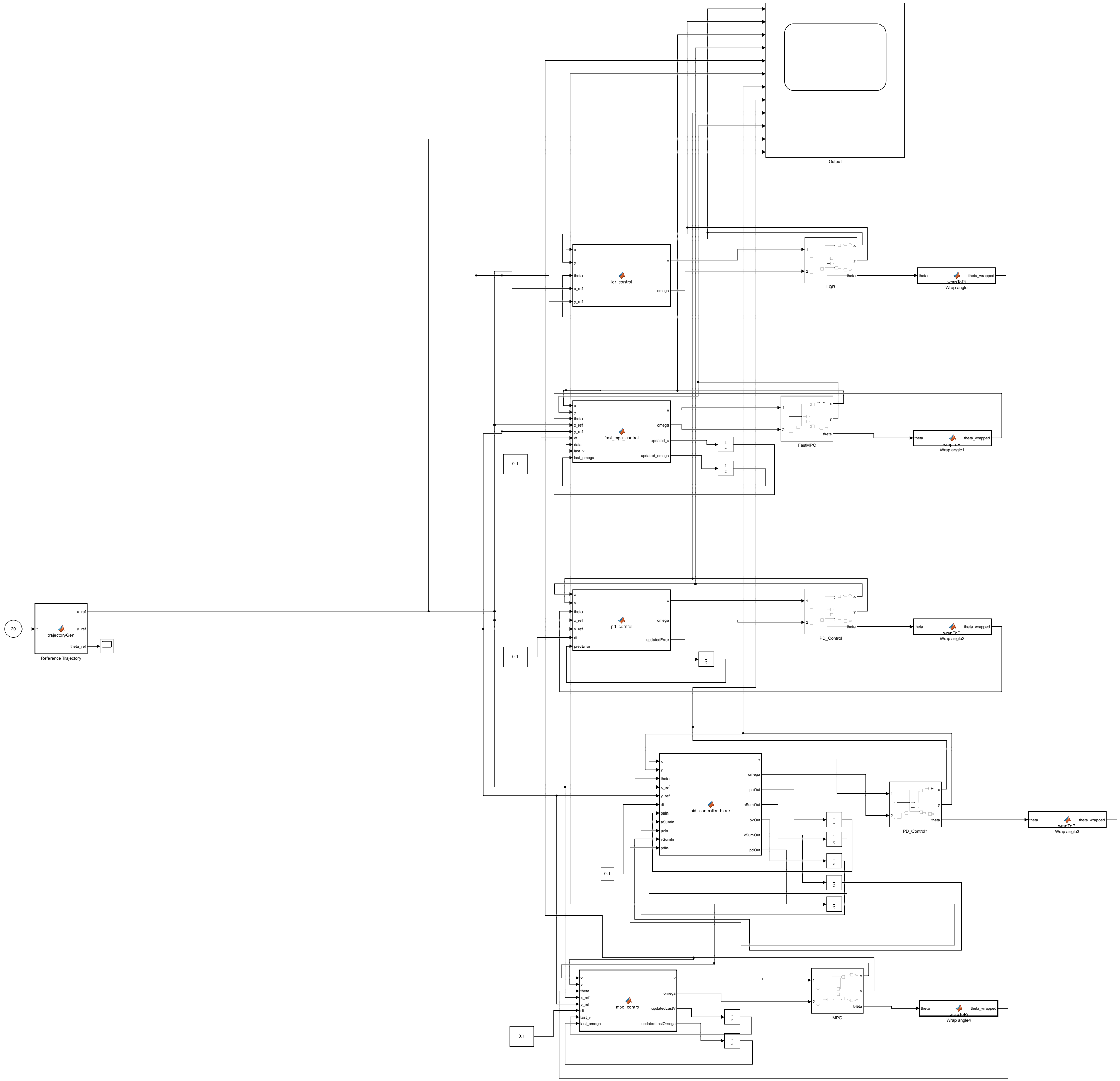
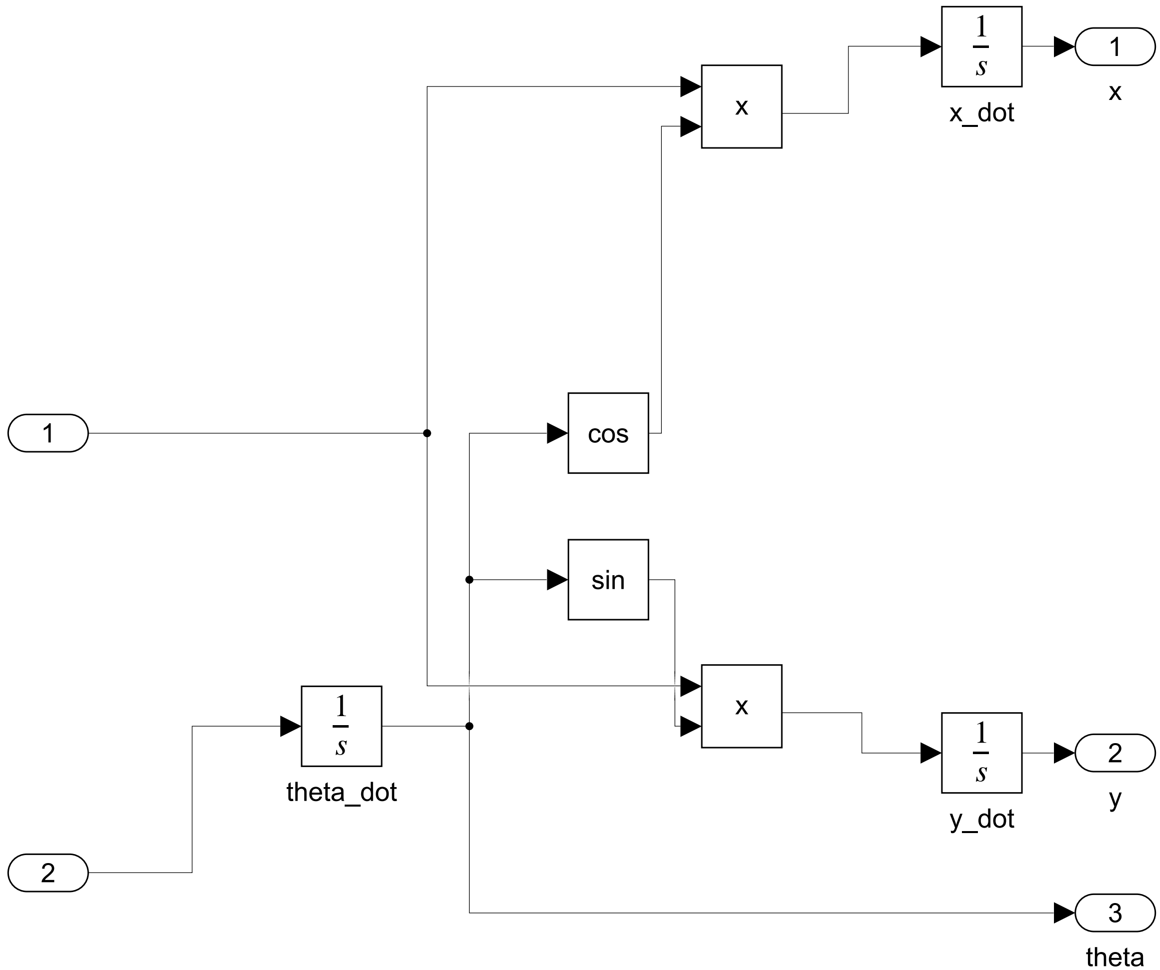
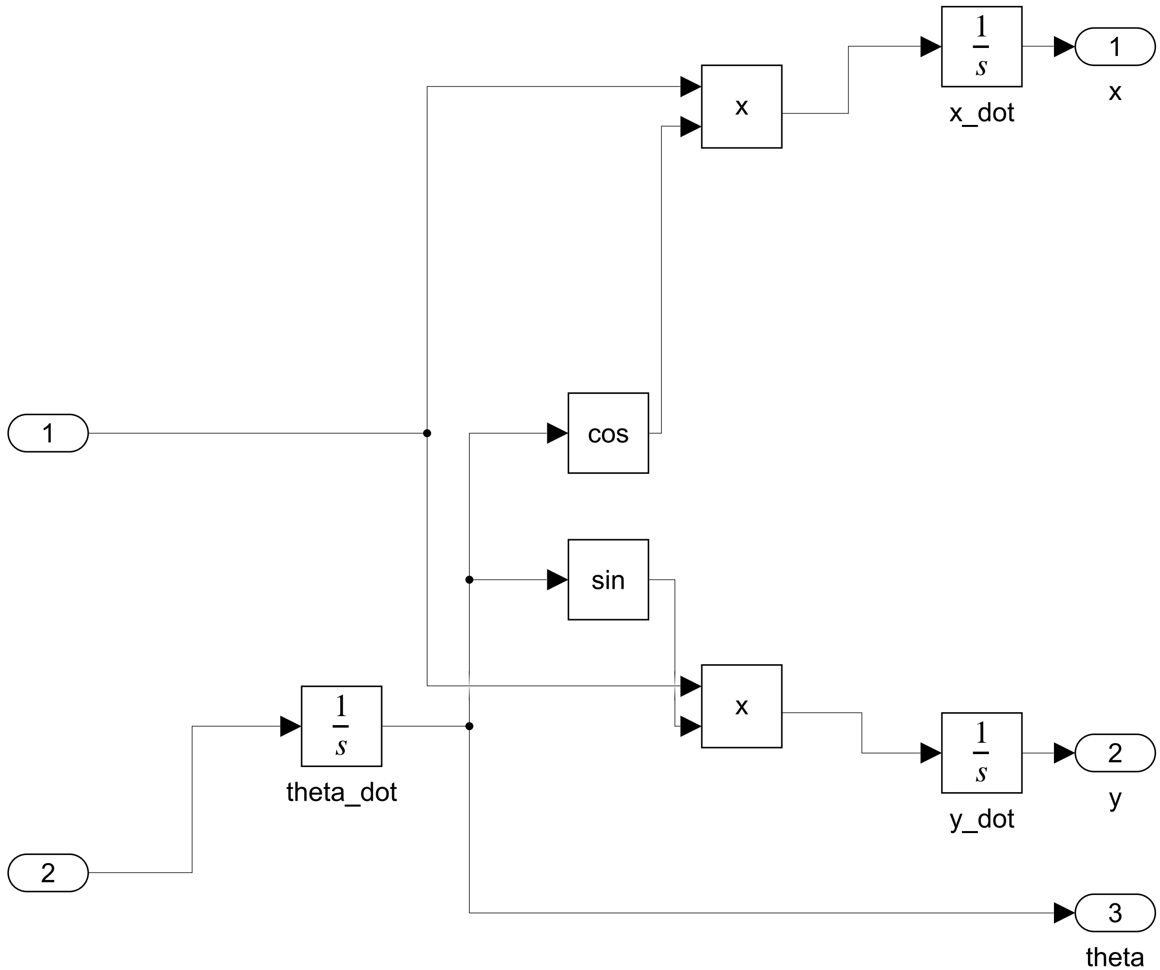


Output
lqr_control
LQR
Wrap angle
fast_mpc_control
FastMPC
Wrap angle1
pd_control
PD_Control
Wrap angle2
20
trajectoryGen
Reference Trajectory
pid_controller_block
PD_Control1
Wrap angle3
mpc_control
MPC
Wrap angle4
0.1



1
2
1/s
theta_dot
cos
sin
x
x
1/s
x_dot
1/s
y_dot
1
x
2
y
3
theta



1
2
theta_dot
cos
sin
x
x
x_dot
y_dot
1
x
2
y
3
theta

```
function [v, omega] = lqr_control(x, y, theta, x_ref, y_ref)
%#codegen
robotState = [x; y; theta];
targetPoint = [x_ref; y_ref];
[v, omega] = LQRController(robotState, targetPoint);

function [v, omega] = LQRController(robotState, targetPoint)
    % Inputs:
    %   robotState = [x; y; theta]
    %   targetPoint = [x_target; y_target]

    x = robotState(1);
    y = robotState(2);
    theta = robotState(3);

    x_target = targetPoint(1);
    y_target = targetPoint(2);

    % LQR Gain matrix (2x3)
    K = [1.5, 0.0, 0.0;
         0.0, 1.5, 2.0];

    % Global error
    dx = x_target - x;
    dy = y_target - y;
    error_global = [dx; dy];

    % Rotation matrix to robot frame
    c = cos(theta);
    s = sin(theta);
    R = [c, s; -s, c];
    error_robot = R * error_global;

    % Theta error
    angle_to_target = atan2(dy, dx);
    theta_error = atan2(sin(angle_to_target - theta), cos(angle_to_target - theta));

    % Form error state
