

Principles of Robot Autonomy I

State space dynamics – computation and simulation



Agenda

- State space dynamics
 - Simulation / numerical integration
 - Efficient computation (auto-differentiation)
- Readings
 - N/A

Simulation

- Suppose we have a state space model for our robot and have fixed $u(t)$, i.e.,
$$\dot{x}(t) = f(x(t), u(t)) = f(x, t)$$
where we “embedded” $u(t)$ within f for simplicity (and keep using “ f ” with a slight abuse of notation). This is an *ordinary differential equation (ODE)* in $x(t)$
- Given an *initial condition* $x(t_0) = x_0$, solving $\dot{x}(t) = f(x(t), t)$ for a *trajectory* $x(t)$ is an *initial value problem (IVP)*
- If f is Lipschitz continuous in x and continuous in t , then the trajectory $x(t)$ exists and is *unique*
- *Simulation* of a system simply means solving an IVP for $x(t)$

Numerical integration

- Simulating a system ODE is done by marching forward in time from the initial condition $x(t_0) = x_0$

- According to the Fundamental Theorem of Calculus,

$$x(t) = x(t_0) + \int_{t_0}^t f(x(\tau), \tau) d\tau = x(t_0) + \sum_{k=0}^{N-1} \int_{t_k}^{t_{k+1}} f(x(\tau), \tau) d\tau$$

for timestamps $t_0 < t_1 < \dots < t_N$ with $t_N = t$. This is *time discretization*

- *Numerical integration* refers to how we compute each discrete step; for example, from t to $t + \Delta t$, we could apply the *Euler step*

$$x(t + \Delta t) \approx x(t) + \Delta t \cdot f(x(t), t)$$

which treats the dynamics as constant from t to $t + \Delta t$

Numerical integration methods

- Euler (just use the slope at the beginning of the interval):

$$x(t + \Delta t) \approx x(t) + \Delta t \cdot f(x(t), t)$$

- Midpoint (use the slope after an Euler step to the middle of the interval):

$$x(t + \Delta t) \approx x(t) + \Delta t \cdot f\left(x(t) + \frac{\Delta t}{2} f(x(t), t), t + \frac{\Delta t}{2}\right)$$

- Runge-Kutta-4 (RK4) (use a weighted average of four slopes across the interval):

$$x(t + \Delta t) \approx x(t) + \frac{\Delta t}{6} (k_1 + 2k_2 + 2k_3 + k_4)$$

$$k_1 = f(x(t), t) \qquad k_2 = f\left(x(t) + \frac{\Delta t}{2} k_1, t + \frac{\Delta t}{2}\right)$$

$$k_3 = f\left(x(t) + \frac{\Delta t}{2} k_2, t + \frac{\Delta t}{2}\right) \qquad k_4 = f(x(t) + \Delta t k_3, t + \Delta t)$$

