L6-1: Trajectory Generation

Richard M. Murray, 31 Dec 2021 (updated 5 May 2024)

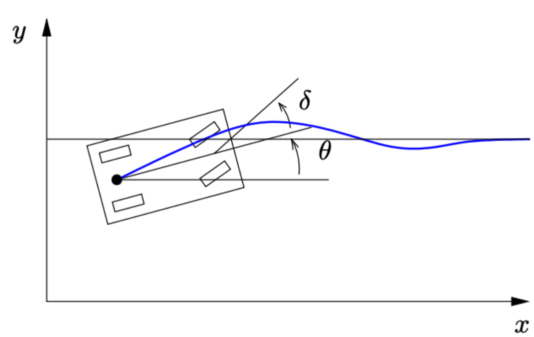
This notebook contains an example of using (optimal) trajectory generation for a vehicle steering system. It illustrates different methods of setting up optimal control problems and solving them using python-control.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import time

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
```

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

Vehicle steering dynamics



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

The vehicle dynamics are given by a simple bicycle model. 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
```

Optimal trajectory generation

The general problem we are solving is of the form:

$$\min_{u(\cdot)} \int_0^T L(x,u)\, dt + V\big(x(T)\big)$$

subject to

$$\dot x = f(x, u), \qquad x\in\mathcal{X} \subset \mathbb{R}^n,\, u\in\mathcal{U} \subset \mathbb{R}^m$$

We consider the problem of changing from one lane to another over a perod of 10 seconds while driving at a forward speed of 10 m/s.

In [3]:
```
# Initial and final conditions
x0 = np.array([  0., -2., 0.]); u0 = np.array([10., 0.])
xf = np.array([100.,  2., 0.]); uf = np.array([10., 0.])
Tf = 10
```

An important part of the optimization procedure is to give a good initial guess. Here are some possibilities:

In [4]:
```
# Define the time horizon (and spacing) for the optimization
# timepts = np.linspace(0, Tf, 5, endpoint=True)
# timepts = np.linspace(0, Tf, 10, endpoint=True)
timepts = np.linspace(0, Tf, 20, endpoint=True)

# Compute some initial guesses to use
bend_left = [10, 0.01]          # slight left veer (will extend over all tim
straight_line = (               # straight line from start to end with nomin
    np.array([x0 + (xf - x0) * t/Tf for t in timepts]).transpose(),
    u0
)
```

Approach 1: standard quadratic cost

We can set up the optimal control problem as trying to minimize the distance from the desired final point while at the same time as not exerting too much control effort to achieve our goal.

$$\min_{u(\cdot)} \int_0^T \left[(x(\tau) - x_\mathrm{f})^T Q_x (x(\tau) - x_\mathrm{f}) + (u(\tau) - u_\mathrm{f})^T Q_u (u(\tau) - u_\mathrm{f})\right] \, d\tau$$

