Trajectory Tracking

Richard M. Murray, 19 Nov 2021 (updated 6 May 2024)

This notebook contains an example of using trajectory tracking for a (nonlinear) state space system. The controller is of the form

$$u = u_\text{d} - K(x - x_\text{d}),$$

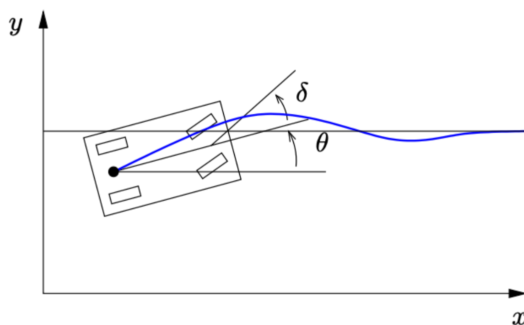
where x_d, u_d is a feasible trajectory, and K is a feedback gain first computed around a nominal condition and then computed using gain scheduling.

In [1]:
```
# Import the packages needed for the examples included in this notebook
import numpy as np
import matplotlib.pyplot as plt
import itertools
from cmath import sqrt
from math import pi

try:
  import control as ct
  print("python-control version:", ct.__version__)
except ImportError:
  # Version 0.10.0 is enough for this notebook
  !pip install control
  import control as ct

# Import the python optimal-control module
import control.optimal as opt
import control.flatsys as fs
```

```
python-control version: 0.10.0
```

Vehicle Steering Dynamics

The vehicle dynamics are given by a simple bicycle model:



$$\begin{aligned}\dot x &= \cos\theta\, v \\ \dot y &= \sin\theta\, v \\ \dot\theta &= \frac{v}{l}\tan\delta\end{aligned}$$

We take the state of the system as (x, y, θ) where (x, y) is the position of the vehicle in the plane and θ is the angle of the vehicle with respect to horizontal. The vehicle input is

given by (v, δ) where v is the forward velocity of the vehicle and δ is the angle of the steering wheel. The model includes saturation of the vehicle steering angle.

In [2]:
```
# Vehicle steering dynamics
#
# System state: x, y, theta
# System input: v, delta
# System output: x, y
# System parameters: wheelbase, maxsteer
#
try:
  from kincar import kincar, plot_lanechange
except ImportError:
  !wget --no-check-certificate https://www.cds.caltech.edu/~murray/courses/c
  from kincar import kincar, plot_lanechange

print(kincar)
print(kincar.params)
```

```
<FlatSystem>: kincar
Inputs (2): ['v', 'delta']
Outputs (3): ['x', 'y', 'theta']
States (3): ['x', 'y', 'theta']

Update: <function _kincar_update at 0x117cb5260>
Output: <function _kincar_output at 0x117cb51c0>

Forward: <function _kincar_flat_forward at 0x117cb4b80>
Reverse: <function _kincar_flat_reverse at 0x117cb4ea0>
{'wheelbase': 3, 'maxsteer': 0.5}
```

The following code is contained in the model:

```
def _kincar_update(t, x, u, params):
    # Get the parameters for the model
    l = params['wheelbase']             # vehicle wheelbase
    deltamax = params['maxsteer']         # max steering
angle (rad)

    # Saturate the steering input
    delta = np.clip(u[1], -deltamax, deltamax)

    # Return the derivative of the state
    return np.array([
        np.cos(x[2]) * u[0],            # xdot = cos(theta) v
        np.sin(x[2]) * u[0],            # ydot = sin(theta) v
        (u[0] / l) * np.tan(delta)      # thdot = v/l
tan(delta)
    ])
```

State feedback controller

We start by designing a state feedback controller that can be used to stabilize the system. We design the controller around a nominal forward speed of 10 m/s and then apply this to the vehicle at different speeds.

