RHC Example: Double integrator with bounded input

Richard M. Murray, 3 Feb 2022 (updated 29 Jan 2023)

To illustrate the implementation of a receding horizon controller, we consider a linear system corresponding to a double integrator with bounded input:

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{clip}(u) \qquad \text{where} \qquad \text{clip}(u) = \begin{cases} -1 & u < -1, \\ u & -1 \leq u \leq 1, \\ 1 & u > 1. \end{cases}$$

We implement a model predictive controller by choosing

$$Q_x = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \qquad Q_u = \begin{bmatrix} 1 \end{bmatrix}, \qquad P_1 = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}.$$

In [1]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import control as ct
import control.optimal as opt
import control.flatsys as fs
import time
```

System definition

The system is defined as a double integrator with bounded input.

In [2]:
```
def doubleint_update(t, x, u, params):
    # Get the parameters
    lb = params.get('lb', -1)
    ub = params.get('ub', 1)
    assert lb < ub

    # bound the input
    u_clip = np.clip(u, lb, ub)

    return np.array([x[1], u_clip[0]])

proc = ct.NonlinearIOSystem(
    doubleint_update, None, name="double integrator",
    inputs = ['u'], outputs=['x[0]', 'x[1]'], states=2)
```

Receding horizon controller

To define a receding horizon controller, we create an optimal control problem (using the `OptimalControlProblem` class) and then use the `compute_trajectory` method to solve for the trajectory from the current state.

We start by defining the cost functions, which consists of a trajectory cost and a terminal cost:

In [3]:
```
Qx = np.diag([1, 0])            # state cost
Qu = np.diag([1])               # input cost
traj_cost=opt.quadratic_cost(proc, Qx, Qu)

P1 = np.diag([0.1, 0.1])        # terminal cost
term_cost = opt.quadratic_cost(proc, P1, None)
```

We also set up a set of constraints the correspond to the fact that the input should have magnitude 1. This can be done using either the `input_range_constraint` function or the `input_poly_constraint` function.

In [4]:
```
traj_constraints = opt.input_range_constraint(proc, -1, 1)
# traj_constraints = opt.input_poly_constraint(
#     proc, np.array([[1], [-1]]),  np.array([1, 1]))
```

We define the horizon for evaluating finite-time, optimal control by setting up a set of time points across the designed horizon. The input will be computed at each time point.

In [5]:
```
Th = 5
timepts = np.linspace(0, Th, 11, endpoint=True)
print(timepts)
```

```
[0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
```

Finally, we define the optimal control problem that we want to solve (without actually solving it).

In [6]:
```
# Set up the optimal control problem
ocp = opt.OptimalControlProblem(
    proc, timepts, traj_cost,
    terminal_cost=term_cost,
    trajectory_constraints=traj_constraints,
    # terminal_constraints=term_constraints,
)
```

To make sure that the problem is properly defined, we solve the problem for a specific initial condition. We also compare the amount of time required to solve the problem from a "cold start" (no initial guess) versus a "warm start" (use the previous solution, shifted forward on point in time).

In [7]:
```
X0 = np.array([1, 1])

