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

#include "quad_utils/fast_terrain_map.h"

TEST(FastTerrainMapTest, testSpeedComparison) {
  // Define map parameters
  double res = 0.01;
  double x_length = 12.0;
  double y_length = 5.0;
  double x_center = 4.0;
  double y_center = 0.0;
  double x_origin = x_center - 0.5 * x_length;
  double y_origin = y_center - 0.5 * y_length;

  // Create GridMap
  grid_map::GridMap grid_map_obj({"z", "z_inpainted", "nx", "ny", "nz",
                                  "z_filt", "nx_filt", "ny_filt", "nz_filt"});
  // grid_map_obj.setBasicLayers({"z","nx","ny","nz","z_filt","nx_filt","ny_filt","nz_filt"});
  grid_map_obj.setFrameId("map");
  grid_map_obj.setGeometry(grid_map::Length(x_length, y_length), res,
                           grid_map::Position(x_center, y_center));
  int x_size = grid_map_obj.getSize()(0);
  int y_size = grid_map_obj.getSize()(1);
  // printf("Created map with size %f x %f m (%i x %i cells).\n",
  //   grid_map_obj.getLength().x(), grid_map_obj.getLength().y(), x_size,
  //   y_size);

  // Load grid map with random noise
  for (grid_map::GridMapIterator it(grid_map_obj); !it.isPastEnd(); ++it) {
    grid_map_obj.at("z", *it) = 0.1 * ((double)rand() / RAND_MAX);
    grid_map_obj.at("z_inpainted", *it) = grid_map_obj.at("z", *it);
    grid_map_obj.at("z_filt", *it) = grid_map_obj.at("z", *it);

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

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

  // Load data into fast terrain map obj
  FastTerrainMap fast_terrain_map;
  fast_terrain_map.loadDataFromGridMap(grid_map_obj);

  // Initialize testing parameters
  const int N = 10001;
  grid_map::Position pos;
  double x, y, z;
  std::chrono::time_point<std::chrono::steady_clock> start_time,
      intermediate_time_1, intermediate_time_2, stop_time;
  std::chrono::duration<double> elapsed;
  std::vector<double> timings(N - 1);

  // Start GridMap nearest neighbor
  double gm_nn_total_elapsed = 0;
  start_time = std::chrono::steady_clock::now();
  for (int i = 0; i < N; i++) {
    // Generate random test point
    x = (x_length - res) * ((double)rand() / RAND_MAX) + x_origin + 0.5 * res;
    y = (y_length - res) * ((double)rand() / RAND_MAX) + y_origin + 0.5 * res;

    // Query
    pos = {x, y};
    intermediate_time_1 = std::chrono::steady_clock::now();
    z += grid_map_obj.atPosition(
             "z", pos, grid_map::InterpolationMethods::INTER_NEAREST) /
         N;
    intermediate_time_2 = std::chrono::steady_clock::now();
    elapsed = std::chrono::duration_cast<std::chrono::duration<double>>(
        intermediate_time_2 - intermediate_time_1);

    if (i > 0) {
      timings[i - 1] = (double)elapsed.count();
    } else {
```

```cpp
      start_time = std::chrono::steady_clock::now();
    }
  }

  stop_time = std::chrono::steady_clock::now();
  elapsed = std::chrono::duration_cast<std::chrono::duration<double>>(
      stop_time - start_time);
  double gm_nn_time = (double)elapsed.count();
  for (int i = 0; i < timings.size(); i++) {
    gm_nn_total_elapsed += timings[i];
    if (fmod(log10(i), 1) == 0) {
      // printf("Duration of nn iteration %d = %.3fus\n", i, 1e6*timings[i]);
    }
  }

  // Start GridMap linear
  double gm_lin_total_elapsed = 0;
  z = 0;
  start_time = std::chrono::steady_clock::now();
  for (int i = 0; i < N; i++) {
    // Generate random test point
    x = (x_length - res) * ((double)rand() / RAND_MAX) + x_origin + 0.5 * res;
    y = (y_length - res) * ((double)rand() / RAND_MAX) + y_origin + 0.5 * res;

    // Query
    pos = {x, y};
    intermediate_time_1 = std::chrono::steady_clock::now();
    z += grid_map_obj.atPosition("z", pos,
                                 grid_map::InterpolationMethods::INTER_LINEAR) /
         N;
    intermediate_time_2 = std::chrono::steady_clock::now();
    elapsed = std::chrono::duration_cast<std::chrono::duration<double>>(
        intermediate_time_2 - intermediate_time_1);

    if (i > 0) {
      timings[i - 1] = (double)elapsed.count();
    } else {
      start_time = std::chrono::steady_clock::now();
    }
  }

  stop_time = std::chrono::steady_clock::now();
  elapsed = std::chrono::duration_cast<std::chrono::duration<double>>(
      stop_time - start_time);
  double gm_lin_time = (double)elapsed.count();
  for (int i = 0; i < timings.size(); i++) {
    gm_lin_total_elapsed += timings[i];
    if (fmod(log10(i), 1) == 0) {
      // printf("Duration of lin iteration %d = %.3fus\n", i, 1e6*timings[i]);
    }
  }

