Ball Predictor Node

```
/* ========================================================================
 * ball_predictor_node.cpp – stereo blob tracker + parabolic landing
 * ------------------------------------------------------------------------
 *  • subscribes   /camera/image_raw   (640×240 stereo image)
 *  • publishes    /ball_centroid      (geometry_msgs/Point, px coords)
 * ======================================================================== */
#include <rclcpp/rclcpp.hpp>
#include <sensor_msgs/msg/image.hpp>
#include <geometry_msgs/msg/point.hpp>
#include <cv_bridge/cv_bridge.h>
#include <opencv2/imgproc.hpp>
#include <deque>
#include <cmath>
#include <algorithm>

using std::placeholders::_1;

/* ── shared vision constants (duplicate of detector.cpp – factor out later) */
namespace vp {
constexpr int    IMG_W = 640, IMG_H = 240;
constexpr int    HALF_W = IMG_W / 2;
constexpr double IMG_CX = HALF_W / 2.0;
constexpr double IMG_CY = IMG_H  / 2.0;

constexpr int    THRESH    = 225;
constexpr int    ERODE_SZ  = 3, DILATE_SZ = 7;
constexpr double MIN_AREA  = 15.0 * 15.0;

constexpr double HFOV_DEG  = 45.0;
constexpr double CAMERA_H  = 85.0;          // camera height above table [cm]
constexpr double BASELINE  = 5.5;           // stereo baseline [cm]
constexpr double FOCAL_PX  = (HALF_W/2.0) /
        std::tan((HFOV_DEG*M_PI/180.0)/2.0);

/* Physics & filter */
constexpr double G_CM      = 981.0;         // gravity [cm·s⁻²]
constexpr double LPF_ALPHA = 0.8;           // pos low-pass
constexpr double VEL_ALPHA = 0.8;           // vel low-pass
constexpr double PRED_ALPHA= 0.8;           // landing-point LPF
constexpr size_t  BUF_MAX  = 4;
constexpr size_t  BUF_MIN  = 2;
constexpr double  RESET_GAP_S = 0.30;
} // namespace vp

/* ---- centroid helper (identical to detector.cpp) ------------------ */
static bool centroid(const cv::Mat& roi,
                     double& cx,double& cy,double& area,
                     const cv::Mat& erodeK,const cv::Mat& dilateK)
{
  cv::Mat g,m; cv::cvtColor(roi,g,cv::COLOR_BGR2GRAY);
  cv::threshold(g,m,vp::THRESH,255,cv::THRESH_BINARY);
  cv::erode(m,m,erodeK); cv::dilate(m,m,dilateK);
```

```
  std::vector<std::vector<cv::Point>> cont;
  cv::findContours(m,cont,cv::RETR_EXTERNAL,cv::CHAIN_APPROX_SIMPLE);
  if(cont.empty()) return false;
  auto it = std::max_element(cont.begin(),cont.end(),
        [](auto&a,auto&b){ return cv::contourArea(a)<cv::contourArea(b); });
  area = cv::contourArea(*it);
  if(area < vp::MIN_AREA) return false;
  cv::Moments mom = cv::moments(*it);
  cx = mom.m10 / mom.m00;
  cy = mom.m01 / mom.m00;
  return true;
}

/* ================================================================== */
class BallPredictorNode : public rclcpp::Node
{
public:
  BallPredictorNode():Node("ball_predictor_node")
  {
    img_sub_ = create_subscription<sensor_msgs::msg::Image>(
        "/camera/image_raw",10,std::bind(&BallPredictorNode::cb_img,this,_1));
    pub_=create_publisher<geometry_msgs::msg::Point>("/ball_centroid",10);

    erodeK_=cv::getStructuringElement(cv::MORPH_ELLIPSE,{vp::ERODE_SZ,vp::ERODE_SZ});

dilateK_=cv::getStructuringElement(cv::MORPH_ELLIPSE,{vp::DILATE_SZ,vp::DILATE_SZ});
    RCLCPP_INFO(get_logger(),"Predictor ready (g=%.1f cm/s²)",vp::G_CM);
  }

private:
/* ---- helper: compute height from area & disparity ----------------- */
  double height_cm(double area,double disparity) const
  {
    const double AREA_REF = 340.0, H_SCALE = 120.0;
    double h_area = std::max(0.0, H_SCALE*(std::sqrt(AREA_REF/area)-1.0));
    if(std::abs(disparity) < 1e-2) return h_area; // avoid div-by-zero
    double dist = (vp::BASELINE*vp::FOCAL_PX)/disparity;
    double h_st = vp::CAMERA_H - dist;
    return 0.6*h_st + 0.4*h_area;               // cheap fuse
  }

/* ---- main callback ------------------------------------------------ */
  void cb_img(const sensor_msgs::msg::Image::ConstSharedPtr& msg)
  {
    /* ··· basic checks ··· */
    auto cv_ptr = cv_bridge::toCvCopy(msg,sensor_msgs::image_encodings::BGR8);
    const cv::Mat& img = cv_ptr->image;
    if(img.empty()) return;
    if(img.cols!=vp::IMG_W || img.rows!=vp::IMG_H){
      RCLCPP_WARN_THROTTLE(get_logger(),*get_clock(),5000,
                           "Unexpected image size %dx%d",img.cols,img.rows);
      return;
    }

    /* ··· extract centroids from left/right eye ··· */
```

```
cv::Rect roiL(0,0,vp::HALF_W,vp::IMG_H), roiR(vp::HALF_W,0,vp::HALF_W,vp::IMG_H);
double lx,ly,aL=0; bool fL = centroid(img(roiL),lx,ly,aL,erodeK_,dilateK_);
double rx,ry,aR=0; bool fR = centroid(img(roiR),rx,ry,aR,erodeK_,dilateK_);

if(!(fL||fR)){   // nothing
  if((get_clock()->now()-last_seen_).seconds()>vp::RESET_GAP_S){
    buf_.clear(); has_prev_h_=has_prev_v_=has_pred_=false;
  }
  return;
}
last_seen_ = get_clock()->now();

/* choose centroid */
double px,py,area;
if(fL&&fR){ px=(lx+rx)/2; py=(ly+ry)/2; area=(aL+aR)/2; }
else if(fL){ px=lx; py=ly; area=aL; }
else        { px=rx; py=ry; area=aR; }

double disp = (fL&&fR)? lx-rx : 0.0;

/* height LPF */
double h_raw = height_cm(area,disp);
if(has_prev_h_) h_raw = vp::LPF_ALPHA*prev_h_ + (1.0-vp::LPF_ALPHA)*h_raw;
prev_h_=h_raw; has_prev_h_=true;

/* perspective correct */
double corr = (vp::CAMERA_H-h_raw)/vp::CAMERA_H;
px = vp::IMG_CX + (px-vp::IMG_CX)*corr;
py = vp::IMG_CY + (py-vp::IMG_CY)*corr;

/* buffer sample */
if(buf_.size()==vp::BUF_MAX) buf_.pop_front();
buf_.push_back({px,py,h_raw,get_clock()->now().seconds()});

if(buf_.size()<vp::BUF_MIN) return;

/* planar velocity */
const auto& q0=buf_[buf_.size()-2];
const auto& q1=buf_.back();
double dt = q1.t - q0.t;
if(dt<1e-3) return;      // avoid div0
double vx=(q1.x - q0.x)/dt, vy=(q1.y - q0.y)/dt;
if(has_prev_v_){
  vx = vp::VEL_ALPHA*prev_vx_ + (1.0-vp::VEL_ALPHA)*vx;
  vy = vp::VEL_ALPHA*prev_vy_ + (1.0-vp::VEL_ALPHA)*vy;
}
prev_vx_=vx; prev_vy_=vy; has_prev_v_=true;

/* time until ground (vertical parabola) */
double vz   = (q1.h - q0.h) / dt;
double h_cl = std::max(0.0, q1.h);              // ← 1. clamp height
double disc = vz*vz + 2*vp::G_CM*h_cl;          // ← 2. safe radicand
if (disc <= 0.0) return;                        // ← 3. give up this frame

double t_land = (-vz - std::sqrt(disc)) / -vp::G_CM;
```

```
    double px_land = px + vx*t_land;
    double py_land = py + vy*t_land;

    if(has_pred_){
      px_land = vp::PRED_ALPHA*prev_pred_x_ + (1.0-vp::PRED_ALPHA)*px_land;
      py_land = vp::PRED_ALPHA*prev_pred_y_ + (1.0-vp::PRED_ALPHA)*py_land;
    }
    prev_pred_x_=px_land; prev_pred_y_=py_land; has_pred_=true;

    px_land = std::clamp(px_land, 0.0, static_cast<double>(vp::HALF_W  - 1)); // 0 …
319
    py_land = std::clamp(py_land, 0.0, static_cast<double>(vp::IMG_H   - 1)); // 0 …
