```
#ifndef REMOTE_HEARTBEAT_H
#define REMOTE_HEARTBEAT_H

#include <quad_msgs/LegCommandArray.h>
#include <quad_utils/ros_utils.h>
#include <ros/ros.h>

//! A class for implementing a remote heartbeat
/*!
   RemoteHeartbeat publishes stamped messages at a fixed rate as a heartbeat
*/
class RemoteHeartbeat {
 public:
  /**
   * @brief Constructor for RemoteHeartbeat Class
   * @param[in] nh ROS NodeHandle to publish and subscribe from
   * @return Constructed object of type RemoteHeartbeat
   */
  RemoteHeartbeat(ros::NodeHandle nh);

  /**
   * @brief Calls ros spinOnce and pubs data at set frequency
   */
  void spin();

 private:
  /**
   * @brief Callback function to handle new robot heartbeat
   * @param[in] msg header containing robot heartbeat
   */
  void robotHeartbeatCallback(const std_msgs::Header::ConstPtr& msg);

  /// Nodehandle to pub to and sub from
  ros::NodeHandle nh_;

  /// Subscriber for robot heartbeat messages
  ros::Subscriber robot_heartbeat_sub_;

  /// ROS publisher for remote heartbeat messages
  ros::Publisher remote_heartbeat_pub_;

  /// Update rate for sending and receiving data
  double update_rate_;

  /// Latency threshold on robot messages for warnings (s)
  double robot_latency_threshold_warn_;

  /// Latency threshold on robot messages for error (s)
  double robot_latency_threshold_error_;
};

#endif  // REMOTE_HEARTBEAT_H
```

```cpp
#ifndef TERRAIN_MAP_PUBLISHER_H
#define TERRAIN_MAP_PUBLISHER_H

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

#include <fstream>  // ifstream
#include <grid_map_core/grid_map_core.hpp>
#include <grid_map_ros/GridMapRosConverter.hpp>
#include <grid_map_ros/grid_map_ros.hpp>
#include <iostream>  // cout
#include <sstream>   // istringstream
#include <string>
#include <vector>

struct Obstacle {
  double x;
  double y;
  double height;
  double radius;
};

struct Step {
  double x;
  double height;
};

//! A terrain map publishing class
/*!
   TerrainMapPublisher is a class for publishing terrain maps from a variety of
   sources, including from scratch.
*/
class TerrainMapPublisher {
 public:
  /**
   * @brief Constructor for TerrainMapPublisher Class
   * @param[in] nh ROS NodeHandle to publish and subscribe from
   * @return Constructed object of type TerrainMapPublisher
   */
  TerrainMapPublisher(ros::NodeHandle nh);

  /**
   * @brief Updates the terrain_map_publisher parameters
   */
  void updateParams();

  /**
   * @brief Creates the map object from scratch
   */
  void createMap();

  /**
   * @brief Updates the map object with params
   */
  void updateMap();

  /**
   * @brief Loads data from a specified CSV file into a nested std::vector
   * structure
   * @param[in] filename Path to the CSV file
   * @return Data from the CSV in vector structure
   */
  std::vector<std::vector<double> > loadCSV(std::string filename);

  /**
   * @brief Loads data into the map object from a CSV
   */
  void loadMapFromCSV();

  /**
   * @brief Loads data into the map object from an image topic
   * @param[in] msg ROS image message
   */
  void loadMapFromImage(const sensor_msgs::Image& msg);

  /**
   * @brief Publishes map data to the terrain_map topic
   */
  void publishMap();

  /**
   * @brief Calls ros spinOnce and pubs data at set frequency
   */
  void spin();

 private:
  /// ROS Subscriber for image data
  ros::Subscriber image_sub_;

  /// ROS Publisher for the terrain map
  ros::Publisher terrain_map_pub_;

  /// Nodehandle to pub to and sub from
  ros::NodeHandle nh_;

  /// Update rate for sending and receiving data, unused since pubs are called
  /// in callbacks
  double update_rate_;

  /// Handle for the map frame
  std::string map_frame_;

  /// grid_map::GridMap object for terrain data
  grid_map::GridMap terrain_map_;

  /// String for the terrain file name
  std::string terrain_type_;

  /// string of the source of the terrain map data
  std::string map_data_source_;

  /// bool to flag if the map has been initialized yet
  bool map_initialized_ = false;

  /// double for map resolution
  double resolution_;

  /// double for map resolution
  double min_height_;

  /// double for map resolution
  double max_height_;

  /// Obstacle object
  Obstacle obstacle_;

  /// Step 1 object
  Step step1_;

  /// Step 2 object
  Step step2_;
};

#endif  // TERRAIN_MAP_PUBLISHER_H
```

```
#ifndef RVIZ_INTERFACE_H
#define RVIZ_INTERFACE_H

#include <geometry_msgs/PoseArray.h>
#include <geometry_msgs/PoseStamped.h>
#include <nav_msgs/Path.h>
#include <quad_msgs/FootPlanDiscrete.h>
#include <quad_msgs/FootState.h>
#include <quad_msgs/GRFArray.h>
#include <quad_msgs/MultiFootPlanContinuous.h>
#include <quad_msgs/MultiFootPlanDiscrete.h>
#include <quad_msgs/MultiFootState.h>
#include <quad_msgs/RobotPlan.h>
#include <quad_msgs/RobotState.h>
#include <quad_utils/ros_utils.h>
#include <ros/ros.h>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2_ros/transform_broadcaster.h>
#include <visualization_msgs/Marker.h>
#include <visualization_msgs/MarkerArray.h>

//! A class for interfacing between RViz and quad-sdk topics.
/*!
   RVizInterface is a container for all of the logic utilized in the template
   node. The implementation must provide a clean and high level interface to the
   core algorithm
*/
class RVizInterface {
 public:
  /**
   * @brief Constructor for RVizInterface Class
   * @param[in] nh ROS NodeHandle to publish and subscribe from
   * @return Constructed object of type RVizInterface
   */
  RVizInterface(ros::NodeHandle nh);

  /**
   * @brief Calls ros spinOnce and pubs data at set frequency
   */
  void spin();

 private:
  /**
   * @brief Callback function to handle new body plan data
   * @param[in] msg plan message contining interpolated output of body planner
   */
  void robotPlanCallback(const quad_msgs::RobotPlan::ConstPtr &msg,
                         const int pub_id);

  /**
   * @brief Callback function to handle new grf data
   * @param[in] msg plan message contining interpolated output of body planner
   */
  void grfCallback(const quad_msgs::GRFArray::ConstPtr &msg);

  /**
   * @brief Callback function to handle new body plan discrete state data
   * @param[in] msg plan message contining discrete output of body planner
   */
  void discreteBodyPlanCallback(const quad_msgs::RobotPlan::ConstPtr &msg);

  /**
   * @brief Callback function to handle new discrete foot plan data
   * @param[in] Footstep plan message containing output of footstep planner
   */
  void footPlanDiscreteCallback(
      const quad_msgs::MultiFootPlanDiscrete::ConstPtr &msg);

  /**
   * @brief Callback function to handle new continous foot plan data
   * @param[in] SwingLegPlan message containing output of swing leg planner
   */
  void footPlanContinuousCallback(
      const quad_msgs::MultiFootPlanContinuous::ConstPtr &msg);

  /**
   * @brief Callback function to handle new state estimate data
   * @param[in] msg RobotState message containing output of the state estimator
   * node
   */
  void stateEstimateCallback(const quad_msgs::RobotState::ConstPtr &msg);

  /**
   * @brief Callback function to handle new robot state data
   * @param[in] msg RobotState message containing output of the state estimator
   * node
   * @param[in] pub_id Identifier of which publisher to use to handle this data
   */
  void robotStateCallback(const quad_msgs::RobotState::ConstPtr &msg,
                          const int pub_id);

  /// ROS subscriber for the global plan
  ros::Subscriber global_plan_sub_;

  /// ROS subscriber for the local plan
  ros::Subscriber local_plan_sub_;

  /// ROS subscriber for the current
  ros::Subscriber grf_sub_;

  /// ROS subscriber for the body plan
  ros::Subscriber discrete_body_plan_sub_;

  /// ROS subscriber for the discrete foot plan
  ros::Subscriber foot_plan_discrete_sub_;

  /// ROS subscriber for the continuous foot plan
  ros::Subscriber foot_plan_continuous_sub_;

  /// ROS Publisher for the interpolated global plan vizualization
  ros::Publisher global_plan_viz_pub_;

  /// ROS Publisher for the interpolated local plan vizualization
