coco_rocket_lander_npross

July 7, 2023

1 Computational Control: Rocket Lander Project

Naoki Sean Pross, ETH Zürich, spring semester 2023

1.1 Installation

The cell below installs the latest version of this software.

[1]:
```
# %load_ext autoreload
# %autoreload all

!apt install -y swig
%pip install --upgrade pip
%pip install poetry
%pip install 'git+https://gitlab.ethz.ch/bsaverio/coco-project.git@main'
%pip install 'git+https://github.com/naopross/coco-project.git@master'
```

[2]:
```
from coco_rocket_lander_npross import notebook, algorithms, simulator
notebook.video_width = 400
```

[3]:
```
import warnings
warnings.simplefilter("ignore", UserWarning)
```

1.2 System Dynamics and Model

We consider the nonlinear dynamics for a rocket modelled as a nonrelativistic point mass with a state variable $z = [x\ y\ \dot{x}\ \dot{y}\ \theta\ \dot{\theta}\ ]^T$ and input $\bar{u} = [F_E\ F_S\ \varphi]^T$. The continuous time state dyamics are described through Newtonian mechanics and will be denoted simply by $\dot{z} = f(z, \bar{u})$.

The system inputs $\bar{u}$ are constrainted by the physics of the actuators, hence they must lie between a minimum and maximum value. For convinience, the input values are normalized and we work with $u \in [0, 1] \times [-1, 1]^2$ and to obtain the real inputs $\bar{u}$, we compute them using a scaling matrix

$$\bar{u} = \begin{bmatrix} F_{E,\max} & & \\ & F_{S,\max} & \\ & & \varphi_{\max} \end{bmatrix} u = \gamma u.$$

The constraint conditions can then be written as a linear system of inequalities $G_u u \leq g_u$. Similarly, we artificially constrain the system state in a subspace that is consiered "safe" by writing $G_z z \leq g_z$.

1.2.1 Linearized Discrete-Time Model

We consider a discrete linear time-invariant state space model obtained by linearizing the state dynamics at the stationary point $z_s = 0$ and $u_s = [mg\ 0\ 0]^T$ where m is the mass of the rocket and g the gravitational accelleration. The continuous LTI system dynamics are then described by the jacobians

$$A_c = \left.\frac{\partial f}{\partial x}\right|_{x_s} \quad \text{and} \quad B_c = \left.\frac{\partial f}{\partial u}\right|_{u_s}.$$

The continous time dynamics are then discretized with a sampling time of T:

$$A = e^{A_c T} \quad \text{and} \quad B = \int_0^T e^{A_c \tau} B_c \, d\tau.$$

1.3 Reference Controller (PID)

The given reference controller is set of three independent PID controllers: one that controls the engine, one that controls the side thrust and finally one that controls the angle of the engine or *vector*.

[4]:
```
pid = algorithms.PID()
print(f"PID Parameters:\n  Engine: {pid.engine}\n  Vector: {pid.vector}\n  Side:
  ↪ {pid.side}")
```

```
PID Parameters:
  Engine: [10, 0, 10]
  Vector: [0.085, 0.001, 10.55]
  Side: [5, 0, 6]
```

The reference PID controller works well when the initial state of the system is near the tuning point, as show in the example below (this is using the default parameters given in the assignment notebook).

[5]:
```
sim = simulator.Simulator(pid, scenario=0, record_video=True)
print(sim.userparams)
notebook.run_and_show_video(sim)
```

```
{'initial_position': (0.5, 0.9, 0.0)}
```

[5]:
```
<IPython.core.display.Video object>
```

1.3.1 Failure Scenario

Although decent under "good" conditions, that is, when the system state is near the set point where the PID was tuned, the controller quickly shows its weakeness when the initial conditions are far removed from the expected conditions.

[6]:
```
sim = simulator.Simulator(pid, scenario=2, record_video=True)
print(sim.userparams)
notebook.run_and_show_video(sim)
```

```
{'initial_position': (0.3, 0.9, 0.0)}
```

[6]:
```
<IPython.core.display.Video object>
```

We can also try with a random initial position and most likely it will fail in a similar way.

