```cpp
#include <ros/ros.h>

#include "quad_utils/mesh_to_grid_map_converter.hpp"

// Standard C++ entry point
int main(int argc, char** argv) {
  // Announce this program to the ROS master
  ros::init(argc, argv, "mesh_to_grid_map");
  ros::NodeHandle nh;
  ros::NodeHandle private_nh("~");

  // Creating the object to do the work.
  mesh_to_grid_map::MeshToGridMapConverter mesh_to_grid_map_converter(
      nh, private_nh);

  ros::spin();
  return 0;
}
```

```cpp
#include <quad_utils/math_utils.h>

namespace math_utils {

std::vector<double> interpMat(const std::vector<double> input_vec,
                              const std::vector<std::vector<double>> input_mat,
                              const double query_point) {
  // Check bounds, throw an error if invalid since this shouldn't ever happen
  if ((query_point < input_vec.front()) || (query_point > input_vec.back())) {
    throw std::runtime_error("Tried to interp out of bounds");
  }

  // Declare variables for interpolating between, both for input and output data
  double t1, t2;
  std::vector<double> y1, y2, interp_data;

  // Find the correct values to interp between
  for (int i = 0; i < input_vec.size(); i++) {
    if (input_vec[i] <= query_point && query_point < input_vec[i + 1]) {
      t1 = input_vec[i];
      t2 = input_vec[i + 1];
      y1 = input_mat[i];
      y2 = input_mat[i + 1];
      break;
    }
  }

  // Apply linear interpolation for each element in the vector
  for (int i = 0; i < input_mat.front().size(); i++) {
    double result = y1[i] + (y2[i] - y1[i]) / (t2 - t1) * (query_point - t1);
    interp_data.push_back(result);
  }

  return interp_data;
}

Eigen::Vector3d interpVector3d(const std::vector<double> input_vec,
                               const std::vector<Eigen::Vector3d> input_mat,
                               const double query_point) {
  // Check bounds, throw an error if invalid since this shouldn't ever happen
  if ((query_point < input_vec.front()) || (query_point > input_vec.back())) {
    throw std::runtime_error("Tried to interp out of bounds");
  }

  // Declare variables for interpolating between, both for input and output data
  double t1, t2;
  Eigen::Vector3d y1, y2, interp_data;

  // Find the correct values to interp between
  for (int i = 0; i < input_vec.size(); i++) {
    if (input_vec[i] <= query_point && query_point < input_vec[i + 1]) {
      t1 = input_vec[i];
      t2 = input_vec[i + 1];
      y1 = input_mat[i];
      y2 = input_mat[i + 1];
      break;
    }
  }

  // Apply linear interpolation for each element in the vector
  interp_data =
      y1.array() + (y2.array() - y1.array()) / (t2 - t1) * (query_point - t1);

  return interp_data;
}

std::vector<Eigen::Vector3d> interpMatVector3d(
    const std::vector<double> input_vec,
    const std::vector<std::vector<Eigen::Vector3d>> output_mat,
    const double query_point) {
  // Check bounds, throw an error if invalid since this shouldn't ever happen
  if ((query_point < input_vec.front()) || (query_point > input_vec.back())) {
    throw std::runtime_error("Tried to interp out of bounds");
  }

  // Declare variables for interpolating between, both for input and output data
  double t1, t2;
```

```
  std::vector<Eigen::Vector3d> y1, y2, interp_data;

  // Find the correct values to interp between
  for (int i = 0; i < input_vec.size(); i++) {
    if (input_vec[i] <= query_point && query_point < input_vec[i + 1]) {
      t1 = input_vec[i];
      t2 = input_vec[i + 1];
      y1 = output_mat[i];
      y2 = output_mat[i + 1];
      break;
    }
  }

  // Apply linear interpolation for each element in the vector
  for (int i = 0; i < output_mat.front().size(); i++) {
    Eigen::Vector3d interp_eigen_vec;
    interp_eigen_vec = y1[i].array() + (y2[i].array() - y1[i].array()) /
                                           (t2 - t1) * (query_point - t1);
    interp_data.push_back(interp_eigen_vec);
  }

  return interp_data;
}

int interpInt(const std::vector<double> input_vec, std::vector<int> output_vec,
              const double query_point) {
  // Check bounds, throw an error if invalid since this shouldn't ever happen
  if ((query_point < input_vec.front()) || (query_point > input_vec.back())) {
    throw std::runtime_error("Tried to interp out of bounds");
  }

  // Declare variables for interpolating between, both for input and output data
  double t1, t2;
  Eigen::Vector3d y1, y2, interp_data;

  // Find the correct values to interp between
  int idx = 0;
  for (int i = 0; i < input_vec.size(); i++) {
    if (input_vec[i] <= query_point && query_point < input_vec[i + 1]) {
      return output_vec[i];
    }
  }

  throw std::runtime_error("Didn't find the query point, something happened");
}

std::vector<double> movingAverageFilter(std::vector<double> data,
                                        int window_size) {
  std::vector<double> filtered_data;
  int N = data.size();

  // Check to ensure window size is an odd integer, if not add one to make it so
  if ((window_size % 2) == 0) {
    window_size += 1;
    ROS_WARN_THROTTLE(
        0.5, "Filter window size is even, adding one to maintain symmetry");
  }

  // Make sure that the window size is acceptable
  if (window_size >= N) {
    ROS_WARN_THROTTLE(0.5, "Filter window size is bigger than data");
  }

  // Loop through the data
  for (int i = 0; i < N; i++) {
    // Initialize sum and count of data samples
    double sum = 0;
    double count = 0;

    // Shrink the window size if it would result in out of bounds data
    int current_window_size = std::min(window_size, 2 * i + 1);
    // int current_window_size = window_size;

    // Loop through the window, adding to the sum and averaging
    for (int j = 0; j < current_window_size; j++) {
      double index = i + (j - (current_window_size - 1) / 2);

```

```
      // Make sure data is in bounds
      if (index >= 0 && index < N) {
        sum += data[index];
        count += 1;
      }
    }

    filtered_data.push_back((float)sum / count);
  }

  return filtered_data;
}

std::vector<double> centralDiff(std::vector<double> data, double dt) {
  std::vector<double> data_diff;

  for (int i = 0; i < data.size(); i++) {
    // Compute lower and upper indices, with forward/backward difference at the
    // ends
    int lower_index = std::max(i - 1, 0);
    int upper_index = std::min(i + 1, (int)data.size() - 1);

    double estimate = (data[upper_index] - data[lower_index]) /
                      (dt * (upper_index - lower_index));
    data_diff.push_back(estimate);
  }

  return data_diff;
}

std::vector<double> unwrap(std::vector<double> data) {
  // Generally to make yaw angle continuous
  std::vector<double> data_unwrapped = data;
  for (int i = 1; i < data.size(); i++) {
    double diff = data[i] - data[i - 1];
    // std::cout << "diff: " << diff*180/M_PI << std::endl;
    if (diff > M_PI) {
      // std::cout << "data[" << i << "]: " << data[i] << "\tdata[" << i-1 <<
      // "]: " << data[i-1] << std::endl; std::cout << "diff > M_PI @ " << i <<
      // " | diff: " << diff*180/M_PI << std::endl;
      for (int j = i; j < data.size(); j++) {
        data_unwrapped[j] = data_unwrapped[j] - 2 * M_PI;
        // std::cout << "data_unwrapped[" << j << "]: " <<
        // data_unwrapped[j]*180/M_PI << std::endl;
      }
    } else if (diff < -M_PI) {
      // std::cout << "data[" << i << "]: " << data[i] << "\tdata[" << i-1 <<
      // "]: " << data[i-1] << std::endl; std::cout << "diff < -M_PI @ " << i <<
      // " | diff: " << diff*180/M_PI << std::endl;
      for (int j = i; j < data.size(); j++) {
        data_unwrapped[j] = data_unwrapped[j] + 2 * M_PI;
        // std::cout << "data_unwrapped[" << j << "]: " <<
        // data_unwrapped[j]*180/M_PI << std::endl;
      }
    }
  }

  return data_unwrapped;
}

Eigen::MatrixXd sdlsInv(const Eigen::MatrixXd &jacobian) {
  Eigen::JacobiSVD<Eigen::MatrixXd> svd(
      jacobian, Eigen::ComputeThinU | Eigen::ComputeThinV);

  Eigen::VectorXd sig = svd.singularValues();
  Eigen::VectorXd sig_inv = sig;

  for (size_t i = 0; i < sig.size(); i++) {
    sig_inv(i) = std::min(1 / sig(i), 1 / (1e-1 * sig.maxCoeff()));
  }

  Eigen::MatrixXd jacobian_inv =
      svd.matrixV() * sig_inv.asDiagonal() * svd.matrixU().transpose();

  return jacobian_inv;
}
}  // namespace math_utils
```

```
#include <ros/ros.h>

#include "quad_utils/terrain_map_publisher.h"

int main(int argc, char** argv) {
  ros::init(argc, argv, "terrain_map_publisher_node");
  ros::NodeHandle nh;

  TerrainMapPublisher terrain_map_publisher(nh);
  terrain_map_publisher.spin();

  return 0;
}
```

```cpp
#include <grid_map_msgs/GridMap.h>
#include <pcl/io/vtk_lib_io.h>
#include <pcl_conversions/pcl_conversions.h>
#include <pcl_ros/point_cloud.h>

#include <grid_map_pcl/GridMapPclConverter.hpp>
#include <grid_map_ros/grid_map_ros.hpp>
#include <quad_utils/mesh_to_grid_map_converter.hpp>
namespace mesh_to_grid_map {

MeshToGridMapConverter::MeshToGridMapConverter(ros::NodeHandle nh,
                                               ros::NodeHandle nh_private)
    : nh_(nh),
      nh_private_(nh_private),
      grid_map_resolution_(kDefaultGridMapResolution),
      layer_name_(kDefaultLayerName),
      latch_grid_map_pub_(kDefaultLatchGridMapPub),
      verbose_(kDefaultVerbose),
      frame_id_mesh_loaded_(kDefaultFrameIdMeshLoaded),
      world_name_(kDefaultWorldName) {
  // Initial interaction with ROS
  subscribeToTopics();
  advertiseTopics();
  advertiseServices();
  getParametersFromRos();

  std::string package_path = ros::package::getPath("gazebo_scripts");
  std::string full_path =
      package_path + "/worlds/" + world_name_ + "/" + world_name_ + ".ply";

  std::cout << full_path << std::endl;
  bool success = loadMeshFromFile(full_path);
}

void MeshToGridMapConverter::subscribeToTopics() {
  mesh_sub_ =
      nh_.subscribe("mesh", 10, &MeshToGridMapConverter::meshCallback, this);
}

void MeshToGridMapConverter::advertiseTopics() {
  grid_map_pub_ = nh_.advertise<grid_map_msgs::GridMap>("terrain_map_raw", 1,
                                                         latch_grid_map_pub_);
}

void MeshToGridMapConverter::advertiseServices() {
  save_grid_map_srv_ = nh_private_.advertiseService(
      "save_grid_map_to_file", &MeshToGridMapConverter::saveGridMapService,
      this);
  load_map_service_server_ = nh_private_.advertiseService(
      "load_mesh_from_file", &MeshToGridMapConverter::loadMeshService, this);
}

void MeshToGridMapConverter::getParametersFromRos() {
  nh_private_.param("grid_map_resolution", grid_map_resolution_,
                    grid_map_resolution_);
  nh_private_.param("layer_name", layer_name_, layer_name_);
  nh_private_.param("latch_grid_map_pub", latch_grid_map_pub_,
                    latch_grid_map_pub_);
  nh_private_.param("verbose", verbose_, verbose_);
  nh_private_.param("frame_id_mesh_loaded", frame_id_mesh_loaded_,
                    frame_id_mesh_loaded_);
  nh_private_.param("world", world_name_, world_name_);
}

void MeshToGridMapConverter::meshCallback(
    const pcl_msgs::PolygonMesh& mesh_msg) {
  if (verbose_) {
    ROS_INFO("Mesh received, starting conversion.");
  }

  // Converting from message to an object
  pcl::PolygonMesh polygon_mesh;
  pcl_conversions::toPCL(mesh_msg, polygon_mesh);
  meshToGridMap(polygon_mesh, mesh_msg.header.frame_id,
                mesh_msg.header.stamp.toNSec());
}

```

```
bool MeshToGridMapConverter::meshToGridMap(
    const pcl::PolygonMesh& polygon_mesh, const std::string& mesh_frame_id,
    const uint64_t& time_stamp_nano_seconds) {
  // Creating the grid map
  grid_map::GridMap map;
  map.setFrameId(mesh_frame_id);

  // Converting
  grid_map::GridMapPclConverter::initializeFromPolygonMesh(
      polygon_mesh, grid_map_resolution_, map);
  const std::string layer_name(layer_name_);
  grid_map::GridMapPclConverter::addLayerFromPolygonMesh(polygon_mesh,
                                                         layer_name, map);

  // Setup x and y matrices for loading
  grid_map::Size map_size = map.getSize();
  Eigen::MatrixXf x_data(map_size(0), map_size(1)),
      y_data(map_size(0), map_size(1));

  // Iterate through map to retrieve x and y data and save to data matrices
  for (grid_map::GridMapIterator iterator(map); !iterator.isPastEnd();
       ++iterator) {
    const grid_map::Index index(*iterator);
    grid_map::Position pos;
    map.getPosition(index, pos);
    x_data(index(0), index(1)) = pos(0);
    y_data(index(0), index(1)) = pos(1);
  }

  // Add x and y layers to map
  map.add("x", x_data);
  map.add("y", y_data);

  // Printing some debug info about the mesh and the map
  if (verbose_) {
    ROS_INFO_STREAM("Number of polygons: " << polygon_mesh.polygons.size());
    ROS_INFO("Created map with size %f x %f m (%i x %i cells).",
             map.getLength().x(), map.getLength().y(), map.getSize()(0),
             map.getSize()(1));
  }

  // Publish grid map.
  map.setTimestamp(time_stamp_nano_seconds);
  grid_map_msgs::GridMap message;
  grid_map::GridMapRosConverter::toMessage(map, message);

  // Publishing the grid map message.
