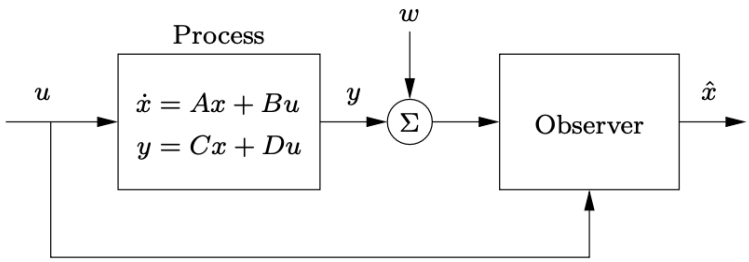
State Estimation

CDS 110/ChE 105, Winter 2024
Richard M. Murray

In this lecture, we will show how to construct an observer for a system in the presence of noise and distrubances.

Recall that an observer is a system that takes as input the (noisy) measured output of a system along with the applied input to the system, and produces as estimate $\hat{x}$ of the current state:



In [1]:
```
# Import the various Python packages that we require
import numpy as np
import matplotlib.pyplot as plt

from math import pi, sin, cos, tan

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

# Need python-control module for generating trajectories
import control.flatsys as fs
```

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

White noise

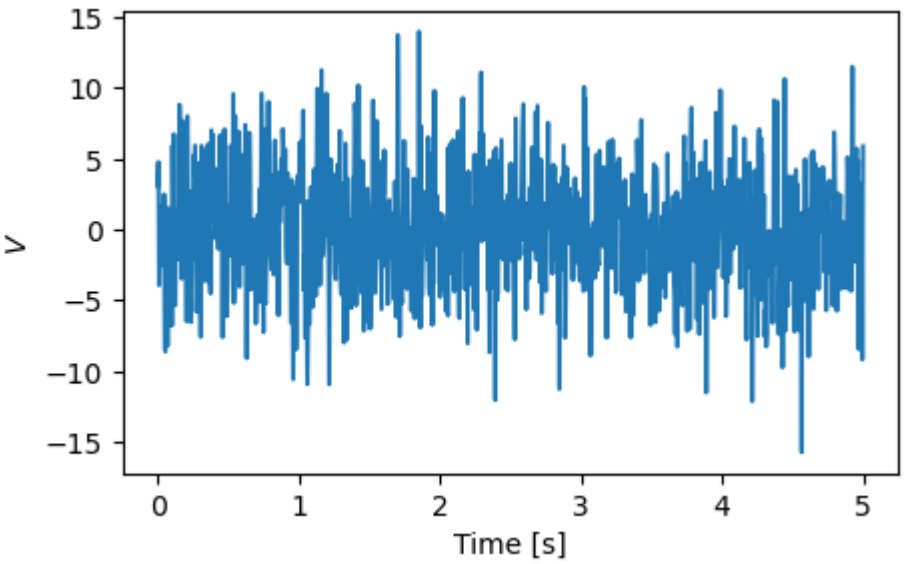
A white noise process $W(t)$ is a signal that has the property that the mean of the signal is 0 and the value of the signal at any point in time t is uncorrelated to the value of the signal at a point in time s, but that has a fixed amount of variance. Mathematically, a white noise process $W(t) \in \mathbb{R}^k$ satisfies

$$\begin{aligned} \mathbb{E}\{W(t)\} &= 0, && \text{for all } t \\ \mathbb{E}\{W^{\mathrm{T}}(t)W(s)\} &= Q\,\delta(t-s) && \text{for all } s, t, \end{aligned}$$

where $Q \in \mathbb{R}^{k \times k}$ is the "intensity" of the white noise process.

The python-control function `white_noise` can be used to create an instantiation of a white noise process:

In [2]:
```
# Create the time vector that we want to use
Tf = 5
T = np.linspace(0, Tf, 1000)
dt = T[1] - T[0]

# Create a white noise signal
?ct.white_noise
Q = np.array([[0.1]])
W = ct.white_noise(T, Q)

plt.figure(figsize=[5, 3])
plt.plot(T, W[0])
plt.xlabel('Time [s]')
plt.ylabel('$V$');
```



To confirm this is a white noise signal, we can compute the correlation function

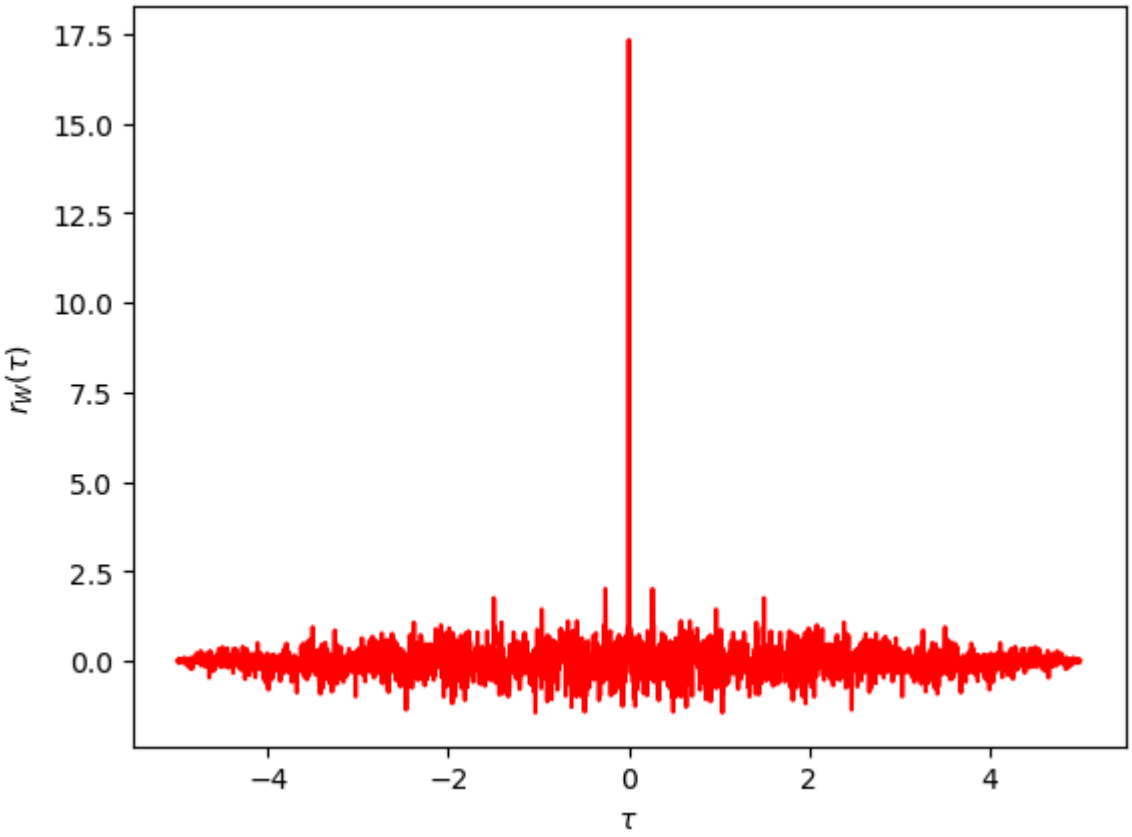
$$\rho(\tau) = \mathbb{E}\{V^{\mathrm{T}}(t)V(t+\tau)\} = Q\,\delta(\tau),$$

where $\delta(\tau)$ is the unit impulse function:

In [3]:
```
# Correlation function for the input
tau, r_W = ct.correlation(T, W)

plt.plot(tau, r_W, 'r-')
plt.xlabel(r'$\tau$')
plt.ylabel(r'$r_W(\tau)$')
```

