

```

1: #include <gtest/gtest.h>
2: #include <ros/ros.h>
3:
4: #include "quad_utils/fast_terrain_map.h"
5:
6: TEST(FastTerrainMapTest, testSpeedComparison) {
7:     // Define map parameters
8:     double res = 0.01;
9:     double x_length = 12.0;
10:    double y_length = 5.0;
11:    double x_center = 4.0;
12:    double y_center = 0.0;
13:    double x_origin = x_center - 0.5 * x_length;
14:    double y_origin = y_center - 0.5 * y_length;
15:
16:    // Create GridMap
17:    grid_map::GridMap grid_map_obj({"z", "z_inpainted", "nx", "ny", "nz",
18:                                   "z_filt", "nx_filt", "ny_filt", "nz_filt"});
19:    // grid_map_obj.setBasicLayers({"z", "nx", "ny", "nz", "z_filt", "nx_filt", "ny_filt", "nz_filt"});
20:    grid_map_obj.setFrameId("map");
21:    grid_map_obj.setGeometry(grid_map::Length(x_length, y_length), res,
22:                             grid_map::Position(x_center, y_center));
23:    int x_size = grid_map_obj.getSize()(0);
24:    int y_size = grid_map_obj.getSize()(1);
25:    // printf("Created map with size %f x %f m (%i x %i cells).\n",
26:    //         grid_map_obj.getLength().x(), grid_map_obj.getLength().y(), x_size,
27:    //         y_size);
28:
29:    // Load grid map with random noise
30:    for (grid_map::GridMapIterator it(grid_map_obj); !it.isPastEnd(); ++it) {
31:        grid_map_obj.at("z", *it) = 0.1 * ((double)rand() / RAND_MAX);
32:        grid_map_obj.at("z_inpainted", *it) = grid_map_obj.at("z", *it);
33:        grid_map_obj.at("z_filt", *it) = grid_map_obj.at("z", *it);
34:
35:        grid_map_obj.at("nx", *it) = 0.0;
36:        grid_map_obj.at("ny", *it) = 0.0;
37:        grid_map_obj.at("nz", *it) = 1.0;
38:
39:        grid_map_obj.at("nx_filt", *it) = 0.0;
40:        grid_map_obj.at("ny_filt", *it) = 0.0;
41:        grid_map_obj.at("nz_filt", *it) = 1.0;
42:    }
43:
44:    // Load data into fast terrain map obj
45:    FastTerrainMap fast_terrain_map;
46:    fast_terrain_map.loadDataFromGridMap(grid_map_obj);
47:
48:    // Initialize testing parameters
49:    const int N = 10001;
50:    grid_map::Position pos;
51:    double x, y, z;
52:    std::chrono::time_point<std::chrono::steady_clock> start_time,
53:        intermediate_time_1, intermediate_time_2, stop_time;
54:    std::chrono::duration<double> elapsed;
55:    std::vector<double> timings(N - 1);
56:
57:    // Start GridMap nearest neighbor
58:    double gm_nn_total_elapsed = 0;
59:    start_time = std::chrono::steady_clock::now();
60:    for (int i = 0; i < N; i++) {
61:        // Generate random test point
62:        x = (x_length - res) * ((double)rand() / RAND_MAX) + x_origin + 0.5 * res;
63:        y = (y_length - res) * ((double)rand() / RAND_MAX) + y_origin + 0.5 * res;
64:
65:        // Query
66:        pos = {x, y};
67:        intermediate_time_1 = std::chrono::steady_clock::now();
68:        z += grid_map_obj.atPosition(
69:            "z", pos, grid_map::InterpolationMethods::INTER_NEAREST) /
70:            N;
71:        intermediate_time_2 = std::chrono::steady_clock::now();
72:        elapsed = std::chrono::duration_cast<std::chrono::duration<double>>(
73:            intermediate_time_2 - intermediate_time_1);
74:
75:        if (i > 0) {
76:            timings[i - 1] = (double)elapsed.count();
77:        } else {

```

```
78:     start_time = std::chrono::steady_clock::now();
79: }
80: }
81:
82: stop_time = std::chrono::steady_clock::now();
83: elapsed = std::chrono::duration_cast<std::chrono::duration<double>>(
84:     stop_time - start_time);
85: double gm_nn_time = (double)elapsed.count();
86: for (int i = 0; i < timings.size(); i++) {
87:     gm_nn_total_elapsed += timings[i];
88:     if (fmod(log10(i), 1) == 0) {
89:         // printf("Duration of nn iteration %d = %.3fus\n", i, 1e6*timings[i]);
90:     }
91: }
92:
93: // Start GridMap linear
94: double gm_lin_total_elapsed = 0;
95: z = 0;
96: start_time = std::chrono::steady_clock::now();
97: for (int i = 0; i < N; i++) {
98:     // Generate random test point
99:     x = (x_length - res) * ((double)rand() / RAND_MAX) + x_origin + 0.5 * res;
100:    y = (y_length - res) * ((double)rand() / RAND_MAX) + y_origin + 0.5 * res;
101:
102:    // Query
103:    pos = {x, y};
104:    intermediate_time_1 = std::chrono::steady_clock::now();
105:    z += grid_map_obj.atPosition("z", pos,
106:                                grid_map::InterpolationMethods::INTER_LINEAR) /
107:        N;
108:    intermediate_time_2 = std::chrono::steady_clock::now();
109:    elapsed = std::chrono::duration_cast<std::chrono::duration<double>>(