In [3]:
```
# Compute the linearization of the dynamics at a nominal point
x_nom = np.array([0, 0, 0])
u_nom = np.array([5, 0])
P = ct.linearize(kincar, x_nom, u_nom)     # Linearized systems
print(P)

Qx = np.diag([1, 10, 0.1])
Qu = np.diag([1, 1])
K, _, _ = ct.lqr(P.A, P.B, Qx, Qu)
print(K)
```

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

A = [[ 0.00000000e+00  0.00000000e+00 -2.50022225e-06]
     [ 0.00000000e+00  0.00000000e+00  5.00000000e+00]
     [ 0.00000000e+00  0.00000000e+00  0.00000000e+00]]

B = [[1.         0.        ]
     [0.         0.        ]
     [0.         1.66666667]]

C = [[1. 0. 0.]
     [0. 1. 0.]
     [0. 0. 1.]]

D = [[0. 0.]
     [0. 0.]
     [0. 0.]]

[[ 1.00000000e+00  3.80504356e-07 -7.21956271e-08]
 [-1.20326045e-07  3.16227766e+00  4.36734083e+00]]
```

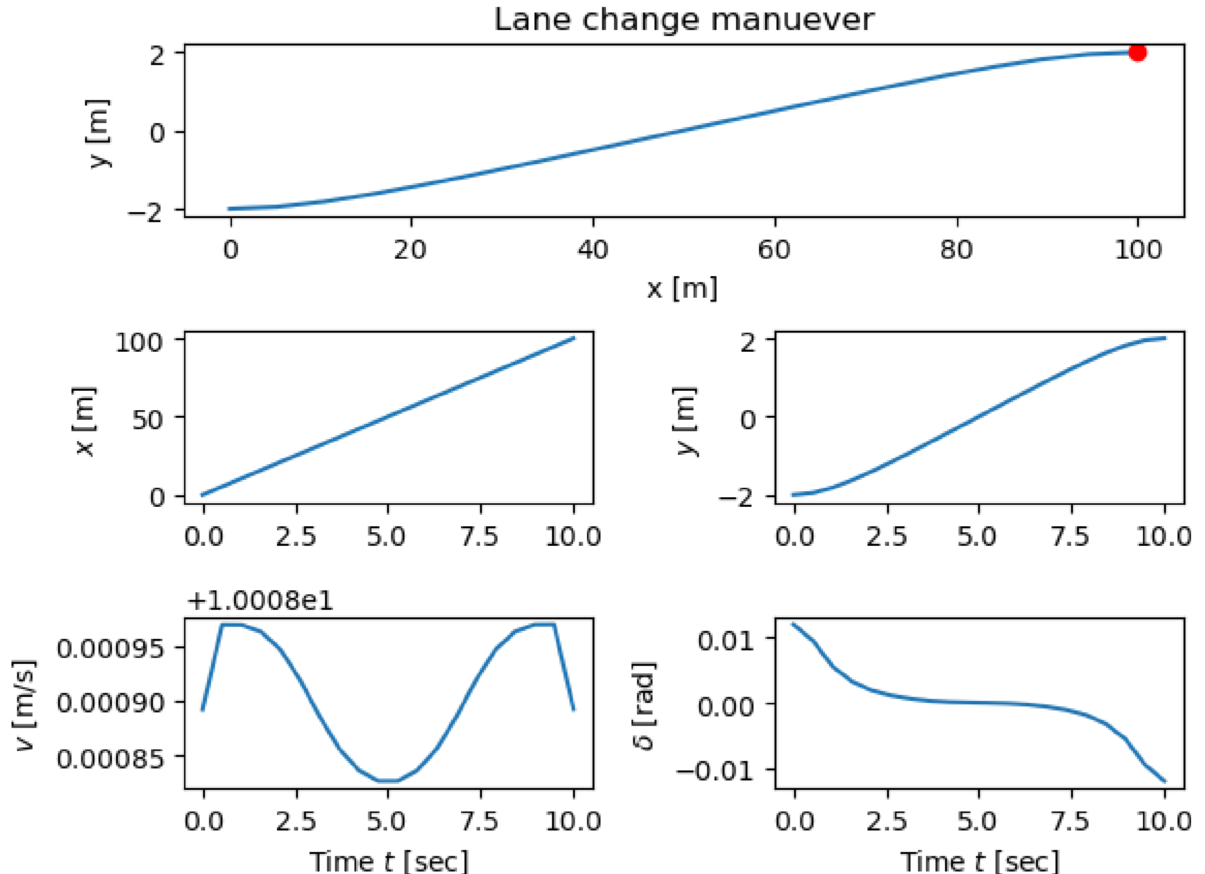
In [4]:
```
# Create the closed loop system using create_statefbk_iosystem
# ?ct.create_statefbk_iosystem
ctrl, clsys = ct.create_statefbk_iosystem(
    kincar, K, xd_labels=['xd', 'yd', 'thetad'], ud_labels=['vd', 'deltad'])
print(clsys)
```