[7]:
```
sim = simulator.Simulator(pid, userparams={"random_initial_position": True},
                          record_video=True)

# notebook.delete_video(sim)
notebook.run_and_show_video(sim)
```

[7]:
```
<IPython.core.display.Video object>
```

This is because the independent PID contorllers are unaware of each other and cannot coordinate.

1.3.2 Implementation Details

The code that ran above is just a thin wrapper around the coded provided by the exercise.

[8]:
```
notebook.show_source(algorithms.PID.setup)
```

[8]:
```
    def setup(self, env, engine=None, vector=None, side=None):
        if engine:
            self.engine = engine
        if vector:
            self.vector = vector
        if side:
            self.side = side

        self.lpos = env.get_landing_position()
        self.pid = PID_Benchmark(self.engine, self.vector, self.side)
```

[9]:
```
notebook.show_source(algorithms.PID.run)
```

[9]:
```
    def run(self, x, env):
        log.debug(f"State x={x}")
        # Remove position offset
        start = time.time()
        x[0] -= self.lpos[0]
        x[1] -= self.lpos[1]
        action = np.array(self.pid.pid_algorithm(x))
        solvetime = (time.time() - start)
        log.debug(f"Computation took {solvetime:.2e}, action={action}")
        return action, solvetime
```

1.4 Model Predictive Control

1.4.1 Basic MPC Theory

To solve this problem we implement the model predictive control algorithm. The idea of MPC is to continuously predict the futures states using a model, and to decide the best action based on this prediction.

To decide what is the *best action* we have to define what is the *cost* of being in a certain state, say z, and applying the input u. This is done by defining a quadratic function in z and u using two positive definite matrices Q and R:

$$\texttt{cost}(z, u) = z^T Q z + u^T R u,$$

We will discuss later how to select Q and R. Having defined cost function, MPC becomes an optimization problem whereby we want to pick the sequence of N inputs $U = \{u_0, \dots, u_{N-1}\}$ from a starting state z, that minimizes the total cost:

$$\begin{aligned} u^\star = \arg_{u_0} \min_U \left\{ z_N^T S z_N + \sum_{k=0}^{N-1} \texttt{cost}(z_k, u_k) \right\} \\ \text{where} \quad z_{k+1} = \texttt{model}(z_k, u_k), \quad z_0 = z. \end{aligned}$$

For tuning reasons we add a *terminal cost* that is computed using a positive definite matrix S on the final state of the prediction z_N. The intuition is that we want to be able to penalize more errors in the final state.

Note that although the optimization provides N inputs in the future, only the first one (u_0) is used, that is because there might be inaccuracies in the model that make the prediction unreliable. After the input is applied the problem will be solved again to find a *new* u_0.

Finally, we can impose that the inputs and states must respect their constraints and specify that at the end of the prediction z_N must be a state of our choice called the final state z_f. This final or terminal state is $z_f = [x_p \ y_p \ 0 \ 0 \ 0 \ 0]^T$ where (x_p, y_p) are the coordinates of the landing pad:

$$\begin{aligned} u^\star = \arg_{u_0} \min_U \left\{ z_N^T S z_N + \sum_{k=0}^{N-1} z_k^T Q z_k + u_k^T R u_k \right\} \\ \text{subject to} \quad z_{k+1} = \texttt{model}(z_k, u_k), \\ z_0 = z, \quad z_N = z_f \\ G_z z_k \leq g_z, \\ G_u u_k \leq u_k. \end{aligned}$$

The formulation above is model predictive control: a parametric optmization problem with parameter z. When applied on a real system the control loop is as follows:

1. At time n the system is in state z_n
2. Compute the current optimal control input $u_n^\star$ by solving the MPC optimization problem above with the parameter z set to z_n
3. Apply the input $u_n^\star$ to go to state z_{n+1}, repeat from step 1