  // Start FastTerrainMap linear
  double ftm_lin_total_elapsed = 0;
  z = 0;
  start_time = std::chrono::steady_clock::now();
  for (int i = 0; i < N; i++) {
    // Generate random test point
    x = (x_length - res) * ((double)rand() / RAND_MAX) + x_origin + 0.5 * res;
    y = (y_length - res) * ((double)rand() / RAND_MAX) + y_origin + 0.5 * res;

    // Query
    pos = {x, y};
    intermediate_time_1 = std::chrono::steady_clock::now();
    z += fast_terrain_map.getGroundHeight(x, y) / N;
    intermediate_time_2 = std::chrono::steady_clock::now();
    elapsed = std::chrono::duration_cast<std::chrono::duration<double>>(
        intermediate_time_2 - intermediate_time_1);

    if (i > 0) {
      timings[i - 1] = (double)elapsed.count();
    } else {
      start_time = std::chrono::steady_clock::now();
    }
  }

  stop_time = std::chrono::steady_clock::now();
```

```cpp
  elapsed = std::chrono::duration_cast<std::chrono::duration<double>>(
      stop_time - start_time);
  double ftm_lin_time = (double)elapsed.count();
  for (int i = 0; i < timings.size(); i++) {
    ftm_lin_total_elapsed += timings[i];
    if (fmod(log10(i), 1) == 0) {
      // printf("Duration of lin iteration %d = %.3fus\n", i, 1e6*timings[i]);
    }
  }

  printf("GridMap NN avg (total duration/iterations) = %.3fus\n",
         (double)1e6 * gm_nn_time / N);
  // printf("GridMap NN avg (sum of individual/iterations) = %.3fus\n",
  // (double)1e6*gm_nn_total_elapsed/timings.size());
  printf("GridMap linear avg (total duration/iterations) = %.3fus\n",
         (double)1e6 * gm_lin_time / N);
  // printf("GridMap linear avg (sum of individual/iterations) = %.3fus\n",
  // (double)1e6*gm_lin_total_elapsed/timings.size());
  printf("FastTerrainMap linear avg (total duration/iterations) = %.3fus\n",
         (double)1e6 * ftm_lin_time / N);
  // printf("FastTerrainMap linear avg (sum of individual/iterations) =
  // %.3fus\n", (double)1e6*ftm_lin_total_elapsed/timings.size()); printf("z avg
  // = %.3f\n", (double)z);

  EXPECT_EQ(1 + 1, 2);
}

TEST(FastTerrainMapTest, testConstructor) {
  FastTerrainMap fast_terrain_map;
  EXPECT_EQ(1 + 1, 2);
}

TEST(FastTerrainMapTest, testProjection) {
  FastTerrainMap fast_terrain_map;

  int x_size = 2;
  int y_size = 2;
  std::vector<double> x_data = {-1, 1};
  std::vector<double> y_data = {-1, 1};
  std::vector<double> z_data_vec = {-1, 1};
  std::vector<std::vector<double>> z_data = {z_data_vec, z_data_vec};

  std::vector<double> dx_data_vec = {0, 0};
  std::vector<double> dz_data_vec = {1, 1};
  std::vector<std::vector<double>> dx_data = {dx_data_vec, dx_data_vec};
  std::vector<std::vector<double>> dy_data = dx_data;
  std::vector<std::vector<double>> dz_data = {dz_data_vec, dz_data_vec};

  fast_terrain_map.loadData(x_size, y_size, x_data, y_data, z_data, dx_data,
                            dy_data, dz_data, z_data, dx_data, dy_data,
                            dz_data);

  // Eigen::Vector3d point = {0,0.5,1};
  // Eigen::Vector3d direction = {0.1,0.1,-1};
  Eigen::Vector3d point = {0, 0, 1};
  Eigen::Vector3d direction = {0, 0, -1};

  auto t_start = std::chrono::steady_clock::now();
  Eigen::Vector3d intersection =
      fast_terrain_map.projectToMap(point, direction);
  auto t_end = std::chrono::steady_clock::now();
  std::chrono::duration<double> time_span =
      std::chrono::duration_cast<std::chrono::duration<double>>(t_end -
                                                                t_start);
  // std::cout << "projectToMap took " << time_span.count() << " seconds." <<
  // std::endl;