start_time = time.process_time()
res = ocp.compute_trajectory(X0, initial_guess=0, return_states=True)
```

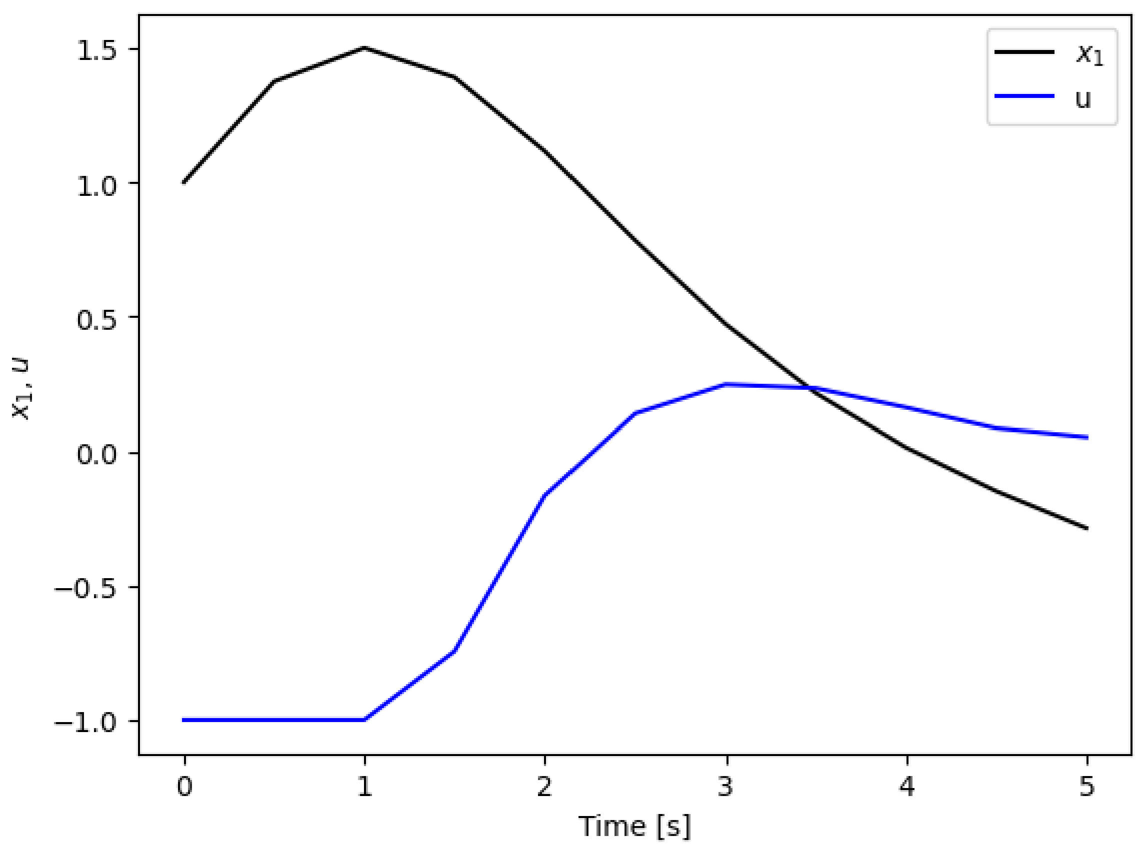
```
stop_time = time.process_time()
print(f'* Cold start: {stop_time-start_time:.3} sec')

# Resolve using previous solution (shifted forward) as initial guess to comp
start_time = time.process_time()
u = res.inputs
u_shift = np.hstack([u[:, 1:], u[:, -1:]])
ocp.compute_trajectory(X0, initial_guess=u_shift, print_summary=False)
stop_time = time.process_time()
print(f'* Warm start: {stop_time-start_time:.3} sec')
```

```
Summary statistics:
* Cost function calls: 238
* Constraint calls: 280
* System simulations: 2
* Final cost: 6.0233830155797055
* Cold start: 0.138 sec
* Warm start: 0.155 sec
```

(In this case the timing is not that different since the system is very simple.)

Plotting the result, we see that the solution is properly computed.

In [8]:
```
plt.plot(res.time, res.states[0], 'k-', label='$x_1$')
plt.plot(res.time, res.inputs[0], 'b-', label='u')
plt.xlabel('Time [s]')
plt.ylabel('$x_1$, $u$')
plt.legend();
```



We implement the receding horicon controller using a function that we can with different versions of the problem.

In [9]:
```
# Create a figure to use for plotting
def run_rhc_and_plot(
        proc, ocp, X0, Tf, print_summary=False, verbose=False, ax=None, plot
    # Start at the initial point
    x = X0

    # Initialize the axes
    if plot and ax is None:
        ax = plt.axes()

    # Initialize arrays to store the final trajectory
    time_, inputs_, outputs_, states_  = [], [], [], []

    # Generate the individual traces for the receding horizon control
    for t in ocp.timepts:
        # Compute the optimal trajectory over the horizon
        start_time = time.process_time()
        res = ocp.compute_trajectory(x, print_summary=print_summary)
        if verbose:
            print(f"{t=}: comp time = {time.process_time() - start_time:0.3}

        # Simulate the system for the update time, with higher res for plott
        tvec = np.linspace(0, res.time[1], 20)
        inputs = res.inputs[:, 0] + np.outer(
            (res.inputs[:, 1] - res.inputs[:, 0]) / (tvec[-1] - tvec[0]), tv
        soln = ct.input_output_response(proc, tvec, inputs, x)