  grid_map_pub_.publish(message);
  if (verbose_) {
    ROS_INFO("Published a grid map message.");
  }

  // Saving the gridmap to the object
  last_grid_map_ptr_.reset(new grid_map::GridMap(map));

  return true;
}

bool MeshToGridMapConverter::saveGridMapService(
    grid_map_msgs::ProcessFile::Request& request,
    grid_map_msgs::ProcessFile::Response& response) {
  // Check there's actually a grid map saved
  if (!last_grid_map_ptr_) {
    ROS_ERROR("No grid map produced yet to save.");
    response.success = static_cast<unsigned char>(false);
  } else {
    response.success = static_cast<unsigned char>(saveGridMap(
        *last_grid_map_ptr_, request.file_path, request.topic_name));
  }

  return true;
}

bool MeshToGridMapConverter::saveGridMap(const grid_map::GridMap& map,
                                         const std::string& path_to_file,
                                         const std::string& topic_name) {
  std::string topic_name_checked = topic_name;
```

```
  if (topic_name.empty()) {
    ROS_WARN(
        "Specified topic name is an empty string, default layer name will be "
        "used as topic name.");
    topic_name_checked = layer_name_;
  }
  // Saving the map
  if (!path_to_file.empty()) {
    if (verbose_) {
      ROS_INFO(
          "Saved the grid map message to file: '%s', with topic name: '%s'.",
          path_to_file.c_str(), topic_name_checked.c_str());
    }
    grid_map::GridMapRosConverter::saveToBag(map, path_to_file,
                                             topic_name_checked);
  } else {
    ROS_ERROR("No rosbag filepath specified where to save grid map.");
    return false;
  }
  return true;
}

bool MeshToGridMapConverter::loadMeshService(
    grid_map_msgs::ProcessFile::Request& request,
    grid_map_msgs::ProcessFile::Response& response) {
  if (!request.topic_name.empty()) {
    ROS_WARN("Field 'topic_name' in service request will not be used.");
  }
  response.success =
      static_cast<unsigned char>(loadMeshFromFile(request.file_path));
  return true;
}

bool MeshToGridMapConverter::loadMeshFromFile(
    const std::string& path_to_mesh_to_load) {
  if (path_to_mesh_to_load.empty()) {
    ROS_ERROR(
        "File path for mesh to load is empty. Please specify a valid path.");
    return false;
  }

  pcl::PolygonMesh mesh_from_file;
  pcl::io::loadPolygonFilePLY(path_to_mesh_to_load, mesh_from_file);

  if (mesh_from_file.polygons.empty()) {
    ROS_ERROR("Mesh read from file is empty!");
    return false;
  }

  bool mesh_converted = meshToGridMap(mesh_from_file, frame_id_mesh_loaded_,
                                      ros::Time::now().toNSec());
  if (!mesh_converted) {
    ROS_ERROR("It was not possible to convert loaded mesh to grid_map object.");
    return false;
  }

  if (verbose_) {
    ROS_INFO("Loaded the mesh from file: %s. Its frame_id is set to '%s'",
             path_to_mesh_to_load.c_str(), frame_id_mesh_loaded_.c_str());
  }

  return true;
}

}  // namespace mesh_to_grid_map
```

```cpp
#include "quad_utils/remote_heartbeat.h"

RemoteHeartbeat::RemoteHeartbeat(ros::NodeHandle nh) {
  nh_ = nh;

  // Load rosparam from parameter server
  std::string remote_heartbeat_topic, robot_heartbeat_topic, leg_control_topic;
  quad_utils::loadROSParam(nh_, "/topics/heartbeat/remote",
                           remote_heartbeat_topic);
  quad_utils::loadROSParam(nh_, "topics/heartbeat/robot",
                           robot_heartbeat_topic);
  quad_utils::loadROSParam(nh_, "remote_heartbeat/robot_latency_threshold_warn",
                           robot_latency_threshold_warn_);
  quad_utils::loadROSParam(nh_,
                           "remote_heartbeat/robot_latency_threshold_error",
                           robot_latency_threshold_error_);
  quad_utils::loadROSParam(nh_, "remote_heartbeat/update_rate", update_rate_);

  // Setup pub
  remote_heartbeat_pub_ =
      nh_.advertise<std_msgs::Header>(remote_heartbeat_topic, 1);
  robot_heartbeat_sub_ = nh_.subscribe(
      robot_heartbeat_topic, 1, &RemoteHeartbeat::robotHeartbeatCallback, this);
}

void RemoteHeartbeat::robotHeartbeatCallback(
    const std_msgs::Header::ConstPtr& msg) {
  // Get the current time and compare to the message time
  double last_robot_heartbeat_time = msg->stamp.toSec();
  double t_now = ros::Time::now().toSec();
  double t_latency = t_now - last_robot_heartbeat_time;

  // ROS_INFO_THROTTLE(1.0,"Robot latency = %6.4fs", t_latency);

  if (abs(t_latency) >= robot_latency_threshold_warn_) {
    // ROS_WARN_THROTTLE(1.0,"Robot latency = %6.4fs which exceeds the warning
    // threshold of %6.4fs\n",
    //   t_latency, robot_latency_threshold_warn_);
  }

  if (abs(t_latency) >= robot_latency_threshold_error_) {
    // ROS_ERROR("Robot latency = %6.4fs which exceeds the maximum threshold of
    // %6.4fs, "
    //   "killing remote heartbeat\n", t_latency,
    //   robot_latency_threshold_error_);
    // throw std::runtime_error("Shutting down remote heartbeat");
  }
}

void RemoteHeartbeat::spin() {
  ros::Rate r(update_rate_);
  while (ros::ok()) {
    std_msgs::Header msg;
    msg.stamp = ros::Time::now();
    remote_heartbeat_pub_.publish(msg);

    // Enforce update rate
    ros::spinOnce();
    r.sleep();
  }
}
```

```
#include <ros/ros.h>

#include "quad_utils/remote_heartbeat.h"

int main(int argc, char** argv) {
  ros::init(argc, argv, "remote_heartbeat_node");
  ros::NodeHandle nh;

  RemoteHeartbeat remote_heartbeat(nh);
  remote_heartbeat.spin();

  return 0;
}
```

```
#include "quad_utils/terrain_map_publisher.h"

TerrainMapPublisher::TerrainMapPublisher(ros::NodeHandle nh)
    : terrain_map_(grid_map::GridMap(
          {"z", "nx", "ny", "nz", "z_filt", "nx_filt", "ny_filt", "nz_filt"})) {
  nh_ = nh;

  // Load rosparams from parameter server
  std::string terrain_map_topic, image_topic;

  nh.param<std::string>("topics/terrain_map_raw", terrain_map_topic,
                        "/terrain_map_raw");
  nh.param<std::string>("/map_frame", map_frame_, "map");
  nh.param<double>("terrain_map_publisher/update_rate", update_rate_, 10);
  nh.param<double>("terrain_map_publisher/obstacle_x", obstacle_.x, 2.0);
  nh.param<double>("terrain_map_publisher/obstacle_y", obstacle_.y, 0.0);
  nh.param<double>("terrain_map_publisher/obstacle_height", obstacle_.height,
                   0.5);
  nh.param<double>("terrain_map_publisher/obstacle_radius", obstacle_.radius,
                   1.0);
  nh.param<double>("terrain_map_publisher/step1_x", step1_.x, 4.0);
  nh.param<double>("terrain_map_publisher/step1_height", step1_.height, 0.3);
  nh.param<double>("terrain_map_publisher/step2_x", step2_.x, 4.0);
  nh.param<double>("terrain_map_publisher/step2_height", step2_.height, 0.3);
  nh.param<double>("terrain_map_publisher/resolution", resolution_, 0.2);
  nh.param<double>("terrain_map_publisher/update_rate", update_rate_, 10);
  nh.param<std::string>("terrain_map_publisher/map_data_source",
                        map_data_source_, "internal");
  nh.param<std::string>("terrain_map_publisher/terrain_type", terrain_type_,
                        "slope");
  // Setup pubs and subs
  terrain_map_pub_ =
      nh_.advertise<grid_map_msgs::GridMap>(terrain_map_topic, 1);

  // Add image subscriber if data source requests an image
  if (map_data_source_.compare("image") == 0) {
    nh_.param<std::string>("topics/image", image_topic,
                           "/image_publisher/image");
    nh_.param<double>("terrain_map_publisher/min_height", min_height_, 0.0);
    nh_.param<double>("terrain_map_publisher/max_height", max_height_, 1.0);
    image_sub_ = nh_.subscribe(image_topic, 1,
                               &TerrainMapPublisher::loadMapFromImage, this);
  }

  // Initialize the elevation layer on the terrain map
  terrain_map_.setBasicLayers(
      {"z", "nx", "ny", "nz", "z_filt", "nx_filt", "ny_filt", "nz_filt"});
}

void TerrainMapPublisher::updateParams() {
  nh_.param<double>("terrain_map_publisher/obstacle_x", obstacle_.x, 2.0);
  nh_.param<double>("terrain_map_publisher/obstacle_y", obstacle_.y, 0.0);
  nh_.param<double>("terrain_map_publisher/obstacle_height", obstacle_.height,
                    0.5);
  nh_.param<double>("terrain_map_publisher/obstacle_radius", obstacle_.radius,
                    1.0);
  nh_.param<double>("terrain_map_publisher/step1_x", step1_.x, 4.0);
  nh_.param<double>("terrain_map_publisher/step1_height", step1_.height, 0.3);
  nh_.param<double>("terrain_map_publisher/step2_x", step2_.x, 6.0);
  nh_.param<double>("terrain_map_publisher/step2_height", step2_.height, -0.3);
}

void TerrainMapPublisher::createMap() {
  // Set initial map parameters and geometry
  terrain_map_.setFrameId(map_frame_);
  terrain_map_.setGeometry(
      grid_map::Length(24.0, 12.0), resolution_,
      grid_map::Position(-0.5 * resolution_, -0.5 * resolution_));
  ROS_INFO("Created map with size %f x %f m (%i x %i cells).",
           terrain_map_.getLength().x(), terrain_map_.getLength().y(),
           terrain_map_.getSize()(0), terrain_map_.getSize()(1));
}

void TerrainMapPublisher::updateMap() {
  // Add terrain info
  for (grid_map::GridMapIterator it(terrain_map_); !it.isPastEnd(); ++it) {
    grid_map::Position position;
```

```
    terrain_map_.getPosition(*it, position);
    double x_diff = position.x() - obstacle_.x;
    double y_diff = position.y() - obstacle_.y;

    if (x_diff * x_diff + y_diff * y_diff <=
        obstacle_.radius * obstacle_.radius) {
      terrain_map_.at("z", *it) = obstacle_.height;
      terrain_map_.at("z_filt", *it) = obstacle_.height;
    } else {
      terrain_map_.at("z", *it) = 0.0;
      terrain_map_.at("z_filt", *it) = 0.0;
    }

    if (position.x() >= step1_.x) {
      terrain_map_.at("z", *it) += step1_.height;
      terrain_map_.at("z_filt", *it) += step1_.height;
    }

    if (position.x() >= step2_.x) {
      terrain_map_.at("z", *it) += step2_.height;
      terrain_map_.at("z_filt", *it) += step2_.height;
    }

    terrain_map_.at("nx", *it) = 0.0;
    terrain_map_.at("ny", *it) = 0.0;
    terrain_map_.at("nz", *it) = 1.0;

    terrain_map_.at("nx_filt", *it) = 0.0;
    terrain_map_.at("ny_filt", *it) = 0.0;
    terrain_map_.at("nz_filt", *it) = 1.0;
  }
}

std::vector<std::vector<double>> TerrainMapPublisher::loadCSV(
    std::string filename) {
  std::vector<std::vector<double>> data;
  std::ifstream inputFile(filename);
  int l = 0;

  while (inputFile) {
    l++;
    std::string s;
    if (!getline(inputFile, s)) break;
    if (s[0] != '#') {
      std::istringstream ss(s);
      std::vector<double> record;

      while (ss) {
        std::string line;
        if (!getline(ss, line, ',')) break;
        try {
          record.push_back(stod(line));
        } catch (const std::invalid_argument e) {
          std::cout << "NaN found in file " << filename << " line " << l
                    << std::endl;
          e.what();
        }
      }

      data.push_back(record);
    }
  }

  if (!inputFile.eof()) {
    std::cerr << "Could not read file " << filename << "\n";
    std::__throw_invalid_argument("File not found.");
  }

  return data;
}

void TerrainMapPublisher::loadMapFromCSV() {
  // Load in all terrain data
  std::string package_path = ros::package::getPath("quad_utils");
  std::vector<std::vector<double>> x_data =
      loadCSV(package_path + "/data/" + terrain_type_ + "/x_data.csv");
  std::vector<std::vector<double>> y_data =
```

```cpp
      loadCSV(package_path + "/data/" + terrain_type_ + "/y_data.csv");
  std::vector<std::vector<double>> z_data =
      loadCSV(package_path + "/data/" + terrain_type_ + "/z_data.csv");
  std::vector<std::vector<double>> nx_data =
      loadCSV(package_path + "/data/" + terrain_type_ + "/nx_data.csv");
  std::vector<std::vector<double>> ny_data =
      loadCSV(package_path + "/data/" + terrain_type_ + "/ny_data.csv");
  std::vector<std::vector<double>> nz_data =
      loadCSV(package_path + "/data/" + terrain_type_ + "/nz_data.csv");
  std::vector<std::vector<double>> z_data_filt =
      loadCSV(package_path + "/data/" + terrain_type_ + "/z_data_filt.csv");
  std::vector<std::vector<double>> nx_data_filt =
      loadCSV(package_path + "/data/" + terrain_type_ + "/nx_data_filt.csv");
  std::vector<std::vector<double>> ny_data_filt =
      loadCSV(package_path + "/data/" + terrain_type_ + "/ny_data_filt.csv");
  std::vector<std::vector<double>> nz_data_filt =
      loadCSV(package_path + "/data/" + terrain_type_ + "/nz_data_filt.csv");

  // Grab map length and resolution parameters, make sure resolution is square
  // (and align grid centers with data points)
  int x_size = z_data[0].size();
  int y_size = z_data.size();
  float x_res = x_data[0][1] - x_data[0][0];
  float y_res = y_data[1][0] - y_data[0][0];
  double x_length = x_data[0].back() - x_data[0].front() + x_res;
  double y_length = y_data.back()[0] - y_data.front()[0] + y_res;
  if (x_res != y_res) {
    throw std::runtime_error(
        "Map did not have square elements, make sure x and y resolution are "
        "equal.");
  }

  // Initialize the map
  terrain_map_.setFrameId(map_frame_);
  terrain_map_.setGeometry(
      grid_map::Length(x_length, y_length), x_res,
      grid_map::Position(x_data[0].front() - 0.5 * x_res + 0.5 * x_length,
                         y_data.front()[0] - 0.5 * y_res + 0.5 * y_length));
  ROS_INFO("Created map with size %f x %f m (%i x %i cells).",
           terrain_map_.getLength().x(), terrain_map_.getLength().y(),
           terrain_map_.getSize()(0), terrain_map_.getSize()(1));

  // Load in the elevation and slope data
  for (grid_map::GridMapIterator iterator(terrain_map_); !iterator.isPastEnd();
       ++iterator) {
    const grid_map::Index index(*iterator);
    grid_map::Position position;
    terrain_map_.getPosition(*iterator, position);
    terrain_map_.at("z", *iterator) =
        z_data[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
    terrain_map_.at("nx", *iterator) =
        nx_data[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
    terrain_map_.at("ny", *iterator) =
        ny_data[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
    terrain_map_.at("nz", *iterator) =
        nz_data[(y_size - 1) - index[1]][(x_size - 1) - index[0]];

    terrain_map_.at("z_filt", *iterator) =
        z_data_filt[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
    terrain_map_.at("nx_filt", *iterator) =
        nx_data_filt[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
