Model Predictive Control - EL2700

On-Orbit Rendezvous - Revisited
A Nonlinear MPC Approach

2021

Automatic Control
School of Electrical Engineering and Computer Science
Kungliga Tekniska Högskolan

Pedro Roque

Introduction

And finally, here we are. Up to now you have learned about system discretization and how the poles of the system influence its behavior; you have learned about continuous time optimization and how to create Linear Programming and Quadratic Programming optimization problems; you learned about infinite horizon LQR controllers and how terminal weights influence the system behavior; you learned about Linear and Nonlinear MPC and how to tune these powerful controllers for optimal performance. Even with all the head-scratching around Python and CasADi, you learned how to formalize these methods in a general form that you could implement with open-source tools, and hopefully be useful for you in the future!

So now, we want to show you how powerful these tools are. In this project, we will solve an on-orbit rendezvous problem using Nonlinear Model Predictive Control, which you will have the chance to see live in the lab session (more information on time slots will come later). The goal is to rendezvous with Honey, an Astrobee that is broken and needs to be de-orbitted. Honey is currently at a fixed position and with constant angular velocity, waiting for some help from her fellow Astrobee, Bumble. The controller that you create will then be used on the NASA Astrobee simulator - the same simulator used by all space robotics researchers before the final deployment on the Astrobee aboard the International Space Station, see Fig. 1 - and its performance compared with your peers. The best team will have a special prize!

To help you, we provide a code-base that we suggest you to start from, and slowly build your way through the tasks we propose here. Follow each step carefully and make sure your knowledge is solid through each step, since this will help you a lot during the debugging stage.

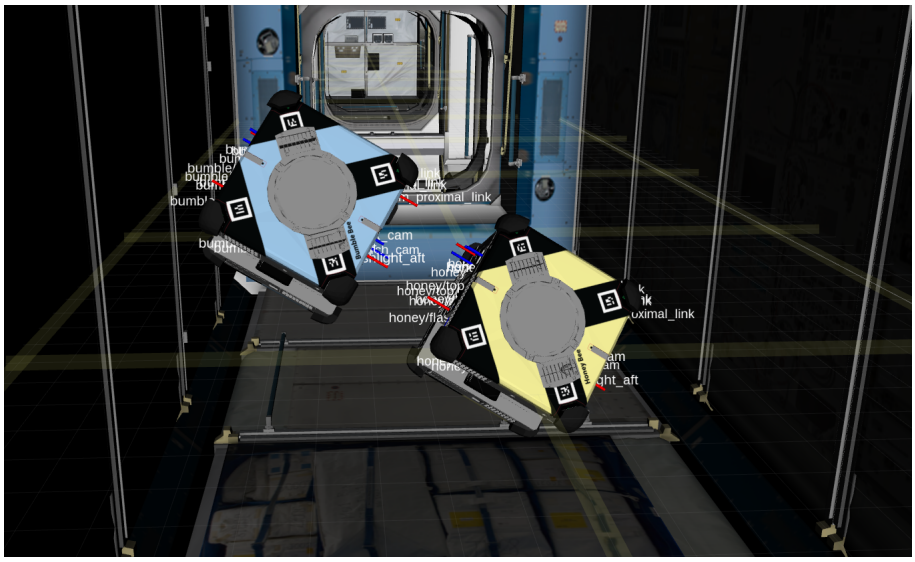

Figure 1: Bumble performing a rendezvous maneuver with Honey, in the simulated International Space Station. The simulation uses ROS (Robotics Operating System) and the NASA Astrobee simulator.

Astrobee Orbital Dynamics

The orbital dynamics of the Astrobee follow the same dynamics used in Assignment 3, except for one component - this time around, we will use quaternions to represent the attitude of the free-flyer. In case you are not comfortable about working with quaternions, worry not - we provide the necessary equations for you. In case you want to learn a little more, please check [4] or [1]. The main reason for this shift is Gimbal lock, a singularity that is present in the Euler angle representation of the attitude of a spacecraft, which can create problems in space applications.