1.4.2 Linearized Dynamics

Finally, we introduce our model to the MPC fomulation. For simplicity we will use the LTI system (A, B) obtained by linearizing the the system dynamics around some stationary point (z_s, u_s) (as discussed above). In the linearized model the optimal input and state trajectory will be computed as a deviation from the stationary point, thus

$$
\begin{aligned}
u^\star - u_s = \arg_{u_0} \min_U \Bigg\{ & z_N^T S z_N + \sum_{k=0}^{N-1} z_k^T Q z_k + u_k^T R u_k \Bigg\} \\
\text{subject to} \quad & z_{k+1} = A z_k + B u_k \quad \text{(dynamics)} \\
& G_z z_k \leq g_z - G_z z_s \quad \text{(state constr.)} \\
& G_u u_k \leq g_u - G_u u_s \quad \text{(input constr.)} \\
& z_N = z_f - z_s \quad \text{(terminal constr.)} \\
& z_0 = z_n - z_s \quad \text{(parametrisation)}
\end{aligned}
$$

1.4.3 Defining the Constraints

As previously discussted, the constraints are given as a polytopic set defined by inqualities, in matrix form $G_z z \leq g_z$ and $G_u u \leq g_u$, however we did not discuss the actual values of the G's and g vectors.

For the inputs we have the constraints that the normalized $u \in [0, 1] \times [-1, 1]^2$, hence

$$
G_u u = \begin{bmatrix} I \\ -I \end{bmatrix} u \leq \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}^T = g_u.
$$

For the state contraints we have the imposed limits (from the assignment) that the rocket must stay in the simulated world, and that $\theta \in [-0.6108, 0.6108]$ (or $\pm 35°$). We will be slightly conservative and set set $\theta \in [-0.6, 0.6] = [\underline{\theta}, \overline{\theta}]$. Further for some particularly difficult scenario when the rocket is very close to the water we will impose that $y \geq y_p + \Delta$, where y_p is the y position of the landing pad and Δ a positive value. The latter constraint, excluding the world limits, is written as the inequality

$$
G_z z = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \end{bmatrix} z \leq \begin{bmatrix} \overline{\theta} \\ \underline{\theta} \\ -y_p - \Delta \end{bmatrix} = g_z.
$$

1.4.4 Demonstration

Before discussing tuning and the implementation, here is a demo of the MPC algorithm as described above in action.

[10]:
```
# Run with known difficult scenario
cmpc = algorithms.ClassicMPC()
sim = simulator.Simulator(cmpc, scenario=10, record_video=True)
notebook.run_and_show_video(sim)
```

[10]: <IPython.core.display.Video object>

1.4.5 Tuning

To tune the MPC controller there are three matrices Q, S and R. Although in principle one could work out a precise structure of Q and S these matrices (eg. using the Hamiltonian of the physical model) here we will simply use diagonal matrices.

To determine the values of the diagonal entries of Q, we first will take care to set higher values to the entries that multiply with the variables that are the least precise (because we are working with a linearized model, and not the true dynamics), that is, x, $\dot{x}$, θ and $\dot{\theta}$. Further, we generally (and intuitively) want the rocket to be upright, hence we set the entries for the angular components θ and $\dot{\theta}$ to higher values relative to the linear components. Finally another criterion is to be considered is the penalty for the vertical velocity $\dot{y}$. After a few iterations the parameters of choice for the current environment are

$$Q = \text{diag}(10, 1, 10, 4, 500, 200).$$

To choose S, since we want the the rocket to reach the landing pad as precisely as possible one could either set it to a high value, or it could also be reasonable to set it as a scalar multiple of Q such that all of the desired proprieties discussed above still apply but more strongly (for example $S = 100\ Q$).

For R, which is the penalty to the input, one could define the entries of the matrix to perform a conversion from effort to fuel. This would make MPC minimize fuel consumption for the landing. However, since this information is not available, we will perform an educated guess as was done in Q and argue that because of model uncertainty and smaller thrust power the main thruster is preferred to the side thruster and thrust vector. Hence we make the cost of F_S and φ more expensive than F_E by setting

$$R = \begin{bmatrix} 1 & & \\ & 10 & \\ & & 10 \end{bmatrix}.$$

1.4.6 Implementation Details

The implementation is very straightforward using CVXPY. The setup code (that only needs to run once) is separated for readibilty in three functions called one after the other.