        # Save this segment for later use (final point will appear in next s
        time_.append(t + soln.time[:-1])
        inputs_.append(soln.inputs[:, :-1])
        outputs_.append(soln.outputs[:, :-1])
        states_.append(soln.states[:, :-1])

        if plot:
            # Plot the results over the full horizon
            h3, = ax.plot(t + res.time, res.states[0], 'k--', linewidth=0.5)
            ax.plot(t + res.time, res.inputs[0], 'b--', linewidth=0.5)

            # Plot the results for this time segment
            h1, = ax.plot(t + soln.time, soln.states[0], 'k-')
            h2, = ax.plot(t + soln.time, soln.inputs[0], 'b-')

        # Update the state to use for the next time point
        x = soln.states[:, -1]

    # Append the final point to the response
    time_.append(t + soln.time[-1:])
    inputs_.append(soln.inputs[:, -1:])
    outputs_.append(soln.outputs[:, -1:])
    states_.append(soln.states[:, -1:])

    # Label the plot
    if plot:
```

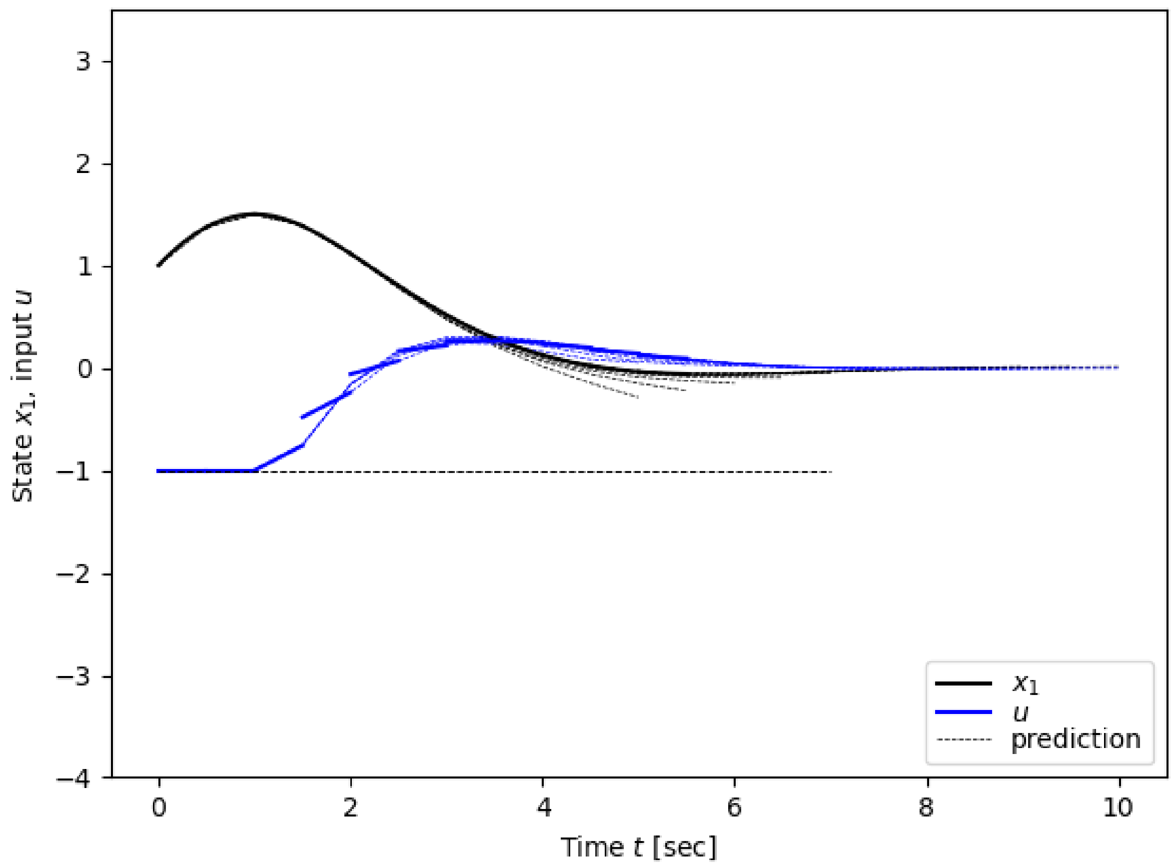
```
        # Adjust the limits for consistency
        ax.set_ylim([-4, 3.5])