110:        intermediate_time_2 - intermediate_time_1);
111:
112:    if (i > 0) {
113:        timings[i - 1] = (double)elapsed.count();
114:    } else {
115:        start_time = std::chrono::steady_clock::now();
116:    }
117: }
118:
119: stop_time = std::chrono::steady_clock::now();
120: elapsed = std::chrono::duration_cast<std::chrono::duration<double>>(
121:     stop_time - start_time);
122: double gm_lin_time = (double)elapsed.count();
123: for (int i = 0; i < timings.size(); i++) {
124:     gm_lin_total_elapsed += timings[i];
125:     if (fmod(log10(i), 1) == 0) {
126:         // printf("Duration of lin iteration %d = %.3fus\n", i, 1e6*timings[i]);
127:     }
128: }
129:
130: // Start FastTerrainMap linear
131: double ftm_lin_total_elapsed = 0;
132: z = 0;
133: start_time = std::chrono::steady_clock::now();
134: for (int i = 0; i < N; i++) {
135:     // Generate random test point
136:     x = (x_length - res) * ((double)rand() / RAND_MAX) + x_origin + 0.5 * res;
137:     y = (y_length - res) * ((double)rand() / RAND_MAX) + y_origin + 0.5 * res;
138:
139:     // Query
140:     pos = {x, y};
141:     intermediate_time_1 = std::chrono::steady_clock::now();
142:     z += fast_terrain_map.getGroundHeight(x, y) / N;
143:     intermediate_time_2 = std::chrono::steady_clock::now();
144:     elapsed = std::chrono::duration_cast<std::chrono::duration<double>>(
145:         intermediate_time_2 - intermediate_time_1);
146:
147:     if (i > 0) {
148:         timings[i - 1] = (double)elapsed.count();
149:     } else {
150:         start_time = std::chrono::steady_clock::now();
151:     }
152: }
153:
154: stop_time = std::chrono::steady_clock::now();
```

```

155: elapsed = std::chrono::duration_cast<std::chrono::duration<double>>(
156:     stop_time - start_time);
157: double ftm_lin_time = (double)elapsed.count();
158: for (int i = 0; i < timings.size(); i++) {
159:     ftm_lin_total_elapsed += timings[i];
160:     if (fmod(log10(i), 1) == 0) {
161:         // printf("Duration of lin iteration %d = %.3fus\n", i, 1e6*timings[i]);
162:     }
163: }
164:
165: printf("GridMap NN avg (total duration/iterations) = %.3fus\n",
166:     (double)1e6 * gm_nn_time / N);
167: // printf("GridMap NN avg (sum of individual/iterations) = %.3fus\n",
168: //     (double)1e6*gm_nn_total_elapsed/timings.size());
169: printf("GridMap linear avg (total duration/iterations) = %.3fus\n",
170:     (double)1e6 * gm_lin_time / N);
171: // printf("GridMap linear avg (sum of individual/iterations) = %.3fus\n",
172: //     (double)1e6*gm_lin_total_elapsed/timings.size());
173: printf("FastTerrainMap linear avg (total duration/iterations) = %.3fus\n",
174:     (double)1e6 * ftm_lin_time / N);
175: // printf("FastTerrainMap linear avg (sum of individual/iterations) =
176: //     %.3fus\n", (double)1e6*ftm_lin_total_elapsed/timings.size()); printf("z avg
177: // = %.3f\n", (double)z);
178:
179: EXPECT_EQ(1 + 1, 2);
180: }
181:
182: TEST(FastTerrainMapTest, testConstructor) {
183:     FastTerrainMap fast_terrain_map;
184:     EXPECT_EQ(1 + 1, 2);
185: }
186:
187: TEST(FastTerrainMapTest, testProjection) {
188:     FastTerrainMap fast_terrain_map;
189:
190:     int x_size = 2;
191:     int y_size = 2;
192:     std::vector<double> x_data = {-1, 1};
193:     std::vector<double> y_data = {-1, 1};
194:     std::vector<double> z_data_vec = {-1, 1};
195:     std::vector<std::vector<double>> z_data = {z_data_vec, z_data_vec};
196:
197:     std::vector<double> dx_data_vec = {0, 0};
198:     std::vector<double> dz_data_vec = {1, 1};
199:     std::vector<std::vector<double>> dx_data = {dx_data_vec, dx_data_vec};
200:     std::vector<std::vector<double>> dy_data = dx_data;
201:     std::vector<std::vector<double>> dz_data = {dz_data_vec, dz_data_vec};
202:
203:     fast_terrain_map.loadData(x_size, y_size, x_data, y_data, z_data, dx_data,
204:         dy_data, dz_data, z_data, dx_data, dy_data,
205:         dz_data);
206:
207:     // Eigen::Vector3d point = {0, 0.5, 1};
208:     // Eigen::Vector3d direction = {0.1, 0.1, -1};
209:     Eigen::Vector3d point = {0, 0, 1};
210:     Eigen::Vector3d direction = {0, 0, -1};
211:
212:     auto t_start = std::chrono::steady_clock::now();
213:     Eigen::Vector3d intersection =
214:         fast_terrain_map.projectToMap(point, direction);
215:     auto t_end = std::chrono::steady_clock::now();
216:     std::chrono::duration<double> time_span =
217:         std::chrono::duration_cast<std::chrono::duration<double>>(t_end -
218:             t_start);