    terrain_map_.at("ny_filt", *iterator) =
        ny_data_filt[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
    terrain_map_.at("nz_filt", *iterator) =
        nz_data_filt[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
  }
}

void TerrainMapPublisher::loadMapFromImage(const sensor_msgs::Image &msg) {
  // Initialize the map from the image message if not already done so
  if (!map_initialized_) {
    grid_map::GridMapRosConverter::initializeFromImage(msg, resolution_,
                                                       terrain_map_);
    ROS_INFO("Initialized map with size %f x %f m (%i x %i cells).",
             terrain_map_.getLength().x(), terrain_map_.getLength().y(),
             terrain_map_.getSize()(0), terrain_map_.getSize()(1));
    map_initialized_ = true;
```

```
  }

  // Add the data layers
  grid_map::GridMapRosConverter::addLayerFromImage(msg, "z", terrain_map_,
                                                   min_height_, max_height_);
  grid_map::GridMapRosConverter::addColorLayerFromImage(msg, "color",
                                                        terrain_map_);

  // Add in slope information
  for (grid_map::GridMapIterator it(terrain_map_); !it.isPastEnd(); ++it) {
    grid_map::Position position;
    terrain_map_.at("nx", *it) = 0.0;
    terrain_map_.at("ny", *it) = 0.0;
    terrain_map_.at("nz", *it) = 1.0;
  }

  // Move the map to place starting location at (0,0)
  grid_map::Position offset = {4.5, 0.0};
  terrain_map_.setPosition(offset);
}

void TerrainMapPublisher::publishMap() {
  // Set the time at which the map was published
  ros::Time time = ros::Time::now();
  terrain_map_.setTimestamp(time.toNSec());

  // Generate grid_map message, convert, and publish
  grid_map_msgs::GridMap terrain_map_msg;
  grid_map::GridMapRosConverter::toMessage(terrain_map_, terrain_map_msg);
  terrain_map_pub_.publish(terrain_map_msg);
}

void TerrainMapPublisher::spin() {
  ros::Rate r(update_rate_);

  // Either wait for an image to show up on the topic or create a map from
  // scratch
  if (map_data_source_.compare("image") == 0) {
    // Spin until image message has been received and processed
    boost::shared_ptr<sensor_msgs::Image const> shared_image;
    while ((shared_image == nullptr) && ros::ok()) {
      shared_image = ros::topic::waitForMessage<sensor_msgs::Image>(
          "/image_publisher/image", nh_);
      ros::spinOnce();
    }
  } else if (map_data_source_.compare("csv") == 0) {
    loadMapFromCSV();
  } else {
    createMap();
  }

  // Continue publishing the map at the update rate
  while (ros::ok()) {
    updateParams();

    if (map_data_source_.compare("internal") == 0) {
      updateMap();
    }

    publishMap();
    ros::spinOnce();
    r.sleep();
  }
}
```

```cpp
#include <quad_utils/ros_utils.h>

namespace quad_utils {

void updateStateHeaders(quad_msgs::RobotState &msg, ros::Time stamp,
                        std::string frame, int traj_index) {
  // Fill in the data across the messages
  msg.header.stamp = stamp;
  msg.header.frame_id = frame;
  msg.header.seq = traj_index;
  msg.body.header = msg.header;
  msg.feet.header = msg.header;
  for (int i = 0; i < msg.feet.feet.size(); i++) {
    msg.feet.feet[i].header = msg.header;
  }
  msg.joints.header = msg.header;

  msg.traj_index = traj_index;
  msg.feet.traj_index = traj_index;
  for (int i = 0; i < msg.feet.feet.size(); i++) {
    msg.feet.feet[i].traj_index = traj_index;
  }
}

void interpHeader(std_msgs::Header header_1, std_msgs::Header header_2,
                  double t_interp, std_msgs::Header &interp_header) {
  // Copy everything from the first header
  interp_header.frame_id = header_1.frame_id;
  interp_header.seq = header_1.seq;

  // Compute the correct ros::Time corresponding to t_interp
  t_interp = std::max(std::min(t_interp, 1.0), 0.0);
  ros::Duration state_duration = header_2.stamp - header_1.stamp;
  ros::Duration interp_duration =
      ros::Duration(t_interp * state_duration.toSec());
  interp_header.stamp = header_1.stamp + ros::Duration(interp_duration);
}

void interpOdometry(quad_msgs::BodyState state_1, quad_msgs::BodyState state_2,
                    double t_interp, quad_msgs::BodyState &interp_state) {
  interpHeader(state_1.header, state_2.header, t_interp, interp_state.header);

  // Interp body position
  interp_state.pose.position.x = math_utils::lerp(
      state_1.pose.position.x, state_2.pose.position.x, t_interp);
  interp_state.pose.position.y = math_utils::lerp(
      state_1.pose.position.y, state_2.pose.position.y, t_interp);
  interp_state.pose.position.z = math_utils::lerp(
      state_1.pose.position.z, state_2.pose.position.z, t_interp);

  // Interp body orientation with smath_utils::lerp
  tf2::Quaternion q_1, q_2, q_interp;
  tf2::convert(state_1.pose.orientation, q_1);
  tf2::convert(state_2.pose.orientation, q_2);
  q_interp = q_1.slerp(q_2, t_interp);
  interp_state.pose.orientation = tf2::toMsg(q_interp);

  // Interp twist
  interp_state.twist.linear.x = math_utils::lerp(
      state_1.twist.linear.x, state_2.twist.linear.x, t_interp);
  interp_state.twist.linear.y = math_utils::lerp(
      state_1.twist.linear.y, state_2.twist.linear.y, t_interp);
  interp_state.twist.linear.z = math_utils::lerp(
      state_1.twist.linear.z, state_2.twist.linear.z, t_interp);

  interp_state.twist.angular.x = math_utils::lerp(
      state_1.twist.angular.x, state_2.twist.angular.x, t_interp);
  interp_state.twist.angular.y = math_utils::lerp(
      state_1.twist.angular.y, state_2.twist.angular.y, t_interp);
  interp_state.twist.angular.z = math_utils::lerp(
      state_1.twist.angular.z, state_2.twist.angular.z, t_interp);
}

void interpJointState(sensor_msgs::JointState state_1,
                      sensor_msgs::JointState state_2, double t_interp,
                      sensor_msgs::JointState &interp_state) {
  interpHeader(state_1.header, state_2.header, t_interp, interp_state.header);
```

```cpp

  // Interp joints
  interp_state.name.resize(state_1.position.size());
  interp_state.position.resize(state_1.position.size());
  interp_state.velocity.resize(state_1.position.size());
  interp_state.effort.resize(state_1.position.size());
  for (int i = 0; i < state_1.position.size(); i++) {
    interp_state.name[i] = state_1.name[i];
    interp_state.position[i] =
        math_utils::lerp(state_1.position[i], state_2.position[i], t_interp);
    interp_state.velocity[i] =
        math_utils::lerp(state_1.velocity[i], state_2.velocity[i], t_interp);
    interp_state.effort[i] =
        math_utils::lerp(state_1.effort[i], state_2.effort[i], t_interp);
  }
}

void interpMultiFootState(quad_msgs::MultiFootState state_1,
                          quad_msgs::MultiFootState state_2, double t_interp,
                          quad_msgs::MultiFootState &interp_state) {
  interpHeader(state_1.header, state_2.header, t_interp, interp_state.header);

  // Interp foot state
  interp_state.feet.resize(state_1.feet.size());
  for (int i = 0; i < interp_state.feet.size(); i++) {
    interp_state.feet[i].header = interp_state.header;

    interp_state.feet[i].position.x = math_utils::lerp(
        state_1.feet[i].position.x, state_2.feet[i].position.x, t_interp);
    interp_state.feet[i].position.y = math_utils::lerp(
        state_1.feet[i].position.y, state_2.feet[i].position.y, t_interp);
    interp_state.feet[i].position.z = math_utils::lerp(
        state_1.feet[i].position.z, state_2.feet[i].position.z, t_interp);

    // Interp foot velocity
    interp_state.feet[i].velocity.x = math_utils::lerp(
        state_1.feet[i].velocity.x, state_2.feet[i].velocity.x, t_interp);
    interp_state.feet[i].velocity.y = math_utils::lerp(
        state_1.feet[i].velocity.y, state_2.feet[i].velocity.y, t_interp);
    interp_state.feet[i].velocity.z = math_utils::lerp(
        state_1.feet[i].velocity.z, state_2.feet[i].velocity.z, t_interp);

    // Interp foot acceleration
    interp_state.feet[i].acceleration.x =
        math_utils::lerp(state_1.feet[i].acceleration.x,
                         state_2.feet[i].acceleration.x, t_interp);
    interp_state.feet[i].acceleration.y =
        math_utils::lerp(state_1.feet[i].acceleration.y,
                         state_2.feet[i].acceleration.y, t_interp);
    interp_state.feet[i].acceleration.z =
        math_utils::lerp(state_1.feet[i].acceleration.z,
                         state_2.feet[i].acceleration.z, t_interp);

    // Set contact state to the first state
    interp_state.feet[i].contact = state_1.feet[i].contact;
  }
}

void interpGRFArray(quad_msgs::GRFArray state_1, quad_msgs::GRFArray state_2,
                    double t_interp, quad_msgs::GRFArray &interp_state) {
  interpHeader(state_1.header, state_2.header, t_interp, interp_state.header);

  // Interp grf state
  interp_state.vectors.resize(state_1.vectors.size());
  interp_state.points.resize(state_1.points.size());
  interp_state.contact_states.resize(state_1.contact_states.size());
  for (int i = 0; i < interp_state.vectors.size(); i++) {
    interp_state.vectors[i].x =
        math_utils::lerp(state_1.vectors[i].x, state_2.vectors[i].x, t_interp);
    interp_state.vectors[i].y =
        math_utils::lerp(state_1.vectors[i].y, state_2.vectors[i].y, t_interp);
    interp_state.vectors[i].z =
        math_utils::lerp(state_1.vectors[i].z, state_2.vectors[i].z, t_interp);

    interp_state.points[i].x =
        math_utils::lerp(state_1.points[i].x, state_2.points[i].x, t_interp);
    interp_state.points[i].y =
```

```
        math_utils::lerp(state_1.points[i].y, state_2.points[i].y, t_interp);
    interp_state.points[i].z =
        math_utils::lerp(state_1.points[i].z, state_2.points[i].z, t_interp);

    // Set contact state to the first state
    interp_state.contact_states[i] = state_1.contact_states[i];
  }
}

void interpRobotState(quad_msgs::RobotState state_1,
                      quad_msgs::RobotState state_2, double t_interp,
                      quad_msgs::RobotState &interp_state) {
  // Interp individual elements
  interpHeader(state_1.header, state_2.header, t_interp, interp_state.header);
  interpOdometry(state_1.body, state_2.body, t_interp, interp_state.body);
  interpJointState(state_1.joints, state_2.joints, t_interp,
                   interp_state.joints);
  interpMultiFootState(state_1.feet, state_2.feet, t_interp, interp_state.feet);
}

void interpRobotPlan(quad_msgs::RobotPlan msg, double t,
                     quad_msgs::RobotState &interp_state,
                     int &interp_primitive_id,
                     quad_msgs::GRFArray &interp_grf) {
  // Define some useful timing parameters
  ros::Time t0_ros = msg.states.front().header.stamp;
  ros::Time t_ros = t0_ros + ros::Duration(t);

  // Declare variables for interpolating between, both for input and output data
  quad_msgs::RobotState state_1, state_2;
  int primitive_id_1, primitive_id_2;
  quad_msgs::GRFArray grf_1, grf_2;

  // Find the correct index for interp (return the first index if t < 0)
  int index = 0;
  if (t >= 0) {
    for (int i = 0; i < msg.states.size() - 1; i++) {
      index = i;
      if (msg.states[i].header.stamp <= t_ros &&
          t_ros < msg.states[i + 1].header.stamp) {
        break;
      }
    }
  }

  // Extract correct states
  state_1 = msg.states[index];
  state_2 = msg.states[index + 1];
  primitive_id_1 = msg.primitive_ids[index];
  grf_1 = msg.grfs[index];
  grf_2 = msg.grfs[index + 1];

  // Compute t_interp = [0,1]
  double t1, t2;
  ros::Duration t1_ros = state_1.header.stamp - t0_ros;
  t1 = t1_ros.toSec();
  ros::Duration t2_ros = state_2.header.stamp - t0_ros;
  t2 = t2_ros.toSec();
  double t_interp = (t - t1) / (t2 - t1);

  // Compute interpolation
  interpRobotState(state_1, state_2, t_interp, interp_state);
  interp_primitive_id = primitive_id_1;
  interpGRFArray(grf_1, grf_2, t_interp, interp_grf);
}

quad_msgs::MultiFootState interpMultiFootPlanContinuous(
    quad_msgs::MultiFootPlanContinuous msg, double t) {
  // Define some useful timing parameters
  ros::Time t0_ros = msg.states.front().header.stamp;
  ros::Time t_ros = t0_ros + ros::Duration(t);

  // Declare variables for interpolating between, both for input and output data
  quad_msgs::MultiFootState state_1, state_2, interp_state;

  // Find the correct index for interp (return the first index if t < 0)
  int index = 0;
```

```
  if (t >= 0) {
    for (int i = 0; i < msg.states.size() - 1; i++) {
      index = i;
      if (msg.states[i].header.stamp <= t_ros &&
          t_ros < msg.states[i + 1].header.stamp) {
        break;
      }
    }
  }

  // Extract correct states
  state_1 = msg.states[index];
  state_2 = msg.states[index + 1];

  // Compute t_interp = [0,1]
  double t1, t2;
  ros::Duration t1_ros = state_1.header.stamp - t0_ros;
  t1 = t1_ros.toSec();
  ros::Duration t2_ros = state_2.header.stamp - t0_ros;
  t2 = t2_ros.toSec();
  double t_interp = (t - t1) / (t2 - t1);

  // Compute interpolation
  interpMultiFootState(state_1, state_2, t_interp, interp_state);

  return interp_state;
}

// quad_msgs::RobotState interpRobotStateTraj(quad_msgs::RobotStateTrajectory
// msg,
//                                            double t) {
//   // Define some useful timing parameters
//   ros::Time t0_ros = msg.states.front().header.stamp;
//   ros::Time tf_ros = msg.states.back().header.stamp;
//   ros::Duration traj_duration = tf_ros - t0_ros;

//   t = std::max(std::min(t, traj_duration.toSec()), 0.0);
//   ros::Time t_ros = t0_ros + ros::Duration(t);

//   // Declare variables for interpolating between, both for input and output
//   data quad_msgs::RobotState state_1, state_2, interp_state;

//   // Find the correct index for interp (return the first index if t < 0)
//   int index = 0;
//   if (t >= 0) {
//     for (int i = 0; i < msg.states.size() - 1; i++) {
//       index = i;
//       if (msg.states[i].header.stamp <= t_ros &&
//           t_ros < msg.states[i + 1].header.stamp) {
//         break;
//       }
//     }
//   }

//   // Extract correct states
//   state_1 = msg.states[index];
//   state_2 = msg.states[index + 1];

//   // Compute t_interp = [0,1]
//   double t1, t2;
//   ros::Duration t1_ros = state_1.header.stamp - t0_ros;
//   t1 = t1_ros.toSec();
//   ros::Duration t2_ros = state_2.header.stamp - t0_ros;
//   t2 = t2_ros.toSec();
//   double t_interp = (t - t1) / (t2 - t1);

//   // Compute interpolation
//   interpRobotState(state_1, state_2, t_interp, interp_state);