```
<InterconnectedSystem>: kincar_sys[1]
Inputs (5): ['xd', 'yd', 'thetad', 'vd', 'deltad']
Outputs (5): ['x', 'y', 'theta', 'v', 'delta']
States (3): ['kincar_x', 'kincar_y', 'kincar_theta']

Update: <function InterconnectedSystem.__init__.<locals>.updfcn at 0x144a6ef
c0>
Output: <function InterconnectedSystem.__init__.<locals>.outfcn at 0x144a6ef
20>
```

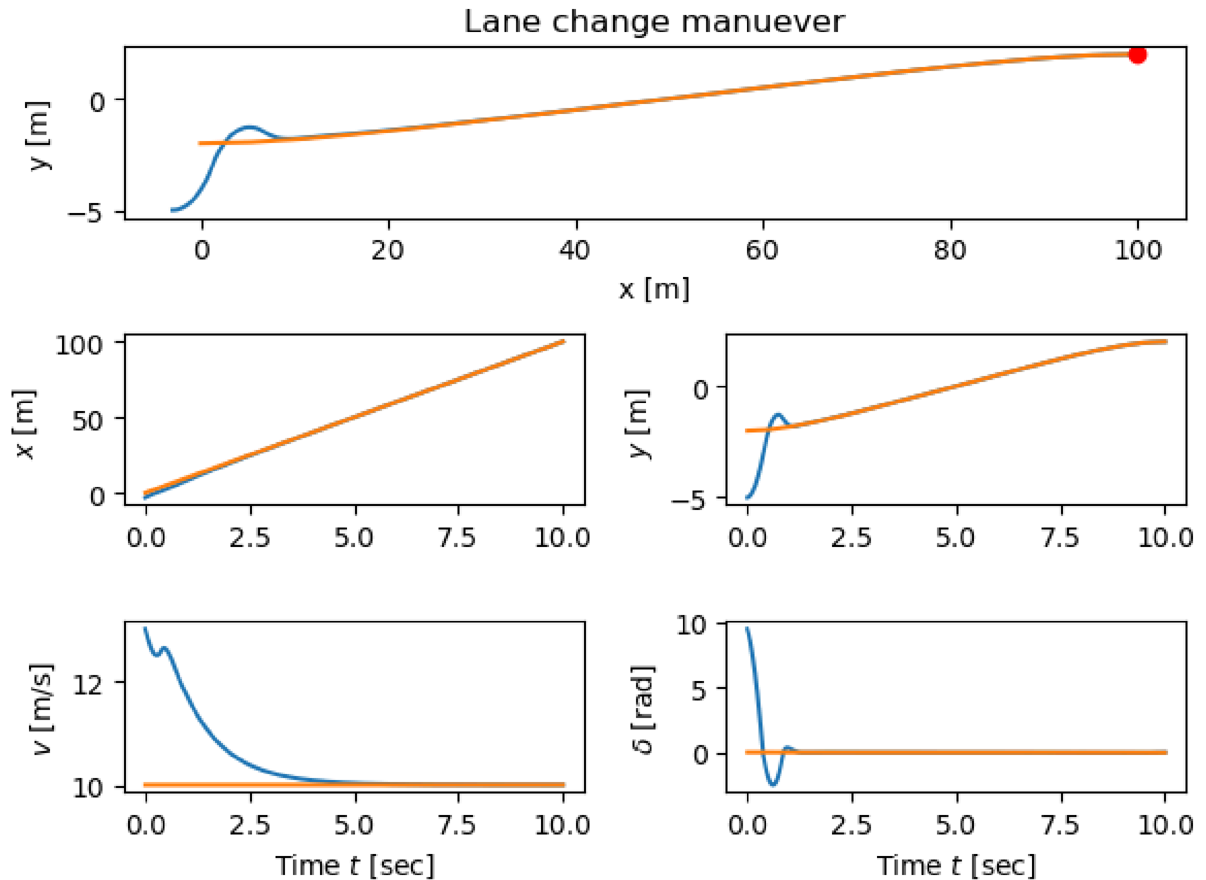
```
/Users/murray/miniconda3/envs/cds110/lib/python3.12/site-packages/control/st
atefbk.py:783: UserWarning: cannot verify system output is system state
  warnings.warn("cannot verify system output is system state")
```

In [5]:
```
# Create a trajectory corresponding to a slow lane change
x0 = np.array([0, -2, 0]); u0 = [10, 0]
xf = np.array([100, 2, 0])
Tf = 10
timepts = np.linspace(0, Tf, 20)