    error_state = [error_robot; theta_error];

    % Control input via LQR gains
    v_r = K(1, :) * error_state;
    v_l = K(2, :) * error_state;

    % Convert to unicycle model control inputs
    v = (v_r + v_l) / 2;
    omega = (v_r - v_l) / 0.2;  % 0.2m wheel base

    % Saturation
    v = max(0.2, min(v, 5.0));
    omega = min(max(omega, -pi), pi);
end
end
```

```
function [v, omega, updated_v, updated_omega] = fast_mpc_control(x, y, theta, x_ref, y_ref, dt, data, last_v, last_omega)
%#codegen
robotState = [x; y; theta];
targetPoint = [x_ref; y_ref];
[v, omega, updated_v, updated_omega] = FastMPCController(robotState, targetPoint, dt, last_v, last_omega);

function [best_v, best_omega, last_v, last_omega] = FastMPCController(robotState, targetPoint, dt, last_v, last_omega)
    % Unpack current state
    x = robotState(1);
    y = robotState(2);
    theta = robotState(3);

    x_target = targetPoint(1);
    y_target = targetPoint(2);

    % MPC parameters
    N = 20;  % Prediction horizon

    % Cost weights (speed-prioritized)
    w_pos = 0.5;
    w_theta = 1.0;
    w_v = 0.05;
    w_omega = 0.1;
    velocity_bias = 0.3;  % Reward for high velocity

    % Control limits
    v_min = 0.0;
    v_max = 5.0;
    omega_min = -pi;
    omega_max = pi;

    % Control search ranges (favoring acceleration)
    v_range_min = max(v_min, last_v - 0.3);
    v_range_max = min(v_max, last_v + 0.7);