  ros::Publisher local_plan_viz_pub_;

  /// ROS Publisher for the current GRFs
  ros::Publisher current_grf_viz_pub_;

  /// ROS Publisher for local plan orientation vizualization
  ros::Publisher local_plan_ori_viz_pub_;

  /// ROS Publisher for the interpolated global plan grf vizualization
  ros::Publisher global_plan_grf_viz_pub_;

  /// ROS Publisher for the interpolated local plan grf vizualization
  ros::Publisher local_plan_grf_viz_pub_;

  /// ROS Publisher for the discrete body plan vizualization
  ros::Publisher discrete_body_plan_viz_pub_;

  /// ROS Publisher for the footstep plan visualization
  ros::Publisher foot_plan_discrete_viz_pub_;

  /// ROS Publisher for the state estimate body trace
  ros::Publisher state_estimate_trace_pub_;

  /// ROS Publisher for the ground truth state body trace
  ros::Publisher ground_truth_state_trace_pub_;

  /// ROS Publisher for the trajectory state body trace
  ros::Publisher trajectory_state_trace_pub_;

  /// ROS Publisher for the swing leg 0 visualization
  ros::Publisher foot_0_plan_continuous_viz_pub_;

  /// ROS Publisher for the foot 1 plan visualization
  ros::Publisher foot_1_plan_continuous_viz_pub_;

  /// ROS Publisher for the foot 2 plan visualization
  ros::Publisher foot_2_plan_continuous_viz_pub_;

  /// ROS Publisher for the foot 3 plan visualization
  ros::Publisher foot_3_plan_continuous_viz_pub_;

  /// ROS Publisher for the estimated joint states visualization
  ros::Publisher estimate_joint_states_viz_pub_;

  /// ROS Publisher for the ground truth joint states visualization
  ros::Publisher ground_truth_joint_states_viz_pub_;

  /// ROS Publisher for the trajectory joint states visualization
  ros::Publisher trajectory_joint_states_viz_pub_;

  /// ROS Subscriber for the state estimate
  ros::Subscriber state_estimate_sub_;

  /// ROS Subscriber for the ground truth state
  ros::Subscriber ground_truth_state_sub_;

  /// ROS Subscriber for the ground truth state
  ros::Subscriber trajectory_state_sub_;

  /// ROS Transform Broadcaster to publish the estimate transform for the base
  /// link
  tf2_ros::TransformBroadcaster estimate_base_tf_br_;

  /// ROS Transform Broadcaster to publish the ground truth transform for the
  /// base link
  tf2_ros::TransformBroadcaster ground_truth_base_tf_br_;

  /// ROS Transform Broadcaster to publish the trajectory transform for the base
  /// link
  tf2_ros::TransformBroadcaster trajectory_base_tf_br_;

  /// Message for state estimate trace
  visualization_msgs::Marker state_estimate_trace_msg_;

  /// Message for ground truth state trace
  visualization_msgs::Marker ground_truth_state_trace_msg_;

  /// Message for trajectory state trace
  visualization_msgs::Marker trajectory_state_trace_msg_;

  /// Distance threshold for resetting the state traces
  const double trace_reset_threshold_ = 0.2;

  /// Nodehandle to pub to and sub from
  ros::NodeHandle nh_;

  /// Update rate for sending and receiving data, unused since pubs are called
  /// in callbacks
  double update_rate_;

  /// Interval for showing orientation of plan
  int orientation_subsample_interval_;

  /// Handle for the map frame
  std::string map_frame_;

  /// Handle multiple robots
  std::string tf_prefix_;

  /// Colors
  std::vector<int> front_left_color_;
  std::vector<int> back_left_color_;
  std::vector<int> front_right_color_;
  std::vector<int> back_right_color_;
  std::vector<int> net_grf_color_;
  std::vector<int> individual_grf_color_;

  /// Publisher IDs
  const int ESTIMATE = 0;
  const int GROUND_TRUTH = 1;
  const int TRAJECTORY = 2;

  const int GLOBAL = 0;
  const int LOCAL = 1;

  const int CONNECT = 0;
  const int LEAP_STANCE = 1;
  const int FLIGHT = 2;
  const int LAND_STANCE = 3;
};

#endif  // RVIZ_INTERFACE_H
```

```
#ifndef FAST_TERRAIN_MAP_H
#define FAST_TERRAIN_MAP_H

#include <quad_utils/function_timer.h>
#include <ros/ros.h>

#include <chrono>
#include <eigen3/Eigen/Eigen>
#include <grid_map_core/grid_map_core.hpp>

//! A terrain map class built for fast and efficient sampling
/*!
   FastTerrainMap is a class built for lightweight and efficient sampling of the
   terrain for height and slope.
*/
class FastTerrainMap {
 public:
  /**
   * @brief Constructor for FastTerrainMap Class
   * @return Constructed object of type FastTerrainMap
   */
  FastTerrainMap();

  /**
   * @brief Load data from a grid_map::GridMap object into a FastTerrainMap
   * object
   * @param[in] int The number of elements in the x direction
   * @param[in] int The number of elements in the xy direction
   * @param[in] std::vector<double> The vector of x data
   * @param[in] std::vector<double> The vector of y data
   * @param[in] std::vector<std::vector<double>> The nested vector of z data at
   * each [x,y] location
   * @param[in] std::vector<std::vector<double>> The nested vector of the x
   * component of the gradient at each [x,y] location
   * @param[in] std::vector<std::vector<double>> The nested vector of the y
   * component of the gradient at each [x,y] location
   * @param[in] std::vector<std::vector<double>> The nested vector of the z
   * component of the gradient at each [x,y] location
   */
  void 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);

  /**
   * @brief Load in a default terrain map 10x10m, four corners with flat terrain
   */
  void loadFlat();

  /**
   * @brief Load in a default terrain map 10x10m, four corners with elevated
   * terrain
   * @param[in] height Height of elevated terrain
   */
  void loadFlatElevated(double height);

  /**
   * @brief Load in a default terrain map 10x10m, four corners with sloped
   * terrain
   * @param[in] grade Grade of terrain data (grade = tan(slope))
   */
  void loadSlope(double grade);

  /**
   * @brief Load in a terrain map with a step at x = 0
   * @param[in] height Height of step
   */
  void loadStep(double height);

  /**
   * @brief Load data from a grid_map::GridMap object into a FastTerrainMap
```

```cpp
   * object
   * @param[in] grid_map::GridMap object with map data
   */
  void loadDataFromGridMap(const grid_map::GridMap map);

  /**
   * @brief Check if map data is defined at a requested location
   * @param[in] double x location
   * @param[in] double y location
   * @return bool location [x,y] is or is not in range
   */
  bool isInRange(const double x, const double y) const;

  /**
   * @brief Return the ground height at a requested location
   * @param[in] double x location
   * @param[in] double y location
   * @return double ground height at location [x,y]
   */
  double getGroundHeight(const double x, const double y) const;

  /**
   * @brief Return the surface normal at a requested location
   * @param[in] double x location
   * @param[in] double y location
   * @return std::array<double, 3> surface normal at location [x,y]
   */
  std::array<double, 3> getSurfaceNormal(const double x, const double y) const;

  /**
   * @brief Return the filtered ground height at a requested location
   * @param[in] double x location
   * @param[in] double y location
   * @return double ground height at location [x,y]
   */
  double getGroundHeightFiltered(const double x, const double y) const;

  /**
   * @brief Return the filtered surface normal at a requested location
   * @param[in] double x location
   * @param[in] double y location
   * @return std::array<double, 3> surface normal at location [x,y]
   */
  std::array<double, 3> getSurfaceNormalFiltered(const double x,
                                                 const double y) const;

  /**
   * @brief Return the filtered surface normal at a requested location
   * @param[in] double x location
   * @param[in] double y location
   * @return std::array<double, 3> surface normal at location [x,y]
   */
  Eigen::Vector3d getSurfaceNormalFilteredEigen(const double x,
                                                const double y) const;

  /**
   * @brief Return the (approximate) intersection of the height map and a
   * vector. Returned point lies exactly on the map but not entirely on the
   * vector.