239


    /* publish */
    geometry_msgs::msg::Point out;
    out.x=px_land; out.y=py_land; out.z=h_raw;
    pub_->publish(out);
  }

/* ---- sample struct & state ------------------------------------- */
  struct Sample{ double x,y,h,t; };
  std::deque<Sample> buf_;

  rclcpp::Time last_seen_{0,0,RCL_ROS_TIME};
  double prev_h_{0}; bool has_prev_h_{false};
  double prev_vx_{0}, prev_vy_{0}; bool has_prev_v_{false};
  double prev_pred_x_{0}, prev_pred_y_{0}; bool has_pred_{false};

  rclcpp::Subscription<sensor_msgs::msg::Image>::SharedPtr img_sub_;
  rclcpp::Publisher<geometry_msgs::msg::Point>::SharedPtr pub_;
  cv::Mat erodeK_, dilateK_;
};

/* ---- main ------------------------------------------------------ */
int main(int argc,char** argv)
{
  rclcpp::init(argc,argv);
  rclcpp::spin(std::make_shared<BallPredictorNode>());
  rclcpp::shutdown();
  return 0;
}
```

Pixel to XY Bridge

```
#include <memory>
#include <functional>
#include "rclcpp/rclcpp.hpp"
#include "geometry_msgs/msg/point.hpp"
#include "controller_msgs/msg/set_xy.hpp"

using SetXY = controller_msgs::msg::SetXY;      // alias once

class PixelBridge : public rclcpp::Node
{
public:
  PixelBridge() : Node("pixel_to_xy_bridge")
  {
    using std::placeholders::_1;
    sub_ = create_subscription<geometry_msgs::msg::Point>(
              "/ball_centroid", 10, std::bind(&PixelBridge::cb, this, _1));

    pub_ = create_publisher<SetXY>("/set_xy", 10);   // ★ changed
  }

private:
  void cb(const geometry_msgs::msg::Point::SharedPtr p)
  {
    constexpr double SCALE = 0.22;
    constexpr double PIX_H = 240.0;
    constexpr double PIX_W = 320.0;

    double dx = (PIX_H / 2.0) - p->y;
    double dy = (PIX_W / 2.0) - p->x;

    SetXY msg;                                       // ★ changed
    msg.x = dx * SCALE;
    msg.y = dy * SCALE;
    pub_->publish(msg);
  }

  rclcpp::Subscription<geometry_msgs::msg::Point>::SharedPtr sub_;
  rclcpp::Publisher<SetXY>::SharedPtr                        pub_;  // ★ changed
};

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<PixelBridge>());
  rclcpp::shutdown();
  return 0;
}
```

Independent Joint Controller

```
/* ====================================================================
 *  Subscribes   :  /set_xy  (controller_msgs/SetXY, cm in camera plane)
 *  Computes     :  planar 2-DOF Independent PID Joint Control ➜ pwm signal
 *  Writes to    :  Dynamixel PWM Registers
 *  Publishes on :  /pwm/joint_plot_data  (controller_msgs/JointPlotData)
 * ==================================================================== */

#include <rclcpp/rclcpp.hpp>
#include <controller_msgs/msg/set_xy.hpp>
#include <controller_msgs/msg/joint_plot_data.hpp>
#include <dynamixel_sdk/dynamixel_sdk.h>
#include "control_common/control_params.hpp"
#include "control_common/control_utils.hpp"
#include <cmath>

using SetXY = controller_msgs::msg::SetXY;
using Plot = controller_msgs::msg::JointPlotData;
using std::placeholders::_1;

class PwmPositionNode : public rclcpp::Node
{
public:
  PwmPositionNode() : Node("pwm_position_node")
  {
    declare_parameter<std::string>("port", "/dev/ttyUSB0");
    declare_parameter<int>("baud", 1000000);
    declare_parameter<double>("kp", 50.0);
    declare_parameter<double>("ki", 0.1);
    declare_parameter<double>("kd", 5.0);
    declare_parameter<int>("max_pwm", 885);
    declare_parameter<int>("ctrl_rate_hz", 1000);

    kp_ = get_parameter("kp").as_double();
    ki_ = get_parameter("ki").as_double();
    kd_ = get_parameter("kd").as_double();
    max_pwm_ = get_parameter("max_pwm").as_int();
    dt_s_ = 1.0 / get_parameter("ctrl_rate_hz").as_int();

    port_ =
dynamixel::PortHandler::getPortHandler(get_parameter("port").as_string().c_str());
    packet_ = dynamixel::PacketHandler::getPacketHandler(2.0);
    if (!port_->openPort() || !port_->setBaudRate(get_parameter("baud").as_int()))
      RCLCPP_FATAL(get_logger(), "Serial open failed");

    for (uint8_t id : control::SERVO_IDS)
    {
      write8(id, control::ADDR_TORQUE_ENABLE, 0);
      write8(id, control::ADDR_OPERATING_MODE, control::MODE_PWM);
      write8(id, control::ADDR_TORQUE_ENABLE, 1);
    }

    sub_xy_ = create_subscription<SetXY>("set_xy", 10,
std::bind(&PwmPositionNode::cb_xy, this, _1));
    pub_plot_ = create_publisher<Plot>("pwm/joint_plot_data", 10);
```

```
    timer_ = create_wall_timer(std::chrono::duration<double>(dt_s_),
std::bind(&PwmPositionNode::control_loop, this));
    last_good_cmd_time_ = now();

    RCLCPP_INFO(get_logger(), "PWM node ready (kp=%.1f ki=%.1f kd=%.1f)", kp_, ki_,
kd_);
  }

  ~PwmPositionNode() override
  {
    for (uint8_t id : control::SERVO_IDS)
      write8(id, control::ADDR_TORQUE_ENABLE, 0);
    port_->closePort();
  }

private:
  void write8(uint8_t id, uint16_t addr, uint8_t d)
  {
    uint8_t e{};
    int rc = packet_->write1ByteTxRx(port_, id, addr, d, &e);
    if (rc != COMM_SUCCESS || e)
      RCLCPP_ERROR_THROTTLE(get_logger(), *get_clock(), 2000, "DXL %u err=%d rc=%d",
id, e, rc);
  }

  // Use GroupSyncWrite to set both motors' PWMs at once
  void send_pwms(int pwm1, int pwm2)
  {
    dynamixel::GroupSyncWrite group_writer(port_, packet_, control::ADDR_GOAL_PWM, 2);
    uint8_t b1[2] = {DXL_LOBYTE(pwm1), DXL_HIBYTE(pwm1)};
    uint8_t b2[2] = {DXL_LOBYTE(pwm2), DXL_HIBYTE(pwm2)};
    bool ok1 = group_writer.addParam(control::ID1, b1);
    bool ok2 = group_writer.addParam(control::ID2, b2);
    if (!ok1 || !ok2)
    {
      RCLCPP_ERROR_THROTTLE(get_logger(), *get_clock(), 2000,
                            "GroupSyncWrite addParam failed! (ok1=%d ok2=%d)", ok1,
ok2);
      return;
    }
    int rc = group_writer.txPacket();
    if (rc != COMM_SUCCESS)
    {
      RCLCPP_ERROR_THROTTLE(get_logger(), *get_clock(), 2000,
                            "GroupSyncWrite PWM failed: %s",
packet_->getTxRxResult(rc));
    }
  }

  void cb_xy(const SetXY::SharedPtr msg)
  {
    if (!std::isfinite(msg->x) || !std::isfinite(msg->y))
    {
      RCLCPP_WARN_THROTTLE(get_logger(), *get_clock(), 2000, "set_xy contains NaN/Inf
— ignoring");
```

```
      return;
    }
    double j1, j2;
    RCLCPP_INFO(get_logger(), "PWM HERE->>>>> (kp=%.1f ki=%.1f kd=%.1f)", kp_, ki_,
kd_);
    if (control::ik_xy(msg->x * 0.01, msg->y * 0.01, j1, j2))
    {
      tgt_j1_ = j1 * 180.0 / M_PI;
      tgt_j2_ = j2 * 180.0 / M_PI;
      last_good_cmd_time_ = now();
      has_target_ = true;
    }
    else
    {
      RCLCPP_WARN_THROTTLE(get_logger(), *get_clock(), 2000, "IK failed for (%.1f,
%.1f)", msg->x, msg->y);
    }
  }

  void control_loop()
  {
    if (!has_target_ || (now() - last_good_cmd_time_).seconds() > 3.0)
      return;
    int32_t tick1{}, tick2{};
