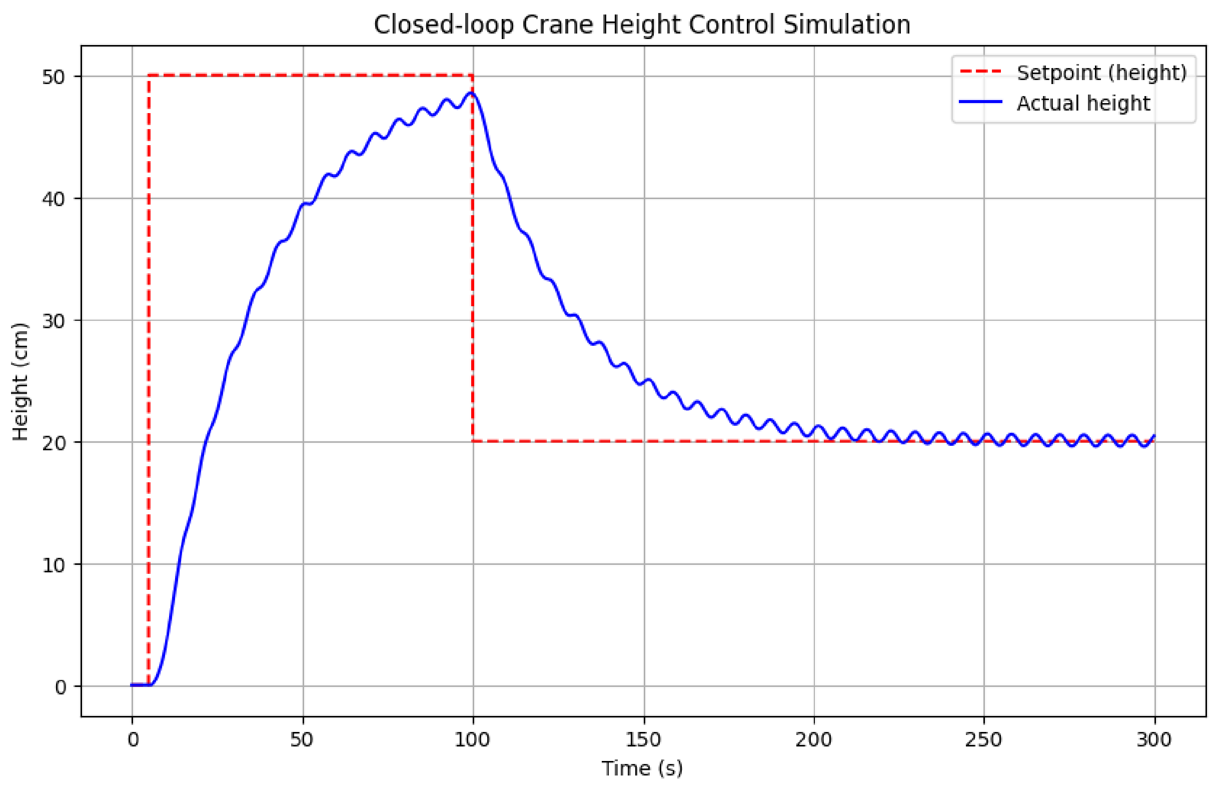
Control Simulation with PI Control, from FOPDT modeling

$$
\boxed{
\begin{aligned}
G(s) &= \frac{K\,e^{-Ls}}{\tau s + 1} \\
C(s) &= K_c\left(1 + \frac{1}{T_i s}\right) \\
K_c &= \frac{\tau}{K(\lambda + L)} \\
T_i &= \min(\tau,\ 4(\lambda + L)) \\
u[k] &= u[k-1] + K_c\left(e[k] - e[k-1] + \frac{T_s}{T_i} e[k]\right)
\end{aligned}
}
$$

In [17]:
```
import numpy as np
import matplotlib.pyplot as plt
from collections import deque

# --- PI Controller ---
def pi_controller(e_k, e_prev, u_prev, Kc, Ti, Ts):
    """
    Discrete PI controller.
    """
    u_k = u_prev + Kc * ( (e_k - e_prev) + (Ts / Ti) * e_k )
    # Clamp PWM between 0 and 1

    # u_k = max(0.0, min(1.0, u_k))# Updated (up and down motion allowed)
    u_k = max(-1.0, min(1.0, u_k))
    return u_k

# --- Actuator (motor velocity) ---
def actuator_response(u_k, v_prev, delay_buffer, tau, Ts, v_min=1.0, v_max=10.0):
    """
    FOPDT motor velocity model.
    """
    v_cmd = (v_max - v_min) * u_k
    delay_buffer.append(v_cmd)
    v_delayed = delay_buffer.popleft()
    v_k = v_prev + (Ts / tau) * (v_delayed - v_prev)
    return v_k, delay_buffer

# --- Height update ---
def update_height(h_prev, v_k, Ts):
    return h_prev + v_k * Ts

# --- Simulation parameters ---
Ts = 0.1        # sampling time (s)
```

```
Tsim = 300      # total simulation time (s)
steps = int(Tsim / Ts)

tau = 2.0      # motor time constant (s)
L = 0.5        # dead time (s)
delay_steps = int(L / Ts)
delay_buffer = deque([0.0]*delay_steps)

# Kc = 0.8       # PI proportional gain (example tuned)
# Ti = 3.0       # PI integral time constant (s)

Kp = 0.9704
tau = 28
thetap = 4
Kc = tau / Kp / (2*thetap)
Ti = tau

# --- Initialize variables ---
h = np.zeros(steps)
v = np.zeros(steps)
u = np.zeros(steps)
e = np.zeros(steps)

# Example setpoint: step from 0 cm to 50 cm at t=5s
h_sp = np.zeros(steps)
h_sp[int(5/Ts):] = 50.0
h_sp[int(100/Ts):] = 20.0

# --- Simulation loop ---
for k in range(1, steps):
    e[k] = h_sp[k] - h[k-1]
    u[k] = pi_controller(e[k], e[k-1], u[k-1], Kc, Ti, Ts)
    v[k], delay_buffer = actuator_response(u[k], v[k-1], delay_buffer, tau, Ts)
    h[k] = update_height(h[k-1], v[k], Ts)

# --- Plot results ---
plt.figure(figsize=(10,6))
plt.plot(np.arange(steps)*Ts, h_sp, 'r--', label='Setpoint (height)')
plt.plot(np.arange(steps)*Ts, h, 'b-', label='Actual height')
plt.xlabel('Time (s)')
plt.ylabel('Height (cm)')
plt.legend()
plt.grid(True)
plt.title('Closed-loop Crane Height Control Simulation')
plt.show()
```



About disturbance rejection:

Our actual control system is not a dynamic response to variable changes.
It is in all honesty a robot. We don't have disturbance rejection within our control loop.
However, we will use disturbance rejection-like control to closely monitor our sensor input, and manage noise there.
We are using three ultrasonic rangefinders, each of which likes to collect sound bouncing off the wrong surfaces and missing our intended objects.
Our disturbance rejection work will be adjusting the sensitivity of the controller to carefully treat those input errors.

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

Description:

This first cell plots the data we gathered from our entire doublet test.

Note: This control system is an integrating system and our actuator is currently on/off. Thus the actuator command is not proportional to the error right now. (We may make crane