subject to

$$\dot x = f(x, u), \qquad x \in \mathbb{R}^n,\, u \in \mathbb{R}^m$$

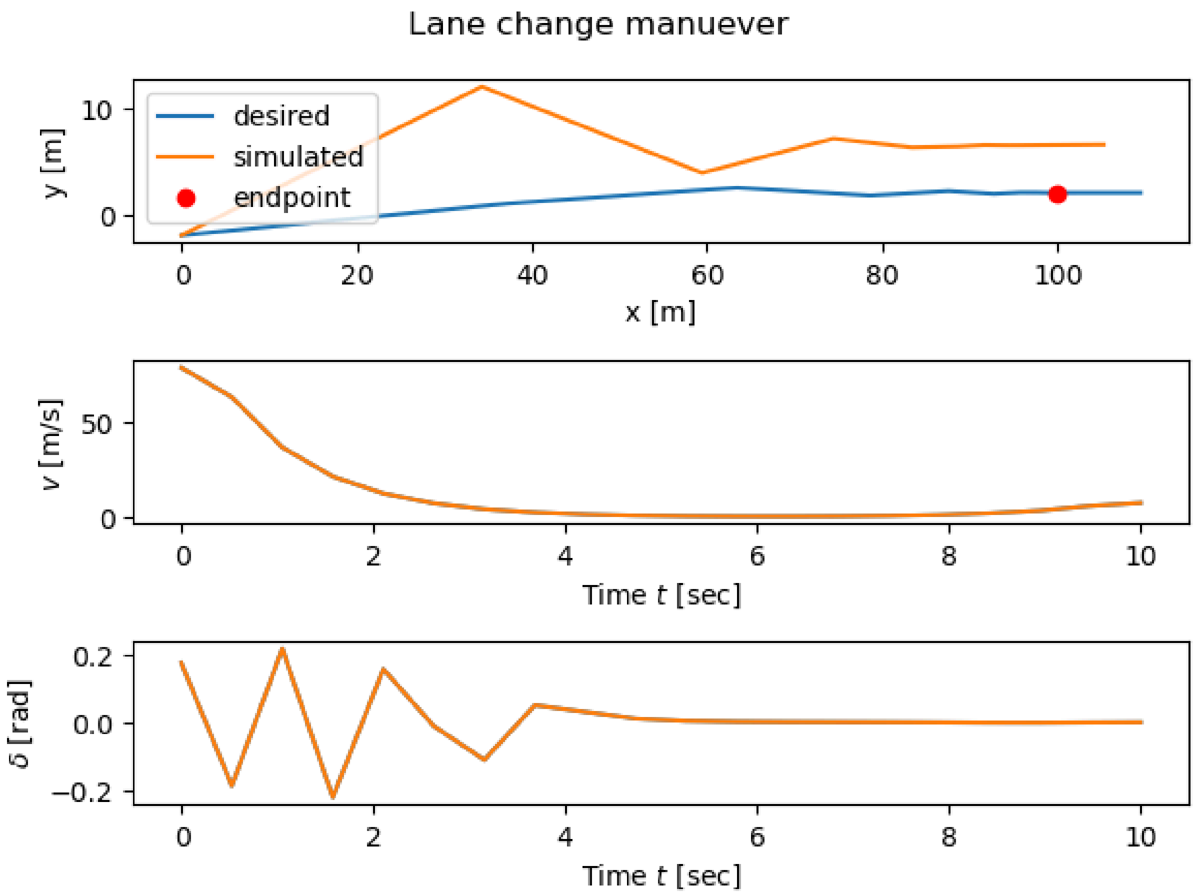
The optimization module solves optimal control problems by choosing the values of the input at each point in the time horizon to try to minimize the cost:

$$u_i(t_j) = \alpha_{i,j}, \qquad u_i(t) = \frac{t_{i+1} - t}{t_{i+1} - t_i} \alpha_{i,j} + \frac{t - t_i}{t_{i+1} - t_i} \alpha_{i+1,j}$$

This means that each input generates a parameter value at each point in the time horizon, so the more refined your time horizon, the more parameters the optimizer has to search over.

In [5]:
```
# Set up the cost functions
Qx = np.diag([.1, 10, .1])       # keep lateral error low
Qu = np.diag([.1, 1])            # minimize applied inputs
quad_cost = opt.quadratic_cost(kincar, Qx, Qu, x0=xf, u0=uf)

# Compute the optimal control, setting step size for gradient calculation (e
start_time = time.process_time()
result1 = opt.solve_ocp(
    kincar, timepts, x0, quad_cost,
    initial_guess=straight_line,
    # initial_guess= bend_left,
    # initial_guess=u0,
    # minimize_method='trust-constr',
    # minimize_options={'finite_diff_rel_step': 0.01},
    # trajectory_method='shooting'
    # solve_ivp_method='LSODA'
)
print("* Total time = %5g seconds\n" % (time.process_time() - start_time))

# Plot the results from the optimization
plot_lanechange(timepts, result1.states, result1.inputs, xf)
print("Final computed state: ", result1.states[:,-1])

# Simulate the system and see what happens
t1, u1 = result1.time, result1.inputs
t1, y1 = ct.input_output_response(kincar, timepts, u1, x0)
plot_lanechange(t1, y1, u1, yf=xf[0:2])
print("Final simulated state:", y1[:,-1])

# Label the different lines
plt.subplot(3, 1, 1)
plt.legend(['desired', 'simulated', 'endpoint'])
plt.tight_layout()
```

```
Summary statistics:
* Cost function calls: 5265
* System simulations: 0
* Final cost: 1001.8795122218194
* Total time = 1.08479 seconds

Final computed state:  [1.09470879e+02 1.99999509e+00 8.74101433e-05]
Final simulated state: [1.05227053e+02 6.51301425e+00 5.69136793e-03]
```