   * @param[in] point The point at which the vector originates
   * @param[in] direction The direction along which to project the point
   */
  Eigen::Vector3d projectToMap(const Eigen::Vector3d point,
                               const Eigen::Vector3d direction);

  /**
   * @brief Return the vector of x_data of the map
   * @return std::vector<double> of x locations in the grid
   */
  std::vector<double> getXData() const;

  /**
   * @brief Return the vector of y_data of the map
   * @return std::vector<double> of y locations in the grid
   */
  std::vector<double> getYData() const;

  /**
   * @brief Determine if the map is empty
   * @return boolean for map emptiness (true = empty)
   */
  bool isEmpty() const;

 private:
  /**
   * @brief Return the x index
   * @param[in] x X location of the point
   * @return X index of location
   */
  inline int getXIndex(const double x) const {
    return std::max(
        std::min((int)floor((x - x_data_[0]) / x_diff_), x_size_ - 2), 0);
  }

  /**
   * @brief Return the y index
   * @param[in] y Y location of the point
   * @return Y index of location
   */
  inline int getYIndex(const double y) const {
    return std::max(
        std::min((int)floor((y - y_data_[0]) / y_diff_), y_size_ - 2), 0);
  }

  /// The number of elements in the x direction
  int x_size_ = 0;

  /// The number of elements in the y direction
  int y_size_ = 0;

  /// Distance between nodes in x
  double x_diff_;

  /// Distance between nodes in y
  double y_diff_;

  /// The vector of x data
  std::vector<double> x_data_;

  /// The vector of y data
  std::vector<double> y_data_;

  /// The nested vector of z data at each [x,y] location
  std::vector<std::vector<double>> z_data_;

  /// The nested vector of the x component of the gradient at each [x,y]
  /// location
  std::vector<std::vector<double>> nx_data_;

  /// The nested vector of the y component of the gradient at each [x,y]
  /// location
  std::vector<std::vector<double>> ny_data_;

  /// The nested vector of the z component of the gradient at each [x,y]
  /// location
  std::vector<std::vector<double>> nz_data_;

  /// The nested vector of filtered z data at each [x,y] location
  std::vector<std::vector<double>> z_data_filt_;

  /// The nested vector of the x component of the filtered gradient at each
  /// [x,y] location
  std::vector<std::vector<double>> nx_data_filt_;

  /// The nested vector of the y component of the filtered gradient at each
  /// [x,y] location
  std::vector<std::vector<double>> ny_data_filt_;

  /// The nested vector of the z component of the filtered gradient at each
  /// [x,y] location
  std::vector<std::vector<double>> nz_data_filt_;
};

#endif  // FAST_TERRAIN_MAP_H
```

```
#ifndef QUAD_MATH_UTILS_H
#define QUAD_MATH_UTILS_H

// Just include ros to access a bunch of other functions, fuck good code
#include <nav_msgs/Odometry.h>
#include <quad_msgs/MultiFootPlanContinuous.h>
#include <quad_msgs/MultiFootState.h>
#include <quad_msgs/RobotPlan.h>
#include <quad_msgs/RobotState.h>
#include <ros/ros.h>
#include <sensor_msgs/JointState.h>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>

#include <cmath>
#include <eigen3/Eigen/Eigen>

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

namespace math_utils {

/**
 * @brief Linearly interpolate data (a + t*(b-a)). DOES NOT CHECK FOR
 * EXTRAPOLATION.
 * @param[in] a
 * @param[in] b
 * @param[in] t
 * @return Double for interpolated value.
 */
inline double lerp(double a, double b, double t) { return (a + t * (b - a)); }

/**
 * @brief Wrap to [0,2*pi)
 * @param[in] val value to wrap
 * @return Wrapped value
 */
inline double wrapTo2Pi(double val) {
  return fmod(2 * M_PI + fmod(val, 2 * M_PI), 2 * M_PI);
}

/**
 * @brief Wrap to [-pi,pi)
 * @param[in] val value to wrap
 * @return Wrapped value
 */
inline double wrapToPi(double val) {
  return -M_PI + wrapTo2Pi(val + M_PI);
  // double new_val = fmod(val + M_PI, 2*M_PI);
  // while (new_val < 0) {
  //   new_val += 2*M_PI;
  // }
  // return new_val-M_PI;
}

/**
 * @brief Wrap data to [-pi,pi)
 * @param[in] data data to wrap
 * @return Wrapped data
 */
inline std::vector<double> wrapToPi(std::vector<double> data) {
  std::vector<double> data_wrapped = data;
  for (int i = 0; i < data.size(); i++) {
    data_wrapped[i] = wrapToPi(data[i]);
  }
  return data_wrapped;
}

/**
 * @brief Interpolate data from column vectors contained in a matrix (vector of
 * row vectors) provided an input vector and query point
 * @param[in] input_vec Input vector
 * @param[in] output_mat Collection of row vectors such that each row
 * corresponds to exactly one element in the input vector
 * @param[in] input_val Query point
 * @return Vector of interpolated values
 */
std::vector<double> interpMat(const std::vector<double> input_vec,
                              const std::vector<std::vector<double>> output_mat,
                              const double query_point);

/**
 * @brief Interpolate data from column vectors contained in a matrix (vector of
 * row vectors) provided an input vector and query point
 * @param[in] input_vec Input vector
 * @param[in] output_mat Collection of row vectors such that each row
 * corresponds to exactly one element in the input vector
 * @param[in] input_val Query point
 * @return Vector of interpolated values
 */
Eigen::Vector3d interpVector3d(const std::vector<double> input_vec,
                               const std::vector<Eigen::Vector3d> output_mat,
                               const double query_point);

/**
 * @brief Interpolate data from Eigen::Vector3d contained in a matrix (vector of
 * row vectors) provided an input vector and query point
 * @param[in] input_vec Input vector
 * @param[in] output_mat Collection of row vectors such that each row
 * corresponds to exactly one element in the input vector
 * @param[in] input_val Query point
 * @return Vector of interpolated values
 */
std::vector<Eigen::Vector3d> interpMatVector3d(
    const std::vector<double> input_vec,
    const std::vector<std::vector<Eigen::Vector3d>> output_mat,
    const double query_point);

/**
 * @brief Obtain the correct int within a parameterized vector of ints
 * @param[in] input_vec Input vector
 * @param[in] output_vec Output vector of ints
 * @param[in] input_val Query point
 * @return Correct output int corresponsing to the query point
 */
int interpInt(const std::vector<double> input_vec, std::vector<int> output_vec,
              const double query_point);

/**
 * @brief Filter a stl vector with a moving average window.
 * @param[in] data Input vector
 * @param[in] window_size the width of the moving window. If even, function will
 * add one to maintain symmetry
 * @return Vector of filtered values
 */
std::vector<double> movingAverageFilter(std::vector<double> data,
                                        int window_size);

/**
 * @brief Differentiate an input vector with the central difference method
 * @param[in] data Input vector
 * @param[in] dt The (constant) timestep between values in data.
