# Principles of Robot Autonomy I

State space dynamics – computation and simulation

Stanford University

Autonomous Systems Lab

# Attendance Form

# Agenda

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

- Readings
  - Chapter 1, sections 1.4 – 1.5 in D. Gammelli, J. Lorenzetti, K. Luo, G. Zardini, M. Pavone. *Principles of Robot Autonomy*. 2026.

# 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 $t_0 < t_1 < \cdots < t_N$ 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):

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

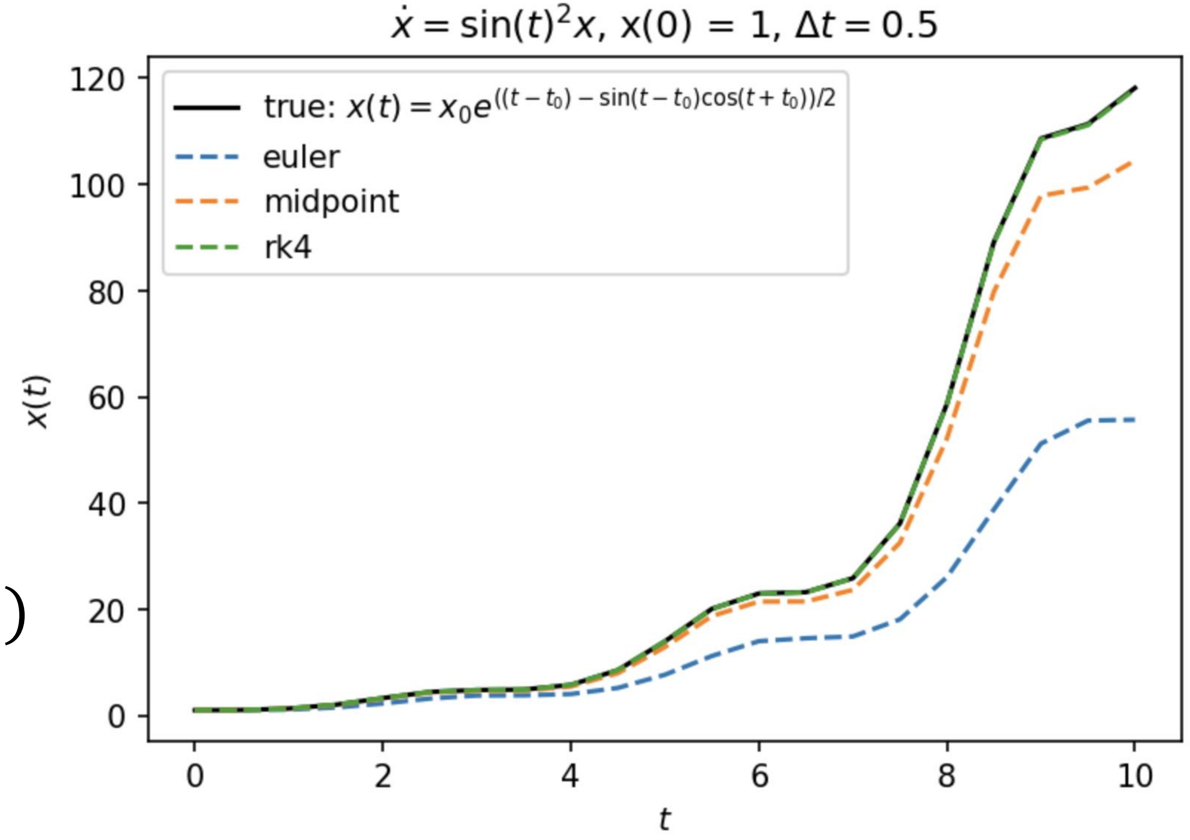
$$k_1 = f(x(t), t) \qquad k_2 = f\left(x(t) + \tfrac{\Delta t}{2} k_1, t + \tfrac{\Delta t}{2}\right)$$