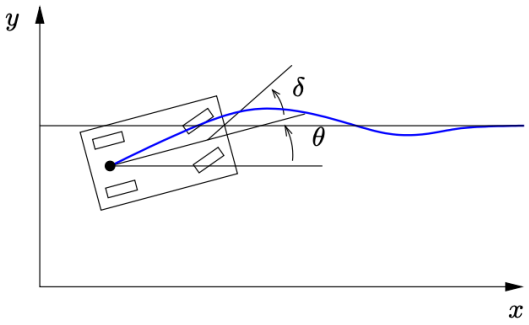
```
# Compute out the area under the peak
print("Signal covariance: ", Q.item())
print("Area under impulse: ", np.max(W) * dt)
```

```
Signal covariance:  0.1
Area under impulse:  0.07007286574805342
```



System definition: kinematic car

We make use of a simple model for a vehicle navigating in the plane, known as the "bicycle model". The kinematics of this vehicle can be written in terms of the contact point (x, y) and the angle θ of the vehicle with respect to the horizontal axis:



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

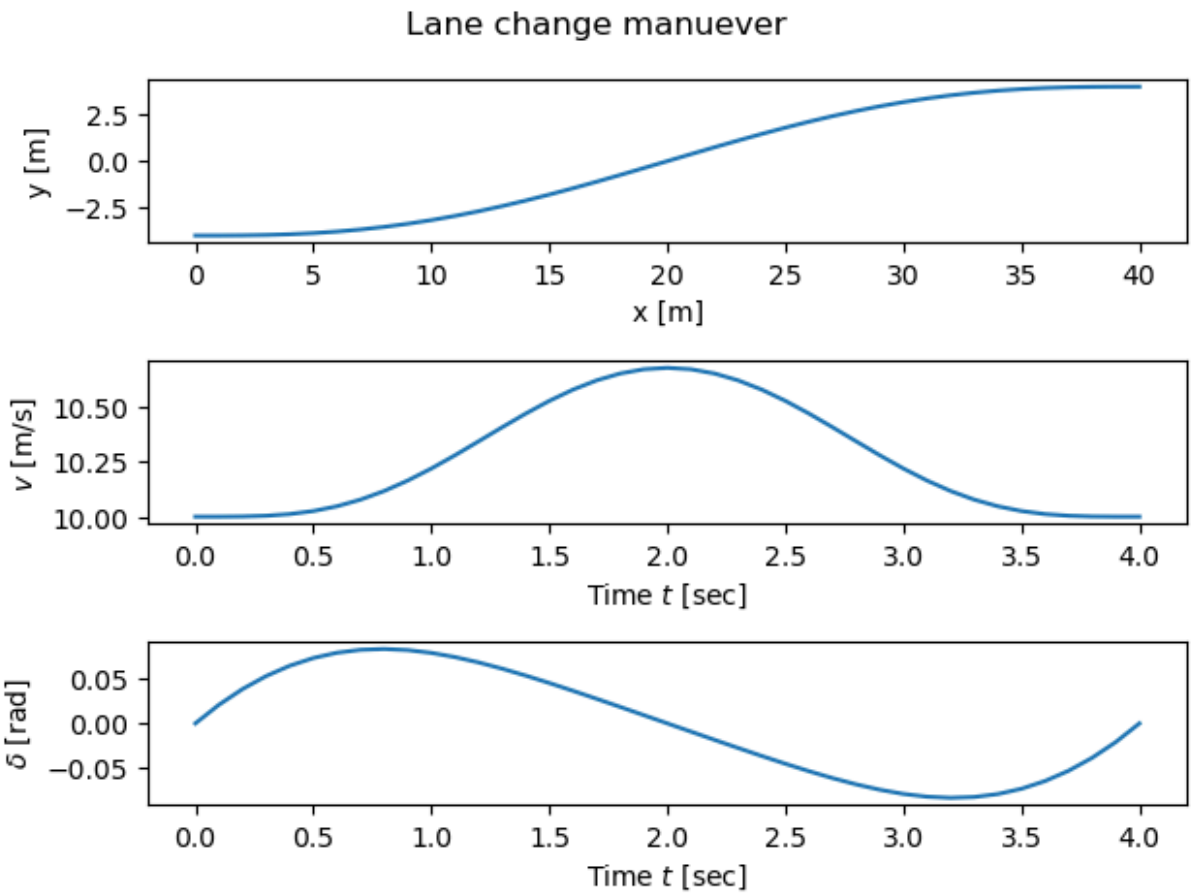
The input v represents the velocity of the vehicle and the input δ represents the turning rate. The parameter l is the wheelbase.

In [4]:
```
# System definition
try:
  from kincar import kincar
except ImportError:
  !wget --no-check-certificate https://www.cds.caltech.edu/~murray/courses/c
  from kincar import kincar
print(kincar)

x0 = np.array([0, 0, 0])
u0 = np.array([10, 0])
```

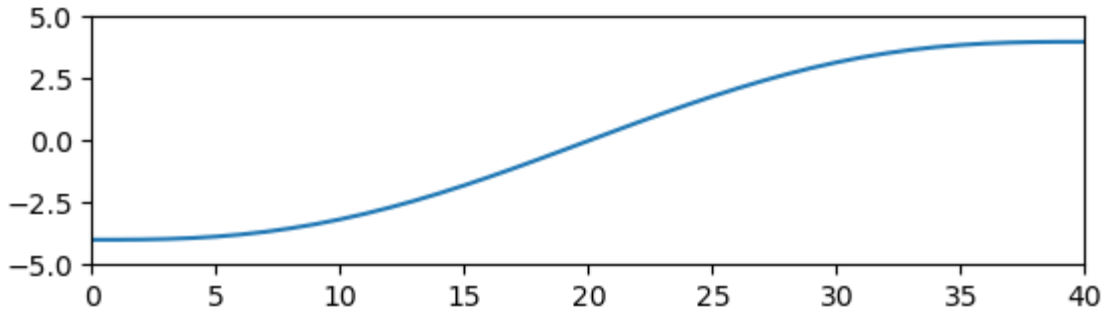
```
<FlatSystem>: kincar
Inputs (2): ['v', 'delta']
Outputs (3): ['x', 'y', 'theta']
States (3): ['x', 'y', 'theta']

Update: <function _kincar_update at 0x143773240>
Output: <function _kincar_output at 0x143771760>

Forward: <function _kincar_flat_forward at 0x1437727a0>
Reverse: <function _kincar_flat_reverse at 0x143773380>
```

We next define a desired trajectory for the vehicle. We will use commands to create a feasible trajectory that we will cover in more detail in W6 of the course:

In [5]:
```
# Generate a trajectory for the vehicle
# Define the endpoints of the trajectory
x0 = [0., -4., 0.]; u0 = [10., 0.]
xf = [40., 4., 0.]; uf = [10., 0.]
Tf = 4

# Find a trajectory between the initial condition and the final condition
traj = fs.point_to_point(kincar, Tf, x0, u0, xf, uf, basis=fs.PolyFamily(6))

# Create the desired trajectory between the initial and final condition
Ts = 0.1
T = np.arange(0, Tf + Ts, Ts)
xd, ud = traj.eval(T)

from kincar import plot_lanechange
plot_lanechange(T, xd, ud)
```



In [6]:
```
# Simulation of the open loop trajectory
sys_resp = ct.input_output_response(kincar, T, ud, xd[:, 0])
plt.plot(sys_resp.states[0], sys_resp.states[1])
plt.axis([0, 40, -5, 5])
plt.gca().set_aspect('equal')
```



State estimation

To illustrate how we can estimate the state of the trajectory, we construct an observer that takes the measured inputs and outputs to the system and computes an estimate of the state, using a estimator with dynamics

$$\dot{\hat{x}} = f(\hat{x}, u) - L(C\hat{x} - y)$$

Note that we go ahead and use the nonlinear dynamics for the prediction term, but the linearization for the correction term.