straight_line = (                 # straight line from start to end with nomin
    np.array([x0 + (xf - x0) * t/Tf for t in timepts]).transpose(),
    u0
)

desired = opt.solve_ocp(
    kincar, timepts, x0,
    cost=opt.quadratic_cost(kincar, None, Qu, u0=u0),
    terminal_constraints=opt.state_range_constraint(kincar, xf, xf),
    initial_guess=straight_line)

plot_lanechange(desired.time, desired.states, desired.inputs, yf=xf)
```

```
Summary statistics:
* Cost function calls: 3435
* Eqconst calls: 3571
* System simulations: 0
* Final cost: 0.0010138325729023509
```



In [6]:
```
# Simulate the system with an initial condition error
# Use t_eval to evaluate at points between inputs
actual = ct.input_output_response(
    clsys, timepts, [desired.states, desired.inputs],
    X0=[-3, -5, 0], t_eval=np.linspace(0, Tf, 500))

plot_lanechange(actual.time, actual.states, actual.outputs[3:])
plot_lanechange(desired.time, desired.states, desired.inputs, yf=xf)
```



Note that the value of δ is very large at the start. This is truncated in the model so that it does not exceed ± 0.5 rad.

Reference trajectory subsystem

In addition to generating a trajectory for the system, we can also create x_d and u_d corresponding to reference inputs r_y and r_v.

The reference trajectory block below generates a simple trajectory for the system given the desired speed (vref) and lateral position (yref). The trajectory consists of a straight line of the form (vref * t, yref, 0) with nominal input (vref, 0).

In [7]:
```
# System state: none
# System input: vref, yref
# System output: xd, yd, thetad, vd, deltad
```

```
# System parameters: none
#
def trajgen_output(t, x, u, params):
    vref, yref = u
    return np.array([vref * t, yref, 0, vref, 0])

# Define the trajectory generator as an input/output system
trajgen = ct.nlsys(
    None, trajgen_output, name='trajgen',
    inputs=('vref', 'yref'),
    outputs=('xd', 'yd', 'thetad', 'vd', 'deltad'))

print(trajgen)
```

```
<NonlinearIOSystem>: trajgen
Inputs (2): ['vref', 'yref']
Outputs (5): ['xd', 'yd', 'thetad', 'vd', 'deltad']
States (0): []