Truncation error

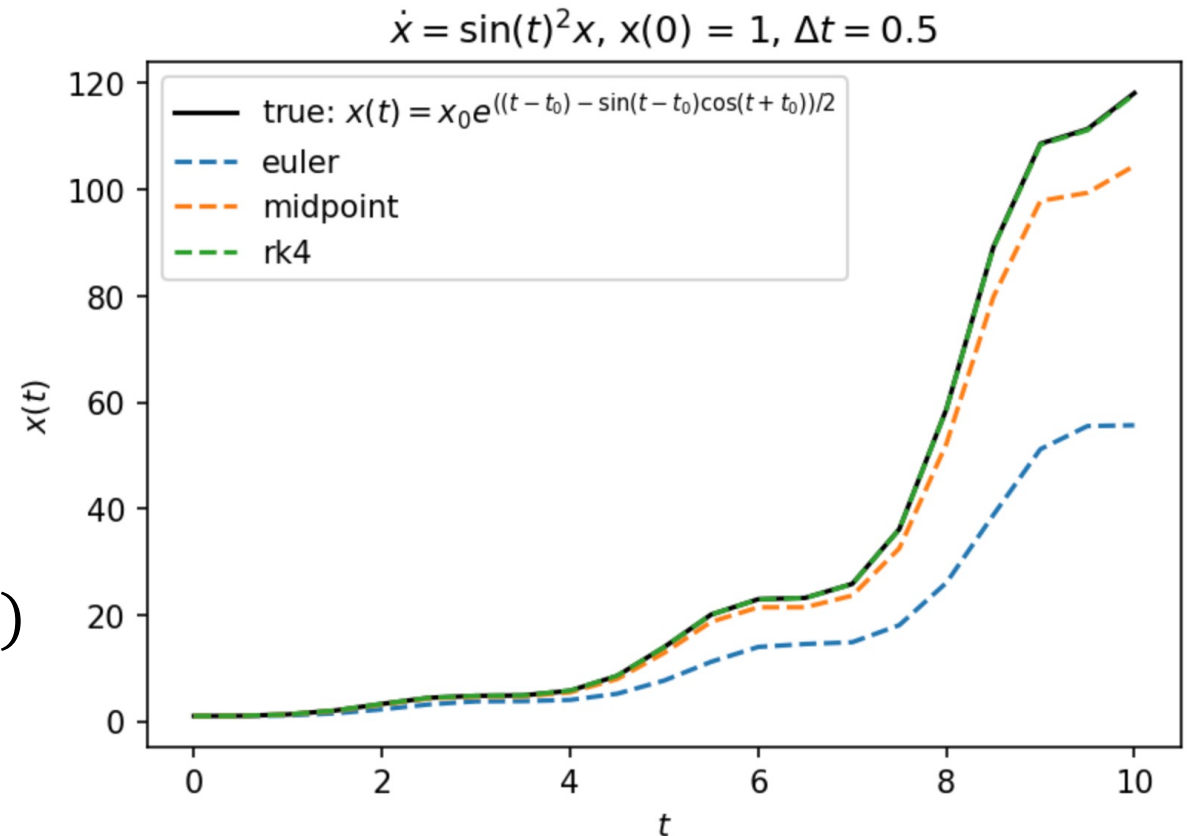
- Higher-order integration schemes use more function evaluations to reduce “local” (i.e., one-step) and “global” (i.e., accumulated) truncation error.

- Local / global truncation errors for each method:

- Euler: $\mathcal{O}(\Delta t^2) / \mathcal{O}(\Delta t)$

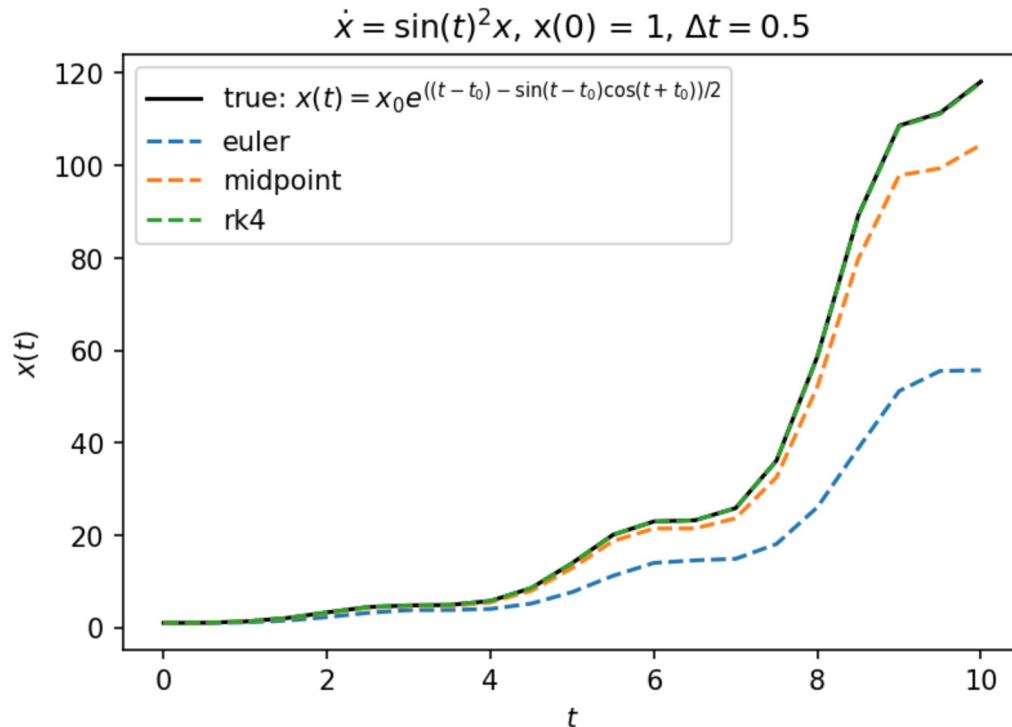
- midpoint: $\mathcal{O}(\Delta t^3) / \mathcal{O}(\Delta t^2)$