    v_options = linspace(v_range_min, v_range_max, 5);
    omega_options = linspace(max(omega_min, last_omega - 0.5), min(omega_max, last_omega + 0.5), 7);

    % Initialize best control
    best_cost = inf;
    best_v = last_v;
    best_omega = last_omega;

    for vi = 1:length(v_options)
        for wi = 1:length(omega_options)
            v = v_options(vi);
            omega = omega_options(wi);

            pred_x = x;
            pred_y = y;
            pred_theta = theta;
            total_cost = 0;

            for i = 0:N-1
                % Predict next state
                pred_x = pred_x + v * cos(pred_theta) * dt;
                pred_y = pred_y + v * sin(pred_theta) * dt;
                pred_theta = pred_theta + omega * dt;
```

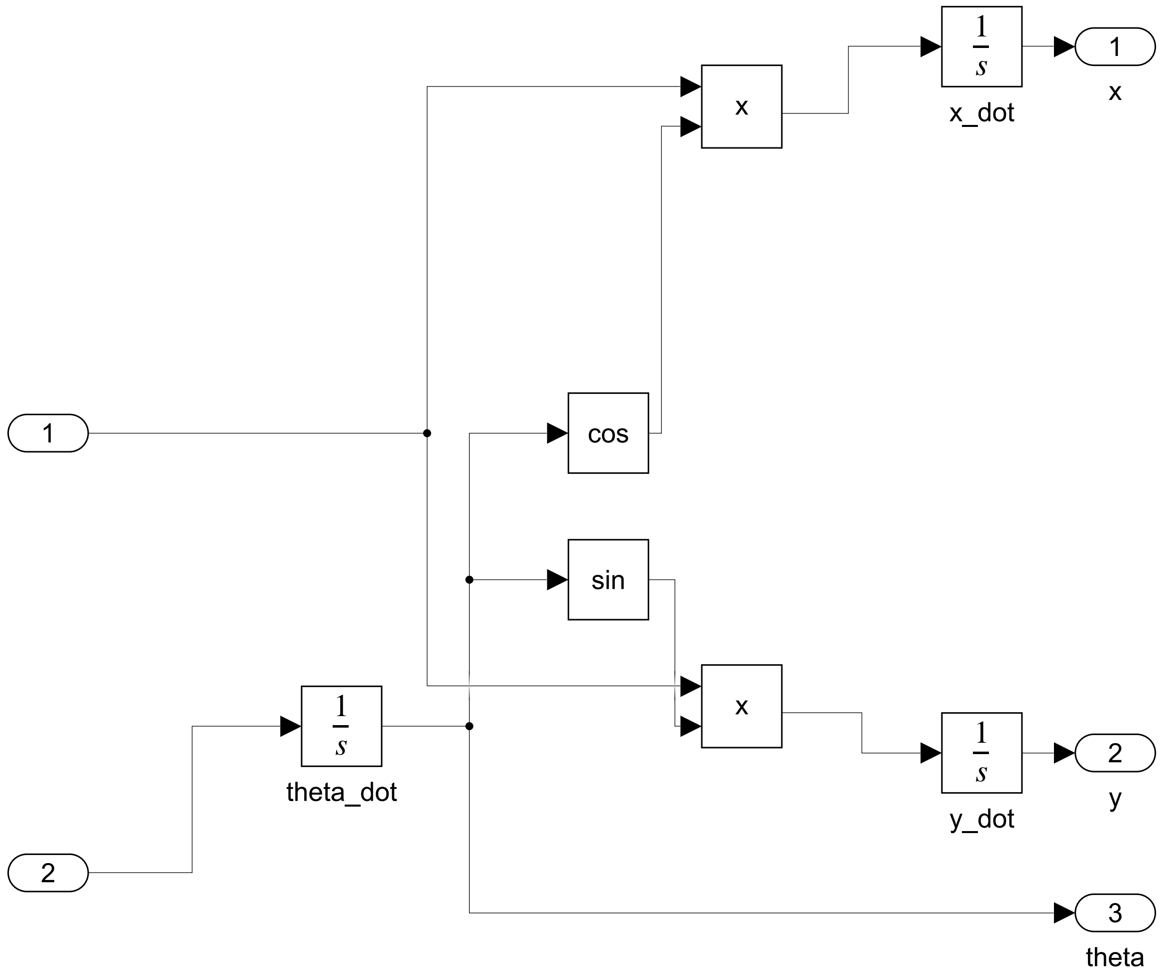
```
            % Errors
            pos_error = sqrt((x_target - pred_x)^2 + (y_target - pred_y)^2);
            desired_theta = atan2(y_target - pred_y, x_target - pred_x);
            theta_error = atan2(sin(desired_theta - pred_theta), cos(desired_theta - pred_theta));

            % Smoothness
            v_change = abs(v - last_v);
            omega_change = abs(omega - last_omega);

            % Reward for speed
            velocity_reward = -velocity_bias * v / v_max;

            % Step cost
            step_cost = w_pos * pos_error + w_theta * abs(theta_error) + w_v * v_change + w_omega * omega_change + velocity_re