  // std::cout << "Result is {" << intersection[0] << ", " << intersection[1] <<
  // ", " << intersection[2] << "}" << std::endl;

  EXPECT_EQ(1 + 1, 2);
}

TEST(FastTerrainMapTest, testSlope) {
  FastTerrainMap map;
  double grade = 0.5;
  map.loadSlope(grade);
```

```cpp

  Eigen::Vector3d normal;
  normal << -sin(atan(grade)), 0, cos(atan(grade));

  double x = -2;
  double y = 0;
  double z = grade * x;
  EXPECT_TRUE(abs(map.getGroundHeight(x, y) - z) < 1e-6);

  x = 2;
  y = 0;
  z = grade * x;
  EXPECT_TRUE(abs(map.getGroundHeight(x, y) - z) < 1e-6);

  x = 0;
  y = 2;
  z = grade * x;
  EXPECT_TRUE(abs(map.getGroundHeight(x, y) - z) < 1e-6);

  EXPECT_TRUE(map.getSurfaceNormalFilteredEigen(x, y).isApprox(normal));
}

TEST(FastTerrainMapTest, testStep) {
  FastTerrainMap map;
  double height = 0.2;
  map.loadStep(height);

  Eigen::Vector3d normal;
  normal << 0, 0, 1;

  double x = -2;
  double y = 0;
  double z = (x > 0) ? height : 0;
  EXPECT_TRUE(abs(map.getGroundHeight(x, y) - z) < 1e-6);

  x = 2;
  y = 0;
  z = (x > 0) ? height : 0;
  EXPECT_TRUE(abs(map.getGroundHeight(x, y) - z) < 1e-6);

  x = 0;
  y = 2;
  z = (x > 0) ? height : 0;
  EXPECT_TRUE(abs(map.getGroundHeight(x, y) - z) < 1e-6);

  EXPECT_TRUE(map.getSurfaceNormalFilteredEigen(x, y).isApprox(normal));
}
```

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

#include "quad_utils/terrain_map_publisher.h"

TEST(TerrainMapPublisherTest, testTrue) {
  ros::NodeHandle nh;
  TerrainMapPublisher terrain_map_publisher(nh);
  EXPECT_EQ(1 + 1, 2);
}
```

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

#include "quad_utils/ros_utils.h"
Eigen::IOFormat CleanFmt(4, 0, ", ", "\n", "[", "]");

TEST(EigenTest, testMap) {
  const int N = 9;
  double data_c[N];
  for (int i = 0; i < N; ++i) {
    data_c[i] = (double)i;
  }

  // std::cout << Eigen::Map<Eigen::VectorXi>(array) << std::endl;

  Eigen::MatrixXd data_eigen;

  data_eigen = Eigen::Map<Eigen::Matrix<double, 1, N>>(data_c);

  // std::cout << data_eigen.format(CleanFmt) << std::endl;
}
```

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

int main(int argc, char** argv) {
  testing::InitGoogleTest(&argc, argv);
  ros::init(argc, argv, "quad_utils_tester");

  return RUN_ALL_TESTS();
}
```

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

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

TEST(MathTest, testWrap) {
  int N = 201;
  double amplitude = 10;
  double period = 4 * M_PI;
  std::vector<double> data(N), t(N);
  for (int i = 0; i < data.size(); i++) {
    t[i] = i * period / N;
    data[i] = amplitude * sin(t[i]);
  }

  std::vector<double> data_wrapped = math_utils::wrapToPi(data);
  std::vector<double> data_unwrapped = math_utils::unwrap(data_wrapped);

  double error = 0;
  for (int i = 0; i < data.size(); i++) {
    error += abs(data[i] - data_unwrapped[i]);
  }

  double tolerance = 1e-4;
  EXPECT_TRUE(error <= tolerance);
}
```

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

#include "quad_utils/rviz_interface.h"

TEST(RVizInterfaceTest, testTrue) {
  ros::NodeHandle nh;
  RVizInterface rviz_interface(nh);
  EXPECT_EQ(1 + 1, 2);
}
```

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

#include <grid_map_core/grid_map_core.hpp>

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

using namespace quad_utils;

const double kinematics_tol = 1e-4;

TEST(KinematicsTest, testDifferentialFKIK) {
  // Declare kinematics object
  QuadKD kinematics;

  for (size_t i = 0; i < 20; i++) {
    // Declare input and output RobotState object
    quad_msgs::RobotState state, state_out;