- Runge-Kutta (RK4): $\mathcal{O}(\Delta t^5) / \mathcal{O}(\Delta t^4)$



Example: ODE integration “by hand”

```
1 import numpy as np
2
3 # Define an IVP, timestamps, and discretization step
4 f = lambda x, t: x * np.sin(t)**2
5 x0 = 1.
6 t0, tf = 0., 10.
7 dt = 0.5
8 t = np.arange(t0, tf + dt, dt)
9
```

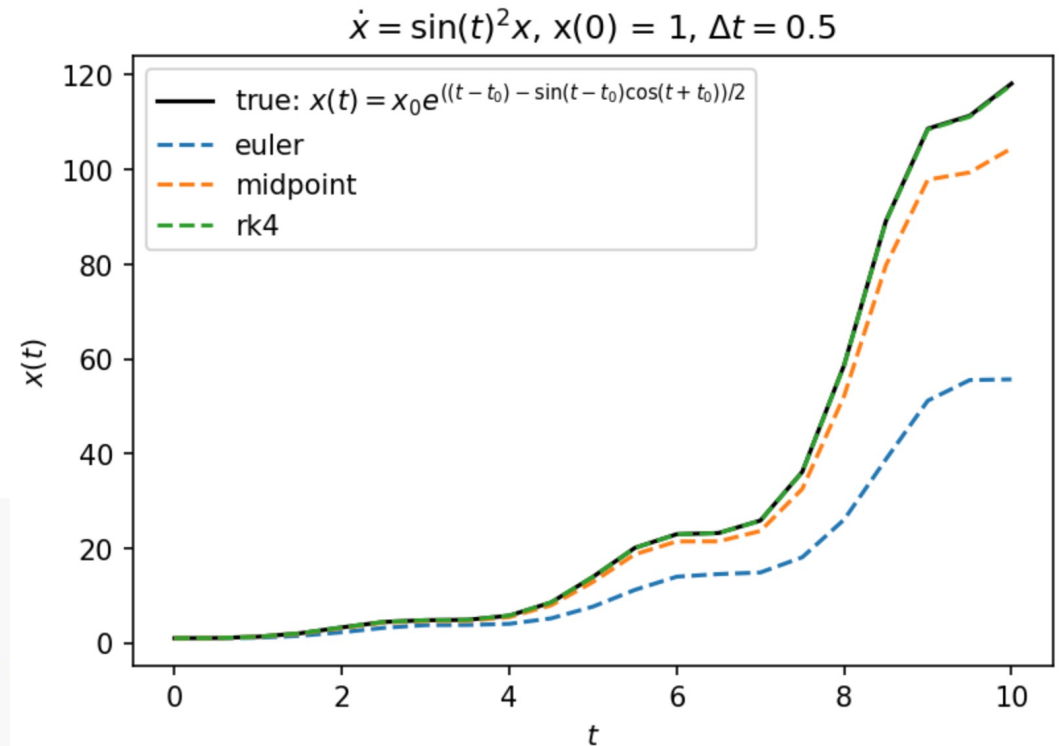


```
10 # Compute true solution for comparison
11 x = {}
12 x['true'] = x0*np.exp(((t-t0) - np.sin(t-t0)*np.cos(t+t0))/2)
13
14 # Loop over timestamps and execute each integration scheme
15 methods = ('true', 'euler', 'midpoint', 'rk4')
16 for m in methods[1:]:
17     x[m] = np.zeros_like(t)
18     x[m][0] = x0
19
20     for i in range(t.size - 1):
21         if m == 'euler':
22             x[m][i + 1] = x[m][i] + dt*f(x[m][i], t[i])
23
24         elif m == 'midpoint':
25             t_mid = t[i] + dt/2
26             x_mid = x[m][i] + (dt/2)*f(x[m][i], t[i])
27             x[m][i + 1] = x[m][i] + dt*f(x_mid, t_mid)
28
29         elif m == 'rk4':
30             k1 = f(x[m][i], t[i])
31             k2 = f(x[m][i] + (dt/2)*k1, t[i] + dt/2)
32             k3 = f(x[m][i] + (dt/2)*k2, t[i] + dt/2)
33             k4 = f(x[m][i] + dt*k3, t[i] + dt)
34             x[m][i + 1] = x[m][i] + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
35
36         else:
37             raise NotImplementedError()
```