//   return interp_state;
// }

void ikRobotState(const quad_utils::QuadKD &kinematics,
                  quad_msgs::BodyState body_state,
                  quad_msgs::MultiFootState multi_foot_state,
                  sensor_msgs::JointState &joint_state) {
  joint_state.header = multi_foot_state.header;
```

```
  // If this message is empty set the joint names
  if (joint_state.name.empty()) {
    joint_state.name = {"8",  "0", "1", "9",  "2", "3",
                        "10", "4", "5", "11", "6", "7"};
  }
  joint_state.position.clear();
  joint_state.velocity.clear();
  joint_state.effort.clear();

  for (int i = 0; i < multi_foot_state.feet.size(); i++) {
    // Get foot position data
    Eigen::Vector3d foot_pos;
    foot_pos[0] = multi_foot_state.feet[i].position.x;
    foot_pos[1] = multi_foot_state.feet[i].position.y;
    foot_pos[2] = multi_foot_state.feet[i].position.z;

    // Get corresponding body plan data
    Eigen::Vector3d body_pos = {body_state.pose.position.x,
                                body_state.pose.position.y,
                                body_state.pose.position.z};

    tf2::Quaternion q;
    tf2::convert(body_state.pose.orientation, q);
    tf2::Matrix3x3 m(q);
    double roll, pitch, yaw;
    m.getRPY(roll, pitch, yaw);
    Eigen::Vector3d body_rpy = {roll, pitch, yaw};

    // Compute IK to get joint data
    Eigen::Vector3d leg_joint_state;
    kinematics.worldToFootIKWorldFrame(i, body_pos, body_rpy, foot_pos,
                                       leg_joint_state);

    // Add to the joint state vector
    joint_state.position.push_back(leg_joint_state[0]);
    joint_state.position.push_back(leg_joint_state[1]);
    joint_state.position.push_back(leg_joint_state[2]);

    joint_state.effort.push_back(0.0);
    joint_state.effort.push_back(0.0);
    joint_state.effort.push_back(0.0);
  }

  // Declare state data as Eigen vectors
  Eigen::VectorXd ref_body_state(12), ref_foot_positions(12),
      ref_foot_velocities(12);

  // Load state data
  ref_body_state = quad_utils::bodyStateMsgToEigen(body_state);
  quad_utils::multiFootStateMsgToEigen(multi_foot_state, ref_foot_positions,
                                       ref_foot_velocities);

  // Define vectors for joint positions and velocities
  Eigen::VectorXd joint_positions(12), joint_velocities(12);

  // Load joint positions
  quad_utils::vectorToEigen(joint_state.position, joint_positions);

  // Define vectors for state positions
  Eigen::VectorXd state_positions(18);

  // Load state positions
  state_positions << joint_positions, ref_body_state.head(6);

  // Compute jacobian
  Eigen::MatrixXd jacobian = Eigen::MatrixXd::Zero(12, 18);
  kinematics.getJacobianBodyAngVel(state_positions, jacobian);

  // Compute joint velocities
  joint_velocities =
      math_utils::sdlsInv(jacobian.leftCols(12)) *
      (ref_foot_velocities - jacobian.rightCols(6) * ref_body_state.tail(6));

  // Populate joint velocities message
  for (int i = 0; i < 12; ++i) {
    joint_state.velocity.push_back(joint_velocities(i));
  }
```

```cpp
}

void ikRobotState(const quad_utils::QuadKD &kinematics,
                  quad_msgs::RobotState &state) {
  ikRobotState(kinematics, state.body, state.feet, state.joints);
}

void fkRobotState(const quad_utils::QuadKD &kinematics,
                  quad_msgs::BodyState body_state,
                  sensor_msgs::JointState joint_state,
                  quad_msgs::MultiFootState &multi_foot_state) {
  multi_foot_state.header = joint_state.header;
  // If this message is empty set the joint names

  int num_feet = 4;
  multi_foot_state.feet.resize(num_feet);

  int joint_index = -1;
  for (int i = 0; i < multi_foot_state.feet.size(); i++) {
    // Get joint data for indexed leg leg
    Eigen::Vector3d leg_joint_state;

    for (int j = 0; j < 3; j++) {
      joint_index++;
      leg_joint_state[j] = joint_state.position.at(joint_index);
    }

    // Get corresponding body plan data
    Eigen::Vector3d body_pos = {body_state.pose.position.x,
                                body_state.pose.position.y,
                                body_state.pose.position.z};

    tf2::Quaternion q;
    tf2::convert(body_state.pose.orientation, q);
    tf2::Matrix3x3 m(q);
    double roll, pitch, yaw;
    m.getRPY(roll, pitch, yaw);
    Eigen::Vector3d body_rpy = {roll, pitch, yaw};

    // Compute IK to get joint data
    Eigen::Vector3d foot_pos;
    kinematics.worldToFootFKWorldFrame(i, body_pos, body_rpy, leg_joint_state,
                                       foot_pos);

    // Add to the foot position vector
    multi_foot_state.feet[i].position.x = foot_pos[0];
    multi_foot_state.feet[i].position.y = foot_pos[1];
    multi_foot_state.feet[i].position.z = foot_pos[2];

    multi_foot_state.feet[i].header = multi_foot_state.header;
  }

  // Declare state data as Eigen vectors
  Eigen::VectorXd ref_body_state(12), foot_velocities(12);

  // Load state data
  ref_body_state = quad_utils::bodyStateMsgToEigen(body_state);

  // Define vectors for joint positions and velocities
  Eigen::VectorXd joint_positions(12), joint_velocities(12);

  // Load joint positions
  quad_utils::vectorToEigen(joint_state.position, joint_positions);

  // Load joint velocities
  quad_utils::vectorToEigen(joint_state.velocity, joint_velocities);

  // Define vectors for state positions
  Eigen::VectorXd state_positions(18), state_velocities(18);

  // Load state positions
  state_positions << joint_positions, ref_body_state.head(6);

  // Load state velocities
  state_velocities << joint_velocities, ref_body_state.tail(6);

  // Compute jacobian
```

```
  Eigen::MatrixXd jacobian = Eigen::MatrixXd::Zero(12, 18);
  kinematics.getJacobianBodyAngVel(state_positions, jacobian);

  // Compute foot velocities
  foot_velocities = jacobian * state_velocities;

  // Populate foot velocities message
  for (int i = 0; i < multi_foot_state.feet.size(); ++i) {
    multi_foot_state.feet[i].velocity.x = foot_velocities(i * 3 + 0);
    multi_foot_state.feet[i].velocity.y = foot_velocities(i * 3 + 1);
    multi_foot_state.feet[i].velocity.z = foot_velocities(i * 3 + 2);
  }
}

void fkRobotState(const quad_utils::QuadKD &kinematics,
                  quad_msgs::RobotState &state) {
  fkRobotState(kinematics, state.body, state.joints, state.feet);
}

quad_msgs::BodyState eigenToBodyStateMsg(const Eigen::VectorXd &state) {
  quad_msgs::BodyState state_msg;

  // Transform from RPY to quat msg
  tf2::Quaternion quat_tf;
  geometry_msgs::Quaternion quat_msg;
  quat_tf.setRPY(state[3], state[4], state[5]);
  quat_msg = tf2::toMsg(quat_tf);

  // Load the data into the message
  state_msg.pose.position.x = state[0];
  state_msg.pose.position.y = state[1];
  state_msg.pose.position.z = state[2];
  state_msg.pose.orientation = quat_msg;

  state_msg.twist.linear.x = state[6];
  state_msg.twist.linear.y = state[7];
  state_msg.twist.linear.z = state[8];
  state_msg.twist.angular.x = state[9];
  state_msg.twist.angular.y = state[10];
  state_msg.twist.angular.z = state[11];

  return state_msg;
}

Eigen::VectorXd bodyStateMsgToEigen(const quad_msgs::BodyState &body) {
  Eigen::VectorXd state = Eigen::VectorXd::Zero(12);

  // Position
  state(0) = body.pose.position.x;
  state(1) = body.pose.position.y;
  state(2) = body.pose.position.z;

  // Orientation
  tf2::Quaternion quat;
  tf2::convert(body.pose.orientation, quat);
  double r, p, y;
  tf2::Matrix3x3 m(quat);
  m.getRPY(r, p, y);
  state(3) = r;
  state(4) = p;
  state(5) = y;

  // Linear Velocity
  state(6) = body.twist.linear.x;
  state(7) = body.twist.linear.y;
  state(8) = body.twist.linear.z;

  // Angular Velocity
  state(9) = body.twist.angular.x;
  state(10) = body.twist.angular.y;
  state(11) = body.twist.angular.z;

  return state;
}

void eigenToGRFArrayMsg(Eigen::VectorXd grf_array,
                        quad_msgs::MultiFootState multi_foot_state_msg,
```

```
                          quad_msgs::GRFArray &grf_msg) {
  grf_msg.vectors.clear();
  grf_msg.points.clear();
  grf_msg.contact_states.clear();

  for (int i = 0; i < multi_foot_state_msg.feet.size(); i++) {
    Eigen::Vector3d grf = grf_array.segment<3>(3 * i);

    geometry_msgs::Vector3 vector_msg;
    vector_msg.x = grf[0];
    vector_msg.y = grf[1];
    vector_msg.z = grf[2];
    geometry_msgs::Point point_msg;
    point_msg.x = multi_foot_state_msg.feet[i].position.x;
    point_msg.y = multi_foot_state_msg.feet[i].position.y;
    point_msg.z = multi_foot_state_msg.feet[i].position.z;

    grf_msg.vectors.push_back(vector_msg);
    grf_msg.points.push_back(point_msg);

    bool contact_state = (grf.norm() >= 1e-6);
    grf_msg.contact_states.push_back(contact_state);
  }
}

Eigen::VectorXd grfArrayMsgToEigen(const quad_msgs::GRFArray &grf_array_msg_) {
  Eigen::VectorXd grf_array(3 * grf_array_msg_.vectors.size());

  for (int i = 0; i < grf_array_msg_.vectors.size(); i++) {
    grf_array(3 * i) = grf_array_msg_.vectors[i].x;
    grf_array(3 * i + 1) = grf_array_msg_.vectors[i].y;
    grf_array(3 * i + 2) = grf_array_msg_.vectors[i].z;
  }

  return grf_array;
}

void footStateMsgToEigen(const quad_msgs::FootState &foot_state_msg,
                         Eigen::Vector3d &foot_position) {
  foot_position[0] = foot_state_msg.position.x;
  foot_position[1] = foot_state_msg.position.y;
  foot_position[2] = foot_state_msg.position.z;
}

void multiFootStateMsgToEigen(
    const quad_msgs::MultiFootState &multi_foot_state_msg,
    Eigen::VectorXd &foot_positions) {
  for (int i = 0; i < multi_foot_state_msg.feet.size(); i++) {
    foot_positions[3 * i] = multi_foot_state_msg.feet[i].position.x;
    foot_positions[3 * i + 1] = multi_foot_state_msg.feet[i].position.y;
    foot_positions[3 * i + 2] = multi_foot_state_msg.feet[i].position.z;
  }
}

void multiFootStateMsgToEigen(
    const quad_msgs::MultiFootState &multi_foot_state_msg,
    Eigen::VectorXd &foot_positions, Eigen::VectorXd &foot_velocities,
    Eigen::VectorXd &foot_acceleration) {
  multiFootStateMsgToEigen(multi_foot_state_msg, foot_positions,
                           foot_velocities);

  for (int i = 0; i < multi_foot_state_msg.feet.size(); i++) {
    foot_acceleration[3 * i] = multi_foot_state_msg.feet[i].acceleration.x;
    foot_acceleration[3 * i + 1] = multi_foot_state_msg.feet[i].acceleration.y;
    foot_acceleration[3 * i + 2] = multi_foot_state_msg.feet[i].acceleration.z;
  }
}

void multiFootStateMsgToEigen(
    const quad_msgs::MultiFootState &multi_foot_state_msg,
    Eigen::VectorXd &foot_positions, Eigen::VectorXd &foot_velocities) {
  for (int i = 0; i < multi_foot_state_msg.feet.size(); i++) {
    foot_positions[3 * i] = multi_foot_state_msg.feet[i].position.x;
    foot_positions[3 * i + 1] = multi_foot_state_msg.feet[i].position.y;
    foot_positions[3 * i + 2] = multi_foot_state_msg.feet[i].position.z;

    foot_velocities[3 * i] = multi_foot_state_msg.feet[i].velocity.x;
```

```
    foot_velocities[3 * i + 1] = multi_foot_state_msg.feet[i].velocity.y;
    foot_velocities[3 * i + 2] = multi_foot_state_msg.feet[i].velocity.z;
  }
}

void eigenToFootStateMsg(Eigen::VectorXd foot_positions,
                         Eigen::VectorXd foot_velocities,
                         Eigen::VectorXd foot_acceleration,
                         quad_msgs::FootState &foot_state_msg) {
  eigenToFootStateMsg(foot_positions, foot_velocities, foot_state_msg);

  foot_state_msg.acceleration.x = foot_acceleration[0];
  foot_state_msg.acceleration.y = foot_acceleration[1];
  foot_state_msg.acceleration.z = foot_acceleration[2];
}

void eigenToFootStateMsg(Eigen::VectorXd foot_positions,
                         Eigen::VectorXd foot_velocities,
                         quad_msgs::FootState &foot_state_msg) {
  foot_state_msg.position.x = foot_positions[0];
  foot_state_msg.position.y = foot_positions[1];
  foot_state_msg.position.z = foot_positions[2];

  foot_state_msg.velocity.x = foot_velocities[0];
  foot_state_msg.velocity.y = foot_velocities[1];
  foot_state_msg.velocity.z = foot_velocities[2];
}

void eigenToVector(const Eigen::VectorXd &eigen_vec, std::vector<double> &vec) {
  vec.resize(eigen_vec.size());
  for (int i = 0; i < eigen_vec.size(); i++) {
    vec[i] = eigen_vec(i);
  }
}

void vectorToEigen(const std::vector<double> &vec, Eigen::VectorXd &eigen_vec) {
  eigen_vec.resize(vec.size());
  for (int i = 0; i < vec.size(); i++) {
    eigen_vec(i) = vec[i];
    // std::cout << vec[i] << std::endl;
  }
}

void vector3MsgToEigen(const geometry_msgs::Vector3 &vec,
                       Eigen::Vector3d &eigen_vec) {
  eigen_vec.x() = vec.x;
  eigen_vec.y() = vec.y;
  eigen_vec.z() = vec.z;
}

void Eigen3ToVector3Msg(const Eigen::Vector3d &eigen_vec,
                        geometry_msgs::Vector3 &vec) {
  vec.x = eigen_vec.x();
  vec.y = eigen_vec.y();
  vec.z = eigen_vec.z();
}

void pointMsgToEigen(const geometry_msgs::Point &vec,
                     Eigen::Vector3d &eigen_vec) {
  eigen_vec.x() = vec.x;
  eigen_vec.y() = vec.y;
  eigen_vec.z() = vec.z;
}

void Eigen3ToPointMsg(const Eigen::Vector3d &eigen_vec,
                      geometry_msgs::Point &vec) {
  vec.x = eigen_vec.x();
  vec.y = eigen_vec.y();
  vec.z = eigen_vec.z();
}
}  // namespace quad_utils
```

```
#include "quad_utils/quad_kd.h"

using namespace quad_utils;

Eigen::IOFormat CleanFmt(4, 0, ", ", "\n", "[", "]");

QuadKD::QuadKD() { initModel(""); }

QuadKD::QuadKD(std::string ns) { initModel("/" + ns + "/"); }

void QuadKD::initModel(std::string ns) {
  std::string robot_description_string;

  if (!ros::param::get("robot_description", robot_description_string)) {
    std::cerr << "Error loading robot_description " << std::endl;
    abort();
  }

  model_ = new RigidBodyDynamics::Model();
  if (!RigidBodyDynamics::Addons::URDFReadFromString(
          robot_description_string.c_str(), model_, true)) {
    std::cerr << "Error loading model " << std::endl;
    abort();
  }

  body_name_list_ = {"toe0", "toe1", "toe2", "toe3"};

  body_id_list_.resize(4);
  for (size_t i = 0; i < body_name_list_.size(); i++) {
    body_id_list_.at(i) = model_->GetBodyId(body_name_list_.at(i).c_str());
  }

  leg_idx_list_.resize(4);
  std::iota(leg_idx_list_.begin(), leg_idx_list_.end(),
            0);  // Just initialize values for leg index.. not sure why
  // std::cout << "leg index initial val: " <<  leg_idx_list_.at(2) <<
  // std::endl;

  // defining a function here.... to check it is in ascending order??