Update: None
Output: <function trajgen_output at 0x145343d80>
```

Step responses

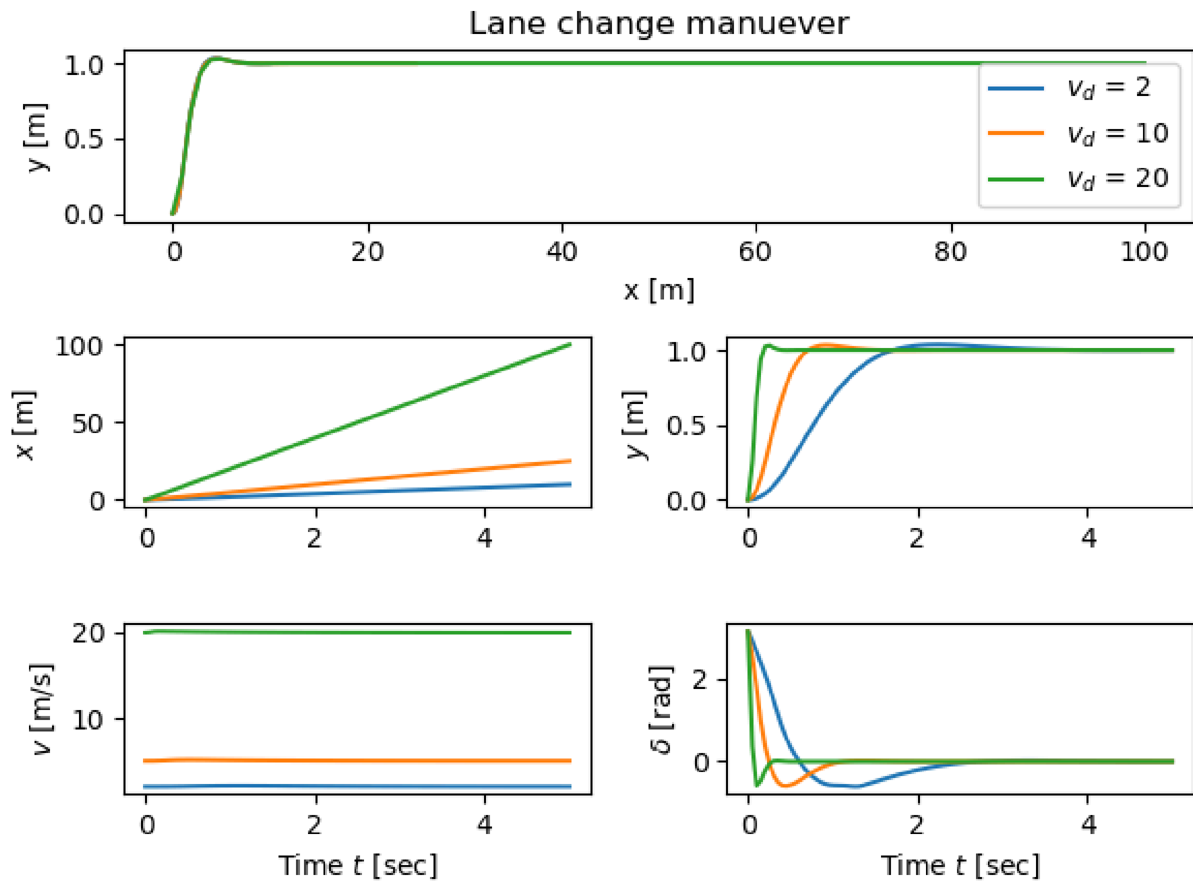
To explore the dynamics of the system, we create a set of lane changes at different forward speeds. Since the linearization depends on the speed, this means that the closed loop performance of the system will vary.

In [8]:
```
steering_fixed = ct.interconnect(
    [kincar, ctrl, trajgen],
    inputs=['vref', 'yref'],
    outputs=kincar.output_labels + kincar.input_labels
)
print(steering_fixed)
```