Note the amount of time required to solve the problem and also any warning messages about to being able to solve the optimization (mainly in earlier versions of python-control). You can try to adjust a number of factors to try to get a better solution:

- Try changing the number of points in the time horizon
- Try using a different initial guess
- Try changing the optimization method (see commented out code)

Approach 2: input cost, input constraints, terminal cost

The previous solution integrates the position error for the entire horizon, and so the car changes lanes very quickly (at the cost of larger inputs). Instead, we can penalize the final state and impose a higher cost on the inputs, resulting in a more gradual lane change.

$$\min_{u(\cdot)} \int_0^T \underbrace{\left[x(\tau)^T Q_x x(\tau) + (u(\tau) - u_\mathrm{f})^T Q_u (u(\tau) - u_\mathrm{f})\right]}_{L(x,u)} d\tau + \underbrace{(x(T) - x_\mathrm{f})^T Q_\mathrm{f} (x(T) - x}_{V(x(T))}$$

subject to

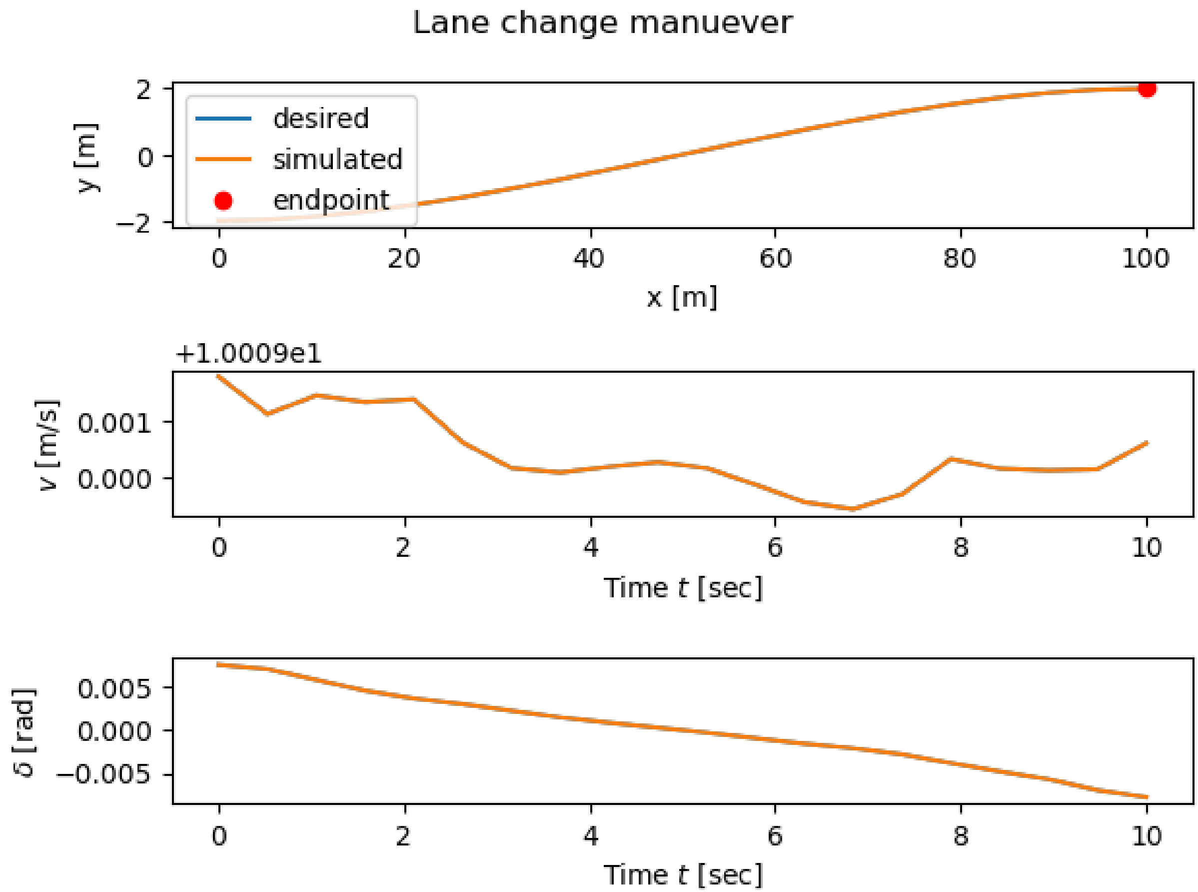
$$\dot{x} = f(x, u), \qquad x \in \mathbb{R}^n,\, u \in \mathbb{R}^m$$