We can determine the estimator gain L via multiple methods:

- Eigenvalue placement
- Optimal estimation (Kalman filter)

Eigenvalue placement

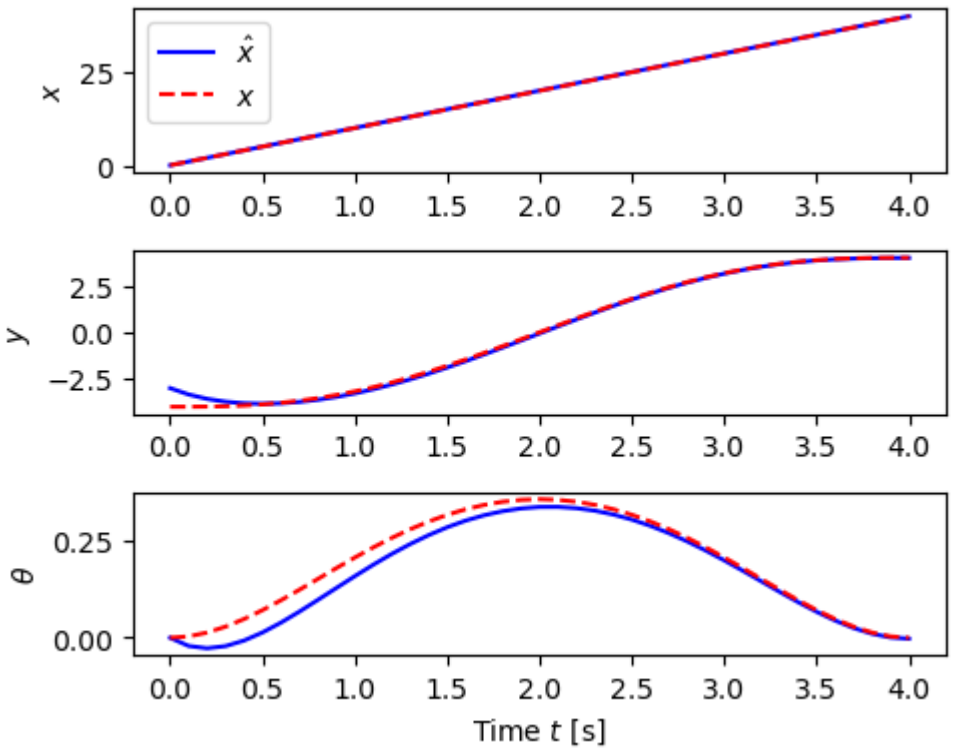
In [7]:
```
# Define the outputs to use for measurements
C = np.eye(2, 3)

# Compute the linearization of the nonlinear dynamics
P = kincar.linearize([0, 0, 0], [10, 0])

# Compute the gains via eigenvalue placement
L = ct.place(P.A.T, C.T, [-1, -2, -3]).T

# Estimator update law
def estimator_update(t, xhat, u, params):
    # Extract the inputs to the estimator
    y = u[0:2]     # first two system outputs
    u = u[2:4]     # inputs that were applied

    # Update the state estimate
    xhatdot = kincar.updfcn(t, xhat, u, params) \
      - params['L'] @ (C @ xhat - y)

    # Return the derivative
    return xhatdot

estimator = ct.nlsys(
    estimator_update, None, name='estimator',
    states=kincar.nstates, params={'L': L},
    inputs= kincar.state_labels[0:2] + kincar.input_labels,
    outputs=[f'xh{i}' for i in range(kincar.nstates)],
)
print(estimator)
```

```
<NonlinearIOSystem>: estimator
Inputs (4): ['x', 'y', 'v', 'delta']
Outputs (3): ['xh0', 'xh1', 'xh2']
States (3): ['x[0]', 'x[1]', 'x[2]']

Update: <function estimator_update at 0x143773e20>
Output: None
```

In [8]:
```
# Run the estimator from a different initial condition
estresp = ct.input_output_response(
    estimator, T, [xd[0:2], ud], [0, -3, 0])

fig, axs = plt.subplots(3, 1, figsize=[5, 4])
```

```
axs[0].plot(estresp.time, estresp.outputs[0], 'b-', T, xd[0], 'r--')
axs[0].set_ylabel("$x$")
axs[0].legend([f"$\hat x$", "$x$"])

axs[1].plot(estresp.time, estresp.outputs[1], 'b-', T, xd[1], 'r--')
axs[1].set_ylabel("$y$")

axs[2].plot(estresp.time, estresp.outputs[2], 'b-', T, xd[2], 'r--')
axs[2].set_ylabel(r"$\theta$")
axs[2].set_xlabel("Time $t$ [s]")

plt.tight_layout()
```

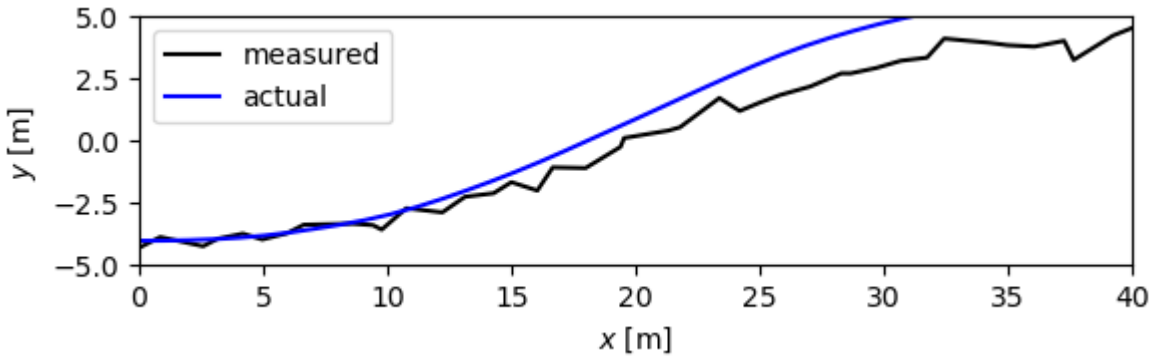
```
<>:9: SyntaxWarning: invalid escape sequence '\h'
<>:9: SyntaxWarning: invalid escape sequence '\h'
/var/folders/3h/8vlrqzts6wnd_p5xvy01zclc0000gn/T/ipykernel_35678/879018274.p
y:9: SyntaxWarning: invalid escape sequence '\h'
  axs[0].legend([f"$\hat x$", "$x$"])
```



Kalman filter

In [9]:
```
# Disturbance and noise covariances
Qv = np.diag([0.1**2, 0.01**2])
Qw = np.eye(2) * 0.1**2

u_noisy = ud + ct.white_noise(T, Qv)
sys_resp = ct.input_output_response(kincar, T, u_noisy, xd[:, 0])

# Create noisy version of the measurements
y_noisy = xd[0:2] + ct.white_noise(T, Qw)

plt.plot(y_noisy[0], y_noisy[1], 'k-')
```