The dynamics model of an Astrobee can be written with the Newton-Euler equations

$$\dot{\mathbf{p}} = \mathbf{v} \tag{1a}$$

$$\dot{\mathbf{v}} = \frac{1}{m} R(\mathbf{q})\mathbf{f}, \tag{1b}$$

$$\dot{\mathbf{q}} = \frac{1}{2}\Omega(\mathbf{q})\boldsymbol{\omega}, \text{ and} \tag{1c}$$

$$\dot{\boldsymbol{\omega}} = J^{-1}(\mathbf{t} - \boldsymbol{\omega} \times J\boldsymbol{\omega}), \tag{1d}$$

where $\mathbf{p} \in \mathbb{R}^3$ is the position, $\mathbf{v} \in \mathbb{R}^3$ the linear velocity, $\mathbf{q} \in \mathbb{SO}(3)$ the unit quaternion, and $\boldsymbol{\omega} \in \mathbb{R}^3$ the angular velocity in the body-frame. In particular, the unit quaternion is assumed to be defined as $\mathbf{q} = [q_x, q_y, q_z, q_w]$, with the scalar component being q_w. The rotation matrix $R(\mathbf{q})$ and attitude jacobian $\Omega(\mathbf{q})$ are defined as

$$R(\mathbf{q}) = \begin{bmatrix} 1-2q_y^2-2q_z^2 & 2q_xq_y-2q_zq_w & 2q_xq_z+2q_yq_w \\ 2q_xq_y+2q_zq_w & 1-2q_x^2-2q_z^2 & 2q_yq_z-2q_xq_w \\ 2q_xq_z-2q_yq_w & 2q_yq_z+2q_xq_w & 1-2q_x^2-2q_y^2 \end{bmatrix} \quad \Omega(\mathbf{q}) = \begin{bmatrix} q_w & -q_z & q_y \\ q_z & q_w & -q_x \\ -q_y & q_x & q_w \\ -q_x & -q_y & -q_z \end{bmatrix} \tag{2}$$

Lastly, $m = 9.6$kg represents the mass of the Astrobee, and $J = \text{diag}([0.1534, 0.1427, 0.1623])$ its positive definite diagonal inertia matrix. Lastly, the system input is a force $\mathbf{f} \in \mathbb{R}^3$ and torque $\mathbf{t} \in \mathbb{R}^3$. Accordingly, let the state vector $\mathbf{x} \in \mathbb{R}^9 \times \mathbb{SO}(3)$ and control input vector $\mathbf{u} \in \mathbb{R}^6$, defined as $\mathbf{x} = \begin{bmatrix} \mathbf{p} & \mathbf{v} & \mathbf{q} & \boldsymbol{\omega} \end{bmatrix}^T$ and $\mathbf{u} = \begin{bmatrix} \mathbf{f} & \mathbf{t} \end{bmatrix}^T$.

Software Requirements

In this lab project, and since we will have to run all your code in the NASA Astrobee simulator, our code should be compatible with Python 2.7. The main difference with respect to Python 3.6 will be on how matrix multiplications are performed - for Python 3.6, we could use A @ b for the inner product of A and b. In Python 2, this functionality is done with `np.dot(A, b)` or `ca.mtimes(A, b)`, when dealing with NumPy or CasADi variables, respectively. As long as you are careful with this difference, your code should be compatible with Python 2.7, and you can still use Python 3.6 or above to simulate your system.

The project will not require any extra dependencies, when compared to Assignment 4, so the same script `install_deps.py` can also be used to ensure that your systems contains all dependencies.

Design Task

And so, we start. The "entry point" for the project code is the script `project.py`. At the beginning of this script, you will find

- `trajectory_quat` variable: complete this variable with the location of the `trajectory_quat.txt` file in your system, which should be inside the `Dataset` folder, bundled with the project code;

- `tuning_file_path` variable: complete this variable with the location of the `tuning.yaml` file in your system, which should be inside the `Project Assignment` folder.

Once these are set, we can for our first task.

Q1: Implement the dynamics described in(1) in the function `astrobee_dynamics_quat` inside the `Astrobee` class. Note that the matrices $R(\mathbf{q})$ and $\Omega(\mathbf{q})$ are implemented in the `util.py` script as `r_mat_quat` and `xi_mat`, respectively.

If your implementation was successful, the function `abee.test_dynamics()` should let the code flow without interruptions. Otherwise, it will stop the code with an error message.