Example: ODE integration “by hand”

- Free, open-source plotting functionality is provided by Matplotlib
- Check out the documentation!
<https://matplotlib.org>

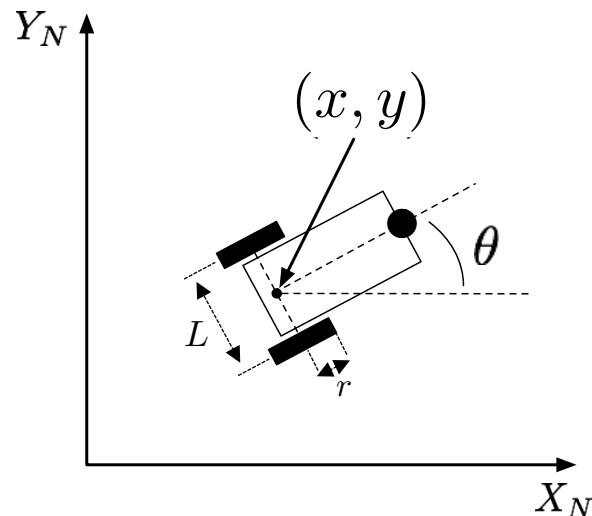
```
1 import matplotlib.pyplot as plt
2
3 fig, ax = plt.subplots(1, 1, figsize=(6, 4), dpi=150)
4 ax.plot(t, x['true'], 'k',
5         label=r'true: $x(t) = x_0 e^{((t-t_0) - \sin(t-t_0)\cos(t+t_0))/2}$')
6 for m in methods[1:]:
7     ax.plot(t, x[m], '--', label=m)
8 ax.legend()
9 ax.set_xlabel(r'$t$')
10 ax.set_ylabel(r'$x(t)$')
11 ax.set_title(r'$\dot{x} = \sin(t)^2 x$, '
12             + r'$x(' + f'\{t0:g\}' + r') = ' + f'\{x0:g\}, '$
13             + r'$\Delta t = '$ + f'\{dt:g\}$')
14 plt.show()
```



Example: Unicycle robot simulation

- Set the control input to guide the robot towards a target point (x_d, y_d) , and simulate the “closed-loop” system

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos \theta \\ \sin \theta \\ 0 \end{pmatrix} v + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \omega$$

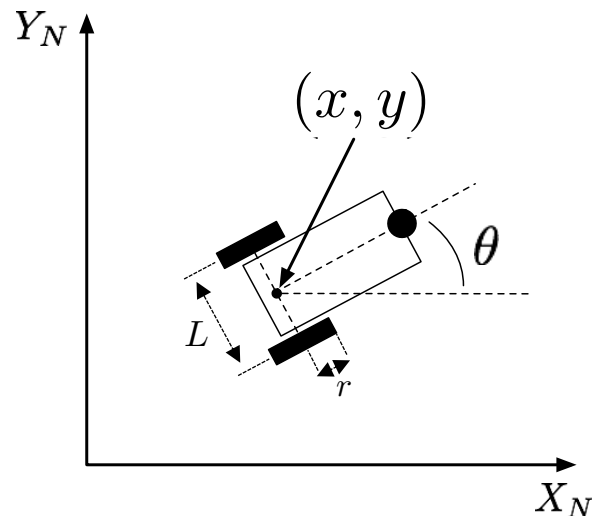


```
1 import numpy as np
2
3 def f(q, t, xd, yd):
4     """Evaluate the state derivative of a unicycle in closed-loop."""
5     x, y, theta = q
6
7     # Set steering velocity proportional to how far the robot has to turn
8     # to face the goal point
9     kw = 1.
10    theta_d = np.arctan2(yd - y, xd - x)
11    omega = -kw * (theta - theta_d)
12
13    # Set forward velocity proportional to the distance from the goal point
14    kv = 0.4
15    v = kv * np.sqrt((x - xd)**2 + (y - yd)**2)
16
17    dq = np.array([v*np.cos(theta), v*np.sin(theta), omega])
18    return dq
```