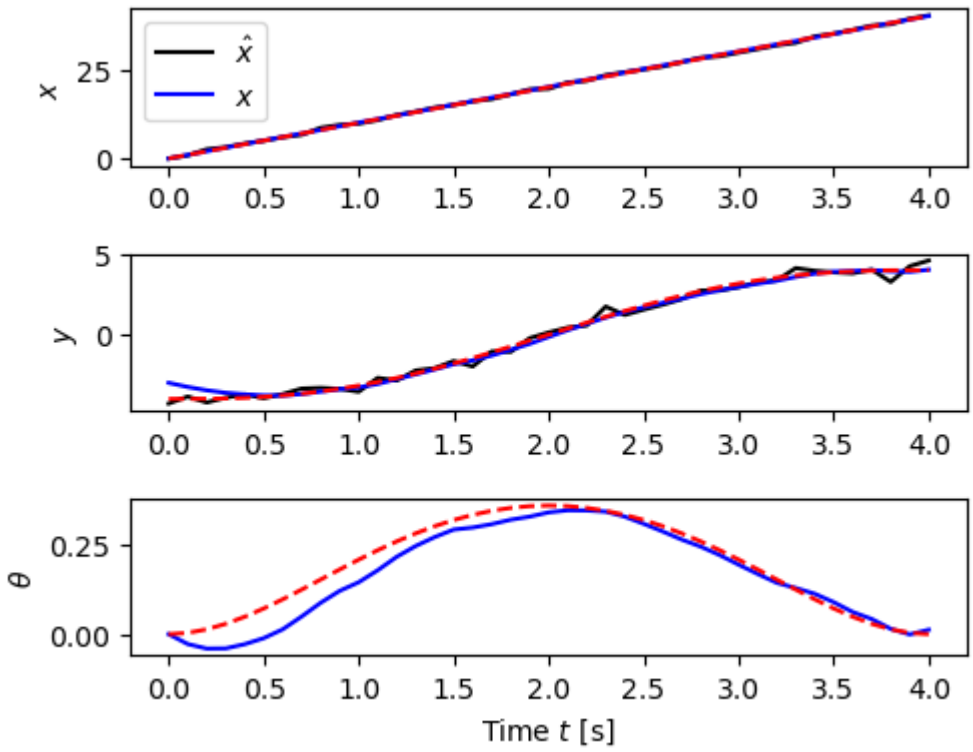
```
plt.plot(sys_resp.outputs[0], sys_resp.outputs[1], 'b-')
plt.axis([0, 40, -5, 5])
plt.xlabel("$x$ [m]")
plt.ylabel("$y$ [m]")
plt.legend(['measured', 'actual'])
plt.gca().set_aspect('equal')
```



In [10]:

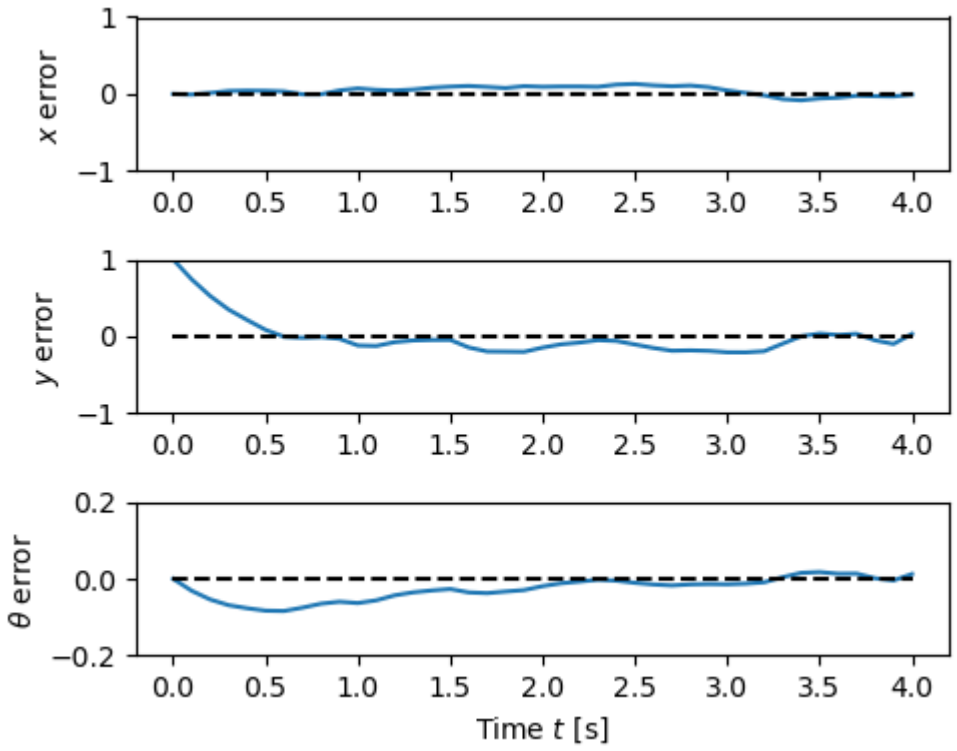
```
# Disturbance and noise covariances
Qv = np.diag([0.1**2, 0.01**2])
Qw = np.eye(2) * 0.1**2

# Compute the Kalman gains (linear quadratic estimator)
L_kf, _, _ = ct.lqe(P.A, P.B, C, Qv, Qw)

kfresp = ct.input_output_response(
    estimator, T, [y_noisy, ud], [0, -3, 0],
    params={'L': L_kf})

fig, axs = plt.subplots(3, 1, figsize=[5, 4])

axs[0].plot(T, y_noisy[0], 'k-')
axs[0].plot(kfresp.time, kfresp.outputs[0], 'b-', T, xd[0], 'r--')
axs[0].set_ylabel("$x$")
axs[0].legend([f"$\hat x$", "$x$"])

axs[1].plot(T, y_noisy[1], 'k-')
axs[1].plot(kfresp.time, kfresp.outputs[1], 'b-', T, xd[1], 'r--')
axs[1].set_ylabel("$y$")

axs[2].plot(kfresp.time, kfresp.outputs[2], 'b-', T, xd[2], 'r--')
axs[2].set_ylabel(r"$\theta$")
axs[2].set_xlabel("Time $t$ [s]")

plt.tight_layout()
```

```
<>:17: SyntaxWarning: invalid escape sequence '\h'
<>:17: SyntaxWarning: invalid escape sequence '\h'
/var/folders/3h/8vlrqzts6wnd_p5xvy01zclc0000gn/T/ipykernel_35678/724642440.p
y:17: SyntaxWarning: invalid escape sequence '\h'
  axs[0].legend([f"$\hat x$", "$x$"])
```



We can get a better view of the convergence by plotting the errors:

In [11]:
```
fig, axs = plt.subplots(3, 1, figsize=[5, 4])

axs[0].plot(kfresp.time, kfresp.outputs[0] - xd[0])
axs[0].plot([T[0], T[-1]], [0, 0], 'k--')
axs[0].set_ylabel("$x$ error")
axs[0].set_ylim([-1, 1])

axs[1].plot(kfresp.time, kfresp.outputs[1] - xd[1])
axs[1].plot([T[0], T[-1]], [0, 0], 'k--')
axs[1].set_ylabel("$y$ error")
axs[1].set_ylim([-1, 1])

axs[2].plot(kfresp.time, kfresp.outputs[2] - xd[2])
axs[2].plot([T[0], T[-1]], [0, 0], 'k--')
axs[2].set_ylabel(r"$\theta$ error")
axs[2].set_xlabel("Time $t$ [s]")
axs[2].set_ylim([-0.2, 0.2])

plt.tight_layout()
```



Output feedback control

We next construct a controller that makes use of the estimated state. We will attempt to control the longitudinal position using the steering angle as an input, with the velocity set to the desired velocity (no tracking of the position).

In [12]:
```
# Compute the linearization of the nonlinear dynamics
P = kincar.linearize([0, 0, 0], [10, 0])

# Extract out the linearized dynamics from delta to y
Alat = P.A[1:3, 1:3]
Blat = P.B[1:3, 1:2]
Clat = P.C[1:2, 1:3]

sys = ct.ss(Alat, Blat, Clat, 0)
print(sys)
```

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

A = [[ 0. 10.]
     [ 0.  0.]]

B = [[0.        ]
     [3.33333333]]

C = [[1. 0.]]