            discount = 0.9 ^ i;
            total_cost = total_cost + discount * step_cost;
        end

        % Update best control
        if total_cost < best_cost
            best_cost = total_cost;
            best_v = v;
            best_omega = omega;
        end
    end
end

% Update memory for next call
last_v = best_v;
last_omega = best_omega;
end
end
```

```
function [v, omega, updatedError] = pd_control(x, y, theta, x_ref, y_ref, dt, prevError)
%#codegen
robotState = [x; y; theta];
targetPoint = [x_ref; y_ref];
[v, omega, updatedError] = PD_Controller(robotState, targetPoint, dt, prevError);

    function [v, omega, updatedPrevAngleError] = PD_Controller(robotState, targetPoint, dt, prevAngleError)
    x = robotState(1); y = robotState(2); theta = robotState(3);
    x_target = targetPoint(1); y_target = targetPoint(2);

    dx = x_target - x;
    dy = y_target - y;
    dist_to_target = sqrt(dx^2 + dy^2);

    angle_to_target = atan2(dy, dx);
    angle_error = atan2(sin(angle_to_target - theta), cos(angle_to_target - theta));
    angle_error_deriv = (angle_error - prevAngleError) / dt;
    updatedPrevAngleError = angle_error;

    % PD Gains
    kp_v = 1.5;
    kp_omega = 2.0;
    kd_omega = 1.0;

    v = min(max(kp_v * dist_to_target, 0.2), 5.0);
    omega = min(max(kp_omega * angle_error + kd_omega * angle_error_deriv, -pi), pi);
end

end
```

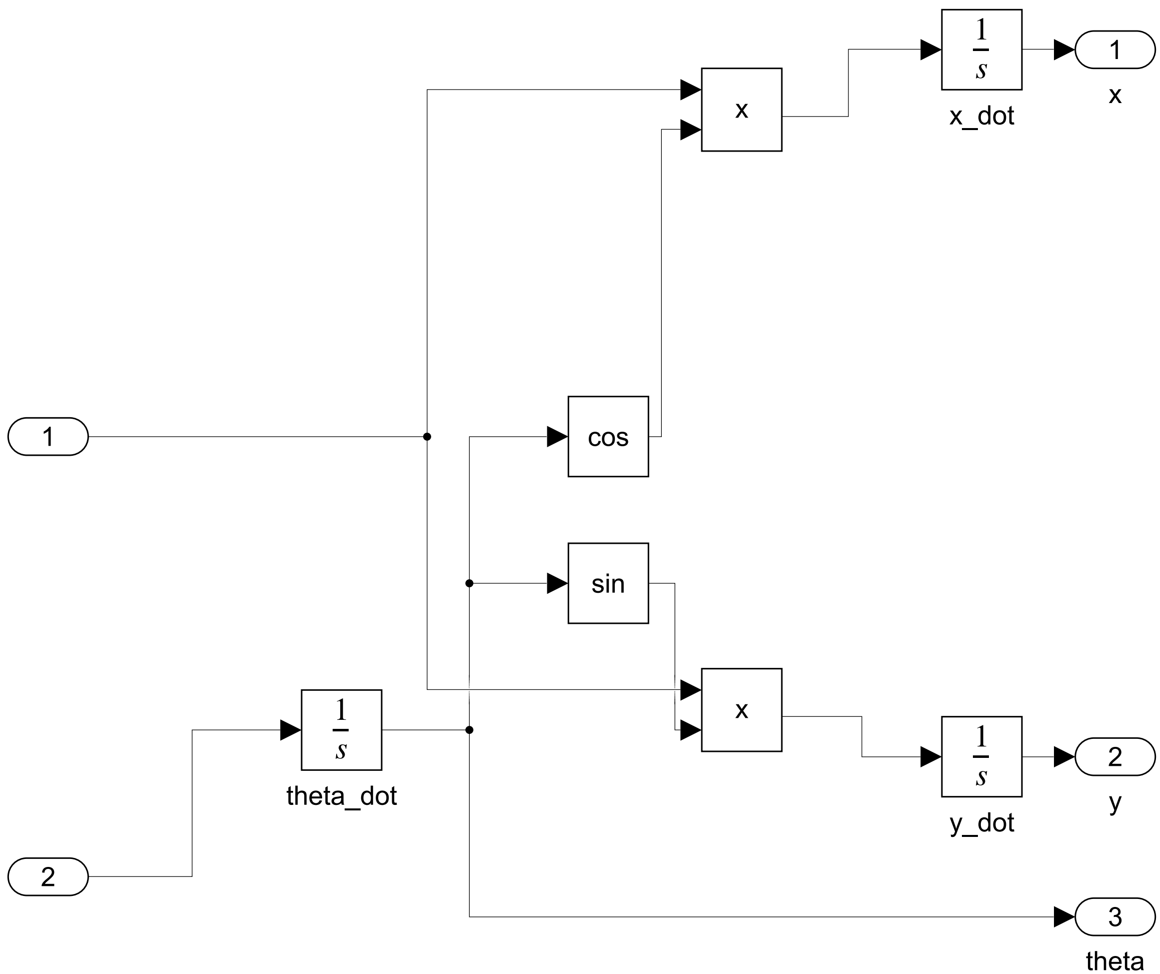