219:     // std::cout << "projectToMap took " << time_span.count() << " seconds." <<
220:     // std::endl;
221:
222:     // std::cout << "Result is {" << intersection[0] << ", " << intersection[1] <<
223:     // ", " << intersection[2] << "}" << std::endl;
224:
225:     EXPECT_EQ(1 + 1, 2);
226: }
227:
228: TEST(FastTerrainMapTest, testSlope) {
229:     FastTerrainMap map;
230:     double grade = 0.5;
231:     map.loadSlope(grade);

```

```
232:
233:   Eigen::Vector3d normal;
234:   normal << -sin(atan(grade)), 0, cos(atan(grade));
235:
236:   double x = -2;
237:   double y = 0;
238:   double z = grade * x;
239:   EXPECT_TRUE(abs(map.getGroundHeight(x, y) - z) < 1e-6);
240:
241:   x = 2;
242:   y = 0;
243:   z = grade * x;
244:   EXPECT_TRUE(abs(map.getGroundHeight(x, y) - z) < 1e-6);
245:
246:   x = 0;
247:   y = 2;
248:   z = grade * x;
249:   EXPECT_TRUE(abs(map.getGroundHeight(x, y) - z) < 1e-6);
250:
251:   EXPECT_TRUE(map.getSurfaceNormalFilteredEigen(x, y).isApprox(normal));
252: }
253:
254: TEST(FastTerrainMapTest, testStep) {
255:   FastTerrainMap map;
256:   double height = 0.2;
257:   map.loadStep(height);
258:
259:   Eigen::Vector3d normal;
260:   normal << 0, 0, 1;
261:
262:   double x = -2;
263:   double y = 0;
264:   double z = (x > 0) ? height : 0;
265:   EXPECT_TRUE(abs(map.getGroundHeight(x, y) - z) < 1e-6);
266:
267:   x = 2;
268:   y = 0;
269:   z = (x > 0) ? height : 0;
270:   EXPECT_TRUE(abs(map.getGroundHeight(x, y) - z) < 1e-6);
271:
272:   x = 0;
273:   y = 2;
274:   z = (x > 0) ? height : 0;
275:   EXPECT_TRUE(abs(map.getGroundHeight(x, y) - z) < 1e-6);
276:
277:   EXPECT_TRUE(map.getSurfaceNormalFilteredEigen(x, y).isApprox(normal));
278: }
```

```
1: #include <gtest/gtest.h>
2: #include <ros/ros.h>
3:
4: #include "quad_utils/terrain_map_publisher.h"
5:
6: TEST(TerrainMapPublisherTest, testTrue) {
7:     ros::NodeHandle nh;
8:     TerrainMapPublisher terrain_map_publisher(nh);
9:     EXPECT_EQ(1 + 1, 2);
10: }
```

```
1: #include <gtest/gtest.h>
2: #include <ros/ros.h>
3:
4: #include "quad_utils/ros_utils.h"
5: Eigen::IOFormat CleanFmt(4, 0, " ", " ", "\n", "[", "]");
6:
7: TEST(EigenTest, testMap) {
8:     const int N = 9;
9:     double data_c[N];
10:    for (int i = 0; i < N; ++i) {
11:        data_c[i] = (double)i;
12:    }
13:
14:    // std::cout << Eigen::Map<Eigen::VectorXi>(array) << std::endl;
15:
16:    Eigen::MatrixXd data_eigen;
17:
18:    data_eigen = Eigen::Map<Eigen::Matrix<double, 1, N>>(data_c);
19:
20:    // std::cout << data_eigen.format(CleanFmt) << std::endl;
21: }
```

```
1: #include <gtest/gtest.h>
2: #include <ros/ros.h>
3:
4: int main(int argc, char** argv) {
5:     testing::InitGoogleTest(&argc, argv);
6:     ros::init(argc, argv, "quad_utils_tester");
7:
8:     return RUN_ALL_TESTS();
9: }
```

```
1: #include <gtest/gtest.h>
2: #include <ros/ros.h>
3:
4: #include "quad_utils/math_utils.h"
5: #include "quad_utils/ros_utils.h"
6:
7: TEST(MathTest, testWrap) {
8:     int N = 201;
9:     double amplitude = 10;
10:    double period = 4 * M_PI;
11:    std::vector<double> data(N), t(N);
12:    for (int i = 0; i < data.size(); i++) {
13:        t[i] = i * period / N;
14:        data[i] = amplitude * sin(t[i]);
15:    }
16:
17:    std::vector<double> data_wrapped = math_utils::wrapToPi(data);
18:    std::vector<double> data_unwrapped = math_utils::unwrap(data_wrapped);
19:
20:    double error = 0;
21:    for (int i = 0; i < data.size(); i++) {
22:        error += abs(data[i] - data_unwrapped[i]);
23:    }
24:
25:    double tolerance = 1e-4;
26:    EXPECT_TRUE(error <= tolerance);
27: }
```



```
1: #include <gtest/gtest.h>
2: #include <ros/ros.h>
3:
4: #include "quad_utils/rviz_interface.h"
5:
6: TEST(RVizInterfaceTest, testTrue) {
7:     ros::NodeHandle nh;
8:     RVizInterface rviz_interface(nh);
9:     EXPECT_EQ(1 + 1, 2);
10: }
```

```
1: #include <gtest/gtest.h>
2: #include <ros/ros.h>
3:
4: #include <grid_map_core/grid_map_core.hpp>
5:
6: #include "quad_utils/quad_kd.h"
7: #include "quad_utils/ros_utils.h"
8:
9: using namespace quad_utils;
10:
11: const double kinematics_tol = 1e-4;
12:
13: TEST(KinematicsTest, testDifferentialFKIK) {
14:     // Declare kinematics object
15:     QuadKD kinematics;
16:
17:     for (size_t i = 0; i < 20; i++) {
18:         // Declare input and output RobotState object
19:         quad_msgs::RobotState state, state_out;
20:
21:         // Random velocities at origin
22:         Eigen::VectorXd body_state(12);
23:         body_state << (double)rand() / RAND_MAX - 0.5,