    control::read_two_positions(port_, packet_, tick1, tick2);
    double cur1 = control::tick2deg(0, tick1);
    double cur2 = control::tick2deg(1, tick2);
    RCLCPP_INFO(get_logger(), "PWM gains (kp=%.1f ki=%.1f kd=%.1f)", kp_, ki_, kd_);
    double pwm1 = pid1_.step(tgt_j1_ - cur1, kp_, ki_, kd_, max_pwm_);
    double pwm2 = pid2_.step(tgt_j2_ - cur2, kp_, ki_, kd_, max_pwm_);
    send_pwms(int(pwm1), int(pwm2));

    Plot p;
    p.stamp = now();
    p.q1_deg = cur1;
    p.q2_deg = cur2;
    p.q1_des_deg = tgt_j1_;
    p.q2_des_deg = tgt_j2_;
    p.e1 = tgt_j1_ - cur1;
    p.e2 = tgt_j2_ - cur2;
    p.pwm1 = pwm1;
    p.pwm2 = pwm2;
    p.measured_q1 = cur1;
    p.measured_q2 = cur2;
    p.desired_q1 = tgt_j1_;
    p.desired_q2 = tgt_j2_;
    p.controller_name = "pwm_position_node";
    pub_plot_->publish(p);
  }

  rclcpp::Subscription<SetXY>::SharedPtr sub_xy_;
  rclcpp::Publisher<Plot>::SharedPtr pub_plot_;
  rclcpp::TimerBase::SharedPtr timer_;
  dynamixel::PortHandler *port_{};
  dynamixel::PacketHandler *packet_{};
```

```
  control::PidState pid1_, pid2_;
  double kp_, ki_, kd_, dt_s_;
  int max_pwm_;
  bool has_target_{false};
  double tgt_j1_{}, tgt_j2_{};
  rclcpp::Time last_good_cmd_time_;
};

/* ---- main ---------------------------------------------------------- */
int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<PwmPositionNode>());
  rclcpp::shutdown();
  return 0;
}
```

Joint Space Inverse Dynamics Controller

```
/* ============================================================================
 * joint_space_idc.cpp — latency-optimised   +   joint-limit clamp
 * ----------------------------------------------------------------------------
 *  Subscribes   :  /set_xy  (controller_msgs/SetXY, cm in camera plane)
 *                  ↳ target (x,y) is converted to desired joint angles via IK
 *  Computes     :  planar 2-DOF **Joint-Space** Inverse Dynamics Control
 *                  τ = B(q) q̈* + n(q, q̇)  with  q̈* = Kp (qd-q) + Kd (-q̇)
 *  Writes to    :  Dynamixel Goal-Position registers (position mode)
 *  Publishes on :  /idc_joint/joint_plot_data  (controller_msgs/JointPlotData)
 *
 *  Style & low-level helpers follow the operational-space IDC node
:contentReference[oaicite:0]{index=0}
 * ============================================================================ */
#include <chrono>
#include <cmath>
#include <algorithm>
#include <rclcpp/rclcpp.hpp>
#include <controller_msgs/msg/set_xy.hpp>
#include <controller_msgs/msg/joint_plot_data.hpp>
#include <dynamixel_sdk/dynamixel_sdk.h>
#include <Eigen/Dense>

#include "control_common/control_params.hpp"
#include "control_common/control_utils.hpp"

#ifndef PROFILING
#define PROFILING 1
#endif

using SetXY = controller_msgs::msg::SetXY;
using Plot  = controller_msgs::msg::JointPlotData;
namespace ch = std::chrono;
using namespace std::chrono_literals;
using std::placeholders::_1;

/* ── Joint hard-limits (deg) ─────────────────────────────────────────── */
constexpr double Q1_MIN_DEG = -90.0;
constexpr double Q1_MAX_DEG =  20.0;
constexpr double Q2_MIN_DEG =  10.0;
constexpr double Q2_MAX_DEG = 140.0;

/* ═══════════════════════════════════════════════════════════════════════ */
class JointSpaceIDCNode final : public rclcpp::Node
{
public:
  JointSpaceIDCNode() : Node("joint_space_idc")
  {
    /* ── parameters ──────────────────────────────────────────────────── */
    declare_parameter("port",           std::string{"/dev/ttyUSB0"});
    declare_parameter("baud",           1'000'000);
    declare_parameter("kp1_joint",      50.0);
    declare_parameter("kd1_joint",       1.0);
```

```
  declare_parameter("kp2_joint",      50.0);
  declare_parameter("kd2_joint",       1.0);
  declare_parameter("ctrl_rate_hz",   500);

  kp1_ = get_parameter("kp1_joint").as_double();
  kd1_ = get_parameter("kd1_joint").as_double();
  kp2_ = get_parameter("kp2_joint").as_double();
  kd2_ = get_parameter("kd2_joint").as_double();
  dt_  = 1.0 / get_parameter("ctrl_rate_hz").as_int();

  /* ── Dynamixel initialisation ──────────────────────────────── */
  port_   = dynamixel::PortHandler::getPortHandler(
              get_parameter("port").as_string().c_str());
  packet_ = dynamixel::PacketHandler::getPacketHandler(2.0);
  if (!port_->openPort() ||
      !port_->setBaudRate(get_parameter("baud").as_int()))
    throw std::runtime_error("DXL: cannot open port or set baud");

  /* bulk read   : 8 bytes / ID  (vel + pos) */
  bulk_ = new dynamixel::GroupBulkRead(port_, packet_);
  sync_ = new dynamixel::GroupSyncWrite(port_, packet_,
                                        control::ADDR_GOAL_POSITION, 4);
  constexpr uint16_t VEL_ADDR = control::ADDR_PRESENT_VELOCITY;
  if (!bulk_->addParam(control::ID1, VEL_ADDR, 8) ||
      !bulk_->addParam(control::ID2, VEL_ADDR, 8))
    throw std::runtime_error("DXL: GroupBulkRead addParam failed");

  for (uint8_t id : control::SERVO_IDS) {
    write8(id, control::ADDR_TORQUE_ENABLE, 0);
    write8(id, control::ADDR_OPERATING_MODE, control::MODE_POSITION);
    write8(id, control::ADDR_TORQUE_ENABLE, 1);
  }

  /* ── ROS 2 I/O ─────────────────────────────────────────────── */
  sub_xy_   = create_subscription<SetXY>(
                "set_xy", 10, std::bind(&JointSpaceIDCNode::cb_target, this, _1));
  pub_plot_ = create_publisher<Plot>("idc_joint/joint_plot_data", 10);
  timer_    = create_wall_timer(
                ch::duration<double>(dt_),
                std::bind(&JointSpaceIDCNode::cb_loop, this));

  RCLCPP_INFO(get_logger(),
    "Joint-Space IDC ready | dt=%.4f s (%.0f Hz)  kp=[%.1f,%.1f] kd=[%.1f,%.1f]",
    dt_, 1.0/dt_, kp1_, kp2_, kd1_, kd2_);
}

~JointSpaceIDCNode() override
{
  for (uint8_t id : control::SERVO_IDS)
    write8(id, control::ADDR_TORQUE_ENABLE, 0);
  delete bulk_;
  delete sync_;
  port_->closePort();
}
```

```
private:
  /* ── low-level helpers ──────────────────────────────────────────── */
  void write8(uint8_t id, uint16_t addr, uint8_t data)
  { uint8_t err{}; packet_->write1ByteTxRx(port_, id, addr, data, &err); }

  bool read_states(double &q1, double &q2, double &dq1, double &dq2)
  {
    if (bulk_->txRxPacket() != COMM_SUCCESS) return false;
    auto ready = [&](uint8_t id, uint16_t a){ return bulk_->isAvailable(id, a, 4); };
    if (!(ready(control::ID1, control::ADDR_PRESENT_POSITION) &&
          ready(control::ID1, control::ADDR_PRESENT_VELOCITY) &&
          ready(control::ID2, control::ADDR_PRESENT_POSITION) &&
          ready(control::ID2, control::ADDR_PRESENT_VELOCITY)))
      return false;

    int32_t v1 = bulk_->getData(control::ID1, control::ADDR_PRESENT_VELOCITY, 4);
    int32_t p1 = bulk_->getData(control::ID1, control::ADDR_PRESENT_POSITION, 4);