[11]:
```
notebook.show_source(algorithms.ClassicMPC.setup_model)
```

[11]:
```
    def setup_model(self, env):
        # Get model parameters from environment
        model = SystemModel(env)
        self.scale_u = np.diag([
            1 / env.cfg.main_engine_thrust,
            1 / env.cfg.side_engine_thrust,
            1 / env.cfg.max_nozzle_angle
        ])
```

```
        # Linearization (stationary) points
        self.zs = np.zeros([6,])
        self.us = np.array([model.mass * model.gravity, 0, 0])
        self.scaled_us = self.scale_u @ self.us

        # Get linearised model of the system
        model.calculate_linear_system_matrices(self.zs, self.us)
        model.discretize_system_matrices(sample_time=self.sample_time)

        # Store to use later
        self.A, self.B = model.get_discrete_linear_system_matrices()
        self.nz, self.nu = self.B.shape
```

[12]:
```
notebook.show_source(algorithms.ClassicMPC.setup_constraints)
```

[12]:
```
    def setup_constraints(self, landing_pos):
        # State constraints
        self.Gx = np.array([
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, -1, 0],
            [0, -1, 0, 0, 0, 0]])
        self.gx = np.array([.6, .6, -landing_pos[1] + 8])
        log.debug(f"State constraints: gx=\n{self.gx},\nGx=\n{self.Gx} ")

        # Input constraints
        self.Gu = np.vstack([np.eye(self.nu), -np.eye(self.nu)])
        self.gu = np.array([1, 1, 1, 0, 1, 1])
        log.debug(f"Input constraints: gu=\n{self.gu},\nGu=\n{self.Gu} ")
```

[13]:
```
notebook.show_source(algorithms.ClassicMPC.setup_problem)
```

[13]:
```
    def setup_problem(self, landing_pos):
        # Terminal state
        self.zf = np.hstack([landing_pos[:2], np.zeros((4,))]) - self.zs
        log.debug(f"Terminal state zf={self.zf}")

        # Tuning parameters for MPC
        self.Q = np.diag([10, 1, 10, 4, 500, 200])
        self.S = np.eye(self.nz) * 100
        self.R = np.diag([1, 10, 10])

        # Set up CVXPY for MPC problem
        self.z0 = cp.Parameter((self.nz,))
        self.z = cp.Variable((self.nz, self.horizon+1))
        self.u = cp.Variable((self.nu, self.horizon))

        # Add initial condition
        constraints = [self.z[:, 0] == self.z0]
```

```
        # Add cost-to-go
        cost = 0
        for k in range(self.horizon):
            cost += cp.quad_form(self.z[:, k], self.Q)
            cost += cp.quad_form(self.u[:, k], self.R)
            constraints += [
                self.z[:, k+1] == self.A @ self.z[:, k] + self.B @ self.u[:, k],
                self.Gx @ self.z[:, k] <= self.gx - self.Gx @ self.zs,
                self.Gu @ self.u[:, k] <= self.gu - self.Gu @ self.scaled_us
            ]

        # Add terminal cost and constraint
        cost += cp.quad_form(self.z[:, self.horizon], self.S)
        constraints.append(self.z[:, self.horizon] == np.zeros((self.nz,)))

        # Create optimization problem
        self.problem = cp.Problem(cp.Minimize(cost), constraints)
```

The core of the algorithm, that runs online is then very short.

[14]:
```
notebook.show_source(algorithms.ClassicMPC.run)
```

[14]:
```
    def run(self, z, env):
        # Stop if there is nothing to do
        if z[6] and z[7]:
            return np.zeros(self.nu), 0

        # Give relative coordinate
        self.z0.value = z[:-2] - self.zf

        log.debug(f"Solving MPC optimization with z0={self.z0.value}")
        self.problem.solve()

        if self.problem.status in ["infeasible", "unbounded"]:
            log.error(f"Problem cannot be solved, it is {self.problem.status}")
            raise RuntimeError("Impossible optimization problem")

        action = self.scaled_us + self.u[:, 0].value
        solvetime = self.problem.solver_stats.solve_time
        log.debug(f"Solver took {solvetime:.4e}, result={action}")

        return action, solvetime
```

1.4.7 Failure Mode of Classic MPC

In spite of all of its many advantages the classic formulation of MPC is not perfect, as is shown in the following scenario. This happend when the optimization problem becomes infeasible. We will address this problem in the next section.