  // interesting
  std::sort(leg_idx_list_.begin(), leg_idx_list_.end(), [&](int i, int j) {
    return body_id_list_.at(i) < body_id_list_.at(j);
  });

  // Read leg geometry from URDF
  legbase_offsets_.resize(4);
  l0_vec_.resize(4);
  std::vector<std::string> hip_name_list = {"hip0", "hip1", "hip2", "hip3"};
  std::vector<std::string> upper_name_list = {"upper0", "upper1", "upper2",
                                              "upper3"};
  std::vector<std::string> lower_name_list = {"lower0", "lower1", "lower2",
                                              "lower3"};
  std::vector<std::string> toe_name_list = {"toe0", "toe1", "toe2", "toe3"};
  RigidBodyDynamics::Math::SpatialTransform tform;
  for (size_t i = 0; i < 4; i++) {
    // From body COM to abad
    tform =
        model_->GetJointFrame(model_->GetBodyId(hip_name_list.at(i).c_str()));
    legbase_offsets_[i] = tform.r;

    // From abad to hip
    tform =
        model_->GetJointFrame(model_->GetBodyId(upper_name_list.at(i).c_str()));
    l0_vec_[i] = tform.r(1);

    // From hip to knee (we know they should be the same and the equation in IK
    // uses the magnitute of it)
    tform =
        model_->GetJointFrame(model_->GetBodyId(lower_name_list.at(i).c_str()));
    l1_ = tform.r.cwiseAbs().maxCoeff();
    knee_offset_ = tform.r;

    // From knee to toe (we know they should be the same and the equation in IK
    // uses the magnitute of it)
    tform =
        model_->GetJointFrame(model_->GetBodyId(toe_name_list.at(i).c_str()));
    l2_ = tform.r.cwiseAbs().maxCoeff();
```

```
    foot_offset_ = tform.r;
  }

  // Abad offset from legbase
  abad_offset_ = {0, 0, 0};

  g_body_legbases_.resize(4);
  for (int leg_index = 0; leg_index < 4; leg_index++) {
    // Compute transforms
    g_body_legbases_[leg_index] =
        createAffineMatrix(legbase_offsets_[leg_index],
                           Eigen::AngleAxisd(0, Eigen::Vector3d::UnitZ()));
  }

  joint_min_.resize(num_feet_);
  joint_max_.resize(num_feet_);

  std::vector<double> joint_min_front = {-0.707, -M_PI * 0.5, 0};
  std::vector<double> joint_min_back = {-0.707, -M_PI, 0};
  std::vector<double> joint_max_front = {0.707, M_PI, M_PI};
  std::vector<double> joint_max_back = {0.707, M_PI * 0.5, M_PI};

  joint_min_ = {joint_min_front, joint_min_back, joint_min_front,
                joint_min_back};
  joint_max_ = {joint_max_front, joint_max_back, joint_max_front,
                joint_max_back};
}

Eigen::Matrix4d QuadKD::createAffineMatrix(Eigen::Vector3d trans,
                                           Eigen::Vector3d rpy) const {
  Eigen::Transform<double, 3, Eigen::Affine> t;
  t = Eigen::Translation<double, 3>(trans);
  t.rotate(Eigen::AngleAxisd(rpy[2], Eigen::Vector3d::UnitZ()));
  t.rotate(Eigen::AngleAxisd(rpy[1], Eigen::Vector3d::UnitY()));
  t.rotate(Eigen::AngleAxisd(rpy[0], Eigen::Vector3d::UnitX()));

  return t.matrix();
}

Eigen::Matrix4d QuadKD::createAffineMatrix(Eigen::Vector3d trans,
                                           Eigen::AngleAxisd rot) const {
  Eigen::Transform<double, 3, Eigen::Affine> t;
  t = Eigen::Translation<double, 3>(trans);
  t.rotate(rot);

  return t.matrix();
}

double QuadKD::getJointLowerLimit(int leg_index, int joint_index) const {
  return joint_min_[leg_index][joint_index];
}

double QuadKD::getJointUpperLimit(int leg_index, int joint_index) const {
  return joint_max_[leg_index][joint_index];
}

double QuadKD::getLinkLength(int leg_index, int link_index) const {
  switch (link_index) {
    case 0:
      return l0_vec_[leg_index];
    case 1:
      return l1_;
    case 2:
      return l2_;
    default:
      throw std::runtime_error("Invalid link index");
  }
}

void QuadKD::transformBodyToWorld(Eigen::Vector3d body_pos,
                                  Eigen::Vector3d body_rpy,
                                  Eigen::Matrix4d transform_body,
                                  Eigen::Matrix4d &transform_world) const {
  // Compute transform from world to body frame
  Eigen::Matrix4d g_world_body = createAffineMatrix(body_pos, body_rpy);

  // Get the desired transform in the world frame
```

```cpp
  transform_world = g_world_body * transform_body;
}

void QuadKD::transformWorldToBody(Eigen::Vector3d body_pos,
                                  Eigen::Vector3d body_rpy,
                                  Eigen::Matrix4d transform_world,
                                  Eigen::Matrix4d &transform_body) const {
  // Compute transform from world to body frame
  Eigen::Matrix4d g_world_body = createAffineMatrix(body_pos, body_rpy);

  // Compute the desired transform in the body frame
  transform_body = g_world_body.inverse() * transform_world;
}

void QuadKD::worldToLegbaseFKWorldFrame(
    int leg_index, Eigen::Vector3d body_pos, Eigen::Vector3d body_rpy,
    Eigen::Matrix4d &g_world_legbase) const {
  // Compute transforms
  Eigen::Matrix4d g_world_body = createAffineMatrix(body_pos, body_rpy);

  // Compute transform for leg base relative to the world frame
  g_world_legbase = g_world_body * g_body_legbases_[leg_index];
}

void QuadKD::worldToLegbaseFKWorldFrame(
    int leg_index, Eigen::Vector3d body_pos, Eigen::Vector3d body_rpy,
    Eigen::Vector3d &leg_base_pos_world) const {
  Eigen::Matrix4d g_world_legbase;
  worldToLegbaseFKWorldFrame(leg_index, body_pos, body_rpy, g_world_legbase);

  leg_base_pos_world = g_world_legbase.block<3, 1>(0, 3);
}

void QuadKD::worldToNominalHipFKWorldFrame(
    int leg_index, Eigen::Vector3d body_pos, Eigen::Vector3d body_rpy,
    Eigen::Vector3d &nominal_hip_pos_world) const {
  // Compute transforms
  Eigen::Matrix4d g_world_body = createAffineMatrix(body_pos, body_rpy);
  // Compute transform from body to legbase but offset by l0
  Eigen::Matrix4d g_body_nominal_hip = g_body_legbases_[leg_index];
  g_body_nominal_hip(1, 3) += 1.0 * l0_vec_[leg_index];

  // Compute transform for offset leg base relative to the world frame
  Eigen::Matrix4d g_world_nominal_hip = g_world_body * g_body_nominal_hip;

  nominal_hip_pos_world = g_world_nominal_hip.block<3, 1>(0, 3);
}

void QuadKD::bodyToFootFKBodyFrame(int leg_index, Eigen::Vector3d joint_state,
                                   Eigen::Matrix4d &g_body_foot) const {
  if (leg_index > (legbase_offsets_.size() - 1) || leg_index < 0) {
    throw std::runtime_error("Leg index is outside valid range");
  }

  // Define hip offset
  Eigen::Vector3d hip_offset = {0, l0_vec_[leg_index], 0};

  // Initialize transforms
  Eigen::Matrix4d g_legbase_abad;
  Eigen::Matrix4d g_abad_hip;
  Eigen::Matrix4d g_hip_knee;
  Eigen::Matrix4d g_knee_foot;

  g_legbase_abad = createAffineMatrix(
      abad_offset_,
      Eigen::AngleAxisd(joint_state[0], Eigen::Vector3d::UnitX()));

  g_abad_hip = createAffineMatrix(
      hip_offset, Eigen::AngleAxisd(joint_state[1], -Eigen::Vector3d::UnitY()));

  g_hip_knee = createAffineMatrix(
      knee_offset_,
      Eigen::AngleAxisd(joint_state[2], Eigen::Vector3d::UnitY()));

  g_knee_foot = createAffineMatrix(
      foot_offset_, Eigen::AngleAxisd(0, Eigen::Vector3d::UnitY()));

```

```cpp
  // Get foot transform in world frame
  g_body_foot = g_body_legbases_[leg_index] * g_legbase_abad * g_abad_hip *
                g_hip_knee * g_knee_foot;
}

void QuadKD::bodyToFootFKBodyFrame(int leg_index, Eigen::Vector3d joint_state,
                                   Eigen::Vector3d &foot_pos_body) const {
  Eigen::Matrix4d g_body_foot;
  QuadKD::bodyToFootFKBodyFrame(leg_index, joint_state, g_body_foot);

  // Extract cartesian position of foot
  foot_pos_body = g_body_foot.block<3, 1>(0, 3);
}

void QuadKD::worldToFootFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
                                     Eigen::Vector3d body_rpy,
                                     Eigen::Vector3d joint_state,
                                     Eigen::Matrix4d &g_world_foot) const {
  if (leg_index > (legbase_offsets_.size() - 1) || leg_index < 0) {
    throw std::runtime_error("Leg index is outside valid range");
  }

  // Define hip offset
  Eigen::Vector3d hip_offset = {0, l0_vec_[leg_index], 0};

  // Initialize transforms
  Eigen::Matrix4d g_body_legbase;
  Eigen::Matrix4d g_legbase_abad;
  Eigen::Matrix4d g_abad_hip;
  Eigen::Matrix4d g_hip_knee;
  Eigen::Matrix4d g_knee_foot;

  // Compute transforms
  Eigen::Matrix4d g_world_legbase;
  worldToLegbaseFKWorldFrame(leg_index, body_pos, body_rpy, g_world_legbase);

  g_legbase_abad = createAffineMatrix(
      abad_offset_,
      Eigen::AngleAxisd(joint_state[0], Eigen::Vector3d::UnitX()));

  g_abad_hip = createAffineMatrix(
      hip_offset, Eigen::AngleAxisd(joint_state[1], -Eigen::Vector3d::UnitY()));

  g_hip_knee = createAffineMatrix(
      knee_offset_,
      Eigen::AngleAxisd(joint_state[2], Eigen::Vector3d::UnitY()));

  g_knee_foot = createAffineMatrix(
      foot_offset_, Eigen::AngleAxisd(0, Eigen::Vector3d::UnitY()));

  // Get foot transform in world frame
  g_world_foot =
      g_world_legbase * g_legbase_abad * g_abad_hip * g_hip_knee * g_knee_foot;
}

void QuadKD::worldToFootFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
                                     Eigen::Vector3d body_rpy,
                                     Eigen::Vector3d joint_state,
                                     Eigen::Vector3d &foot_pos_world) const {
  Eigen::Matrix4d g_world_foot;
  worldToFootFKWorldFrame(leg_index, body_pos, body_rpy, joint_state,
                          g_world_foot);

  // Extract cartesian position of foot
  foot_pos_world = g_world_foot.block<3, 1>(0, 3);
}

void QuadKD::worldToKneeFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
                                     Eigen::Vector3d body_rpy,
                                     Eigen::Vector3d joint_state,
                                     Eigen::Matrix4d &g_world_knee) const {
  if (leg_index > (legbase_offsets_.size() - 1) || leg_index < 0) {
    throw std::runtime_error("Leg index is outside valid range");
  }

  // Define hip offset
  Eigen::Vector3d hip_offset = {0, l0_vec_[leg_index], 0};
```

```cpp

  // Initialize transforms
  Eigen::Matrix4d g_body_legbase;
  Eigen::Matrix4d g_legbase_abad;
  Eigen::Matrix4d g_abad_hip;
  Eigen::Matrix4d g_hip_knee;

  // Compute transforms
  Eigen::Matrix4d g_world_legbase;
  worldToLegbaseFKWorldFrame(leg_index, body_pos, body_rpy, g_world_legbase);

  g_legbase_abad = createAffineMatrix(
      abad_offset_,
      Eigen::AngleAxisd(joint_state[0], Eigen::Vector3d::UnitX()));

  g_abad_hip = createAffineMatrix(
      hip_offset, Eigen::AngleAxisd(joint_state[1], -Eigen::Vector3d::UnitY()));

  g_hip_knee = createAffineMatrix(
      knee_offset_,
      Eigen::AngleAxisd(joint_state[2], Eigen::Vector3d::UnitY()));

  // Get foot transform in world frame
  g_world_knee = g_world_legbase * g_legbase_abad * g_abad_hip * g_hip_knee;
}

void QuadKD::worldToKneeFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
                                     Eigen::Vector3d body_rpy,
                                     Eigen::Vector3d joint_state,
                                     Eigen::Vector3d &knee_pos_world) const {
  Eigen::Matrix4d g_world_knee;
  worldToKneeFKWorldFrame(leg_index, body_pos, body_rpy, joint_state,
                          g_world_knee);

  // Extract cartesian position of foot
  knee_pos_world = g_world_knee.block<3, 1>(0, 3);
}

bool QuadKD::worldToFootIKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
                                     Eigen::Vector3d body_rpy,
                                     Eigen::Vector3d foot_pos_world,
                                     Eigen::Vector3d &joint_state) const {
  if (leg_index > (legbase_offsets_.size() - 1) || leg_index < 0) {
    throw std::runtime_error("Leg index is outside valid range");
  }

  // Calculate offsets
  Eigen::Vector3d legbase_offset = legbase_offsets_[leg_index];
  double l0 = l0_vec_[leg_index];

  // Initialize transforms
  Eigen::Matrix4d g_world_legbase;
  Eigen::Matrix4d g_world_foot;
  Eigen::Matrix4d g_legbase_foot;
  Eigen::Vector3d foot_pos_legbase;

  // Compute transforms
  worldToLegbaseFKWorldFrame(leg_index, body_pos, body_rpy, g_world_legbase);

  g_world_foot = createAffineMatrix(
      foot_pos_world, Eigen::AngleAxisd(0, Eigen::Vector3d::UnitY()));

  // Compute foot position relative to the leg base in cartesian coordinates
  g_legbase_foot = g_world_legbase.inverse() * g_world_foot;
  foot_pos_legbase = g_legbase_foot.block<3, 1>(0, 3);

  return legbaseToFootIKLegbaseFrame(leg_index, foot_pos_legbase, joint_state);
}

bool QuadKD::legbaseToFootIKLegbaseFrame(int leg_index,
                                         Eigen::Vector3d foot_pos_legbase,
                                         Eigen::Vector3d &joint_state) const {
  // Initialize exact bool
  bool is_exact = true;

  // Calculate offsets
  Eigen::Vector3d legbase_offset = legbase_offsets_[leg_index];
```

```
  double l0 = l0_vec_[leg_index];

  // Extract coordinates and declare joint variables
  double x = foot_pos_legbase[0];
  double y = foot_pos_legbase[1];
  double z = foot_pos_legbase[2];
  double q0;
  double q1;
  double q2;

  // Start IK, check foot pos is at least l0 away from leg base, clamp otherwise
  double temp = l0 / sqrt(z * z + y * y);
  if (abs(temp) > 1) {
    ROS_DEBUG_THROTTLE(0.5, "Foot too close, choosing closest alternative\n");
    is_exact = false;
    temp = std::max(std::min(temp, 1.0), -1.0);
  }

  // Compute both solutions of q0, use hip-above-knee if z<0 (preferred)
  // Store the inverted solution in case hip limits are exceeded
  if (z > 0) {
    q0 = -acos(temp) + atan2(z, y);
  } else {
    q0 = acos(temp) + atan2(z, y);
  }

  // Make sure abad is within joint limits, clamp otherwise
  if (q0 > joint_max_[leg_index][0] || q0 < joint_min_[leg_index][0]) {
    q0 = std::max(std::min(q0, joint_max_[leg_index][0]),
                  joint_min_[leg_index][0]);
    is_exact = false;
    ROS_DEBUG_THROTTLE(0.5, "Abad limits exceeded, clamping to %5.3f \n", q0);
  }

  // Rotate to ab-ad fixed frame
  double z_body_frame = z;
  z = -sin(q0) * y + cos(q0) * z_body_frame;

  // Check reachibility for hip
  double acos_eps = 1.0;
  double temp2 =
      (l1_ * l1_ + x * x + z * z - l2_ * l2_) / (2 * l1_ * sqrt(x * x + z * z));
  if (abs(temp2) > acos_eps) {
    ROS_DEBUG_THROTTLE(0.5,
                       "Foot location too far for hip, choosing closest"
                       " alternative \n");
    is_exact = false;
    temp2 = std::max(std::min(temp2, acos_eps), -acos_eps);
  }

  // Check reachibility for knee
  double temp3 = (l1_ * l1_ + l2_ * l2_ - x * x - z * z) / (2 * l1_ * l2_);

  if (temp3 > acos_eps || temp3 < -acos_eps) {
    ROS_DEBUG_THROTTLE(0.5,
                       "Foot location too far for knee, choosing closest"
                       " alternative \n");
    is_exact = false;

    temp3 = std::max(std::min(temp3, acos_eps), -acos_eps);
  }

  // Compute joint angles
  q1 = 0.5 * M_PI + atan2(x, -z) - acos(temp2);

  // Make sure hip is within joint limits
  if (q1 > joint_max_[leg_index][1] || q1 < joint_min_[leg_index][1]) {
    q1 = std::max(std::min(q1, joint_max_[leg_index][1]),
                  joint_min_[leg_index][1]);
    is_exact = false;
    ROS_DEBUG_THROTTLE(0.5, "Hip limits exceeded, clamping to %5.3f \n", q1);
  }

  // Compute knee val to get closest toe position in the plane
  Eigen::Vector2d knee_pos, toe_pos, toe_offset;
  knee_pos << -l1_ * cos(q1), -l1_ * sin(q1);
  toe_pos << x, z;
```

```
  toe_offset = toe_pos - knee_pos;
  q2 = atan2(-toe_offset(1), toe_offset(0)) + q1;

  // Make sure knee is within joint limits
  if (q2 > joint_max_[leg_index][2] || q2 < joint_min_[leg_index][2]) {
    q2 = std::max(std::min(q2, joint_max_[leg_index][2]),
                  joint_min_[leg_index][2]);
    is_exact = false;
    ROS_DEBUG_THROTTLE(0.5, "Knee limit exceeded, clamping to %5.3f \n", q2);
  }

  // q1 is undefined if q2=0, resolve this
  if (q2 == 0) {
    q1 = 0;
    ROS_DEBUG_THROTTLE(0.5,
                       "Hip value undefined (in singularity), setting to"
                       " %5.3f \n",
                       q1);
    is_exact = false;
  }

  if (z_body_frame - l0 * sin(q0) > 0) {
    ROS_DEBUG_THROTTLE(0.5, "IK solution is in hip-inverted region! Beware!\n");
    is_exact = false;
  }

  joint_state = {q0, q1, q2};
  return is_exact;
}

void QuadKD::getJacobianGenCoord(const Eigen::VectorXd &state,
                                 Eigen::MatrixXd &jacobian) const {
  this->getJacobianBodyAngVel(state, jacobian);

  // RBDL uses Jacobian w.r.t. floating base angular velocity in body frame,
  // which is multiplied by Jacobian to map it to Euler angle change rate here
  for (size_t i = 0; i < 4; i++) {
    Eigen::MatrixXd transform_jac(3, 3);
    transform_jac << 1, 0, -sin(state(16)), 0, cos(state(15)),
        cos(state(16)) * sin(state(15)), 0, -sin(state(15)),
        cos(state(15)) * cos(state(16));
    jacobian.block(3 * i, 15, 3, 3) =
        jacobian.block(3 * i, 15, 3, 3) * transform_jac;
  }
}

void QuadKD::getJacobianBodyAngVel(const Eigen::VectorXd &state,
                                   Eigen::MatrixXd &jacobian) const {
  assert(state.size() == 18);

  // RBDL state vector has the floating base state in the front and the joint
  // state in the back When reading from URDF, the order of the legs is 2301,
  // which should be corrected by sorting the bodyID
  Eigen::VectorXd q(19);
  q.setZero();

  q.head(3) = state.segment(12, 3);

  tf2::Quaternion quat_tf;
  quat_tf.setRPY(state(15), state(16), state(17));
  q(3) = quat_tf.getX();
  q(4) = quat_tf.getY();
  q(5) = quat_tf.getZ();

  // RBDL uses quaternion for floating base direction, but w is placed at the
  // end of the state vector
  q(18) = quat_tf.getW();

  for (size_t i = 0; i < leg_idx_list_.size(); i++) {
    q.segment(6 + 3 * i, 3) = state.segment(3 * leg_idx_list_.at(i), 3);
  }

  jacobian.setZero();

  for (size_t i = 0; i < body_id_list_.size(); i++) {
    Eigen::MatrixXd jac_block(3, 18);
    jac_block.setZero();
```

```cpp
    RigidBodyDynamics::CalcPointJacobian(*model_, q, body_id_list_.at(i),
                                         Eigen::Vector3d::Zero(), jac_block);

    for (size_t j = 0; j < 4; j++) {
      jacobian.block(3 * i, 3 * leg_idx_list_.at(j), 3, 3) =
          jac_block.block(0, 6 + 3 * j, 3, 3);
    }
    jacobian.block(3 * i, 12, 3, 6) = jac_block.block(0, 0, 3, 6);
  }
}

void QuadKD::getJacobianWorldAngVel(const Eigen::VectorXd &state,
                                    Eigen::MatrixXd &jacobian) const {
  this->getJacobianBodyAngVel(state, jacobian);

  // RBDL uses Jacobian w.r.t. floating base angular velocity in body frame,
  // which is multiplied by rotation matrix to map it to angular velocity in
  // world frame here
  for (size_t i = 0; i < 4; i++) {
    Eigen::Matrix3d rot;
    this->getRotationMatrix(state.segment(15, 3), rot);
    jacobian.block(3 * i, 15, 3, 3) = jacobian.block(3 * i, 15, 3, 3) * rot;
  }
}

void QuadKD::getRotationMatrix(const Eigen::VectorXd &rpy,
                               Eigen::Matrix3d &rot) const {
  rot = Eigen::AngleAxisd(rpy(2), Eigen::Vector3d::UnitZ()) *
        Eigen::AngleAxisd(rpy(1), Eigen::Vector3d::UnitY()) *
        Eigen::AngleAxisd(rpy(0), Eigen::Vector3d::UnitX());
}

void QuadKD::computeInverseDynamics(const Eigen::VectorXd &state_pos,
                                    const Eigen::VectorXd &state_vel,
                                    const Eigen::VectorXd &foot_acc,
                                    const Eigen::VectorXd &grf,
                                    const std::vector<int> &contact_mode,
                                    Eigen::VectorXd &tau) const {
  // Convert q, q_dot into RBDL order
  Eigen::VectorXd q(19), q_dot(18);
  q.setZero();
  q_dot.setZero();

  q.head(3) = state_pos.segment(12, 3);
  q_dot.head(3) = state_vel.segment(12, 3);

  tf2::Quaternion quat_tf;
  quat_tf.setRPY(state_pos(15), state_pos(16), state_pos(17));
  q(3) = quat_tf.getX();
  q(4) = quat_tf.getY();
  q(5) = quat_tf.getZ();

  // RBDL uses quaternion for floating base direction, but w is placed at the
  // end of the state vector
  q(18) = quat_tf.getW();

  q_dot.segment(3, 3) = state_vel.segment(15, 3);

  // std::cout << "leg idx list size: " << leg_idx_list_.size() << "\n";
  for (size_t i = 0; i < leg_idx_list_.size(); i++) {
    q.segment(6 + 3 * i, 3) = state_pos.segment(3 * leg_idx_list_.at(i), 3);
    q_dot.segment(6 + 3 * i, 3) = state_vel.segment(3 * leg_idx_list_.at(i), 3);
  }

  // Compute jacobians
  Eigen::MatrixXd jacobian = Eigen::MatrixXd::Zero(12, 18);
  jacobian.setZero();
  // std::cout << "body id list size:  " << body_id_list_.size() << "\n"; //
  // Still note sure about this body_id_list_
  for (size_t i = 0; i < body_id_list_.size(); i++) {
    // std::cout << "i: " << i << "\n";
    // std::cout << "body id list: " << body_id_list_[i] << "\n";
    Eigen::MatrixXd jac_block(3, 18);
    jac_block.setZero();
    RigidBodyDynamics::CalcPointJacobian(*model_, q, body_id_list_.at(i),
                                         Eigen::Vector3d::Zero(), jac_block);
    jacobian.block(3 * i, 0, 3, 18) = jac_block;
```

```
  }
  /*
  std::cout << "Jacobian block num coefficients: " << jacobian.size() <<
  std::endl; std::cout << "Jacobian rows: " << jacobian.rows() << " columns: "
  << jacobian.cols() << std::endl;
  */
  // Compute the equivalent force in generalized coordinates
  Eigen::VectorXd tau_stance =
      -jacobian.transpose() * grf;  // So this is for stance torque
  /*
  std::cout << "grf length: " << grf.size() << std::endl;
  std::cout << "tau_st length: " << tau_stance.size() << std::endl;
  */
  // Compute EOM
  Eigen::MatrixXd M(18, 18);
  M.setZero();
  Eigen::VectorXd N(18);
  RigidBodyDynamics::CompositeRigidBodyAlgorithm(*model_, q, M);
  RigidBodyDynamics::NonlinearEffects(*model_, q, q_dot, N);

  // Compute J_dot*q_dot
  Eigen::VectorXd foot_acc_J_dot(12);
  for (size_t i = 0; i < 4; i++) {
    foot_acc_J_dot.segment(3 * i, 3) = RigidBodyDynamics::CalcPointAcceleration(
        *model_, q, q_dot, Eigen::VectorXd::Zero(18),
        body_id_list_.at(
            i),  // body id is toes... why is it called foot_acc_J_dot
        Eigen::Vector3d::Zero());
  }

  // Compute constraint Jacobian A and A_dot*q_dot
  int constraints_num =
      3 * std::count(contact_mode.begin(), contact_mode.end(), true);
  // std::cout << "Number of constraints: " << constraints_num << std::endl;
  Eigen::MatrixXd A(constraints_num, 18);
  Eigen::VectorXd A_dotq_dot(constraints_num);
  // std::cout << "A row: " << A.rows() << "cols: " << A.cols() << std::endl;
  int constraints_count = 0;
  for (size_t i = 0; i < 4; i++) {
    if (contact_mode.at(i)) {
      A.block(3 * constraints_count, 0, 3, 18) =
          jacobian.block(3 * i, 0, 3, 18);
      A_dotq_dot.segment(3 * constraints_count, 3) =
          foot_acc_J_dot.segment(3 * i, 3);
      constraints_count++;
    }
  }

  // Compute acceleration from J*q_ddot
  Eigen::VectorXd foot_acc_q_ddot =
      foot_acc - foot_acc_J_dot;  // what does this mean... foot acceleration
                                  // relative to global???