```
<InterconnectedSystem>: sys[2]
Inputs (2): ['vref', 'yref']
Outputs (5): ['x', 'y', 'theta', 'v', 'delta']
States (3): ['kincar_x', 'kincar_y', 'kincar_theta']

Update: <function InterconnectedSystem.__init__.<locals>.updfcn at 0x1454ec9
a0>
Output: <function InterconnectedSystem.__init__.<locals>.outfcn at 0x1454ec7
20>
```

In [9]:
```
# Set up the simulation conditions
yref = 1
T = np.linspace(0, 5, 100)

# Do an iteration through different speeds
for vref in [2, 5, 20]:
    # Simulate the closed loop controller response
    tout, yout = ct.input_output_response(
        steering_fixed, T, [vref * np.ones(len(T)), yref * np.ones(len(T))],
```

```
        params={'maxsteer': 1})

    # Plot the results
    plot_lanechange(tout, yout, yout[3:])

# Label the different curves
plt.subplot(3, 1, 1)
plt.legend(["$v_d$ = " + f"{vref}" for vref in [2, 10, 20]])
plt.tight_layout()
```



Gain scheduled controller

For this system we use a simple schedule on the forward vehicle velocity and place the poles of the system at fixed values. The controller takes the current and desired vehicle position and orientation plus the velocity velocity as inputs, and returns the velocity and steering commands.

Linearizing the system about the desired trajectory, we obtain

$$A(x_\mathrm{d}) = \left.\frac{\partial f}{\partial x}\right|_{(x_\mathrm{d}, u_\mathrm{d})} = \left.\begin{bmatrix} 0 & 0 & -\sin\theta_\mathrm{d}\, v_\mathrm{d} \\ 0 & 0 & \cos\theta_\mathrm{d}\, v_\mathrm{d} \\ 0 & 0 & 0 \end{bmatrix}\right|_{(x_\mathrm{d}, u_\mathrm{d})} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & v_\mathrm{d} \\ 0 & 0 & 0 \end{bmatrix},$$

$$B(x_\mathrm{d}) = \left.\frac{\partial f}{\partial u}\right|_{(x_\mathrm{d}, u_\mathrm{d})} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & v_\mathrm{d}/l \end{bmatrix}.$$

We see that these matrices depend only on θ_d and v_d, so we choose these as the scheduling variables and design a controller of the form

$$u = u_\mathrm{d} - K(\mu)(x - x_\mathrm{d})$$

where $\mu = (\theta_\mathrm{d}, v_\mathrm{d})$ and we interpolate the gains based on LQR controllers computed at a fixed set of points μ_i.

In [10]:
```
# Define the points for the scheduling variables
gs_speeds = [2, 10, 20]
gs_angles = np.linspace(-pi, pi, 4)

# Create controllers at each scheduling point (
points = [np.array([speed, angle])
          for speed in gs_speeds for angle in gs_angles]
gains = [np.array(ct.lqr(kincar.linearize(
    [0, 0, angle], [speed, 0]), Qx, Qu)[0])
    for speed in gs_speeds for angle in gs_angles]
print(f"{points=}")
print(f"{gains=}")

# Create the gain scheduled system
ctrl_gs, _ = ct.create_statefbk_iosystem(
    kincar, (gains, points), name='controller',
    xd_labels=['xd', 'yd', 'thetad'], ud_labels=['vd', 'deltad'],
    gainsched_indices=['vd', 'theta'], gainsched_method='linear')
print(ctrl_gs)
```