 * @return Vector of differentiated signal
 */
std::vector<double> centralDiff(std::vector<double> data, double dt);

/**
 * @brief Unwrap a phase variable by filtering out differences > pi
 * @param[in] data Input vector containing a wrapped signal
 * @return Vector of unwrapped signal
 */
std::vector<double> unwrap(std::vector<double> data);

/**
 * @brief Selective damping least square matrix inverse
 * @param[in] jacobian Input matrix
 * @return Pseudo-inverse of the input matrix
 */
Eigen::MatrixXd sdlsInv(const Eigen::MatrixXd &jacobian);
}  // namespace math_utils

#endif  // QUAD_MATH_UTILS_H
```

```
#ifndef MATRIX_ALGEBRA_H
#define MATRIX_ALGEBRA_H

#include <eigen3/Eigen/Eigen>

namespace math {
/**
 * @brief Compute the Kronecker product. A composite array made of blocks of the
 second array scaled by the first

 * @param[in] m1 first matrix
 * @param[in] m2 second matrix
 *
 * @return A result of the Kronecker product
 */
Eigen::MatrixXd kron(const Eigen::MatrixXd &m1, const Eigen::MatrixXd &m2) {
  uint32_t m1r = m1.rows();
  uint32_t m1c = m1.cols();
  uint32_t m2r = m2.rows();
  uint32_t m2c = m2.cols();

  Eigen::MatrixXd m3(m1r * m2r, m1c * m2c);

  for (int i = 0; i < m1r; i++) {
    for (int j = 0; j < m1c; j++) {
      m3.block(i * m2r, j * m2c, m2r, m2c) = m1(i, j) * m2;
    }
  }

  return m3;
}

/**
 * @brief Create a block diagonal matrix from provided matrices 3 input version
 *
 * @param m1 first matrix
 * @param m2 second matrix
 *
 * @return Created a block diagonal matrix
 */
Eigen::MatrixXd block_diag(const Eigen::MatrixXd &m1,
                           const Eigen::MatrixXd &m2) {
  uint32_t m1r = m1.rows();
  uint32_t m1c = m1.cols();
  uint32_t m2r = m2.rows();
  uint32_t m2c = m2.cols();

  Eigen::MatrixXd mf = Eigen::MatrixXd::Zero(m1r + m2r, m1c + m2c);
  mf.block(0, 0, m1r, m1c) = m1;
  mf.block(m1r, m1c, m2r, m2c) = m2;

  return mf;
}

/**
 * @brief Create a block diagonal matrix from provided matrices 3 input version
 *
 * @param[in] m1 first matrix
 * @param[in] m2 second matrix
 * @param[in] m3 third matrix
 *
 * @return Created a block diagonal matrix
 */
Eigen::MatrixXd block_diag(const Eigen::MatrixXd &m1, const Eigen::MatrixXd &m2,
                           const Eigen::MatrixXd &m3) {
  uint32_t m1r = m1.rows();
  uint32_t m1c = m1.cols();
  uint32_t m2r = m2.rows();
  uint32_t m2c = m2.cols();
  uint32_t m3r = m3.rows();
  uint32_t m3c = m3.cols();

  Eigen::MatrixXd bdm = Eigen::MatrixXd::Zero(m1r + m2r + m3r, m1c + m2c + m3c);
  bdm.block(0, 0, m1r, m1c) = m1;
  bdm.block(m1r, m1c, m2r, m2c) = m2;
  bdm.block(m1r + m2r, m1c + m2c, m3r, m3c) = m3;

  return bdm;
}

/**
 * @brief  Gives a new shape to an array without changing its data.
 *
 * @param[in] x input matrix
 * @param[in] r the number of row elements
 * @param[in] c the number of collum elements
 *
 * @return The new shape matrix
 */
Eigen::MatrixXd reshape(Eigen::MatrixXd x, uint32_t r, uint32_t c) {
  Eigen::Map<Eigen::MatrixXd> rx(x.data(), r, c);

  return rx;
}
}  // namespace math
#endif
```

```
#ifndef QUAD_KD_H
#define QUAD_KD_H

#include <math.h>
#include <rbdl/addons/urdfreader/urdfreader.h>
#include <rbdl/rbdl.h>
#include <rbdl/rbdl_utils.h>
#include <ros/ros.h>
#include <tf2/LinearMath/Quaternion.h>

#include <Eigen/Geometry>
#include <chrono>
#include <grid_map_core/GridMap.hpp>
#include <random>
#include <vector>

#include "quad_utils/function_timer.h"
#include "quad_utils/math_utils.h"

namespace quad_utils {

//! A lightweight library for quad kinematic functions
/*!
  This library includes several functions and classes to aid in quad kinematic
  calculations. It relies on Eigen, as well as some MATLAB codegen for more
  complicated computations that would be a pain to write out by hand.
*/
class QuadKD {
 public:
  /**
   * @brief Constructor for QuadKD Class
   * @return Constructed object of type QuadKD
   */
  QuadKD();

  /**
   * @brief Constructor for QuadKD Class
   * @param[in] ns Namespace
   * @return Constructed object of type QuadKD
   */
  QuadKD(std::string ns);

  /**
   * @brief Initialize model for the class
   * @param[in] ns Namespace
   */
  void initModel(std::string ns);

  /**
   * @brief Create an Eigen Eigen::Matrix4d containing a homogeneous transform
   * from a specified translation and a roll, pitch, and yaw vector
   * @param[in] trans Translation from input frame to output frame
   * @param[in] rpy Rotation from input frame to output frame as roll, pitch,
   * yaw
   * @return Homogenous transformation matrix
   */
  Eigen::Matrix4d createAffineMatrix(Eigen::Vector3d trans,
                                     Eigen::Vector3d rpy) const;

  /**
   * @brief Create an Eigen Eigen::Matrix4d containing a homogeneous transform
   * from a specified translation and an AngleAxis object
   * @param[in] trans Translation from input frame to output frame
   * @param[in] rot Rotation from input frame to output frame as AngleAxis
   * @return Homogenous transformation matrix
   */
  Eigen::Matrix4d createAffineMatrix(Eigen::Vector3d trans,
                                     Eigen::AngleAxisd rot) const;

  /**
   * @brief Transform a transformation matrix from the body frame to the world
   * frame
   * @param[in] body_pos Position of center of body frame
   * @param[in] body_rpy Orientation of body frame in roll, pitch, yaw
   * @param[in] transform_body Specified transform in the body frame
   * @param[out] transform_world Specified transform in the world frame
   */
  void transformBodyToWorld(Eigen::Vector3d body_pos, Eigen::Vector3d body_rpy,
                            Eigen::Matrix4d transform_body,
                            Eigen::Matrix4d &transform_world) const;

  /**
   * @brief Transform a transformation matrix from the world frame to the body
   * frame
   * @param[in] body_pos Position of center of body frame
   * @param[in] body_rpy Orientation of body frame in roll, pitch, yaw
   * @param[in] transform_world Specified transform in the world frame
   * @param[out] transform_body Specified transform in the body frame
   */
  void transformWorldToBody(Eigen::Vector3d body_pos, Eigen::Vector3d body_rpy,
                            Eigen::Matrix4d transform_world,
                            Eigen::Matrix4d &transform_body) const;

  /**
   * @brief Compute forward kinematics for a specified leg from the body COM
   * @param[in] leg_index Quad leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] joint_state Joint states for the specified leg (abad, hip, knee)
   * @param[out] g_body_foot Transform of the specified foot in world frame
   */
  void bodyToFootFKBodyFrame(int leg_index, Eigen::Vector3d joint_state,
                             Eigen::Matrix4d &g_body_foot) const;

  /**
   * @brief Compute forward kinematics for a specified leg from the body COM
   * @param[in] leg_index Quad leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] joint_state Joint states for the specified leg (abad, hip, knee)
   * @param[out] foot_pos_world Position of the specified foot in world frame
   */
  void bodyToFootFKBodyFrame(int leg_index, Eigen::Vector3d joint_state,
                             Eigen::Vector3d &foot_pos_body) const;

  /**
   * @brief Compute forward kinematics for a specified leg
   * @param[in] leg_index Quad leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] body_pos Position of center of body frame
   * @param[in] body_rpy Orientation of body frame in roll, pitch, yaw
   * @param[in] joint_state Joint states for the specified leg (abad, hip, knee)
   * @param[out] g_world_foot Transform of the specified foot in world frame
   */