        # Add reference line for input lower bound
        ax.plot([0, 7], [-1, -1], 'k--', linewidth=0.666)

        # Label the results
        ax.set_xlabel("Time $t$ [sec]")
        ax.set_ylabel("State $x_1$, input $u$")
        ax.legend(
            [h1, h2, h3], ['$x_1$', '$u$', 'prediction'],
            loc='lower right', labelspacing=0)
        plt.tight_layout()

    # Append
    return ct.TimeResponseData(
        np.hstack(time_), np.hstack(outputs_), np.hstack(states_), np.hstack
```

Finally, we call the controller and plot the response. The solid lines show the portions of the trajectory that we follow. The dashed lines are the trajectory over the full horizon, but which are not followed since we update the computation at each time step. (To get rid of the statistics of each optimization call, use `print_summary=False`.)

In [10]:

```
Tf = 10
rhc_resp = run_rhc_and_plot(proc, ocp, X0, Tf, verbose=True, print_summary=F
print(f"xf = {rhc_resp.states[:, -1]}")
```

```
t=0.0: comp time = 0.13
t=0.5: comp time = 0.113
t=1.0: comp time = 0.115
t=1.5: comp time = 0.129
t=2.0: comp time = 0.115
t=2.5: comp time = 0.113
t=3.0: comp time = 0.117
t=3.5: comp time = 0.112
t=4.0: comp time = 0.114
t=4.5: comp time = 0.0964
t=5.0: comp time = 0.0953
xf = [-0.06056343 -0.02329116]
```



RHC vs LQR vs LQR terminal cost

In the example above, we used a receding horizon controller with the terminal cost as $P_1 = \text{diag}(0.1, 0.1)$. An alternative is to set the terminal cost to be the LQR terminal cost that goes along with the trajectory cost, which then provides a "cost to go" that matches the LQR "cost to go" (but keeping in mind that the LQR controller does not necessarily respect the constraints).

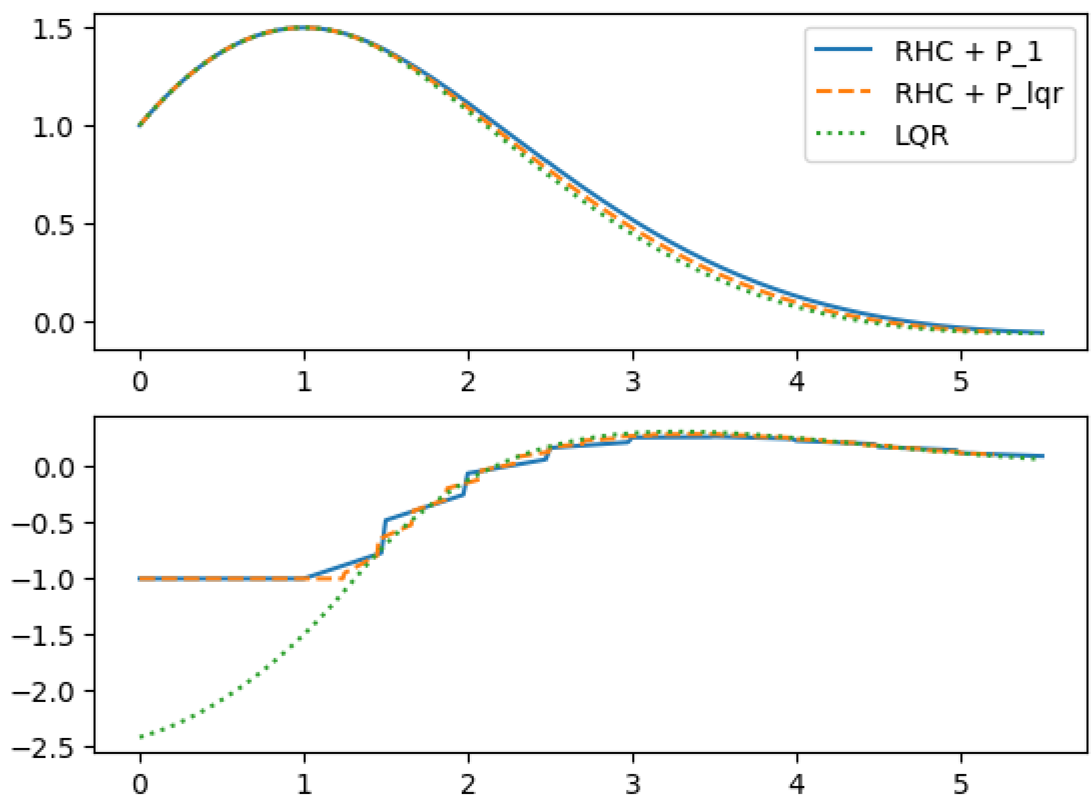
The following code compares the original RHC formulation with a receding horizon controller using an LQR terminal cost versus an LQR controller.

In [11]:
```
# Get the LQR solution
K, P_lqr, E = ct.lqr(proc.linearize(0, 0), Qx, Qu)
print(f"P_lqr = \n{P_lqr}")