```
points=[array([ 2.        , -3.14159265]), array([ 2.        , -1.0471975
5]), array([2.        , 1.04719755]), array([2.        , 3.14159265]), array
([10.        , -3.14159265]), array([10.        , -1.04719755]), array([10.
,  1.04719755]), array([10.        ,  3.14159265]), array([20.        , -3.1
4159265]), array([20.        , -1.04719755]), array([20.        ,  1.0471975
5]), array([20.        ,  3.14159265])]
gains=[array([[-1.00000000e+00, -2.59398437e-07, -1.23043480e-07],
       [ 8.20289867e-08, -3.16227766e+00,  4.36734083e+00]]), array([[ 0.305
13041, -3.01147046, -0.53921814],
       [ 0.95231058,  0.96490708,  2.76628217]]), array([[ 0.30513017,  3.01
14707 ,  0.53921711],
       [-0.95231065,  0.96490633,  2.76628164]]), array([[-1.00000000e+00, -
2.59398435e-07, -1.23043482e-07],
       [ 8.20289878e-08, -3.16227766e+00,  4.36734083e+00]]), array([[-1.000
00000e+00, -4.35732785e-07, -4.13372416e-08],
       [ 1.37790806e-07, -3.16227766e+00,  4.36734083e+00]]), array([[ 0.676
73285, -2.32815948, -0.44555847],
       [ 0.73622867,  2.14001716,  3.18544973]]), array([[ 0.67673224,  2.32
816123,  0.44555789],
       [-0.73622922,  2.14001525,  3.18544797]]), array([[-1.00000000e+00, -
4.35732785e-07, -4.13372417e-08],
       [ 1.37790805e-07, -3.16227766e+00,  4.36734083e+00]]), array([[-1.000
00000e+00, -4.66709521e-07, -2.21379767e-08],
       [ 1.47586512e-07, -3.16227766e+00,  4.36734083e+00]]), array([[ 0.776
29505, -1.99340411, -0.271618  ],
       [ 0.63036973,  2.4548605 ,  3.26593138]]), array([[ 0.77629448,  1.99
340635,  0.27161771],
       [-0.63037044,  2.45485868,  3.26592947]]), array([[-1.00000000e+00, -
4.66709522e-07, -2.21379765e-08],
       [ 1.47586509e-07, -3.16227766e+00,  4.36734083e+00]])]
<NonlinearIOSystem>: controller
Inputs (8): ['xd', 'yd', 'thetad', 'vd', 'deltad', 'x', 'y', 'theta']
Outputs (2): ['v', 'delta']
States (0): []