We can also try using a different solver for this example. You can pass the solver using the `minimize_method` keyword and send options to the solver using the `minimize_options` keyword (which should be set to a dictionary of options).

In [6]:

```
# Add input constraint, input cost, terminal cost
constraints = [ opt.input_range_constraint(kincar, [8, -0.1], [12, 0.1]) ]
traj_cost = opt.quadratic_cost(kincar, None, np.diag([0.1, 1]), u0=uf)
term_cost = opt.quadratic_cost(kincar, np.diag([1, 10, 100]), None, x0=xf)

# Compute the optimal control
start_time = time.process_time()
result2 = opt.solve_ocp(
    kincar, timepts, x0, traj_cost, constraints, terminal_cost=term_cost,
    initial_guess=straight_line,
    # minimize_method='trust-constr',
    # minimize_options={'finite_diff_rel_step': 0.01},
    # minimize_method='SLSQP', minimize_options={'eps': 0.01},
    # log=True,
)
print("* Total time = %5g seconds\n" % (time.process_time() - start_time))

# Plot the results from the optimization
plot_lanechange(timepts, result2.states, result2.inputs, xf)
print("Final computed state: ", result2.states[:,-1])

# Simulate the system and see what happens
t2, u2 = result2.time, result2.inputs
t2, y2 = ct.input_output_response(kincar, timepts, u2, x0)
plot_lanechange(t2, y2, u2, yf=xf[0:2])
print("Final simulated state:", y2[:,-1])

# Label the different lines
plt.subplot(3, 1, 1)
plt.legend(['desired', 'simulated', 'endpoint'], loc='upper left')
plt.tight_layout()
```

```
Summary statistics:
* Cost function calls: 4952
* Constraint calls: 5103
* System simulations: 0
* Final cost: 0.00026522491634759827
* Total time = 1.34648 seconds

Final computed state:  [9.99990914e+01 1.99998888e+00 2.58287939e-05]
Final simulated state: [ 9.99998025e+01  1.99655518e+00 -6.57683013e-04]
```



Approach 3: terminal constraints

We can also remove the cost function on the state and replace it with a terminal *constraint* on the state as well as bounds on the inputs. If a solution is found, it guarantees we get to exactly the final state:

$$\min_{u(\cdot)} \int_0^T \underbrace{(u(\tau) - u_\text{f})^T Q_u (u(\tau) - u_\text{f})}_{L(x,u)} \, d\tau$$