    // Random velocities at origin
    Eigen::VectorXd body_state(12);
    body_state << (double)rand() / RAND_MAX - 0.5,
        (double)rand() / RAND_MAX - 0.5, (double)rand() / RAND_MAX - 0.5,
        1.5 * (double)rand() / RAND_MAX - 0.75,
        1.5 * (double)rand() / RAND_MAX - 0.75,
        1.5 * (double)rand() / RAND_MAX - 0.75,
        10 * (double)rand() / RAND_MAX - 5, 10 * (double)rand() / RAND_MAX - 5,
        10 * (double)rand() / RAND_MAX - 5,
        3.14 * (double)rand() / RAND_MAX - 1.57,
        3.14 * (double)rand() / RAND_MAX - 1.57,
        3.14 * (double)rand() / RAND_MAX - 1.57;

    state.body = eigenToBodyStateMsg(body_state);

    state.joints.name = {"8",  "0", "1", "9",  "2", "3",
                         "10", "4", "5", "11", "6", "7"};
    state.joints.position.clear();
    state.joints.velocity.clear();
    state.joints.effort.clear();

    for (int j = 0; j < 4; j++) {
      // Just some arbitary joints position
      state.joints.position.push_back(0.1);
      state.joints.position.push_back(0.2);
      state.joints.position.push_back(0.3);

      // Random joints velocity
      state.joints.velocity.push_back(3.14 * (double)rand() / RAND_MAX - 1.57);
      state.joints.velocity.push_back(3.14 * (double)rand() / RAND_MAX - 1.57);
      state.joints.velocity.push_back(3.14 * (double)rand() / RAND_MAX - 1.57);

      // We don’t need joints effort here
      state.joints.effort.push_back(0.0);
      state.joints.effort.push_back(0.0);
      state.joints.effort.push_back(0.0);
    }

    // Run FK get foot velocities and IK them back
    quad_utils::fkRobotState(kinematics, state.body, state.joints, state.feet);
    quad_utils::ikRobotState(kinematics, state.body, state.feet,
                             state_out.joints);

    // Extract input joint velocities
    Eigen::VectorXd vel(12), vel_out(12);
    vectorToEigen(state.joints.velocity, vel);
    vectorToEigen(state_out.joints.velocity, vel_out);

    // Check the answers
    Eigen::VectorXd error = vel - vel_out;
    EXPECT_TRUE(error.norm() <= kinematics_tol);
  }
}

TEST(KinematicsTest, testFootForces) {
  // Declare kinematics object
  QuadKD kinematics;
```

```cpp

  // Length parameters from URDF
  // TODO(yanhaoy): load these from parameters rather than hard-coding
  Eigen::MatrixXd ls(4, 3);
  ls << 0.2263, 0.07, 0.0,  // abad from body
      0.0, 0.10098, 0.0,    // hip from abad
      -0.206, 0.0, 0.0,     // knee from hip
      0.206, 0.0, 0.0;      // toe from knee

  double pi = 3.14159265359;

  // Define vectors for states, forces, and torques
  Eigen::VectorXd state_positions(18), forces(12), torques(18),
      torques_solution(18);

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

  // Set up known solution problem 1 ----------------------------------
  state_positions = Eigen::VectorXd::Zero(18);
  for (int i = 0; i < 3; i++) {
    // move the CG around randomly -- it should not matter
    state_positions(12 + i) = (double)rand() / RAND_MAX - 0.5;
  }
  forces = Eigen::VectorXd::Zero(12);
  forces(2) = 3.0;  // front left toe Z
  forces(3) = 2.0;  // back left toe X

  // Known solution
  torques_solution << 3.0 * ls(1, 1), 0.0, 3.0 * -ls(3, 0), 0.0, 0.0,
      0.0,                                             // leg 2
      0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 2.0, 0.0, 3.0,  // net forces
      3.0 * (ls(0, 1) + ls(1, 1)), 3.0 * -ls(0, 0),
      -2.0 * (ls(0, 1) + ls(1, 1));

  // Compute joint torques
  kinematics.getJacobianGenCoord(state_positions, jacobian);
  torques = jacobian.transpose() * forces;

  // Check the answers
  Eigen::VectorXd error = torques - torques_solution;
  Eigen::MatrixXd toPrint(18, 2);
  toPrint << torques, torques_solution;
  // std::cout << "Test 1:\n" << toPrint << std::endl;
  EXPECT_TRUE(error.norm() <= kinematics_tol);

  // Set up known solution problem 2 ----------------------------------
  state_positions = Eigen::VectorXd::Zero(18);
  for (int i = 0; i < 3; i++) {
    // move the CG around randomly -- it should not matter
    state_positions(12 + i) = (double)rand() / RAND_MAX - 0.5;
  }
  state_positions(17) = pi / 2;  // yaw 90 deg left
  state_positions(7) = pi / 4;   // front right hip 45 deg down
  state_positions(8) = pi / 2;   // front right knee 90 deg down
  forces = Eigen::VectorXd::Zero(12);
  forces(6) = 3.0;  // front right toe X
  forces(8) = 5.0;  // front right toe Z