```
function [v, omega, updatedLastV, updatedLastOmega] = mpc_control(x, y, theta, x_ref, y_ref, dt, last_v, last_omega)
%#codegen
robotState = [x; y; theta];
targetPoint = [x_ref; y_ref];
[v, omega, updatedLastV, updatedLastOmega] = MPCController(robotState, targetPoint, dt, last_v, last_omega);

function [best_v, best_omega, last_v, last_omega] = MPCController(robotState, targetPoint, dt, last_v, last_omega)
    % Unpack state
    x = robotState(1);
    y = robotState(2);
    theta = robotState(3);

    x_target = targetPoint(1);
    y_target = targetPoint(2);

    % MPC Parameters
    N = 10;  % Prediction horizon

    % Weights
    w_pos = 1.0;
    w_theta = 1.5;
    w_v = 0.1;
    w_omega = 0.2;

    % Control limits
    v_min = 0.2; v_max = 5.0;
    omega_min = -pi; omega_max = pi;

    % Discretized control space
    v_options = linspace(max(v_min, last_v - 0.5), min(v_max, last_v + 0.5), 5);
    omega_options = linspace(max(omega_min, last_omega - 0.5), min(omega_max, last_omega + 0.5), 7);

    % Initialize best control
    best_cost = inf;
    best_v = last_v;
    best_omega = last_omega;

    % Evaluate control sequences
    for vi = 1:length(v_options)
        for wi = 1:length(omega_options)
            v = v_options(vi);
            omega = omega_options(wi);

            pred_x = x;
            pred_y = y;
            pred_theta = theta;
            total_cost = 0;