# Create an LQR controller (and run it)
lqr_ctrl, lqr_clsys = ct.create_statefbk_iosystem(proc, K)
lqr_resp = ct.input_output_response(lqr_clsys, rhc_resp.time, 0, X0)

# Create a new optimal control problem using the LQR terminal cost
# (need use more refined time grid as well, to approximate LQR rate)
lqr_timepts = np.linspace(0, Th, 25, endpoint=True)
lqr_term_cost=opt.quadratic_cost(proc, P_lqr, None)
ocp_lqr = opt.OptimalControlProblem(
    proc, lqr_timepts, traj_cost, terminal_cost=lqr_term_cost,
```

```
    trajectory_constraints=traj_constraints,
)

# Create the response for the new controller
rhc_lqr_resp = run_rhc_and_plot(
    proc, ocp_lqr, X0, 10, plot=False, print_summary=False)

# Plot the different responses to compare them
fig, ax = plt.subplots(2, 1)
ax[0].plot(rhc_resp.time, rhc_resp.states[0], label='RHC + P_1')
ax[0].plot(rhc_lqr_resp.time, rhc_lqr_resp.states[0], '--', label='RHC + P_l
ax[0].plot(lqr_resp.time, lqr_resp.outputs[0], ':', label='LQR')
ax[0].legend()

ax[1].plot(rhc_resp.time, rhc_resp.inputs[0], label='RHC + P_1')
ax[1].plot(rhc_lqr_resp.time, rhc_lqr_resp.inputs[0], '--', label='RHC + P_l
ax[1].plot(lqr_resp.time, lqr_resp.outputs[2], ':', label='LQR')
```

