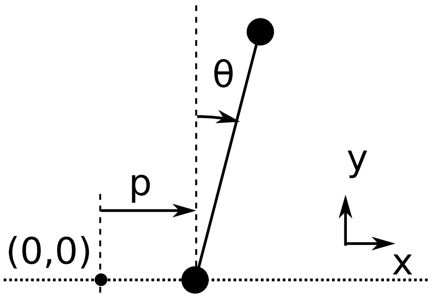
Hands-on CasADi course on optimal control, Yacoda

Exercise: Model predictive control

In this exercise we consider an inverted pendulum on a horizontally constrained cart:



The system is controlled by a force acting in the x-direction on the cart. The system is described by a four-dimensional state-space:

$$x := \begin{bmatrix} p \\ \theta \\ \dot{p} \\ \dot{\theta} \end{bmatrix}.$$

We discretize this system over a horizon of $T = 2$ seconds using $N = 160$ control intervals, to obtain a multiple shooting formulation for an optimal control problem:

$$\begin{aligned} \underset{x_1, x_2, \dots x_{N+1}, u_1, u_2, \dots, u_N}{\text{minimize}} \quad & \sum_{k=1}^{N} u_k^2 \\ \text{subject to} \quad & F(x_k, u_k) = x_{k+1}, \quad k = 1 \dots N \\ & -3 \leq p_k \leq 3, \quad k = 1 \dots N+1 \\ & -1.2 \leq u_k \leq 1.2, \quad k = 1 \dots N \\ & x_1 = [1; 0; 0; 0], \\ & x_{N+1} = [0; 0; 0; 0]. \end{aligned} \tag{1}$$

Start from `inverted_pendulum`, the reference optimal control implementation for this problem. Verify that this file runs succesfully.

1 Basic implementation of MPC

Tasks:

1. Add a strong regularization term on the cart position to the objective: $1000 \sum_{k=1}^{N+1} p_k^2$.
2. Introduce a parameter for the initial state of the OCP:

```
x0 = opti.parameter(nx)
```

In the initial constraint, replace the numerical value for the initial location by this parameter. Set the value of that parameter a new the initial location, and solve:

```
opti.set_value(x0,vertcat(0.5,0,0,0))
sol = opti.solve()
```

3. Implement a basic MPC loop that simulates for $N_{\text{sim}} = 5$ samples. Use the following boilerplate code:

```
current_x = vertcat(0.5,0,0,0)

for i in range(Nsim):
    t0 = time.time()
    # What control signal should I apply?
    u_sol = ...

    u_history[:,i] = u_sol

    # Simulate the system over dt
    current_x = ...

    # Set the value of parameter x0 to the current x
    opti.set_value(x0, current_x)

    # Solve the NLP
    sol = opti.solve()

    print(time.time()-t0)
```

Verify that your code runs without errors.

4. Let's silence the NLP solver output by passing the following options to the `opti.solver` command:

```
options = dict()
options["ipopt"] = {"print_level": 0}
opti.solver('ipopt',options)
```

How long does each MPC iteration take on your computer? (2 secs for me).

2 Speeding up the implementation

Tasks:

1. Hot-starting: In the MPC loop, start with an initial guess taken from the converged solution of the previous iteration:

   ```
   opti.set_initial(opti.x, sol.value(opti.x))
   opti.set_initial(opti.lam_g, sol.value(opti.lam_g))
   ```

   How long does each MPC iteration take on your computer? (1 sec for me).

2. SQP: Interior point methods (like IPOPT) are generally robust at finding a minimizer, but are not well suited for online problems: the barrier parameter will make the algorithm walk away from a perfect initial guess, only to come back (in the best case!) after a while.

   Let's switch to an SQP method, using an active-set QP solver:

   ```
   options["qpsol"] = 'qrqp'
   opti.solver('sqpmethod',options)
   ```

   As before, you may wish to silence the output a bit:

   ```
   options["qpsol"] = 'qrqp';
   options["qpsol_options"] = {"print_iter": False, "print_header": False}
   options["print_iteration"] = False
   options["print_header"] = False
   options["print_status"] = False
   options["print_time"] = False
   opti.solver('sqpmethod',options)
   ```

   How long does each MPC iteration take on your computer? (0.2 sec for me).

3. Inlining and expanding:

   Add the option `simplify True` to the integrator command. With this option, the result of the integrator command will be a simple MX Function, as if you had written the Runge-Kutta integrator recipe explictly using `x` and `u` MX symbols:

   ```
   k1 = f(x, u)
   k2 = f(x + dt/2 * k1, u)
   k3 = f(x + dt/2 * k2, u)
   k4 = f(x + dt * k3, u)
   xf = x+dt/6*(k1 +2*k2 +2*k3 +k4)
   ```

   Add yet another option, `expand true` to the list of solver options. This option will expand the entire expression graph (MX), including any Functions, into one flat scalar expression graph (SX). This will generally run faster, at the expense of more memory usage.

   How long does each MPC iteration take on your computer? (0.05 sec for me).

4. (extra) The use of `Opti` incurs some overhead. Convert the NLP problem into a regular CasADi Function with an `nlpsol` embedded with:

```
inputs = [x0,opti.x,opti.lam_g]
outputs = [U[0],opti.x,opti.lam_g]
mpc_step = opti.to_function('mpc_step',inputs,outputs)
```

   The resulting CasADi Function has three inputs (the current state measurement, initial values for all decision variables, initial values for all multipliers) and three outputs (the control signal to apply next interval, decision variables at the solution, multipliers at the solution).

   How long does each MPC iteration take on your computer? (0.03 sec for me).

   Next step could be codegeneration. Since CasADi 3.5, you can codegenerate `sqpmethod` and `qrqp` qp solver.

3 Analyzing the control performance

Tasks:

1. Extend the MPC loop to $N_{\text{sim}} = 200$ samples. Collect the applied controls and current states to make a plot over the entire simulation horizon N_{sim}.

   Is the system regulated towards $[0; 0; 0; 0]$ (such as mandated by the terminal constraint)?

   Why is that desired state not exactly met when $i = 161$?

2. Add random noise on the system simulation `[0;0;0;0.01*rand]`. Is the system regulated towards $[0; 0; 0; 0]$? Why not? What would a well-tuned PID controller do in this case?