  // Known solution
  torques_solution << 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,  // leg 2
      -5.0 * ls(1, 1) -
          3.0 * (-ls(2, 0) * sin(pi / 4) + ls(3, 0) * sin(pi / 4)),
      0.0, 5.0 * -ls(3, 0) * cos(pi / 4), 0.0, 0.0, 0.0, 3.0, 0.0,
      5.0,  // net forces
      -5.0 * (ls(0, 1) + ls(1, 1)) -
          3.0 * (-ls(2, 0) * sin(pi / 4) + ls(3, 0) * sin(pi / 4)),
      -5.0 * ls(0, 0), -3.0 * ls(0, 0);

  // Compute joint torques
  kinematics.getJacobianGenCoord(state_positions, jacobian);
  torques = jacobian.transpose() * forces;

  // Check the answers
  error = torques - torques_solution;
  toPrint << torques, torques_solution;
  // std::cout << "Test 2:\n" << toPrint << std::endl;
```

```cpp
  EXPECT_TRUE(error.norm() <= kinematics_tol);

  // Set up known solution problem 3 ----------------------------------
  state_positions = Eigen::VectorXd::Zero(18);
  for (int i = 0; i < 3; i++) {
    // move the CG around randomly -- it should not matter
    state_positions(12 + i) = (double)rand() / RAND_MAX - 0.5;
  }
  state_positions(15) = pi / 2;  // roll 90 deg right
  state_positions(17) = pi / 2;  // yaw 90 deg left
  forces = Eigen::VectorXd::Zero(12);
  forces(0) = 1.0;  // front left toe X

  // Known solution
  torques_solution << ls(1, 1), 0.0, -ls(3, 0), 0.0, 0.0, 0.0,  // leg 2
      0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,              // net forces
      ls(0, 1) + ls(1, 1), 0.0, -ls(0, 0);

  // Compute joint torques
  kinematics.getJacobianGenCoord(state_positions, jacobian);
  torques = jacobian.transpose() * forces;

  // Check the answers
  error = torques - torques_solution;
  toPrint << torques, torques_solution;
  // std::cout << "Test 3:\n" << toPrint << std::endl;
  EXPECT_TRUE(error.norm() <= kinematics_tol);

  // Set up known solution problem 4 ----------------------------------
  state_positions = Eigen::VectorXd::Zero(18);
  for (int i = 0; i < 3; i++) {
    // move the CG around randomly -- it should not matter
    state_positions(12 + i) = (double)rand() / RAND_MAX - 0.5;
  }
  state_positions(16) = pi / 4;  // pitch 45 deg down
  state_positions(17) = pi;      // yaw 180 deg
  state_positions(1) = pi / 4;   // front left hip 60 deg down
  state_positions(4) = -pi / 4;  // back left hip 45 deg up
  state_positions(7) = -pi / 4;  // front right hip 45 deg up
  state_positions(10) = pi / 4;  // back right hip 60 deg down
  forces = Eigen::VectorXd::Zero(12);
  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;

  // Known solution
  torques_solution << sqrt(2) * ls(1, 1), 0.0, -ls(3, 0), sqrt(2) * ls(1, 1),
      0.0, -ls(3, 0),  // leg 2
      -sqrt(2) * ls(1, 1), 0.0, -ls(3, 0), -sqrt(2) * ls(1, 1), 0.0, -ls(3, 0),
      -4.0, 0.0, 4.0,  // net forces
      0.0, 0.0, 0.0;

  // Compute joint torques
  kinematics.getJacobianGenCoord(state_positions, jacobian);
  torques = jacobian.transpose() * forces;

  // Check the answers
  error = torques - torques_solution;
  toPrint << torques, torques_solution;
  // std::cout << "Test 4:\n" << toPrint << std::endl;
  EXPECT_TRUE(error.norm() <= kinematics_tol);
}

TEST(KinematicsTest, testFKIKFeasibleConfigurations) {
  ros::NodeHandle nh;

  // Declare kinematics object
  QuadKD quad;

  // Set up problem variables
  Eigen::Vector3d body_pos = {0, 0, 0};
  Eigen::Vector3d body_rpy = {0, 0, 0};
  Eigen::Vector3d foot_pos_world;
  Eigen::Vector3d joint_state_test;
  Eigen::Vector3d foot_pos_world_test;

  // Compute the kinematics
  int N = 10000;
  for (int config = 0; config < N; config++) {
```

```cpp
    for (int i = 0; i < 4; i++) {
      int leg_index = i;