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

# Truncation error

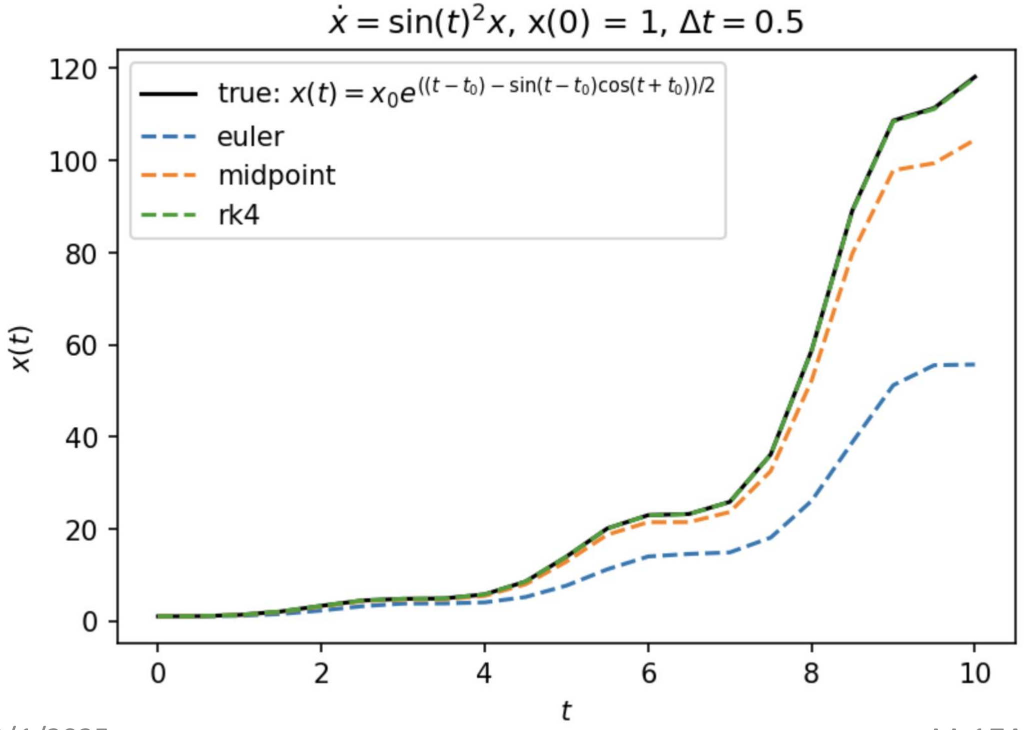
- Higher-order integration schemes use more function evaluations to reduce "local" (i.e., one-step) and "global" (i.e., accumulated) truncation error.

- 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”

```python
import numpy as np

# Define an IVP, timestamps, and discretization step
f = lambda x, t: x * np.sin(t)**2
x0 = 1.
t0, tf = 0., 10.
dt = 0.5
t = np.arange(t0, tf + dt, dt)