Example: Unicycle robot simulation

- ODE integration is done by `odeint` from `scipy.integrate`, which uses the RK45 scheme (i.e., Runge-Kutta with an adaptive step size)

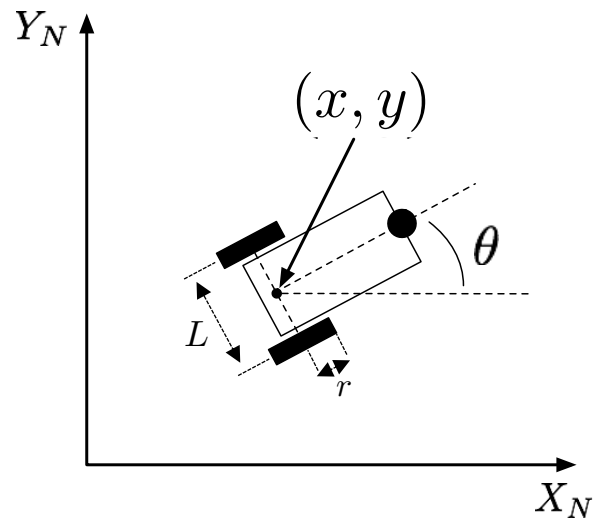
$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos \theta \\ \sin \theta \\ 0 \end{pmatrix} v + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \omega$$



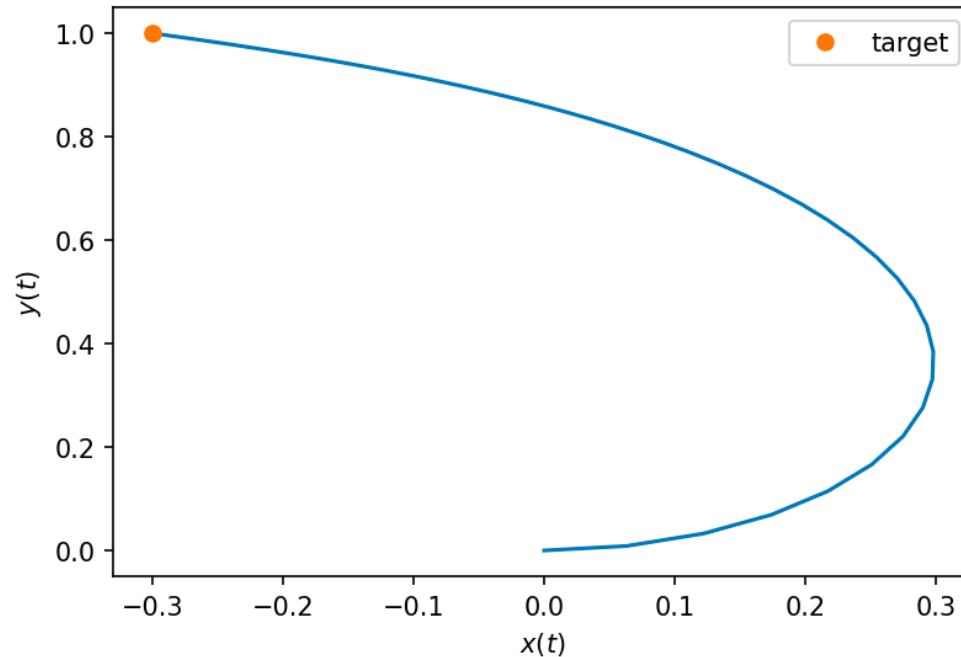
```
1 from scipy.integrate import odeint
2
3 q0 = np.array([0., 0., 0.])
4 xd, yd = -0.3, 0.5
5 T = 15.
6
7 t = np.linspace(0, T, num=100)
8 q = odeint(f, q0, t, args=(xd, yd))
9 x, y, theta = q.T
```