    int32_t v2 = bulk_->getData(control::ID2, control::ADDR_PRESENT_VELOCITY, 4);
    int32_t p2 = bulk_->getData(control::ID2, control::ADDR_PRESENT_POSITION, 4);

    dq1 = control::vel_tick2rad(v1);
    dq2 = control::vel_tick2rad(v2);
    q1  = control::tick2deg(0, p1) * M_PI/180.0;
    q2  = control::tick2deg(1, p2) * M_PI/180.0;
    return true;
  }

  void send_goal(uint32_t t1, uint32_t t2)
  {
    uint8_t b1[4] = { DXL_LOBYTE(DXL_LOWORD(t1)), DXL_HIBYTE(DXL_LOWORD(t1)),
                      DXL_LOBYTE(DXL_HIWORD(t1)), DXL_HIBYTE(DXL_HIWORD(t1)) };
    uint8_t b2[4] = { DXL_LOBYTE(DXL_LOWORD(t2)), DXL_HIBYTE(DXL_LOWORD(t2)),
                      DXL_LOBYTE(DXL_HIWORD(t2)), DXL_HIBYTE(DXL_HIWORD(t2)) };
    sync_->clearParam();
    sync_->addParam(control::ID1, b1);
    sync_->addParam(control::ID2, b2);
    sync_->txPacket();
  }

#ifdef PROFILING
  struct Stat { double avg=0, worst=0;
    void update(double s){ const double k=.01; avg = (1-k)*avg + k*s; worst =
std::max(worst, s); } } st_[5];
  uint64_t loops_{0};
#endif

  /* ── callbacks ──────────────────────────────────────────────────── */
  void cb_target(const SetXY::SharedPtr msg)
  {
    if (!std::isfinite(msg->x) || !std::isfinite(msg->y)) return;

    double j1, j2;
    if (!control::ik_xy(msg->x * 0.01, msg->y * 0.01, j1, j2)) {
      RCLCPP_WARN_THROTTLE(get_logger(), *get_clock(), 2000,
                           "IK failed for (%.1f, %.1f) cm", msg->x, msg->y);
```

```
    return;
  }
  qd_ = { j1, j2 };
  has_target_ = true;
}

void cb_loop()
{
  auto t0 = ch::steady_clock::now();
  if (!has_target_) return;

  /* 1 ▸ state read ------------------------------------------------ */
  double q1, q2, dq1, dq2;
  if (!read_states(q1, q2, dq1, dq2)) return;
  auto t1 = ch::steady_clock::now();

  Eigen::Vector2d q{ q1, q2 }, dq{ dq1, dq2 };

  /* 2 ▸ inverse dynamics ------------------------------------------ */
  Eigen::Vector2d e  = qd_ - q;
  Eigen::Vector2d ed = -dq;                    // desired qd̈ = 0
  Eigen::Vector2d qdd{ kp1_*e[0] + kd1_*ed[0],
                       kp2_*e[1] + kd2_*ed[1] };

  Eigen::Vector2d tau = control::mass_matrix(q) * qdd +
                        control::coriolis(q, dq) +
                        control::gravity(q);
  auto t2 = ch::steady_clock::now();

  /* 3 ▸ convert τ → Δθdeg and command ----------------------------- */
  Eigen::Vector2d dqdeg = control::tau_to_deg(tau);
  Eigen::Vector2d qdeg  = (q * 180.0 / M_PI) + dqdeg;

  qdeg[0] = std::clamp(qdeg[0], Q1_MIN_DEG, Q1_MAX_DEG);
  qdeg[1] = std::clamp(qdeg[1], Q2_MIN_DEG, Q2_MAX_DEG);

  send_goal(control::deg2tick(0, qdeg[0]),
            control::deg2tick(1, qdeg[1]));
  auto t3 = ch::steady_clock::now();

  /* 4 ▸ publish plot data ----------------------------------------- */
  Eigen::Vector2d xy = control::fk_xy(q);   // for visualisation
  Plot p;
  p.stamp       = now();
  p.q1_deg      = qdeg[0];
  p.q2_deg      = qdeg[1];
  p.q1_des_deg  = qd_[0] * 180.0 / M_PI;
  p.q2_des_deg  = qd_[1] * 180.0 / M_PI;
  p.e1          = p.q1_des_deg - qdeg[0];
  p.e2          = p.q2_des_deg - qdeg[1];
  p.x           = xy[0];
  p.y           = xy[1];
  // Add for logger:
  p.measured_q1 = qdeg[0];
  p.measured_q2 = qdeg[1];
```

```
    p.desired_q1  = p.q1_des_deg;
    p.desired_q2  = p.q2_des_deg;
    p.pwm1 = 0.0;
    p.pwm2 = 0.0;
    p.controller_name = "joint_space_idc";
    pub_plot_->publish(p);
    auto t4 = ch::steady_clock::now();

#ifdef PROFILING
    double us_r = ch::duration_cast<ch::microseconds>(t1-t0).count();
    double us_d = ch::duration_cast<ch::microseconds>(t2-t1).count();
    double us_w = ch::duration_cast<ch::microseconds>(t3-t2).count();
    double us_p = ch::duration_cast<ch::microseconds>(t4-t3).count();
    double us_t = ch::duration_cast<ch::microseconds>(t4-t0).count();
    st_[0].update(us_r); st_[1].update(us_d); st_[2].update(us_w);
    st_[3].update(us_p); st_[4].update(us_t);

    if (++loops_ % 100 == 0 || us_t > dt_ * 1e6)
      RCLCPP_INFO(get_logger(),
                  "LOOP#%llu  avgµs [read %.1f | dyn %.1f | write %.1f | pub %.1f |
total %.1f]  worst %.1f",
                  static_cast<unsigned long long>(loops_),
                  st_[0].avg, st_[1].avg, st_[2].avg, st_[3].avg,
                  st_[4].avg, st_[4].worst);
#endif
  }

  /* ── members ──────────────────────────────────────────────────────── */
  rclcpp::Subscription<SetXY>::SharedPtr sub_xy_;
  rclcpp::Publisher<Plot>::SharedPtr     pub_plot_;
  rclcpp::TimerBase::SharedPtr           timer_;

  dynamixel::PortHandler*   port_{};
  dynamixel::PacketHandler* packet_{};
  dynamixel::GroupBulkRead*  bulk_{};
  dynamixel::GroupSyncWrite* sync_{};

  Eigen::Vector2d qd_{0, 0};   // desired joint angles [rad]
  bool            has_target_{false};

  double kp1_, kd1_, kp2_, kd2_, dt_;
};

/* ── main ─────────────────────────────────────────────────────────────── */
int main(int argc, char** argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<JointSpaceIDCNode>());
  rclcpp::shutdown();
  return 0;
}
```

Operational Space Inverse Dynamics Controller

```
/* ============================================================================
 * inverse_dynamics_node.cpp — latency-optimised  +  joint-limit clamp
 * ----------------------------------------------------------------------------
 *  Subscribes   :  /set_xy  (controller_msgs/SetXY, cm in camera plane)
 *  Computes     :  planar 2-DOF Operational Space ID  ➜ pwm signal
 *  Writes to    :  Dynamixel PWM Registers
 *  Publishes on :  /idc/joint_plot_data  (controller_msgs/JointPlotData)
 * ============================================================================ */
#include <chrono>
#include <cmath>
#include <algorithm>
#include <rclcpp/rclcpp.hpp>
#include <controller_msgs/msg/set_xy.hpp>
#include <controller_msgs/msg/joint_plot_data.hpp>
#include <dynamixel_sdk/dynamixel_sdk.h>
#include <Eigen/Dense>

#include "control_common/control_params.hpp"
#include "control_common/control_utils.hpp"

#ifndef PROFILING
#define PROFILING 1
#endif

using SetXY = controller_msgs::msg::SetXY;
using Plot  = controller_msgs::msg::JointPlotData;
namespace ch = std::chrono;
using namespace std::chrono_literals;
using std::placeholders::_1;

/* ── Joint hard-limits (deg) ─────────────────────────────────────────── */
constexpr double Q1_MIN_DEG = -90.0;
constexpr double Q1_MAX_DEG =  20.0;
constexpr double Q2_MIN_DEG =  10.0;
constexpr double Q2_MAX_DEG = 140.0;

/* ═══════════════════════════════════════════════════════════════════════ */
class InverseDynamicsNode final : public rclcpp::Node
{
public:
  InverseDynamicsNode() : Node("inverse_dynamics_node")
  {