[15]:
```
cmpc = algorithms.ClassicMPC()
sim = simulator.Simulator(cmpc, scenario=11, record_video=True)
print(sim.userparams)
sim.run()
print(sim.alg.problem.status)
notebook.show_video(sim)
```

```
{'initial_position': (0.1, 0.5, 0.6)}
  2%|                                                                | 90/5000
[00:02<03:05, 26.41it/s]

Problem cannot be solved, it is infeasible
Simulation failed: Impossible optimization problem


Simulation terminated early! (may need more iterations)

infeasible
```

[15]:
```
<IPython.core.display.Video object>
```

1.4.8 Relaxation

As previously discussed above MPC is (convex) optimization problem, and thus has the problem of potentially becoming infeasible. To work around this we relax the optimization problem by adding *slack variables* that allow to infringe the state constraints. This is not ideal but having a bad solution is still better than having no solutions.

To relax the state constraints we introduce the variables $\epsilon_0, \epsilon_1, \dots, \epsilon_N$ that have the same size as g_z and linearly penalize their 1-norm in the cost with factor of v. Thus, the problem fomulation becomes

$$
\begin{aligned}
u^\star = \arg_{u_0} \min_U \Big\{ & z_N^T S z_N + v\|\epsilon_N\|_1 + \sum_{k=0}^{N-1} z_k^T Q z_k + u_k^T R u_k + v\|\epsilon_k\|_1 \Big\} \\
\text{subject to} \quad & z_{k+1} = \texttt{model}(z_k, u_k), \\
& z_0 = z, \quad z_N = z_f \\
& G_z z_k \leq g_z + \epsilon_k, \\
& G_u u_k \leq u_k.
\end{aligned}
$$

The newly introduced v is a tuning parameter that must satisfy some conditions for the relaxation to be theoretically optimal (for LTI systems they are given in KERRIGAN, Eric C.; MACIEJOWSKI, Jan M. Soft constraints and exact penalty functions in model predictive control. 2000).

[16]:
```
rmpc = algorithms.RelaxedMPC()
sim = simulator.Simulator(rmpc, scenario=2, record_video=True)
print(sim.userparams)
notebook.run_and_show_video(sim)
```

```
{'initial_position': (0.3, 0.9, 0.0)}
```

[16]:
```
<IPython.core.display.Video object>
```

Relaxing the MPC problem provides a solution to the scenario where classical MPC failed.

[17]:
```
rmpc = algorithms.RelaxedMPC()
sim = simulator.Simulator(rmpc, scenario=11, record_video=True)
print(sim.userparams)
notebook.run_and_show_video(sim)
```

```
{'initial_position': (0.1, 0.5, 0.6)}
```

[17]:
```
<IPython.core.display.Video object>
```

1.4.9 Implementation Details

The only difference between `ClassicMPC` and `RelaxedMCP` lies in the addition of the slack variables `s`.

[18]:
```
notebook.show_source(algorithms.RelaxedMPC.setup_problem)
```

[18]:
```
    def setup_problem(self, landing_pos):
        # Terminal state
        self.zf = np.hstack([landing_pos[:2], np.zeros((4,))]) - self.zs
        log.debug(f"Terminal state zf={self.zf}")

       # Tuning parameters for MPC
        self.Q = np.diag([10, 1, 10, 4, 500, 200])
        self.S = np.eye(self.nz) * 100
        self.R = np.diag([1, 10, 10])
        self.v = 1

        # Set up CVXPY for MPC problem
        self.z0 = cp.Parameter((self.nz,))
        self.z = cp.Variable((self.nz, self.horizon+1))
        self.u = cp.Variable((self.nu, self.horizon))

        # Slack variables (relaxation)
        self.s = cp.Variable((self.gx.shape[0], self.horizon+1))

        # Add initial condition
        constraints = [self.z[:, 0] == self.z0]

        # Add cost-to-go
        cost = 0
        for k in range(self.horizon):
            cost += cp.quad_form(self.z[:, k], self.Q)
            cost += self.v * cp.norm(self.s[:, k], 1)
            cost += cp.quad_form(self.u[:, k], self.R)
            constraints += [
```

```
                self.z[:, k+1] == self.A @ self.z[:, k] + self.B @ self.u[:, k],
                self.Gx @ self.z[:, k] <= self.gx + self.s[:, k] - self.Gx @␣
↪self.zs,
                self.Gu @ self.u[:, k] <= self.gu - self.Gu @ self.scaled_us
            ]

        # Add terminal cost and constraint
        cost += cp.quad_form(self.z[:, self.horizon], self.S)
        cost += self.v * cp.norm(self.s[:, self.horizon], 1)
        constraints.append(self.z[:, self.horizon] == np.zeros((self.nz,)))

        # Create optimization problem
        self.problem = cp.Problem(cp.Minimize(cost), constraints)
```