Example: Unicycle robot simulation

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos \theta \\ \sin \theta \\ 0 \end{pmatrix} v + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \omega$$



```
1 import matplotlib.pyplot as plt
2
3 fig, ax = plt.subplots(1, 1, figsize=(6, 4), dpi=150)
4 ax.plot(x, y)
5 ax.plot(xd, yd, 'o', label='target')
6 ax.legend()
7 ax.set_xlabel(r'$x(t)$')
8 ax.set_ylabel(r'$y(t)$')
9 plt.show()
```



Auto-differentiation in Python via JAX

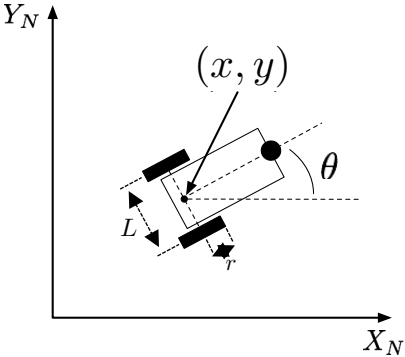
- Previously, we discussed linearizing nonlinear systems so we could apply tools from linear system analysis and control
- This requires derivatives, specifically Jacobians. *Auto-differentiation (AD, autodiff)* libraries (e.g., JAX) can automatically compute derivative *functions*
- E.g., for $f(x) = \frac{1}{2}\|x\|_2^2 = \frac{1}{2}\sum_i x_i^2$, we can use `jax.grad` to return the function $\nabla f(x) = x$

```
1 import jax
2 import jax.numpy as jnp
3
4 def f(x):
5     return jnp.sum(x**2)/2 # identical to NumPy syntax!
6
7 grad_f = jax.grad(f) # compute the gradient function
8 x = jnp.array([0., 1., 2.]) # use JAX arrays!
9
10 print('x:      ', x)
11 print('f(x):    ', f(x))
12 print('grad_f(x):', grad_f(x))
13 # x:          [0. 1. 2.]
14 # f(x):       2.5
15 # grad_f(x): [0. 1. 2.]
```

Example: Jacobians of unicycle dynamics

- You can derive the Jacobians of

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos \theta \\ \sin \theta \\ 0 \end{pmatrix} v + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \omega$$



around (\bar{q}, \bar{u}) to show

$$\frac{\partial f}{\partial q} = \begin{bmatrix} 0 & 0 & -\bar{v} \sin \bar{\theta} \\ 0 & 0 & \bar{v} \cos \bar{\theta} \\ 0 & 0 & 0 \end{bmatrix}, \quad \frac{\partial f}{\partial u} = \begin{bmatrix} \cos \bar{\theta} & 0 \\ \sin \bar{\theta} & 0 \\ 0 & 1 \end{bmatrix}$$

or use JAX to compute them automatically (useful for more complicated systems later)

```

1 import jax
2 import jax.numpy as jnp
3
4 def f(q, u):
5     """Evaluate the unicycle dynamics."""
6     x, y, theta = q
7     v, omega = u
8     dq = jnp.array([v*jnp.cos(theta), v*jnp.sin(theta), omega])
9     return dq
10
11 # Linearize around zero heading, zero steering rate,
12 # and a non-zero forward velocity
13 df = jax.jacobian(f, argnums=(0, 1)) # get Jacobian function `df`
14 q = jnp.array([0., 0., 0.])
15 u = jnp.array([1., 0.])
16 A, B = df(q, u)
17
18 print(A)
19 # [[0. 0. 0.]
20 # [0. 0. 1.]
21 # [0. 0. 0.]]
22
23 print(B)
24 # [[1. 0.]
25 # [0. 0.]
26 # [0. 1.]]

```

Next time

$$\begin{aligned} \min_u \quad & \int_0^T g(x(t), u(t), t) dt \\ \text{s.t.} \quad & \dot{x}(t) = f(x(t), u(t), t), \\ & x(t) \in \mathcal{X}, \\ & u(t) \in \mathcal{U}. \end{aligned}$$