    /* parameters */
    declare_parameter("port",          std::string{"/dev/ttyUSB0"});
    declare_parameter("baud",          1'000'000);
    declare_parameter("kp1_task",      50.0);
    declare_parameter("kd1_task",       1.0);
    declare_parameter("kp2_task",      50.0);
    declare_parameter("kd2_task",       1.0);
    declare_parameter("ctrl_rate_hz",  500);

    kp1_ = get_parameter("kp1_task").as_double();
    kd1_ = get_parameter("kd1_task").as_double();
    kp2_ = get_parameter("kp2_task").as_double();
```

```
    kd2_ = get_parameter("kd2_task").as_double();
    dt_  = 1.0 / get_parameter("ctrl_rate_hz").as_int();

    /* Dynamixel init */
    port_   = dynamixel::PortHandler::getPortHandler(
                get_parameter("port").as_string().c_str());
    packet_ = dynamixel::PacketHandler::getPacketHandler(2.0);
    if (!port_->openPort() ||
        !port_->setBaudRate(get_parameter("baud").as_int()))
      throw std::runtime_error("DXL: cannot open port or set baud");

    /* bulk read: 8 bytes per ID  (vel-pos) */
    bulk_ = new dynamixel::GroupBulkRead(port_, packet_);
    sync_ = new dynamixel::GroupSyncWrite(port_, packet_,
                                          control::ADDR_GOAL_POSITION, 4);
    constexpr uint16_t VEL_ADDR = control::ADDR_PRESENT_VELOCITY; // 128
    if (!bulk_->addParam(control::ID1, VEL_ADDR, 8) ||
        !bulk_->addParam(control::ID2, VEL_ADDR, 8))
      throw std::runtime_error("DXL: GroupBulkRead addParam failed");

    for (uint8_t id : control::SERVO_IDS) {
      write8(id, control::ADDR_TORQUE_ENABLE, 0);
      write8(id, control::ADDR_OPERATING_MODE, control::MODE_POSITION);
      write8(id, control::ADDR_TORQUE_ENABLE, 1);
    }

    /* ROS 2 I/O */
    sub_xy_   = create_subscription<SetXY>(
                  "set_xy", 10, std::bind(&InverseDynamicsNode::cb_target, this, _1));
    pub_plot_ = create_publisher<Plot>("idc/joint_plot_data", 10);
    timer_    = create_wall_timer(
                  ch::duration<double>(dt_), std::bind(&InverseDynamicsNode::cb_loop,
this));

    RCLCPP_INFO(get_logger(), "IDC ready | dt=%.4f s (%.0f Hz)", dt_, 1.0/dt_);
  }

  ~InverseDynamicsNode() override
  {
    for (uint8_t id : control::SERVO_IDS)
      write8(id, control::ADDR_TORQUE_ENABLE, 0);
    delete bulk_;
    delete sync_;
    port_->closePort();
  }

private:
  /* low-level helpers */
  void write8(uint8_t id,uint16_t addr,uint8_t data)
  { uint8_t err=0; packet_->write1ByteTxRx(port_,id,addr,data,&err); }

  bool read_states(double &q1,double &q2,double &dq1,double &dq2)
  {
    if (bulk_->txRxPacket() != COMM_SUCCESS) return false;
    auto ok=[&](uint8_t id,uint16_t a){return bulk_->isAvailable(id,a,4);};
```

```
    if(!(ok(control::ID1, control::ADDR_PRESENT_POSITION ) &&
         ok(control::ID1, control::ADDR_PRESENT_VELOCITY ) &&
         ok(control::ID2, control::ADDR_PRESENT_POSITION ) &&
         ok(control::ID2, control::ADDR_PRESENT_VELOCITY ))) return false;

    int32_t v1 = bulk_->getData(control::ID1, control::ADDR_PRESENT_VELOCITY, 4);
    int32_t t1 = bulk_->getData(control::ID1, control::ADDR_PRESENT_POSITION, 4);
    int32_t v2 = bulk_->getData(control::ID2, control::ADDR_PRESENT_VELOCITY, 4);
    int32_t t2 = bulk_->getData(control::ID2, control::ADDR_PRESENT_POSITION, 4);

    dq1 = control::vel_tick2rad(v1);
    dq2 = control::vel_tick2rad(v2);
    q1  = control::tick2deg(0, t1) * M_PI/180.0;
    q2  = control::tick2deg(1, t2) * M_PI/180.0;
    return true;
  }

  void send_goal(uint32_t t1,uint32_t t2)
  {
    uint8_t b1[4]={DXL_LOBYTE(DXL_LOWORD(t1)),DXL_HIBYTE(DXL_LOWORD(t1)),
                   DXL_LOBYTE(DXL_HIWORD(t1)),DXL_HIBYTE(DXL_HIWORD(t1))};
    uint8_t b2[4]={DXL_LOBYTE(DXL_LOWORD(t2)),DXL_HIBYTE(DXL_LOWORD(t2)),
                   DXL_LOBYTE(DXL_HIWORD(t2)),DXL_HIBYTE(DXL_HIWORD(t2))};
    sync_->clearParam();
    sync_->addParam(control::ID1,b1);
    sync_->addParam(control::ID2,b2);
    sync_->txPacket();
  }

#ifdef PROFILING
  struct Stat{double avg=0,worst=0;void u(double s){const double
k=.01;avg=(1-k)*avg+k*s;worst=std::max(worst,s);} } st_[5];
  uint64_t loops_=0;
#endif

  /* callbacks */
  void cb_target(const SetXY::SharedPtr m)
  {
    if(!std::isfinite(m->x)||!std::isfinite(m->y)) return;
    target_xy_ = { m->x*0.01, m->y*0.01 };
    has_target_= true;
  }

  void cb_loop()
  {
    auto t0 = ch::steady_clock::now();
    if(!has_target_) return;

    /* 1 ▸ read */
    double q1,q2,dq1,dq2;
    if(!read_states(q1,q2,dq1,dq2)) return;
    auto t1 = ch::steady_clock::now();

    /* 2 ▸ dynamics */
    Eigen::Vector2d q{q1,q2}, dq{dq1,dq2};
```

```
    Eigen::Vector2d x  = control::fk_xy(q);
    Eigen::Matrix2d J  = control::jacobian(q);
    Eigen::Vector2d xd = J*dq;
    Eigen::Vector2d e  = target_xy_ - x;
    Eigen::Vector2d ed = -xd;
    Eigen::Vector2d xdd{ kp1_*e[0]+kd1_*ed[0], kp2_*e[1]+kd2_*ed[1] };
    Eigen::Vector2d qdd = J.inverse() * (xdd - control::jdot_qdot(q,dq));
    Eigen::Vector2d tau = control::mass_matrix(q)*qdd +
                          control::coriolis(q,dq) +
                          control::gravity(q);
    auto t2 = ch::steady_clock::now();

    /* 3 ▸ write (with clamp) */
    Eigen::Vector2d dqdeg = control::tau_to_deg(tau);
    Eigen::Vector2d qdeg  = (q*180.0/M_PI) + dqdeg;

    qdeg[0] = std::clamp(qdeg[0], Q1_MIN_DEG, Q1_MAX_DEG);
    qdeg[1] = std::clamp(qdeg[1], Q2_MIN_DEG, Q2_MAX_DEG);

    send_goal(control::deg2tick(0,qdeg[0]), control::deg2tick(1,qdeg[1]));
    auto t3 = ch::steady_clock::now();

    /* 4 ▸ publish */
    Plot p; p.stamp=now(); p.q1_deg=qdeg[0]; p.q2_deg=qdeg[1];
    p.x = x[0]; p.y = x[1];
    // Add for logger:
    p.measured_q1 = qdeg[0];
    p.measured_q2 = qdeg[1];
    p.desired_q1  = qdeg[0]; // No explicit desired q in operational space, so use
current
    p.desired_q2  = qdeg[1];
    p.pwm1 = 0.0;
    p.pwm2 = 0.0;
    p.controller_name = "inverse_dynamics_node";
    pub_plot_->publish(p);
    auto t4 = ch::steady_clock::now();

#ifdef PROFILING
    double us_r=ch::duration_cast<ch::microseconds>(t1-t0).count();
    double us_d=ch::duration_cast<ch::microseconds>(t2-t1).count();
    double us_w=ch::duration_cast<ch::microseconds>(t3-t2).count();
    double us_p=ch::duration_cast<ch::microseconds>(t4-t3).count();
    double us_t=ch::duration_cast<ch::microseconds>(t4-t0).count();
    st_[0].u(us_r); st_[1].u(us_d); st_[2].u(us_w); st_[3].u(us_p); st_[4].u(us_t);
    if(++loops_%100==0 || us_t>dt_*1e6)
      RCLCPP_INFO(get_logger(),
        "LOOP #%llu  avg_us [read %.1f | dyn %.1f | write %.1f | pub %.1f | total
%.1f]  worst_total %.1f",
        static_cast<unsigned long long>(loops_),