1.4.10 Robustness: Moving Barge and Wind

This was not requested by the assignment, but just to test the robustness there is a paramtetric version of `RelaxedMPC` where z_f is updated live in `ParametricMPC.run` (vs setting it up once in `ParametricMPC.setup_problem`). It works reasonably well, but since the model does account for wind the performance is rather suboptimal (compared to what MPC could do with a wind model).

[19]:
```
userparams = {"random_initial_position": True, "enable_wind": True,␣
 ↪"enable_moving_barge": True }
pmpc = algorithms.ParametricMPC()
sim = simulator.Simulator(pmpc, userparams, record_video=True, seed=31415)
# notebook.delete_video(sim)
notebook.run_and_show_video(sim)
```

[19]:
```
<IPython.core.display.Video object>
```

1.4.11 Implementation Details

In this case `self.zf` is a paramter that is set in `ParametricMPC.run`.

[20]:
```
notebook.show_source(algorithms.ParametricMPC.setup_problem)
```

[20]:
```
    def setup_problem(self, _lpos):
        # Tuning parameters for MPC
        self.Q = np.diag([10, 1, 10, 7, 500, 200])
        self.S = np.eye(self.nz) * 100
        self.R = np.diag([1, 10, 10])
        self.v = 1

        # Set up CVXPY for MPC problem
        self.z0 = cp.Parameter((self.nz,))
        self.zf = cp.Parameter((self.nz,))
        self.z = cp.Variable((self.nz, self.horizon+1))
```

```
        self.u = cp.Variable((self.nu, self.horizon))

        # Slack variables (relaxation)
        self.s = cp.Variable((self.gx.shape[0], self.horizon+1))

        # Add initial condition
        constraints = [self.z[:, 0] == self.z0]

        # Add cost-to-go
        cost = 0
        for k in range(self.horizon):
            cost += cp.quad_form(self.z[:, k], self.Q)
            cost += self.v * cp.norm(self.s[:, k], 1)
            cost += cp.quad_form(self.u[:, k], self.R)
            constraints += [
                self.z[:, k+1] == self.A @ self.z[:, k] + self.B @ self.u[:, k],
                self.Gx @ self.z[:, k] <= self.gx + self.s[:, k] - self.Gx @␣
↪self.zs,
                self.Gu @ self.u[:, k] <= self.gu - self.Gu @ self.scaled_us
            ]

        # Add terminal cost and constraint
        cost += cp.quad_form(self.z[:, self.horizon], self.S)
        cost += self.v * cp.norm(self.s[:, self.horizon], 1)
        constraints.append(self.z[:, self.horizon] == np.zeros((self.nz,)))

        # Create optimization problem
        self.problem = cp.Problem(cp.Minimize(cost), constraints)
```

[21]:
```
notebook.show_source(algorithms.ParametricMPC.run)
```

[21]:
```
    def run(self, z, env):
        # Stop if there is nothing to do
        if z[6] and z[7]:
            return np.zeros(self.nu), 0

        # Setup parameters
        lpos = env.get_landing_position()
        self.zf.value = np.hstack([lpos[:2], np.zeros((4,))]) - self.zs
        self.z0.value = z[:-2] - self.zf.value

        log.debug(f"Solving MPC optimization with z0={self.z0.value}")
        self.problem.solve()

        if self.problem.status in ["infeasible", "unbounded"]:
            log.error(f"Problem cannot be solved, it is {self.problem.status}")
            raise RuntimeError("Impossible optimization problem")
```

```
action = self.scaled_us + self.u[:, 0].value
solvetime = self.problem.solver_stats.solve_time
log.debug(f"Solver took {solvetime:.4e}, result={action}")

return action, solvetime
```