subject to

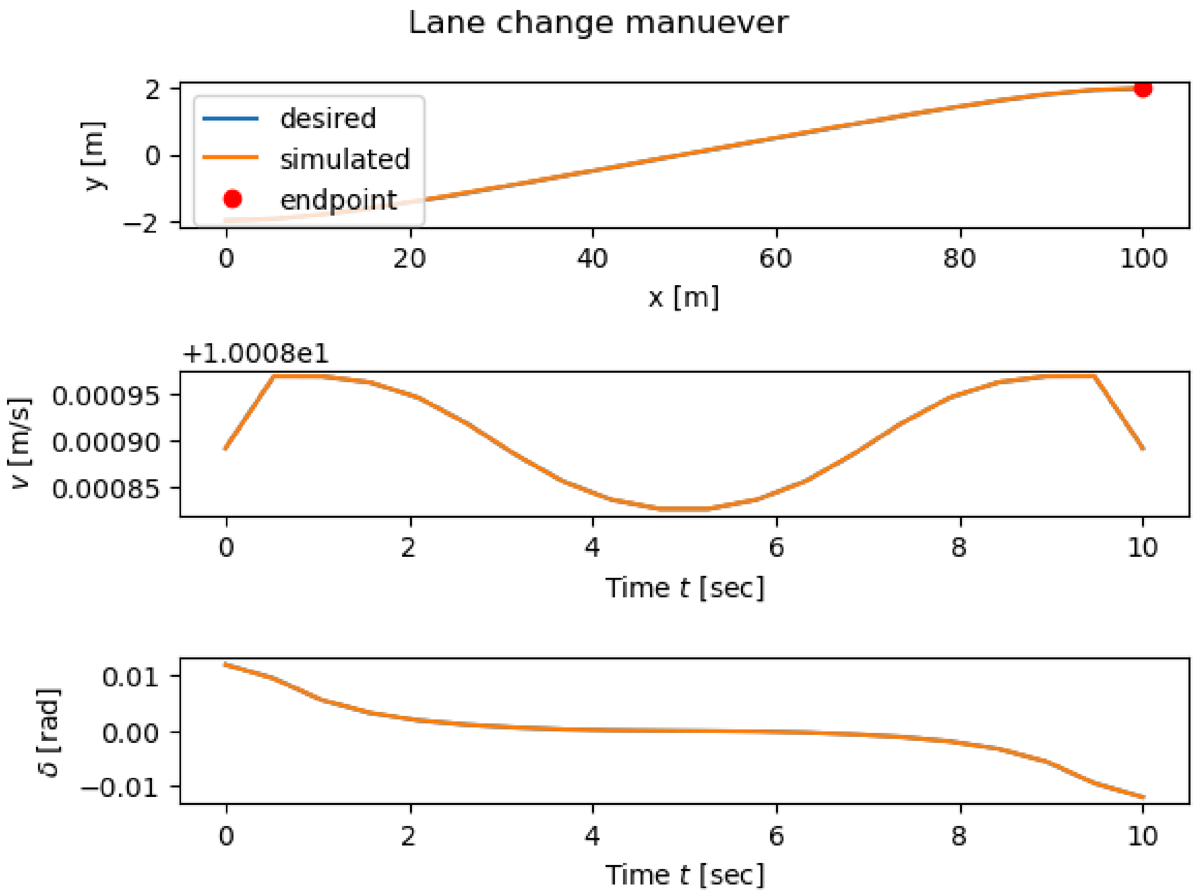
$$\begin{aligned} \dot x &= f(x, u), & \qquad & x \in \mathbb{R}^n,\, u \in \mathbb{R}^m \\ x(T) &= x_\text{f} & & u_\text{lb} \leq u(t) \leq u_\text{ub},\, \text{for all } t \end{aligned}$$

Note that trajectory and terminal constraints can be very difficult to satisfy for a general optimization.