        st_[0].avg, st_[1].avg, st_[2].avg, st_[3].avg, st_[4].avg, st_[4].worst);
#endif
  }

  /* members */
  rclcpp::Subscription<SetXY>::SharedPtr sub_xy_;
```

```
  rclcpp::Publisher<Plot>::SharedPtr     pub_plot_;
  rclcpp::TimerBase::SharedPtr           timer_;

  dynamixel::PortHandler*   port_{};
  dynamixel::PacketHandler* packet_{};
  dynamixel::GroupBulkRead*  bulk_{};
  dynamixel::GroupSyncWrite* sync_{};

  Eigen::Vector2d target_xy_{0,0};
  bool            has_target_{false};

  double kp1_,kd1_,kp2_,kd2_,dt_;
};

/* ── main ──────────────────────────────────────────────────────────── */
int main(int argc,char** argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<InverseDynamicsNode>());
  rclcpp::shutdown();
  return 0;
}
```

Robust Joint Controller

```
/* ============================================================================
 * robust_joint_control.cpp  —  Robust Joint-Space Inverse-Dynamics Control
 * ----------------------------------------------------------------------------
 *  Subscribes   :  /set_xy  (controller_msgs/SetXY, cm in camera plane)
 *  Computes     :  planar 2-DOF Operational Space ID  ➜ pwm signal
 *  Writes to    :  Dynamixel PWM Registers
 *  Publishes on :  /robust_joint/joint_plot_data  (controller_msgs/JointPlotData)

 * ============================================================================ */
#include <chrono>
#include <cmath>
#include <algorithm>
#include <rclcpp/rclcpp.hpp>
#include <controller_msgs/msg/set_xy.hpp>
#include <controller_msgs/msg/joint_plot_data.hpp>
#include <dynamixel_sdk/dynamixel_sdk.h>
#include <Eigen/Dense>

#include "control_common/control_params.hpp"
#include "control_common/control_utils.hpp"

using SetXY = controller_msgs::msg::SetXY;
using Plot  = controller_msgs::msg::JointPlotData;
namespace ch = std::chrono;
using namespace std::chrono_literals;
using std::placeholders::_1;

/* ---- joint hard limits (deg) ------------------------------------------- */
constexpr double Q1_MIN_DEG = -90.0, Q1_MAX_DEG =  20.0;
constexpr double Q2_MIN_DEG =  10.0, Q2_MAX_DEG = 140.0;

/* ============================================================================ */
class RobustJointControl : public rclcpp::Node
{
public:
  RobustJointControl() : Node("robust_joint_control")
  {
    /* ---- parameters ------------------------------------------------- */
    declare_parameter("port",          std::string{"/dev/ttyUSB0"});
    declare_parameter("baud",          1'000'000);
    declare_parameter("kp1_joint",     50.0);
    declare_parameter("kd1_joint",      1.0);
    declare_parameter("kp2_joint",     50.0);
    declare_parameter("kd2_joint",      1.0);
    declare_parameter("rho",            5.0);
    declare_parameter("epsilon",        0.01);
    declare_parameter("ctrl_rate_hz",  500);

    kp1_ = get_parameter("kp1_joint").as_double();
    kd1_ = get_parameter("kd1_joint").as_double();
    kp2_ = get_parameter("kp2_joint").as_double();
    kd2_ = get_parameter("kd2_joint").as_double();
    rho_ = get_parameter("rho").as_double();
    eps_ = get_parameter("epsilon").as_double();
```

```
    dt_  = 1.0 / get_parameter("ctrl_rate_hz").as_int();

    /* ---- Dynamixel init (identical to joint_space_idc) --------------- */
    port_   = dynamixel::PortHandler::getPortHandler(
                get_parameter("port").as_string().c_str());
    packet_ = dynamixel::PacketHandler::getPacketHandler(2.0);
    if (!port_->openPort() || !port_->setBaudRate(get_parameter("baud").as_int()))
      throw std::runtime_error("DXL: cannot open / set baud");

    bulk_ = new dynamixel::GroupBulkRead(port_, packet_);
    sync_ = new dynamixel::GroupSyncWrite(port_, packet_,
                                          control::ADDR_GOAL_POSITION, 4);
    constexpr uint16_t VEL_ADDR = control::ADDR_PRESENT_VELOCITY;
    if (!bulk_->addParam(control::ID1, VEL_ADDR, 8) ||
        !bulk_->addParam(control::ID2, VEL_ADDR, 8))
      throw std::runtime_error("DXL: GroupBulkRead addParam failed");

    for (uint8_t id : control::SERVO_IDS) {
      write8(id, control::ADDR_TORQUE_ENABLE, 0);
      write8(id, control::ADDR_OPERATING_MODE, control::MODE_POSITION);
      write8(id, control::ADDR_TORQUE_ENABLE, 1);
    }

    /* ---- ROS I/O ----------------------------------------------------- */
    sub_xy_   = create_subscription<SetXY>("set_xy", 10,
                  std::bind(&RobustJointControl::cb_target, this, _1));
    pub_plot_ = create_publisher<Plot>("robust_joint/joint_plot_data", 10);
    timer_    = create_wall_timer(ch::duration<double>(dt_),
                  std::bind(&RobustJointControl::cb_loop, this));

    RCLCPP_INFO(get_logger(),
      "Robust IDC ready  dt=%g s | rho=%g eps=%g", dt_, rho_, eps_);
  }

  ~RobustJointControl() override
  {
    for (uint8_t id : control::SERVO_IDS)
      write8(id, control::ADDR_TORQUE_ENABLE, 0);
    delete bulk_; delete sync_; port_->closePort();
  }

private:
  /* ---- low-level helpers ------------------------------------------- */
  void write8(uint8_t id, uint16_t addr, uint8_t data)
  { uint8_t err{}; packet_->write1ByteTxRx(port_, id, addr, data, &err); }

  bool read_state(double &q1,double &q2,double &dq1,double &dq2)
  {
    if (bulk_->txRxPacket() != COMM_SUCCESS) return false;
    auto ok=[&](uint8_t id,uint16_t a){return bulk_->isAvailable(id,a,4);};
    if(!(ok(control::ID1, control::ADDR_PRESENT_POSITION) &&
         ok(control::ID1, control::ADDR_PRESENT_VELOCITY) &&
         ok(control::ID2, control::ADDR_PRESENT_POSITION) &&
         ok(control::ID2, control::ADDR_PRESENT_VELOCITY))) return false;
```

```
  int32_t v1=bulk_->getData(control::ID1, control::ADDR_PRESENT_VELOCITY,4);
  int32_t p1=bulk_->getData(control::ID1, control::ADDR_PRESENT_POSITION,4);
  int32_t v2=bulk_->getData(control::ID2, control::ADDR_PRESENT_VELOCITY,4);
  int32_t p2=bulk_->getData(control::ID2, control::ADDR_PRESENT_POSITION,4);

  dq1 = control::vel_tick2rad(v1);
  dq2 = control::vel_tick2rad(v2);
  q1  = control::tick2deg(0,p1)*M_PI/180.0;
  q2  = control::tick2deg(1,p2)*M_PI/180.0;
  return true;
}

void send_goal(uint32_t t1,uint32_t t2)
{
  uint8_t b1[4]={DXL_LOBYTE(DXL_LOWORD(t1)),DXL_HIBYTE(DXL_LOWORD(t1)),
                 DXL_LOBYTE(DXL_HIWORD(t1)),DXL_HIBYTE(DXL_HIWORD(t1))};
  uint8_t b2[4]={DXL_LOBYTE(DXL_LOWORD(t2)),DXL_HIBYTE(DXL_LOWORD(t2)),
                 DXL_LOBYTE(DXL_HIWORD(t2)),DXL_HIBYTE(DXL_HIWORD(t2))};
  sync_->clearParam();
  sync_->addParam(control::ID1,b1);
  sync_->addParam(control::ID2,b2);
  sync_->txPacket();
}

/* ---- callbacks ------------------------------------------------------ */
void cb_target(const SetXY::SharedPtr m)
{
  if(!std::isfinite(m->x)||!std::isfinite(m->y)) return;
  double j1,j2;