  void worldToFootFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
                               Eigen::Vector3d body_rpy,
                               Eigen::Vector3d joint_state,
                               Eigen::Matrix4d &g_world_foot) const;

  /**
   * @brief Compute forward kinematics for a specified leg
   * @param[in] leg_index Quad leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] body_pos Position of center of body frame
   * @param[in] body_rpy Orientation of body frame in roll, pitch, yaw
   * @param[in] joint_state Joint states for the specified leg (abad, hip, knee)
   * @param[out] foot_pos_world Position of the specified foot in world frame
   */
  void worldToFootFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
                               Eigen::Vector3d body_rpy,
                               Eigen::Vector3d joint_state,
                               Eigen::Vector3d &foot_pos_world) const;

  /**
   * @brief Compute forward kinematics for a specified leg
   * @param[in] leg_index Spirit leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] body_pos Position of center of body frame
   * @param[in] body_rpy Orientation of body frame in roll, pitch, yaw
   * @param[in] joint_state Joint states for the specified leg (abad, hip, knee)
   * @param[out] g_world_knee Transform of the specified knee in world frame
   */
  void worldToKneeFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
                               Eigen::Vector3d body_rpy,
                               Eigen::Vector3d joint_state,
                               Eigen::Matrix4d &g_world_knee) const;

  /**
   * @brief Compute forward kinematics for a specified leg
   * @param[in] leg_index Spirit leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] body_pos Position of center of body frame
   * @param[in] body_rpy Orientation of body frame in roll, pitch, yaw
   * @param[in] joint_state Joint states for the specified leg (abad, hip, knee)
   * @param[out] knee_pos_world Position of the specified knee in world frame
   */
  void worldToKneeFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
                               Eigen::Vector3d body_rpy,
                               Eigen::Vector3d joint_state,
                               Eigen::Vector3d &knee_pos_world) const;

  /**
   * @brief Compute inverse kinematics for a specified leg
   * @param[in] leg_index Quad leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] body_pos Position of center of body frame
   * @param[in] body_rpy Orientation of body frame in roll, pitch, yaw
   * @param[in] foot_pos_world Position of the specified foot in world frame
   * @param[out] joint_state Joint states for the specified leg (abad, hip,
   * knee)
   */
  bool worldToFootIKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
                               Eigen::Vector3d body_rpy,
                               Eigen::Vector3d foot_pos_world,
                               Eigen::Vector3d &joint_state) const;

  /**
   * @brief Compute inverse kinematics for a specified leg in the leg base frame
   * @param[in] leg_index Quad leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] foot_pos_legbase Position of the specified foot in leg base
   * frame
   * @param[out] joint_state Joint states for the specified leg (abad, hip,
   * knee)
   */
  bool legbaseToFootIKLegbaseFrame(int leg_index,
                                   Eigen::Vector3d foot_pos_legbase,
                                   Eigen::Vector3d &joint_state) const;

  /**
   * @brief Get the lower joint limit of a particular joint
   * @param[in] leg_index Quad leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] joint_index Index for joint (0 = abad, 1 = hip, 2 = knee)
   * @return Requested joint limit
   */
  double getJointLowerLimit(int leg_index, int joint_index) const;

  /**
   * @brief Get the upper joint limit of a particular joint
   * @param[in] leg_index Quad leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] joint_index Index for joint (0 = abad, 1 = hip, 2 = knee)
   * @return Requested joint limit
   */
  double getJointUpperLimit(int leg_index, int joint_index) const;

  /**
   * @brief Get the upper joint limit of a particular joint
   * @param[in] leg_index Quad leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] link_index Index for link (0 = abad, 1 = upper, 2 = lower)
   * @return Requested link length
   */
  double getLinkLength(int leg_index, int link_index) const;

  /**
   * @brief Get the transform from the world frame to the leg base
   * @param[in] leg_index Quad leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] body_pos Position of center of body frame
   * @param[in] body_rpy Orientation of body frame in roll, pitch, yaw
   * @param[out] g_world_legbase Transformation matrix of world to leg base
   */
  void worldToLegbaseFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
                                  Eigen::Vector3d body_rpy,
                                  Eigen::Matrix4d &g_world_legbase) const;

  /**
   * @brief Get the position of the leg base frame origin in the world frame
   * @param[in] leg_index Quad leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] body_pos Position of center of body frame
   * @param[in] body_rpy Orientation of body frame in roll, pitch, yaw
   * @param[out] leg_base_pos_world Origin of leg base frame in world frame
   */
  void worldToLegbaseFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
                                  Eigen::Vector3d body_rpy,
                                  Eigen::Vector3d &leg_base_pos_world) const;

  /**
   * @brief Get the position of the nominal hip location in the world frame
   * @param[in] leg_index Quad leg (0 = FL, 1 = BL, 2 = FR, 3 = BR)
   * @param[in] body_pos Position of center of body frame
   * @param[in] body_rpy Orientation of body frame in roll, pitch, yaw
   * @param[out] nominal_hip_pos_world Location of nominal hip in world frame
   */
  void worldToNominalHipFKWorldFrame(
      int leg_index, Eigen::Vector3d body_pos, Eigen::Vector3d body_rpy,
      Eigen::Vector3d &nominal_hip_pos_world) const;

  /**
   * @brief Compute Jacobian for generalized coordinates
   * @param[in] state Joint and body states
   * @param[out] jacobian Jacobian for generalized coordinates
   */
  void getJacobianGenCoord(const Eigen::VectorXd &state,
                           Eigen::MatrixXd &jacobian) const;

  /**
   * @brief Compute Jacobian for angular velocity in body frame
   * @param[in] state Joint and body states
   * @param[out] jacobian Jacobian for angular velocity in body frame
   */
  void getJacobianBodyAngVel(const Eigen::VectorXd &state,
                             Eigen::MatrixXd &jacobian) const;

  /**
   * @brief Compute Jacobian for angular velocity in world frame
   * @param[in] state Joint and body states
   * @param[out] jacobian Jacobian for angular velocity in world frame
   */
  void getJacobianWorldAngVel(const Eigen::VectorXd &state,
                              Eigen::MatrixXd &jacobian) const;

  /**
   * @brief Compute rotation matrix given roll pitch and yaw
   * @param[in] rpy Roll pitch and yaw
   * @param[out] rot Rotation matrix
   */
  void getRotationMatrix(const Eigen::VectorXd &rpy,
                         Eigen::Matrix3d &rot) const;

  /**
   * @brief Compute inverse dynamics for swing leg
   * @param[in] state_pos Position states
   * @param[in] state_vel Velocity states
   * @param[in] foot_acc Foot absolute acceleration in world frame
   * @param[in] grf Ground reaction force
   * @param[in] contact_mode Contact mode of the legs
   * @param[out] tau Joint torques