            for i = 0:N-1
                % Predict next state
                pred_x = pred_x + v * cos(pred_theta) * dt;
                pred_y = pred_y + v * sin(pred_theta) * dt;
                pred_theta = pred_theta + omega * dt;

                % Compute errors
                pos_error = sqrt((x_target - pred_x)^2 + (y_target - pred_y)^2);
                desired_theta = atan2(y_target - pred_y, x_target - pred_x);
                theta_error = atan2(sin(desired_theta - pred_theta), cos(desired_theta - pred_theta));
```

```
                % Control smoothness
                v_change = abs(v - last_v);
                omega_change = abs(omega - last_omega);

                % Cost with discount
                step_cost = w_pos * pos_error + w_theta * abs(theta_error) + w_v * v_change + w_omega * omega_change;
                discount = 0.9 ^ i;
                total_cost = total_cost + discount * step_cost;
            end

            % Update best control
            if total_cost < best_cost
                best_cost = total_cost;
                best_v = v;
                best_omega = omega;
            end
        end
    end

    % Update memory for next step
    last_v = best_v;
    last_omega = best_omega;
end

end
```

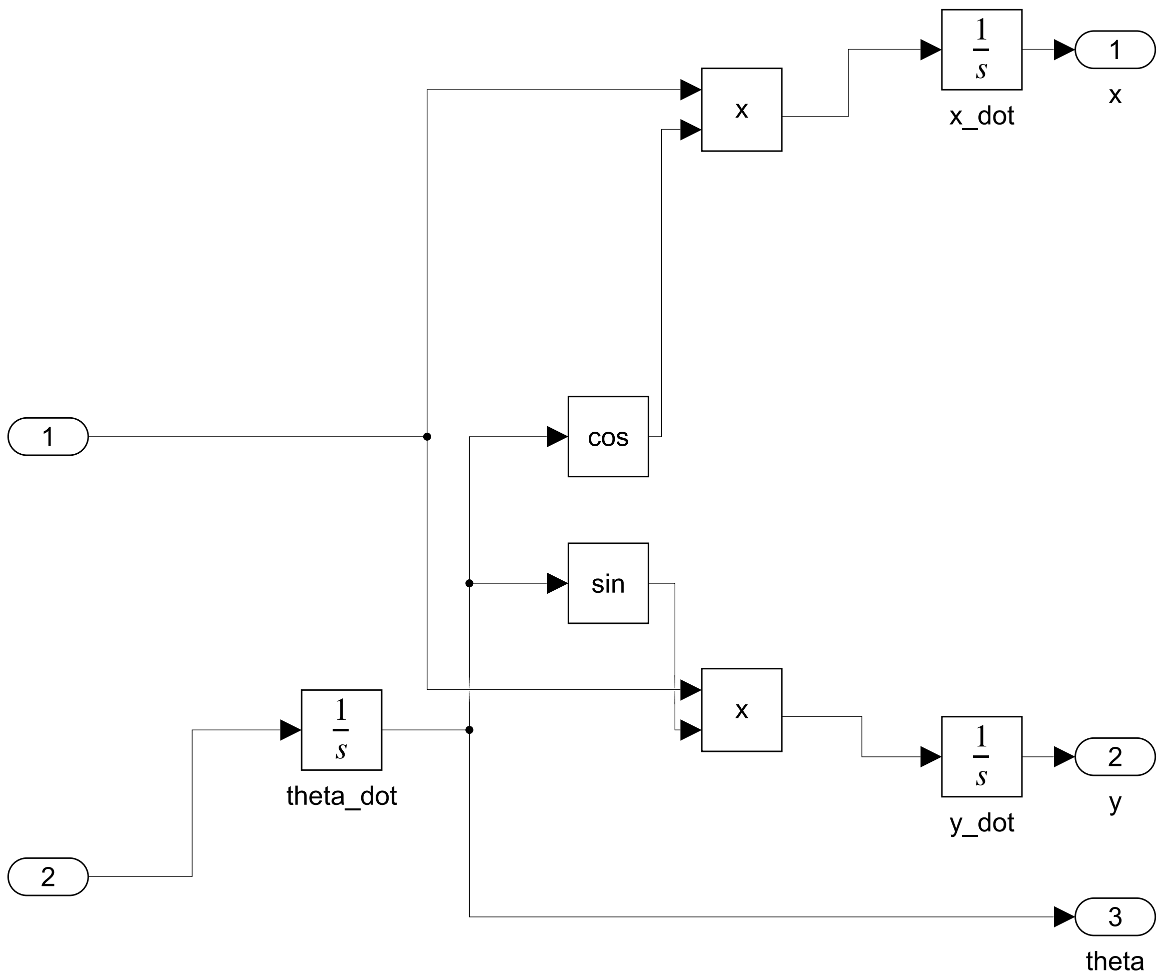
```
function [v, omega, paOut, aSumOut, pvOut, vSumOut, pdOut] = pid_controller_block(x, y, theta, x_ref, y_ref, dt, paIn, aSumIn, pvI