24:             (double)rand() / RAND_MAX - 0.5, (double)rand() / RAND_MAX - 0.5,
25:             1.5 * (double)rand() / RAND_MAX - 0.75,
26:             1.5 * (double)rand() / RAND_MAX - 0.75,
27:             1.5 * (double)rand() / RAND_MAX - 0.75,
28:             10 * (double)rand() / RAND_MAX - 5, 10 * (double)rand() / RAND_MAX - 5,
29:             10 * (double)rand() / RAND_MAX - 5,
30:             3.14 * (double)rand() / RAND_MAX - 1.57,
31:             3.14 * (double)rand() / RAND_MAX - 1.57,
32:             3.14 * (double)rand() / RAND_MAX - 1.57;
33:
34:         state.body = eigenToBodyStateMsg(body_state);
35:
36:         state.joints.name = {"8", "0", "1", "9", "2", "3",
37:             "10", "4", "5", "11", "6", "7"};
38:         state.joints.position.clear();
39:         state.joints.velocity.clear();
40:         state.joints.effort.clear();
41:
42:         for (int j = 0; j < 4; j++) {
43:             // Just some arbitrary joints position
44:             state.joints.position.push_back(0.1);
45:             state.joints.position.push_back(0.2);
46:             state.joints.position.push_back(0.3);
47:
48:             // Random joints velocity
49:             state.joints.velocity.push_back(3.14 * (double)rand() / RAND_MAX - 1.57);
50:             state.joints.velocity.push_back(3.14 * (double)rand() / RAND_MAX - 1.57);
51:             state.joints.velocity.push_back(3.14 * (double)rand() / RAND_MAX - 1.57);
52:
53:             // We don't need joints effort here
54:             state.joints.effort.push_back(0.0);
55:             state.joints.effort.push_back(0.0);
56:             state.joints.effort.push_back(0.0);
57:         }
58:
59:         // Run FK get foot velocities and IK them back
60:         quad_utils::fkRobotState(kinematics, state.body, state.joints, state.feet);
61:         quad_utils::ikRobotState(kinematics, state.body, state.feet,
62:             state_out.joints);
63:
64:         // Extract input joint velocities
65:         Eigen::VectorXd vel(12), vel_out(12);
66:         vectorToEigen(state.joints.velocity, vel);
67:         vectorToEigen(state_out.joints.velocity, vel_out);
68:
69:         // Check the answers
70:         Eigen::VectorXd error = vel - vel_out;
71:         EXPECT_TRUE(error.norm() <= kinematics_tol);
72:     }
73: }
74:
75: TEST(KinematicsTest, testFootForces) {
76:     // Declare kinematics object
77:     QuadKD kinematics;
```

```
78:
79:  // Length parameters from URDF
80:  // TODO(yanhaoy): load these from parameters rather than hard-coding
81:  Eigen::MatrixX<double> ls(4, 3);
82:  ls << 0.2263, 0.07, 0.0,  // abad from body
83:        0.0, 0.10098, 0.0,  // hip from abad
84:        -0.206, 0.0, 0.0,  // knee from hip
85:        0.206, 0.0, 0.0;  // toe from knee
86:
87:  double pi = 3.14159265359;
88:
89:  // Define vectors for states, forces, and torques
90:  Eigen::VectorX<double> state_positions(18), forces(12), torques(18),
91:    torques_solution(18);
92:
93:  // Compute jacobian
94:  Eigen::MatrixX<double> jacobian = Eigen::MatrixX<double>::Zero(12, 18);
95:
96:  // Set up known solution problem 1 -----
97:  state_positions = Eigen::VectorX<double>::Zero(18);
98:  for (int i = 0; i < 3; i++) {
99:    // move the CG around randomly -- it should not matter
100:    state_positions(12 + i) = (double)rand() / RAND_MAX - 0.5;
101:  }
102:  forces = Eigen::VectorX<double>::Zero(12);
103:  forces(2) = 3.0;  // front left toe Z
104:  forces(3) = 2.0;  // back left toe X
105:
106:  // Known solution
107:  torques_solution << 3.0 * ls(1, 1), 0.0, 3.0 * -ls(3, 0), 0.0, 0.0,
108:    0.0,  // leg 2
109:    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 2.0, 0.0, 3.0,  // net forces
110:    3.0 * (ls(0, 1) + ls(1, 1)), 3.0 * -ls(0, 0),
111:    -2.0 * (ls(0, 1) + ls(1, 1));
112:
113:  // Compute joint torques
114:  kinematics.getJacobianGenCoord(state_positions, jacobian);
115:  torques = jacobian.transpose() * forces;
116:
117:  // Check the answers
118:  Eigen::VectorX<double> error = torques - torques_solution;
119:  Eigen::MatrixX<double> toPrint(18, 2);
120:  toPrint << torques, torques_solution;
121:  // std::cout << "Test 1:\n" << toPrint << std::endl;
122:  EXPECT_TRUE(error.norm() <= kinematics_tol);
123:
124:  // Set up known solution problem 2 -----
125:  state_positions = Eigen::VectorX<double>::Zero(18);
126:  for (int i = 0; i < 3; i++) {
127:    // move the CG around randomly -- it should not matter
128:    state_positions(12 + i) = (double)rand() / RAND_MAX - 0.5;
129:  }
130:  state_positions(17) = pi / 2;  // yaw 90 deg left