In [7]:
```
# Input cost and terminal constraints
R = np.diag([1, 1])                 # minimize applied inputs
cost3 = opt.quadratic_cost(kincar, np.zeros((3,3)), R, u0=uf)
constraints = [
    opt.input_range_constraint(kincar, [8, -0.1], [12, 0.1]) ]
terminal = [ opt.state_range_constraint(kincar, xf, xf) ]
```

```python
# Compute the optimal control
start_time = time.process_time()
result3 = opt.solve_ocp(
    kincar, timepts, x0, cost3, constraints,
    terminal_constraints=terminal, initial_guess=straight_line,
#     solve_ivp_kwargs={'atol': 1e-3, 'rtol': 1e-2},
#     minimize_method='trust-constr',
#     minimize_options={'finite_diff_rel_step': 0.01},
)
print("* Total time = %5g seconds\n" % (time.process_time() - start_time))

# Plot the results from the optimization
plot_lanechange(timepts, result3.states, result3.inputs, xf)
print("Final computed state: ", result3.states[:,-1])

# Simulate the system and see what happens
t3, u3 = result3.time, result3.inputs
t3, y3 = ct.input_output_response(kincar, timepts, u3, x0)
plot_lanechange(t3, y3, u3, yf=xf[0:2])
print("Final state: ", y3[:,-1])

# Label the different lines
plt.subplot(3, 1, 1)
plt.legend(['desired', 'simulated', 'endpoint'], loc='upper left')
plt.tight_layout()
```

```
Summary statistics:
* Cost function calls: 3435
* Constraint calls: 3571
* Eqconst calls: 3571
* System simulations: 0
* Final cost: 0.0010138325759370772
* Total time = 1.31754 seconds

Final computed state:  [100.   2.   0.]
Final state:  [ 1.00000284e+02  2.01110692e+00 -7.83697871e-04]
```



Approach 4: terminal constraints w/ basis functions (if time)

As a final example, we can use a basis function to reduce the size of the problem and get faster answers with more temporal resolution:

$$\min_{u(\cdot)} \int_0^T L(x,u)\, d\tau + V\left(x(T)\right)$$

subject to

$$\begin{aligned} \dot x &= f(x, u), \qquad x \in \mathcal{X} \subset \mathbb{R}^n,\, u \in \mathcal{U} \subset \mathbb{R}^m \\ u(t) &= \sum_i \alpha_i \phi^i(t), \end{aligned}$$

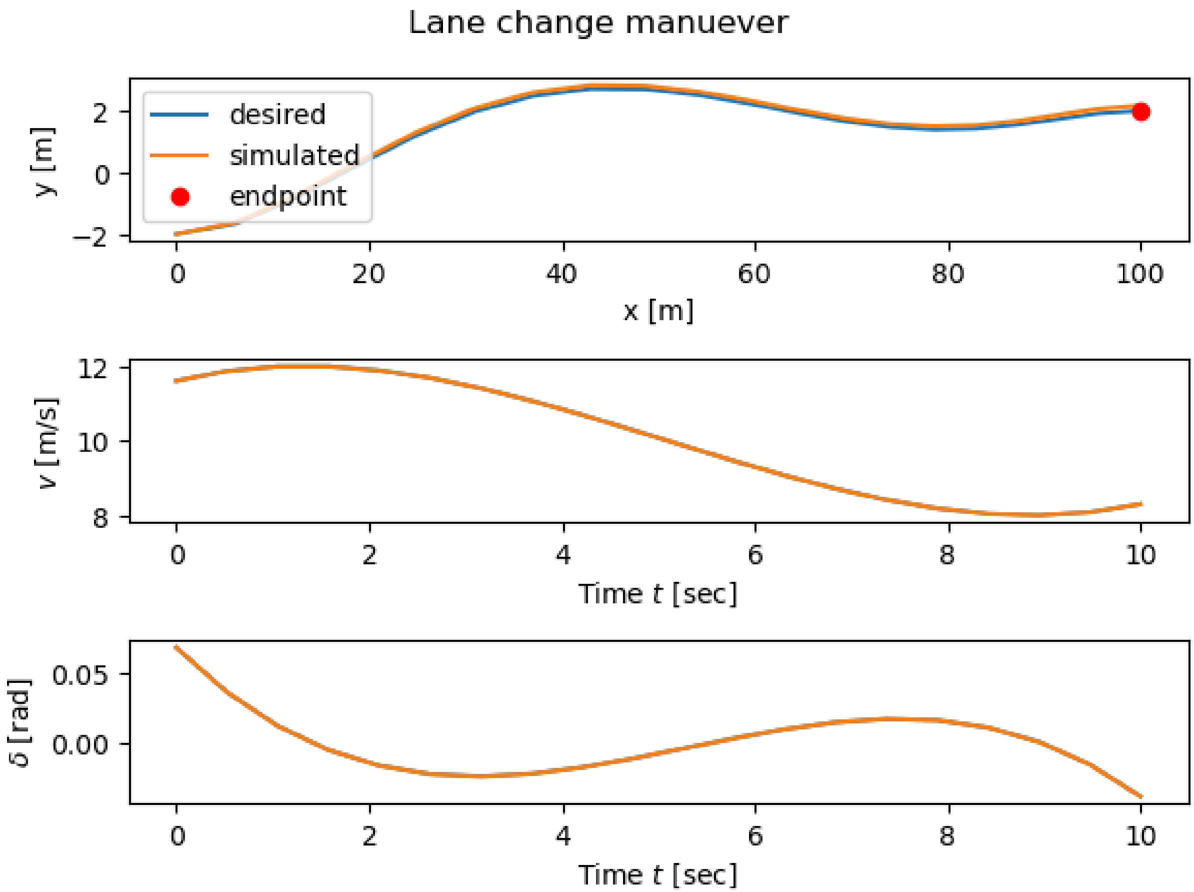
where $\phi^i(t)$ are a set of basis functions.