  if(!control::ik_xy(m->x*0.01, m->y*0.01, j1, j2)){
    RCLCPP_WARN_THROTTLE(get_logger(), *get_clock(), 2000,
      "IK fail (%.1f,%.1f)", m->x, m->y);
    return;
  }
  qd_ = {j1,j2};  has_target_ = true;
}

void cb_loop()
{
  if(!has_target_) return;

  double q1,q2,dq1,dq2;
  if(!read_state(q1,q2,dq1,dq2)) return;

  Eigen::Vector2d q{q1,q2},               dq{dq1,dq2};
  Eigen::Vector2d qd = qd_,               dqd = {0,0},  ddqd = {0,0};
  Eigen::Vector2d q_tilde  = qd - q;
  Eigen::Vector2d dq_tilde = dqd - dq;

  /* ---- error state ξ & sliding vector z ---------------------------- */
  Eigen::Vector4d xi; xi << q_tilde, dq_tilde;
  Eigen::Matrix<double,2,4> D_T;  // actually Dᵀ with rows that pick velocities
  D_T.setZero();  D_T(0,2)=1; D_T(1,3)=1;
  Eigen::Vector2d z = D_T * xi;   // Q = I so Dᵀ Q ξ = Dᵀ ξ
```

```
    /* ---- robust term ------------------------------------------------ */
    Eigen::Vector2d w = control::robust_w(z, rho_, eps_);

    /* ---- desired accel (PD+robust) ---------------------------------- */
    Eigen::Vector2d qdd_star;
    qdd_star[0] = ddqd[0] + kd1_*dq_tilde[0] + kp1_*q_tilde[0] + w[0];
    qdd_star[1] = ddqd[1] + kd2_*dq_tilde[1] + kp2_*q_tilde[1] + w[1];

    Eigen::Vector2d tau = control::mass_matrix(q)*qdd_star +
                          control::coriolis(q,dq) +
                          control::gravity(q);

    /* ---- position-mode conversion ----------------------------------- */
    Eigen::Vector2d qdeg = (q*180.0/M_PI) + control::tau_to_deg(tau);
    qdeg[0]=std::clamp(qdeg[0],Q1_MIN_DEG,Q1_MAX_DEG);
    qdeg[1]=std::clamp(qdeg[1],Q2_MIN_DEG,Q2_MAX_DEG);
    send_goal(control::deg2tick(0,qdeg[0]), control::deg2tick(1,qdeg[1]));

    /* ---- publish plot ----------------------------------------------- */
    Plot p;
    p.stamp=now();
    p.q1_deg=qdeg[0]; p.q2_deg=qdeg[1];
    p.q1_des_deg=qd[0]*180.0/M_PI;
    p.q2_des_deg=qd[1]*180.0/M_PI;
    p.e1=p.q1_des_deg-p.q1_deg;
    p.e2=p.q2_des_deg-p.q2_deg;
    auto xy = control::fk_xy(q);
    p.x=xy[0]; p.y=xy[1];
    // Add for logger:
    p.measured_q1 = qdeg[0];
    p.measured_q2 = qdeg[1];
    p.desired_q1  = p.q1_des_deg;
    p.desired_q2  = p.q2_des_deg;
    p.pwm1 = 0.0;
    p.pwm2 = 0.0;
    p.controller_name = "robust_joint_control";
    pub_plot_->publish(p);
  }

  /* ---- members ------------------------------------------------------ */
  rclcpp::Subscription<SetXY>::SharedPtr sub_xy_;
  rclcpp::Publisher<Plot>::SharedPtr     pub_plot_;
  rclcpp::TimerBase::SharedPtr           timer_;

  dynamixel::PortHandler*   port_{};
  dynamixel::PacketHandler* packet_{};
  dynamixel::GroupBulkRead*  bulk_{};
  dynamixel::GroupSyncWrite* sync_{};

  Eigen::Vector2d qd_{0,0};
  bool   has_target_{false};
  double kp1_,kd1_,kp2_,kd2_,rho_,eps_,dt_;
};

/* main ------------------------------------------------------------------ */
```

```
int main(int argc,char** argv)
{
  rclcpp::init(argc,argv);
  rclcpp::spin(std::make_shared<RobustJointControl>());
  rclcpp::shutdown();
  return 0;
}


       Vision Parameters
#pragma once
#include <opencv2/imgproc.hpp>
#include <vector>
#include <algorithm>
#include <cmath>

/* ── shared vision constants ──────────────────────────────────────── */
namespace vp {
constexpr int    IMG_W = 640, IMG_H = 240;
constexpr int    HALF_W = IMG_W / 2;
constexpr double IMG_CX = HALF_W / 2.0;
constexpr double IMG_CY = IMG_H  / 2.0;

constexpr int    THRESH    = 225;
constexpr int    ERODE_SZ  = 3, DILATE_SZ = 7;
constexpr double MIN_AREA  = 15.0 * 15.0;

constexpr double HFOV_DEG  = 45.0;
constexpr double CAMERA_H  = 85.0;          // camera height [cm]
constexpr double BASELINE  = 5.5;           // stereo baseline [cm]
constexpr double FOCAL_PX  = (HALF_W / 2.0) /
        std::tan((HFOV_DEG * M_PI / 180.0) / 2.0);

/* Physics & filter */
constexpr double G_CM       = 981.0;        // gravity [cm s⁻²]
constexpr double LPF_ALPHA  = 0.7;
constexpr double VEL_ALPHA  = 0.7;
constexpr double PRED_ALPHA = 0.6;
constexpr std::size_t BUF_MAX      = 4;
constexpr std::size_t BUF_MIN      = 2;
constexpr double      RESET_GAP_S  = 0.30;
} // namespace vp

/* ── centroid helper (optional: move to vision_utils.hpp) ─────────── */
inline bool centroid(const cv::Mat& roi,
                     double& cx, double& cy, double& area,
                     const cv::Mat& erodeK, const cv::Mat& dilateK)
{
  cv::Mat g, m;  cv::cvtColor(roi, g, cv::COLOR_BGR2GRAY);
  cv::threshold(g, m, vp::THRESH, 255, cv::THRESH_BINARY);
  cv::erode(m, m, erodeK);  cv::dilate(m, m, dilateK);

  std::vector<std::vector<cv::Point>> cont;
  cv::findContours(m, cont, cv::RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE);
  if (cont.empty()) return false;
```

```
  auto it = std::max_element(cont.begin(), cont.end(),
            [](auto& a, auto& b){ return cv::contourArea(a) < cv::contourArea(b); });
  area = cv::contourArea(*it);
  if (area < vp::MIN_AREA) return false;

  cv::Moments mom = cv::moments(*it);
  cx = mom.m10 / mom.m00;
  cy = mom.m01 / mom.m00;
  return true;
}
```

Control Parameters

```
#pragma once
#include <array>
#include <cstdint>

namespace control
{
/* ================== Hardware ================== */
constexpr std::array<int,2> SERVO_IDS{4,1};   // ID1, ID2
constexpr uint8_t ID1 = 4, ID2 = 1;

/* Dynamixel registers (XM430-W210) */
constexpr uint16_t ADDR_TORQUE_ENABLE   = 64;
constexpr uint16_t ADDR_OPERATING_MODE  = 11;
constexpr uint16_t ADDR_GOAL_POSITION   = 116;
constexpr uint16_t ADDR_GOAL_PWM        = 100;
constexpr uint16_t ADDR_PRESENT_POSITION= 132;
constexpr uint16_t ADDR_PRESENT_VELOCITY= 128;

/* Operating-mode codes */
constexpr uint8_t MODE_POSITION = 3;
constexpr uint8_t MODE_PWM      = 16;

/* ================== Geometry ================== */
constexpr double L1 = 0.30;                 // [m]
constexpr double L2 = 0.30;                 // [m]
constexpr double BASE_X = -0.42;            // [m] world frame
constexpr double BASE_Y =  0.01;            // [m]
constexpr double Q1_MIN = -80.0 * M_PI / 180.0;
constexpr double Q1_MAX = -10 * M_PI / 180.0;
constexpr double Q2_MIN =  20.0 * M_PI / 180.0;
constexpr double Q2_MAX =  140 * M_PI / 180.0;

/* ================== Servo calibration ========= */
constexpr double ZERO_TICK_1 = 2048;        // ticks when q1 = 0 deg
constexpr double ZERO_TICK_2 = 2048;
constexpr double TICKS_PER_DEG = 4096.0/360.0;

/* Velocity conversion (tick/s → rad/s) */
constexpr double VELOCITY_SCALE = 0.229 * 2*M_PI / 60.0;

/* ================== Dynamics ================== */
constexpr double M1 = 0.17;                 // [kg] link 1
constexpr double M2 = 0.088;                // [kg] link 2
constexpr double LC1 = L1*0.7;              // COM offsets