131:  state_positions(7) = pi / 4;  // front right hip 45 deg down
132:  state_positions(8) = pi / 2;  // front right knee 90 deg down
133:  forces = Eigen::VectorX<double>::Zero(12);
134:  forces(6) = 3.0;  // front right toe X
135:  forces(8) = 5.0;  // front right toe Z
136:
137:  // Known solution
138:  torques_solution << 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,  // leg 2
139:    -5.0 * ls(1, 1) -
140:    3.0 * (-ls(2, 0) * sin(pi / 4) + ls(3, 0) * sin(pi / 4)),
141:    0.0, 5.0 * -ls(3, 0) * cos(pi / 4), 0.0, 0.0, 0.0, 3.0, 0.0,
142:    5.0,  // net forces
143:    -5.0 * (ls(0, 1) + ls(1, 1)) -
144:    3.0 * (-ls(2, 0) * sin(pi / 4) + ls(3, 0) * sin(pi / 4)),
145:    -5.0 * ls(0, 0), -3.0 * ls(0, 0);
146:
147:  // Compute joint torques
148:  kinematics.getJacobianGenCoord(state_positions, jacobian);
149:  torques = jacobian.transpose() * forces;
150:
151:  // Check the answers
152:  error = torques - torques_solution;
153:  toPrint << torques, torques_solution;
154:  // std::cout << "Test 2:\n" << toPrint << std::endl;
```

```

155: EXPECT_TRUE(error.norm() <= kinematics_tol);
156:
157: // Set up known solution problem 3 -----
158: state_positions = Eigen::VectorXd::Zero(18);
159: for (int i = 0; i < 3; i++) {
160:     // move the CG around randomly -- it should not matter
161:     state_positions(12 + i) = (double)rand() / RAND_MAX - 0.5;
162: }
163: state_positions(15) = pi / 2; // roll 90 deg right
164: state_positions(17) = pi / 2; // yaw 90 deg left
165: forces = Eigen::VectorXd::Zero(12);
166: forces(0) = 1.0; // front left toe X
167:
168: // Known solution
169: torques_solution << ls(1, 1), 0.0, -ls(3, 0), 0.0, 0.0, 0.0, // leg 2
170:     0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, // net forces
171:     ls(0, 1) + ls(1, 1), 0.0, -ls(0, 0);
172:
173: // Compute joint torques
174: kinematics.getJacobianGenCoord(state_positions, jacobian);
175: torques = jacobian.transpose() * forces;
176:
177: // Check the answers
178: error = torques - torques_solution;
179: toPrint << torques, torques_solution;
180: // std::cout << "Test 3:\n" << toPrint << std::endl;
181: EXPECT_TRUE(error.norm() <= kinematics_tol);
182:
183: // Set up known solution problem 4 -----
184: state_positions = Eigen::VectorXd::Zero(18);
185: for (int i = 0; i < 3; i++) {
186:     // move the CG around randomly -- it should not matter
187:     state_positions(12 + i) = (double)rand() / RAND_MAX - 0.5;
188: }
189: state_positions(16) = pi / 4; // pitch 45 deg down
190: state_positions(17) = pi; // yaw 180 deg
191: state_positions(1) = pi / 4; // front left hip 60 deg down
192: state_positions(4) = -pi / 4; // back left hip 45 deg up
193: state_positions(7) = -pi / 4; // front right hip 45 deg up
194: state_positions(10) = pi / 4; // back right hip 60 deg down
195: forces = Eigen::VectorXd::Zero(12);
196: forces << -1.0, 2.0, 1.0, -1.0, 2.0, 1.0, -1.0, -2.0, 1.0, -1.0, -2.0, 1.0;
197:
198: // Known solution
199: torques_solution << sqrt(2) * ls(1, 1), 0.0, -ls(3, 0), sqrt(2) * ls(1, 1),
200:     0.0, -ls(3, 0), // leg 2
201:     -sqrt(2) * ls(1, 1), 0.0, -ls(3, 0), -sqrt(2) * ls(1, 1), 0.0, -ls(3, 0),
202:     -4.0, 0.0, 4.0, // net forces
203:     0.0, 0.0, 0.0;
204:
205: // Compute joint torques
206: kinematics.getJacobianGenCoord(state_positions, jacobian);
207: torques = jacobian.transpose() * forces;
208:
209: // Check the answers
210: error = torques - torques_solution;
211: toPrint << torques, torques_solution;
212: // std::cout << "Test 4:\n" << toPrint << std::endl;
213: EXPECT_TRUE(error.norm() <= kinematics_tol);
214: }
215:
216: TEST(KinematicsTest, testFKIKFeasibleConfigurations) {
217:     ros::NodeHandle nh;
218:
219:     // Declare kinematics object
220:     QuadKD quad;
221:
222:     // Set up problem variables
223:     Eigen::Vector3d body_pos = {0, 0, 0};
224:     Eigen::Vector3d body_rpy = {0, 0, 0};
225:     Eigen::Vector3d foot_pos_world;
226:     Eigen::Vector3d joint_state_test;
227:     Eigen::Vector3d foot_pos_world_test;
228:
229:     // Compute the kinematics
230:     int N = 10000;
231:     for (int config = 0; config < N; config++) {

```

```
232:     for (int i = 0; i < 4; i++) {
233:         int leg_index = i;
234:
235:         // Generate valid joint configurations
236:         Eigen::Vector3d joint_state = {(quad.getJointUpperLimit(leg_index, 0) -
237:             quad.getJointLowerLimit(leg_index, 0)) *
238:             (double)rand() / RAND_MAX +
239:             quad.getJointLowerLimit(leg_index, 0),