Here we parameterize the input by a set of 4 Bezier curves but solve for a much more time resolved set of inputs. Note that while we are using the `control.flatsys` module to define the basis functions, we are not exploiting the fact that the system is differentially flat.

In [8]:
```
# Get basis functions for flat systems module
import control.flatsys as flat
```

```python
# Compute the optimal control
start_time = time.process_time()
result4 = opt.solve_ocp(
    kincar, timepts, x0, quad_cost, constraints,
    terminal_constraints=terminal,
    initial_guess=straight_line,
    basis=flat.PolyFamily(4, T=Tf),
    # solve_ivp_kwargs={'method': 'RK45', 'atol': 1e-2, 'rtol': 1e-2},
    # solve_ivp_kwargs={'atol': 1e-3, 'rtol': 1e-2},
    # minimize_method='trust-constr', minimize_options={'disp': True},
    log=False
)
print("* Total time = %5g seconds\n" % (time.process_time() - start_time))

# Plot the results from the optimization
plot_lanechange(timepts, result4.states, result4.inputs, xf)
print("Final computed state: ", result3.states[:,-1])

# Simulate the system and see what happens
t4, u4 = result4.time, result4.inputs
t4, y4 = ct.input_output_response(kincar, timepts, u4, x0)
plot_lanechange(t4, y4, u4, yf=xf[0:2])
print("Final simulated state: ", y4[:,-1])

# Label the different lines
plt.subplot(3, 1, 1)
plt.legend(['desired', 'simulated', 'endpoint'], loc='upper left')
plt.tight_layout()
```

```
Summary statistics:
* Cost function calls: 967
* Constraint calls: 1051
* Eqconst calls: 1051
* System simulations: 0
* Final cost: 3138.6461924295
* Total time = 4.21479 seconds

Final computed state:  [100.   2.   0.]
Final simulated state:  [9.99839202e+01 2.17438528e+00 2.42643089e-03]
```



Note how much smoother the inputs look, although the solver can still have a hard time satisfying the final constraints, resulting in longer computation times.

Additional things to try

- Compare the results here with what we go last week exploiting the property of differential flatness (computation time, in particular)
- Try using different weights, solvers, initial guess and other properties and see how things change.
- Try using different values for `initial_guess` to get faster convergence and/or different classes of solutions.

In [ ]:

```

```