D = [[0.]]
```

In [13]:
```
# Construct a state space controller, using LQR
Qx = np.diag([1, 10])
Qu = np.diag([1])

K, _, _ = ct.lqr(Alat, Blat, Qx, Qu)
print(f"{K=}")

kf = -1 / (Clat @ np.linalg.inv(Alat - Blat @ K) @ Blat)
print(f"{kf=}")
```

```
K=array([[1., 4.]])
kf=array([[1.]])
```

Direct state space feedback

We start by checking the response of the system assuming that we measure the state directly.

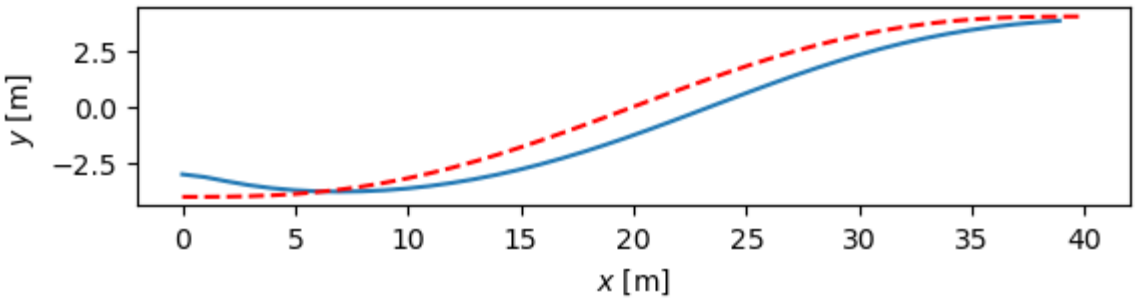
In [14]:
```
# Construct a controller for the full system
def ctrl_output(t, x, u, params):
  r_v, r_y = u[0:2]
  x = u[3:5]                # y, theta
  return np.vstack([r_v, -K @ x + kf * r_y])
ctrl = ct.nlsys(
    None, ctrl_output, name='ctrl',
    inputs=['r_v', 'r_y', 'x', 'y', 'theta'],
    outputs=['v', 'delta']
)
print(ctrl)
```

```
<NonlinearIOSystem>: ctrl
Inputs (5): ['r_v', 'r_y', 'x', 'y', 'theta']
Outputs (2): ['v', 'delta']
States (0): []

Update: None
Output: <function ctrl_output at 0x144680400>
```

In [15]:
```
# Direct state feedback
clsys_direct = ct.interconnect(
    [kincar, ctrl],
    inputs=['r_v', 'r_y'],
    outputs=['x', 'y', 'theta', 'v', 'delta'],
)
print(clsys_direct)
```

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

Update: <function InterconnectedSystem.__init__.<locals>.updfcn at 0x1446800
40>
Output: <function InterconnectedSystem.__init__.<locals>.outfcn at 0x1446804
a0>
```

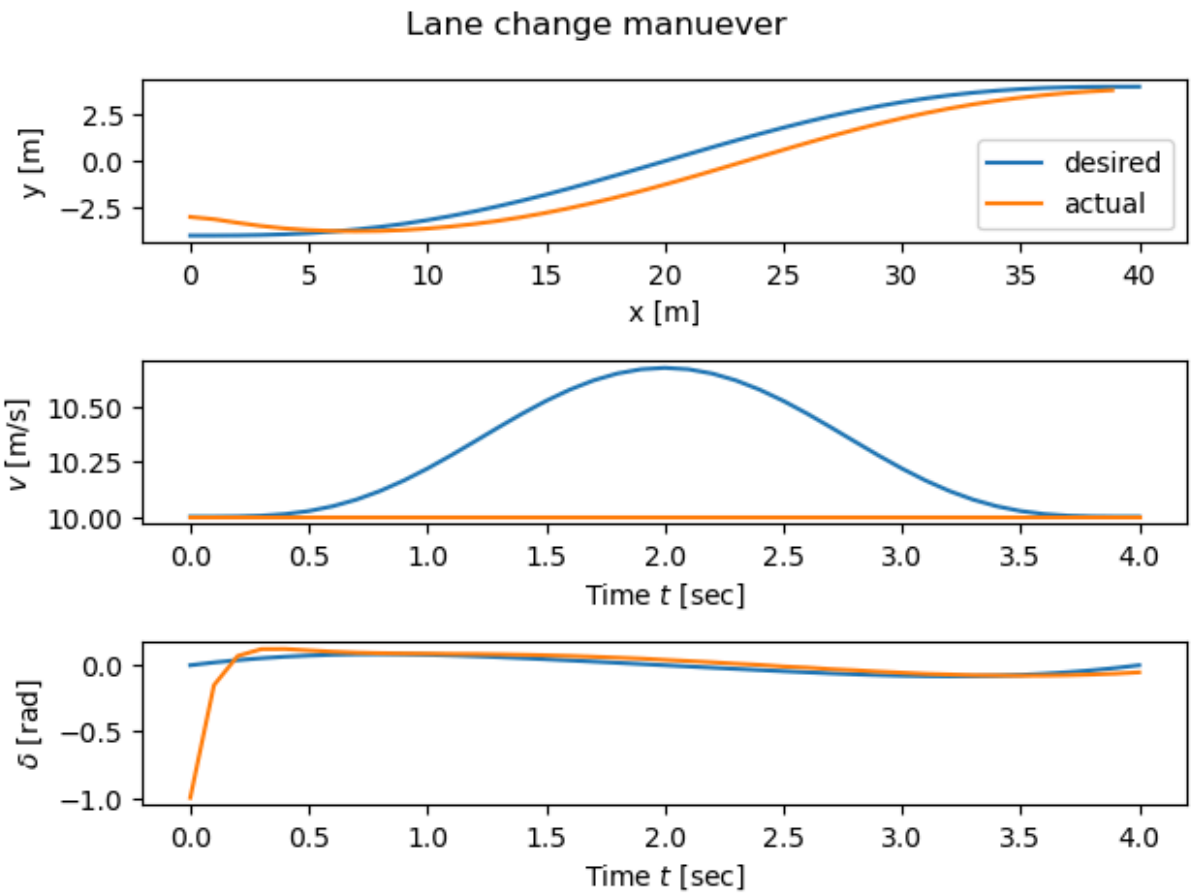
In [16]:
```
# Run a simulation
clresp_direct = ct.input_output_response(
    clsys_direct, T, [10, xd[1]], X0=[0, -3, 0])

plt.plot(clresp_direct.outputs[0], clresp_direct.outputs[1])
plt.plot(xd[0], xd[1], 'r--')
# plt.plot(clresp.time, clresp.outputs[1])
plt.xlabel("$x$ [m]")
plt.ylabel("$y$ [m]")
plt.gca().set_aspect('equal')
```



Note the "lag" in the x coordinate. This comes from the fact that we did not use feedback to maintain the longitudinal position as a function of time, compared with the desired trajectory. To see this, we can look at the commanded speed versus the desired speed:

In [17]:
```
plot_lanechange(T, xd, ud)
plot_lanechange(T, clresp_direct.outputs[0:2], clresp_direct.outputs[-2:])
plt.subplot(3, 1, 1)
plt.legend(['desired', 'actual'], loc='lower right')
```

Out[17]:
```
<matplotlib.legend.Legend at 0x144685d90>
```



From this plot we can also see that there is a very large input δ applied at $t = 0$. This is something we would have to fix if we were to implement this on a physical system (-1 rad $\approx -60^\circ$!).