Update: <function create_statefbk_iosystem.<locals>._control_update at 0x145
6f5ee0>
Output: <function create_statefbk_iosystem.<locals>._control_output at 0x145
6f6020>
```

System construction

The input to the full closed loop system is the desired lateral position and the desired forward velocity. The output for the system is taken as the full vehicle state plus the velocity of the vehicle.

We construct the system using the `ct.interconnect` function and use signal labels to keep track of everything.

In [11]:
```
steering_gainsched = ct.interconnect(
    [trajgen, ctrl_gs, kincar], name='steering',
    inputs=['vref', 'yref'],
    outputs=kincar.output_labels + kincar.input_labels
```

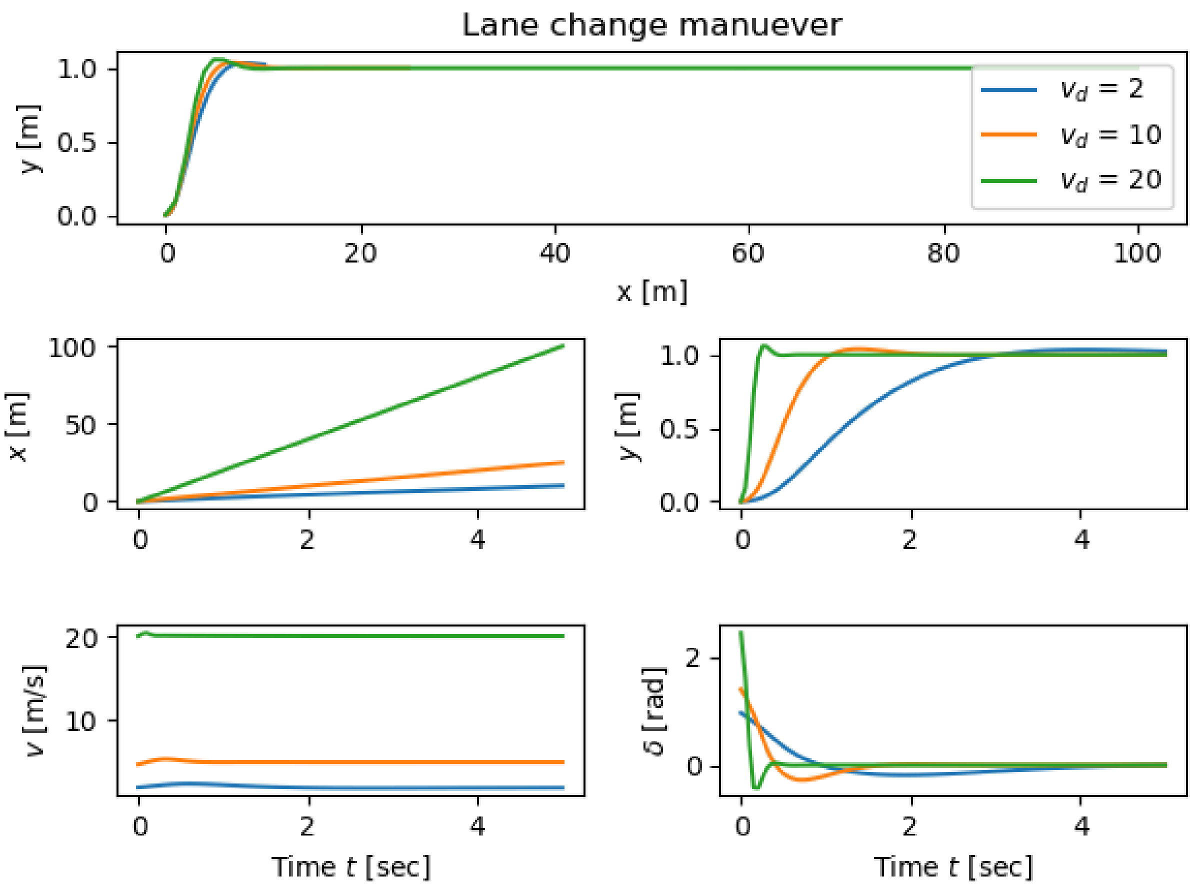
```
)
print(steering_gainsched)
```

```
<InterconnectedSystem>: steering
Inputs (2): ['vref', 'yref']
Outputs (5): ['x', 'y', 'theta', 'v', 'delta']
States (3): ['kincar_x', 'kincar_y', 'kincar_theta']

Update: <function InterconnectedSystem.__init__.<locals>.updfcn at 0x1456f4a
e0>
Output: <function InterconnectedSystem.__init__.<locals>.outfcn at 0x1456f45
40>
```

System simulation

We now simulate the gain scheduled controller for a step input in the y position, using a range of vehicle speeds v_d:

In [12]:
```
# Plot the reference trajectory for the y position
# plt.plot([0, 5], [yref, yref], 'k-', linewidth=0.6)

# Find the signals we want to plot
y_index = steering_gainsched.find_output('y')
v_index = steering_gainsched.find_output('v')

# Do an iteration through different speeds
for vref in [2, 5, 20]:
    # Simulate the closed loop controller response
    tout, yout = ct.input_output_response(
        steering_gainsched, T, [vref * np.ones(len(T)), yref * np.ones(len(T
        X0=[0, 0, 0], params={'maxsteer': 0.5}
    )

    # Plot the results
    plot_lanechange(tout, yout, yout[3:])

# Label the different curves
plt.subplot(3, 1, 1)
plt.legend(["$v_d$ = " + f"{vref}" for vref in [2, 10, 20]])
plt.tight_layout()
```




```
In [ ]:
```