion equality constraints in the optimization problem. In the following section, we define the NMPC in the QP framework.

4.2 (Nonlinear) MPC Problem Definition

As we use the google-OSQP solver, we define the problem with the QP program terminology that OSQP documentation uses. A general QP program is described by the following generic problem structure;

$$\begin{aligned} \min_x \quad & \frac{1}{2}x^T P x \\ \text{s.t.} \quad & l < Ax < u \end{aligned} \tag{9}$$

As our aim is to design controllers that track a reference, we can modify the generic problem definition as;

$$\begin{aligned} \min_x \quad & \frac{1}{2}(x - x_r)^T P (x - x_r) \\ \text{s.t.} \quad & l < A_c x < u \end{aligned} \tag{10}$$

so that we can include the desired states as a reference input.

In Eq. 11, l and u are lower and upper bounds for the constraint equations and x is a vector of decision variables. In our applications, the decision variables are the states and controls. To remove the ambiguity, we use $x_d = [x_{0:N}, u_{0:N}]$ as the decision variables and A_c for the constraint equality and inequality matrix from this point onward. The lengths of the control and states along the prediction horizon are the same in our codes, namely, we compute an additional control at the end of the prediction horizon. We keep this computations in the implementation so that we can easily switch from ZOH computation to FOH computations, where u_N is used to interpolate the inputs withing the integration boundries.

After some algebraic manipulation and re-organizing the reference and decision variables in Eq. 11, we arrive the the final formulation of the QP problem structure we used in the implementation given in the following equation.

$$\begin{aligned} \min_x \quad & \frac{1}{2}x^T P x + qx \\ \text{s.t.} \quad & l < A_c x < u \end{aligned} \tag{11}$$

where the optimization vector $q = Px_r$. The constraint matrix A_c is a concatenation of equality and inequality constraints.

In the NMPC problem, we define the following problem structure in the dense QP form.

$$\begin{aligned} \min_{x=[x,u,\Delta u]} \quad & x_N^T Q_N x_N + \sum_{k=0}^{N-1} \left(x^T Q x + u^T R u + \Delta u R_j \Delta u \right) = x^T P x + q x \\ \text{s.t.} \quad & l < A_c x < u \end{aligned} \tag{12}$$

The decision variables $[x,\ u,\ u]$ are represented by a single decision variable x in the QP problem. In this case, the cost matrix P and the optimization vector q take the forms of $P = block\_diag([P_x,\ P_u,\ P_{\Delta u}])$ where ;

$$P_x = \begin{bmatrix} Q & 0 & 0 & \cdots & 0 \\ 0 & Q & 0 & \cdots & 0 \\ 0 & 0 & & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & Q_N \end{bmatrix} \quad P_u = \begin{bmatrix} R & 0 & 0 & \cdots & 0 \\ 0 & R & 0 & \cdots & 0 \\ 0 & 0 & & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & R \end{bmatrix}_{N-1} \quad P_{\Delta u} = \begin{bmatrix} R_j & 0 & 0 & \cdots & 0 \\ 0 & R_j & 0 & \cdots & 0 \\ 0 & 0 & & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & R_j \end{bmatrix}_{N-2}$$

The constraint matrix $A_c = \begin{bmatrix} A_{equality} \\ A_{inequality} \end{bmatrix}$ where ;

$$A_{equality} = \begin{bmatrix} (A-I) & B & 0 \\ 0 & I_u & I \end{bmatrix} \begin{bmatrix} X \\ U \\ \Delta U \end{bmatrix}$$

and

$$I_u = \begin{bmatrix} I & -I & 0 & \cdots & 0 \\ 0 & I & -I & \cdots & 0 \\ 0 & 0 & & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & I & -I \end{bmatrix}_{N-2}$$

The inequality matrix is an identify matrix that picks the decision variables. The upper and lower bounds for the decision variables are stored in a yaml file.

References

[1] Google. "google-OSQP". In: 2021. URL: `https://github.com/google/osqp-cpp`.

[2] Sébastien Gros et al. "From linear to nonlinear MPC: bridging the gap via the real-time iteration". In: *International Journal of Control* 93.1 (2020), pp. 62–80.

[3] Johannes Köhler, Matthias A Müller, and Frank Allgöwer. “A nonlinear model predictive control framework using reference generic terminal ingredients”. In: *IEEE Transactions on Automatic Control* 65.8 (2019), pp. 3576–3583.

[4] Johannes Köhler et al. “A computationally efficient robust model predictive control framework for uncertain nonlinear systems”. In: *IEEE Transactions on Automatic Control* (2020).

[5] Jason Kong et al. “Kinematic and dynamic vehicle models for autonomous driving control design”. In: *2015 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2015, pp. 1094–1099.

[6] João Rui Leal. “Critical Software”. In: 2021. URL: `https://github.com/joaoleal/CppADCodeGen`.

[7] Danylo Malyuta et al. “Discretization performance and accuracy analysis for the rocket powered descent guidance problem”. In: *AIAA Scitech 2019 Forum*. 2019, p. 0925.

[8] Bartolomeo Stellato et al. “OSQP: An operator splitting solver for quadratic programs”. In: *Mathematical Programming Computation* (2020), pp. 1–36.