constexpr double LC2 = L2*0.5;
constexpr double I1 = 0.00110;              // [kg·m²]
constexpr double I2 = 0.00030;

/* PWM / torque limits */
constexpr double TAUMAX1 = 2.5;             // [N·m]
constexpr double TAUMAX2 = 2.5;
constexpr int    PWM_LIM = 885;             // full-scale
} // namespace control
```

Control Utilities

```
#pragma once
#include "control_params.hpp"
#include <Eigen/Dense>
#include <cmath>
#include <algorithm>
#include <dynamixel_sdk/dynamixel_sdk.h>

namespace control
{
/* ---------- tick/deg helpers ----------------------------------- */
inline uint32_t deg2tick(int j,double deg){
  const double z = (j==0?ZERO_TICK_1:ZERO_TICK_2);
  return static_cast<uint32_t>(z + deg*TICKS_PER_DEG);
}
inline double tick2deg(int j,int32_t t){
  const double z = (j==0?ZERO_TICK_1:ZERO_TICK_2);
  return (t - z)/TICKS_PER_DEG;
}
inline double vel_tick2rad(int32_t v){ return v*VELOCITY_SCALE; }

/* ---------- FK  (planar) --------------------------------------- */
inline Eigen::Vector2d fk_xy(const Eigen::Vector2d& q){
  double c1=std::cos(q[0]), s1=std::sin(q[0]);
  double c12=std::cos(q[0]+q[1]), s12=std::sin(q[0]+q[1]);
  return { BASE_X + L1*c1 + L2*c12,
           BASE_Y + L1*s1 + L2*s12 };
}

/* ---------- Planar IK (camera XY → joint rad) ------------------- */
inline bool ik_xy(double x,double y,double& q1,double& q2)
{
  const double dx = x - BASE_X;
  const double dy = y - BASE_Y;
  const double r2 = dx*dx + dy*dy;
  const double L12 = L1 + L2;
  if (r2 > L12*L12) return false;                 // unreachable

  double c2 = (r2 - L1*L1 - L2*L2) / (2*L1*L2);
  c2 = std::clamp(c2, -1.0, 1.0);
  double s2 = std::sqrt(1.0 - c2*c2);
  q2 = std::atan2(s2, c2);
  if (q2 < Q2_MIN || q2 > Q2_MAX) return false;

  q1 = std::atan2(dy, dx) - std::atan2(L2*s2, L1 + L2*c2);
  if (q1 < Q1_MIN || q1 > Q1_MAX) return false;
  return true;                                    // always inside limits for now
}

/* ---------- simple PID struct (for PWM node) -------------------- */
struct PidState
{
  double i = 0, prev = 0;
  double step(double err, double kp, double ki, double kd, int max_out)
  {
```

```
    i += ki * err;
    double d = err - prev;  prev = err;
    double u = kp * err + i + kd * d;
    u = std::clamp(u, -double(max_out), double(max_out));
    if (std::abs(u) >= max_out) i = 0;      // crude anti-windup
    return u;
  }
};

/* ---------- blocking read of the two servos (helper) ------------ */
inline void read_two_positions(dynamixel::PortHandler* port,
                               dynamixel::PacketHandler* pkt,
                               int32_t& t1, int32_t& t2)
{
  uint8_t err{};
  pkt->read4ByteTxRx(port, ID1, ADDR_PRESENT_POSITION, (uint32_t*)&t1, &err);
  pkt->read4ByteTxRx(port, ID2, ADDR_PRESENT_POSITION, (uint32_t*)&t2, &err);
}


/* ---------- Jacobian -------------------------------------------- */
inline Eigen::Matrix2d jacobian(const Eigen::Vector2d& q){
  double s1=std::sin(q[0]), c1=std::cos(q[0]);
  double s12=std::sin(q[0]+q[1]), c12=std::cos(q[0]+q[1]);
  Eigen::Matrix2d J;
  J << -L1*s1 - L2*s12 ,  -L2*s12,
        L1*c1 + L2*c12 ,   L2*c12;
  return J;
}

/* J˙q˙------------------------------------------------------------ */
inline Eigen::Vector2d jdot_qdot(const Eigen::Vector2d& q,
                                 const Eigen::Vector2d& dq){
  double dq12=dq[0]+dq[1];
  double c1=std::cos(q[0]), s1=std::sin(q[0]);
  double c12=std::cos(q[0]+q[1]), s12=std::sin(q[0]+q[1]);

  Eigen::Vector2d v;
  v[0]=(-L1*c1*dq[0]-L2*c12*dq12)*dq[0] + (-L2*c12*dq12)*dq[1];
  v[1]=(-L1*s1*dq[0]-L2*s12*dq12)*dq[0] + (-L2*s12*dq12)*dq[1];
  return v;
}

/* ---------- Mass matrix, Coriolis, Gravity ---------------------- */
inline Eigen::Matrix2d mass_matrix(const Eigen::Vector2d& q){
  double c2=std::cos(q[1]);
  double m11 = I1 + I2 +
               M1*LC1*LC1 +
               M2*(L1*L1 + LC2*LC2 + 2*L1*LC2*c2);
  double m12 = I2 + M2*(LC2*LC2 + L1*LC2*c2);
  double m22 = I2 + M2*LC2*LC2;
  Eigen::Matrix2d M; M << m11, m12, m12, m22; return M;
}
inline Eigen::Vector2d coriolis(const Eigen::Vector2d& q,
                                const Eigen::Vector2d& dq){
```

```
  double s2=std::sin(q[1]);
  double h = -M2*L1*LC2*s2;
  return { h*(2*dq[0]*dq[1] + dq[1]*dq[1]),
          -h*dq[0]*dq[0] };
}
inline Eigen::Vector2d gravity(const Eigen::Vector2d& q){
  /* ignored (arm horizontal) → return zeros */
  return {0,0};
}

/* ---------- Torque → small position step (heuristic) ------------ */
inline Eigen::Vector2d tau_to_deg(const Eigen::Vector2d& tau){
  return { tau[0]/TAUMAX1 * (PWM_LIM/20.0),   //  ~±45 deg FS
           tau[1]/TAUMAX2 * (PWM_LIM/2.0) };
}

/* ---------- Robust Control Parameters ------------ */
inline Eigen::Vector2d robust_w(const Eigen::Vector2d& z,
                                double rho, double eps)
{
  const double norm_z = z.norm();
  const double gain   = (norm_z >= eps) ? (rho / norm_z)
                                        : (rho / eps);
  return gain * z;
}

/* ================================================================
 *  Adaptive-control helpers  (planar 2-DOF arm)
 * ================================================================ */
namespace detail
{
/* dynamic regressor  Y(q,q˙,q¨r)  — 2×10, fixed-size, no heap
   π = [α1 α2 α3 α4 α5  α6 α7 α8 α9 α10]^T encodes link masses,
       inertias, COM offsets.  Expressions follow Spong '06.          */
inline Eigen::Matrix<double,2,10> regressor(
        const Eigen::Vector2d& q,
        const Eigen::Vector2d& dq,
        const Eigen::Vector2d& ddqr)
{
  const double c2 = std::cos(q[1]),
               s2 = std::sin(q[1]);

  /* shorthand */
  const double q1dd = ddqr[0],  q2dd = ddqr[1];
  const double q1d  = dq[0],    q2d  = dq[1];

  Eigen::Matrix<double,2,10> Y; Y.setZero();

  /* Row 1 ---------------------------------------------------------- */
  Y(0,0) = q1dd;
  Y(0,1) = c2*q1dd - s2*q2d*q1d;
  Y(0,2) = q2dd;
  Y(0,3) = q1dd;
  Y(0,4) = 0.0;
  Y(0,5) = 2*c2*q1dd + c2*q2dd - 2*s2*q1d*q2d - s2*q2d*q2d;
```

```
  Y(0,6) = q2dd;
  Y(0,7) = 0.0;
  Y(0,8) = 0.0;
  Y(0,9) = 0.0;

  /* Row 2 ------------------------------------------------------------ */
  Y(1,0) = 0.0;
  Y(1,1) = s2*q1d*q1d + c2*q2dd;
  Y(1,2) = 0.0;
  Y(1,3) = 0.0;
  Y(1,4) = q1dd + q2dd;
  Y(1,5) = s2*q1d*q1d + c2*q1dd;
  Y(1,6) = q1dd + q2dd;
  Y(1,7) = 0.0;
  Y(1,8) = 0.0;
  Y(1,9) = 0.0;

  return Y;
}
} // namespace detail

/* ======== small helper that clamps a vector entry-wise ========== */
template<int N>
inline void clamp_vec(Eigen::Matrix<double,N,1>& v, double lim)
{
  for(int i=0;i<N;++i) v[i] = std::clamp(v[i], -lim, lim);
}


} // namespace control
```