Q2: At this point we are ready to do a static setpoint tracking test. The Nonlinear MPC problem is defined in (3),

$$\min_{u_{t+\cdot|t}} \sum_{k=0}^{N-1} \mathbf{e}_{t+k|t}^T Q \mathbf{e}_{t+k|t} + \mathbf{u}_{t+k|t}^T R \mathbf{u}_{t+k|t} + \mathbf{e}_{t+N|t}^T P \mathbf{e}_{t+N|t} \tag{3a}$$

$$\text{subject to: } \mathbf{x}_{t+k+1|t} = f(\mathbf{x}_{t+k|t}, \mathbf{u}_{t+k|t}) \tag{3b}$$

$$\mathbf{e}_{t+k|t} = \Gamma(\mathbf{x}_{t+k|t}, \mathbf{x}_{t+k|t}^{ref}) \tag{3c}$$

$$\mathbf{x}_{t+k|t} \in \mathcal{X} \tag{3d}$$

$$\mathbf{u}_{t+k|t} \in \mathcal{U} \tag{3e}$$

$$\mathbf{x}_{t|t} = \mathbf{x}_t \tag{3f}$$

where $\mathbf{e}$ is the error of the state with respect to a reference $\mathbf{x}^{ref}$, calculated with a function $\Gamma(\mathbf{x}, \mathbf{x}^{ref})$, and where the function $f(\cdot, \cdot)$ corresponds to the discretized dynamics of (1). The error is calculated in `set_cost_functions_quat` in the `MPC` class, so you may assume that the function $\Gamma(\cdot, \cdot)$ is taken care of.

In this case, we will use a simplified version of NMPC, which does not contain terminal constraints, but does contain a terminal cost - more information can be seen in [3]. In short, the NMPC problem is can be shown to be stable considering only a terminal cost $P = \lambda Q, \lambda > 1$, that is, a scaled up version of the state cost. The associated terminal set will be dependent on how large λ is, noting that a too large λ will also affect the performance of the NMPC. Your cost matrices Q and R will be diagonal matrices with appropriate dimensions.

For the simulation, the functions `abee.get_static_setpoint()` and `abee.get_initial_pose()` are set with good reference points for our realistic simulator. Moreover, the function `u_lim, x_lim = abee.get_limits()` is already set with the correct control and state limits for the MPC. Given these, run the simulation for the desired setpoint x_d. Change the tuning configuration on your `tuning.yaml` file to ensure that the performance of the system is affected accordingly.

Clarification: Make sure to try different tuning settings, and consider being conservative on how fast your system moves. Having different tuning parameters for different performance settings is, in general, a good approach.

Q3: Now, we will modify our MPC so that we can track a time-varying reference (such as the rendezvous motion of a satellite). Inside the MPC class, update the parameter `x_ref` for the tracking scenario. Note that this variable shall contain $13 \times (N+1)$ variables, since our full-state dimension is 13 and our horizon is N-steps - although not necessarily shaped like a matrix. Then, in the optimization loop (particularly, in line 123), implement the necessary logic to obtain the desired reference at each time-step. **NOTE:** the trajectory used was recorded with a sampling period of $10Hz$. For this and the next question, we recommend you to keep the same sampling time.

At this point, you should be able to run the trajectory tracking MPC in `tracking_ctl` and observe that your trajectory tracking error converges to 0 on all state variables! Only one last step to go before we are ready!

In question **Q3** we use a previously recorded trajectory of Honey in the simulator, and we feed N-steps of this trajectory to our controller. However, in a real-system, we don't know *a-priori* where our target will be in the future, so we must estimate its state evolution. Different and complex techniques can be used for this [2]. In this project, however, we suggest a simple one: first-order forward propagation. With a first-order propagation, we can forward-estimate the position and attitude of the target, based on the assumption of constant velocity. For instance, we can use

$$\dot{\mathbf{p}} = \bar{\mathbf{v}} \tag{4}$$

$$\dot{\mathbf{q}} = \frac{1}{2}\Omega(\mathbf{q})\bar{\boldsymbol{\omega}}, , \tag{5}$$

for some constant linear $\bar{\mathbf{v}}$ and angular $\bar{\boldsymbol{\omega}}$ velocities, and then perform a first-order integration with

$$\mathbf{p}_{k+1} = \mathbf{p}_k + \dot{\mathbf{p}} \cdot h \tag{6}$$

$$\mathbf{q}_{k+1} = \mathbf{q}_k + \dot{\mathbf{q}} \cdot h. \tag{7}$$

where h is the sampling time. In particular, for the quaternion integration, we must preserve the unit-norm. To this end, after the integration, it is important to re-normalize the quaternion with