240:             (quad.getJointUpperLimit(leg_index, 1) -
241:             quad.getJointLowerLimit(leg_index, 1)) *
242:             (double)rand() / RAND_MAX +
243:             quad.getJointLowerLimit(leg_index, 1),
244:             (quad.getJointUpperLimit(leg_index, 2) -
245:             quad.getJointLowerLimit(leg_index, 2)) *
246:             (double)rand() / RAND_MAX +
247:             quad.getJointLowerLimit(leg_index, 2)};
248:
249:         // Compute foot positions in this configuration
250:         quad.worldToFootFKWorldFrame(leg_index, body_pos, body_rpy, joint_state,
251:             foot_pos_world);
252:
253:         // Run IK to compute corresponding joint angles, then back through FK
254:         // This ensures that we are enforcing a hip-above-knee configuration if
255:         // otherwise ambiguous.
256:         quad.worldToFootIKWorldFrame(leg_index, body_pos, body_rpy,
257:             foot_pos_world, joint_state_test);
258:
259:         // Skip if original configuration was in an alternate configuration
260:         if (!joint_state_test.isApprox(joint_state)) continue;
261:
262:         quad.worldToFootFKWorldFrame(leg_index, body_pos, body_rpy,
263:             joint_state_test, foot_pos_world_test);
264:
265:         // Check the answers
266:         Eigen::Vector3d error = (foot_pos_world - foot_pos_world_test);
267:         EXPECT_LE(error.norm(), kinematics_tol);
268:     }
269: }
270: }
271:
272: TEST(KinematicsTest, testFKIKInfeasibleConfigurations) {
273:     ros::NodeHandle nh;
274:
275:     QuadKD quad;
276:
277:     // Set up problem variables
278:     Eigen::Vector3d body_pos = {0, 0, 0};
279:     Eigen::Vector3d body_rpy = {0, 0, 0};
280:     Eigen::Vector3d foot_pos_world;
281:     Eigen::Vector3d foot_pos_world_test;
282:     Eigen::Vector3d joint_state_test;
283:
284:     // Define arbitrary maximum foot offset for IK testing
285:     double max_offset = 2.0;
286:
287:     // Test random foot positions to make sure nothing breaks
288:     int N = 10000;
289:     for (int config = 0; config < N; config++) {
290:         // Generate random foot offset
291:         Eigen::Vector3d foot_offset = {
292:             2 * max_offset * (double)rand() / RAND_MAX - max_offset,
293:             2 * max_offset * (double)rand() / RAND_MAX - max_offset,
294:             2 * max_offset * (double)rand() / RAND_MAX - max_offset};
295:
296:         for (int i = 0; i < 4; i++) {
297:             int leg_index = i;
298:
299:             // Transform foot offset into world frame
300:             Eigen::Vector3d shoulder_pos;
301:             quad.worldToLegbaseFKWorldFrame(leg_index, body_pos, body_rpy,
302:                 shoulder_pos);
303:             foot_pos_world = shoulder_pos + foot_offset;
304:
305:             // Run IK and make sure there aren't any errors
306:             quad.worldToFootIKWorldFrame(leg_index, body_pos, body_rpy,
307:                 foot_pos_world, joint_state_test);
308:         }
```

```

309:      // To do: Check these solutions and make sure they are what we want
310:  }
311: }
312:
313: EXPECT_EQ(1 + 1, 2);
314: }
315:
316: TEST(KinematicsTest, testBodyToFootFK) {
317:     ros::NodeHandle nh;
318:
319:     // Declare kinematics object
320:     QuadKD quad;
321:
322:     // Set up problem variables
323:     Eigen::Matrix4d g_world_foot;
324:     Eigen::Matrix4d g_body_foot;
325:     Eigen::Vector3d foot_pos_body;
326:
327:     Eigen::Matrix4d g_body_foot_test;
328:     Eigen::Vector3d foot_pos_body_test;
329:
330:     double pos_min = -1.0;
331:     double pos_max = 1.0;
332:     double roll_min = -M_PI;
333:     double roll_max = M_PI;
334:     double pitch_min = -0.5 * M_PI;
335:     double pitch_max = 0.5 * M_PI;
336:     double yaw_min = -M_PI;
337:     double yaw_max = M_PI;
338:
339:     // Compute the kinematics
340:     int N = 10000;
341:     for (int config = 0; config < N; config++) {
342:         // Generate valid body orientations
343:         Eigen::Vector3d body_pos = {
344:             (pos_max - pos_min) * rand() / RAND_MAX + pos_min,
345:             (pos_max - pos_min) * rand() / RAND_MAX + pos_min,
346:             (pos_max - pos_min) * rand() / RAND_MAX + pos_min};
347:
348:         Eigen::Vector3d body_rpy = {
349:             (pos_max - roll_min) * rand() / RAND_MAX + roll_min,
350:             (pitch_max - pitch_min) * rand() / RAND_MAX + pitch_min,
351:             (yaw_max - yaw_min) * rand() / RAND_MAX + yaw_min};
352:
353:         Eigen::Matrix4d g_world_body = quad.createAffineMatrix(body_pos, body_rpy);