```
P_lqr =
[[1.41421356 1.        ]
 [1.         1.41421356]]
```

Out[11]: `[<matplotlib.lines.Line2D at 0x7f8d0897f850>]`



Discrete time RHC

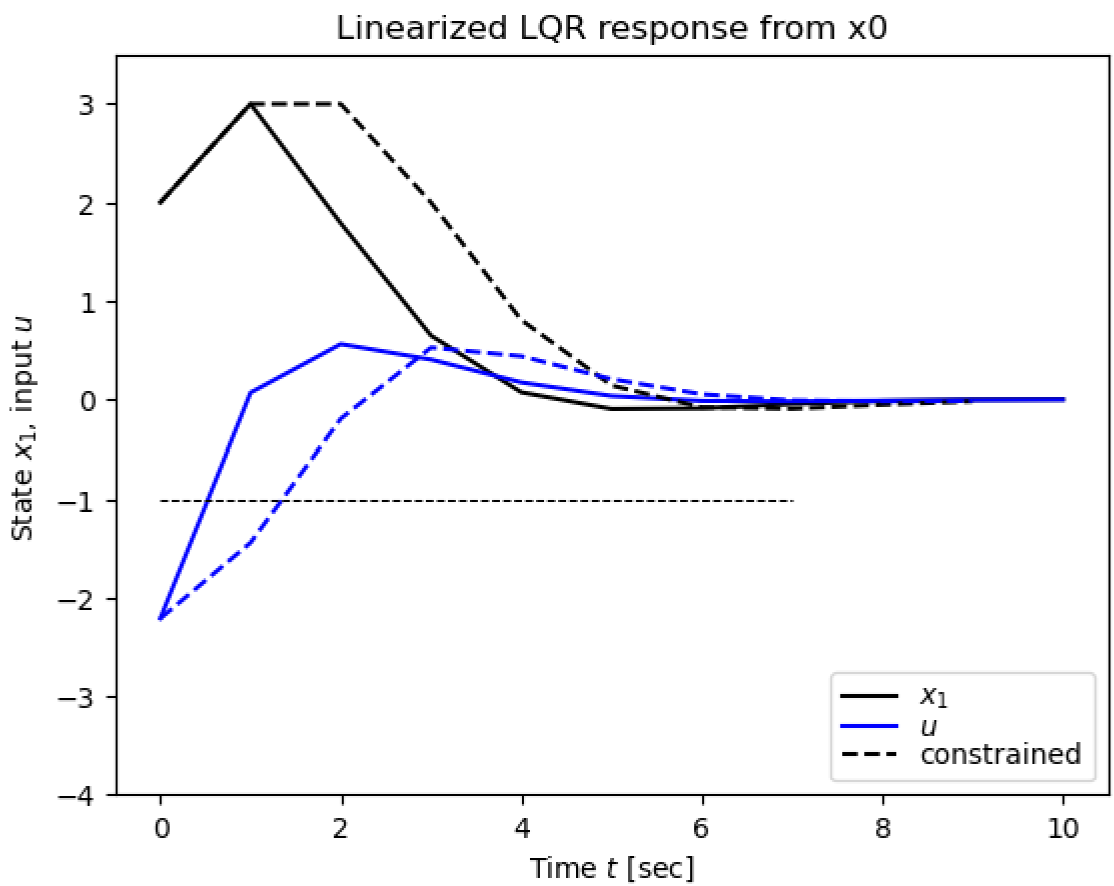
Many receding horizon control problems are solved based on a discrete time model. We show here how to implement this for a "double integrator" system, which in discrete time has the form

$$x[k+1] = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x[k] + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{clip}(u[k])$$

In [12]:
```
#
# System definition
#

def doubleint_update(t, x, u, params):
    # Get the parameters
    lb = params.get('lb', -1)
    ub = params.get('ub', 1)
    assert lb < ub

    # Get the sampling time
    dt = params.get('dt', 1)

    # bound the input
    u_clip = np.clip(u, lb, ub)

    return np.array([x[0] + dt * x[1], x[1] + dt * u_clip[0]])

proc = ct.NonlinearIOSystem(
    doubleint_update, None, name="double integrator",
    inputs = ['u'], outputs=['x[0]', 'x[1]'], states=2,
    params={'dt': 1}, dt=1)

#
# Linear quadratic regulator
#

# Define the cost functions to use
Qx = np.diag([1, 0])             # state cost
Qu = np.diag([1])                # input cost
P1 = np.diag([0.1, 0.1])         # terminal cost

# Get the LQR solution
K, P, E = ct.dlqr(proc.linearize(0, 0), Qx, Qu)

# Test out the LQR controller, with no constraints
linsys = proc.linearize(0, 0)
clsys_lin = ct.ss(linsys.A - linsys.B @ K, linsys.B, linsys.C, 0, dt=proc.dt

X0 = np.array([2, 1])            # initial conditions
Tf = 10                          # simulation time
res = ct.initial_response(clsys_lin, Tf, X0=X0)

# Plot the results
plt.figure(1); plt.clf(); ax = plt.axes()
ax.plot(res.time, res.states[0], 'k-', label='$x_1$')
ax.plot(res.time, (-K @ res.states)[0], 'b-', label='$u$')

# Test out the LQR controller with constraints
clsys_lqr = ct.feedback(proc, -K, 1)
tvec = np.arange(0, Tf, proc.dt)
res_lqr_const = ct.input_output_response(clsys_lqr, tvec, 0, X0)
```

```
# Plot the results
ax.plot(res_lqr_const.time, res_lqr_const.states[0], 'k--', label='constrain
ax.plot(res_lqr_const.time, (-K @ res_lqr_const.states)[0], 'b--')
ax.plot([0, 7], [-1, -1], 'k--', linewidth=0.75)