  // std::cout << "foot_acc_q_ddot: " << foot_acc_q_ddot.size() << std::endl;

  // Compuate damped jacobian inverser
  Eigen::MatrixXd jacobian_inv =
      math_utils::sdlsInv(jacobian.block(0, 6, 12, 12));

  // std::cout << "Jacobian inverse rows: " << jacobian_inv.rows() << " cols: "
  // << jacobian_inv.cols() << std::endl;

  // In the EOM, we know M, N, tau_grf, and a = J_b*q_ddot_b + J_l*q_ddot_l, we
  // need to solve q_ddot_b and tau_swing
  Eigen::MatrixXd blk_mat =
      Eigen::MatrixXd::Zero(18 + constraints_num, 18 + constraints_num);
  blk_mat.block(0, 0, 6, 6) =
      -M.block(0, 0, 6, 6) +
      M.block(0, 6, 6, 12) * jacobian_inv * jacobian.block(0, 0, 12, 6);
  blk_mat.block(6, 0, 12, 6) =
      -M.block(6, 0, 12, 6) +
      M.block(6, 6, 12, 12) * jacobian_inv * jacobian.block(0, 0, 12, 6);
  for (size_t i = 0; i < 4; i++) {
    if (!contact_mode.at(leg_idx_list_.at(i))) {
      blk_mat.block(3 * i + 6, 3 * i + 6, 3, 3).diagonal().fill(1);
    }
  }
  blk_mat.block(0, 18, 18, constraints_num) = -A.transpose();
```

```
  blk_mat.block(18, 0, constraints_num, 6) =
      -A.leftCols(6) +
      A.rightCols(12) * jacobian_inv * jacobian.block(0, 0, 12, 6);

  // Perform inverse dynamics
  Eigen::VectorXd tau_swing(12), blk_sol(18 + constraints_num),
      blk_vec(18 + constraints_num);
  blk_vec.segment(0, 6) << N.segment(0, 6) + M.block(0, 6, 6, 12) *
                                               jacobian_inv * foot_acc_q_ddot;
  blk_vec.segment(6, 12) << N.segment(6, 12) +
                              M.block(6, 6, 12, 12) * jacobian_inv *
                                  foot_acc_q_ddot -
                              tau_stance.segment(6, 12);
  blk_vec.segment(18, constraints_num)
      << A_dotq_dot + A.leftCols(12) * jacobian_inv * foot_acc_q_ddot;
  blk_sol =
      blk_mat.bdcSvd(Eigen::ComputeThinU | Eigen::ComputeThinV).solve(blk_vec);
  tau_swing = blk_sol.segment(6, 12);

  // Convert the order back
  for (size_t i = 0; i < 4; i++) {
    if (contact_mode.at(leg_idx_list_.at(i))) {
      tau.segment(3 * leg_idx_list_.at(i), 3) =
          tau_stance.segment(6 + 3 * i, 3);
    } else {
      tau.segment(3 * leg_idx_list_.at(i), 3) = tau_swing.segment(3 * i, 3);
      // tau.segment(3 * leg_idx_list_.at(i), 3) = Eigen::VectorXd::Zero(3);
    }
  }

  // Check inf or nan
  if (!(tau.array() == tau.array()).all() ||
      !((tau - tau).array() == (tau - tau).array()).all()) {
    tau.setZero();
  }
}

bool QuadKD::convertCentroidalToFullBody(const Eigen::VectorXd &body_state,
                                         const Eigen::VectorXd &foot_positions,
                                         const Eigen::VectorXd &foot_velocities,
                                         const Eigen::VectorXd &grfs,
                                         Eigen::VectorXd &joint_positions,
                                         Eigen::VectorXd &joint_velocities,
                                         Eigen::VectorXd &torques) {
  // Assume the conversion is exact unless a check below fails
  bool is_exact = true;

  // Extract kinematic quantities
  Eigen::Vector3d body_pos = body_state.segment<3>(0);
  Eigen::Vector3d body_rpy = body_state.segment<3>(3);

  auto t_start = std::chrono::steady_clock::now();
  // Perform IK for each leg
  for (int i = 0; i < num_feet_; i++) {
    Eigen::Vector3d leg_joint_state;
    Eigen::Vector3d foot_pos = foot_positions.segment<3>(3 * i);
    is_exact = is_exact && worldToFootIKWorldFrame(i, body_pos, body_rpy,
                                                   foot_pos, leg_joint_state);
    joint_positions.segment<3>(3 * i) = leg_joint_state;
  }

  auto t_ik = std::chrono::steady_clock::now();

  // Load state positions
  Eigen::VectorXd state_positions(18), state_velocities(18);
  state_positions << joint_positions, body_pos, body_rpy;

  // Compute jacobian
  Eigen::MatrixXd jacobian = Eigen::MatrixXd::Zero(12, 18);
  getJacobianBodyAngVel(state_positions, jacobian);

  auto t_jacob = std::chrono::steady_clock::now();

  // Compute joint velocities
  joint_velocities = jacobian.leftCols(12).colPivHouseholderQr().solve(
      foot_velocities - jacobian.rightCols(6) * body_state.tail(6));
  state_velocities << joint_velocities, body_state.tail(6);
```

```

  auto t_ik_vel = std::chrono::steady_clock::now();

  torques = -jacobian.leftCols(12).transpose() * grfs;

  // computeInverseDynamics(state_positions, state_velocities, foot_acc, grfs,
  // contact_mode, torques);

  auto t_id = std::chrono::steady_clock::now();

  std::chrono::duration<double> t_diff_ik =
      std::chrono::duration_cast<std::chrono::duration<double>>(t_ik - t_start);
  std::chrono::duration<double> t_diff_jacob =
      std::chrono::duration_cast<std::chrono::duration<double>>(t_jacob - t_ik);
  std::chrono::duration<double> t_diff_ik_vel =
      std::chrono::duration_cast<std::chrono::duration<double>>(t_ik_vel -
                                                                t_jacob);
  std::chrono::duration<double> t_diff_id =
      std::chrono::duration_cast<std::chrono::duration<double>>(t_id -
                                                                t_ik_vel);

  // std::cout << "t_diff_ik = " << t_diff_ik.count() << std::endl;
  // std::cout << "t_diff_jacob = " << t_diff_jacob.count() << std::endl;
  // std::cout << "t_diff_ik_vel = " << t_diff_ik_vel.count() << std::endl;
  // std::cout << "t_diff_id = " << t_diff_id.count() << std::endl;

  return is_exact;
}

bool QuadKD::applyMotorModel(const Eigen::VectorXd &torques,
                             Eigen::VectorXd &constrained_torques) {
  // Constrain torques to max values
  constrained_torques.resize(torques.size());
  constrained_torques = torques.cwiseMax(-tau_max_).cwiseMin(tau_max_);

  // Check if torques was modified
  return constrained_torques.isApprox(torques);
}

bool QuadKD::applyMotorModel(const Eigen::VectorXd &joint_torques,
                             const Eigen::VectorXd &joint_velocities,
                             Eigen::VectorXd &constrained_joint_torques) {
  // Constrain torques to max values
  Eigen::VectorXd constraint_violation(joint_torques.size());
  constrained_joint_torques.resize(joint_torques.size());
  constrained_joint_torques =
      joint_torques.cwiseMax(-tau_max_).cwiseMin(tau_max_);

  // Apply linear motor model
  Eigen::VectorXd emf = joint_velocities.cwiseProduct(mm_slope_);
  constrained_joint_torques =
      constrained_joint_torques.cwiseMax(-tau_max_ - emf)
          .cwiseMin(tau_max_ - emf);

  // Check if torques were modified
  return constrained_joint_torques.isApprox(joint_torques);
}

bool QuadKD::isValidFullState(const Eigen::VectorXd &body_state,
                              const Eigen::VectorXd &joint_state,
                              const Eigen::VectorXd &joint_torques,
                              const grid_map::GridMap &terrain,
                              Eigen::VectorXd &state_violation,
                              Eigen::VectorXd &control_violation) {
  // Check state constraints
  // Kinematics
  state_violation.setZero(num_feet_);
  for (int i = 0; i < num_feet_; i++) {
    Eigen::Vector3d knee_pos_world;
    worldToKneeFKWorldFrame(i, body_state.segment<3>(0),
                            body_state.segment<3>(3),
                            joint_state.segment<3>(3 * i), knee_pos_world);
    state_violation[i] = getGroundClearance(knee_pos_world, terrain);
  }
  bool state_valid = (state_violation.array() >= 0).all();

  // Check control constraints
```

```
  // Motor model
  Eigen::VectorXd constrained_joint_torques(12);
  bool control_valid = applyMotorModel(joint_torques, joint_state.tail(12),
                                       constrained_joint_torques);
  control_violation.setZero(joint_torques.size());
  control_violation = -(constrained_joint_torques - joint_torques).cwiseAbs();

  // Only valid if each subcheck is valid
  return (state_valid && control_valid);
}

bool QuadKD::isValidCentroidalState(
    const Eigen::VectorXd &body_state, const Eigen::VectorXd &foot_positions,
    const Eigen::VectorXd &foot_velocities, const Eigen::VectorXd &grfs,
    const grid_map::GridMap &terrain, Eigen::VectorXd &joint_positions,
    Eigen::VectorXd &joint_velocities, Eigen::VectorXd &joint_torques,
    Eigen::VectorXd &state_violation, Eigen::VectorXd &control_violation) {
  // Convert to full
  bool is_exact = convertCentroidalToFullBody(
      body_state, foot_positions, foot_velocities, grfs, joint_positions,
      joint_velocities, joint_torques);

  Eigen::VectorXd joint_state(24);
  joint_state << joint_positions, joint_velocities;
  bool is_valid = isValidFullState(body_state, joint_state, joint_torques,
                                   terrain, state_violation, control_violation);

  return (is_exact && is_valid);
}
```

```cpp
#include "quad_utils/rviz_interface.h"

RVizInterface::RVizInterface(ros::NodeHandle nh) {
  nh_ = nh;

  // Load rosparams from parameter server
  std::string global_plan_topic, local_plan_topic, discrete_global_plan_topic,
      foot_plan_discrete_topic, foot_plan_continuous_topic,
      state_estimate_topic, ground_truth_state_topic, trajectory_state_topic,
      grf_topic;

  // Load topic names from parameter server
  quad_utils::loadROSParam(nh_, "topics/global_plan", global_plan_topic);
  quad_utils::loadROSParam(nh_, "topics/local_plan", local_plan_topic);
  quad_utils::loadROSParam(nh_, "topics/control/grfs", grf_topic);
  quad_utils::loadROSParam(nh_, "topics/global_plan_discrete",
                           discrete_global_plan_topic);
  quad_utils::loadROSParam(nh_, "topics/foot_plan_discrete",
                           foot_plan_discrete_topic);
  quad_utils::loadROSParam(nh_, "topics/foot_plan_continuous",
                           foot_plan_continuous_topic);
  quad_utils::loadROSParam(nh_, "topics/state/estimate", state_estimate_topic);
  quad_utils::loadROSParam(nh_, "topics/state/ground_truth",
                           ground_truth_state_topic);
  quad_utils::loadROSParam(nh_, "topics/state/trajectory",
                           trajectory_state_topic);
  quad_utils::loadROSParamDefault(nh_, "tf_prefix", tf_prefix_,
                                  std::string(""));

  std::string global_plan_viz_topic, local_plan_viz_topic,
      local_plan_ori_viz_topic, global_plan_grf_viz_topic,
      local_plan_grf_viz_topic, current_grf_viz_topic,
      discrete_body_plan_viz_topic, foot_plan_discrete_viz_topic,
      estimate_joint_states_viz_topic, ground_truth_joint_states_viz_topic,
      trajectory_joint_states_viz_topic, state_estimate_trace_viz_topic,
      ground_truth_trace_viz_topic, trajectory_state_trace_viz_topic;

  quad_utils::loadROSParam(nh_, "topics/visualization/global_plan",
                           global_plan_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/local_plan",
                           local_plan_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/local_plan_ori",
                           local_plan_ori_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/global_plan_grf",
                           global_plan_grf_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/local_plan_grf",
                           local_plan_grf_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/current_grf",
                           current_grf_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/global_plan_discrete",
                           discrete_body_plan_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/foot_plan_discrete",
                           foot_plan_discrete_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/joint_states/estimate",
                           estimate_joint_states_viz_topic);
  quad_utils::loadROSParam(nh_,
                           "topics/visualization/joint_states/ground_truth",
                           ground_truth_joint_states_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/joint_states/trajectory",
                           trajectory_joint_states_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/state/estimate_trace",
                           state_estimate_trace_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/state/ground_truth_trace",
                           ground_truth_trace_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/state/trajectory_trace",
                           trajectory_state_trace_viz_topic);

  // Setup rviz_interface parameters
  quad_utils::loadROSParam(nh_, "/map_frame", map_frame_);
  quad_utils::loadROSParam(nh_, "/rviz_interface/update_rate", update_rate_);
  quad_utils::loadROSParam(nh_, "/rviz_interface/colors/front_left",
                           front_left_color_);
  quad_utils::loadROSParam(nh_, "/rviz_interface/colors/back_left",
                           back_left_color_);
  quad_utils::loadROSParam(nh_, "/rviz_interface/colors/front_right",
                           front_right_color_);
  quad_utils::loadROSParam(nh_, "/rviz_interface/colors/back_right",
```

```cpp
                           back_right_color_);
  quad_utils::loadROSParam(nh_, "/rviz_interface/colors/net_grf",
                           net_grf_color_);
  quad_utils::loadROSParam(nh_, "/rviz_interface/colors/individual_grf",
                           individual_grf_color_);

  double period, dt;
  quad_utils::loadROSParam(nh_, "/local_footstep_planner/period", period);
  quad_utils::loadROSParam(nh_, "/local_planner/timestep", dt);
  orientation_subsample_interval_ = int(period / dt);

  // Setup plan subs
  global_plan_sub_ = nh_.subscribe<quad_msgs::RobotPlan>(
      global_plan_topic, 1,
      boost::bind(&RVizInterface::robotPlanCallback, this, _1, GLOBAL));
  local_plan_sub_ = nh_.subscribe<quad_msgs::RobotPlan>(
      local_plan_topic, 1,
      boost::bind(&RVizInterface::robotPlanCallback, this, _1, LOCAL));
  grf_sub_ = nh_.subscribe(grf_topic, 1, &RVizInterface::grfCallback, this);
  foot_plan_discrete_sub_ =
      nh_.subscribe(foot_plan_discrete_topic, 1,
                    &RVizInterface::footPlanDiscreteCallback, this);
  foot_plan_continuous_sub_ =
      nh_.subscribe(foot_plan_continuous_topic, 1,
                    &RVizInterface::footPlanContinuousCallback, this);

  // Setup plan visual pubs
  global_plan_viz_pub_ =
      nh_.advertise<visualization_msgs::Marker>(global_plan_viz_topic, 1);
  local_plan_viz_pub_ =
      nh_.advertise<visualization_msgs::Marker>(local_plan_viz_topic, 1);
  global_plan_grf_viz_pub_ = nh_.advertise<visualization_msgs::MarkerArray>(
      global_plan_grf_viz_topic, 1);
  local_plan_grf_viz_pub_ = nh_.advertise<visualization_msgs::MarkerArray>(
      local_plan_grf_viz_topic, 1);
  current_grf_viz_pub_ =
      nh_.advertise<visualization_msgs::MarkerArray>(current_grf_viz_topic, 1);
  discrete_body_plan_viz_pub_ = nh_.advertise<visualization_msgs::Marker>(
      discrete_body_plan_viz_topic, 1);
  foot_plan_discrete_viz_pub_ = nh_.advertise<visualization_msgs::Marker>(
      foot_plan_discrete_viz_topic, 1);
  local_plan_ori_viz_pub_ =
      nh_.advertise<geometry_msgs::PoseArray>(local_plan_ori_viz_topic, 1);

  // Setup publishers for state traces
  state_estimate_trace_pub_ = nh_.advertise<visualization_msgs::Marker>(
      state_estimate_trace_viz_topic, 1);
  ground_truth_state_trace_pub_ = nh_.advertise<visualization_msgs::Marker>(
      ground_truth_trace_viz_topic, 1);
  trajectory_state_trace_pub_ = nh_.advertise<visualization_msgs::Marker>(
      trajectory_state_trace_viz_topic, 1);

  // Setup state subs to call the same callback but with pub ID included
  state_estimate_sub_ = nh_.subscribe<quad_msgs::RobotState>(
      state_estimate_topic, 1,
      boost::bind(&RVizInterface::robotStateCallback, this, _1, ESTIMATE));
  ground_truth_state_sub_ = nh_.subscribe<quad_msgs::RobotState>(
      ground_truth_state_topic, 1,
      boost::bind(&RVizInterface::robotStateCallback, this, _1, GROUND_TRUTH));
  trajectory_state_sub_ = nh_.subscribe<quad_msgs::RobotState>(
      trajectory_state_topic, 1,
      boost::bind(&RVizInterface::robotStateCallback, this, _1, TRAJECTORY));

  // Setup state visual pubs
  estimate_joint_states_viz_pub_ = nh_.advertise<sensor_msgs::JointState>(
      estimate_joint_states_viz_topic, 1);
  ground_truth_joint_states_viz_pub_ = nh_.advertise<sensor_msgs::JointState>(
      ground_truth_joint_states_viz_topic, 1);
  trajectory_joint_states_viz_pub_ = nh_.advertise<sensor_msgs::JointState>(
      trajectory_joint_states_viz_topic, 1);

  std::string foot_0_plan_continuous_viz_topic,
      foot_1_plan_continuous_viz_topic, foot_2_plan_continuous_viz_topic,
      foot_3_plan_continuous_viz_topic;

  quad_utils::loadROSParam(nh_, "topics/visualization/foot_0_plan_continuous",
                           foot_0_plan_continuous_viz_topic);
```

```
  quad_utils::loadROSParam(nh_, "topics/visualization/foot_1_plan_continuous",
                           foot_1_plan_continuous_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/foot_2_plan_continuous",
                           foot_2_plan_continuous_viz_topic);
  quad_utils::loadROSParam(nh_, "topics/visualization/foot_3_plan_continuous",
                           foot_3_plan_continuous_viz_topic);

  foot_0_plan_continuous_viz_pub_ =
      nh_.advertise<nav_msgs::Path>(foot_0_plan_continuous_viz_topic, 1);
  foot_1_plan_continuous_viz_pub_ =
      nh_.advertise<nav_msgs::Path>(foot_1_plan_continuous_viz_topic, 1);
  foot_2_plan_continuous_viz_pub_ =
      nh_.advertise<nav_msgs::Path>(foot_2_plan_continuous_viz_topic, 1);
  foot_3_plan_continuous_viz_pub_ =
      nh_.advertise<nav_msgs::Path>(foot_3_plan_continuous_viz_topic, 1);

  // Initialize Path message to visualize body plan
  state_estimate_trace_msg_.action = visualization_msgs::Marker::ADD;
  state_estimate_trace_msg_.pose.orientation.w = 1;
  state_estimate_trace_msg_.type = visualization_msgs::Marker::LINE_STRIP;
  state_estimate_trace_msg_.scale.x = 0.02;
  state_estimate_trace_msg_.header.frame_id = map_frame_;
  geometry_msgs::Point dummy_point;