```

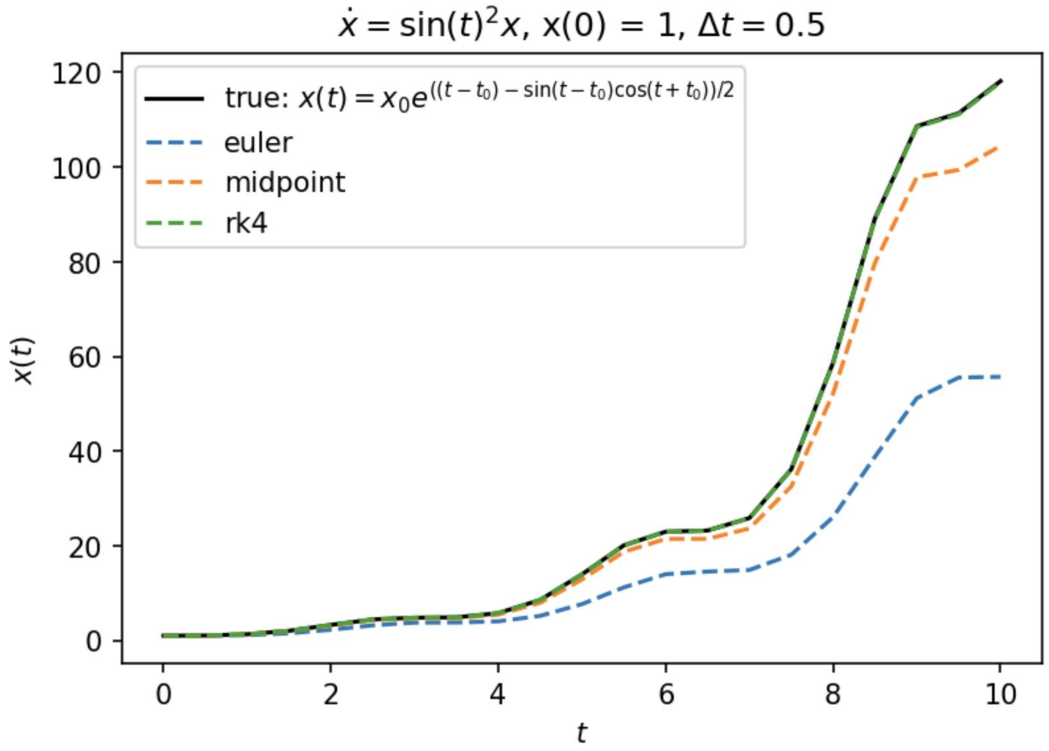



```python
# Compute true solution for comparison
x = {}
x['true'] = x0*np.exp(((t-t0) - np.sin(t-t0)*np.cos(t+t0))/2)

# Loop over timestamps and execute each integration scheme
methods = ('true', 'euler', 'midpoint', 'rk4')
for m in methods[1:]:
    x[m] = np.zeros_like(t)
    x[m][0] = x0

    for i in range(t.size - 1):
        if m == 'euler':
            x[m][i + 1] = x[m][i] + dt*f(x[m][i], t[i])

        elif m == 'midpoint':
            t_mid = t[i] + dt/2
            x_mid = x[m][i] + (dt/2)*f(x[m][i], t[i])
            x[m][i + 1] = x[m][i] + dt*f(x_mid, t_mid)

        elif m == 'rk4':
            k1 = f(x[m][i], t[i])
            k2 = f(x[m][i] + (dt/2)*k1, t[i] + dt/2)
            k3 = f(x[m][i] + (dt/2)*k2, t[i] + dt/2)
            k4 = f(x[m][i] + dt*k3, t[i] + dt)
            x[m][i + 1] = x[m][i] + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)

        else:
            raise NotImplementedError()
```

# Example: ODE integration “by hand”

- Free, open-source plotting functionality is provided by Matplotlib
- Check out the documentation! [https://matplotlib.org](https://matplotlib.org)