   */
  void 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;

  /**
   * @brief Convert centroidal model states (foot coordinates and grfs) to full
   * body (joints and torques)
   * @param[in] body_state Position states
   * @param[in] foot_positions Foot positions in the world frame
   * @param[in] foot_velocities Foot velocities in the world frame
   * @param[in] grfs Ground reaction forces
   * @param[out] joint_positions Joint positions
   * @param[out] joint_velocities Joint velocities
   * @param[out] tau Joint torques
   * @return boolean for exactness of kinematics
   */
  bool 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);

  /**
   * @brief Apply a uniform maximum torque to a given set of joint torques
   * @param[in] torques Joint torques. in Nm
   * @param[in] constrained_torques Joint torques after applying max, in Nm
   * @return Boolean to indicate if initial torques is feasible (checks if
   * torques == constrained_torques)
   */
  bool applyMotorModel(const Eigen::VectorXd &torques,
                       Eigen::VectorXd &constrained_torques);

  /**
   * @brief Apply a linear motor model to a given set of joint torques and
   * velocities
   * @param[in] torques Joint torques. in Nm
   * @param[in] joint_velocities Velocities of each joint. in rad/s
   * @param[in] constrained_torques Joint torques after applying motor model, in
   * Nm
   * @return Boolean to indicate if initial torques is feasible (checks if
   * torques == constrained_torques)
   */
  bool applyMotorModel(const Eigen::VectorXd &torques,
                       const Eigen::VectorXd &joint_velocities,
                       Eigen::VectorXd &constrained_torques);

  /**
   * @brief Check if state is valid
   * @param[in] body_state Robot body positions and velocities
   * @param[in] joint_state Joint positions and velocities
   * @param[in] torques Joint torques
   * @param[in] terrain Map of the terrain for collision checking
   * @return Boolean for state validity
   */
  bool isValidFullState(const Eigen::VectorXd &body_state,
                        const Eigen::VectorXd &joint_state,
                        const Eigen::VectorXd &torques,
                        const grid_map::GridMap &terrain,
                        Eigen::VectorXd &state_violation,
                        Eigen::VectorXd &control_violation);

  /**
   * @brief Check if state is valid
   * @param[in] body_state Robot body positions and velocities
   * @param[in] foot_positions Foot positions
   * @param[in] foot_velocities Foot velocities
   * @param[in] grfs Ground reaction forces in the world frame
   * @param[in] terrain Map of the terrain for collision checking
   * @return Boolean for state validity
   */
  bool 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 &torques,
      Eigen::VectorXd &state_violation, Eigen::VectorXd &control_violation);

  inline double getGroundClearance(const Eigen::Vector3d &point,
                                   const grid_map::GridMap &terrain) {
    grid_map::Position pos = {point.x(), point.y()};
    return (point.z() - terrain.atPosition("z", pos));
  }

 private:
  /// Number of feet
  const int num_feet_ = 4;

  /// Vector of the abad link lengths
  std::vector<double> l0_vec_;

  /// Upper link length
  double l1_;

  /// Lower link length
  double l2_;

  /// Abad offset from legbase
  Eigen::Vector3d abad_offset_;

  /// Knee offset from hip
  Eigen::Vector3d knee_offset_;

  /// Foot offset from knee
  Eigen::Vector3d foot_offset_;

  /// Vector of legbase offsets
  std::vector<Eigen::Vector3d> legbase_offsets_;

  /// Vector of legbase offsets
  std::vector<Eigen::Matrix4d> g_body_legbases_;

  /// Epsilon offset for joint bounds
  const double joint_eps = 0.1;

  /// Vector of the joint lower limits
  std::vector<std::vector<double>> joint_min_;

  /// Vector of the joint upper limits
  std::vector<std::vector<double>> joint_max_;

  RigidBodyDynamics::Model *model_;

  std::vector<std::string> body_name_list_;

  std::vector<unsigned int> body_id_list_;

  std::vector<int> leg_idx_list_;

  /// Abad max joint torque
  const double abad_tau_max_ = 21;

  /// Hip max joint torque
  const double hip_tau_max_ = 21;

  /// Knee max joint torque
  const double knee_tau_max_ = 32;

  /// Vector of max torques
  const Eigen::VectorXd tau_max_ =
      (Eigen::VectorXd(12) << abad_tau_max_, hip_tau_max_, knee_tau_max_,
       abad_tau_max_, hip_tau_max_, knee_tau_max_, abad_tau_max_, hip_tau_max_,
       knee_tau_max_, abad_tau_max_, hip_tau_max_, knee_tau_max_)
          .finished();

  /// Abad max joint velocity
  const double abad_vel_max_ = 37.7;

  /// Hip max joint velocity
  const double hip_vel_max_ = 37.7;

  /// Knee max joint velocity
  const double knee_vel_max_ = 25.1;

  /// Vector of max velocities
  const Eigen::VectorXd vel_max_ =
      (Eigen::VectorXd(12) << abad_vel_max_, hip_vel_max_, knee_vel_max_,
       abad_vel_max_, hip_vel_max_, knee_vel_max_, abad_vel_max_, hip_vel_max_,
       knee_vel_max_, abad_vel_max_, hip_vel_max_, knee_vel_max_)
          .finished();

  const Eigen::VectorXd mm_slope_ = tau_max_.cwiseQuotient(vel_max_);
};

}  // namespace quad_utils

#endif  // QUAD_KD_H
```

```
#ifndef TAIL_TYPE_H
#define TAIL_TYPE_H

enum Tail_type { NONE, CENTRALIZED, DISTRIBUTED, DECENTRALIZED };

#endif  // TAIL_TYPE_H
```

```
#ifndef TRAJECTORY_PUBLISHER_H
#define TRAJECTORY_PUBLISHER_H

#include <quad_msgs/RobotPlan.h>
#include <quad_msgs/RobotState.h>
#include <ros/ros.h>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include <visualization_msgs/MarkerArray.h>

#include <eigen3/Eigen/Eigen>

#include "quad_utils/function_timer.h"
#include "quad_utils/math_utils.h"
#include "quad_utils/quad_kd.h"
#include "quad_utils/ros_utils.h"

//! A class for publishing the current state of a trajectory
/*!
   TrajectoryPublisher is a class for publishing the current state of a given
   robot trajectory. It subscribes to a topic of type RobotPlan or can be
   customized to import data directly (such as from a .csv), then interpolates
   that trajectory to find the state at the current time and publishes it to the
   trajectory state topic.
*/
class TrajectoryPublisher {
 public:
  /**
   * @brief Constructor for TrajectoryPublisher Class
   * @param[in] nh ROS NodeHandle to publish and subscribe from
   * @return Constructed object of type TrajectoryPublisher
   */
  TrajectoryPublisher(ros::NodeHandle nh);

  /**
   * @brief Calls ros spinOnce and pubs data at set frequency
   */
  void spin();

 private:
  /**
   * @brief Import trajectory from external source (user implemented)
   */
  void importTrajectory();

  /**
   * @brief Callback function to handle new body plan data
   * @param[in] msg Body plan message contining interpolated output of body
   * planner
   */
  void robotPlanCallback(const quad_msgs::RobotPlan::ConstPtr& msg);

  /**
   * @brief Publish the current trajectory state
   */
  void publishTrajectoryState();

  /// ROS Subscriber for the body plan
  ros::Subscriber body_plan_sub_;

  /// ROS Publisher for the current trajectory state
  ros::Publisher trajectory_state_pub_;

  /// Nodehandle to pub to and sub from
  ros::NodeHandle nh_;

  /// Vector of body states to store the body plan
  quad_msgs::RobotPlan body_plan_msg_;

  /// Robot state message
  quad_msgs::RobotState::ConstPtr robot_state_msg_;

  /// Update rate for sending and receiving data
  double update_rate_;

  /// Handle for the map frame
  std::string map_frame_;
```

```cpp

  /// The source of the current trajectory (import or topic)
  std::string traj_source_;

  /// QuadKD class
  std::shared_ptr<quad_utils::QuadKD> quadKD_;
};

#endif  // TRAJECTORY_PUBLISHER_H
```

```
#ifndef SPIRIT_ROS_UTILS_H
#define SPIRIT_ROS_UTILS_H

#include <geometry_msgs/Point.h>
#include <geometry_msgs/Vector3.h>
#include <quad_utils/math_utils.h>
#include <ros/ros.h>
#include <std_msgs/Header.h>

namespace quad_utils {
/**
 * @brief Gets the relative age of a timestamped header
 * @param[in] header ROS Header that we wish to compute the age of
 * @param[in] t_compare ROS time we wish to compare to
 * @return Age in ms (compared to t_compare)
 */
inline double getROSMessageAgeInMs(std_msgs::Header header,
                                   ros::Time t_compare) {
  return (t_compare - header.stamp).toSec() * 1000.0;
}

/**
 * @brief Gets the relative age of a timestamped header
 * @param[in] header ROS Header that we wish to compute the age of
 * @return Age in ms (compared to ros::Time::now())
 */
inline double getROSMessageAgeInMs(std_msgs::Header header) {
  ros::Time t_compare = ros::Time::now();
  return quad_utils::getROSMessageAgeInMs(header, t_compare);
}

/**
 * @brief Gets the relative time (in s) since the beginning of the plan
 * @param[in] plan_start ROS Time to to compare to
 * @return Time in plan (compared to ros::Time::now())
 */
inline double getDurationSinceTime(ros::Time plan_start) {
  return (ros::Time::now() - plan_start).toSec();
}

/**
 * @brief Gets the index associated with a given time
 * @param[out] index Index in plan (compared to ros::Time::now())
 * @param[out] first_element_duration Time duration to next index in plan
 * (compared to ros::Time::now())
 * @param[in] plan_start ROS Time to to compare to
 * @param[in] dt Timestep used to discretize the plan
 */
inline void getPlanIndex(ros::Time plan_start, double dt, int &index,
                         double &first_element_duration) {
  double duration = getDurationSinceTime(plan_start);
  index = std::floor(duration / dt);
  first_element_duration = (index + 1) * dt - duration;
}

/**
 * @brief Load ros parameter into class variable
 * @param[in] nh ROS nodehandle
 * @param[in] paramName string storing key of param in rosparam server
 * @param[in] varName address of variable to store loaded param
 * @return boolean success
 */
template <class ParamType>
inline bool loadROSParam(ros::NodeHandle nh, std::string paramName,
                         ParamType &varName) {
  if (!nh.getParam(paramName, varName)) {
    ROS_ERROR("Can't find param %s from parameter server", paramName.c_str());
    return false;
  }
  return true;
}

/**
 * @brief Load ros parameter into class variable
 * @param[in] nh ROS nodehandle
 * @param[in] paramName string storing key of param in rosparam server
 * @param[in] varName address of variable to store loaded param
 * @param[in] defaultVal default value to use if rosparam server doesn't contain
 * key
 * @return boolean (true if found rosparam, false if loaded default)
 */
template <class ParamType>
inline bool loadROSParamDefault(ros::NodeHandle nh, std::string paramName,
                                ParamType &varName, ParamType defaultVal) {
  if (!nh.getParam(paramName, varName)) {
    varName = defaultVal;
    ROS_INFO("Can't find param %s on rosparam server, loading default value.",
             paramName.c_str());
    return false;
  }
  return true;
}

// /**
//  * @brief Interpolate two headers
//  * @param[out] msg State message to popluate
//  * @param[in] stamp Timestamp for the state message
//  * @param[in] frame Frame_id for the state message
//  */
// void updateStateHeaders(quad_msgs::RobotState &msg, ros::Time stamp,
// std::string frame);

/**
 * @brief Interpolate two headers
 * @param[out] msg State message to popluate
 * @param[in] stamp Timestamp for the state message
 * @param[in] frame Frame_id for the state message
 * @param[in] traj_index Trajectory index of this state message
 */
void updateStateHeaders(quad_msgs::RobotState &msg, ros::Time stamp,
                        std::string frame, int traj_index);

/**
 * @brief Interpolate two headers
 * @param[in] header_1 First header message
 * @param[in] header_2 Second header message
 * @param[in] t_interp Fraction of time between the messages [0,1]
 * @param[out] interp_state Interpolated header
 */
void interpHeader(std_msgs::Header header_1, std_msgs::Header header_2,
                  double t_interp, std_msgs::Header &interp_header);

/**
 * @brief Interpolate data between two Odometry messages.
 * @param[in] state_1 First Odometry message
 * @param[in] state_2 Second Odometry message
 * @param[in] t_interp Fraction of time between the messages [0,1]
 * @param[out] interp_state Interpolated Odometry message
 */
void interpOdometry(quad_msgs::BodyState state_1, quad_msgs::BodyState state_2,
                    double t_interp, quad_msgs::BodyState &interp_state);

/**
 * @brief Interpolate data between two JointState messages.
 * @param[in] state_1 First JointState message
 * @param[in] state_2 Second JointState message
 * @param[in] t_interp Fraction of time between the messages [0,1]
 * @param[out] interp_state Interpolated JointState message
 */
void interpJointState(sensor_msgs::JointState state_1,
                      sensor_msgs::JointState state_2, double t_interp,
                      sensor_msgs::JointState &interp_state);

/**
 * @brief Interpolate data between two FootState messages.
 * @param[in] state_1 First FootState message
 * @param[in] state_2 Second FootState message
 * @param[in] t_interp Fraction of time between the messages [0,1]
 * @param[out] interp_state Interpolated FootState message
 */
void interpMultiFootState(quad_msgs::MultiFootState state_1,
                          quad_msgs::MultiFootState state_2, double t_interp,
                          quad_msgs::MultiFootState &interp_state);

```

```
/**
 * @brief Interpolate data between two GRFArray messages.
 * @param[in] state_1 First GRFArray message
 * @param[in] state_2 Second GRFArray message
 * @param[in] t_interp Fraction of time between the messages [0,1]
 * @param[out] interp_state Interpolated GRFArray message
 */
void interpGRFArray(quad_msgs::GRFArray state_1, quad_msgs::GRFArray state_2,
                    double t_interp, quad_msgs::GRFArray &interp_state);

/**
 * @brief Interpolate data between two RobotState messages.
 * @param[in] state_1 First RobotState message
 * @param[in] state_2 Second RobotState message
 * @param[in] t_interp Fraction of time between the messages [0,1]
 * @param[out] interp_state Interpolated RobotState message
 */
void interpRobotState(quad_msgs::RobotState state_1,
                      quad_msgs::RobotState state_2, double t_interp,
                      quad_msgs::RobotState &interp_state);

/**
 * @brief Interpolate data from a BodyPlan message.
 * @param[in] msg BodyPlan message
 * @param[in] t Time since beginning of trajectory (will return last state if
 * too large)
 * @param[out] interp_state Interpolated Odometry message
 * @param[out] interp_primitive_id Interpolated primitive id
 * @param[out] interp_grf Interpolated GRF array
 */
void interpRobotPlan(quad_msgs::RobotPlan msg, double t,
                     quad_msgs::RobotState &interp_state,
                     int &interp_primitive_id, quad_msgs::GRFArray &interp_grf);

/**
 * @brief Interpolate data from a MultiFootPlanContinuous message.
 * @param[in] msg MultiFootPlanContinuous message
 * @param[in] t Time since beginning of trajectory (will return last state if
 * too large)
 * @return MultiFootState message
 */
quad_msgs::MultiFootState interpMultiFootPlanContinuous(
    quad_msgs::MultiFootPlanContinuous msg, double t);

// /**
//  * @brief Interpolate data from a robot state trajectory message.