# Adjust the limits for consistency
ax.set_ylim([-4, 3.5])

# Label the results
ax.set_xlabel("Time $t$ [sec]")
ax.set_ylabel("State $x_1$, input $u$")
ax.legend(loc='lower right', labelspacing=0)
plt.title("Linearized LQR response from x0")
```

Out[12]: Text(0.5, 1.0, 'Linearized LQR response from x0')



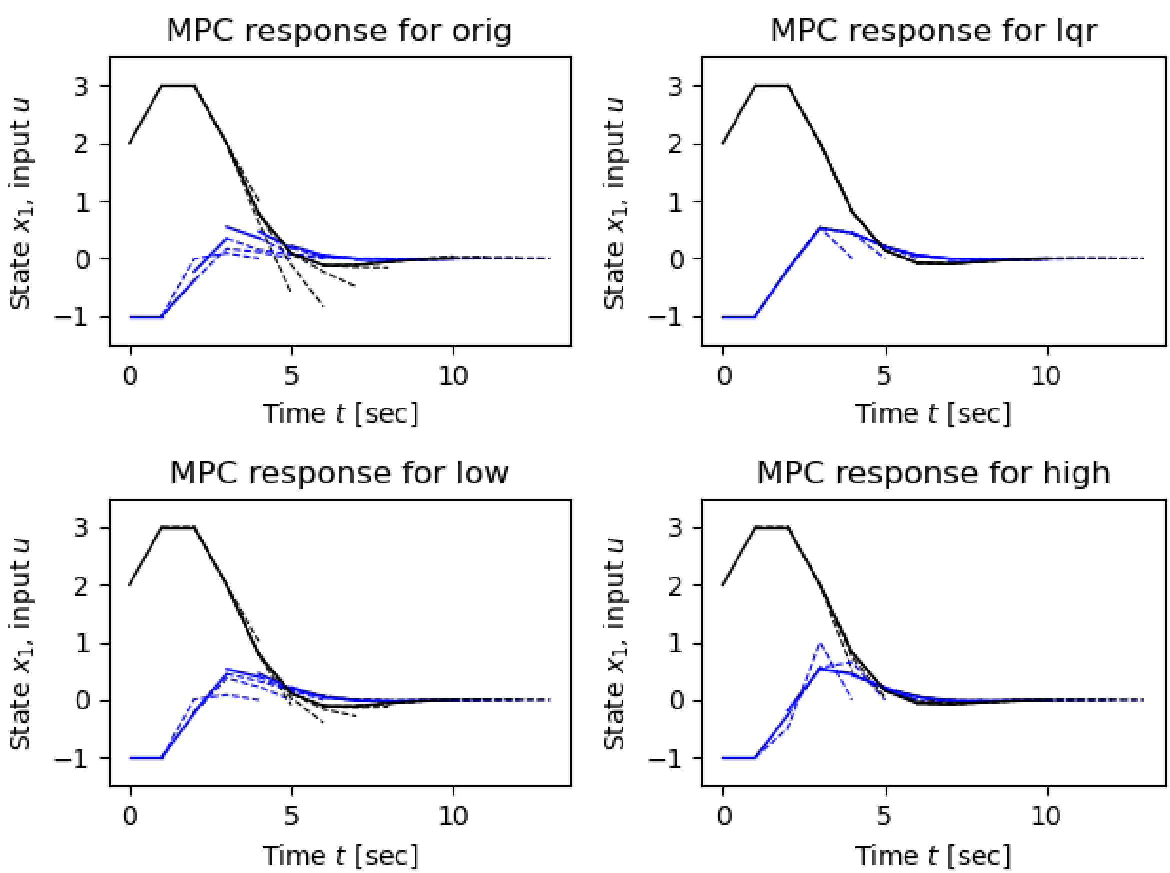
In [13]:

```
#
# Receding horizon controller
#

# Create the constraints
traj_constraints = opt.input_range_constraint(proc, -1, 1)
term_constraints = opt.state_range_constraint(proc, [0, 0], [0, 0])