  state_estimate_trace_msg_.points.push_back(dummy_point);
  ground_truth_state_trace_msg_ = state_estimate_trace_msg_;
  trajectory_state_trace_msg_ = state_estimate_trace_msg_;

  // Define visual properties for traces
  state_estimate_trace_msg_.id = 5;
  state_estimate_trace_msg_.color.a = 1.0;
  state_estimate_trace_msg_.color.r = (float)front_left_color_[0] / 255.0;
  state_estimate_trace_msg_.color.g = (float)front_left_color_[1] / 255.0;
  state_estimate_trace_msg_.color.b = (float)front_left_color_[2] / 255.0;

  ground_truth_state_trace_msg_.id = 6;
  ground_truth_state_trace_msg_.color.a = 1.0;
  ground_truth_state_trace_msg_.color.r = (float)back_left_color_[0] / 255.0;
  ground_truth_state_trace_msg_.color.g = (float)back_left_color_[1] / 255.0;
  ground_truth_state_trace_msg_.color.b = (float)back_left_color_[2] / 255.0;

  trajectory_state_trace_msg_.id = 7;
  trajectory_state_trace_msg_.color.a = 1.0;
  trajectory_state_trace_msg_.color.r = (float)front_right_color_[0] / 255.0;
  trajectory_state_trace_msg_.color.g = (float)front_right_color_[1] / 255.0;
  trajectory_state_trace_msg_.color.b = (float)front_right_color_[2] / 255.0;
}

void RVizInterface::robotPlanCallback(const quad_msgs::RobotPlan::ConstPtr &msg,
                                      const int pub_id) {
  // Initialize Path message to visualize body plan
  visualization_msgs::Marker body_plan_viz;
  body_plan_viz.header = msg->header;
  body_plan_viz.action = visualization_msgs::Marker::ADD;
  body_plan_viz.pose.orientation.w = 1;
  body_plan_viz.id = 5;
  body_plan_viz.type = visualization_msgs::Marker::LINE_STRIP;
  body_plan_viz.scale.x = 0.03;

  // Construct MarkerArray for body plan orientation
  geometry_msgs::PoseArray body_plan_ori_viz;
  body_plan_ori_viz.header = msg->header;

  // Loop through the BodyPlan message to get the state info
  int length = msg->states.size();
  for (int i = 0; i < length; i++) {
    // Load in the pose data directly from the Odometry message
    geometry_msgs::PoseStamped pose_stamped;
    pose_stamped.header = msg->states[i].header;
    pose_stamped.pose = msg->states[i].body.pose;

    std_msgs::ColorRGBA color;
    color.a = 1;
    if (pub_id == LOCAL) {
      color.g = 1.0;
    } else {
      if (msg->primitive_ids[i] == FLIGHT) {
        color.r = (float)back_left_color_[0] / 255.0;
```

```cpp
        color.g = (float)back_left_color_[1] / 255.0;
        color.b = (float)back_left_color_[2] / 255.0;
      } else if (msg->primitive_ids[i] == LEAP_STANCE ||
                 msg->primitive_ids[i] == LAND_STANCE) {
        color.r = (float)front_right_color_[0] / 255.0;
        color.g = (float)front_right_color_[1] / 255.0;
        color.b = (float)front_right_color_[2] / 255.0;
      } else if (msg->primitive_ids[i] == CONNECT) {
        color.r = (float)front_left_color_[0] / 255.0;
        color.g = (float)front_left_color_[1] / 255.0;
        color.b = (float)front_left_color_[2] / 255.0;
      } else {
        ROS_WARN_THROTTLE(1, "Invalid primitive ID received in RViz interface");
      }
    }
    body_plan_viz.colors.push_back(color);
    body_plan_viz.points.push_back(msg->states[i].body.pose.position);

    // Add poses to the orientation message
    if (i % orientation_subsample_interval_ == 0) {
      body_plan_ori_viz.poses.push_back(pose_stamped.pose);
    }
  }

  // Publish the full path
  if (pub_id == GLOBAL) {
    global_plan_viz_pub_.publish(body_plan_viz);
  } else if (pub_id == LOCAL) {
    local_plan_viz_pub_.publish(body_plan_viz);
    local_plan_ori_viz_pub_.publish(body_plan_ori_viz);
  }

  // Construct MarkerArray and Marker message for GRFs
  visualization_msgs::MarkerArray grfs_viz_msg;
  visualization_msgs::Marker marker;

  // Initialize the headers and types
  marker.header = msg->header;
  marker.type = visualization_msgs::Marker::ARROW;

  // Define the shape of the discrete states
  double arrow_diameter = 0.01;
  marker.scale.x = arrow_diameter;
  marker.scale.y = 4 * arrow_diameter;
  marker.color.g = 0.733f;
  marker.pose.orientation.w = 1.0;

  for (int i = 0; i < length; i++) {
    for (int j = 0; j < msg->grfs[i].vectors.size(); j++) {
      // Reset the marker message
      marker.points.clear();
      marker.color.a = 1.0;
      marker.id = i * msg->grfs[i].vectors.size() + j;

      if (msg->grfs[i].vectors.size() > 1) {
        marker.color.r = (float)individual_grf_color_[0] / 255.0;
        marker.color.g = (float)individual_grf_color_[1] / 255.0;
        marker.color.b = (float)individual_grf_color_[2] / 255.0;
      } else {
        marker.color.r = (float)net_grf_color_[0] / 255.0;
        marker.color.g = (float)net_grf_color_[1] / 255.0;
        marker.color.b = (float)net_grf_color_[2] / 255.0;
      }

      // Define point messages for the base and tip of each GRF arrow
      geometry_msgs::Point p_base, p_tip;
      p_base = msg->grfs[i].points[j];

      /// Define the endpoint of the GRF arrow
      double grf_length_scale = 0.002;
      p_tip.x = p_base.x + grf_length_scale * msg->grfs[i].vectors[j].x;
      p_tip.y = p_base.y + grf_length_scale * msg->grfs[i].vectors[j].y;
      p_tip.z = p_base.z + grf_length_scale * msg->grfs[i].vectors[j].z;

      // if GRF = 0, set alpha to zero
      if (msg->grfs[i].contact_states[j] == false) {
        marker.color.a = 0.0;
```

```
      }

      // Add the points to the marker and add the marker to the array
      marker.points.push_back(p_base);
      marker.points.push_back(p_tip);
      grfs_viz_msg.markers.push_back(marker);
    }
  }

  // Publish grfs
  if (pub_id == GLOBAL) {
    global_plan_grf_viz_pub_.publish(grfs_viz_msg);
  } else if (pub_id == LOCAL) {
    local_plan_grf_viz_pub_.publish(grfs_viz_msg);
  }
}

void RVizInterface::grfCallback(const quad_msgs::GRFArray::ConstPtr &msg) {
  if (msg->vectors.empty()) {
    return;
  }

  // Construct MarkerArray and Marker message for GRFs
  visualization_msgs::MarkerArray grfs_viz_msg;
  visualization_msgs::Marker marker;

  // Initialize the headers and types
  marker.header = msg->header;
  marker.type = visualization_msgs::Marker::ARROW;

  // Define the shape of the discrete states
  double arrow_diameter = 0.01;
  marker.scale.x = arrow_diameter;
  marker.scale.y = 4 * arrow_diameter;
  marker.color.g = 0.733f;
  marker.pose.orientation.w = 1.0;

  for (int i = 0; i < msg->vectors.size(); i++) {
    // Reset the marker message
    marker.points.clear();
    marker.color.a = 1.0;
    marker.id = i;

    marker.color.r = (float)individual_grf_color_[0] / 255.0;
    marker.color.g = (float)individual_grf_color_[1] / 255.0;
    marker.color.b = (float)individual_grf_color_[2] / 255.0;

    // Define point messages for the base and tip of each GRF arrow
    geometry_msgs::Point p_base, p_tip;
    p_base = msg->points[i];

    /// Define the endpoint of the GRF arrow
    double grf_length_scale = 0.002;
    p_tip.x = p_base.x + grf_length_scale * msg->vectors[i].x;
    p_tip.y = p_base.y + grf_length_scale * msg->vectors[i].y;
    p_tip.z = p_base.z + grf_length_scale * msg->vectors[i].z;

    // if GRF = 0, set alpha to zero
    if (msg->contact_states[i] == false) {
      marker.color.a = 0.0;
    }

    // Add the points to the marker and add the marker to the array
    marker.points.push_back(p_base);
    marker.points.push_back(p_tip);
    grfs_viz_msg.markers.push_back(marker);
  }

  current_grf_viz_pub_.publish(grfs_viz_msg);
}

void RVizInterface::discreteBodyPlanCallback(
    const quad_msgs::RobotPlan::ConstPtr &msg) {
  // Construct Marker message
  visualization_msgs::Marker discrete_body_plan;

  // Initialize the headers and types
```

```
  discrete_body_plan.header = msg->header;
  discrete_body_plan.id = 0;
  discrete_body_plan.type = visualization_msgs::Marker::POINTS;

  // Define the shape of the discrete states
  double scale = 0.2;
  discrete_body_plan.scale.x = scale;
  discrete_body_plan.scale.y = scale;
  discrete_body_plan.color.r = 0.733f;
  discrete_body_plan.color.a = 1.0;

  // Loop through the discrete states
  int length = msg->states.size();
  for (int i = 0; i < length; i++) {
    geometry_msgs::Point p;
    p.x = msg->states[i].body.pose.position.x;
    p.y = msg->states[i].body.pose.position.y;
    p.z = msg->states[i].body.pose.position.z;
    discrete_body_plan.points.push_back(p);
  }

  // Publish both interpolated body plan and discrete states
  discrete_body_plan_viz_pub_.publish(discrete_body_plan);
}

void RVizInterface::footPlanDiscreteCallback(
    const quad_msgs::MultiFootPlanDiscrete::ConstPtr &msg) {
  // Initialize Marker message to visualize footstep plan as points
  visualization_msgs::Marker points;
  points.header = msg->header;
  points.action = visualization_msgs::Marker::ADD;
  points.pose.orientation.w = 1.0;
  points.id = 0;
  points.type = visualization_msgs::Marker::SPHERE_LIST;

  // POINTS markers use x and y scale for width/height respectively
  points.scale.x = 0.05;
  points.scale.y = 0.05;
  points.scale.z = 0.05;

  // Loop through each foot
  int num_feet = msg->feet.size();
  for (int i = 0; i < num_feet; ++i) {
    // Loop through footstep in the plan
    int num_steps = msg->feet[i].footholds.size();
    for (int j = 0; j < num_steps; ++j) {
      // Create point message from FootstepPlan message, adjust height
      geometry_msgs::Point p;
      p.x = msg->feet[i].footholds[j].position.x;
      p.y = msg->feet[i].footholds[j].position.y;
      p.z = msg->feet[i].footholds[j].position.z;

      // Set the color of each marker (green for front, blue for back)
      std_msgs::ColorRGBA color;
      color.a = 1.0;
      if (i == 0) {
        color.r = (float)front_left_color_[0] / 255.0;
        color.g = (float)front_left_color_[1] / 255.0;
        color.b = (float)front_left_color_[2] / 255.0;
      } else if (i == 1) {
        color.r = (float)back_left_color_[0] / 255.0;
        color.g = (float)back_left_color_[1] / 255.0;
        color.b = (float)back_left_color_[2] / 255.0;
      } else if (i == 2) {
        color.r = (float)front_right_color_[0] / 255.0;
        color.g = (float)front_right_color_[1] / 255.0;
        color.b = (float)front_right_color_[2] / 255.0;
      } else if (i == 3) {
        color.r = (float)back_right_color_[0] / 255.0;
        color.g = (float)back_right_color_[1] / 255.0;
        color.b = (float)back_right_color_[2] / 255.0;
      }

      // Add to the Marker message
      points.colors.push_back(color);
      points.points.push_back(p);
    }
```

```
  }

  // Publish the full marker array
  foot_plan_discrete_viz_pub_.publish(points);
}

void RVizInterface::footPlanContinuousCallback(
    const quad_msgs::MultiFootPlanContinuous::ConstPtr &msg) {
  std::vector<nav_msgs::Path> foot_paths;
  foot_paths.resize(4);

  for (int j = 0; j < msg->states[0].feet.size(); j++) {
    foot_paths[j].header.frame_id = map_frame_;

    for (int i = 0; i < msg->states.size(); i++) {
      geometry_msgs::PoseStamped foot;
      foot.pose.position.x = msg->states[i].feet[j].position.x;
      foot.pose.position.y = msg->states[i].feet[j].position.y;
      foot.pose.position.z = msg->states[i].feet[j].position.z;

      foot_paths[j].poses.push_back(foot);
    }
  }

  foot_0_plan_continuous_viz_pub_.publish(foot_paths[0]);
  foot_1_plan_continuous_viz_pub_.publish(foot_paths[1]);
  foot_2_plan_continuous_viz_pub_.publish(foot_paths[2]);
  foot_3_plan_continuous_viz_pub_.publish(foot_paths[3]);
}

void RVizInterface::robotStateCallback(
    const quad_msgs::RobotState::ConstPtr &msg, const int pub_id) {
  // Make a transform message for the body, populate with state estimate data
  geometry_msgs::TransformStamped transformStamped;
  transformStamped.header = msg->header;
  transformStamped.header.stamp = ros::Time::now();
  transformStamped.header.frame_id = map_frame_;
  transformStamped.transform.translation.x = msg->body.pose.position.x;
  transformStamped.transform.translation.y = msg->body.pose.position.y;
  transformStamped.transform.translation.z = msg->body.pose.position.z;
  transformStamped.transform.rotation = msg->body.pose.orientation;

  // Copy the joint portion of the state estimate message to a new message
  sensor_msgs::JointState joint_msg;
  joint_msg = msg->joints;

  // Set the header to the main header of the state estimate message and publish
  joint_msg.header = msg->header;
  joint_msg.header.stamp = ros::Time::now();

  Eigen::Vector3d current_pos, last_pos;
  quad_utils::pointMsgToEigen(msg->body.pose.position, current_pos);

  if (pub_id == ESTIMATE) {
    transformStamped.child_frame_id = tf_prefix_ + "_estimate/body";
    estimate_base_tf_br_.sendTransform(transformStamped);
    estimate_joint_states_viz_pub_.publish(joint_msg);

    quad_utils::pointMsgToEigen(state_estimate_trace_msg_.points.back(),
                                last_pos);

    // Erase trace if state displacement exceeds threshold, otherwise show
    if ((current_pos - last_pos).norm() >= trace_reset_threshold_) {
      state_estimate_trace_msg_.action = visualization_msgs::Marker::DELETEALL;
      state_estimate_trace_msg_.points.clear();
    } else {
      state_estimate_trace_msg_.action = visualization_msgs::Marker::ADD;
    }

    state_estimate_trace_msg_.points.push_back(msg->body.pose.position);
    state_estimate_trace_msg_.header.stamp = joint_msg.header.stamp;
    state_estimate_trace_pub_.publish(state_estimate_trace_msg_);

  } else if (pub_id == GROUND_TRUTH) {
    transformStamped.child_frame_id = tf_prefix_ + "_ground_truth/body";

    ground_truth_base_tf_br_.sendTransform(transformStamped);
```

```
    ground_truth_joint_states_viz_pub_.publish(joint_msg);

    quad_utils::pointMsgToEigen(ground_truth_state_trace_msg_.points.back(),
                                last_pos);

    // Erase trace if state displacement exceeds threshold, otherwise show
    if ((current_pos - last_pos).norm() >= trace_reset_threshold_) {
      ground_truth_state_trace_msg_.action =
          visualization_msgs::Marker::DELETEALL;
      ground_truth_state_trace_msg_.points.clear();
    } else {
      ground_truth_state_trace_msg_.action = visualization_msgs::Marker::ADD;
    }

    ground_truth_state_trace_msg_.points.push_back(msg->body.pose.position);
    ground_truth_state_trace_msg_.header.stamp = joint_msg.header.stamp;
    ground_truth_state_trace_pub_.publish(ground_truth_state_trace_msg_);

  } else if (pub_id == TRAJECTORY) {
    transformStamped.child_frame_id = tf_prefix_ + "_trajectory/body";
    trajectory_base_tf_br_.sendTransform(transformStamped);
    trajectory_joint_states_viz_pub_.publish(joint_msg);

    quad_utils::pointMsgToEigen(trajectory_state_trace_msg_.points.back(),
                                last_pos);

    // Erase trace if state displacement exceeds threshold, otherwise show
    if ((current_pos - last_pos).norm() >= trace_reset_threshold_) {
      trajectory_state_trace_msg_.action =
          visualization_msgs::Marker::DELETEALL;
      trajectory_state_trace_msg_.points.clear();
    } else {
      trajectory_state_trace_msg_.action = visualization_msgs::Marker::ADD;
    }

    trajectory_state_trace_msg_.points.push_back(msg->body.pose.position);
    trajectory_state_trace_msg_.header.stamp = joint_msg.header.stamp;
    trajectory_state_trace_pub_.publish(trajectory_state_trace_msg_);
  } else {
    ROS_WARN_THROTTLE(
        0.5, "Invalid publisher id, not publishing robot state to rviz");
  }
}

void RVizInterface::spin() {
  ros::Rate r(update_rate_);
  while (ros::ok()) {
    // Collect new messages on subscriber topics
    ros::spinOnce();

    // Enforce update rate
    r.sleep();
  }
}
```

```cpp
#include "quad_utils/trajectory_publisher.h"

TrajectoryPublisher::TrajectoryPublisher(ros::NodeHandle nh) {
  nh_ = nh;

  // Load rosparams from parameter server
  std::string body_plan_topic, trajectory_state_topic;

  nh.param<std::string>("topics/global_plan", body_plan_topic, "/body_plan");
  nh.param<std::string>("topics/state/trajectory", trajectory_state_topic,
                        "/state/trajectory");

  nh.param<std::string>("map_frame", map_frame_, "map");
  nh.param<std::string>("trajectory_publisher/traj_source", traj_source_,
                        "topic");
  nh.param<double>("trajectory_publisher/update_rate", update_rate_, 30);

  // Setup subs and pubs
  body_plan_sub_ = nh_.subscribe(body_plan_topic, 1,
                                 &TrajectoryPublisher::robotPlanCallback, this);

  trajectory_state_pub_ =
      nh_.advertise<quad_msgs::RobotState>(trajectory_state_topic, 1);

  // Initialize kinematics object
  quadKD_ = std::make_shared<quad_utils::QuadKD>();
}

void TrajectoryPublisher::importTrajectory() {
  // Load the desired values into body_plan_msg_ here
  return;
}

void TrajectoryPublisher::robotPlanCallback(
    const quad_msgs::RobotPlan::ConstPtr& msg) {
  // Save the most recent body plan
  body_plan_msg_ = (*msg);
}

void TrajectoryPublisher::publishTrajectoryState() {
  // Wait until we actually have data
  if (body_plan_msg_.states.empty()) {
    return;
  }

  // Get the current time in the trajectory since the beginning of the plan
  double traj_duration = (body_plan_msg_.states.back().header.stamp -
                          body_plan_msg_.states.front().header.stamp)
                             .toSec();
  double t =
      (ros::Time::now() - body_plan_msg_.states.front().header.stamp).toSec();

  // Ensure the trajectory remains valid
  t = std::min(t, traj_duration);

  // Interpolate to get the correct state and publish it
  quad_msgs::RobotState interp_state;
  int interp_primitive_id;
  quad_msgs::GRFArray interp_grf;

  quad_utils::interpRobotPlan(body_plan_msg_, t, interp_state,
                              interp_primitive_id, interp_grf);

  // Fill joints and feet with dummy data
  if (interp_state.joints.name.empty()) {
    interp_state.joints.name = {"8",  "0", "1", "9",  "2", "3",
                                "10", "4", "5", "11", "6", "7"};