Estimator-based control

We now consider the case were we cannot directly measure the state, but instead have to estimate the state from the commanded input and measured output. We can insert the estimator into the system model by reconnecting the inputs and outputs. The `ct.interconnect` function provides the needed flexibility:

In [18]:
```
print("ct.interconnect(syslist, connections=None, inplist=None, outlist=None
print(ct.interconnect.__doc__)
```

```
ct.interconnect(syslist, connections=None, inplist=None, outlist=None, param
s=None, check_unused=True, add_unused=False, ignore_inputs=None, ignore_outp
uts=None, warn_duplicate=None, debug=False, **kwargs)

Interconnect a set of input/output systems.

    This function creates a new system that is an interconnection of a set o
f
    input/output systems.  If all of the input systems are linear I/O system
s
    (type :class:`~control.StateSpace`) then the resulting system will be
    a linear interconnected I/O system (type :class:`~control.LinearICSystem
`)
    with the appropriate inputs, outputs, and states.  Otherwise, an
    interconnected I/O system (type :class:`~control.InterconnectedSystem`)
    will be created.

    Parameters
    ----------
    syslist : list of InputOutputSystems
        The list of input/output systems to be connected

    connections : list of connections, optional
        Description of the internal connections between the subsystems:

            [connection1, connection2, ...]

        Each connection is itself a list that describes an input to one of t
he
        subsystems.  The entries are of the form:

            [input-spec, output-spec1, output-spec2, ...]

        The input-spec can be in a number of different forms.  The lowest
        level representation is a tuple of the form `(subsys_i, inp_j)`
        where `subsys_i` is the index into `syslist` and `inp_j` is the
        index into the input vector for the subsystem.  If the signal index
        is omitted, then all subsystem inputs are used.  If systems and
        signals are given names, then the forms 'sys.sig' or ('sys', 'sig')
        are also recognized.  Finally, for multivariable systems the signal
        index can be given as a list, for example '(subsys_i, [inp_j1, ...,
        inp_jn])'; or as a slice, for example, 'sys.sig[i:j]'; or as a base
        name `sys.sig` (which matches `sys.sig[i]`).

        Similarly, each output-spec should describe an output signal from
        one of the subsystems.  The lowest level representation is a tuple
        of the form `(subsys_i, out_j, gain)`.  The input will be
        constructed by summing the listed outputs after multiplying by the
        gain term.  If the gain term is omitted, it is assumed to be 1.  If
        the subsystem index `subsys_i` is omitted, then all outputs of the
        subsystem are used.  If systems and signals are given names, then
        the form 'sys.sig', ('sys', 'sig') or ('sys', 'sig', gain) are also
        recognized, and the special form '-sys.sig' can be used to specify
        a signal with gain -1.  Lists, slices, and base namess can also be
        used, as long as the number of elements for each output spec
        matches the input spec.
```

If omitted, the `interconnect` function will attempt to create the interconnection map by connecting all signals with the same base nam
es
(ignoring the system name). Specifically, for each input signal nam
e
in the list of systems, if that signal name corresponds to the outpu
t
signal in any of the systems, it will be connected to that input (wi
th
a summation across all signals if the output name occurs in more tha
n
one system).

The `connections` keyword can also be set to `False`, which will lea
ve
the connection map empty and it can be specified instead using the low-level :func:`~control.InterconnectedSystem.set_connect_map` method.

inplist : list of input connections, optional
List of connections for how the inputs for the overall system are mapped to the subsystem inputs. The input specification is similar
to
the form defined in the connection specification, except that connections do not specify an input-spec, since these are the system inputs. The entries for a connection are thus of the form:

```
[input-spec1, input-spec2, ...]
```

Each system input is added to the input for the listed subsystem. If the system input connects to a subsystem with a single input, a single input specification can be given (without the inner list).

If omitted the `input` parameter will be used to identify the list of input signals to the overall system.

outlist : list of output connections, optional
List of connections for how the outputs from the subsystems are mapped to overall system outputs. The output connection description is the same as the form defined in the inplist specification (including the optional gain term). Numbered outputs must be chosen from the list of subsystem outputs, but named outputs can also be contained in the list of subsystem inputs.

If an output connection contains more than one signal specification, then those signals are added together (multiplying by the any gain term) to form the system output.

If omitted, the output map can be specified using the :func:`~control.InterconnectedSystem.set_output_map` method.

inputs : int, list of str or None, optional
Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If
an

integer count is specified, the names of the signal will be of the
form `s[i]` (where `s` is one of `u`, `y`, or `x`). If this paramet
er
is not given or given as `None`, the relevant quantity will be
determined when possible based on other information provided to
functions using the system.

outputs : int, list of str or None, optional
Description of the system outputs. Same format as `inputs`.

states : int, list of str, or None, optional
Description of the system states. Same format as `inputs`. The
default is `None`, in which case the states will be given names of t
he
form '<subsys_name>.<state_name>', for each subsys in syslist and ea
ch
state_name of each subsys.

params : dict, optional
Parameter values for the systems. Passed to the evaluation function
s
for the system as default values, overriding internal defaults.

dt : timebase, optional
The timebase for the system, used to specify whether the system is
operating in continuous or discrete time. It can have the following
values:

* dt = 0: continuous time system (default)
* dt > 0: discrete time system with sampling period 'dt'
* dt = True: discrete time with unspecified sampling period
* dt = None: no timebase specified

name : string, optional
System name (used for specifying signals). If unspecified, a generic
name <sys[id]> is generated with a unique integer id.

check_unused : bool, optional
If True, check for unused sub-system signals. This check is
not done if connections is False, and neither input nor output
mappings are specified.

add_unused : bool, optional
If True, subsystem signals that are not connected to other component
s
are added as inputs and outputs of the interconnected system.

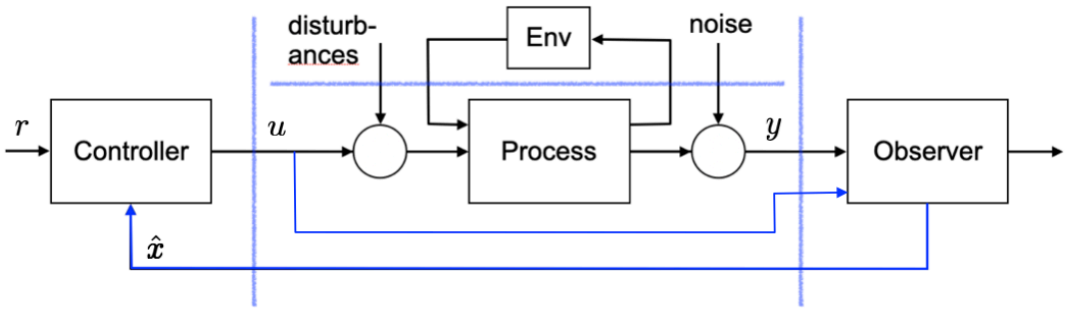
ignore_inputs : list of input-spec, optional
A list of sub-system inputs known not to be connected. This is
only used in checking for unused signals, and does not
disable use of the input.