%#codegen
robotState = [x; y; theta];
targetPoint = [x_ref; y_ref];
[v, omega, paOut, aSumOut, pvOut, vSumOut, pdOut] = PIDController(robotState, targetPoint, dt, paIn, aSumIn, pvIn, vSumIn, pdIn);

    function [v, omega, prevAngleError, angleErrorSum, prevVError, vErrorSum, prevDistance] = PIDController(robotState, targetPoin
    % Unpack robot state
    x = robotState(1);
    y = robotState(2);
    theta = robotState(3);

    x_target = targetPoint(1);
    y_target = targetPoint(2);

    % Desired forward velocity
    v_desired = 5.0;

    %% --- Angular control (omega) ---
    angle_to_target = atan2(y_target - y, x_target - x);
    angle_error = atan2(sin(angle_to_target - theta), cos(angle_to_target - theta));
    angle_error_deriv = (angle_error - prevAngleError) / dt;
    angleErrorSum = angleErrorSum + angle_error * dt;
    angleErrorSum = min(max(angleErrorSum, -1.0), 1.0); % Anti-windup
    prevAngleError = angle_error;

    % PID gains for angular velocity
    kp_omega = 2.0;
    ki_omega = 1.0;
    kd_omega = 0.6;
    omega = kp_omega * angle_error + ki_omega * angleErrorSum + kd_omega * angle_error_deriv;
    omega = min(max(omega, -pi), pi);

    %% --- Linear velocity control ---
    dist_to_target = sqrt((x_target - x)^2 + (y_target - y)^2);
    v_current = (prevDistance - dist_to_target) / dt;
    prevDistance = dist_to_target;

    v_error = v_desired - v_current;