      // Generate valid joint configurations
      Eigen::Vector3d joint_state = {(quad.getJointUpperLimit(leg_index, 0) -
                                      quad.getJointLowerLimit(leg_index, 0)) *
                                             (double)rand() / RAND_MAX +
                                         quad.getJointLowerLimit(leg_index, 0),
                                     (quad.getJointUpperLimit(leg_index, 1) -
                                      quad.getJointLowerLimit(leg_index, 1)) *
                                             (double)rand() / RAND_MAX +
                                         quad.getJointLowerLimit(leg_index, 1),
                                     (quad.getJointUpperLimit(leg_index, 2) -
                                      quad.getJointLowerLimit(leg_index, 2)) *
                                             (double)rand() / RAND_MAX +
                                         quad.getJointLowerLimit(leg_index, 2)};

      // Compute foot positions in this configuration
      quad.worldToFootFKWorldFrame(leg_index, body_pos, body_rpy, joint_state,
                                   foot_pos_world);

      // Run IK to compute corresponding joint angles, then back through FK
      // This ensures that we are enforcing a hip-above-knee configuration if
      // otherwise ambiguous.
      quad.worldToFootIKWorldFrame(leg_index, body_pos, body_rpy,
                                   foot_pos_world, joint_state_test);

      // Skip if original configuration was in an alternate configuration
      if (!joint_state_test.isApprox(joint_state)) continue;

      quad.worldToFootFKWorldFrame(leg_index, body_pos, body_rpy,
                                   joint_state_test, foot_pos_world_test);

      // Check the answers
      Eigen::Vector3d error = (foot_pos_world - foot_pos_world_test);
      EXPECT_LE(error.norm(), kinematics_tol);
    }
  }
}

TEST(KinematicsTest, testFKIKInfeasibleConfigurations) {
  ros::NodeHandle nh;

  QuadKD quad;

  // Set up problem variables
  Eigen::Vector3d body_pos = {0, 0, 0};
  Eigen::Vector3d body_rpy = {0, 0, 0};
  Eigen::Vector3d foot_pos_world;
  Eigen::Vector3d foot_pos_world_test;
  Eigen::Vector3d joint_state_test;

  // Define arbitrary maximum foot offset for IK testing
  double max_offset = 2.0;

  // Test random foot positions to make sure nothing breaks
  int N = 10000;
  for (int config = 0; config < N; config++) {
    // Generate random foot offset
    Eigen::Vector3d foot_offset = {
        2 * max_offset * (double)rand() / RAND_MAX - max_offset,
        2 * max_offset * (double)rand() / RAND_MAX - max_offset,
        2 * max_offset * (double)rand() / RAND_MAX - max_offset};

    for (int i = 0; i < 4; i++) {
      int leg_index = i;

      // Transform foot offset into world frame
      Eigen::Vector3d shoulder_pos;
      quad.worldToLegbaseFKWorldFrame(leg_index, body_pos, body_rpy,
                                      shoulder_pos);
      foot_pos_world = shoulder_pos + foot_offset;

      // Run IK and make sure there aren't any errors
      quad.worldToFootIKWorldFrame(leg_index, body_pos, body_rpy,
                                   foot_pos_world, joint_state_test);

```

```cpp
      // To do: Check these solutions and make sure they are what we want
    }
  }

  EXPECT_EQ(1 + 1, 2);
}

TEST(KinematicsTest, testBodyToFootFK) {
  ros::NodeHandle nh;

  // Declare kinematics object
  QuadKD quad;

  // Set up problem variables
  Eigen::Matrix4d g_world_foot;
  Eigen::Matrix4d g_body_foot;
  Eigen::Vector3d foot_pos_body;

  Eigen::Matrix4d g_body_foot_test;
  Eigen::Vector3d foot_pos_body_test;

  double pos_min = -1.0;
  double pos_max = 1.0;
  double roll_min = -M_PI;
  double roll_max = M_PI;
  double pitch_min = -0.5 * M_PI;
  double pitch_max = 0.5 * M_PI;
  double yaw_min = -M_PI;
  double yaw_max = M_PI;

  // Compute the kinematics
  int N = 10000;
  for (int config = 0; config < N; config++) {
    // Generate valid body orientations
    Eigen::Vector3d body_pos = {
        (pos_max - pos_min) * rand() / RAND_MAX + pos_min,
        (pos_max - pos_min) * rand() / RAND_MAX + pos_min,
        (pos_max - pos_min) * rand() / RAND_MAX + pos_min};

    Eigen::Vector3d body_rpy = {
        (pos_max - roll_min) * rand() / RAND_MAX + roll_min,
        (pitch_max - pitch_min) * rand() / RAND_MAX + pitch_min,
        (yaw_max - yaw_min) * rand() / RAND_MAX + yaw_min};