Besides the usual input-spec forms (see `connections`), an
input-spec can be just the signal base name, in which case all
signals from all sub-systems with that base name are
considered ignored.

```
ignore_outputs : list of output-spec, optional
    A list of sub-system outputs known not to be connected.  This
    is *only* used in checking for unused signals, and does not
    disable use of the output.

    Besides the usual output-spec forms (see `connections`), an
    output-spec can be just the signal base name, in which all
    outputs from all sub-systems with that base name are
    considered ignored.

warn_duplicate : None, True, or False, optional
    Control how warnings are generated if duplicate objects or names are
    detected.  In `None` (default), then warnings are generated for
    systems that have non-generic names.  If `False`, warnings are not
    generated and if `True` then warnings are always generated.

debug : bool, default=False
    Print out information about how signals are being processed that
    may be useful in understanding why something is not working.


Examples
--------
>>> P = ct.rss(2, 2, 2, strictly_proper=True, name='P')
>>> C = ct.rss(2, 2, 2, name='C')
>>> T = ct.interconnect(
...     [P, C],
...     connections=[
...         ['P.u[0]', 'C.y[0]'], ['P.u[1]', 'C.y[1]'],
...         ['C.u[0]', '-P.y[0]'], ['C.u[1]', '-P.y[1]']],
...     inplist=['C.u[0]', 'C.u[1]'],
...     outlist=['P.y[0]', 'P.y[1]'],
... )

This expression can be simplified using either slice notation or
just signal basenames:

>>> T = ct.interconnect(
...     [P, C], connections=[['P.u[:]', 'C.y[:]'], ['C.u', '-P.y']],
...     inplist='C.u', outlist='P.y[:]')

or further simplified by omitting the input and output signal
specifications (since all inputs and outputs are used):

>>> T = ct.interconnect(
...     [P, C], connections=[['P', 'C'], ['C', '-P']],
...     inplist=['C'], outlist=['P'])

A feedback system can also be constructed using the
:func:`~control.summing_block` function and the ability to
automatically interconnect signals with the same names:

>>> P = ct.tf(1, [1, 0], inputs='u', outputs='y')
>>> C = ct.tf(10, [1, 1], inputs='e', outputs='u')
>>> sumblk = ct.summing_junction(inputs=['r', '-y'], output='e')
```

```
>>> T = ct.interconnect([P, C, sumblk], inputs='r', outputs='y')

Notes
-----
If a system is duplicated in the list of systems to be connected,
a warning is generated and a copy of the system is created with the
name of the new system determined by adding the prefix and suffix
strings in config.defaults['iosys.duplicate_system_name_prefix']
and config.defaults['iosys.duplicate_system_name_suffix'], with the
default being to add the suffix '$copy' to the system name.

In addition to explicit lists of system signals, it is possible to
lists vectors of signals, using one of the following forms::

  (subsys, [i1, ..., iN], gain)     signals with indices i1, ..., in
  'sysname.signal[i:j]'             range of signal names, i through j-1
  'sysname.signal[:]'               all signals with given prefix

While in many Python functions tuples can be used in place of lists,
for the interconnect() function the only use of tuples should be in the
specification of an input- or output-signal via the tuple notation
`(subsys_i, signal_j, gain)` (where `gain` is optional).  If you get an
unexpected error message about a specification being of the wrong type
or not being found, check to make sure you are not using a tuple where
you should be using a list.

In addition to its use for general nonlinear I/O systems, the
:func:`~control.interconnect` function allows linear systems to be
interconnected using named signals (compared with the
:func:`~control.connect` function, which uses signal indices) and to be
treated as both a :class:`~control.StateSpace` system as well as an
:class:`~control.InputOutputSystem`.

The `input` and `output` keywords can be used instead of `inputs` and
`outputs`, for more natural naming of SISO systems.
```

We now create the system model that includes the estimator (observer). Here is the system we are trying to construct:

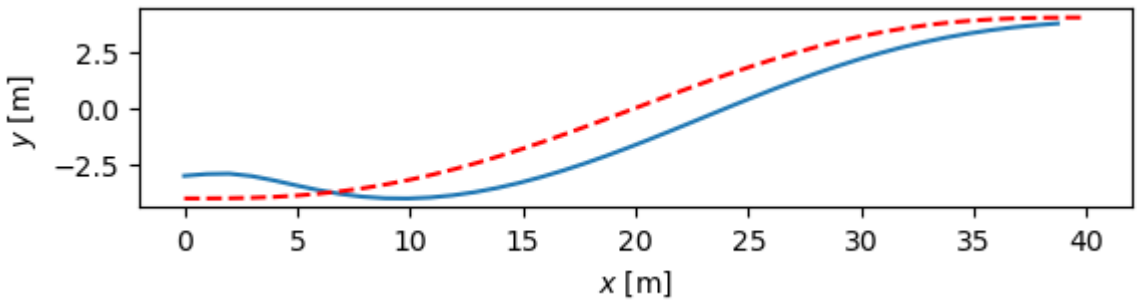


(Be careful with the notation: in the diagram above y is the measured outputs, which for our system are the x and y position of the vehicle, so overusing the symbol y.)

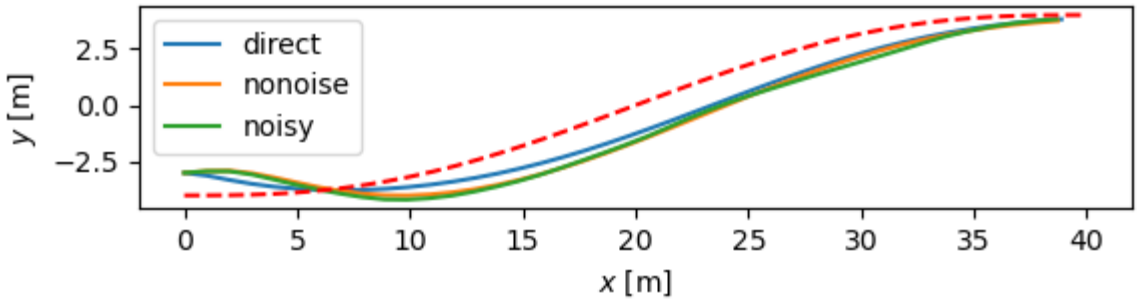
In [19]:
```
# Connect the system, estimator, and controller
clsys_estim = ct.interconnect(
    [kincar, estimator, ctrl],
    inplist=['ctrl.r_v', 'ctrl.r_y', 'estimator.x', 'estimator.y'],
    inputs=['r_v', 'r_y', 'noise_x', 'noise_y'],
    outlist=[
        'kincar.x', 'kincar.y', 'kincar.theta',
        'estimator.xh0', 'estimator.xh1', 'estimator.xh2',
        'ctrl.v', 'ctrl.delta'
    ],
    outputs=['x', 'y', 'theta', 'xhat', 'yhat', 'thhat', 'v', 'delta'],
    connections=[
        ['kincar.v', 'ctrl.v'],
        ['kincar.delta', 'ctrl.delta'],
        ['estimator.x', 'kincar.x'],
        ['estimator.y', 'kincar.y'],
        ['estimator.delta', 'ctrl.delta'],
        ['estimator.v', 'ctrl.v'],
        ['ctrl.x', 'estimator.xh0'],
        ['ctrl.y', 'estimator.xh1'],
        ['ctrl.theta', 'estimator.xh2'],
    ],
)
print(clsys_estim)
```

```
<InterconnectedSystem>: sys[4]
Inputs (4): ['r_v', 'r_y', 'noise_x', 'noise_y']
Outputs (8): ['x', 'y', 'theta', 'xhat', 'yhat', 'thhat', 'v', 'delta']
States (6): ['kincar_x', 'kincar_y', 'kincar_theta', 'estimator_x[0]', 'esti
mator_x[1]', 'estimator_x[2]']

Update: <function InterconnectedSystem.__init__.<locals>.updfcn at 0x1444079
20>
Output: <function InterconnectedSystem.__init__.<locals>.outfcn at 0x144407a
60>
```

In [20]:
```
# Run a simulation with no noise first
clresp_nonoise = ct.input_output_response(
    clsys_estim, T, [10, xd[1], 0, 0], X0=[0, -3, 0, 0, -5, 0])

plt.plot(clresp_nonoise.outputs[0], clresp_nonoise.outputs[1])
plt.plot(xd[0], xd[1], 'r--')

plt.xlabel("$x$ [m]")
plt.ylabel("$y$ [m]")
plt.gca().set_aspect('equal')
```