354:
355:         for (int leg_index = 0; leg_index < 4; leg_index++) {
356:             // Generate valid joint configurations
357:             Eigen::Vector3d joint_state = {(quad.getJointUpperLimit(leg_index, 0) -
358:                 quad.getJointLowerLimit(leg_index, 0)) *
359:                 (double)rand() / RAND_MAX +
360:                 quad.getJointLowerLimit(leg_index, 0),
361:                 (quad.getJointUpperLimit(leg_index, 1) -
362:                 quad.getJointLowerLimit(leg_index, 1)) *
363:                 (double)rand() / RAND_MAX +
364:                 quad.getJointLowerLimit(leg_index, 1),
365:                 (quad.getJointUpperLimit(leg_index, 2) -
366:                 quad.getJointLowerLimit(leg_index, 2)) *
367:                 (double)rand() / RAND_MAX +
368:                 quad.getJointLowerLimit(leg_index, 2)};
369:
370:             // Compute the foot position in world frame with FK then transform into
371:             // body frame
372:             quad.worldToFootFKWorldFrame(leg_index, body_pos, body_rpy, joint_state,
373:                 g_world_foot);
374:             quad.transformWorldToBody(body_pos, body_rpy, g_world_foot, g_body_foot);
375:             foot_pos_body = g_body_foot.block<3, 1>(0, 3);
376:
377:             // Compute foot positions directly from the body frame
378:             quad.bodyToFootFKBodyFrame(leg_index, joint_state, g_body_foot_test);
379:             quad.bodyToFootFKBodyFrame(leg_index, joint_state, foot_pos_body_test);
380:
381:             // Check the answers
382:             EXPECT_TRUE(foot_pos_body_test.isApprox(foot_pos_body));
383:             EXPECT_TRUE(g_body_foot_test.isApprox(g_body_foot));
384:         }
385:     }

```

```
386: }
387:
388: TEST(KinematicsTest, testMotorModel) {
389:     // Declare kinematics object
390:     QuadKD quad_kd;
391:
392:     Eigen::VectorXd state_vel(12);
393:     Eigen::VectorXd valid_input(12);
394:     Eigen::VectorXd invalid_input(12);
395:     Eigen::VectorXd constrained_input(12);
396:
397:     state_vel << 0, 0, 0, 10, 10, 10, 0, 0, 0, 10, 10, 10;
398:     valid_input << 10, 10, 10, 10, 10, 10, -10, -10, -10, -10, -10, -10;
399:     invalid_input << 40, 10, 10, 10, 10, 10, -10, -10, -10, -10, -10, -10;
400:
401:     bool valid_result =
402:         quad_kd.applyMotorModel(valid_input, state_vel, constrained_input);
403:     bool invalid_result =
404:         quad_kd.applyMotorModel(invalid_input, state_vel, constrained_input);
405:
406:     EXPECT_TRUE(valid_result == true);
407:     EXPECT_TRUE(invalid_result == false);
408:
409:     int N = 1000;
410:     int count = 0;
411:     auto t_start = std::chrono::steady_clock::now();
412:     for (int i = 0; i < N; i++) {
413:         count++;
414:         bool valid_result =
415:             quad_kd.applyMotorModel(valid_input, state_vel, constrained_input);
416:     }
417:     auto t_end = std::chrono::steady_clock::now();
418:
419:     std::chrono::duration<double> t_diff =
420:         std::chrono::duration_cast<std::chrono::duration<double>>(t_end -
421:                                                                     t_start);
422:     double average_time = t_diff.count() / count;
423:
424:     std::cout << "Average applyMotorModel time = " << average_time << " s"
425:         << std::endl;
426:
427:     EXPECT_TRUE(average_time <= 1e-6);
428: }
429:
430: TEST(KinematicsTest, testConvertCentroidalToFullBody) {
431:     // Declare kinematics object
432:     QuadKD quad_kd;
433:
434:     // Declare known variables
435:     Eigen::VectorXd body_state(12);
436:     Eigen::VectorXd foot_positions(12);
437:     Eigen::VectorXd foot_velocities(12);
438:     Eigen::VectorXd foot_acc(12);
439:     Eigen::VectorXd grfs(12);
440:     std::vector<int> contact_mode;
441:
442:     // Declare unknown variables
443:     Eigen::VectorXd joint_positions(12);
444:     Eigen::VectorXd joint_velocities(12);
445:     Eigen::VectorXd torques(12);
446:     Eigen::VectorXd state_violation, control_violation;
447:
448:     // Define terrain map
449:     grid_map::GridMap map({"z"});
450:     double map_height = 0;
451:     map.setGeometry(grid_map::Length(10.0, 10.0), 0.1,
452:                    grid_map::Position(0.0, 0.0));
453:     for (grid_map::GridMapIterator it(map); !it.isPastEnd(); ++it) {
454:         grid_map::Position position;
455:         map.getPosition(*it, position);
456:         map.at("z", *it) = map_height;
457:     }
458:
459:     int N_yaw = 10;
460:     for (int i = 0; i < N_yaw; i++) {