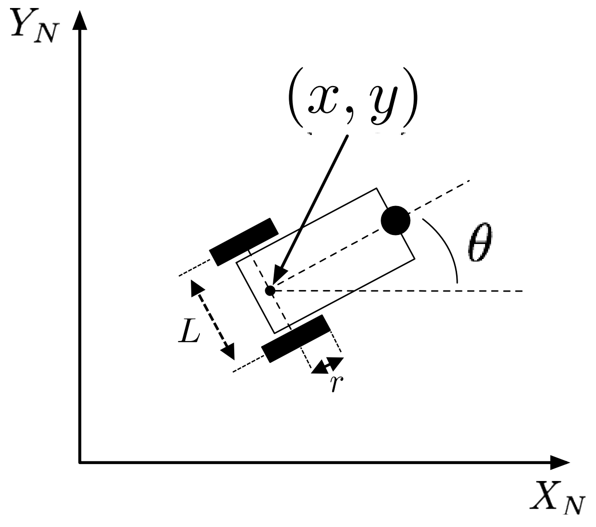
```python
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 1, figsize=(6, 4), dpi=150)
ax.plot(t, x['true'], 'k',
        label=r'true: $x(t) = x_0e^{((t-t_0) - \sin(t-t_0)\cos(t+t_0))/2}$')
for m in methods[1:]:
    ax.plot(t, x[m], '--', label=m)
ax.legend()
ax.set_xlabel(r'$t$')
ax.set_ylabel(r'$x(t)$')
ax.set_title(r'$\dot{x} = \sin(t)^2x$, '
             + r'x(' + f'{t0:g}' + r') = ' + f'{x0:g}, '
             + r'$\Delta t = $' + f'{dt:g}')
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

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

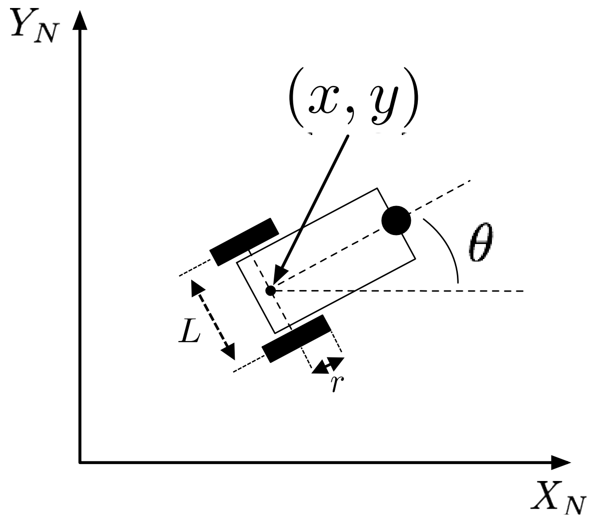



```python
import numpy as np

def f(q, t, xd, yd):
    """Evaluate the state derivative of a unicycle in closed-loop."""
    x, y, θ = q

    # Set steering velocity proportional to how far the robot has to turn
    # to face the goal point
    kω = 1.
    θd = np.arctan2(yd - y, xd - x)
    ω = -kω * (θ - θd)

    # Set forward velocity proportional to the distance from the goal point
    kv = 0.4
    v = kv * np.sqrt((x - xd)**2 + (y - yd)**2)

    dq = np.array([v*np.cos(θ), v*np.sin(θ), ω])
    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)

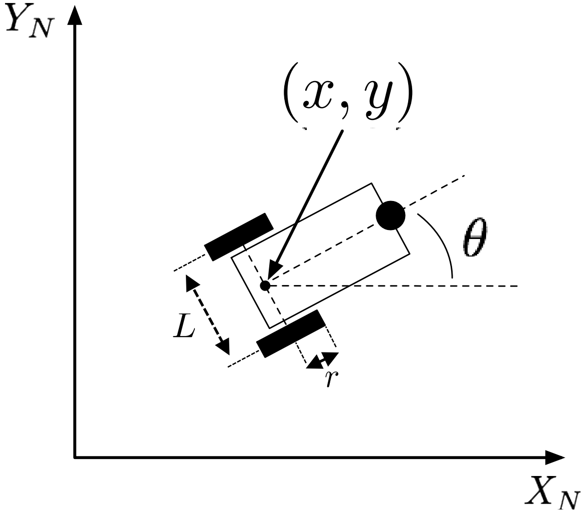
$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos\theta \\ \sin\theta \\ 0 \end{pmatrix} v + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \omega$$