    interp_state.joints.position = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    interp_state.joints.velocity = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    interp_state.joints.effort = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
  }
  quad_utils::fkRobotState(*quadKD_, interp_state.body, interp_state.joints,
                           interp_state.feet);

  trajectory_state_pub_.publish(interp_state);
}
```

```
void TrajectoryPublisher::spin() {
  ros::Rate r(update_rate_);
  if (traj_source_.compare("import") == 0) {
    importTrajectory();
  }
  while (ros::ok()) {
    // Publish the trajectory state
    publishTrajectoryState();

    // Collect new messages on subscriber topics
    ros::spinOnce();

    // Enforce update rate
    r.sleep();
  }
}
```

```cpp
#include "quad_utils/fast_terrain_map.h"

#include <grid_map_core/grid_map_core.hpp>
#include <iostream>

FastTerrainMap::FastTerrainMap() {}

void FastTerrainMap::loadData(int x_size, int y_size,
                              std::vector<double> x_data,
                              std::vector<double> y_data,
                              std::vector<std::vector<double>> z_data,
                              std::vector<std::vector<double>> nx_data,
                              std::vector<std::vector<double>> ny_data,
                              std::vector<std::vector<double>> nz_data,
                              std::vector<std::vector<double>> z_data_filt,
                              std::vector<std::vector<double>> nx_data_filt,
                              std::vector<std::vector<double>> ny_data_filt,
                              std::vector<std::vector<double>> nz_data_filt) {
  // Load the data into the object's private variables
  x_size_ = x_size;
  y_size_ = y_size;
  x_data_ = x_data;
  y_data_ = y_data;

  x_diff_ = x_data_[1] - x_data_[0];
  y_diff_ = y_data_[1] - y_data_[0];

  z_data_ = z_data;
  nx_data_ = nx_data;
  ny_data_ = ny_data;
  nz_data_ = nz_data;

  z_data_filt_ = z_data_filt;
  nx_data_filt_ = nx_data_filt;
  ny_data_filt_ = ny_data_filt;
  nz_data_filt_ = nz_data_filt;
}

void FastTerrainMap::loadFlat() {
  int x_size = 2;
  int y_size = 2;
  std::vector<double> x_data = {-5, 5};
  std::vector<double> y_data = {-5, 5};
  std::vector<double> z_data_vec = {0, 0};
  std::vector<double> nz_data_vec = {1, 1};
  std::vector<std::vector<double>> z_data = {z_data_vec, z_data_vec};
  std::vector<std::vector<double>> nx_data = z_data;
  std::vector<std::vector<double>> ny_data = z_data;
  std::vector<std::vector<double>> nz_data = {nz_data_vec, nz_data_vec};
  std::vector<std::vector<double>> z_data_filt = z_data;
  std::vector<std::vector<double>> nx_data_filt = nx_data;
  std::vector<std::vector<double>> ny_data_filt = ny_data;
  std::vector<std::vector<double>> nz_data_filt = nz_data;

  this->loadData(x_size, y_size, x_data, y_data, z_data, nx_data, ny_data,
                 nz_data, z_data_filt, nx_data_filt, ny_data_filt,
                 nz_data_filt);
}

void FastTerrainMap::loadFlatElevated(double height) {
  int x_size = 2;
  int y_size = 2;
  std::vector<double> x_data = {-5, 5};
  std::vector<double> y_data = {-5, 5};
  std::vector<double> z_data_vec = {height, height};
  std::vector<double> nx_data_vec = {0, 0};
  std::vector<double> nz_data_vec = {1, 1};
  std::vector<std::vector<double>> z_data = {z_data_vec, z_data_vec};
  std::vector<std::vector<double>> nx_data = {nx_data_vec, nx_data_vec};
  std::vector<std::vector<double>> ny_data = nx_data;
  std::vector<std::vector<double>> nz_data = {nz_data_vec, nz_data_vec};
  std::vector<std::vector<double>> z_data_filt = z_data;
  std::vector<std::vector<double>> nx_data_filt = nx_data;
  std::vector<std::vector<double>> ny_data_filt = ny_data;
  std::vector<std::vector<double>> nz_data_filt = nz_data;

  this->loadData(x_size, y_size, x_data, y_data, z_data, nx_data, ny_data,
```

```
                 nz_data, z_data_filt, nx_data_filt, ny_data_filt,
                 nz_data_filt);
}

void FastTerrainMap::loadSlope(double grade) {
  double slope = atan(grade);
  int x_size = 2;
  int y_size = 2;
  double length = 5;
  std::vector<double> x_data = {-length, length};
  std::vector<double> y_data = {-length, length};
  std::vector<double> z_data_vec_1 = {-length * grade, -length * grade};
  std::vector<double> z_data_vec_2 = {length * grade, length * grade};
  std::vector<double> nx_data_vec = {-sin(slope), -sin(slope)};
  std::vector<double> ny_data_vec = {0, 0};
  std::vector<double> nz_data_vec = {cos(slope), cos(slope)};
  std::vector<std::vector<double>> z_data = {z_data_vec_1, z_data_vec_2};
  std::vector<std::vector<double>> nx_data = {nx_data_vec, nx_data_vec};
  std::vector<std::vector<double>> ny_data = {ny_data_vec, ny_data_vec};
  std::vector<std::vector<double>> nz_data = {nz_data_vec, nz_data_vec};
  std::vector<std::vector<double>> z_data_filt = {z_data_vec_1, z_data_vec_2};
  std::vector<std::vector<double>> nx_data_filt = {nx_data_vec, nx_data_vec};
  std::vector<std::vector<double>> ny_data_filt = {ny_data_vec, ny_data_vec};
  std::vector<std::vector<double>> nz_data_filt = {nz_data_vec, nz_data_vec};

  this->loadData(x_size, y_size, x_data, y_data, z_data, nx_data, ny_data,
                 nz_data, z_data_filt, nx_data_filt, ny_data_filt,
                 nz_data_filt);
}

void FastTerrainMap::loadStep(double height) {
  double res = 0.05;
  double length = 2;
  int x_size = length * 2 / res + 1;
  int y_size = x_size;

  std::vector<double> x_data;
  std::vector<double> y_data;
  std::vector<std::vector<double>> z_data(x_size);
  std::vector<std::vector<double>> nx_data(x_size);
  std::vector<std::vector<double>> ny_data(x_size);
  std::vector<std::vector<double>> nz_data(x_size);
  std::vector<std::vector<double>> z_data_filt(x_size);
  std::vector<std::vector<double>> nx_data_filt(x_size);
  std::vector<std::vector<double>> ny_data_filt(x_size);
  std::vector<std::vector<double>> nz_data_filt(x_size);

  for (int i = 0; i < x_size; i++) {
    double x = i * res - length;
    x_data.push_back(i * res - length);
    y_data.push_back(i * res - length);

    z_data[i].resize(y_size);
    nx_data[i].resize(y_size);
    ny_data[i].resize(y_size);
    nz_data[i].resize(y_size);
    z_data_filt[i].resize(y_size);
    nx_data_filt[i].resize(y_size);
    ny_data_filt[i].resize(y_size);
    nz_data_filt[i].resize(y_size);

    for (int j = 0; j < y_size; j++) {
      double y = j * res - length;
      z_data[i][j] = (x > 0) ? height : 0;
      nx_data[i][j] = 0;
      ny_data[i][j] = 0;
      nz_data[i][j] = 1;
    }
  }

  z_data_filt = z_data;
  nx_data_filt = nx_data;
  ny_data_filt = ny_data;
  nz_data_filt = nz_data;

  this->loadData(x_size, y_size, x_data, y_data, z_data, nx_data, ny_data,
                 nz_data, z_data_filt, nx_data_filt, ny_data_filt,
```

```
                  nz_data_filt);
}

void FastTerrainMap::loadDataFromGridMap(const grid_map::GridMap map) {
  // Initialize the data structures for the map
  int x_size = map.getSize()(0);
  int y_size = map.getSize()(1);
  std::vector<double> x_data(x_size);
  std::vector<double> y_data(y_size);
  std::vector<std::vector<double>> z_data(x_size);
  std::vector<std::vector<double>> nx_data(x_size);
  std::vector<std::vector<double>> ny_data(x_size);
  std::vector<std::vector<double>> nz_data(x_size);
  std::vector<std::vector<double>> z_data_filt(x_size);
  std::vector<std::vector<double>> nx_data_filt(x_size);
  std::vector<std::vector<double>> ny_data_filt(x_size);
  std::vector<std::vector<double>> nz_data_filt(x_size);

  // Load the x and y data coordinates
  for (int i = 0; i < x_size; i++) {
    grid_map::Index index = {(x_size - 1) - i, 0};
    grid_map::Position position;
    map.getPosition(index, position);
    x_data[i] = position.x();
  }
  for (int i = 0; i < y_size; i++) {
    grid_map::Index index = {0, (y_size - 1) - i};
    grid_map::Position position;
    map.getPosition(index, position);
    y_data[i] = position.y();
  }

  // Loop through the map and get the height and slope info
  for (int i = 0; i < x_size; i++) {
    for (int j = 0; j < y_size; j++) {
      grid_map::Index index = {(x_size - 1) - i, (y_size - 1) - j};
      double height = (double)map.at("z_inpainted", index);
      z_data[i].push_back(height);

      if (map.exists("normal_vectors_x") == true) {
        double nx = (double)map.at("normal_vectors_x", index);
        double ny = (double)map.at("normal_vectors_y", index);
        double nz = (double)map.at("normal_vectors_z", index);
        nx_data[i].push_back(nx);
        ny_data[i].push_back(ny);
        nz_data[i].push_back(nz);
      } else {
        nx_data[i].push_back(0.0);
        ny_data[i].push_back(0.0);
        nz_data[i].push_back(1.0);
      }

      if (map.exists("z_smooth") == true) {
        double z_filt = (double)map.at("z_smooth", index);
        double nx_filt = (double)map.at("smooth_normal_vectors_x", index);
        double ny_filt = (double)map.at("smooth_normal_vectors_y", index);
        double nz_filt = (double)map.at("smooth_normal_vectors_z", index);
        z_data_filt[i].push_back(z_filt);
        nx_data_filt[i].push_back(nx_filt);
        ny_data_filt[i].push_back(ny_filt);
        nz_data_filt[i].push_back(nz_filt);
      } else {
        z_data_filt[i].push_back(height);
        nx_data_filt[i].push_back(0.0);
        ny_data_filt[i].push_back(0.0);
        nz_data_filt[i].push_back(1.0);
      }
    }
  }

  // Update the private terrain member
  x_size_ = x_size;
  y_size_ = y_size;
  x_data_ = x_data;
  y_data_ = y_data;

  x_diff_ = x_data_[1] - x_data_[0];
```

```
  y_diff_ = y_data_[1] - y_data_[0];

  z_data_ = z_data;
  nx_data_ = nx_data;
  ny_data_ = ny_data;
  nz_data_ = nz_data;

  z_data_filt_ = z_data_filt;
  nx_data_filt_ = nx_data_filt;
  ny_data_filt_ = ny_data_filt;
  nz_data_filt_ = nz_data_filt;
}

bool FastTerrainMap::isInRange(const double x, const double y) const {
  double epsilon = 0.5;
  if (((x - epsilon) >= x_data_.front()) && ((x + epsilon) <= x_data_.back()) &&
      ((y - epsilon) >= y_data_.front()) && ((y + epsilon) <= y_data_.back())) {
    return true;
  } else {
    return false;
  }
}

double FastTerrainMap::getGroundHeight(const double x, const double y) const {
  // quad_utils::FunctionTimer timer(__FUNCTION__);
  int ix = getXIndex(x);
  int iy = getYIndex(y);

  double x1 = x_data_[ix];
  double x2 = x_data_[ix + 1];
  double y1 = y_data_[iy];
  double y2 = y_data_[iy + 1];

  // Perform bilinear interpolation
  double fx1y1 = z_data_[ix][iy];
  double fx1y2 = z_data_[ix][iy + 1];
  double fx2y1 = z_data_[ix + 1][iy];
  double fx2y2 = z_data_[ix + 1][iy + 1];
  double height = 1.0 / ((x2 - x1) * (y2 - y1)) *
                  (fx1y1 * (x2 - x) * (y2 - y) + fx2y1 * (x - x1) * (y2 - y) +
                   fx1y2 * (x2 - x) * (y - y1) + fx2y2 * (x - x1) * (y - y1));

  // timer.reportStatistics();
  return height;
}

double FastTerrainMap::getGroundHeightFiltered(const double x,
                                               const double y) const {
  // quad_utils::FunctionTimer timer(__FUNCTION__);

  int ix = getXIndex(x);
  int iy = getYIndex(y);
  double x1 = x_data_[ix];
  double x2 = x_data_[ix + 1];
  double y1 = y_data_[iy];
  double y2 = y_data_[iy + 1];

  // Perform bilinear interpolation
  double fx1y1 = z_data_filt_[ix][iy];
  double fx1y2 = z_data_filt_[ix][iy + 1];
  double fx2y1 = z_data_filt_[ix + 1][iy];
  double fx2y2 = z_data_filt_[ix + 1][iy + 1];
  double height = 1.0 / ((x2 - x1) * (y2 - y1)) *
                  (fx1y1 * (x2 - x) * (y2 - y) + fx2y1 * (x - x1) * (y2 - y) +
                   fx1y2 * (x2 - x) * (y - y1) + fx2y2 * (x - x1) * (y - y1));

  // timer.reportStatistics();
  return height;
}

std::array<double, 3> FastTerrainMap::getSurfaceNormal(const double x,
                                                       const double y) const {
  std::array<double, 3> surf_norm;

  int ix = getXIndex(x);
  int iy = getYIndex(y);
  double x1 = x_data_[ix];
```

```
  double x2 = x_data_[ix + 1];
  double y1 = y_data_[iy];
  double y2 = y_data_[iy + 1];

  double fx_x1y1 = nx_data_[ix][iy];
  double fx_x1y2 = nx_data_[ix][iy + 1];
  double fx_x2y1 = nx_data_[ix + 1][iy];
  double fx_x2y2 = nx_data_[ix + 1][iy + 1];

  surf_norm[0] =
      1.0 / ((x2 - x1) * (y2 - y1)) *
      (fx_x1y1 * (x2 - x) * (y2 - y) + fx_x2y1 * (x - x1) * (y2 - y) +
       fx_x1y2 * (x2 - x) * (y - y1) + fx_x2y2 * (x - x1) * (y - y1));

  double fy_x1y1 = ny_data_[ix][iy];
  double fy_x1y2 = ny_data_[ix][iy + 1];
  double fy_x2y1 = ny_data_[ix + 1][iy];
  double fy_x2y2 = ny_data_[ix + 1][iy + 1];

  surf_norm[1] =
      1.0 / ((x2 - x1) * (y2 - y1)) *
      (fy_x1y1 * (x2 - x) * (y2 - y) + fy_x2y1 * (x - x1) * (y2 - y) +
       fy_x1y2 * (x2 - x) * (y - y1) + fy_x2y2 * (x - x1) * (y - y1));

  double fz_x1y1 = nz_data_[ix][iy];
  double fz_x1y2 = nz_data_[ix][iy + 1];
  double fz_x2y1 = nz_data_[ix + 1][iy];
  double fz_x2y2 = nz_data_[ix + 1][iy + 1];

  surf_norm[2] =
      1.0 / ((x2 - x1) * (y2 - y1)) *
      (fz_x1y1 * (x2 - x) * (y2 - y) + fz_x2y1 * (x - x1) * (y2 - y) +
       fz_x1y2 * (x2 - x) * (y - y1) + fz_x2y2 * (x - x1) * (y - y1));
  return surf_norm;
}

std::array<double, 3> FastTerrainMap::getSurfaceNormalFiltered(
    const double x, const double y) const {
  std::array<double, 3> surf_norm;

  int ix = getXIndex(x);
  int iy = getYIndex(y);
  double x1 = x_data_[ix];
  double x2 = x_data_[ix + 1];
  double y1 = y_data_[iy];
  double y2 = y_data_[iy + 1];

  double fx_x1y1 = nx_data_filt_[ix][iy];
  double fx_x1y2 = nx_data_filt_[ix][iy + 1];
  double fx_x2y1 = nx_data_filt_[ix + 1][iy];
  double fx_x2y2 = nx_data_filt_[ix + 1][iy + 1];

  surf_norm[0] =
      1.0 / ((x2 - x1) * (y2 - y1)) *
      (fx_x1y1 * (x2 - x) * (y2 - y) + fx_x2y1 * (x - x1) * (y2 - y) +
       fx_x1y2 * (x2 - x) * (y - y1) + fx_x2y2 * (x - x1) * (y - y1));

  double fy_x1y1 = ny_data_filt_[ix][iy];
  double fy_x1y2 = ny_data_filt_[ix][iy + 1];
  double fy_x2y1 = ny_data_filt_[ix + 1][iy];
  double fy_x2y2 = ny_data_filt_[ix + 1][iy + 1];

  surf_norm[1] =
      1.0 / ((x2 - x1) * (y2 - y1)) *
      (fy_x1y1 * (x2 - x) * (y2 - y) + fy_x2y1 * (x - x1) * (y2 - y) +
       fy_x1y2 * (x2 - x) * (y - y1) + fy_x2y2 * (x - x1) * (y - y1));

  double fz_x1y1 = nz_data_filt_[ix][iy];
  double fz_x1y2 = nz_data_filt_[ix][iy + 1];
  double fz_x2y1 = nz_data_filt_[ix + 1][iy];
  double fz_x2y2 = nz_data_filt_[ix + 1][iy + 1];

  surf_norm[2] =
      1.0 / ((x2 - x1) * (y2 - y1)) *
      (fz_x1y1 * (x2 - x) * (y2 - y) + fz_x2y1 * (x - x1) * (y2 - y) +
       fz_x1y2 * (x2 - x) * (y - y1) + fz_x2y2 * (x - x1) * (y - y1));
  return surf_norm;
```

```cpp
}

Eigen::Vector3d FastTerrainMap::getSurfaceNormalFilteredEigen(
    const double x, const double y) const {
  Eigen::Vector3d surf_norm;

  int ix = getXIndex(x);
  int iy = getYIndex(y);
  double x1 = x_data_[ix];
  double x2 = x_data_[ix + 1];
  double y1 = y_data_[iy];
  double y2 = y_data_[iy + 1];

  double fx_x1y1 = nx_data_filt_[ix][iy];
  double fx_x1y2 = nx_data_filt_[ix][iy + 1];
  double fx_x2y1 = nx_data_filt_[ix + 1][iy];
  double fx_x2y2 = nx_data_filt_[ix + 1][iy + 1];

  surf_norm[0] =
      1.0 / ((x2 - x1) * (y2 - y1)) *
      (fx_x1y1 * (x2 - x) * (y2 - y) + fx_x2y1 * (x - x1) * (y2 - y) +
       fx_x1y2 * (x2 - x) * (y - y1) + fx_x2y2 * (x - x1) * (y - y1));

  double fy_x1y1 = ny_data_filt_[ix][iy];
  double fy_x1y2 = ny_data_filt_[ix][iy + 1];
  double fy_x2y1 = ny_data_filt_[ix + 1][iy];
  double fy_x2y2 = ny_data_filt_[ix + 1][iy + 1];

  surf_norm[1] =
      1.0 / ((x2 - x1) * (y2 - y1)) *
      (fy_x1y1 * (x2 - x) * (y2 - y) + fy_x2y1 * (x - x1) * (y2 - y) +
       fy_x1y2 * (x2 - x) * (y - y1) + fy_x2y2 * (x - x1) * (y - y1));

  double fz_x1y1 = nz_data_filt_[ix][iy];
  double fz_x1y2 = nz_data_filt_[ix][iy + 1];
  double fz_x2y1 = nz_data_filt_[ix + 1][iy];
  double fz_x2y2 = nz_data_filt_[ix + 1][iy + 1];

  surf_norm[2] =
      1.0 / ((x2 - x1) * (y2 - y1)) *
      (fz_x1y1 * (x2 - x) * (y2 - y) + fz_x2y1 * (x - x1) * (y2 - y) +
       fz_x1y2 * (x2 - x) * (y - y1) + fz_x2y2 * (x - x1) * (y - y1));
  // std::cout << "surf_norm:\n" << surf_norm << std::endl;
  return surf_norm;
}

Eigen::Vector3d FastTerrainMap::projectToMap(const Eigen::Vector3d point,
                                             const Eigen::Vector3d direction) {
  // quad_utils::FunctionTimer timer(__FUNCTION__);

  Eigen::Vector3d direction_norm = direction;
  direction_norm.normalize();
  Eigen::Vector3d result = point;
  Eigen::Vector3d new_point = point;
  Eigen::Vector3d old_point = point;
  double step_size = 0.01;
  double clearance = 0;
  while (clearance >= 0) {
    old_point = new_point;
    for (int i = 0; i < 3; i++) {
      new_point[i] += direction_norm[i] * step_size;
    }
    if (isInRange(new_point[0], new_point[1])) {
      clearance = new_point[2] - getGroundHeight(new_point[0], new_point[1]);
    } else {
      result = {old_point[0], old_point[1],
                -std::numeric_limits<double>::max()};
      ROS_WARN_THROTTLE(0.5, "Tried to project to a point off the map.");
      return result;
    }
  }

  result = {old_point[0], old_point[1],
            getGroundHeight(old_point[0], old_point[1])};

  // timer.reportStatistics();
  return result;
```

```
}

std::vector<double> FastTerrainMap::getXData() const { return x_data_; }

std::vector<double> FastTerrainMap::getYData() const { return y_data_; }

bool FastTerrainMap::isEmpty() const {
  if (x_size_ == 0 || y_size_ == 0) {
    return true;
  } else {
    return false;
  }
}
```

```cpp
#include <ros/ros.h>

#include "quad_utils/rviz_interface.h"

int main(int argc, char** argv) {
  ros::init(argc, argv, "rviz_interface_node");
  ros::NodeHandle nh;

  RVizInterface rviz_interface(nh);
  rviz_interface.spin();

  return 0;
}
```

```cpp
#include <ros/ros.h>

#include "quad_utils/trajectory_publisher.h"

int main(int argc, char** argv) {
  ros::init(argc, argv, "trajectory_publisher_node");
  ros::NodeHandle nh;

  TrajectoryPublisher trajectory_publisher(nh);
  trajectory_publisher.spin();

  return 0;
}
```