    Eigen::Matrix4d g_world_body = quad.createAffineMatrix(body_pos, body_rpy);

    for (int leg_index = 0; leg_index < 4; leg_index++) {
      // Generate valid joint configurations
      Eigen::Vector3d joint_state = {(quad.getJointUpperLimit(leg_index, 0) -
                                      quad.getJointLowerLimit(leg_index, 0)) *
                                             (double)rand() / RAND_MAX +
                                         quad.getJointLowerLimit(leg_index, 0),
                                     (quad.getJointUpperLimit(leg_index, 1) -
                                      quad.getJointLowerLimit(leg_index, 1)) *
                                             (double)rand() / RAND_MAX +
                                         quad.getJointLowerLimit(leg_index, 1),
                                     (quad.getJointUpperLimit(leg_index, 2) -
                                      quad.getJointLowerLimit(leg_index, 2)) *
                                             (double)rand() / RAND_MAX +
                                         quad.getJointLowerLimit(leg_index, 2)};

      // Compute the foot position in world frame with FK then tranform into
      // body frame
      quad.worldToFootFKWorldFrame(leg_index, body_pos, body_rpy, joint_state,
                                   g_world_foot);
      quad.transformWorldToBody(body_pos, body_rpy, g_world_foot, g_body_foot);
      foot_pos_body = g_body_foot.block<3, 1>(0, 3);

      // Compute foot positions directly from the body frame
      quad.bodyToFootFKBodyFrame(leg_index, joint_state, g_body_foot_test);
      quad.bodyToFootFKBodyFrame(leg_index, joint_state, foot_pos_body_test);

      // Check the answers
      EXPECT_TRUE(foot_pos_body_test.isApprox(foot_pos_body));
      EXPECT_TRUE(g_body_foot_test.isApprox(g_body_foot));
    }
  }
```

```
}

TEST(KinematicsTest, testMotorModel) {
  // Declare kinematics object
  QuadKD quad_kd;

  Eigen::VectorXd state_vel(12);
  Eigen::VectorXd valid_input(12);
  Eigen::VectorXd invalid_input(12);
  Eigen::VectorXd constrained_input(12);

  state_vel << 0, 0, 0, 10, 10, 10, 0, 0, 0, 10, 10, 10;
  valid_input << 10, 10, 10, 10, 10, 10, -10, -10, -10, -10, -10, -10;
  invalid_input << 40, 10, 10, 10, 10, 10, -10, -10, -10, -10, -10, -10;

  bool valid_result =
      quad_kd.applyMotorModel(valid_input, state_vel, constrained_input);
  bool invalid_result =
      quad_kd.applyMotorModel(invalid_input, state_vel, constrained_input);

  EXPECT_TRUE(valid_result == true);
  EXPECT_TRUE(invalid_result == false);

  int N = 1000;
  int count = 0;
  auto t_start = std::chrono::steady_clock::now();
  for (int i = 0; i < N; i++) {
    count++;
    bool valid_result =
        quad_kd.applyMotorModel(valid_input, state_vel, constrained_input);
  }
  auto t_end = std::chrono::steady_clock::now();

  std::chrono::duration<double> t_diff =
      std::chrono::duration_cast<std::chrono::duration<double>>(t_end -
                                                                 t_start);
  double average_time = t_diff.count() / count;

  std::cout << "Average applyMotorModel time = " << average_time << " s"
            << std::endl;

  EXPECT_TRUE(average_time <= 1e-6);
}

TEST(KinematicsTest, testConvertCentroidalToFullBody) {
  // Declare kinematics object
  QuadKD quad_kd;

  // Declare known variables
  Eigen::VectorXd body_state(12);
  Eigen::VectorXd foot_positions(12);
  Eigen::VectorXd foot_velocities(12);
  Eigen::VectorXd foot_acc(12);
  Eigen::VectorXd grfs(12);
  std::vector<int> contact_mode;

  // Declare unknown variables
  Eigen::VectorXd joint_positions(12);
  Eigen::VectorXd joint_velocities(12);
  Eigen::VectorXd torques(12);
  Eigen::VectorXd state_violation, control_violation;

  // Define terrain map
  grid_map::GridMap map({"z"});
  double map_height = 0;
  map.setGeometry(grid_map::Length(10.0, 10.0), 0.1,
                  grid_map::Position(0.0, 0.0));
  for (grid_map::GridMapIterator it(map); !it.isPastEnd(); ++it) {
    grid_map::Position position;
    map.getPosition(*it, position);
    map.at("z", *it) = map_height;
  }

  int N_yaw = 10;
  for (int i = 0; i < N_yaw; i++) {
    // Define the nominal standing height and random x,y,yaw
    double h = 0.3;
```