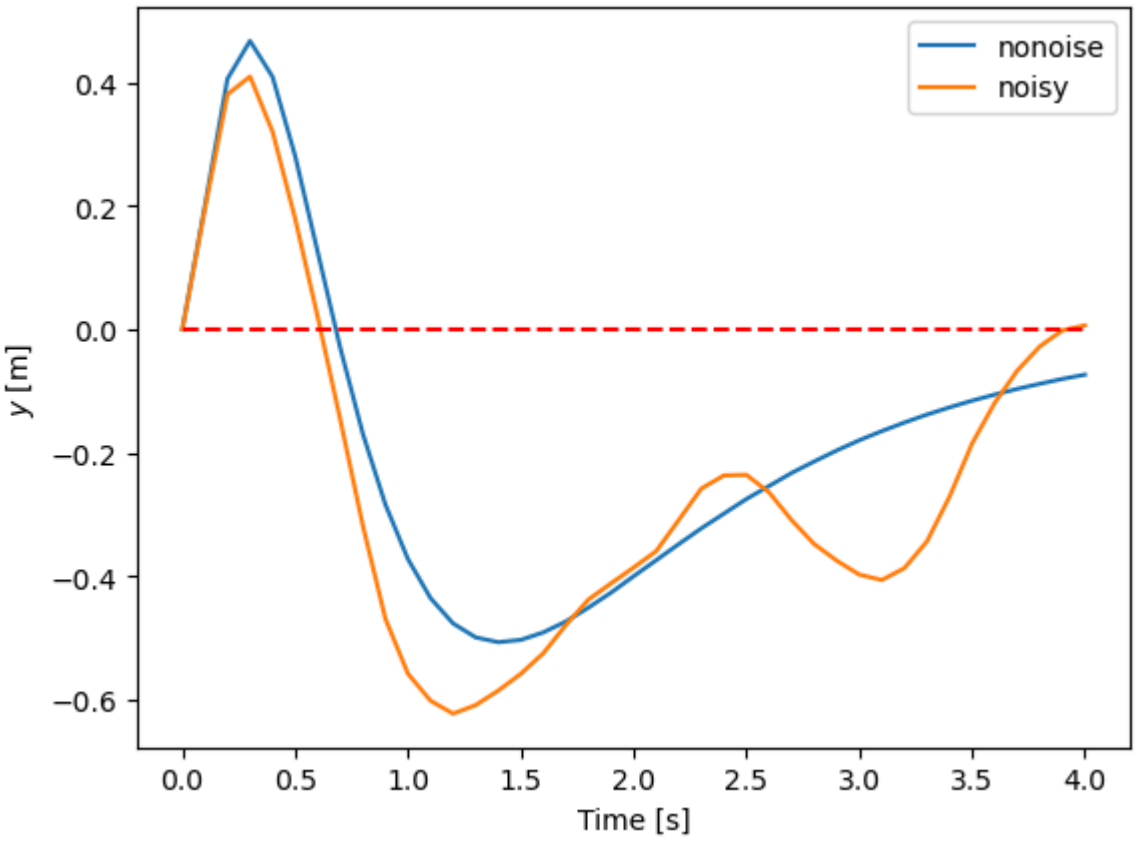


In [21]:
```
# Add some noise
Qv = np.diag([0.1**2, 0.01**2])
Qw = np.eye(2) * 0.1**2

u_noise = ct.white_noise(T, Qv)
y_noise = ct.white_noise(T, Qw)

# Run a simulation
clresp_noisy = ct.input_output_response(
    clsys_estim, T, [10, xd[1], y_noise], X0=[0, -3, 0, 0, -5, 0])

plt.plot(clresp_direct.outputs[0], clresp_direct.outputs[1], label='direct')
plt.plot(clresp_nonoise.outputs[0], clresp_nonoise.outputs[1], label='nonois
plt.plot(clresp_noisy.outputs[0], clresp_noisy.outputs[1], label='noisy')
plt.legend()
plt.plot(xd[0], xd[1], 'r--')

plt.xlabel("$x$ [m]")
plt.ylabel("$y$ [m]")
plt.gca().set_aspect('equal')
```



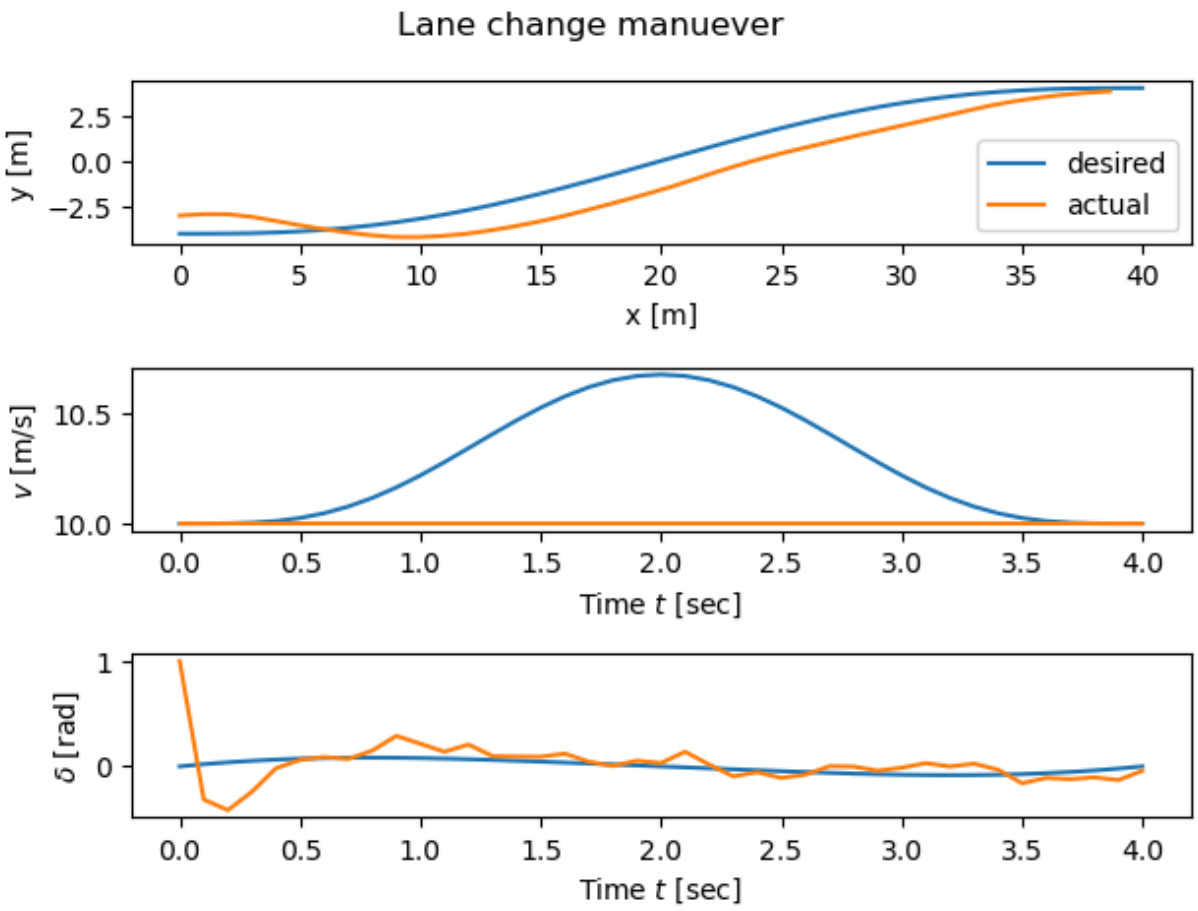
In [22]:
```
# Plot the differences in y to make differences more clear
plt.plot(
    clresp_nonoise.time, clresp_nonoise.outputs[1] - clresp_direct.outputs[1
    label='nonoise')
plt.plot(
    clresp_noisy.time, clresp_noisy.outputs[1] - clresp_direct.outputs[1],
    label='noisy')
plt.legend()
plt.plot([clresp_nonoise.time[0], clresp_nonoise.time[-1]], [0, 0], 'r--')

plt.xlabel("Time [s]")
plt.ylabel("$y$ [m]")
```

Out[22]: Text(0, 0.5, 'y [m]')



In [23]:
```
# Show the control inputs as well as the final trajectory
plot_lanechange(T, xd, ud)
plot_lanechange(T, clresp_noisy.outputs[0:2], clresp_noisy.outputs[-2:])
plt.subplot(3, 1, 1)
plt.legend(['desired', 'actual'], loc='lower right')
```

Out[23]: <matplotlib.legend.Legend at 0x1443907a0>



Things to try

- Wrap a controller around the velocity (or x position) in addition to the lateral (y) position
- Change the amounts of noise in the sensor signal
- Add disturbances to the dynamics (corresponding to wind, hills, etc)

In [ ]: