

```
1: #include <ros/ros.h>
2:
3: #include "quad_utils/mesh_to_grid_map_converter.hpp"
4:
5: // Standard C++ entry point
6: int main(int argc, char** argv) {
7:     // Announce this program to the ROS master
8:     ros::init(argc, argv, "mesh_to_grid_map");
9:     ros::NodeHandle nh;
10:    ros::NodeHandle private_nh("~");
11:
12:    // Creating the object to do the work.
13:    mesh_to_grid_map::MeshToGridMapConverter mesh_to_grid_map_converter(
14:        nh, private_nh);
15:
16:    ros::spin();
17:    return 0;
18: }
```

```
1: #include <quad_utils/math_utils.h>
2:
3: namespace math_utils {
4:
5: std::vector<double> interpMat(const std::vector<double> input_vec,
6:                             const std::vector<std::vector<double>> input_mat,
7:                             const double query_point) {
8:     // Check bounds, throw an error if invalid since this shouldn't ever happen
9:     if ((query_point < input_vec.front()) || (query_point > input_vec.back())) {
10:         throw std::runtime_error("Tried to interp out of bounds");
11:     }
12:
13:     // Declare variables for interpolating between, both for input and output data
14:     double t1, t2;
15:     std::vector<double> y1, y2, interp_data;
16:
17:     // Find the correct values to interp between
18:     for (int i = 0; i < input_vec.size(); i++) {
19:         if (input_vec[i] <= query_point && query_point < input_vec[i + 1]) {
20:             t1 = input_vec[i];
21:             t2 = input_vec[i + 1];
22:             y1 = input_mat[i];
23:             y2 = input_mat[i + 1];
24:             break;
25:         }
26:     }
27:
28:     // Apply linear interpolation for each element in the vector
29:     for (int i = 0; i < input_mat.front().size(); i++) {
30:         double result = y1[i] + (y2[i] - y1[i]) / (t2 - t1) * (query_point - t1);
31:         interp_data.push_back(result);
32:     }
33:
34:     return interp_data;
35: }
36:
37: Eigen::Vector3d interpVector3d(const std::vector<double> input_vec,
38:                               const std::vector<Eigen::Vector3d> input_mat,
39:                               const double query_point) {
40:     // Check bounds, throw an error if invalid since this shouldn't ever happen
41:     if ((query_point < input_vec.front()) || (query_point > input_vec.back())) {
42:         throw std::runtime_error("Tried to interp out of bounds");
43:     }
44:
45:     // Declare variables for interpolating between, both for input and output data
46:     double t1, t2;
47:     Eigen::Vector3d y1, y2, interp_data;
48:
49:     // Find the correct values to interp between
50:     for (int i = 0; i < input_vec.size(); i++) {
51:         if (input_vec[i] <= query_point && query_point < input_vec[i + 1]) {
52:             t1 = input_vec[i];
53:             t2 = input_vec[i + 1];
54:             y1 = input_mat[i];
55:             y2 = input_mat[i + 1];
56:             break;
57:         }
58:     }
59:
60:     // Apply linear interpolation for each element in the vector
61:     interp_data =
62:         y1.array() + (y2.array() - y1.array()) / (t2 - t1) * (query_point - t1);
63:
64:     return interp_data;
65: }
66:
67: std::vector<Eigen::Vector3d> interpMatVector3d(
68:     const std::vector<double> input_vec,
69:     const std::vector<std::vector<Eigen::Vector3d>> output_mat,
70:     const double query_point) {
71:     // Check bounds, throw an error if invalid since this shouldn't ever happen
72:     if ((query_point < input_vec.front()) || (query_point > input_vec.back())) {
73:         throw std::runtime_error("Tried to interp out of bounds");
74:     }
75:
76:     // Declare variables for interpolating between, both for input and output data
77:     double t1, t2;
```

```
78:     std::vector<Eigen::Vector3d> y1, y2, interp_data;
79:
80:     // Find the correct values to interp between
81:     for (int i = 0; i < input_vec.size(); i++) {
82:         if (input_vec[i] <= query_point && query_point < input_vec[i + 1]) {
83:             t1 = input_vec[i];
84:             t2 = input_vec[i + 1];
85:             y1 = output_mat[i];
86:             y2 = output_mat[i + 1];
87:             break;
88:         }
89:     }
90:
91:     // Apply linear interpolation for each element in the vector
92:     for (int i = 0; i < output_mat.front().size(); i++) {
93:         Eigen::Vector3d interp_eigen_vec;
94:         interp_eigen_vec = y1[i].array() + (y2[i].array() - y1[i].array()) /
95:             (t2 - t1) * (query_point - t1);
96:         interp_data.push_back(interp_eigen_vec);
97:     }
98:
99:     return interp_data;
100: }
101:
102: int interpInt(const std::vector<double> input_vec, std::vector<int> output_vec,
103:             const double query_point) {
104:     // Check bounds, throw an error if invalid since this shouldn't ever happen
105:     if ((query_point < input_vec.front()) || (query_point > input_vec.back())) {
106:         throw std::runtime_error("Tried to interp out of bounds");
107:     }
108:
109:     // Declare variables for interpolating between, both for input and output data
110:     double t1, t2;
111:     Eigen::Vector3d y1, y2, interp_data;
112:
113:     // Find the correct values to interp between
114:     int idx = 0;
115:     for (int i = 0; i < input_vec.size(); i++) {
116:         if (input_vec[i] <= query_point && query_point < input_vec[i + 1]) {
117:             return output_vec[i];
118:         }
119:     }
120:
121:     throw std::runtime_error("Didn't find the query point, something happened");
122: }
123:
124: std::vector<double> movingAverageFilter(std::vector<double> data,
125:                                       int window_size) {
126:     std::vector<double> filtered_data;
127:     int N = data.size();
128:
129:     // Check to ensure window size is an odd integer, if not add one to make it so
130:     if ((window_size % 2) == 0) {
131:         window_size += 1;
132:         ROS_WARN_THROTTLE(
133:             0.5, "Filter window size is even, adding one to maintain symmetry");
134:     }
135:
136:     // Make sure that the window size is acceptable
137:     if (window_size >= N) {
138:         ROS_WARN_THROTTLE(0.5, "Filter window size is bigger than data");
139:     }
140:
141:     // Loop through the data
142:     for (int i = 0; i < N; i++) {
143:         // Initialize sum and count of data samples
144:         double sum = 0;
145:         double count = 0;
146:
147:         // Shrink the window size if it would result in out of bounds data
148:         int current_window_size = std::min(window_size, 2 * i + 1);
149:         // int current_window_size = window_size;
150:
151:         // Loop through the window, adding to the sum and averaging
152:         for (int j = 0; j < current_window_size; j++) {
153:             double index = i + (j - (current_window_size - 1) / 2);
154:         }
155:     }
156: }
```

```
155:         // Make sure data is in bounds
156:         if (index >= 0 && index < N) {
157:             sum += data[index];
158:             count += 1;
159:         }
160:     }
161:
162:     filtered_data.push_back((float)sum / count);
163: }
164:
165: return filtered_data;
166: }
167:
168: std::vector<double> centralDiff(std::vector<double> data, double dt) {
169:     std::vector<double> data_diff;
170:
171:     for (int i = 0; i < data.size(); i++) {
172:         // Compute lower and upper indices, with forward/backward difference at the
173:         // ends
174:         int lower_index = std::max(i - 1, 0);
175:         int upper_index = std::min(i + 1, (int)data.size() - 1);
176:
177:         double estimate = (data[upper_index] - data[lower_index]) /
178:             (dt * (upper_index - lower_index));
179:         data_diff.push_back(estimate);
180:     }
181:
182:     return data_diff;
183: }
184:
185: std::vector<double> unwrap(std::vector<double> data) {
186:     // Generally to make yaw angle continuous
187:     std::vector<double> data_unwrapped = data;
188:     for (int i = 1; i < data.size(); i++) {
189:         double diff = data[i] - data[i - 1];
190:         // std::cout << "diff: " << diff*180/M_PI << std::endl;
191:         if (diff > M_PI) {
192:             // std::cout << "data[" << i << "]: " << data[i] << "\tdata[" << i-1 <<
193:             // "]: " << data[i-1] << std::endl; std::cout << "diff > M_PI @ " << i <<
194:             // " | diff: " << diff*180/M_PI << std::endl;
195:             for (int j = i; j < data.size(); j++) {
196:                 data_unwrapped[j] = data_unwrapped[j] - 2 * M_PI;
197:                 // std::cout << "data_unwrapped[" << j << "]: " <<
198:                 // data_unwrapped[j]*180/M_PI << std::endl;
199:             }
200:         } else if (diff < -M_PI) {
201:             // std::cout << "data[" << i << "]: " << data[i] << "\tdata[" << i-1 <<
202:             // "]: " << data[i-1] << std::endl; std::cout << "diff < -M_PI @ " << i <<
203:             // " | diff: " << diff*180/M_PI << std::endl;
204:             for (int j = i; j < data.size(); j++) {
205:                 data_unwrapped[j] = data_unwrapped[j] + 2 * M_PI;
206:                 // std::cout << "data_unwrapped[" << j << "]: " <<
207:                 // data_unwrapped[j]*180/M_PI << std::endl;
208:             }
209:         }
210:     }
211:
212:     return data_unwrapped;
213: }
214:
215: Eigen::MatrixXd sdsInv(const Eigen::MatrixXd &jacobian) {
216:     Eigen::JacobiSVD<Eigen::MatrixXd> svd(
217:         jacobian, Eigen::ComputeThinU | Eigen::ComputeThinV);
218:
219:     Eigen::VectorXd sig = svd.singularValues();
220:     Eigen::VectorXd sig_inv = sig;
221:
222:     for (size_t i = 0; i < sig.size(); i++) {
223:         sig_inv(i) = std::min(1 / sig(i), 1 / (1e-1 * sig.maxCoeff()));
224:     }
225:
226:     Eigen::MatrixXd jacobian_inv =
227:         svd.matrixV() * sig_inv.asDiagonal() * svd.matrixU().transpose();
228:
229:     return jacobian_inv;
230: }
231: } // namespace math_utils
```

232:

```
1: #include <ros/ros.h>
2:
3: #include "quad_utils/terrain_map_publisher.h"
4:
5: int main(int argc, char** argv) {
6:     ros::init(argc, argv, "terrain_map_publisher_node");
7:     ros::NodeHandle nh;
8:
9:     TerrainMapPublisher terrain_map_publisher(nh);
10:    terrain_map_publisher.spin();
11:
12:    return 0;
13: }
```

```
1: #include <grid_map_msgs/GridMap.h>
2: #include <pcl/io/vtk_lib_io.h>
3: #include <pcl_conversions/pcl_conversions.h>
4: #include <pcl_ros/point_cloud.h>
5:
6: #include <grid_map_pcl/GridMapPclConverter.hpp>
7: #include <grid_map_ros/grid_map_ros.hpp>
8: #include <quad_utils/mesh_to_grid_map_converter.hpp>
9: namespace mesh_to_grid_map {
10:
11: MeshToGridMapConverter::MeshToGridMapConverter(ros::NodeHandle nh,
12:                                                ros::NodeHandle nh_private)
13:     : nh_(nh),
14:       nh_private_(nh_private),
15:       grid_map_resolution_(kDefaultGridMapResolution),
16:       layer_name_(kDefaultLayerName),
17:       latch_grid_map_pub_(kDefaultLatchGridMapPub),
18:       verbose_(kDefaultVerbose),
19:       frame_id_mesh_loaded_(kDefaultFrameIdMeshLoaded),
20:       world_name_(kDefaultWorldName) {
21:     // Initial interaction with ROS
22:     subscribeToTopics();
23:     advertiseTopics();
24:     advertiseServices();
25:     getParametersFromRos();
26:
27:     std::string package_path = ros::package::getPath("gazebo_scripts");
28:     std::string full_path =
29:         package_path + "/worlds/" + world_name_ + "/" + world_name_ + ".ply";
30:
31:     std::cout << full_path << std::endl;
32:     bool success = loadMeshFromFile(full_path);
33: }
34:
35: void MeshToGridMapConverter::subscribeToTopics() {
36:     mesh_sub_ =
37:         nh_.subscribe("mesh", 10, &MeshToGridMapConverter::meshCallback, this);
38: }
39:
40: void MeshToGridMapConverter::advertiseTopics() {
41:     grid_map_pub_ = nh_.advertise<grid_map_msgs::GridMap>("terrain_map_raw", 1,
42:                                                            latch_grid_map_pub_);
43: }
44:
45: void MeshToGridMapConverter::advertiseServices() {
46:     save_grid_map_srv_ = nh_private_.advertiseService(
47:         "save_grid_map_to_file", &MeshToGridMapConverter::saveGridMapService,
48:         this);
49:     load_map_service_server_ = nh_private_.advertiseService(
50:         "load_mesh_from_file", &MeshToGridMapConverter::loadMeshService, this);
51: }
52:
53: void MeshToGridMapConverter::getParametersFromRos() {
54:     nh_private_.param("grid_map_resolution", grid_map_resolution_,
55:                      grid_map_resolution_);
56:     nh_private_.param("layer_name", layer_name_, layer_name_);
57:     nh_private_.param("latch_grid_map_pub", latch_grid_map_pub_,
58:                      latch_grid_map_pub_);
59:     nh_private_.param("verbose", verbose_, verbose_);
60:     nh_private_.param("frame_id_mesh_loaded", frame_id_mesh_loaded_,
61:                      frame_id_mesh_loaded_);
62:     nh_private_.param("world", world_name_, world_name_);
63: }
64:
65: void MeshToGridMapConverter::meshCallback(
66:     const pcl_msgs::PolygonMesh& mesh_msg) {
67:     if (verbose_) {
68:         ROS_INFO("Mesh received, starting conversion.");
69:     }
70:
71:     // Converting from message to an object
72:     pcl::PolygonMesh polygon_mesh;
73:     pcl_conversions::toPCL(mesh_msg, polygon_mesh);
74:     meshToGridMap(polygon_mesh, mesh_msg.header.frame_id,
75:                  mesh_msg.header.stamp.toNSec());
76: }
77: }
```

```
78: bool MeshToGridMapConverter::meshToGridMap(  
79:     const pcl::PolygonMesh& polygon_mesh, const std::string& mesh_frame_id,  
80:     const uint64_t& time_stamp_nano_seconds) {  
81:     // Creating the grid map  
82:     grid_map::GridMap map;  
83:     map.setFrameId(mesh_frame_id);  
84:  
85:     // Converting  
86:     grid_map::GridMapPclConverter::initializeFromPolygonMesh(  
87:         polygon_mesh, grid_map_resolution_, map);  
88:     const std::string layer_name(layer_name_);  
89:     grid_map::GridMapPclConverter::addLayerFromPolygonMesh(polygon_mesh,  
90:                                                             layer_name, map);  
91:  
92:     // Setup x and y matrices for loading  
93:     grid_map::Size map_size = map.getSize();  
94:     Eigen::MatrixXf x_data(map_size(0), map_size(1)),  
95:         y_data(map_size(0), map_size(1));  
96:  
97:     // Iterate through map to retrieve x and y data and save to data matrices  
98:     for (grid_map::GridMapIterator iterator(map); !iterator.isPastEnd();  
99:         ++iterator) {  
100:         const grid_map::Index index(*iterator);  
101:         grid_map::Position pos;  
102:         map.getPosition(index, pos);  
103:         x_data(index(0), index(1)) = pos(0);  
104:         y_data(index(0), index(1)) = pos(1);  
105:     }  
106:  
107:     // Add x and y layers to map  
108:     map.add("x", x_data);  
109:     map.add("y", y_data);  
110:  
111:     // Printing some debug info about the mesh and the map  
112:     if (verbose_) {  
113:         ROS_INFO_STREAM("Number of polygons: " << polygon_mesh.polygons.size());  
114:         ROS_INFO("Created map with size %f x %f m (%i x %i cells).",  
115:                 map.getLength().x(), map.getLength().y(), map.getSize()(0),  
116:                 map.getSize()(1));  
117:     }  
118:  
119:     // Publish grid map.  
120:     map.setTimestamp(time_stamp_nano_seconds);  
121:     grid_map_msgs::GridMap message;  
122:     grid_map::GridMapRosConverter::toMessage(map, message);  
123:  
124:     // Publishing the grid map message.  
125:     grid_map_pub_.publish(message);  
126:     if (verbose_) {  
127:         ROS_INFO("Published a grid map message.");  
128:     }  
129:  
130:     // Saving the gridmap to the object  
131:     last_grid_map_ptr_.reset(new grid_map::GridMap(map));  
132:  
133:     return true;  
134: }  
135:  
136: bool MeshToGridMapConverter::saveGridMapService(  
137:     grid_map_msgs::ProcessFile::Request& request,  
138:     grid_map_msgs::ProcessFile::Response& response) {  
139:     // Check there's actually a grid map saved  
140:     if (!last_grid_map_ptr_) {  
141:         ROS_ERROR("No grid map produced yet to save.");  
142:         response.success = static_cast<unsigned char>(false);  
143:     } else {  
144:         response.success = static_cast<unsigned char>(saveGridMap(  
145:             *last_grid_map_ptr_, request.file_path, request.topic_name));  
146:     }  
147:  
148:     return true;  
149: }  
150:  
151: bool MeshToGridMapConverter::saveGridMap(const grid_map::GridMap& map,  
152:     const std::string& path_to_file,  
153:     const std::string& topic_name) {  
154:     std::string topic_name_checked = topic_name;
```



```
155:     if (topic_name.empty()) {
156:         ROS_WARN(
157:             "Specified topic name is an empty string, default layer name will be "
158:             "used as topic name.");
159:         topic_name_checked = layer_name_;
160:     }
161:     // Saving the map
162:     if (!path_to_file.empty()) {
163:         if (verbose_) {
164:             ROS_INFO(
165:                 "Saved the grid map message to file: '%s', with topic name: '%s'.",
166:                 path_to_file.c_str(), topic_name_checked.c_str());
167:         }
168:         grid_map::GridMapRosConverter::saveToBag(map, path_to_file,
169:             topic_name_checked);
170:     } else {
171:         ROS_ERROR("No rosbag filepath specified where to save grid map.");
172:         return false;
173:     }
174:     return true;
175: }
176:
177: bool MeshToGridMapConverter::loadMeshService(
178:     grid_map_msgs::ProcessFile::Request& request,
179:     grid_map_msgs::ProcessFile::Response& response) {
180:     if (!request.topic_name.empty()) {
181:         ROS_WARN("Field 'topic_name' in service request will not be used.");
182:     }
183:     response.success =
184:         static_cast<unsigned char>(loadMeshFromFile(request.file_path));
185:     return true;
186: }
187:
188: bool MeshToGridMapConverter::loadMeshFromFile(
189:     const std::string& path_to_mesh_to_load) {
190:     if (path_to_mesh_to_load.empty()) {
191:         ROS_ERROR(
192:             "File path for mesh to load is empty. Please specify a valid path.");
193:         return false;
194:     }
195:
196:     pcl::PolygonMesh mesh_from_file;
197:     pcl::io::loadPolygonFilePLY(path_to_mesh_to_load, mesh_from_file);
198:
199:     if (mesh_from_file.polygons.empty()) {
200:         ROS_ERROR("Mesh read from file is empty!");
201:         return false;
202:     }
203:
204:     bool mesh_converted = meshToGridMap(mesh_from_file, frame_id_mesh_loaded_,
205:         ros::Time::now().toNSec());
206:     if (!mesh_converted) {
207:         ROS_ERROR("It was not possible to convert loaded mesh to grid_map object.");
208:         return false;
209:     }
210:
211:     if (verbose_) {
212:         ROS_INFO("Loaded the mesh from file: %s. Its frame_id is set to '%s'",
213:             path_to_mesh_to_load.c_str(), frame_id_mesh_loaded_.c_str());
214:     }
215:
216:     return true;
217: }
218:
219: } // namespace mesh_to_grid_map
220:
```

```

1: #include "quad_utils/remote_heartbeat.h"
2:
3: RemoteHeartbeat::RemoteHeartbeat(ros::NodeHandle nh) {
4:     nh_ = nh;
5:
6:     // Load rosparam from parameter server
7:     std::string remote_heartbeat_topic, robot_heartbeat_topic, leg_control_topic;
8:     quad_utils::loadROSParam(nh_, "/topics/heartbeat/remote",
9:                             remote_heartbeat_topic);
10:    quad_utils::loadROSParam(nh_, "topics/heartbeat/robot",
11:                             robot_heartbeat_topic);
12:    quad_utils::loadROSParam(nh_, "remote_heartbeat/robot_latency_threshold_warn",
13:                             robot_latency_threshold_warn_);
14:    quad_utils::loadROSParam(nh_,
15:                             "remote_heartbeat/robot_latency_threshold_error",
16:                             robot_latency_threshold_error_);
17:    quad_utils::loadROSParam(nh_, "remote_heartbeat/update_rate", update_rate_);
18:
19:    // Setup pub
20:    remote_heartbeat_pub_ =
21:        nh_.advertise<std_msgs::Header>(remote_heartbeat_topic, 1);
22:    robot_heartbeat_sub_ = nh_.subscribe(
23:        robot_heartbeat_topic, 1, &RemoteHeartbeat::robotHeartbeatCallback, this);
24: }
25:
26: void RemoteHeartbeat::robotHeartbeatCallback(
27:     const std_msgs::Header::ConstPtr& msg) {
28:     // Get the current time and compare to the message time
29:     double last_robot_heartbeat_time = msg->stamp.toSec();
30:     double t_now = ros::Time::now().toSec();
31:     double t_latency = t_now - last_robot_heartbeat_time;
32:
33:     // ROS_INFO_THROTTLE(1.0, "Robot latency = %6.4fs", t_latency);
34:
35:     if (abs(t_latency) >= robot_latency_threshold_warn_) {
36:         // ROS_WARN_THROTTLE(1.0, "Robot latency = %6.4fs which exceeds the warning
37:         // threshold of %6.4fs\n",
38:         //     t_latency, robot_latency_threshold_warn_);
39:     }
40:
41:     if (abs(t_latency) >= robot_latency_threshold_error_) {
42:         // ROS_ERROR("Robot latency = %6.4fs which exceeds the maximum threshold of
43:         // %6.4fs, "
44:         //     "killing remote heartbeat\n", t_latency,
45:         //     robot_latency_threshold_error_);
46:         // throw std::runtime_error("Shutting down remote heartbeat");
47:     }
48: }
49:
50: void RemoteHeartbeat::spin() {
51:     ros::Rate r(update_rate_);
52:     while (ros::ok()) {
53:         std_msgs::Header msg;
54:         msg.stamp = ros::Time::now();
55:         remote_heartbeat_pub_.publish(msg);
56:
57:         // Enforce update rate
58:         ros::spinOnce();
59:         r.sleep();
60:     }
61: }

```

```
1: #include <ros/ros.h>
2:
3: #include "quad_utils/remote_heartbeat.h"
4:
5: int main(int argc, char** argv) {
6:     ros::init(argc, argv, "remote_heartbeat_node");
7:     ros::NodeHandle nh;
8:
9:     RemoteHeartbeat remote_heartbeat(nh);
10:    remote_heartbeat.spin();
11:
12:    return 0;
13: }
```

```

1: #include "quad_utils/terrain_map_publisher.h"
2:
3: TerrainMapPublisher::TerrainMapPublisher(ros::NodeHandle nh)
4: : terrain_map_(grid_map::GridMap(
5:   {"z", "nx", "ny", "nz", "z_filt", "nx_filt", "ny_filt", "nz_filt"})) {
6:   nh_ = nh;
7:
8:   // Load rosparms from parameter server
9:   std::string terrain_map_topic, image_topic;
10:
11:   nh.param<std::string>("topics/terrain_map_raw", terrain_map_topic,
12:     "/terrain_map_raw");
13:   nh.param<std::string>("/map_frame", map_frame_, "map");
14:   nh.param<double>("terrain_map_publisher/update_rate", update_rate_, 10);
15:   nh.param<double>("terrain_map_publisher/obstacle_x", obstacle_.x, 2.0);
16:   nh.param<double>("terrain_map_publisher/obstacle_y", obstacle_.y, 0.0);
17:   nh.param<double>("terrain_map_publisher/obstacle_height", obstacle_.height,
18:     0.5);
19:   nh.param<double>("terrain_map_publisher/obstacle_radius", obstacle_.radius,
20:     1.0);
21:   nh.param<double>("terrain_map_publisher/step1_x", step1_.x, 4.0);
22:   nh.param<double>("terrain_map_publisher/step1_height", step1_.height, 0.3);
23:   nh.param<double>("terrain_map_publisher/step2_x", step2_.x, 4.0);
24:   nh.param<double>("terrain_map_publisher/step2_height", step2_.height, 0.3);
25:   nh.param<double>("terrain_map_publisher/resolution", resolution_, 0.2);
26:   nh.param<double>("terrain_map_publisher/update_rate", update_rate_, 10);
27:   nh.param<std::string>("terrain_map_publisher/map_data_source",
28:     map_data_source_, "internal");
29:   nh.param<std::string>("terrain_map_publisher/terrain_type", terrain_type_,
30:     "slope");
31:   // Setup pubs and subs
32:   terrain_map_pub_ =
33:     nh_.advertise<grid_map_msgs::GridMap>(terrain_map_topic, 1);
34:
35:   // Add image subscriber if data source requests an image
36:   if (map_data_source_.compare("image") == 0) {
37:     nh_.param<std::string>("topics/image", image_topic,
38:       "/image_publisher/image");
39:     nh_.param<double>("terrain_map_publisher/min_height", min_height_, 0.0);
40:     nh_.param<double>("terrain_map_publisher/max_height", max_height_, 1.0);
41:     image_sub_ = nh_.subscribe(image_topic, 1,
42:       &TerrainMapPublisher::loadMapFromImage, this);
43:   }
44:
45:   // Initialize the elevation layer on the terrain map
46:   terrain_map_.setBasicLayers(
47:     {"z", "nx", "ny", "nz", "z_filt", "nx_filt", "ny_filt", "nz_filt"});
48: }
49:
50: void TerrainMapPublisher::updateParams() {
51:   nh_.param<double>("terrain_map_publisher/obstacle_x", obstacle_.x, 2.0);
52:   nh_.param<double>("terrain_map_publisher/obstacle_y", obstacle_.y, 0.0);
53:   nh_.param<double>("terrain_map_publisher/obstacle_height", obstacle_.height,
54:     0.5);
55:   nh_.param<double>("terrain_map_publisher/obstacle_radius", obstacle_.radius,
56:     1.0);
57:   nh_.param<double>("terrain_map_publisher/step1_x", step1_.x, 4.0);
58:   nh_.param<double>("terrain_map_publisher/step1_height", step1_.height, 0.3);
59:   nh_.param<double>("terrain_map_publisher/step2_x", step2_.x, 6.0);
60:   nh_.param<double>("terrain_map_publisher/step2_height", step2_.height, -0.3);
61: }
62:
63: void TerrainMapPublisher::createMap() {
64:   // Set initial map parameters and geometry
65:   terrain_map_.setFrameId(map_frame_);
66:   terrain_map_.setGeometry(
67:     grid_map::Length(24.0, 12.0), resolution_,
68:     grid_map::Position(-0.5 * resolution_, -0.5 * resolution_));
69:   ROS_INFO("Created map with size %f x %f m (%i x %i cells).",
70:     terrain_map_.getLength().x(), terrain_map_.getLength().y(),
71:     terrain_map_.getSize()(0), terrain_map_.getSize()(1));
72: }
73:
74: void TerrainMapPublisher::updateMap() {
75:   // Add terrain info
76:   for (grid_map::GridMapIterator it(terrain_map_); !it.isPastEnd(); ++it) {
77:     grid_map::Position position;

```

```
78:     terrain_map_.getPosition(*it, position);
79:     double x_diff = position.x() - obstacle_.x;
80:     double y_diff = position.y() - obstacle_.y;
81:
82:     if (x_diff * x_diff + y_diff * y_diff <=
83:         obstacle_.radius * obstacle_.radius) {
84:         terrain_map_.at("z", *it) = obstacle_.height;
85:         terrain_map_.at("z_filt", *it) = obstacle_.height;
86:     } else {
87:         terrain_map_.at("z", *it) = 0.0;
88:         terrain_map_.at("z_filt", *it) = 0.0;
89:     }
90:
91:     if (position.x() >= step1_.x) {
92:         terrain_map_.at("z", *it) += step1_.height;
93:         terrain_map_.at("z_filt", *it) += step1_.height;
94:     }
95:
96:     if (position.x() >= step2_.x) {
97:         terrain_map_.at("z", *it) += step2_.height;
98:         terrain_map_.at("z_filt", *it) += step2_.height;
99:     }
100:
101:     terrain_map_.at("nx", *it) = 0.0;
102:     terrain_map_.at("ny", *it) = 0.0;
103:     terrain_map_.at("nz", *it) = 1.0;
104:
105:     terrain_map_.at("nx_filt", *it) = 0.0;
106:     terrain_map_.at("ny_filt", *it) = 0.0;
107:     terrain_map_.at("nz_filt", *it) = 1.0;
108: }
109: }
110:
111: std::vector<std::vector<double>> TerrainMapPublisher::loadCSV(
112:     std::string filename) {
113:     std::vector<std::vector<double>> data;
114:     std::ifstream inputFile(filename);
115:     int l = 0;
116:
117:     while (inputFile) {
118:         l++;
119:         std::string s;
120:         if (!getline(inputFile, s)) break;
121:         if (s[0] != '#') {
122:             std::istringstream ss(s);
123:             std::vector<double> record;
124:
125:             while (ss) {
126:                 std::string line;
127:                 if (!getline(ss, line, ',')) break;
128:                 try {
129:                     record.push_back(stod(line));
130:                 } catch (const std::invalid_argument e) {
131:                     std::cout << "NaN found in file " << filename << " line " << l
132:                         << std::endl;
133:                     e.what();
134:                 }
135:             }
136:
137:             data.push_back(record);
138:         }
139:     }
140:
141:     if (!inputFile.eof()) {
142:         std::cerr << "Could not read file " << filename << "\n";
143:         std::throw_invalid_argument("File not found.");
144:     }
145:
146:     return data;
147: }
148:
149: void TerrainMapPublisher::loadMapFromCSV() {
150:     // Load in all terrain data
151:     std::string package_path = ros::package::getPath("quad_utils");
152:     std::vector<std::vector<double>> x_data =
153:         loadCSV(package_path + "/data/" + terrain_type_ + "/x_data.csv");
154:     std::vector<std::vector<double>> y_data =
```

```

155:     loadCSV(package_path + "/data/" + terrain_type_ + "/y_data.csv");
156:     std::vector<std::vector<double>> z_data =
157:     loadCSV(package_path + "/data/" + terrain_type_ + "/z_data.csv");
158:     std::vector<std::vector<double>> nx_data =
159:     loadCSV(package_path + "/data/" + terrain_type_ + "/nx_data.csv");
160:     std::vector<std::vector<double>> ny_data =
161:     loadCSV(package_path + "/data/" + terrain_type_ + "/ny_data.csv");
162:     std::vector<std::vector<double>> nz_data =
163:     loadCSV(package_path + "/data/" + terrain_type_ + "/nz_data.csv");
164:     std::vector<std::vector<double>> z_data_filt =
165:     loadCSV(package_path + "/data/" + terrain_type_ + "/z_data_filt.csv");
166:     std::vector<std::vector<double>> nx_data_filt =
167:     loadCSV(package_path + "/data/" + terrain_type_ + "/nx_data_filt.csv");
168:     std::vector<std::vector<double>> ny_data_filt =
169:     loadCSV(package_path + "/data/" + terrain_type_ + "/ny_data_filt.csv");
170:     std::vector<std::vector<double>> nz_data_filt =
171:     loadCSV(package_path + "/data/" + terrain_type_ + "/nz_data_filt.csv");
172:
173:     // Grab map length and resolution parameters, make sure resolution is square
174:     // (and align grid centers with data points)
175:     int x_size = z_data[0].size();
176:     int y_size = z_data.size();
177:     float x_res = x_data[0][1] - x_data[0][0];
178:     float y_res = y_data[1][0] - y_data[0][0];
179:     double x_length = x_data[0].back() - x_data[0].front() + x_res;
180:     double y_length = y_data.back()[0] - y_data.front()[0] + y_res;
181:     if (x_res != y_res) {
182:         throw std::runtime_error(
183:             "Map did not have square elements, make sure x and y resolution are "
184:             "equal.");
185:     }
186:
187:     // Initialize the map
188:     terrain_map_.setFrameId(map_frame_);
189:     terrain_map_.setGeometry(
190:         grid_map::Length(x_length, y_length), x_res,
191:         grid_map::Position(x_data[0].front() - 0.5 * x_res + 0.5 * x_length,
192:             y_data.front()[0] - 0.5 * y_res + 0.5 * y_length));
193:     ROS_INFO("Created map with size %f x %f m (%i x %i cells).",
194:         terrain_map_.getLength().x(), terrain_map_.getLength().y(),
195:         terrain_map_.getSize()(0), terrain_map_.getSize()(1));
196:
197:     // Load in the elevation and slope data
198:     for (grid_map::GridMapIterator iterator(terrain_map_); !iterator.isPastEnd();
199:         ++iterator) {
200:         const grid_map::Index index(*iterator);
201:         grid_map::Position position;
202:         terrain_map_.getPosition(*iterator, position);
203:         terrain_map_.at("z", *iterator) =
204:             z_data[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
205:         terrain_map_.at("nx", *iterator) =
206:             nx_data[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
207:         terrain_map_.at("ny", *iterator) =
208:             ny_data[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
209:         terrain_map_.at("nz", *iterator) =
210:             nz_data[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
211:
212:         terrain_map_.at("z_filt", *iterator) =
213:             z_data_filt[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
214:         terrain_map_.at("nx_filt", *iterator) =
215:             nx_data_filt[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
216:         terrain_map_.at("ny_filt", *iterator) =
217:             ny_data_filt[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
218:         terrain_map_.at("nz_filt", *iterator) =
219:             nz_data_filt[(y_size - 1) - index[1]][(x_size - 1) - index[0]];
220:     }
221: }
222:
223: void TerrainMapPublisher::loadMapFromImage(const sensor_msgs::Image &msg) {
224:     // Initialize the map from the image message if not already done so
225:     if (!map_initialized_) {
226:         grid_map::GridMapRosConverter::initializeFromImage(msg, resolution_,
227:             terrain_map_);
228:         ROS_INFO("Initialized map with size %f x %f m (%i x %i cells).",
229:             terrain_map_.getLength().x(), terrain_map_.getLength().y(),
230:             terrain_map_.getSize()(0), terrain_map_.getSize()(1));
231:         map_initialized_ = true;

```

```
232: }
233:
234: // Add the data layers
235: grid_map::GridMapRosConverter::addLayerFromImage(msg, "z", terrain_map_,
236:                                                  min_height_, max_height_);
237: grid_map::GridMapRosConverter::addColorLayerFromImage(msg, "color",
238:                                                         terrain_map_);
239:
240: // Add in slope information
241: for (grid_map::GridMapIterator it(terrain_map_); !it.isPastEnd(); ++it) {
242:     grid_map::Position position;
243:     terrain_map_.at("nx", *it) = 0.0;
244:     terrain_map_.at("ny", *it) = 0.0;
245:     terrain_map_.at("nz", *it) = 1.0;
246: }
247:
248: // Move the map to place starting location at (0,0)
249: grid_map::Position offset = {4.5, 0.0};
250: terrain_map_.setPosition(offset);
251: }
252:
253: void TerrainMapPublisher::publishMap() {
254:     // Set the time at which the map was published
255:     ros::Time time = ros::Time::now();
256:     terrain_map_.setTimestamp(time.toNSec());
257:
258:     // Generate grid_map message, convert, and publish
259:     grid_map_msgs::GridMap terrain_map_msg;
260:     grid_map::GridMapRosConverter::toMessage(terrain_map_, terrain_map_msg);
261:     terrain_map_pub_.publish(terrain_map_msg);
262: }
263:
264: void TerrainMapPublisher::spin() {
265:     ros::Rate r(update_rate_);
266:
267:     // Either wait for an image to show up on the topic or create a map from
268:     // scratch
269:     if (map_data_source_.compare("image") == 0) {
270:         // Spin until image message has been received and processed
271:         boost::shared_ptr<sensor_msgs::Image const> shared_image;
272:         while ((shared_image == nullptr) && ros::ok()) {
273:             shared_image = ros::topic::waitForMessage<sensor_msgs::Image>(
274:                 "/image_publisher/image", nh_);
275:             ros::spinOnce();
276:         }
277:     } else if (map_data_source_.compare("csv") == 0) {
278:         loadMapFromCSV();
279:     } else {
280:         createMap();
281:     }
282:
283:     // Continue publishing the map at the update rate
284:     while (ros::ok()) {
285:         updateParams();
286:
287:         if (map_data_source_.compare("internal") == 0) {
288:             updateMap();
289:         }
290:
291:         publishMap();
292:         ros::spinOnce();
293:         r.sleep();
294:     }
295: }
```

```
1: #include <quad_utils/ros_utils.h>
2:
3: namespace quad_utils {
4:
5: void updateStateHeaders(quad_msgs::RobotState &msg, ros::Time stamp,
6:                         std::string frame, int traj_index) {
7:     // Fill in the data across the messages
8:     msg.header.stamp = stamp;
9:     msg.header.frame_id = frame;
10:    msg.header.seq = traj_index;
11:    msg.body.header = msg.header;
12:    msg.feet.header = msg.header;
13:    for (int i = 0; i < msg.feet.feet.size(); i++) {
14:        msg.feet.feet[i].header = msg.header;
15:    }
16:    msg.joints.header = msg.header;
17:
18:    msg.traj_index = traj_index;
19:    msg.feet.traj_index = traj_index;
20:    for (int i = 0; i < msg.feet.feet.size(); i++) {
21:        msg.feet.feet[i].traj_index = traj_index;
22:    }
23: }
24:
25: void interpHeader(std_msgs::Header header_1, std_msgs::Header header_2,
26:                  double t_interp, std_msgs::Header &interp_header) {
27:     // Copy everything from the first header
28:     interp_header.frame_id = header_1.frame_id;
29:     interp_header.seq = header_1.seq;
30:
31:     // Compute the correct ros::Time corresponding to t_interp
32:     t_interp = std::max(std::min(t_interp, 1.0), 0.0);
33:     ros::Duration state_duration = header_2.stamp - header_1.stamp;
34:     ros::Duration interp_duration =
35:         ros::Duration(t_interp * state_duration.toSec());
36:     interp_header.stamp = header_1.stamp + ros::Duration(interp_duration);
37: }
38:
39: void interpOdometry(quad_msgs::BodyState state_1, quad_msgs::BodyState state_2,
40:                    double t_interp, quad_msgs::BodyState &interp_state) {
41:     interpHeader(state_1.header, state_2.header, t_interp, interp_state.header);
42:
43:     // Interp body position
44:     interp_state.pose.position.x = math_utils::lerp(
45:         state_1.pose.position.x, state_2.pose.position.x, t_interp);
46:     interp_state.pose.position.y = math_utils::lerp(
47:         state_1.pose.position.y, state_2.pose.position.y, t_interp);
48:     interp_state.pose.position.z = math_utils::lerp(
49:         state_1.pose.position.z, state_2.pose.position.z, t_interp);
50:
51:     // Interp body orientation with smath_utils::lerp
52:     tf2::Quaternion q_1, q_2, q_interp;
53:     tf2::convert(state_1.pose.orientation, q_1);
54:     tf2::convert(state_2.pose.orientation, q_2);
55:     q_interp = q_1.slerp(q_2, t_interp);
56:     interp_state.pose.orientation = tf2::toMsg(q_interp);
57:
58:     // Interp twist
59:     interp_state.twist.linear.x = math_utils::lerp(
60:         state_1.twist.linear.x, state_2.twist.linear.x, t_interp);
61:     interp_state.twist.linear.y = math_utils::lerp(
62:         state_1.twist.linear.y, state_2.twist.linear.y, t_interp);
63:     interp_state.twist.linear.z = math_utils::lerp(
64:         state_1.twist.linear.z, state_2.twist.linear.z, t_interp);
65:
66:     interp_state.twist.angular.x = math_utils::lerp(
67:         state_1.twist.angular.x, state_2.twist.angular.x, t_interp);
68:     interp_state.twist.angular.y = math_utils::lerp(
69:         state_1.twist.angular.y, state_2.twist.angular.y, t_interp);
70:     interp_state.twist.angular.z = math_utils::lerp(
71:         state_1.twist.angular.z, state_2.twist.angular.z, t_interp);
72: }
73:
74: void interpJointState(sensor_msgs::JointState state_1,
75:                      sensor_msgs::JointState state_2, double t_interp,
76:                      sensor_msgs::JointState &interp_state) {
77:     interpHeader(state_1.header, state_2.header, t_interp, interp_state.header);
```



```
78:
79:  // Interp joints
80:  interp_state.name.resize(state_1.position.size());
81:  interp_state.position.resize(state_1.position.size());
82:  interp_state.velocity.resize(state_1.position.size());
83:  interp_state.effort.resize(state_1.position.size());
84:  for (int i = 0; i < state_1.position.size(); i++) {
85:    interp_state.name[i] = state_1.name[i];
86:    interp_state.position[i] =
87:      math_utils::lerp(state_1.position[i], state_2.position[i], t_interp);
88:    interp_state.velocity[i] =
89:      math_utils::lerp(state_1.velocity[i], state_2.velocity[i], t_interp);
90:    interp_state.effort[i] =
91:      math_utils::lerp(state_1.effort[i], state_2.effort[i], t_interp);
92:  }
93: }
94:
95: void interpMultiFootState(quad_msgs::MultiFootState state_1,
96:                           quad_msgs::MultiFootState state_2, double t_interp,
97:                           quad_msgs::MultiFootState &interp_state) {
98:   interpHeader(state_1.header, state_2.header, t_interp, interp_state.header);
99:
100:  // Interp foot state
101:  interp_state.feet.resize(state_1.feet.size());
102:  for (int i = 0; i < interp_state.feet.size(); i++) {
103:    interp_state.feet[i].header = interp_state.header;
104:
105:    interp_state.feet[i].position.x = math_utils::lerp(
106:      state_1.feet[i].position.x, state_2.feet[i].position.x, t_interp);
107:    interp_state.feet[i].position.y = math_utils::lerp(
108:      state_1.feet[i].position.y, state_2.feet[i].position.y, t_interp);
109:    interp_state.feet[i].position.z = math_utils::lerp(
110:      state_1.feet[i].position.z, state_2.feet[i].position.z, t_interp);
111:
112:    // Interp foot velocity
113:    interp_state.feet[i].velocity.x = math_utils::lerp(
114:      state_1.feet[i].velocity.x, state_2.feet[i].velocity.x, t_interp);
115:    interp_state.feet[i].velocity.y = math_utils::lerp(
116:      state_1.feet[i].velocity.y, state_2.feet[i].velocity.y, t_interp);
117:    interp_state.feet[i].velocity.z = math_utils::lerp(
118:      state_1.feet[i].velocity.z, state_2.feet[i].velocity.z, t_interp);
119:
120:    // Interp foot acceleration
121:    interp_state.feet[i].acceleration.x =
122:      math_utils::lerp(state_1.feet[i].acceleration.x,
123:        state_2.feet[i].acceleration.x, t_interp);
124:    interp_state.feet[i].acceleration.y =
125:      math_utils::lerp(state_1.feet[i].acceleration.y,
126:        state_2.feet[i].acceleration.y, t_interp);
127:    interp_state.feet[i].acceleration.z =
128:      math_utils::lerp(state_1.feet[i].acceleration.z,
129:        state_2.feet[i].acceleration.z, t_interp);
130:
131:    // Set contact state to the first state
132:    interp_state.feet[i].contact = state_1.feet[i].contact;
133:  }
134: }
135:
136: void interpGRFArray(quad_msgs::GRFArray state_1, quad_msgs::GRFArray state_2,
137:                    double t_interp, quad_msgs::GRFArray &interp_state) {
138:   interpHeader(state_1.header, state_2.header, t_interp, interp_state.header);
139:
140:  // Interp grf state
141:  interp_state.vectors.resize(state_1.vectors.size());
142:  interp_state.points.resize(state_1.points.size());
143:  interp_state.contact_states.resize(state_1.contact_states.size());
144:  for (int i = 0; i < interp_state.vectors.size(); i++) {
145:    interp_state.vectors[i].x =
146:      math_utils::lerp(state_1.vectors[i].x, state_2.vectors[i].x, t_interp);
147:    interp_state.vectors[i].y =
148:      math_utils::lerp(state_1.vectors[i].y, state_2.vectors[i].y, t_interp);
149:    interp_state.vectors[i].z =
150:      math_utils::lerp(state_1.vectors[i].z, state_2.vectors[i].z, t_interp);
151:
152:    interp_state.points[i].x =
153:      math_utils::lerp(state_1.points[i].x, state_2.points[i].x, t_interp);
154:    interp_state.points[i].y =
```

```
155:     math_utils::lerp(state_1.points[i].y, state_2.points[i].y, t_interp);
156:     interp_state.points[i].z =
157:     math_utils::lerp(state_1.points[i].z, state_2.points[i].z, t_interp);
158:
159:     // Set contact state to the first state
160:     interp_state.contact_states[i] = state_1.contact_states[i];
161: }
162: }
163:
164: void interpRobotState(quad_msgs::RobotState state_1,
165:     quad_msgs::RobotState state_2, double t_interp,
166:     quad_msgs::RobotState &interp_state) {
167:     // Interp individual elements
168:     interpHeader(state_1.header, state_2.header, t_interp, interp_state.header);
169:     interpOdometry(state_1.body, state_2.body, t_interp, interp_state.body);
170:     interpJointState(state_1.joints, state_2.joints, t_interp,
171:         interp_state.joints);
172:     interpMultiFootState(state_1.feet, state_2.feet, t_interp, interp_state.feet);
173: }
174:
175: void interpRobotPlan(quad_msgs::RobotPlan msg, double t,
176:     quad_msgs::RobotState &interp_state,
177:     int &interp_primitive_id,
178:     quad_msgs::GRFArray &interp_grf) {
179:     // Define some useful timing parameters
180:     ros::Time t0_ros = msg.states.front().header.stamp;
181:     ros::Time t_ros = t0_ros + ros::Duration(t);
182:
183:     // Declare variables for interpolating between, both for input and output data
184:     quad_msgs::RobotState state_1, state_2;
185:     int primitive_id_1, primitive_id_2;
186:     quad_msgs::GRFArray grf_1, grf_2;
187:
188:     // Find the correct index for interp (return the first index if t < 0)
189:     int index = 0;
190:     if (t >= 0) {
191:         for (int i = 0; i < msg.states.size() - 1; i++) {
192:             index = i;
193:             if (msg.states[i].header.stamp <= t_ros &&
194:                 t_ros < msg.states[i + 1].header.stamp) {
195:                 break;
196:             }
197:         }
198:     }
199:
200:     // Extract correct states
201:     state_1 = msg.states[index];
202:     state_2 = msg.states[index + 1];
203:     primitive_id_1 = msg.primitive_ids[index];
204:     grf_1 = msg.grfs[index];
205:     grf_2 = msg.grfs[index + 1];
206:
207:     // Compute t_interp = [0,1]
208:     double t1, t2;
209:     ros::Duration t1_ros = state_1.header.stamp - t0_ros;
210:     t1 = t1_ros.toSec();
211:     ros::Duration t2_ros = state_2.header.stamp - t0_ros;
212:     t2 = t2_ros.toSec();
213:     double t_interp = (t - t1) / (t2 - t1);
214:
215:     // Compute interpolation
216:     interpRobotState(state_1, state_2, t_interp, interp_state);
217:     interp_primitive_id = primitive_id_1;
218:     interpGRFArray(grf_1, grf_2, t_interp, interp_grf);
219: }
220:
221: quad_msgs::MultiFootState interpMultiFootPlanContinuous(
222:     quad_msgs::MultiFootPlanContinuous msg, double t) {
223:     // Define some useful timing parameters
224:     ros::Time t0_ros = msg.states.front().header.stamp;
225:     ros::Time t_ros = t0_ros + ros::Duration(t);
226:
227:     // Declare variables for interpolating between, both for input and output data
228:     quad_msgs::MultiFootState state_1, state_2, interp_state;
229:
230:     // Find the correct index for interp (return the first index if t < 0)
231:     int index = 0;
```

```

232:     if (t >= 0) {
233:         for (int i = 0; i < msg.states.size() - 1; i++) {
234:             index = i;
235:             if (msg.states[i].header.stamp <= t_ros &&
236:                 t_ros < msg.states[i + 1].header.stamp) {
237:                 break;
238:             }
239:         }
240:     }
241:
242:     // Extract correct states
243:     state_1 = msg.states[index];
244:     state_2 = msg.states[index + 1];
245:
246:     // Compute t_interp = [0,1]
247:     double t1, t2;
248:     ros::Duration t1_ros = state_1.header.stamp - t0_ros;
249:     t1 = t1_ros.toSec();
250:     ros::Duration t2_ros = state_2.header.stamp - t0_ros;
251:     t2 = t2_ros.toSec();
252:     double t_interp = (t - t1) / (t2 - t1);
253:
254:     // Compute interpolation
255:     interpMultiFootState(state_1, state_2, t_interp, interp_state);
256:
257:     return interp_state;
258: }
259:
260: // quad_msgs::RobotState interpRobotStateTraj(quad_msgs::RobotStateTrajectory
261: // msg,
262: //                                     double t) {
263: //     // Define some useful timing parameters
264: //     ros::Time t0_ros = msg.states.front().header.stamp;
265: //     ros::Time tf_ros = msg.states.back().header.stamp;
266: //     ros::Duration traj_duration = tf_ros - t0_ros;
267: //
268: //     t = std::max(std::min(t, traj_duration.toSec()), 0.0);
269: //     ros::Time t_ros = t0_ros + ros::Duration(t);
270: //
271: //     // Declare variables for interpolating between, both for input and output
272: //     data quad_msgs::RobotState state_1, state_2, interp_state;
273: //
274: //     // Find the correct index for interp (return the first index if t < 0)
275: //     int index = 0;
276: //     if (t >= 0) {
277: //         for (int i = 0; i < msg.states.size() - 1; i++) {
278: //             index = i;
279: //             if (msg.states[i].header.stamp <= t_ros &&
280: //                 t_ros < msg.states[i + 1].header.stamp) {
281: //                 break;
282: //             }
283: //         }
284: //     }
285: //
286: //     // Extract correct states
287: //     state_1 = msg.states[index];
288: //     state_2 = msg.states[index + 1];
289: //
290: //     // Compute t_interp = [0,1]
291: //     double t1, t2;
292: //     ros::Duration t1_ros = state_1.header.stamp - t0_ros;
293: //     t1 = t1_ros.toSec();
294: //     ros::Duration t2_ros = state_2.header.stamp - t0_ros;
295: //     t2 = t2_ros.toSec();
296: //     double t_interp = (t - t1) / (t2 - t1);
297: //
298: //     // Compute interpolation
299: //     interpRobotState(state_1, state_2, t_interp, interp_state);
300: //
301: //     return interp_state;
302: // }
303:
304: void ikRobotState(const quad_utils::QuadKD &kinematics,
305:                  quad_msgs::BodyState body_state,
306:                  quad_msgs::MultiFootState multi_foot_state,
307:                  sensor_msgs::JointState &joint_state) {
308:     joint_state.header = multi_foot_state.header;

```

```
309: // If this message is empty set the joint names
310: if (joint_state.name.empty()) {
311:     joint_state.name = {"8", "0", "1", "9", "2", "3",
312:                         "10", "4", "5", "11", "6", "7"};
313: }
314: joint_state.position.clear();
315: joint_state.velocity.clear();
316: joint_state.effort.clear();
317:
318: for (int i = 0; i < multi_foot_state.feet.size(); i++) {
319:     // Get foot position data
320:     Eigen::Vector3d foot_pos;
321:     foot_pos[0] = multi_foot_state.feet[i].position.x;
322:     foot_pos[1] = multi_foot_state.feet[i].position.y;
323:     foot_pos[2] = multi_foot_state.feet[i].position.z;
324:
325:     // Get corresponding body plan data
326:     Eigen::Vector3d body_pos = {body_state.pose.position.x,
327:                                body_state.pose.position.y,
328:                                body_state.pose.position.z};
329:
330:     tf2::Quaternion q;
331:     tf2::convert(body_state.pose.orientation, q);
332:     tf2::Matrix3x3 m(q);
333:     double roll, pitch, yaw;
334:     m.getRPY(roll, pitch, yaw);
335:     Eigen::Vector3d body_rpy = {roll, pitch, yaw};
336:
337:     // Compute IK to get joint data
338:     Eigen::Vector3d leg_joint_state;
339:     kinematics.worldToFootIKWorldFrame(i, body_pos, body_rpy, foot_pos,
340:                                         leg_joint_state);
341:
342:     // Add to the joint state vector
343:     joint_state.position.push_back(leg_joint_state[0]);
344:     joint_state.position.push_back(leg_joint_state[1]);
345:     joint_state.position.push_back(leg_joint_state[2]);
346:
347:     joint_state.effort.push_back(0.0);
348:     joint_state.effort.push_back(0.0);
349:     joint_state.effort.push_back(0.0);
350: }
351:
352: // Declare state data as Eigen vectors
353: Eigen::VectorXd ref_body_state(12), ref_foot_positions(12),
354: ref_foot_velocities(12);
355:
356: // Load state data
357: ref_body_state = quad_utils::bodyStateMsgToEigen(body_state);
358: quad_utils::multiFootStateMsgToEigen(multi_foot_state, ref_foot_positions,
359:                                       ref_foot_velocities);
360:
361: // Define vectors for joint positions and velocities
362: Eigen::VectorXd joint_positions(12), joint_velocities(12);
363:
364: // Load joint positions
365: quad_utils::vectorToEigen(joint_state.position, joint_positions);
366:
367: // Define vectors for state positions
368: Eigen::VectorXd state_positions(18);
369:
370: // Load state positions
371: state_positions << joint_positions, ref_body_state.head(6);
372:
373: // Compute jacobian
374: Eigen::MatrixXd jacobian = Eigen::MatrixXd::Zero(12, 18);
375: kinematics.getJacobianBodyAngVel(state_positions, jacobian);
376:
377: // Compute joint velocities
378: joint_velocities =
379:     math_utils::sdlInv(jacobian.leftCols(12)) *
380:     (ref_foot_velocities - jacobian.rightCols(6) * ref_body_state.tail(6));
381:
382: // Populate joint velocities message
383: for (int i = 0; i < 12; ++i) {
384:     joint_state.velocity.push_back(joint_velocities(i));
385: }
```

```
386: }
387:
388: void ikRobotState(const quad_utils::QuadKD &kinematics,
389:                  quad_msgs::RobotState &state) {
390:     ikRobotState(kinematics, state.body, state.feet, state.joints);
391: }
392:
393: void fkRobotState(const quad_utils::QuadKD &kinematics,
394:                  quad_msgs::BodyState body_state,
395:                  sensor_msgs::JointState joint_state,
396:                  quad_msgs::MultiFootState &multi_foot_state) {
397:     multi_foot_state.header = joint_state.header;
398:     // If this message is empty set the joint names
399:
400:     int num_feet = 4;
401:     multi_foot_state.feet.resize(num_feet);
402:
403:     int joint_index = -1;
404:     for (int i = 0; i < multi_foot_state.feet.size(); i++) {
405:         // Get joint data for indexed leg leg
406:         Eigen::Vector3d leg_joint_state;
407:
408:         for (int j = 0; j < 3; j++) {
409:             joint_index++;
410:             leg_joint_state[j] = joint_state.position.at(joint_index);
411:         }
412:
413:         // Get corresponding body plan data
414:         Eigen::Vector3d body_pos = {body_state.pose.position.x,
415:                                     body_state.pose.position.y,
416:                                     body_state.pose.position.z};
417:
418:         tf2::Quaternion q;
419:         tf2::convert(body_state.pose.orientation, q);
420:         tf2::Matrix3x3 m(q);
421:         double roll, pitch, yaw;
422:         m.getRPY(roll, pitch, yaw);
423:         Eigen::Vector3d body_rpy = {roll, pitch, yaw};
424:
425:         // Compute IK to get joint data
426:         Eigen::Vector3d foot_pos;
427:         kinematics.worldToFootFKWorldFrame(i, body_pos, body_rpy, leg_joint_state,
428:                                             foot_pos);
429:
430:         // Add to the foot position vector
431:         multi_foot_state.feet[i].position.x = foot_pos[0];
432:         multi_foot_state.feet[i].position.y = foot_pos[1];
433:         multi_foot_state.feet[i].position.z = foot_pos[2];
434:
435:         multi_foot_state.feet[i].header = multi_foot_state.header;
436:     }
437:
438:     // Declare state data as Eigen vectors
439:     Eigen::VectorXd ref_body_state(12), foot_velocities(12);
440:
441:     // Load state data
442:     ref_body_state = quad_utils::bodyStateMsgToEigen(body_state);
443:
444:     // Define vectors for joint positions and velocities
445:     Eigen::VectorXd joint_positions(12), joint_velocities(12);
446:
447:     // Load joint positions
448:     quad_utils::vectorToEigen(joint_state.position, joint_positions);
449:
450:     // Load joint velocities
451:     quad_utils::vectorToEigen(joint_state.velocity, joint_velocities);
452:
453:     // Define vectors for state positions
454:     Eigen::VectorXd state_positions(18), state_velocities(18);
455:
456:     // Load state positions
457:     state_positions << joint_positions, ref_body_state.head(6);
458:
459:     // Load state velocities
460:     state_velocities << joint_velocities, ref_body_state.tail(6);
461:
462:     // Compute jacobian
```

```
463: Eigen::MatrixX<double> jacobian = Eigen::MatrixX<double>::Zero(12, 18);
464: kinematics.getJacobianBodyAngVel(state_positions, jacobian);
465:
466: // Compute foot velocities
467: foot_velocities = jacobian * state_velocities;
468:
469: // Populate foot velocities message
470: for (int i = 0; i < multi_foot_state.feet.size(); ++i) {
471:     multi_foot_state.feet[i].velocity.x = foot_velocities(i * 3 + 0);
472:     multi_foot_state.feet[i].velocity.y = foot_velocities(i * 3 + 1);
473:     multi_foot_state.feet[i].velocity.z = foot_velocities(i * 3 + 2);
474: }
475: }
476:
477: void fkRobotState(const quad_utils::QuadKD &kinematics,
478:                  quad_msgs::RobotState &state) {
479:     fkRobotState(kinematics, state.body, state.joints, state.feet);
480: }
481:
482: quad_msgs::BodyState eigenToBodyStateMsg(const Eigen::VectorX<double> &state) {
483:     quad_msgs::BodyState state_msg;
484:
485:     // Transform from RPY to quat msg
486:     tf2::Quaternion quat_tf;
487:     geometry_msgs::Quaternion quat_msg;
488:     quat_tf.setRPY(state[3], state[4], state[5]);
489:     quat_msg = tf2::toMsg(quat_tf);
490:
491:     // Load the data into the message
492:     state_msg.pose.position.x = state[0];
493:     state_msg.pose.position.y = state[1];
494:     state_msg.pose.position.z = state[2];
495:     state_msg.pose.orientation = quat_msg;
496:
497:     state_msg.twist.linear.x = state[6];
498:     state_msg.twist.linear.y = state[7];
499:     state_msg.twist.linear.z = state[8];
500:     state_msg.twist.angular.x = state[9];
501:     state_msg.twist.angular.y = state[10];
502:     state_msg.twist.angular.z = state[11];
503:
504:     return state_msg;
505: }
506:
507: Eigen::VectorX<double> bodyStateMsgToEigen(const quad_msgs::BodyState &body) {
508:     Eigen::VectorX<double> state = Eigen::VectorX<double>::Zero(12);
509:
510:     // Position
511:     state(0) = body.pose.position.x;
512:     state(1) = body.pose.position.y;
513:     state(2) = body.pose.position.z;
514:
515:     // Orientation
516:     tf2::Quaternion quat;
517:     tf2::convert(body.pose.orientation, quat);
518:     double r, p, y;
519:     tf2::Matrix3x3 m(quat);
520:     m.getRPY(r, p, y);
521:     state(3) = r;
522:     state(4) = p;
523:     state(5) = y;
524:
525:     // Linear Velocity
526:     state(6) = body.twist.linear.x;
527:     state(7) = body.twist.linear.y;
528:     state(8) = body.twist.linear.z;
529:
530:     // Angular Velocity
531:     state(9) = body.twist.angular.x;
532:     state(10) = body.twist.angular.y;
533:     state(11) = body.twist.angular.z;
534:
535:     return state;
536: }
537:
538: void eigenToGRFArrayMsg(Eigen::VectorX<double> grf_array,
539:                         quad_msgs::MultiFootState multi_foot_state_msg,
```

```
540:         quad_msgs::GRFArray &grf_msg) {
541:     grf_msg.vectors.clear();
542:     grf_msg.points.clear();
543:     grf_msg.contact_states.clear();
544:
545:     for (int i = 0; i < multi_foot_state_msg.feet.size(); i++) {
546:         Eigen::Vector3d grf = grf_array.segment<3>(3 * i);
547:
548:         geometry_msgs::Vector3 vector_msg;
549:         vector_msg.x = grf[0];
550:         vector_msg.y = grf[1];
551:         vector_msg.z = grf[2];
552:         geometry_msgs::Point point_msg;
553:         point_msg.x = multi_foot_state_msg.feet[i].position.x;
554:         point_msg.y = multi_foot_state_msg.feet[i].position.y;
555:         point_msg.z = multi_foot_state_msg.feet[i].position.z;
556:
557:         grf_msg.vectors.push_back(vector_msg);
558:         grf_msg.points.push_back(point_msg);
559:
560:         bool contact_state = (grf.norm() >= 1e-6);
561:         grf_msg.contact_states.push_back(contact_state);
562:     }
563: }
564:
565: Eigen::VectorXd grfArrayMsgToEigen(const quad_msgs::GRFArray &grf_array_msg_) {
566:     Eigen::VectorXd grf_array(3 * grf_array_msg_.vectors.size());
567:
568:     for (int i = 0; i < grf_array_msg_.vectors.size(); i++) {
569:         grf_array(3 * i) = grf_array_msg_.vectors[i].x;
570:         grf_array(3 * i + 1) = grf_array_msg_.vectors[i].y;
571:         grf_array(3 * i + 2) = grf_array_msg_.vectors[i].z;
572:     }
573:
574:     return grf_array;
575: }
576:
577: void footStateMsgToEigen(const quad_msgs::FootState &foot_state_msg,
578:     Eigen::Vector3d &foot_position) {
579:     foot_position[0] = foot_state_msg.position.x;
580:     foot_position[1] = foot_state_msg.position.y;
581:     foot_position[2] = foot_state_msg.position.z;
582: }
583:
584: void multiFootStateMsgToEigen(
585:     const quad_msgs::MultiFootState &multi_foot_state_msg,
586:     Eigen::VectorXd &foot_positions) {
587:     for (int i = 0; i < multi_foot_state_msg.feet.size(); i++) {
588:         foot_positions[3 * i] = multi_foot_state_msg.feet[i].position.x;
589:         foot_positions[3 * i + 1] = multi_foot_state_msg.feet[i].position.y;
590:         foot_positions[3 * i + 2] = multi_foot_state_msg.feet[i].position.z;
591:     }
592: }
593:
594: void multiFootStateMsgToEigen(
595:     const quad_msgs::MultiFootState &multi_foot_state_msg,
596:     Eigen::VectorXd &foot_positions, Eigen::VectorXd &foot_velocities,
597:     Eigen::VectorXd &foot_acceleration) {
598:     multiFootStateMsgToEigen(multi_foot_state_msg, foot_positions,
599:         foot_velocities);
600:
601:     for (int i = 0; i < multi_foot_state_msg.feet.size(); i++) {
602:         foot_acceleration[3 * i] = multi_foot_state_msg.feet[i].acceleration.x;
603:         foot_acceleration[3 * i + 1] = multi_foot_state_msg.feet[i].acceleration.y;
604:         foot_acceleration[3 * i + 2] = multi_foot_state_msg.feet[i].acceleration.z;
605:     }
606: }
607:
608: void multiFootStateMsgToEigen(
609:     const quad_msgs::MultiFootState &multi_foot_state_msg,
610:     Eigen::VectorXd &foot_positions, Eigen::VectorXd &foot_velocities) {
611:     for (int i = 0; i < multi_foot_state_msg.feet.size(); i++) {
612:         foot_positions[3 * i] = multi_foot_state_msg.feet[i].position.x;
613:         foot_positions[3 * i + 1] = multi_foot_state_msg.feet[i].position.y;
614:         foot_positions[3 * i + 2] = multi_foot_state_msg.feet[i].position.z;
615:
616:         foot_velocities[3 * i] = multi_foot_state_msg.feet[i].velocity.x;
```

```
617:     foot_velocities[3 * i + 1] = multi_foot_state_msg.feet[i].velocity.y;
618:     foot_velocities[3 * i + 2] = multi_foot_state_msg.feet[i].velocity.z;
619: }
620: }
621:
622: void eigenToFootStateMsg(Eigen::VectorXd foot_positions,
623:     Eigen::VectorXd foot_velocities,
624:     Eigen::VectorXd foot_acceleration,
625:     quad_msgs::FootState &foot_state_msg) {
626:     eigenToFootStateMsg(foot_positions, foot_velocities, foot_state_msg);
627:
628:     foot_state_msg.acceleration.x = foot_acceleration[0];
629:     foot_state_msg.acceleration.y = foot_acceleration[1];
630:     foot_state_msg.acceleration.z = foot_acceleration[2];
631: }
632:
633: void eigenToFootStateMsg(Eigen::VectorXd foot_positions,
634:     Eigen::VectorXd foot_velocities,
635:     quad_msgs::FootState &foot_state_msg) {
636:     foot_state_msg.position.x = foot_positions[0];
637:     foot_state_msg.position.y = foot_positions[1];
638:     foot_state_msg.position.z = foot_positions[2];
639:
640:     foot_state_msg.velocity.x = foot_velocities[0];
641:     foot_state_msg.velocity.y = foot_velocities[1];
642:     foot_state_msg.velocity.z = foot_velocities[2];
643: }
644:
645: void eigenToVector(const Eigen::VectorXd &eigen_vec, std::vector<double> &vec) {
646:     vec.resize(eigen_vec.size());
647:     for (int i = 0; i < eigen_vec.size(); i++) {
648:         vec[i] = eigen_vec(i);
649:     }
650: }
651:
652: void vectorToEigen(const std::vector<double> &vec, Eigen::VectorXd &eigen_vec) {
653:     eigen_vec.resize(vec.size());
654:     for (int i = 0; i < vec.size(); i++) {
655:         eigen_vec(i) = vec[i];
656:         // std::cout << vec[i] << std::endl;
657:     }
658: }
659:
660: void vector3MsgToEigen(const geometry_msgs::Vector3 &vec,
661:     Eigen::Vector3d &eigen_vec) {
662:     eigen_vec.x() = vec.x;
663:     eigen_vec.y() = vec.y;
664:     eigen_vec.z() = vec.z;
665: }
666:
667: void Eigen3ToVector3Msg(const Eigen::Vector3d &eigen_vec,
668:     geometry_msgs::Vector3 &vec) {
669:     vec.x = eigen_vec.x();
670:     vec.y = eigen_vec.y();
671:     vec.z = eigen_vec.z();
672: }
673:
674: void pointMsgToEigen(const geometry_msgs::Point &vec,
675:     Eigen::Vector3d &eigen_vec) {
676:     eigen_vec.x() = vec.x;
677:     eigen_vec.y() = vec.y;
678:     eigen_vec.z() = vec.z;
679: }
680:
681: void Eigen3ToPointMsg(const Eigen::Vector3d &eigen_vec,
682:     geometry_msgs::Point &vec) {
683:     vec.x = eigen_vec.x();
684:     vec.y = eigen_vec.y();
685:     vec.z = eigen_vec.z();
686: }
687: } // namespace quad_utils
688:
```



```
1: #include "quad_utils/quad_kd.h"
2:
3: using namespace quad_utils;
4:
5: Eigen::IOFormat CleanFmt(4, 0, " ", " ", "\n", "[", "]");
6:
7: QuadKD::QuadKD() { initModel(""); }
8:
9: QuadKD::QuadKD(std::string ns) { initModel("/") + ns + "/"; }
10:
11: void QuadKD::initModel(std::string ns) {
12:     std::string robot_description_string;
13:
14:     if (!ros::param::get("robot_description", robot_description_string)) {
15:         std::cerr << "Error loading robot_description " << std::endl;
16:         abort();
17:     }
18:
19:     model_ = new RigidBodyDynamics::Model();
20:     if (!RigidBodyDynamics::Addons::URDFReadFromString(
21:         robot_description_string.c_str(), model_, true)) {
22:         std::cerr << "Error loading model " << std::endl;
23:         abort();
24:     }
25:
26:     body_name_list_ = {"toe0", "toe1", "toe2", "toe3"};
27:
28:     body_id_list_.resize(4);
29:     for (size_t i = 0; i < body_name_list_.size(); i++) {
30:         body_id_list_.at(i) = model_>GetBodyId(body_name_list_.at(i).c_str());
31:     }
32:
33:     leg_idx_list_.resize(4);
34:     std::iota(leg_idx_list_.begin(), leg_idx_list_.end(),
35:         0); // Just initialize values for leg index.. not sure why
36:     // std::cout << "leg index initial val: " << leg_idx_list_.at(2) <<
37:     // std::endl;
38:
39:     // defining a function here.... to check it is in ascending order??
40:     // interesting
41:     std::sort(leg_idx_list_.begin(), leg_idx_list_.end(), [&](int i, int j) {
42:         return body_id_list_.at(i) < body_id_list_.at(j);
43:     });
44:
45:     // Read leg geometry from URDF
46:     legbase_offsets_.resize(4);
47:     l0_vec_.resize(4);
48:     std::vector<std::string> hip_name_list = {"hip0", "hip1", "hip2", "hip3"};
49:     std::vector<std::string> upper_name_list = {"upper0", "upper1", "upper2",
50:         "upper3"};
51:     std::vector<std::string> lower_name_list = {"lower0", "lower1", "lower2",
52:         "lower3"};
53:     std::vector<std::string> toe_name_list = {"toe0", "toe1", "toe2", "toe3"};
54:     RigidBodyDynamics::Math::SpatialTransform tform;
55:     for (size_t i = 0; i < 4; i++) {
56:         // From body COM to abad
57:         tform =
58:             model_>GetJointFrame(model_>GetBodyId(hip_name_list.at(i).c_str()));
59:         legbase_offsets_[i] = tform.r;
60:
61:         // From abad to hip
62:         tform =
63:             model_>GetJointFrame(model_>GetBodyId(upper_name_list.at(i).c_str()));
64:         l0_vec_[i] = tform.r(1);
65:
66:         // From hip to knee (we know they should be the same and the equation in IK
67:         // uses the magnitude of it)
68:         tform =
69:             model_>GetJointFrame(model_>GetBodyId(lower_name_list.at(i).c_str()));
70:         l1_ = tform.r.cwiseAbs().maxCoeff();
71:         knee_offset_ = tform.r;
72:
73:         // From knee to toe (we know they should be the same and the equation in IK
74:         // uses the magnitude of it)
75:         tform =
76:             model_>GetJointFrame(model_>GetBodyId(toe_name_list.at(i).c_str()));
77:         l2_ = tform.r.cwiseAbs().maxCoeff();
```

```

78:     foot_offset_ = tform.r;
79: }
80:
81: // Abad offset from legbase
82: abad_offset_ = {0, 0, 0};
83:
84: g_body_legbases_.resize(4);
85: for (int leg_index = 0; leg_index < 4; leg_index++) {
86:     // Compute transforms
87:     g_body_legbases_[leg_index] =
88:         createAffineMatrix(legbase_offsets_[leg_index],
89:                             Eigen::AngleAxisd(0, Eigen::Vector3d::UnitZ()));
90: }
91:
92: joint_min_.resize(num_feet_);
93: joint_max_.resize(num_feet_);
94:
95: std::vector<double> joint_min_front = {-0.707, -M_PI * 0.5, 0};
96: std::vector<double> joint_min_back = {-0.707, -M_PI, 0};
97: std::vector<double> joint_max_front = {0.707, M_PI, M_PI};
98: std::vector<double> joint_max_back = {0.707, M_PI * 0.5, M_PI};
99:
100: joint_min_ = {joint_min_front, joint_min_back, joint_min_front,
101:               joint_min_back};
102: joint_max_ = {joint_max_front, joint_max_back, joint_max_front,
103:               joint_max_back};
104: }
105:
106: Eigen::Matrix4d QuadKD::createAffineMatrix(Eigen::Vector3d trans,
107:                                             Eigen::Vector3d rpy) const {
108:     Eigen::Transform<double, 3, Eigen::Affine> t;
109:     t = Eigen::Translation<double, 3>(trans);
110:     t.rotate(Eigen::AngleAxisd(rpy[2], Eigen::Vector3d::UnitZ()));
111:     t.rotate(Eigen::AngleAxisd(rpy[1], Eigen::Vector3d::UnitY()));
112:     t.rotate(Eigen::AngleAxisd(rpy[0], Eigen::Vector3d::UnitX()));
113:
114:     return t.matrix();
115: }
116:
117: Eigen::Matrix4d QuadKD::createAffineMatrix(Eigen::Vector3d trans,
118:                                             Eigen::AngleAxisd rot) const {
119:     Eigen::Transform<double, 3, Eigen::Affine> t;
120:     t = Eigen::Translation<double, 3>(trans);
121:     t.rotate(rot);
122:
123:     return t.matrix();
124: }
125:
126: double QuadKD::getJointLowerLimit(int leg_index, int joint_index) const {
127:     return joint_min_[leg_index][joint_index];
128: }
129:
130: double QuadKD::getJointUpperLimit(int leg_index, int joint_index) const {
131:     return joint_max_[leg_index][joint_index];
132: }
133:
134: double QuadKD::getLinkLength(int leg_index, int link_index) const {
135:     switch (link_index) {
136:         case 0:
137:             return l0_vec_[leg_index];
138:         case 1:
139:             return l1_;
140:         case 2:
141:             return l2_;
142:         default:
143:             throw std::runtime_error("Invalid link index");
144:     }
145: }
146:
147: void QuadKD::transformBodyToWorld(Eigen::Vector3d body_pos,
148:                                    Eigen::Vector3d body_rpy,
149:                                    Eigen::Matrix4d transform_body,
150:                                    Eigen::Matrix4d &transform_world) const {
151:     // Compute transform from world to body frame
152:     Eigen::Matrix4d g_world_body = createAffineMatrix(body_pos, body_rpy);
153:
154:     // Get the desired transform in the world frame

```

```

155:   transform_world = g_world_body * transform_body;
156: }
157:
158: void QuadKD::transformWorldToBody(Eigen::Vector3d body_pos,
159:                                   Eigen::Vector3d body_rpy,
160:                                   Eigen::Matrix4d transform_world,
161:                                   Eigen::Matrix4d &transform_body) const {
162:   // Compute transform from world to body frame
163:   Eigen::Matrix4d g_world_body = createAffineMatrix(body_pos, body_rpy);
164:
165:   // Compute the desired transform in the body frame
166:   transform_body = g_world_body.inverse() * transform_world;
167: }
168:
169: void QuadKD::worldToLegbaseFKWorldFrame(
170:   int leg_index, Eigen::Vector3d body_pos, Eigen::Vector3d body_rpy,
171:   Eigen::Matrix4d &g_world_legbase) const {
172:   // Compute transforms
173:   Eigen::Matrix4d g_world_body = createAffineMatrix(body_pos, body_rpy);
174:
175:   // Compute transform for leg base relative to the world frame
176:   g_world_legbase = g_world_body * g_body_legbases_[leg_index];
177: }
178:
179: void QuadKD::worldToLegbaseFKWorldFrame(
180:   int leg_index, Eigen::Vector3d body_pos, Eigen::Vector3d body_rpy,
181:   Eigen::Vector3d &leg_base_pos_world) const {
182:   Eigen::Matrix4d g_world_legbase;
183:   worldToLegbaseFKWorldFrame(leg_index, body_pos, body_rpy, g_world_legbase);
184:
185:   leg_base_pos_world = g_world_legbase.block<3, 1>(0, 3);
186: }
187:
188: void QuadKD::worldToNominalHipFKWorldFrame(
189:   int leg_index, Eigen::Vector3d body_pos, Eigen::Vector3d body_rpy,
190:   Eigen::Vector3d &nominal_hip_pos_world) const {
191:   // Compute transforms
192:   Eigen::Matrix4d g_world_body = createAffineMatrix(body_pos, body_rpy);
193:   // Compute transform from body to legbase but offset by l0
194:   Eigen::Matrix4d g_body_nominal_hip = g_body_legbases_[leg_index];
195:   g_body_nominal_hip(1, 3) += 1.0 * l0_vec_[leg_index];
196:
197:   // Compute transform for offset leg base relative to the world frame
198:   Eigen::Matrix4d g_world_nominal_hip = g_world_body * g_body_nominal_hip;
199:
200:   nominal_hip_pos_world = g_world_nominal_hip.block<3, 1>(0, 3);
201: }
202:
203: void QuadKD::bodyToFootFKBodyFrame(int leg_index, Eigen::Vector3d joint_state,
204:                                     Eigen::Matrix4d &g_body_foot) const {
205:   if (leg_index > (legbase_offsets_.size() - 1) || leg_index < 0) {
206:     throw std::runtime_error("Leg index is outside valid range");
207:   }
208:
209:   // Define hip offset
210:   Eigen::Vector3d hip_offset = {0, l0_vec_[leg_index], 0};
211:
212:   // Initialize transforms
213:   Eigen::Matrix4d g_legbase_abad;
214:   Eigen::Matrix4d g_abad_hip;
215:   Eigen::Matrix4d g_hip_knee;
216:   Eigen::Matrix4d g_knee_foot;
217:
218:   g_legbase_abad = createAffineMatrix(
219:     abad_offset_,
220:     Eigen::AngleAxisd(joint_state[0], Eigen::Vector3d::UnitX()));
221:
222:   g_abad_hip = createAffineMatrix(
223:     hip_offset, Eigen::AngleAxisd(joint_state[1], -Eigen::Vector3d::UnitY()));
224:
225:   g_hip_knee = createAffineMatrix(
226:     knee_offset_,
227:     Eigen::AngleAxisd(joint_state[2], Eigen::Vector3d::UnitY()));
228:
229:   g_knee_foot = createAffineMatrix(
230:     foot_offset_, Eigen::AngleAxisd(0, Eigen::Vector3d::UnitY()));
231:

```

```
232: // Get foot transform in world frame
233: g_body_foot = g_body_legbases_[leg_index] * g_legbase_abad * g_abad_hip *
234:             g_hip_knee * g_knee_foot;
235: }
236:
237: void QuadKD::bodyToFootFKBodyFrame(int leg_index, Eigen::Vector3d joint_state,
238:                                     Eigen::Vector3d &foot_pos_body) const {
239:     Eigen::Matrix4d g_body_foot;
240:     QuadKD::bodyToFootFKBodyFrame(leg_index, joint_state, g_body_foot);
241:
242:     // Extract cartesian position of foot
243:     foot_pos_body = g_body_foot.block<3, 1>(0, 3);
244: }
245:
246: void QuadKD::worldToFootFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
247:                                       Eigen::Vector3d body_rpy,
248:                                       Eigen::Vector3d joint_state,
249:                                       Eigen::Matrix4d &g_world_foot) const {
250:     if (leg_index > (legbase_offsets_.size() - 1) || leg_index < 0) {
251:         throw std::runtime_error("Leg index is outside valid range");
252:     }
253:
254:     // Define hip offset
255:     Eigen::Vector3d hip_offset = {0, 10_vec_[leg_index], 0};
256:
257:     // Initialize transforms
258:     Eigen::Matrix4d g_body_legbase;
259:     Eigen::Matrix4d g_legbase_abad;
260:     Eigen::Matrix4d g_abad_hip;
261:     Eigen::Matrix4d g_hip_knee;
262:     Eigen::Matrix4d g_knee_foot;
263:
264:     // Compute transforms
265:     Eigen::Matrix4d g_world_legbase;
266:     worldToLegbaseFKWorldFrame(leg_index, body_pos, body_rpy, g_world_legbase);
267:
268:     g_legbase_abad = createAffineMatrix(
269:         abad_offset_,
270:         Eigen::AngleAxisd(joint_state[0], Eigen::Vector3d::UnitX()));
271:
272:     g_abad_hip = createAffineMatrix(
273:         hip_offset, Eigen::AngleAxisd(joint_state[1], -Eigen::Vector3d::UnitY()));
274:
275:     g_hip_knee = createAffineMatrix(
276:         knee_offset_,
277:         Eigen::AngleAxisd(joint_state[2], Eigen::Vector3d::UnitY()));
278:
279:     g_knee_foot = createAffineMatrix(
280:         foot_offset_, Eigen::AngleAxisd(0, Eigen::Vector3d::UnitY()));
281:
282:     // Get foot transform in world frame
283:     g_world_foot =
284:         g_world_legbase * g_legbase_abad * g_abad_hip * g_hip_knee * g_knee_foot;
285: }
286:
287: void QuadKD::worldToFootFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
288:                                       Eigen::Vector3d body_rpy,
289:                                       Eigen::Vector3d joint_state,
290:                                       Eigen::Vector3d &foot_pos_world) const {
291:     Eigen::Matrix4d g_world_foot;
292:     worldToFootFKWorldFrame(leg_index, body_pos, body_rpy, joint_state,
293:                             g_world_foot);
294:
295:     // Extract cartesian position of foot
296:     foot_pos_world = g_world_foot.block<3, 1>(0, 3);
297: }
298:
299: void QuadKD::worldToKneeFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
300:                                       Eigen::Vector3d body_rpy,
301:                                       Eigen::Vector3d joint_state,
302:                                       Eigen::Matrix4d &g_world_knee) const {
303:     if (leg_index > (legbase_offsets_.size() - 1) || leg_index < 0) {
304:         throw std::runtime_error("Leg index is outside valid range");
305:     }
306:
307:     // Define hip offset
308:     Eigen::Vector3d hip_offset = {0, 10_vec_[leg_index], 0};
```

```

309:
310:     // Initialize transforms
311:     Eigen::Matrix4d g_body_legbase;
312:     Eigen::Matrix4d g_legbase_abad;
313:     Eigen::Matrix4d g_abad_hip;
314:     Eigen::Matrix4d g_hip_knee;
315:
316:     // Compute transforms
317:     Eigen::Matrix4d g_world_legbase;
318:     worldToLegbaseFKWorldFrame(leg_index, body_pos, body_rpy, g_world_legbase);
319:
320:     g_legbase_abad = createAffineMatrix(
321:         abad_offset_,
322:         Eigen::AngleAxisd(joint_state[0], Eigen::Vector3d::UnitX()));
323:
324:     g_abad_hip = createAffineMatrix(
325:         hip_offset, Eigen::AngleAxisd(joint_state[1], -Eigen::Vector3d::UnitY()));
326:
327:     g_hip_knee = createAffineMatrix(
328:         knee_offset_,
329:         Eigen::AngleAxisd(joint_state[2], Eigen::Vector3d::UnitY()));
330:
331:     // Get foot transform in world frame
332:     g_world_knee = g_world_legbase * g_legbase_abad * g_abad_hip * g_hip_knee;
333: }
334:
335: void QuadKD::worldToKneeFKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
336:     Eigen::Vector3d body_rpy,
337:     Eigen::Vector3d joint_state,
338:     Eigen::Vector3d &knee_pos_world) const {
339:     Eigen::Matrix4d g_world_knee;
340:     worldToKneeFKWorldFrame(leg_index, body_pos, body_rpy, joint_state,
341:         g_world_knee);
342:
343:     // Extract cartesian position of foot
344:     knee_pos_world = g_world_knee.block<3, 1>(0, 3);
345: }
346:
347: bool QuadKD::worldToFootIKWorldFrame(int leg_index, Eigen::Vector3d body_pos,
348:     Eigen::Vector3d body_rpy,
349:     Eigen::Vector3d foot_pos_world,
350:     Eigen::Vector3d &joint_state) const {
351:     if (leg_index > (legbase_offsets_.size() - 1) || leg_index < 0) {
352:         throw std::runtime_error("Leg index is outside valid range");
353:     }
354:
355:     // Calculate offsets
356:     Eigen::Vector3d legbase_offset = legbase_offsets_[leg_index];
357:     double l0 = l0_vec[leg_index];
358:
359:     // Initialize transforms
360:     Eigen::Matrix4d g_world_legbase;
361:     Eigen::Matrix4d g_world_foot;
362:     Eigen::Matrix4d g_legbase_foot;
363:     Eigen::Vector3d foot_pos_legbase;
364:
365:     // Compute transforms
366:     worldToLegbaseFKWorldFrame(leg_index, body_pos, body_rpy, g_world_legbase);
367:
368:     g_world_foot = createAffineMatrix(
369:         foot_pos_world, Eigen::AngleAxisd(0, Eigen::Vector3d::UnitY()));
370:
371:     // Compute foot position relative to the leg base in cartesian coordinates
372:     g_legbase_foot = g_world_legbase.inverse() * g_world_foot;
373:     foot_pos_legbase = g_legbase_foot.block<3, 1>(0, 3);
374:
375:     return legbaseToFootIKLegbaseFrame(leg_index, foot_pos_legbase, joint_state);
376: }
377:
378: bool QuadKD::legbaseToFootIKLegbaseFrame(int leg_index,
379:     Eigen::Vector3d foot_pos_legbase,
380:     Eigen::Vector3d &joint_state) const {
381:     // Initialize exact bool
382:     bool is_exact = true;
383:
384:     // Calculate offsets
385:     Eigen::Vector3d legbase_offset = legbase_offsets_[leg_index];

```

```
386: double l0 = l0_vec_[leg_index];
387:
388: // Extract coordinates and declare joint variables
389: double x = foot_pos_legbase[0];
390: double y = foot_pos_legbase[1];
391: double z = foot_pos_legbase[2];
392: double q0;
393: double q1;
394: double q2;
395:
396: // Start IK, check foot pos is at least l0 away from leg base, clamp otherwise
397: double temp = l0 / sqrt(z * z + y * y);
398: if (abs(temp) > 1) {
399:     ROS_DEBUG_THROTTLE(0.5, "Foot too close, choosing closest alternative\n");
400:     is_exact = false;
401:     temp = std::max(std::min(temp, 1.0), -1.0);
402: }
403:
404: // Compute both solutions of q0, use hip-above-knee if z<0 (preferred)
405: // Store the inverted solution in case hip limits are exceeded
406: if (z > 0) {
407:     q0 = -acos(temp) + atan2(z, y);
408: } else {
409:     q0 = acos(temp) + atan2(z, y);
410: }
411:
412: // Make sure abad is within joint limits, clamp otherwise
413: if (q0 > joint_max_[leg_index][0] || q0 < joint_min_[leg_index][0]) {
414:     q0 = std::max(std::min(q0, joint_max_[leg_index][0]),
415:                  joint_min_[leg_index][0]);
416:     is_exact = false;
417:     ROS_DEBUG_THROTTLE(0.5, "Abad limits exceeded, clamping to %5.3f \n", q0);
418: }
419:
420: // Rotate to ab-ad fixed frame
421: double z_body_frame = z;
422: z = -sin(q0) * y + cos(q0) * z_body_frame;
423:
424: // Check reachability for hip
425: double acos_eps = 1.0;
426: double temp2 =
427:     (l1_ * l1_ + x * x + z * z - l2_ * l2_) / (2 * l1_ * sqrt(x * x + z * z));
428: if (abs(temp2) > acos_eps) {
429:     ROS_DEBUG_THROTTLE(0.5,
430:         "Foot location too far for hip, choosing closest"
431:         " alternative \n");
432:     is_exact = false;
433:     temp2 = std::max(std::min(temp2, acos_eps), -acos_eps);
434: }
435:
436: // Check reachability for knee
437: double temp3 = (l1_ * l1_ + l2_ * l2_ - x * x - z * z) / (2 * l1_ * l2_);
438:
439: if (temp3 > acos_eps || temp3 < -acos_eps) {
440:     ROS_DEBUG_THROTTLE(0.5,
441:         "Foot location too far for knee, choosing closest"
442:         " alternative \n");
443:     is_exact = false;
444:
445:     temp3 = std::max(std::min(temp3, acos_eps), -acos_eps);
446: }
447:
448: // Compute joint angles
449: q1 = 0.5 * M_PI + atan2(x, -z) - acos(temp2);
450:
451: // Make sure hip is within joint limits
452: if (q1 > joint_max_[leg_index][1] || q1 < joint_min_[leg_index][1]) {
453:     q1 = std::max(std::min(q1, joint_max_[leg_index][1]),
454:                  joint_min_[leg_index][1]);
455:     is_exact = false;
456:     ROS_DEBUG_THROTTLE(0.5, "Hip limits exceeded, clamping to %5.3f \n", q1);
457: }
458:
459: // Compute knee val to get closest toe position in the plane
460: Eigen::Vector2d knee_pos, toe_pos, toe_offset;
461: knee_pos << -l1_ * cos(q1), -l1_ * sin(q1);
462: toe_pos << x, z;
```

```
463: toe_offset = toe_pos - knee_pos;
464: q2 = atan2(-toe_offset(1), toe_offset(0)) + q1;
465:
466: // Make sure knee is within joint limits
467: if (q2 > joint_max_[leg_index][2] || q2 < joint_min_[leg_index][2]) {
468:     q2 = std::max(std::min(q2, joint_max_[leg_index][2]),
469:         joint_min_[leg_index][2]);
470:     is_exact = false;
471:     ROS_DEBUG_THROTTLE(0.5, "Knee limit exceeded, clamping to %5.3f \n", q2);
472: }
473:
474: // q1 is undefined if q2=0, resolve this
475: if (q2 == 0) {
476:     q1 = 0;
477:     ROS_DEBUG_THROTTLE(0.5,
478:         "Hip value undefined (in singularity), setting to"
479:         " %5.3f \n",
480:         q1);
481:     is_exact = false;
482: }
483:
484: if (z_body_frame - 10 * sin(q0) > 0) {
485:     ROS_DEBUG_THROTTLE(0.5, "IK solution is in hip-inverted region! Beware!\n");
486:     is_exact = false;
487: }
488:
489: joint_state = {q0, q1, q2};
490: return is_exact;
491: }
492:
493: void QuadKD::getJacobianGenCoord(const Eigen::VectorXd &state,
494:     Eigen::MatrixXd &jacobian) const {
495:     this->getJacobianBodyAngVel(state, jacobian);
496:
497:     // RBDL uses Jacobian w.r.t. floating base angular velocity in body frame,
498:     // which is multiplied by Jacobian to map it to Euler angle change rate here
499:     for (size_t i = 0; i < 4; i++) {
500:         Eigen::MatrixXd transform_jac(3, 3);
501:         transform_jac << 1, 0, -sin(state(16)), 0, cos(state(15)),
502:             cos(state(16)) * sin(state(15)), 0, -sin(state(15)),
503:             cos(state(15)) * cos(state(16));
504:         jacobian.block(3 * i, 15, 3, 3) =
505:             jacobian.block(3 * i, 15, 3, 3) * transform_jac;
506:     }
507: }
508:
509: void QuadKD::getJacobianBodyAngVel(const Eigen::VectorXd &state,
510:     Eigen::MatrixXd &jacobian) const {
511:     assert(state.size() == 18);
512:
513:     // RBDL state vector has the floating base state in the front and the joint
514:     // state in the back When reading from URDF, the order of the legs is 2301,
515:     // which should be corrected by sorting the bodyID
516:     Eigen::VectorXd q(19);
517:     q.setZero();
518:
519:     q.head(3) = state.segment(12, 3);
520:
521:     tf2::Quaternion quat_tf;
522:     quat_tf.setRPY(state(15), state(16), state(17));
523:     q(3) = quat_tf.getX();
524:     q(4) = quat_tf.getY();
525:     q(5) = quat_tf.getZ();
526:
527:     // RBDL uses quaternion for floating base direction, but w is placed at the
528:     // end of the state vector
529:     q(18) = quat_tf.getW();
530:
531:     for (size_t i = 0; i < leg_idx_list_.size(); i++) {
532:         q.segment(6 + 3 * i, 3) = state.segment(3 * leg_idx_list_.at(i), 3);
533:     }
534:
535:     jacobian.setZero();
536:
537:     for (size_t i = 0; i < body_id_list_.size(); i++) {
538:         Eigen::MatrixXd jac_block(3, 18);
539:         jac_block.setZero();
```

```

540:     RigidBodyDynamics::CalcPointJacobian(*model_, q, body_id_list_.at(i),
541:                                           Eigen::Vector3d::Zero(), jac_block);
542:
543:     for (size_t j = 0; j < 4; j++) {
544:         jacobian.block(3 * i, 3 * leg_idx_list_.at(j), 3, 3) =
545:             jac_block.block(0, 6 + 3 * j, 3, 3);
546:     }
547:     jacobian.block(3 * i, 12, 3, 6) = jac_block.block(0, 0, 3, 6);
548: }
549: }
550:
551: void QuadKD::getJacobianWorldAngVel(const Eigen::VectorXd &state,
552:                                     Eigen::MatrixXd &jacobian) const {
553:     this->getJacobianBodyAngVel(state, jacobian);
554:
555:     // RBDL uses Jacobian w.r.t. floating base angular velocity in body frame,
556:     // which is multiplied by rotation matrix to map it to angular velocity in
557:     // world frame here
558:     for (size_t i = 0; i < 4; i++) {
559:         Eigen::Matrix3d rot;
560:         this->getRotationMatrix(state.segment(15, 3), rot);
561:         jacobian.block(3 * i, 15, 3, 3) = jacobian.block(3 * i, 15, 3, 3) * rot;
562:     }
563: }
564:
565: void QuadKD::getRotationMatrix(const Eigen::VectorXd &rpy,
566:                               Eigen::Matrix3d &rot) const {
567:     rot = Eigen::AngleAxisd(rpy(2), Eigen::Vector3d::UnitZ()) *
568:         Eigen::AngleAxisd(rpy(1), Eigen::Vector3d::UnitY()) *
569:         Eigen::AngleAxisd(rpy(0), Eigen::Vector3d::UnitX());
570: }
571:
572: void QuadKD::computeInverseDynamics(const Eigen::VectorXd &state_pos,
573:                                     const Eigen::VectorXd &state_vel,
574:                                     const Eigen::VectorXd &foot_acc,
575:                                     const Eigen::VectorXd &grf,
576:                                     const std::vector<int> &contact_mode,
577:                                     Eigen::VectorXd &tau) const {
578:     // Convert q, q_dot into RBDL order
579:     Eigen::VectorXd q(19), q_dot(18);
580:     q.setZero();
581:     q_dot.setZero();
582:
583:     q.head(3) = state_pos.segment(12, 3);
584:     q_dot.head(3) = state_vel.segment(12, 3);
585:
586:     tf2::Quaternion quat_tf;
587:     quat_tf.setRPY(state_pos(15), state_pos(16), state_pos(17));
588:     q(3) = quat_tf.getX();
589:     q(4) = quat_tf.getY();
590:     q(5) = quat_tf.getZ();
591:
592:     // RBDL uses quaternion for floating base direction, but w is placed at the
593:     // end of the state vector
594:     q(18) = quat_tf.getW();
595:
596:     q_dot.segment(3, 3) = state_vel.segment(15, 3);
597:
598:     // std::cout << "leg idx list size: " << leg_idx_list_.size() << "\n";
599:     for (size_t i = 0; i < leg_idx_list_.size(); i++) {
600:         q.segment(6 + 3 * i, 3) = state_pos.segment(3 * leg_idx_list_.at(i), 3);
601:         q_dot.segment(6 + 3 * i, 3) = state_vel.segment(3 * leg_idx_list_.at(i), 3);
602:     }
603:
604:     // Compute jacobians
605:     Eigen::MatrixXd jacobian = Eigen::MatrixXd::Zero(12, 18);
606:     jacobian.setZero();
607:     // std::cout << "body id list size: " << body_id_list_.size() << "\n"; //
608:     // Still note sure about this body_id_list_
609:     for (size_t i = 0; i < body_id_list_.size(); i++) {
610:         // std::cout << "i: " << i << "\n";
611:         // std::cout << "body id list: " << body_id_list_[i] << "\n";
612:         Eigen::MatrixXd jac_block(3, 18);
613:         jac_block.setZero();
614:         RigidBodyDynamics::CalcPointJacobian(*model_, q, body_id_list_.at(i),
615:                                               Eigen::Vector3d::Zero(), jac_block);
616:         jacobian.block(3 * i, 0, 3, 18) = jac_block;

```



```

617: }
618: /*
619: std::cout << "Jacobian block num coefficients: " << jacobian.size() <<
620: std::endl; std::cout << "Jacobian rows: " << jacobian.rows() << " columns: "
621: << jacobian.cols() << std::endl;
622: */
623: // Compute the equivalent force in generalized coordinates
624: Eigen::VectorXd tau_stance =
625:     -jacobian.transpose() * grf; // So this is for stance torque
626: /*
627: std::cout << "grf length: " << grf.size() << std::endl;
628: std::cout << "tau_st length: " << tau_stance.size() << std::endl;
629: */
630: // Compute EOM
631: Eigen::MatrixXd M(18, 18);
632: M.setZero();
633: Eigen::VectorXd N(18);
634: RigidBodyDynamics::CompositeRigidBodyAlgorithm(*model_, q, M);
635: RigidBodyDynamics::NonlinearEffects(*model_, q, q_dot, N);
636:
637: // Compute J_dot*q_dot
638: Eigen::VectorXd foot_acc_J_dot(12);
639: for (size_t i = 0; i < 4; i++) {
640:     foot_acc_J_dot.segment(3 * i, 3) = RigidBodyDynamics::CalcPointAcceleration(
641:         *model_, q, q_dot, Eigen::VectorXd::Zero(18),
642:         body_id_list_.at(
643:             i), // body id is toes... why is it called foot_acc_J_dot
644:         Eigen::Vector3d::Zero());
645: }
646:
647: // Compute constraint Jacobian A and A_dot*q_dot
648: int constraints_num =
649:     3 * std::count(contact_mode.begin(), contact_mode.end(), true);
650: // std::cout << "Number of constraints: " << constraints_num << std::endl;
651: Eigen::MatrixXd A(constraints_num, 18);
652: Eigen::VectorXd A_dotq_dot(constraints_num);
653: // std::cout << "A row: " << A.rows() << "cols: " << A.cols() << std::endl;
654: int constraints_count = 0;
655: for (size_t i = 0; i < 4; i++) {
656:     if (contact_mode.at(i)) {
657:         A.block(3 * constraints_count, 0, 3, 18) =
658:             jacobian.block(3 * i, 0, 3, 18);
659:         A_dotq_dot.segment(3 * constraints_count, 3) =
660:             foot_acc_J_dot.segment(3 * i, 3);
661:         constraints_count++;
662:     }
663: }
664:
665: // Compute acceleration from J*q_ddot
666: Eigen::VectorXd foot_acc_q_ddot =
667:     foot_acc - foot_acc_J_dot; // what does this mean... foot acceleration
668:                               // relative to global???
669: // std::cout << "foot_acc_q_ddot: " << foot_acc_q_ddot.size() << std::endl;
670:
671: // Compute damped jacobian inversed
672: Eigen::MatrixXd jacobian_inv =
673:     math_utils::sdlInv(jacobian.block(0, 6, 12, 12));
674:
675: // std::cout << "Jacobian inverse rows: " << jacobian_inv.rows() << " cols: "
676: // << jacobian_inv.cols() << std::endl;
677:
678: // In the EOM, we know M, N, tau_grf, and a = J_b*q_ddot_b + J_l*q_ddot_l, we
679: // need to solve q_ddot_b and tau_swing
680: Eigen::MatrixXd blk_mat =
681:     Eigen::MatrixXd::Zero(18 + constraints_num, 18 + constraints_num);
682: blk_mat.block(0, 0, 6, 6) =
683:     -M.block(0, 0, 6, 6) +
684:     M.block(0, 6, 6, 12) * jacobian_inv * jacobian.block(0, 0, 12, 6);
685: blk_mat.block(6, 0, 12, 6) =
686:     -M.block(6, 0, 12, 6) +
687:     M.block(6, 6, 12, 12) * jacobian_inv * jacobian.block(0, 0, 12, 6);
688: for (size_t i = 0; i < 4; i++) {
689:     if (!contact_mode.at(leg_idx_list_.at(i))) {
690:         blk_mat.block(3 * i + 6, 3 * i + 6, 3, 3).diagonal().fill(1);
691:     }
692: }
693: blk_mat.block(0, 18, 18, constraints_num) = -A.transpose();

```

```

694:   blk_mat.block(18, 0, constraints_num, 6) =
695:       -A.leftCols(6) +
696:       A.rightCols(12) * jacobian_inv * jacobian.block(0, 0, 12, 6);
697:
698:   // Perform inverse dynamics
699:   Eigen::VectorXd tau_swing(12), blk_sol(18 + constraints_num),
700:       blk_vec(18 + constraints_num);
701:   blk_vec.segment(0, 6) << N.segment(0, 6) + M.block(0, 6, 6, 12) *
702:       jacobian_inv * foot_acc_q_ddot;
703:   blk_vec.segment(6, 12) << N.segment(6, 12) +
704:       M.block(6, 6, 12, 12) * jacobian_inv *
705:       foot_acc_q_ddot -
706:       tau_stance.segment(6, 12);
707:   blk_vec.segment(18, constraints_num)
708:       << A.dotq_dot + A.leftCols(12) * jacobian_inv * foot_acc_q_ddot;
709:   blk_sol =
710:       blk_mat.bdcSvd(Eigen::ComputeThinU | Eigen::ComputeThinV).solve(blk_vec);
711:   tau_swing = blk_sol.segment(6, 12);
712:
713:   // Convert the order back
714:   for (size_t i = 0; i < 4; i++) {
715:       if (contact_mode.at(leg_idx_list_.at(i))) {
716:           tau.segment(3 * leg_idx_list_.at(i), 3) =
717:               tau_stance.segment(6 + 3 * i, 3);
718:       } else {
719:           tau.segment(3 * leg_idx_list_.at(i), 3) = tau_swing.segment(3 * i, 3);
720:           // tau.segment(3 * leg_idx_list_.at(i), 3) = Eigen::VectorXd::Zero(3);
721:       }
722:   }
723:
724:   // Check inf or nan
725:   if (!(tau.array() == tau.array()).all() ||
726:       !((tau - tau).array() == (tau - tau).array()).all()) {
727:       tau.setZero();
728:   }
729: }
730:
731: bool QuadKD::convertCentroidalToFullBody(const Eigen::VectorXd &body_state,
732:     const Eigen::VectorXd &foot_positions,
733:     const Eigen::VectorXd &foot_velocities,
734:     const Eigen::VectorXd &grfs,
735:     Eigen::VectorXd &joint_positions,
736:     Eigen::VectorXd &joint_velocities,
737:     Eigen::VectorXd &torques) {
738:   // Assume the conversion is exact unless a check below fails
739:   bool is_exact = true;
740:
741:   // Extract kinematic quantities
742:   Eigen::Vector3d body_pos = body_state.segment<3>(0);
743:   Eigen::Vector3d body_rpy = body_state.segment<3>(3);
744:
745:   auto t_start = std::chrono::steady_clock::now();
746:   // Perform IK for each leg
747:   for (int i = 0; i < num_feet_; i++) {
748:       Eigen::Vector3d leg_joint_state;
749:       Eigen::Vector3d foot_pos = foot_positions.segment<3>(3 * i);
750:       is_exact = is_exact && worldToFootIKWorldFrame(i, body_pos, body_rpy,
751:           foot_pos, leg_joint_state);
752:       joint_positions.segment<3>(3 * i) = leg_joint_state;
753:   }
754:
755:   auto t_ik = std::chrono::steady_clock::now();
756:
757:   // Load state positions
758:   Eigen::VectorXd state_positions(18), state_velocities(18);
759:   state_positions << joint_positions, body_pos, body_rpy;
760:
761:   // Compute jacobian
762:   Eigen::MatrixXd jacobian = Eigen::MatrixXd::Zero(12, 18);
763:   getJacobianBodyAngVel(state_positions, jacobian);
764:
765:   auto t_jacob = std::chrono::steady_clock::now();
766:
767:   // Compute joint velocities
768:   joint_velocities = jacobian.leftCols(12).colPivHouseholderQr().solve(
769:       foot_velocities - jacobian.rightCols(6) * body_state.tail(6));
770:   state_velocities << joint_velocities, body_state.tail(6);

```

```

771:
772:     auto t_ik_vel = std::chrono::steady_clock::now();
773:
774:     torques = -jacobian.leftCols(12).transpose() * grfs;
775:
776:     // computeInverseDynamics(state_positions, state_velocities, foot_acc, grfs,
777:     // contact_mode, torques);
778:
779:     auto t_id = std::chrono::steady_clock::now();
780:
781:     std::chrono::duration<double> t_diff_ik =
782:         std::chrono::duration_cast<std::chrono::duration<double>>(t_ik - t_start);
783:     std::chrono::duration<double> t_diff_jacob =
784:         std::chrono::duration_cast<std::chrono::duration<double>>(t_jacob - t_ik);
785:     std::chrono::duration<double> t_diff_ik_vel =
786:         std::chrono::duration_cast<std::chrono::duration<double>>(t_ik_vel -
787:             t_jacob);
788:     std::chrono::duration<double> t_diff_id =
789:         std::chrono::duration_cast<std::chrono::duration<double>>(t_id -
790:             t_ik_vel);
791:
792:     // std::cout << "t_diff_ik = " << t_diff_ik.count() << std::endl;
793:     // std::cout << "t_diff_jacob = " << t_diff_jacob.count() << std::endl;
794:     // std::cout << "t_diff_ik_vel = " << t_diff_ik_vel.count() << std::endl;
795:     // std::cout << "t_diff_id = " << t_diff_id.count() << std::endl;
796:
797:     return is_exact;
798: }
799:
800: bool QuadKD::applyMotorModel(const Eigen::VectorXd &torques,
801:                             Eigen::VectorXd &constrained_torques) {
802:     // Constrain torques to max values
803:     constrained_torques.resize(torques.size());
804:     constrained_torques = torques.cwiseMax(-tau_max_).cwiseMin(tau_max_);
805:
806:     // Check if torques was modified
807:     return constrained_torques.isApprox(torques);
808: }
809:
810: bool QuadKD::applyMotorModel(const Eigen::VectorXd &joint_torques,
811:                             const Eigen::VectorXd &joint_velocities,
812:                             Eigen::VectorXd &constrained_joint_torques) {
813:     // Constrain torques to max values
814:     Eigen::VectorXd constraint_violation(joint_torques.size());
815:     constrained_joint_torques.resize(joint_torques.size());
816:     constrained_joint_torques =
817:         joint_torques.cwiseMax(-tau_max_).cwiseMin(tau_max_);
818:
819:     // Apply linear motor model
820:     Eigen::VectorXd emf = joint_velocities.cwiseProduct(mmm_slope_);
821:     constrained_joint_torques =
822:         constrained_joint_torques.cwiseMax(-tau_max_ - emf)
823:         .cwiseMin(tau_max_ - emf);
824:
825:     // Check if torques were modified
826:     return constrained_joint_torques.isApprox(joint_torques);
827: }
828:
829: bool QuadKD::isValidFullState(const Eigen::VectorXd &body_state,
830:                              const Eigen::VectorXd &joint_state,
831:                              const Eigen::VectorXd &joint_torques,
832:                              const grid_map::GridMap &terrain,
833:                              Eigen::VectorXd &state_violation,
834:                              Eigen::VectorXd &control_violation) {
835:     // Check state constraints
836:     // Kinematics
837:     state_violation.setZero(num_feet_);
838:     for (int i = 0; i < num_feet_; i++) {
839:         Eigen::Vector3d knee_pos_world;
840:         worldToKneeFKWorldFrame(i, body_state.segment<3>(0),
841:                                 body_state.segment<3>(3),
842:                                 joint_state.segment<3>(3 * i), knee_pos_world);
843:         state_violation[i] = getGroundClearance(knee_pos_world, terrain);
844:     }
845:     bool state_valid = (state_violation.array() >= 0).all();
846:
847:     // Check control constraints

```

```
848: // Motor model
849: Eigen::VectorXd constrained_joint_torques(12);
850: bool control_valid = applyMotorModel(joint_torques, joint_state.tail(12),
851:                                     constrained_joint_torques);
852: control_violation.setZero(joint_torques.size());
853: control_violation = -(constrained_joint_torques - joint_torques).cwiseAbs();
854:
855: // Only valid if each subcheck is valid
856: return (state_valid && control_valid);
857: }
858:
859: bool QuadKD::isValidCentroidalState(
860:     const Eigen::VectorXd &body_state, const Eigen::VectorXd &foot_positions,
861:     const Eigen::VectorXd &foot_velocities, const Eigen::VectorXd &grfs,
862:     const grid_map::GridMap &terrain, Eigen::VectorXd &joint_positions,
863:     Eigen::VectorXd &joint_velocities, Eigen::VectorXd &joint_torques,
864:     Eigen::VectorXd &state_violation, Eigen::VectorXd &control_violation) {
865: // Convert to full
866: bool is_exact = convertCentroidalToFullBody(
867:     body_state, foot_positions, foot_velocities, grfs, joint_positions,
868:     joint_velocities, joint_torques);
869:
870: Eigen::VectorXd joint_state(24);
871: joint_state << joint_positions, joint_velocities;
872: bool is_valid = isValidFullState(body_state, joint_state, joint_torques,
873:                                 terrain, state_violation, control_violation);
874:
875: return (is_exact && is_valid);
876: }
```

```
1: #include "quad_utils/rviz_interface.h"
2:
3: RVizInterface::RVizInterface(ros::NodeHandle nh) {
4:     nh_ = nh;
5:
6:     // Load rosparms from parameter server
7:     std::string global_plan_topic, local_plan_topic, discrete_global_plan_topic,
8:         foot_plan_discrete_topic, foot_plan_continuous_topic,
9:         state_estimate_topic, ground_truth_state_topic, trajectory_state_topic,
10:         grf_topic;
11:
12:     // Load topic names from parameter server
13:     quad_utils::loadROSParam(nh_, "topics/global_plan", global_plan_topic);
14:     quad_utils::loadROSParam(nh_, "topics/local_plan", local_plan_topic);
15:     quad_utils::loadROSParam(nh_, "topics/control/grfs", grf_topic);
16:     quad_utils::loadROSParam(nh_, "topics/global_plan_discrete",
17:         discrete_global_plan_topic);
18:     quad_utils::loadROSParam(nh_, "topics/foot_plan_discrete",
19:         foot_plan_discrete_topic);
20:     quad_utils::loadROSParam(nh_, "topics/foot_plan_continuous",
21:         foot_plan_continuous_topic);
22:     quad_utils::loadROSParam(nh_, "topics/state/estimate", state_estimate_topic);
23:     quad_utils::loadROSParam(nh_, "topics/state/ground_truth",
24:         ground_truth_state_topic);
25:     quad_utils::loadROSParam(nh_, "topics/state/trajectory",
26:         trajectory_state_topic);
27:     quad_utils::loadROSParamDefault(nh_, "tf_prefix", tf_prefix_,
28:         std::string(""));
29:
30:     std::string global_plan_viz_topic, local_plan_viz_topic,
31:         local_plan_ori_viz_topic, global_plan_grf_viz_topic,
32:         local_plan_grf_viz_topic, current_grf_viz_topic,
33:         discrete_body_plan_viz_topic, foot_plan_discrete_viz_topic,
34:         estimate_joint_states_viz_topic, ground_truth_joint_states_viz_topic,
35:         trajectory_joint_states_viz_topic, state_estimate_trace_viz_topic,
36:         ground_truth_trace_viz_topic, trajectory_state_trace_viz_topic;
37:
38:     quad_utils::loadROSParam(nh_, "topics/visualization/global_plan",
39:         global_plan_viz_topic);
40:     quad_utils::loadROSParam(nh_, "topics/visualization/local_plan",
41:         local_plan_viz_topic);
42:     quad_utils::loadROSParam(nh_, "topics/visualization/local_plan_ori",
43:         local_plan_ori_viz_topic);
44:     quad_utils::loadROSParam(nh_, "topics/visualization/global_plan_grf",
45:         global_plan_grf_viz_topic);
46:     quad_utils::loadROSParam(nh_, "topics/visualization/local_plan_grf",
47:         local_plan_grf_viz_topic);
48:     quad_utils::loadROSParam(nh_, "topics/visualization/current_grf",
49:         current_grf_viz_topic);
50:     quad_utils::loadROSParam(nh_, "topics/visualization/global_plan_discrete",
51:         discrete_body_plan_viz_topic);
52:     quad_utils::loadROSParam(nh_, "topics/visualization/foot_plan_discrete",
53:         foot_plan_discrete_viz_topic);
54:     quad_utils::loadROSParam(nh_, "topics/visualization/joint_states/estimate",
55:         estimate_joint_states_viz_topic);
56:     quad_utils::loadROSParam(nh_,
57:         "topics/visualization/joint_states/ground_truth",
58:         ground_truth_joint_states_viz_topic);
59:     quad_utils::loadROSParam(nh_, "topics/visualization/joint_states/trajectory",
60:         trajectory_joint_states_viz_topic);
61:     quad_utils::loadROSParam(nh_, "topics/visualization/state/estimate_trace",
62:         state_estimate_trace_viz_topic);
63:     quad_utils::loadROSParam(nh_, "topics/visualization/state/ground_truth_trace",
64:         ground_truth_trace_viz_topic);
65:     quad_utils::loadROSParam(nh_, "topics/visualization/state/trajectory_trace",
66:         trajectory_state_trace_viz_topic);
67:
68:     // Setup rviz_interface parameters
69:     quad_utils::loadROSParam(nh_, "/map_frame", map_frame_);
70:     quad_utils::loadROSParam(nh_, "rviz_interface/update_rate", update_rate_);
71:     quad_utils::loadROSParam(nh_, "rviz_interface/colors/front_left",
72:         front_left_color_);
73:     quad_utils::loadROSParam(nh_, "rviz_interface/colors/back_left",
74:         back_left_color_);
75:     quad_utils::loadROSParam(nh_, "rviz_interface/colors/front_right",
76:         front_right_color_);
77:     quad_utils::loadROSParam(nh_, "rviz_interface/colors/back_right",
```

```
78:         back_right_color_);
79:     quad_utils::loadROSParam(nh_, "/rviz_interface/colors/net_grf",
80:         net_grf_color_);
81:     quad_utils::loadROSParam(nh_, "/rviz_interface/colors/individual_grf",
82:         individual_grf_color_);
83:
84:     double period, dt;
85:     quad_utils::loadROSParam(nh_, "/local_footstep_planner/period", period);
86:     quad_utils::loadROSParam(nh_, "/local_planner/timestep", dt);
87:     orientation_subsample_interval_ = int(period / dt);
88:
89:     // Setup plan subs
90:     global_plan_sub_ = nh_.subscribe<quad_msgs::RobotPlan>(
91:         global_plan_topic, 1,
92:         boost::bind(&RVizInterface::robotPlanCallback, this, _1, GLOBAL));
93:     local_plan_sub_ = nh_.subscribe<quad_msgs::RobotPlan>(
94:         local_plan_topic, 1,
95:         boost::bind(&RVizInterface::robotPlanCallback, this, _1, LOCAL));
96:     grf_sub_ = nh_.subscribe(grf_topic, 1, &RVizInterface::grfCallback, this);
97:     foot_plan_discrete_sub_ =
98:         nh_.subscribe(foot_plan_discrete_topic, 1,
99:             &RVizInterface::footPlanDiscreteCallback, this);
100:     foot_plan_continuous_sub_ =
101:         nh_.subscribe(foot_plan_continuous_topic, 1,
102:             &RVizInterface::footPlanContinuousCallback, this);
103:
104:     // Setup plan visual pubs
105:     global_plan_viz_pub_ =
106:         nh_.advertise<visualization_msgs::Marker>(global_plan_viz_topic, 1);
107:     local_plan_viz_pub_ =
108:         nh_.advertise<visualization_msgs::Marker>(local_plan_viz_topic, 1);
109:     global_plan_grf_viz_pub_ = nh_.advertise<visualization_msgs::MarkerArray>(
110:         global_plan_grf_viz_topic, 1);
111:     local_plan_grf_viz_pub_ = nh_.advertise<visualization_msgs::MarkerArray>(
112:         local_plan_grf_viz_topic, 1);
113:     current_grf_viz_pub_ =
114:         nh_.advertise<visualization_msgs::MarkerArray>(current_grf_viz_topic, 1);
115:     discrete_body_plan_viz_pub_ = nh_.advertise<visualization_msgs::Marker>(
116:         discrete_body_plan_viz_topic, 1);
117:     foot_plan_discrete_viz_pub_ = nh_.advertise<visualization_msgs::Marker>(
118:         foot_plan_discrete_viz_topic, 1);
119:     local_plan_ori_viz_pub_ =
120:         nh_.advertise<geometry_msgs::PoseArray>(local_plan_ori_viz_topic, 1);
121:
122:     // Setup publishers for state traces
123:     state_estimate_trace_pub_ = nh_.advertise<visualization_msgs::Marker>(
124:         state_estimate_trace_viz_topic, 1);
125:     ground_truth_state_trace_pub_ = nh_.advertise<visualization_msgs::Marker>(
126:         ground_truth_trace_viz_topic, 1);
127:     trajectory_state_trace_pub_ = nh_.advertise<visualization_msgs::Marker>(
128:         trajectory_state_trace_viz_topic, 1);
129:
130:     // Setup state subs to call the same callback but with pub ID included
131:     state_estimate_sub_ = nh_.subscribe<quad_msgs::RobotState>(
132:         state_estimate_topic, 1,
133:         boost::bind(&RVizInterface::robotStateCallback, this, _1, ESTIMATE));
134:     ground_truth_state_sub_ = nh_.subscribe<quad_msgs::RobotState>(
135:         ground_truth_state_topic, 1,
136:         boost::bind(&RVizInterface::robotStateCallback, this, _1, GROUND_TRUTH));
137:     trajectory_state_sub_ = nh_.subscribe<quad_msgs::RobotState>(
138:         trajectory_state_topic, 1,
139:         boost::bind(&RVizInterface::robotStateCallback, this, _1, TRAJECTORY));
140:
141:     // Setup state visual pubs
142:     estimate_joint_states_viz_pub_ = nh_.advertise<sensor_msgs::JointState>(
143:         estimate_joint_states_viz_topic, 1);
144:     ground_truth_joint_states_viz_pub_ = nh_.advertise<sensor_msgs::JointState>(
145:         ground_truth_joint_states_viz_topic, 1);
146:     trajectory_joint_states_viz_pub_ = nh_.advertise<sensor_msgs::JointState>(
147:         trajectory_joint_states_viz_topic, 1);
148:
149:     std::string foot_0_plan_continuous_viz_topic,
150:         foot_1_plan_continuous_viz_topic, foot_2_plan_continuous_viz_topic,
151:         foot_3_plan_continuous_viz_topic;
152:
153:     quad_utils::loadROSParam(nh_, "topics/visualization/foot_0_plan_continuous",
154:         foot_0_plan_continuous_viz_topic);
```

```
155: quad_utils::loadROSParam(nh_, "topics/visualization/foot_1_plan_continuous",
156:                             foot_1_plan_continuous_viz_topic);
157: quad_utils::loadROSParam(nh_, "topics/visualization/foot_2_plan_continuous",
158:                             foot_2_plan_continuous_viz_topic);
159: quad_utils::loadROSParam(nh_, "topics/visualization/foot_3_plan_continuous",
160:                             foot_3_plan_continuous_viz_topic);
161:
162: foot_0_plan_continuous_viz_pub_ =
163:     nh_.advertise<nav_msgs::Path>(foot_0_plan_continuous_viz_topic, 1);
164: foot_1_plan_continuous_viz_pub_ =
165:     nh_.advertise<nav_msgs::Path>(foot_1_plan_continuous_viz_topic, 1);
166: foot_2_plan_continuous_viz_pub_ =
167:     nh_.advertise<nav_msgs::Path>(foot_2_plan_continuous_viz_topic, 1);
168: foot_3_plan_continuous_viz_pub_ =
169:     nh_.advertise<nav_msgs::Path>(foot_3_plan_continuous_viz_topic, 1);
170:
171: // Initialize Path message to visualize body plan
172: state_estimate_trace_msg_.action = visualization_msgs::Marker::ADD;
173: state_estimate_trace_msg_.pose.orientation.w = 1;
174: state_estimate_trace_msg_.type = visualization_msgs::Marker::LINE_STRIP;
175: state_estimate_trace_msg_.scale.x = 0.02;
176: state_estimate_trace_msg_.header.frame_id = map_frame_;
177: geometry_msgs::Point dummy_point;
178: state_estimate_trace_msg_.points.push_back(dummy_point);
179: ground_truth_state_trace_msg_ = state_estimate_trace_msg_;
180: trajectory_state_trace_msg_ = state_estimate_trace_msg_;
181:
182: // Define visual properties for traces
183: state_estimate_trace_msg_.id = 5;
184: state_estimate_trace_msg_.color.a = 1.0;
185: state_estimate_trace_msg_.color.r = (float)front_left_color_[0] / 255.0;
186: state_estimate_trace_msg_.color.g = (float)front_left_color_[1] / 255.0;
187: state_estimate_trace_msg_.color.b = (float)front_left_color_[2] / 255.0;
188:
189: ground_truth_state_trace_msg_.id = 6;
190: ground_truth_state_trace_msg_.color.a = 1.0;
191: ground_truth_state_trace_msg_.color.r = (float)back_left_color_[0] / 255.0;
192: ground_truth_state_trace_msg_.color.g = (float)back_left_color_[1] / 255.0;
193: ground_truth_state_trace_msg_.color.b = (float)back_left_color_[2] / 255.0;
194:
195: trajectory_state_trace_msg_.id = 7;
196: trajectory_state_trace_msg_.color.a = 1.0;
197: trajectory_state_trace_msg_.color.r = (float)front_right_color_[0] / 255.0;
198: trajectory_state_trace_msg_.color.g = (float)front_right_color_[1] / 255.0;
199: trajectory_state_trace_msg_.color.b = (float)front_right_color_[2] / 255.0;
200: }
201:
202: void RVizInterface::robotPlanCallback(const quad_msgs::RobotPlan::ConstPtr &msg,
203:                                       const int pub_id) {
204:     // Initialize Path message to visualize body plan
205:     visualization_msgs::Marker body_plan_viz;
206:     body_plan_viz.header = msg->header;
207:     body_plan_viz.action = visualization_msgs::Marker::ADD;
208:     body_plan_viz.pose.orientation.w = 1;
209:     body_plan_viz.id = 5;
210:     body_plan_viz.type = visualization_msgs::Marker::LINE_STRIP;
211:     body_plan_viz.scale.x = 0.03;
212:
213:     // Construct MarkerArray for body plan orientation
214:     geometry_msgs::PoseArray body_plan_ori_viz;
215:     body_plan_ori_viz.header = msg->header;
216:
217:     // Loop through the BodyPlan message to get the state info
218:     int length = msg->states.size();
219:     for (int i = 0; i < length; i++) {
220:         // Load in the pose data directly from the Odometry message
221:         geometry_msgs::PoseStamped pose_stamped;
222:         pose_stamped.header = msg->states[i].header;
223:         pose_stamped.pose = msg->states[i].body.pose;
224:
225:         std_msgs::ColorRGBA color;
226:         color.a = 1;
227:         if (pub_id == LOCAL) {
228:             color.g = 1.0;
229:         } else {
230:             if (msg->primitive_ids[i] == FLIGHT) {
231:                 color.r = (float)back_left_color_[0] / 255.0;
```



```

232:     color.g = (float)back_left_color_[1] / 255.0;
233:     color.b = (float)back_left_color_[2] / 255.0;
234:   } else if (msg->primitive_ids[i] == LEAP_STANCE ||
235:             msg->primitive_ids[i] == LAND_STANCE) {
236:     color.r = (float)front_right_color_[0] / 255.0;
237:     color.g = (float)front_right_color_[1] / 255.0;
238:     color.b = (float)front_right_color_[2] / 255.0;
239:   } else if (msg->primitive_ids[i] == CONNECT) {
240:     color.r = (float)front_left_color_[0] / 255.0;
241:     color.g = (float)front_left_color_[1] / 255.0;
242:     color.b = (float)front_left_color_[2] / 255.0;
243:   } else {
244:     ROS_WARN_THROTTLE(1, "Invalid primitive ID received in RViz interface");
245:   }
246: }
247: body_plan_viz.colors.push_back(color);
248: body_plan_viz.points.push_back(msg->states[i].body.pose.position);
249:
250: // Add poses to the orientation message
251: if (i % orientation_subsample_interval_ == 0) {
252:   body_plan_ori_viz.poses.push_back(pose_stamped.pose);
253: }
254: }
255:
256: // Publish the full path
257: if (pub_id == GLOBAL) {
258:   global_plan_viz_pub_.publish(body_plan_viz);
259: } else if (pub_id == LOCAL) {
260:   local_plan_viz_pub_.publish(body_plan_viz);
261:   local_plan_ori_viz_pub_.publish(body_plan_ori_viz);
262: }
263:
264: // Construct MarkerArray and Marker message for GRFs
265: visualization_msgs::MarkerArray grfs_viz_msg;
266: visualization_msgs::Marker marker;
267:
268: // Initialize the headers and types
269: marker.header = msg->header;
270: marker.type = visualization_msgs::Marker::ARROW;
271:
272: // Define the shape of the discrete states
273: double arrow_diameter = 0.01;
274: marker.scale.x = arrow_diameter;
275: marker.scale.y = 4 * arrow_diameter;
276: marker.color.g = 0.733f;
277: marker.pose.orientation.w = 1.0;
278:
279: for (int i = 0; i < length; i++) {
280:   for (int j = 0; j < msg->grfs[i].vectors.size(); j++) {
281:     // Reset the marker message
282:     marker.points.clear();
283:     marker.color.a = 1.0;
284:     marker.id = i * msg->grfs[i].vectors.size() + j;
285:
286:     if (msg->grfs[i].vectors.size() > 1) {
287:       marker.color.r = (float)individual_grf_color_[0] / 255.0;
288:       marker.color.g = (float)individual_grf_color_[1] / 255.0;
289:       marker.color.b = (float)individual_grf_color_[2] / 255.0;
290:     } else {
291:       marker.color.r = (float)net_grf_color_[0] / 255.0;
292:       marker.color.g = (float)net_grf_color_[1] / 255.0;
293:       marker.color.b = (float)net_grf_color_[2] / 255.0;
294:     }
295:
296:     // Define point messages for the base and tip of each GRF arrow
297:     geometry_msgs::Point p_base, p_tip;
298:     p_base = msg->grfs[i].points[j];
299:
300:     // Define the endpoint of the GRF arrow
301:     double grf_length_scale = 0.002;
302:     p_tip.x = p_base.x + grf_length_scale * msg->grfs[i].vectors[j].x;
303:     p_tip.y = p_base.y + grf_length_scale * msg->grfs[i].vectors[j].y;
304:     p_tip.z = p_base.z + grf_length_scale * msg->grfs[i].vectors[j].z;
305:
306:     // if GRF = 0, set alpha to zero
307:     if (msg->grfs[i].contact_states[j] == false) {
308:       marker.color.a = 0.0;

```



```
309:     }
310:
311:     // Add the points to the marker and add the marker to the array
312:     marker.points.push_back(p_base);
313:     marker.points.push_back(p_tip);
314:     grfs_viz_msg.markers.push_back(marker);
315: }
316: }
317:
318: // Publish grfs
319: if (pub_id == GLOBAL) {
320:     global_plan_grf_viz_pub_.publish(grfs_viz_msg);
321: } else if (pub_id == LOCAL) {
322:     local_plan_grf_viz_pub_.publish(grfs_viz_msg);
323: }
324: }
325:
326: void RVizInterface::grfCallback(const quad_msgs::GRFArray::ConstPtr &msg) {
327:     if (msg->vectors.empty()) {
328:         return;
329:     }
330:
331:     // Construct MarkerArray and Marker message for GRFs
332:     visualization_msgs::MarkerArray grfs_viz_msg;
333:     visualization_msgs::Marker marker;
334:
335:     // Initialize the headers and types
336:     marker.header = msg->header;
337:     marker.type = visualization_msgs::Marker::ARROW;
338:
339:     // Define the shape of the discrete states
340:     double arrow_diameter = 0.01;
341:     marker.scale.x = arrow_diameter;
342:     marker.scale.y = 4 * arrow_diameter;
343:     marker.color.g = 0.733f;
344:     marker.pose.orientation.w = 1.0;
345:
346:     for (int i = 0; i < msg->vectors.size(); i++) {
347:         // Reset the marker message
348:         marker.points.clear();
349:         marker.color.a = 1.0;
350:         marker.id = i;
351:
352:         marker.color.r = (float)individual_grf_color_[0] / 255.0;
353:         marker.color.g = (float)individual_grf_color_[1] / 255.0;
354:         marker.color.b = (float)individual_grf_color_[2] / 255.0;
355:
356:         // Define point messages for the base and tip of each GRF arrow
357:         geometry_msgs::Point p_base, p_tip;
358:         p_base = msg->points[i];
359:
360:         /// Define the endpoint of the GRF arrow
361:         double grf_length_scale = 0.002;
362:         p_tip.x = p_base.x + grf_length_scale * msg->vectors[i].x;
363:         p_tip.y = p_base.y + grf_length_scale * msg->vectors[i].y;
364:         p_tip.z = p_base.z + grf_length_scale * msg->vectors[i].z;
365:
366:         // if GRF = 0, set alpha to zero
367:         if (msg->contact_states[i] == false) {
368:             marker.color.a = 0.0;
369:         }
370:
371:         // Add the points to the marker and add the marker to the array
372:         marker.points.push_back(p_base);
373:         marker.points.push_back(p_tip);
374:         grfs_viz_msg.markers.push_back(marker);
375:     }
376:
377:     current_grf_viz_pub_.publish(grfs_viz_msg);
378: }
379:
380: void RVizInterface::discreteBodyPlanCallback(
381:     const quad_msgs::RobotPlan::ConstPtr &msg) {
382:     // Construct Marker message
383:     visualization_msgs::Marker discrete_body_plan;
384:
385:     // Initialize the headers and types
```

```
386: discrete_body_plan.header = msg->header;
387: discrete_body_plan.id = 0;
388: discrete_body_plan.type = visualization_msgs::Marker::POINTS;
389:
390: // Define the shape of the discrete states
391: double scale = 0.2;
392: discrete_body_plan.scale.x = scale;
393: discrete_body_plan.scale.y = scale;
394: discrete_body_plan.color.r = 0.733f;
395: discrete_body_plan.color.a = 1.0;
396:
397: // Loop through the discrete states
398: int length = msg->states.size();
399: for (int i = 0; i < length; i++) {
400:     geometry_msgs::Point p;
401:     p.x = msg->states[i].body.pose.position.x;
402:     p.y = msg->states[i].body.pose.position.y;
403:     p.z = msg->states[i].body.pose.position.z;
404:     discrete_body_plan.points.push_back(p);
405: }
406:
407: // Publish both interpolated body plan and discrete states
408: discrete_body_plan_viz_pub_.publish(discrete_body_plan);
409: }
410:
411: void RVizInterface::footPlanDiscreteCallback(
412:     const quad_msgs::MultiFootPlanDiscrete::ConstPtr &msg) {
413:     // Initialize Marker message to visualize footstep plan as points
414:     visualization_msgs::Marker points;
415:     points.header = msg->header;
416:     points.action = visualization_msgs::Marker::ADD;
417:     points.pose.orientation.w = 1.0;
418:     points.id = 0;
419:     points.type = visualization_msgs::Marker::SPHERE_LIST;
420:
421:     // POINTS markers use x and y scale for width/height respectively
422:     points.scale.x = 0.05;
423:     points.scale.y = 0.05;
424:     points.scale.z = 0.05;
425:
426:     // Loop through each foot
427:     int num_feet = msg->feet.size();
428:     for (int i = 0; i < num_feet; ++i) {
429:         // Loop through footstep in the plan
430:         int num_steps = msg->feet[i].footholds.size();
431:         for (int j = 0; j < num_steps; ++j) {
432:             // Create point message from FootstepPlan message, adjust height
433:             geometry_msgs::Point p;
434:             p.x = msg->feet[i].footholds[j].position.x;
435:             p.y = msg->feet[i].footholds[j].position.y;
436:             p.z = msg->feet[i].footholds[j].position.z;
437:
438:             // Set the color of each marker (green for front, blue for back)
439:             std_msgs::ColorRGBA color;
440:             color.a = 1.0;
441:             if (i == 0) {
442:                 color.r = (float)front_left_color_[0] / 255.0;
443:                 color.g = (float)front_left_color_[1] / 255.0;
444:                 color.b = (float)front_left_color_[2] / 255.0;
445:             } else if (i == 1) {
446:                 color.r = (float)back_left_color_[0] / 255.0;
447:                 color.g = (float)back_left_color_[1] / 255.0;
448:                 color.b = (float)back_left_color_[2] / 255.0;
449:             } else if (i == 2) {
450:                 color.r = (float)front_right_color_[0] / 255.0;
451:                 color.g = (float)front_right_color_[1] / 255.0;
452:                 color.b = (float)front_right_color_[2] / 255.0;
453:             } else if (i == 3) {
454:                 color.r = (float)back_right_color_[0] / 255.0;
455:                 color.g = (float)back_right_color_[1] / 255.0;
456:                 color.b = (float)back_right_color_[2] / 255.0;
457:             }
458:
459:             // Add to the Marker message
460:             points.colors.push_back(color);
461:             points.points.push_back(p);
462:         }
463:     }
```

```

463: }
464:
465: // Publish the full marker array
466: foot_plan_discrete_viz_pub_.publish(points);
467: }
468:
469: void RVizInterface::footPlanContinuousCallback(
470:     const quad_msgs::MultiFootPlanContinuous::ConstPtr &msg) {
471:     std::vector<nav_msgs::Path> foot_paths;
472:     foot_paths.resize(4);
473:
474:     for (int j = 0; j < msg->states[0].feet.size(); j++) {
475:         foot_paths[j].header.frame_id = map_frame_;
476:
477:         for (int i = 0; i < msg->states.size(); i++) {
478:             geometry_msgs::PoseStamped foot;
479:             foot.pose.position.x = msg->states[i].feet[j].position.x;
480:             foot.pose.position.y = msg->states[i].feet[j].position.y;
481:             foot.pose.position.z = msg->states[i].feet[j].position.z;
482:
483:             foot_paths[j].poses.push_back(foot);
484:         }
485:     }
486:
487:     foot_0_plan_continuous_viz_pub_.publish(foot_paths[0]);
488:     foot_1_plan_continuous_viz_pub_.publish(foot_paths[1]);
489:     foot_2_plan_continuous_viz_pub_.publish(foot_paths[2]);
490:     foot_3_plan_continuous_viz_pub_.publish(foot_paths[3]);
491: }
492:
493: void RVizInterface::robotStateCallback(
494:     const quad_msgs::RobotState::ConstPtr &msg, const int pub_id) {
495:     // Make a transform message for the body, populate with state estimate data
496:     geometry_msgs::TransformStamped transformStamped;
497:     transformStamped.header = msg->header;
498:     transformStamped.header.stamp = ros::Time::now();
499:     transformStamped.header.frame_id = map_frame_;
500:     transformStamped.transform.translation.x = msg->body.pose.position.x;
501:     transformStamped.transform.translation.y = msg->body.pose.position.y;
502:     transformStamped.transform.translation.z = msg->body.pose.position.z;
503:     transformStamped.transform.rotation = msg->body.pose.orientation;
504:
505:     // Copy the joint portion of the state estimate message to a new message
506:     sensor_msgs::JointState joint_msg;
507:     joint_msg = msg->joints;
508:
509:     // Set the header to the main header of the state estimate message and publish
510:     joint_msg.header = msg->header;
511:     joint_msg.header.stamp = ros::Time::now();
512:
513:     Eigen::Vector3d current_pos, last_pos;
514:     quad_utils::pointMsgToEigen(msg->body.pose.position, current_pos);
515:
516:     if (pub_id == ESTIMATE) {
517:         transformStamped.child_frame_id = tf_prefix_ + "_estimate/body";
518:         estimate_base_tf_br_.sendTransform(transformStamped);
519:         estimate_joint_states_viz_pub_.publish(joint_msg);
520:
521:         quad_utils::pointMsgToEigen(state_estimate_trace_msg_.points.back(),
522:                                     last_pos);
523:
524:         // Erase trace if state displacement exceeds threshold, otherwise show
525:         if ((current_pos - last_pos).norm() >= trace_reset_threshold_) {
526:             state_estimate_trace_msg_.action = visualization_msgs::Marker::DELETEALL;
527:             state_estimate_trace_msg_.points.clear();
528:         } else {
529:             state_estimate_trace_msg_.action = visualization_msgs::Marker::ADD;
530:         }
531:
532:         state_estimate_trace_msg_.points.push_back(msg->body.pose.position);
533:         state_estimate_trace_msg_.header.stamp = joint_msg.header.stamp;
534:         state_estimate_trace_pub_.publish(state_estimate_trace_msg_);
535:
536:     } else if (pub_id == GROUND_TRUTH) {
537:         transformStamped.child_frame_id = tf_prefix_ + "_ground_truth/body";
538:
539:         ground_truth_base_tf_br_.sendTransform(transformStamped);

```

```
540:     ground_truth_joint_states_viz_pub_.publish(joint_msg);
541:
542:     quad_utils::pointMsgToEigen(ground_truth_state_trace_msg_.points.back(),
543:                                   last_pos);
544:
545:     // Erase trace if state displacement exceeds threshold, otherwise show
546:     if ((current_pos - last_pos).norm() >= trace_reset_threshold_) {
547:         ground_truth_state_trace_msg_.action =
548:             visualization_msgs::Marker::DELETEALL;
549:         ground_truth_state_trace_msg_.points.clear();
550:     } else {
551:         ground_truth_state_trace_msg_.action = visualization_msgs::Marker::ADD;
552:     }
553:
554:     ground_truth_state_trace_msg_.points.push_back(msg->body.pose.position);
555:     ground_truth_state_trace_msg_.header.stamp = joint_msg.header.stamp;
556:     ground_truth_state_trace_pub_.publish(ground_truth_state_trace_msg_);
557:
558: } else if (pub_id == TRAJECTORY) {
559:     transformStamped.child_frame_id = tf_prefix_ + "_trajectory/body";
560:     trajectory_base_tf_br_.sendTransform(transformStamped);
561:     trajectory_joint_states_viz_pub_.publish(joint_msg);
562:
563:     quad_utils::pointMsgToEigen(trajectory_state_trace_msg_.points.back(),
564:                                   last_pos);
565:
566:     // Erase trace if state displacement exceeds threshold, otherwise show
567:     if ((current_pos - last_pos).norm() >= trace_reset_threshold_) {
568:         trajectory_state_trace_msg_.action =
569:             visualization_msgs::Marker::DELETEALL;
570:         trajectory_state_trace_msg_.points.clear();
571:     } else {
572:         trajectory_state_trace_msg_.action = visualization_msgs::Marker::ADD;
573:     }
574:
575:     trajectory_state_trace_msg_.points.push_back(msg->body.pose.position);
576:     trajectory_state_trace_msg_.header.stamp = joint_msg.header.stamp;
577:     trajectory_state_trace_pub_.publish(trajectory_state_trace_msg_);
578: } else {
579:     ROS_WARN_THROTTLE(
580:         0.5, "Invalid publisher id, not publishing robot state to rviz");
581: }
582: }
583:
584: void RVizInterface::spin() {
585:     ros::Rate r(update_rate_);
586:     while (ros::ok()) {
587:         // Collect new messages on subscriber topics
588:         ros::spinOnce();
589:
590:         // Enforce update rate
591:         r.sleep();
592:     }
593: }
```

```
1: #include "quad_utils/trajectory_publisher.h"
2:
3: TrajectoryPublisher::TrajectoryPublisher(ros::NodeHandle nh) {
4:     nh_ = nh;
5:
6:     // Load rosparms from parameter server
7:     std::string body_plan_topic, trajectory_state_topic;
8:
9:     nh.param<std::string>("topics/global_plan", body_plan_topic, "/body_plan");
10:    nh.param<std::string>("topics/state/trajectory", trajectory_state_topic,
11:        "/state/trajectory");
12:
13:    nh.param<std::string>("map_frame", map_frame_, "map");
14:    nh.param<std::string>("trajectory_publisher/traj_source", traj_source_,
15:        "topic");
16:    nh.param<double>("trajectory_publisher/update_rate", update_rate_, 30);
17:
18:    // Setup subs and pubs
19:    body_plan_sub_ = nh_.subscribe(body_plan_topic, 1,
20:        &TrajectoryPublisher::robotPlanCallback, this);
21:
22:    trajectory_state_pub_ =
23:        nh_.advertise<quad_msgs::RobotState>(trajectory_state_topic, 1);
24:
25:    // Initialize kinematics object
26:    quadKD_ = std::make_shared<quad_utils::QuadKD>();
27: }
28:
29: void TrajectoryPublisher::importTrajectory() {
30:     // Load the desired values into body_plan_msg_ here
31:     return;
32: }
33:
34: void TrajectoryPublisher::robotPlanCallback(
35:     const quad_msgs::RobotPlan::ConstPtr& msg) {
36:     // Save the most recent body plan
37:     body_plan_msg_ = (*msg);
38: }
39:
40: void TrajectoryPublisher::publishTrajectoryState() {
41:     // Wait until we actually have data
42:     if (body_plan_msg_.states.empty()) {
43:         return;
44:     }
45:
46:     // Get the current time in the trajectory since the beginning of the plan
47:     double traj_duration = (body_plan_msg_.states.back().header.stamp -
48:        body_plan_msg_.states.front().header.stamp)
49:        .toSec();
50:     double t =
51:        (ros::Time::now() - body_plan_msg_.states.front().header.stamp).toSec();
52:
53:     // Ensure the trajectory remains valid
54:     t = std::min(t, traj_duration);
55:
56:     // Interpolate to get the correct state and publish it
57:     quad_msgs::RobotState interp_state;
58:     int interp_primitive_id;
59:     quad_msgs::GRFArray interp_grf;
60:
61:     quad_utils::interpRobotPlan(body_plan_msg_, t, interp_state,
62:        interp_primitive_id, interp_grf);
63:
64:     // Fill joints and feet with dummy data
65:     if (interp_state.joints.name.empty()) {
66:         interp_state.joints.name = {"8", "0", "1", "9", "2", "3",
67:            "10", "4", "5", "11", "6", "7"};
68:         interp_state.joints.position = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
69:         interp_state.joints.velocity = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
70:         interp_state.joints.effort = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
71:     }
72:     quad_utils::fkRobotState(*quadKD_, interp_state.body, interp_state.joints,
73:        interp_state.feet);
74:
75:     trajectory_state_pub_.publish(interp_state);
76: }
77:
```

```
78: void TrajectoryPublisher::spin() {
79:     ros::Rate r(update_rate_);
80:     if (traj_source_.compare("import") == 0) {
81:         importTrajectory();
82:     }
83:     while (ros::ok()) {
84:         // Publish the trajectory state
85:         publishTrajectoryState();
86:
87:         // Collect new messages on subscriber topics
88:         ros::spinOnce();
89:
90:         // Enforce update rate
91:         r.sleep();
92:     }
93: }
```

```
1: #include "quad_utils/fast_terrain_map.h"
2:
3: #include <grid_map_core/grid_map_core.hpp>
4: #include <iostream>
5:
6: FastTerrainMap::FastTerrainMap() {}
7:
8: void FastTerrainMap::loadData(int x_size, int y_size,
9:                               std::vector<double> x_data,
10:                              std::vector<double> y_data,
11:                              std::vector<std::vector<double>> z_data,
12:                              std::vector<std::vector<double>> nx_data,
13:                              std::vector<std::vector<double>> ny_data,
14:                              std::vector<std::vector<double>> nz_data,
15:                              std::vector<std::vector<double>> z_data_filt,
16:                              std::vector<std::vector<double>> nx_data_filt,
17:                              std::vector<std::vector<double>> ny_data_filt,
18:                              std::vector<std::vector<double>> nz_data_filt) {
19:     // Load the data into the object's private variables
20:     x_size_ = x_size;
21:     y_size_ = y_size;
22:     x_data_ = x_data;
23:     y_data_ = y_data;
24:
25:     x_diff_ = x_data[1] - x_data[0];
26:     y_diff_ = y_data[1] - y_data[0];
27:
28:     z_data_ = z_data;
29:     nx_data_ = nx_data;
30:     ny_data_ = ny_data;
31:     nz_data_ = nz_data;
32:
33:     z_data_filt_ = z_data_filt;
34:     nx_data_filt_ = nx_data_filt;
35:     ny_data_filt_ = ny_data_filt;
36:     nz_data_filt_ = nz_data_filt;
37: }
38:
39: void FastTerrainMap::loadFlat() {
40:     int x_size = 2;
41:     int y_size = 2;
42:     std::vector<double> x_data = {-5, 5};
43:     std::vector<double> y_data = {-5, 5};
44:     std::vector<double> z_data_vec = {0, 0};
45:     std::vector<double> nz_data_vec = {1, 1};
46:     std::vector<std::vector<double>> z_data = {z_data_vec, z_data_vec};
47:     std::vector<std::vector<double>> nx_data = z_data;
48:     std::vector<std::vector<double>> ny_data = z_data;
49:     std::vector<std::vector<double>> nz_data = {nz_data_vec, nz_data_vec};
50:     std::vector<std::vector<double>> z_data_filt = z_data;
51:     std::vector<std::vector<double>> nx_data_filt = nx_data;
52:     std::vector<std::vector<double>> ny_data_filt = ny_data;
53:     std::vector<std::vector<double>> nz_data_filt = nz_data;
54:
55:     this->loadData(x_size, y_size, x_data, y_data, z_data, nx_data, ny_data,
56:                   nz_data, z_data_filt, nx_data_filt, ny_data_filt,
57:                   nz_data_filt);
58: }
59:
60: void FastTerrainMap::loadFlatElevated(double height) {
61:     int x_size = 2;
62:     int y_size = 2;
63:     std::vector<double> x_data = {-5, 5};
64:     std::vector<double> y_data = {-5, 5};
65:     std::vector<double> z_data_vec = {height, height};
66:     std::vector<double> nx_data_vec = {0, 0};
67:     std::vector<double> nz_data_vec = {1, 1};
68:     std::vector<std::vector<double>> z_data = {z_data_vec, z_data_vec};
69:     std::vector<std::vector<double>> nx_data = {nx_data_vec, nx_data_vec};
70:     std::vector<std::vector<double>> ny_data = nx_data;
71:     std::vector<std::vector<double>> nz_data = {nz_data_vec, nz_data_vec};
72:     std::vector<std::vector<double>> z_data_filt = z_data;
73:     std::vector<std::vector<double>> nx_data_filt = nx_data;
74:     std::vector<std::vector<double>> ny_data_filt = ny_data;
75:     std::vector<std::vector<double>> nz_data_filt = nz_data;
76:
77:     this->loadData(x_size, y_size, x_data, y_data, z_data, nx_data, ny_data,
```

```
78:         nz_data, z_data_filt, nx_data_filt, ny_data_filt,
79:         nz_data_filt);
80: }
81:
82: void FastTerrainMap::loadSlope(double grade) {
83:     double slope = atan(grade);
84:     int x_size = 2;
85:     int y_size = 2;
86:     double length = 5;
87:     std::vector<double> x_data = {-length, length};
88:     std::vector<double> y_data = {-length, length};
89:     std::vector<double> z_data_vec_1 = {-length * grade, -length * grade};
90:     std::vector<double> z_data_vec_2 = {length * grade, length * grade};
91:     std::vector<double> nx_data_vec = {-sin(slope), -sin(slope)};
92:     std::vector<double> ny_data_vec = {0, 0};
93:     std::vector<double> nz_data_vec = {cos(slope), cos(slope)};
94:     std::vector<std::vector<double>> z_data = {z_data_vec_1, z_data_vec_2};
95:     std::vector<std::vector<double>> nx_data = {nx_data_vec, nx_data_vec};
96:     std::vector<std::vector<double>> ny_data = {ny_data_vec, ny_data_vec};
97:     std::vector<std::vector<double>> nz_data = {nz_data_vec, nz_data_vec};
98:     std::vector<std::vector<double>> z_data_filt = {z_data_vec_1, z_data_vec_2};
99:     std::vector<std::vector<double>> nx_data_filt = {nx_data_vec, nx_data_vec};
100:    std::vector<std::vector<double>> ny_data_filt = {ny_data_vec, ny_data_vec};
101:    std::vector<std::vector<double>> nz_data_filt = {nz_data_vec, nz_data_vec};
102:
103:    this->loadData(x_size, y_size, x_data, y_data, z_data, nx_data, ny_data,
104:                  nz_data, z_data_filt, nx_data_filt, ny_data_filt,
105:                  nz_data_filt);
106: }
107:
108: void FastTerrainMap::loadStep(double height) {
109:     double res = 0.05;
110:     double length = 2;
111:     int x_size = length * 2 / res + 1;
112:     int y_size = x_size;
113:
114:     std::vector<double> x_data;
115:     std::vector<double> y_data;
116:     std::vector<std::vector<double>> z_data(x_size);
117:     std::vector<std::vector<double>> nx_data(x_size);
118:     std::vector<std::vector<double>> ny_data(x_size);
119:     std::vector<std::vector<double>> nz_data(x_size);
120:     std::vector<std::vector<double>> z_data_filt(x_size);
121:     std::vector<std::vector<double>> nx_data_filt(x_size);
122:     std::vector<std::vector<double>> ny_data_filt(x_size);
123:     std::vector<std::vector<double>> nz_data_filt(x_size);
124:
125:     for (int i = 0; i < x_size; i++) {
126:         double x = i * res - length;
127:         x_data.push_back(i * res - length);
128:         y_data.push_back(i * res - length);
129:
130:         z_data[i].resize(y_size);
131:         nx_data[i].resize(y_size);
132:         ny_data[i].resize(y_size);
133:         nz_data[i].resize(y_size);
134:         z_data_filt[i].resize(y_size);
135:         nx_data_filt[i].resize(y_size);
136:         ny_data_filt[i].resize(y_size);
137:         nz_data_filt[i].resize(y_size);
138:
139:         for (int j = 0; j < y_size; j++) {
140:             double y = j * res - length;
141:             z_data[i][j] = (x > 0) ? height : 0;
142:             nx_data[i][j] = 0;
143:             ny_data[i][j] = 0;
144:             nz_data[i][j] = 1;
145:         }
146:     }
147:
148:     z_data_filt = z_data;
149:     nx_data_filt = nx_data;
150:     ny_data_filt = ny_data;
151:     nz_data_filt = nz_data;
152:
153:    this->loadData(x_size, y_size, x_data, y_data, z_data, nx_data, ny_data,
154:                  nz_data, z_data_filt, nx_data_filt, ny_data_filt,
```



```
155:         nz_data_filt);
156: }
157:
158: void FastTerrainMap::loadDataFromGridMap(const grid_map::GridMap map) {
159:     // Initialize the data structures for the map
160:     int x_size = map.getSize()(0);
161:     int y_size = map.getSize()(1);
162:     std::vector<double> x_data(x_size);
163:     std::vector<double> y_data(y_size);
164:     std::vector<std::vector<double>> z_data(x_size);
165:     std::vector<std::vector<double>> nx_data(x_size);
166:     std::vector<std::vector<double>> ny_data(x_size);
167:     std::vector<std::vector<double>> nz_data(x_size);
168:     std::vector<std::vector<double>> z_data_filt(x_size);
169:     std::vector<std::vector<double>> nx_data_filt(x_size);
170:     std::vector<std::vector<double>> ny_data_filt(x_size);
171:     std::vector<std::vector<double>> nz_data_filt(x_size);
172:
173:     // Load the x and y data coordinates
174:     for (int i = 0; i < x_size; i++) {
175:         grid_map::Index index = {(x_size - 1) - i, 0};
176:         grid_map::Position position;
177:         map.getPosition(index, position);
178:         x_data[i] = position.x();
179:     }
180:     for (int i = 0; i < y_size; i++) {
181:         grid_map::Index index = {0, (y_size - 1) - i};
182:         grid_map::Position position;
183:         map.getPosition(index, position);
184:         y_data[i] = position.y();
185:     }
186:
187:     // Loop through the map and get the height and slope info
188:     for (int i = 0; i < x_size; i++) {
189:         for (int j = 0; j < y_size; j++) {
190:             grid_map::Index index = {(x_size - 1) - i, (y_size - 1) - j};
191:             double height = (double)map.at("z_inpainted", index);
192:             z_data[i].push_back(height);
193:
194:             if (map.exists("normal_vectors_x") == true) {
195:                 double nx = (double)map.at("normal_vectors_x", index);
196:                 double ny = (double)map.at("normal_vectors_y", index);
197:                 double nz = (double)map.at("normal_vectors_z", index);
198:                 nx_data[i].push_back(nx);
199:                 ny_data[i].push_back(ny);
200:                 nz_data[i].push_back(nz);
201:             } else {
202:                 nx_data[i].push_back(0.0);
203:                 ny_data[i].push_back(0.0);
204:                 nz_data[i].push_back(1.0);
205:             }
206:
207:             if (map.exists("z_smooth") == true) {
208:                 double z_filt = (double)map.at("z_smooth", index);
209:                 double nx_filt = (double)map.at("smooth_normal_vectors_x", index);
210:                 double ny_filt = (double)map.at("smooth_normal_vectors_y", index);
211:                 double nz_filt = (double)map.at("smooth_normal_vectors_z", index);
212:                 z_data_filt[i].push_back(z_filt);
213:                 nx_data_filt[i].push_back(nx_filt);
214:                 ny_data_filt[i].push_back(ny_filt);
215:                 nz_data_filt[i].push_back(nz_filt);
216:             } else {
217:                 z_data_filt[i].push_back(height);
218:                 nx_data_filt[i].push_back(0.0);
219:                 ny_data_filt[i].push_back(0.0);
220:                 nz_data_filt[i].push_back(1.0);
221:             }
222:         }
223:     }
224:
225:     // Update the private terrain member
226:     x_size_ = x_size;
227:     y_size_ = y_size;
228:     x_data_ = x_data;
229:     y_data_ = y_data;
230:
231:     x_diff_ = x_data_[1] - x_data_[0];
```

```
232:   y_diff_ = y_data_[1] - y_data_[0];
233:
234:   z_data_ = z_data;
235:   nx_data_ = nx_data;
236:   ny_data_ = ny_data;
237:   nz_data_ = nz_data;
238:
239:   z_data_filt_ = z_data_filt;
240:   nx_data_filt_ = nx_data_filt;
241:   ny_data_filt_ = ny_data_filt;
242:   nz_data_filt_ = nz_data_filt;
243: }
244:
245: bool FastTerrainMap::isInRange(const double x, const double y) const {
246:     double epsilon = 0.5;
247:     if (((x - epsilon) >= x_data_.front()) && ((x + epsilon) <= x_data_.back()) &&
248:         ((y - epsilon) >= y_data_.front()) && ((y + epsilon) <= y_data_.back())) {
249:         return true;
250:     } else {
251:         return false;
252:     }
253: }
254:
255: double FastTerrainMap::getGroundHeight(const double x, const double y) const {
256:     // quad_utils::FunctionTimer timer(__FUNCTION__);
257:     int ix = getXIndex(x);
258:     int iy = getYIndex(y);
259:
260:     double x1 = x_data_[ix];
261:     double x2 = x_data_[ix + 1];
262:     double y1 = y_data_[iy];
263:     double y2 = y_data_[iy + 1];
264:
265:     // Perform bilinear interpolation
266:     double fx1y1 = z_data_[ix][iy];
267:     double fx1y2 = z_data_[ix][iy + 1];
268:     double fx2y1 = z_data_[ix + 1][iy];
269:     double fx2y2 = z_data_[ix + 1][iy + 1];
270:     double height = 1.0 / ((x2 - x1) * (y2 - y1)) *
271:         (fx1y1 * (x2 - x) * (y2 - y) + fx2y1 * (x - x1) * (y2 - y) +
272:          fx1y2 * (x2 - x) * (y - y1) + fx2y2 * (x - x1) * (y - y1));
273:
274:     // timer.reportStatistics();
275:     return height;
276: }
277:
278: double FastTerrainMap::getGroundHeightFiltered(const double x,
279:                                                 const double y) const {
280:     // quad_utils::FunctionTimer timer(__FUNCTION__);
281:
282:     int ix = getXIndex(x);
283:     int iy = getYIndex(y);
284:     double x1 = x_data_[ix];
285:     double x2 = x_data_[ix + 1];
286:     double y1 = y_data_[iy];
287:     double y2 = y_data_[iy + 1];
288:
289:     // Perform bilinear interpolation
290:     double fx1y1 = z_data_filt_[ix][iy];
291:     double fx1y2 = z_data_filt_[ix][iy + 1];
292:     double fx2y1 = z_data_filt_[ix + 1][iy];
293:     double fx2y2 = z_data_filt_[ix + 1][iy + 1];
294:     double height = 1.0 / ((x2 - x1) * (y2 - y1)) *
295:         (fx1y1 * (x2 - x) * (y2 - y) + fx2y1 * (x - x1) * (y2 - y) +
296:          fx1y2 * (x2 - x) * (y - y1) + fx2y2 * (x - x1) * (y - y1));
297:
298:     // timer.reportStatistics();
299:     return height;
300: }
301:
302: std::array<double, 3> FastTerrainMap::getSurfaceNormal(const double x,
303:                                                         const double y) const {
304:     std::array<double, 3> surf_norm;
305:
306:     int ix = getXIndex(x);
307:     int iy = getYIndex(y);
308:     double x1 = x_data_[ix];
```

```
309: double x2 = x_data_[ix + 1];
310: double y1 = y_data_[iy];
311: double y2 = y_data_[iy + 1];
312:
313: double fx_x1y1 = nx_data_[ix][iy];
314: double fx_x1y2 = nx_data_[ix][iy + 1];
315: double fx_x2y1 = nx_data_[ix + 1][iy];
316: double fx_x2y2 = nx_data_[ix + 1][iy + 1];
317:
318: surf_norm[0] =
319:     1.0 / ((x2 - x1) * (y2 - y1)) *
320:     (fx_x1y1 * (x2 - x) * (y2 - y) + fx_x2y1 * (x - x1) * (y2 - y) +
321:     fx_x1y2 * (x2 - x) * (y - y1) + fx_x2y2 * (x - x1) * (y - y1));
322:
323: double fy_x1y1 = ny_data_[ix][iy];
324: double fy_x1y2 = ny_data_[ix][iy + 1];
325: double fy_x2y1 = ny_data_[ix + 1][iy];
326: double fy_x2y2 = ny_data_[ix + 1][iy + 1];
327:
328: surf_norm[1] =
329:     1.0 / ((x2 - x1) * (y2 - y1)) *
330:     (fy_x1y1 * (x2 - x) * (y2 - y) + fy_x2y1 * (x - x1) * (y2 - y) +
331:     fy_x1y2 * (x2 - x) * (y - y1) + fy_x2y2 * (x - x1) * (y - y1));
332:
333: double fz_x1y1 = nz_data_[ix][iy];
334: double fz_x1y2 = nz_data_[ix][iy + 1];
335: double fz_x2y1 = nz_data_[ix + 1][iy];
336: double fz_x2y2 = nz_data_[ix + 1][iy + 1];
337:
338: surf_norm[2] =
339:     1.0 / ((x2 - x1) * (y2 - y1)) *
340:     (fz_x1y1 * (x2 - x) * (y2 - y) + fz_x2y1 * (x - x1) * (y2 - y) +
341:     fz_x1y2 * (x2 - x) * (y - y1) + fz_x2y2 * (x - x1) * (y - y1));
342: return surf_norm;
343: }
344:
345: std::array<double, 3> FastTerrainMap::getSurfaceNormalFiltered(
346:     const double x, const double y) const {
347:     std::array<double, 3> surf_norm;
348:
349:     int ix = getXIndex(x);
350:     int iy = getYIndex(y);
351:     double x1 = x_data_[ix];
352:     double x2 = x_data_[ix + 1];
353:     double y1 = y_data_[iy];
354:     double y2 = y_data_[iy + 1];
355:
356:     double fx_x1y1 = nx_data_filt_[ix][iy];
357:     double fx_x1y2 = nx_data_filt_[ix][iy + 1];
358:     double fx_x2y1 = nx_data_filt_[ix + 1][iy];
359:     double fx_x2y2 = nx_data_filt_[ix + 1][iy + 1];
360:
361:     surf_norm[0] =
362:         1.0 / ((x2 - x1) * (y2 - y1)) *
363:         (fx_x1y1 * (x2 - x) * (y2 - y) + fx_x2y1 * (x - x1) * (y2 - y) +
364:         fx_x1y2 * (x2 - x) * (y - y1) + fx_x2y2 * (x - x1) * (y - y1));
365:
366:     double fy_x1y1 = ny_data_filt_[ix][iy];
367:     double fy_x1y2 = ny_data_filt_[ix][iy + 1];
368:     double fy_x2y1 = ny_data_filt_[ix + 1][iy];
369:     double fy_x2y2 = ny_data_filt_[ix + 1][iy + 1];
370:
371:     surf_norm[1] =
372:         1.0 / ((x2 - x1) * (y2 - y1)) *
373:         (fy_x1y1 * (x2 - x) * (y2 - y) + fy_x2y1 * (x - x1) * (y2 - y) +
374:         fy_x1y2 * (x2 - x) * (y - y1) + fy_x2y2 * (x - x1) * (y - y1));
375:
376:     double fz_x1y1 = nz_data_filt_[ix][iy];
377:     double fz_x1y2 = nz_data_filt_[ix][iy + 1];
378:     double fz_x2y1 = nz_data_filt_[ix + 1][iy];
379:     double fz_x2y2 = nz_data_filt_[ix + 1][iy + 1];
380:
381:     surf_norm[2] =
382:         1.0 / ((x2 - x1) * (y2 - y1)) *
383:         (fz_x1y1 * (x2 - x) * (y2 - y) + fz_x2y1 * (x - x1) * (y2 - y) +
384:         fz_x1y2 * (x2 - x) * (y - y1) + fz_x2y2 * (x - x1) * (y - y1));
385:     return surf_norm;
```

```

386: }
387:
388: Eigen::Vector3d FastTerrainMap::getSurfaceNormalFilteredEigen(
389:     const double x, const double y) const {
390:     Eigen::Vector3d surf_norm;
391:
392:     int ix = getXIndex(x);
393:     int iy = getYIndex(y);
394:     double x1 = x_data_[ix];
395:     double x2 = x_data_[ix + 1];
396:     double y1 = y_data_[iy];
397:     double y2 = y_data_[iy + 1];
398:
399:     double fx_x1y1 = nx_data_filt_[ix][iy];
400:     double fx_x1y2 = nx_data_filt_[ix][iy + 1];
401:     double fx_x2y1 = nx_data_filt_[ix + 1][iy];
402:     double fx_x2y2 = nx_data_filt_[ix + 1][iy + 1];
403:
404:     surf_norm[0] =
405:         1.0 / ((x2 - x1) * (y2 - y1)) *
406:         (fx_x1y1 * (x2 - x) * (y2 - y) + fx_x2y1 * (x - x1) * (y2 - y) +
407:          fx_x1y2 * (x2 - x) * (y - y1) + fx_x2y2 * (x - x1) * (y - y1));
408:
409:     double fy_x1y1 = ny_data_filt_[ix][iy];
410:     double fy_x1y2 = ny_data_filt_[ix][iy + 1];
411:     double fy_x2y1 = ny_data_filt_[ix + 1][iy];
412:     double fy_x2y2 = ny_data_filt_[ix + 1][iy + 1];
413:
414:     surf_norm[1] =
415:         1.0 / ((x2 - x1) * (y2 - y1)) *
416:         (fy_x1y1 * (x2 - x) * (y2 - y) + fy_x2y1 * (x - x1) * (y2 - y) +
417:          fy_x1y2 * (x2 - x) * (y - y1) + fy_x2y2 * (x - x1) * (y - y1));
418:
419:     double fz_x1y1 = nz_data_filt_[ix][iy];
420:     double fz_x1y2 = nz_data_filt_[ix][iy + 1];
421:     double fz_x2y1 = nz_data_filt_[ix + 1][iy];
422:     double fz_x2y2 = nz_data_filt_[ix + 1][iy + 1];
423:
424:     surf_norm[2] =
425:         1.0 / ((x2 - x1) * (y2 - y1)) *
426:         (fz_x1y1 * (x2 - x) * (y2 - y) + fz_x2y1 * (x - x1) * (y2 - y) +
427:          fz_x1y2 * (x2 - x) * (y - y1) + fz_x2y2 * (x - x1) * (y - y1));
428:     // std::cout << "surf_norm:\n" << surf_norm << std::endl;
429:     return surf_norm;
430: }
431:
432: Eigen::Vector3d FastTerrainMap::projectToMap(const Eigen::Vector3d point,
433:                                              const Eigen::Vector3d direction) {
434:     // quad_utils::FunctionTimer timer(__FUNCTION__);
435:
436:     Eigen::Vector3d direction_norm = direction;
437:     direction_norm.normalize();
438:     Eigen::Vector3d result = point;
439:     Eigen::Vector3d new_point = point;
440:     Eigen::Vector3d old_point = point;
441:     double step_size = 0.01;
442:     double clearance = 0;
443:     while (clearance >= 0) {
444:         old_point = new_point;
445:         for (int i = 0; i < 3; i++) {
446:             new_point[i] += direction_norm[i] * step_size;
447:         }
448:         if (isInRange(new_point[0], new_point[1])) {
449:             clearance = new_point[2] - getGroundHeight(new_point[0], new_point[1]);
450:         } else {
451:             result = {old_point[0], old_point[1],
452:                     -std::numeric_limits<double>::max()};
453:             ROS_WARN_THROTTLE(0.5, "Tried to project to a point off the map.");
454:             return result;
455:         }
456:     }
457:
458:     result = {old_point[0], old_point[1],
459:             getGroundHeight(old_point[0], old_point[1])};
460:
461:     // timer.reportStatistics();
462:     return result;

```

```
463: }
464:
465: std::vector<double> FastTerrainMap::getXData() const { return x_data_; }
466:
467: std::vector<double> FastTerrainMap::getYData() const { return y_data_; }
468:
469: bool FastTerrainMap::isEmpty() const {
470:     if (x_size_ == 0 || y_size_ == 0) {
471:         return true;
472:     } else {
473:         return false;
474:     }
475: }
```

```
1: #include <ros/ros.h>
2:
3: #include "quad_utils/rviz_interface.h"
4:
5: int main(int argc, char** argv) {
6:     ros::init(argc, argv, "rviz_interface_node");
7:     ros::NodeHandle nh;
8:
9:     RVizInterface rviz_interface(nh);
10:    rviz_interface.spin();
11:
12:    return 0;
13: }
```

```
1: #include <ros/ros.h>
2:
3: #include "quad_utils/trajectory_publisher.h"
4:
5: int main(int argc, char** argv) {
6:     ros::init(argc, argv, "trajectory_publisher_node");
7:     ros::NodeHandle nh;
8:
9:     TrajectoryPublisher trajectory_publisher(nh);
10:    trajectory_publisher.spin();
11:
12:    return 0;
13: }
```