```cpp
    double yaw = 2 * M_PI * (double)rand() / RAND_MAX - M_PI;
    double x = 2 * (double)rand() / RAND_MAX - 1;
    double y = 2 * (double)rand() / RAND_MAX - 1;
    grid_map::Position pos = {x, y};
    body_state << x, y, h + map.atPosition("z", pos), 0, 0, yaw, 0, 0, 0, 0, 0,
        0;

    // Extract components of the state
    Eigen::Vector3d body_pos = body_state.segment<3>(0);
    Eigen::Vector3d body_rpy = body_state.segment<3>(3);
    Eigen::VectorXd body_vel = body_state.tail(6);

    // Solve FK for nominal joint angles to get foot positions
    for (int i = 0; i < 4; i++) {
      Eigen::Vector3d nominal_hip_pos_world;
      quad_kd.worldToNominalHipFKWorldFrame(i, body_pos, body_rpy,
                                            nominal_hip_pos_world);
      nominal_hip_pos_world[2] = 0;
      foot_positions.segment<3>(3 * i) = nominal_hip_pos_world;
    }

    // Define dynamic parameters for a trot
    double m = 11.5;
    double g = 9.81;
    grfs << 0, 0, 0.5 * m * g, 0, 0, 0, 0, 0, 0, 0, 0, 0.5 * m * g;
    contact_mode = {1, 0, 0, 1};

    // Define foot velocities (feet not in contact have upwards velocity)
    foot_velocities.setZero();
    double foot_vel_z = 1.0;
    foot_velocities[5] = foot_vel_z;
    foot_velocities[8] = foot_vel_z;
    foot_acc.setZero();

    // Perform conversion
    bool is_exact = quad_kd.convertCentroidalToFullBody(
        body_state, foot_positions, foot_velocities, grfs, joint_positions,
        joint_velocities, torques);

    // Compute expected joint positions
    double l1 = quad_kd.getLinkLength(0, 2);
    Eigen::VectorXd joint_positions_expected(12), joint_velocities_expected(12);
    joint_positions_expected << 0, asin(0.5 * h / l1), 2 * asin(0.5 * h / l1),
        0, asin(0.5 * h / l1), 2 * asin(0.5 * h / l1), 0, asin(0.5 * h / l1),
        2 * asin(0.5 * h / l1), 0, asin(0.5 * h / l1), 2 * asin(0.5 * h / l1);

    // Compute expected joint velocities
    double hip_vel_expected =
        -0.5 * foot_vel_z / (l1 * cos(joint_positions_expected[4]));
    double knee_vel_expected = 2 * hip_vel_expected;
    joint_velocities_expected << 0, 0, 0, 0, hip_vel_expected,
        knee_vel_expected, 0, hip_vel_expected, knee_vel_expected, 0, 0, 0;

    // Check joint positions and velocities match
    EXPECT_TRUE(is_exact);
    EXPECT_TRUE(joint_positions.isApprox(joint_positions_expected));
    EXPECT_TRUE(joint_velocities.isApprox(joint_velocities_expected));

    // Check validity
    bool is_state_valid = quad_kd.isValidCentroidalState(
        body_state, foot_positions, foot_velocities, grfs, map, joint_positions,
        joint_velocities, torques, state_violation, control_violation);
    EXPECT_TRUE(is_state_valid);

    body_state[2] += 0.5;
    is_exact = quad_kd.convertCentroidalToFullBody(
        body_state, foot_positions, foot_velocities, grfs, joint_positions,
        joint_velocities, torques);
    EXPECT_FALSE(is_exact);

    // Check validity
    is_state_valid = quad_kd.isValidCentroidalState(
        body_state, foot_positions, foot_velocities, grfs, map, joint_positions,
        joint_velocities, torques, state_violation, control_violation);
    EXPECT_FALSE(is_state_valid);
  }
```

```
  // Print results if desired
  // std::cout << "joint_positions\n" << joint_positions << std::endl;
  // std::cout << "joint_velocities\n" << joint_velocities << std::endl;
  // std::cout << "torques\n" << torques << std::endl;

  // Check timing characteristics
  int N = 1000;
  int count = 0;
  auto t_start = std::chrono::steady_clock::now();
  for (int i = 0; i < N; i++) {
    count++;
    quad_kd.convertCentroidalToFullBody(body_state, foot_positions,
                                        foot_velocities, grfs, joint_positions,
                                        joint_velocities, torques);
  }
  auto t_end = std::chrono::steady_clock::now();

  std::chrono::duration<double> t_diff =
      std::chrono::duration_cast<std::chrono::duration<double>>(t_end -
                                                                t_start);
  double average_time = t_diff.count() / count;

  std::cout << "Average convertCentroidalToFullBody time = " << average_time
            << " s" << std::endl;

  EXPECT_TRUE(average_time < 1e-4);
}
```