//  * @param[in] msg robot state trajectory message
//  * @param[in] t Time since beginning of trajectory (will return last state if
//  * too large)
//  * @return Robot state message
//  */
// quad_msgs::RobotState interpRobotStateTraj(quad_msgs::RobotStateTrajectory
// msg,
//                                            double t);

/**
 * @brief Perform IK to compute a joint state message corresponding to body and
 * foot messages
 * @param[in] kinematics Pointer to kinematics object
 * @param[in] body_state message of body state
 * @param[in] multi_foot_state message of state of each foot
 * @param[out] joint_state message of the corresponding joint state
 */
void ikRobotState(const quad_utils::QuadKD &kinematics,
                  quad_msgs::BodyState body_state,
                  quad_msgs::MultiFootState multi_foot_state,
                  sensor_msgs::JointState &joint_state);

/**
 * @brief Perform IK and save to the state.joint field
 * @param[in] kinematics Pointer to kinematics object
 * @param[out] state RobotState message to which to add joint data
 */
void ikRobotState(const quad_utils::QuadKD &kinematics,
                  quad_msgs::RobotState &state);

/**
```

```
 * @brief Perform FK to compute a foot state message corresponding to body and
 * joint messages
 * @param[in] kinematics Pointer to kinematics object
 * @param[in] body_state message of body state
 * @param[in] joint_state message of the corresponding joint state
 * @param[out] multi_foot_state message of state of each foot
 */
void fkRobotState(const quad_utils::QuadKD &kinematics,
                  quad_msgs::BodyState body_state,
                  sensor_msgs::JointState joint_state,
                  quad_msgs::MultiFootState &multi_foot_state);

/**
 * @brief Perform FK and save to the state.feet field
 * @param[in] kinematics Pointer to kinematics object
 * @param[out] state RobotState message to which to add joint data
 */
void fkRobotState(const quad_utils::QuadKD &kinematics,
                  quad_msgs::RobotState &state);

/**
 * @brief Convert robot state message to Eigen
 * @param[in] state Eigen vector with body state data
 * @return Odometry msg with body state data
 */
quad_msgs::BodyState eigenToBodyStateMsg(const Eigen::VectorXd &state);

/**
 * @brief Convert robot state message to Eigen
 * @param[in] body Odometry msg with body state data
 * @return Eigen vector with body state data
 */
Eigen::VectorXd bodyStateMsgToEigen(const quad_msgs::BodyState &body);

/**
 * @brief Convert Eigen vector of GRFs to GRFArray msg
 * @param[in] grf_array Eigen vector with grf data in leg order
 * @param[in] multi_foot_state_msg MultiFootState msg containing foot position
 * information
 * @param[out] grf_msg GRFArray msg containing GRF data
 */
void eigenToGRFArrayMsg(Eigen::VectorXd grf_array,
                        quad_msgs::MultiFootState multi_foot_state_msg,
                        quad_msgs::GRFArray &grf_msg);

/**
 * @brief Convert GRFArray msg to Eigen vector of GRFs
 * @param[in] grf_array_msg_ GRFArray msg with grf data
 * @return grf_array Eigen vector with grf data in leg order
 */
Eigen::VectorXd grfArrayMsgToEigen(const quad_msgs::GRFArray &grf_array_msg_);

/**
 * @brief Convert robot foot state message to Eigen
 * @param[in] foot_state_msg MultiFootState msg containing foot position
 * information
 * @param[out] foot_position Eigen vector with foot position
 */
void footStateMsgToEigen(const quad_msgs::FootState &foot_state_msg,
                         Eigen::Vector3d &foot_position);

/**
 * @brief Convert robot multi foot state message to Eigen
 * @param[in] multi_foot_state_msg MultiFootState msg containing foot position
 * information
 * @param[out] foot_positions Eigen vector with foot state data
 */
void multiFootStateMsgToEigen(
    const quad_msgs::MultiFootState &multi_foot_state_msg,
    Eigen::VectorXd &foot_positions);

/**
 * @brief Convert robot multi foot state message to Eigen
 * @param[in] multi_foot_state_msg MultiFootState msg containing foot position
 * information
 * @param[out] foot_positions Eigen vector with foot position data
 * @param[out] foot_velocities Eigen vector with foot velocity data
 */
void multiFootStateMsgToEigen(
    const quad_msgs::MultiFootState &multi_foot_state_msg,
    Eigen::VectorXd &foot_positions, Eigen::VectorXd &foot_velocities);

/**
 * @brief Convert robot multi foot state message to Eigen
 * @param[in] multi_foot_state_msg MultiFootState msg containing foot position
 * information
 * @param[out] foot_positions Eigen vector with foot position data
 * @param[out] foot_velocities Eigen vector with foot velocity data
 * @param[out] foot_acceleration Eigen vector with foot acceleration data
 */
void multiFootStateMsgToEigen(
    const quad_msgs::MultiFootState &multi_foot_state_msg,
    Eigen::VectorXd &foot_positions, Eigen::VectorXd &foot_velocities,
    Eigen::VectorXd &foot_acceleration);

/**
 * @brief Convert eigen vectors to foot state messages
 * @param[in] foot_position Eigen vector with foot position data
 * @param[in] foot_velocity Eigen vector with foot velocity data
 * @param[out] foot_state_msg FootState msg containing foot position and
 * velocity data
 */
void eigenToFootStateMsg(Eigen::VectorXd foot_position,
                         Eigen::VectorXd foot_velocity,
                         quad_msgs::FootState &foot_state_msg);

/**
 * @brief Convert eigen vectors to foot state messages
 * @param[in] foot_position Eigen vector with foot position data
 * @param[in] foot_velocity Eigen vector with foot velocity data
 * @param[in] foot_acceleration Eigen vector with foot acceleration data
 * @param[out] foot_state_msg FootState msg containing foot position and
 * velocity data
 */
void eigenToFootStateMsg(Eigen::VectorXd foot_position,
                         Eigen::VectorXd foot_velocity,
                         Eigen::VectorXd foot_acceleration,
                         quad_msgs::FootState &foot_state_msg);

/**
 * @brief Convert eigen vector to stl vector
 * @param[in] eigen_vec Eigen vector with data
 * @param[out] vec stl vector
 */
void eigenToVector(const Eigen::VectorXd &eigen_vec, std::vector<double> &vec);

/**
 * @brief Convert stl vector to eigen vector
 * @param[in] vec stl vector
 * @param[out] eigen_vec Eigen vector with data
 */
void vectorToEigen(const std::vector<double> &vec, Eigen::VectorXd &eigen_vec);

/**
 * @brief Convert eigen vector to geometry_msgs::Vector3
 * @param[in] vec Eigen vector
 * @param[out] eigen_vec msg vector
 */
void Eigen3ToVector3Msg(const Eigen::Vector3d &eigen_vec,
                        geometry_msgs::Vector3 &vec);

/**
 * @brief Convert geometry_msgs::Vector3 vector to eigen vector
 * @param[in] vec msg vector
 * @param[out] eigen_vec Eigen vector
 */
void vector3MsgToEigen(const geometry_msgs::Vector3 &vec,
                       Eigen::Vector3d &eigen_vec);

/**
 * @brief Convert eigen vector to geometry_msgs::Point
 * @param[in] vec Eigen vector
 * @param[out] eigen_vec msg point
 */
```

```
void Eigen3ToPointMsg(const Eigen::Vector3d &eigen_vec,
                      geometry_msgs::Point &vec);

/**
 * @brief Convert geometry_msgs::Point vector to eigen vector
 * @param[in] vec msg point
 * @param[out] eigen_vec Eigen vector
 */
void pointMsgToEigen(const geometry_msgs::Point &vec,
                     Eigen::Vector3d &eigen_vec);
}  // namespace quad_utils

#endif
```

```
#ifndef FUNCTION_TIMER_H
#define FUNCTION_TIMER_H

#include <chrono>
#include <iostream>

namespace quad_utils {

//! A lightweight class for measuring and reporting the duration of functions
//! calls
/*!
  FunctionTimer keeps track of the amount of time elapsed between start and stop
  calls, and reporting this along with the name of the function. For some reason
  the logic in this class takes about 1e-7 s to run so timing functions faster
  than that will yield inaccurate solutions compared to standard steady clock
  methods. For functions that take longer than 1e-6 s it should work.
*/
class FunctionTimer {
 public:
  /**
   * @brief Constructor for FunctionTimer Class
   * @return Constructed object of type FunctionTimer
   */
  FunctionTimer(const char* function_name) {
    function_name_ = const_cast<char*>(function_name);
    start_time_ = std::chrono::steady_clock::now();
  }

  /**
   * @brief Report the statistics without printing to the terminal
   * @return Time in seconds
   */
  double reportSilent() {
    stop_time_ = std::chrono::steady_clock::now();
    std::chrono::duration<double> elapsed =
        std::chrono::duration_cast<std::chrono::duration<double>>(stop_time_ -
                                                                  start_time_);
    double current_time = elapsed.count();
    return current_time;
  }

  /**
   * @brief Report the statistics to the terminal
   */
  double reportStatistics() {
    double current_time = reportSilent();
    printf("Time spent in %s = %.2es\n", function_name_, current_time);
    return current_time;
  }

  /**
   * @brief Report the averaged statistics to the terminal over a given number
   * of iterations
   * @param[in] n Number of iterations executed during elapsed time (used for
   * averaging)
   */
  double reportStatistics(int n) {
    double avg_time = reportSilent() / n;
    printf("Average time spent in %s = %.2es\n", function_name_, avg_time);
    return avg_time;
  }

  /**
   * @brief Report the statistics to the terminal and restart the clock
   */
  void reportAndRestart() {
    reportStatistics();
    start_time_ = std::chrono::steady_clock::now();
  }

 private:
  /// The time at the start of the function call
  std::chrono::time_point<std::chrono::steady_clock> start_time_;

  /// The time at the which the report is queried
  std::chrono::time_point<std::chrono::steady_clock> stop_time_;

  /// Name of the function being timed
  char* function_name_;
};

}  // namespace quad_utils

#endif  // FUNCTION_TIMER_H
```