# Define the optimal control problem we want to solve
T = 5
timepts = np.arange(0, T * proc.dt, proc.dt)
```

```python
# Set up the optimal control problems
ocp_orig = opt.OptimalControlProblem(
    proc, timepts,
    opt.quadratic_cost(proc, Qx, Qu),
    trajectory_constraints=traj_constraints,
    terminal_cost=opt.quadratic_cost(proc, P1, None),
)

ocp_lqr = opt.OptimalControlProblem(
    proc, timepts,
    opt.quadratic_cost(proc, Qx, Qu),
    trajectory_constraints=traj_constraints,
    terminal_cost=opt.quadratic_cost(proc, P, None),
)

ocp_low = opt.OptimalControlProblem(
    proc, timepts,
    opt.quadratic_cost(proc, Qx, Qu),
    trajectory_constraints=traj_constraints,
    terminal_cost=opt.quadratic_cost(proc, P/10, None),
)

ocp_high = opt.OptimalControlProblem(
    proc, timepts,
    opt.quadratic_cost(proc, Qx, Qu),
    trajectory_constraints=traj_constraints,
    terminal_cost=opt.quadratic_cost(proc, P*10, None),
)
weight_list = [P1, P, P/10, P*10]
ocp_list = [ocp_orig, ocp_lqr, ocp_low, ocp_high]

# Do a test run to figure out how long computation takes
start_time = time.process_time()
ocp_lqr.compute_trajectory(X0)
stop_time = time.process_time()
print("* Process time: %0.2g s\n" % (stop_time - start_time))

# Create a figure to use for plotting
fig, [[ax_orig, ax_lqr], [ax_low, ax_high]] = plt.subplots(2, 2)
ax_list = [ax_orig, ax_lqr, ax_low, ax_high]
ax_name = ['orig', 'lqr', 'low', 'high']

# Generate the individual traces for the receding horizon control
for ocp, ax, name, Pf in zip(ocp_list, ax_list, ax_name, weight_list):
    x, t = X0, 0
    for i in np.arange(0, Tf, proc.dt):
        # Calculate the optimal trajectory
        res = ocp.compute_trajectory(x, print_summary=False)
        soln = ct.input_output_response(proc, res.time, res.inputs, x)

        # Plot the results for this time instant
        ax.plot(res.time[:2] + t, res.inputs[0, :2], 'b-', linewidth=1)
        ax.plot(res.time[:2] + t, soln.outputs[0, :2], 'k-', linewidth=1)

        # Plot the results projected forward
        ax.plot(res.time[1:] + t, res.inputs[0, 1:], 'b--', linewidth=0.75)
```

```
        ax.plot(res.time[1:] + t, soln.outputs[0, 1:], 'k--', linewidth=0.75

        # Update the state to use for the next time point
        x = soln.states[:, 1]
        t += proc.dt

    # Adjust the limits for consistency
    ax.set_ylim([-1.5, 3.5])

    # Label the results
    ax.set_xlabel("Time $t$ [sec]")
    ax.set_ylabel("State $x_1$, input $u$")
    ax.set_title(f"MPC response for {name}")
    plt.tight_layout()
```

```
Summary statistics:
* Cost function calls: 38
* Constraint calls: 51
* System simulations: 81
* Final cost: 29.24889270852881
* Process time: 0.22 s
```



We can also implement a receding horizon controller for a discrete time system using `opt.create_mpc_iosystem`. This creates a controller that accepts the current state as the input and generates the control to apply from that state.

In [14]:
```
# Construct using create_mpc_iosystem
clsys = opt.create_mpc_iosystem(
```

```
    proc, timepts, opt.quadratic_cost(proc, Qx, Qu), traj_constraints,
    terminal_cost=opt.quadratic_cost(proc, P1, None),
)
print(clsys)
```

```
<NonlinearIOSystem>: sys[9]
Inputs (2): ['u[0]', 'u[1]']
Outputs (1): ['y[0]']
States (5): ['x[0]', 'x[1]', 'x[2]', 'x[3]', 'x[4]']

Update: <function OptimalControlProblem.create_mpc_iosystem.<locals>._updat
e at 0x7f8d01eef2e0>
Output: <function OptimalControlProblem.create_mpc_iosystem.<locals>._outpu
t at 0x7f8d01eef370>
```

(This function needs some work to be more user-friendly, e.g. renaming of the inputs and outputs.)

In [ ]:

```

```