461:         // Define the nominal standing height and random x,y,yaw
462:         double h = 0.3;
```

```
463: double yaw = 2 * M_PI * (double)rand() / RAND_MAX - M_PI;
464: double x = 2 * (double)rand() / RAND_MAX - 1;
465: double y = 2 * (double)rand() / RAND_MAX - 1;
466: grid_map::Position pos = {x, y};
467: body_state << x, y, h + map.atPosition("z", pos), 0, 0, yaw, 0, 0, 0, 0, 0,
468: 0;
469:
470: // Extract components of the state
471: Eigen::Vector3d body_pos = body_state.segment<3>(0);
472: Eigen::Vector3d body_rpy = body_state.segment<3>(3);
473: Eigen::VectorXd body_vel = body_state.tail(6);
474:
475: // Solve FK for nominal joint angles to get foot positions
476: for (int i = 0; i < 4; i++) {
477:     Eigen::Vector3d nominal_hip_pos_world;
478:     quad_kd.worldToNominalHipFKWorldFrame(i, body_pos, body_rpy,
479:         nominal_hip_pos_world);
480:     nominal_hip_pos_world[2] = 0;
481:     foot_positions.segment<3>(3 * i) = nominal_hip_pos_world;
482: }
483:
484: // Define dynamic parameters for a trot
485: double m = 11.5;
486: double g = 9.81;
487: grfs << 0, 0, 0.5 * m * g, 0, 0, 0, 0, 0, 0, 0, 0.5 * m * g;
488: contact_mode = {1, 0, 0, 1};
489:
490: // Define foot velocities (feet not in contact have upwards velocity)
491: foot_velocities.setZero();
492: double foot_vel_z = 1.0;
493: foot_velocities[5] = foot_vel_z;
494: foot_velocities[8] = foot_vel_z;
495: foot_acc.setZero();
496:
497: // Perform conversion
498: bool is_exact = quad_kd.convertCentroidalToFullBody(
499:     body_state, foot_positions, foot_velocities, grfs, joint_positions,
500:     joint_velocities, torques);
501:
502: // Compute expected joint positions
503: double l1 = quad_kd.getLinkLength(0, 2);
504: Eigen::VectorXd joint_positions_expected(12), joint_velocities_expected(12);
505: joint_positions_expected << 0, asin(0.5 * h / l1), 2 * asin(0.5 * h / l1),
506: 0, asin(0.5 * h / l1), 2 * asin(0.5 * h / l1), 0, asin(0.5 * h / l1),
507: 2 * asin(0.5 * h / l1), 0, asin(0.5 * h / l1), 2 * asin(0.5 * h / l1);
508:
509: // Compute expected joint velocities
510: double hip_vel_expected =
511:     -0.5 * foot_vel_z / (l1 * cos(joint_positions_expected[4]));
512: double knee_vel_expected = 2 * hip_vel_expected;
513: joint_velocities_expected << 0, 0, 0, 0, hip_vel_expected,
514:     knee_vel_expected, 0, hip_vel_expected, knee_vel_expected, 0, 0, 0;
515:
516: // Check joint positions and velocities match
517: EXPECT_TRUE(is_exact);
518: EXPECT_TRUE(joint_positions.isApprox(joint_positions_expected));
519: EXPECT_TRUE(joint_velocities.isApprox(joint_velocities_expected));
520:
521: // Check validity
522: bool is_state_valid = quad_kd.isValidCentroidalState(
523:     body_state, foot_positions, foot_velocities, grfs, map, joint_positions,
524:     joint_velocities, torques, state_violation, control_violation);
525: EXPECT_TRUE(is_state_valid);
526:
527: body_state[2] += 0.5;
528: is_exact = quad_kd.convertCentroidalToFullBody(
529:     body_state, foot_positions, foot_velocities, grfs, joint_positions,
530:     joint_velocities, torques);
531: EXPECT_FALSE(is_exact);
532:
533: // Check validity
534: is_state_valid = quad_kd.isValidCentroidalState(
535:     body_state, foot_positions, foot_velocities, grfs, map, joint_positions,
536:     joint_velocities, torques, state_violation, control_violation);
537: EXPECT_FALSE(is_state_valid);
538: }
539:
```



```
540: // Print results if desired
541: // std::cout << "joint_positions\n" << joint_positions << std::endl;
542: // std::cout << "joint_velocities\n" << joint_velocities << std::endl;
543: // std::cout << "torques\n" << torques << std::endl;
544:
545: // Check timing characteristics
546: int N = 1000;
547: int count = 0;
548: auto t_start = std::chrono::steady_clock::now();
549: for (int i = 0; i < N; i++) {
550:     count++;
551:     quad_kd.convertCentroidalToFullBody(body_state, foot_positions,
552:                                         foot_velocities, grfs, joint_positions,
553:                                         joint_velocities, torques);
554: }
555: auto t_end = std::chrono::steady_clock::now();
556:
557: std::chrono::duration<double> t_diff =
558:     std::chrono::duration_cast<std::chrono::duration<double>>(t_end -
559:                                                                t_start);
560: double average_time = t_diff.count() / count;
561:
562: std::cout << "Average convertCentroidalToFullBody time = " << average_time
563:             << " s" << std::endl;
564:
565: EXPECT_TRUE(average_time < 1e-4);
566: }
```