    v_error_deriv = (v_error - prevVError) / dt;
    vErrorSum = vErrorSum + v_error * dt;
    vErrorSum = min(max(vErrorSum, -2.0), 2.0); % Anti-windup
    prevVError = v_error;

    % PID gains for linear velocity
    kp_v = 1.0;
    ki_v = 0.5;
    kd_v = 0.2;
    v = kp_v * v_error + ki_v * vErrorSum + kd_v * v_error_deriv;

    % Reduce speed when not aligned with target
    v = v * cos(angle_error);

    % Clamp final velocity
    v = min(max(v, 0.2), 5.0);
end

end
```



1
2
1/s
theta_dot
cos
sin
x
x
1/s
x_dot
1/s
y_dot
1
x
2
y
3
theta



1
x
1/s
x_dot
1
x
cos
sin
x
1/s
y_dot
2
y
1
1/s
theta_dot
2
3
theta



1
2
theta_dot
cos
sin
x
x
x_dot
y_dot
x
y
theta
1
2
3

```
function [x_ref, y_ref, theta_ref] = trajectoryGen(t)
    radius = 5;
    omega_ref = 0.2;
    x_ref = radius * cos(omega_ref * t);
    y_ref = radius * sin(omega_ref * t);
    theta_ref = omega_ref * t + pi/2;
end
```

```
function theta_wrapped = wrapToPi(theta)
    theta_wrapped = mod(theta + pi, 2*pi) - pi;
end
```

```
function theta_wrapped = wrapToPi(theta)
    theta_wrapped = mod(theta + pi, 2*pi) - pi;
end
```

```
function theta_wrapped = wrapToPi(theta)
    theta_wrapped = mod(theta + pi, 2*pi) - pi;
end
```

```
function theta_wrapped = wrapToPi(theta)
    theta_wrapped = mod(theta + pi, 2*pi) - pi;
end
```

```
function theta_wrapped = wrapToPi(theta)
    theta_wrapped = mod(theta + pi, 2*pi) - pi;
end
```