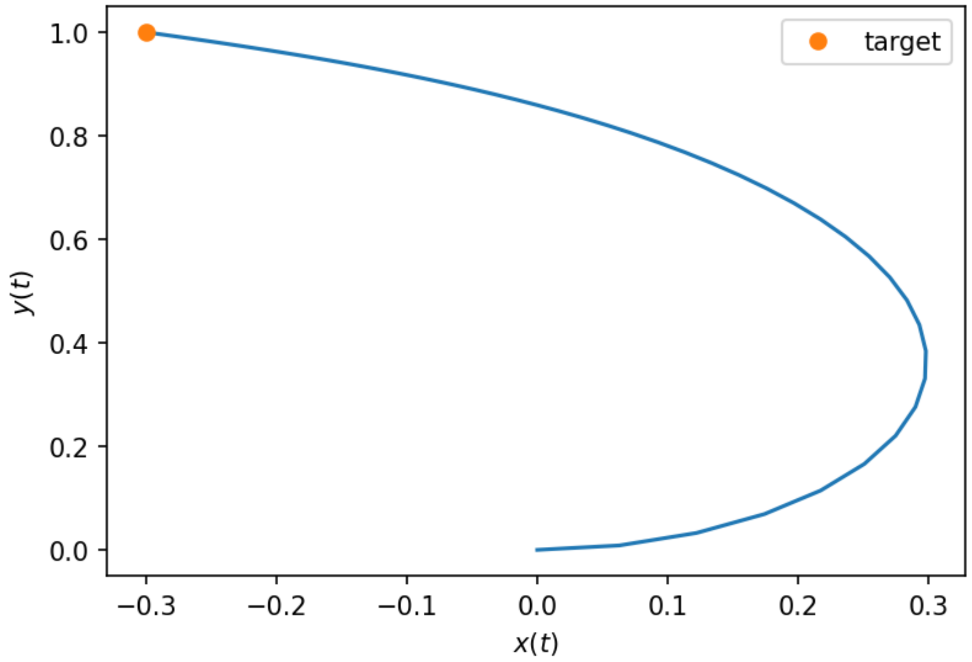

```python
from scipy.integrate import odeint

q0 = np.array([0., 0., 0.])
xd, yd = -0.3, 0.5
T = 15.

t = np.linspace(0, T, num=100)
q = odeint(f, q0, t, args=(xd, yd))
x, y, θ = q.T
```

# Example: Unicycle robot simulation

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




```python
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 1, figsize=(6, 4), dpi=150)
ax.plot(x, y)
ax.plot(xd, yd, 'o', label='target')
ax.legend()
ax.set_xlabel(r'$x(t)$')
ax.set_ylabel(r'$y(t)$')
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*

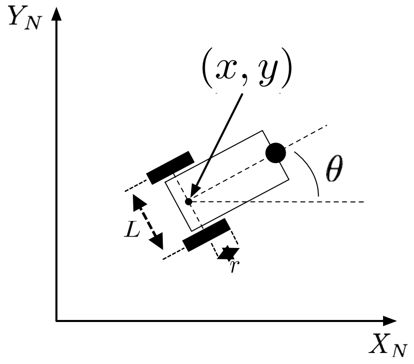
- E.g., for $f(x) = \frac{1}{2}\|x\|_2^2 = \frac{1}{2}\sum_i x_i^2$, we can use jax.grad to return the function $\nabla f(x) = x$

```python
import jax
import jax.numpy as jnp

def f(x):
    return jnp.sum(x**2)/2   # identical to NumPy syntax!

grad_f = jax.grad(f)          # compute the gradient function
x = jnp.array([0., 1., 2.])  # use JAX arrays!

print('x:         ', x)
print('f(x):      ', f(x))
print('grad_f(x):', grad_f(x))
# x:         [0. 1. 2.]
# f(x):      2.5
# 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)

```python
import jax
import jax.numpy as jnp

def f(q, u):
    """Evaluate the unicycle dynamics."""
    x, y, θ = q
    v, ω = u
    dq = jnp.array([v*jnp.cos(θ), v*jnp.sin(θ), ω])
    return dq

# Linearize around zero heading, zero steering rate,
# and a non-zero forward velocity
df = jax.jacobian(f, argnums=(0, 1))  # get Jacobian function `df`
q = jnp.array([0., 0., 0.])
u = jnp.array([1., 0.])
A, B = df(q, u)

print(A)
# [[0. 0. 0.]
#  [0. 0. 1.]
#  [0. 0. 0.]]

print(B)
# [[1. 0.]
#  [0. 0.]
#  [0. 1.]]
```

# Next time

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

# Attendance Form