$$\mathbf{q}_{k+1} = \frac{\mathbf{q}_{k+1}}{\|\mathbf{q}_{k+1}\|}. \tag{8}$$

At this point, we have an estimate of how Honey's state will evolve through time. However, for a good rendezvous, we should approach Honey at a fixed radius distance, and follow its motion afterwards. Let us take a mild assumption that Honey's tumble is rather slow and that we are sufficiently close to Honey such that a point $\mathbf{p}_B$ moves with velocity $\mathbf{v} \approx 0$. Then, for a given position $\mathbf{p}$ and attitude $R(\mathbf{q})$ of Honey, we can calculate a desired position for Bumble to rendezvous with

$$\mathbf{p}_B = \mathbf{p} + R_{[:,0]}(\mathbf{q}) \cdot r \tag{9}$$

where $R_{[:,0]}(\mathbf{q})$ is the first column of the rotation matrix of $\mathbf{q}$ and for a given distance $r \in \mathbb{R} > 0.3$m, assuming that we want to dock at a desired relative point $[r, 0, 0]$ in the body frame of Honey. This rendezvous point is depicted in Figure 1, with $r = 0.5$[m].

Q4: Implemented the forward propagation law in (4)-(9) on the function `forward_propagate` inside the Astrobee class. The output of this function should be a concatenation of $[\mathbf{p}_B, \mathbf{v}, \mathbf{q}, \boldsymbol{\omega}]$ for `npoints` that correspond to $N+1$ points of the prediction horizon - that is, $\mathbf{x}^{ref}$ should be 13×`npoints`. If your implementation was successful, the function `abee.test_forward_propagation()` will not through any error. **NOTE:** As in **Q3**, we recommend you to keep the sampling time of $10Hz$.

Congratulations!

Your project is now complete! For details on the competition and improvements on your algorithm, keep going.

Competition

The final lab session will be a (friendly!) competition among all your peers. We will run your algorithm in the Astrobee simulator and evaluate its performance.

One important factor is that, much alike what happens for the real tests on the space station, after code-freeze (that is, your deadline), you won't be able to submit changes to your code. This means that all tuning parameters that were investigated are now frozen and you must only select from that pool during run-time, and based on the perceived performance.

Regarding the performance metrics, more information will follow in the next days...

Improvements

At this point you should have a basic-working Nonlinear MPC for trajectory tracking. However, if you really want to win this competition, we suggest you to improve the solution considering:

1. Computational Time: try to squeeze as much as possible from the solver options. CasADi as extended documentation for the solver options, and you should look for information on this on GitHub as well. Try different solvers, such as 'sqpmethod' and tune their parameters to minimize your solution time;

2. Noise: Take advantage of the simple simulation environment that is provided to modify it and include noise in the state measurements. Tune your controller against reasonable noise values to make sure that you are robust, up to some degree, to random noise in the system;

3. Model Mismatch: try to modify the model used for simulation vs. the model used in the MPC controller (you should focus on mass and inertia mismatch), and how that will affect the performance of your controller;

4. Available Parameters: make sure to have a diversified set of parameters available for your run. You should be able to identify, online, which parameter setting you want for your second run, in case you want to improve your performance, knowing that you cannot change weights during the lab session. Code freeze means code freeze!

5. Better Estimation: improve the assumption that $\mathbf{v} \approx 0$ in Q4 for a more accurate representation.

To complete this design project, you should upload a zip file containing the provided code, properly completed. The task grading is based on a successful run of `project.py`, which should provide all the results. Post all your questions on the Slack workspace `mpc-el2700-2021.slack.com`.

Good Luck!

References

[1] Quaternions and 3d rotation, explained interactively. `https://www.youtube.com/watch?v=zjMuIxRvygQ`. Accessed: 2021-09-28.

[2] Keenan Albee, Charles Oestreich, Caroline Specht, Antonio Terán Espinoza, Jessica Todd, Ian Hokaj, Roberto Lampariello, and Richard Linares. A robust observation, planning, and control pipeline for autonomous rendezvous with tumbling targets. *Frontiers in Robotics and AI*, 8:234, 2021.

[3] Andrea Boccia, Lars Grüne, and Karl Worthmann. Stability and feasibility of state constrained mpc without stabilizing terminal constraints. *Systems & Control Letters*, 72:14–21, 2014.

[4] Nikolas Trawny and Stergios I Roumeliotis. Indirect kalman filter for 3d attitude estimation. *University of Minnesota, Dept. of Comp. Sci. & Eng., Tech. Rep*, 2:2005, 2005.