```
/*
 *  Copyright (C) 2010, CCNY Robotics Lab
 *  Ivan Dryanovski <ivan.dryanovski@gmail.com>
 *
 *  http://robotics.ccny.cuny.edu
 *
 *  Based on implementation of Madgwick's IMU and AHRS algorithms.
 *  http://www.x-io.co.uk/node/8#open_source_ahrs_and_imu_algorithms
 *
 *
 *  This program is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <cmath>
#include "imu_filter_madgwick/imu_filter.h"

// Fast inverse square-root
// See: http://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Reciprocal_of_the_sq
uare_root
static float invSqrt(float x)
{
  float xhalf = 0.5f * x;
  union
  {
    float x;
    int i;
  } u;
  u.x = x;
  u.i = 0x5f3759df - (u.i >> 1);
  /* The next line can be repeated any number of times to increase accuracy */
  u.x = u.x * (1.5f - xhalf * u.x * u.x);
  return u.x;
}

template<typename T>
static inline void normalizeVector(T& vx, T& vy, T& vz)
{
  T recipNorm = invSqrt (vx * vx + vy * vy + vz * vz);
  vx *= recipNorm;
  vy *= recipNorm;
  vz *= recipNorm;
}

template<typename T>
static inline void normalizeQuaternion(T& q0, T& q1, T& q2, T& q3)
{
  T recipNorm = invSqrt (q0 * q0 + q1 * q1 + q2 * q2 + q3 * q3);
  q0 *= recipNorm;
  q1 *= recipNorm;
  q2 *= recipNorm;
  q3 *= recipNorm;
}

static inline void rotateAndScaleVector(
    float q0, float q1, float q2, float q3,
    float _2dx, float _2dy, float _2dz,
    float& rx, float& ry, float& rz) {
```

```
  // result is half as long as input
  rx = _2dx * (0.5f - q2 * q2 - q3 * q3)
     + _2dy * (q0 * q3 + q1 * q2)
     + _2dz * (q1 * q3 - q0 * q2);
  ry = _2dx * (q1 * q2 - q0 * q3)
     + _2dy * (0.5f - q1 * q1 - q3 * q3)
     + _2dz * (q0 * q1 + q2 * q3);
  rz = _2dx * (q0 * q2 + q1 * q3)
     + _2dy * (q2 * q3 - q0 * q1)
     + _2dz * (0.5f - q1 * q1 - q2 * q2);
}


static inline void compensateGyroDrift(
    float q0, float q1, float q2, float q3,
    float s0, float s1, float s2, float s3,
    float dt, float zeta,
    float& w_bx, float& w_by, float& w_bz,
    float& gx, float& gy, float& gz)
{
  // w_err = 2 q x s
  float w_err_x = 2.0f * q0 * s1 - 2.0f * q1 * s0 - 2.0f * q2 * s3 + 2.0f * q3 * s2;
  float w_err_y = 2.0f * q0 * s2 + 2.0f * q1 * s3 - 2.0f * q2 * s0 - 2.0f * q3 * s1;
  float w_err_z = 2.0f * q0 * s3 - 2.0f * q1 * s2 + 2.0f * q2 * s1 - 2.0f * q3 * s0;

  w_bx += w_err_x * dt * zeta;
  w_by += w_err_y * dt * zeta;
  w_bz += w_err_z * dt * zeta;

  gx -= w_bx;
  gy -= w_by;
  gz -= w_bz;
}

static inline void orientationChangeFromGyro(
    float q0, float q1, float q2, float q3,
    float gx, float gy, float gz,
    float& qDot1, float& qDot2, float& qDot3, float& qDot4)
{
  // Rate of change of quaternion from gyroscope
  // See EQ 12
  qDot1 = 0.5f * (-q1 * gx - q2 * gy - q3 * gz);
  qDot2 = 0.5f * (q0 * gx + q2 * gz - q3 * gy);
  qDot3 = 0.5f * (q0 * gy - q1 * gz + q3 * gx);
  qDot4 = 0.5f * (q0 * gz + q1 * gy - q2 * gx);
}

static inline void addGradientDescentStep(
    float q0, float q1, float q2, float q3,
    float _2dx, float _2dy, float _2dz,
    float mx, float my, float mz,
    float& s0, float& s1, float& s2, float& s3)
{
  float f0, f1, f2;

  // Gradient decent algorithm corrective step
  // EQ 15, 21
  rotateAndScaleVector(q0,q1,q2,q3, _2dx, _2dy, _2dz, f0, f1, f2);

  f0 -= mx;
  f1 -= my;
  f2 -= mz;


  // EQ 22, 34
  // Jt * f
  s0 += (_2dy * q3 - _2dz * q2) * f0
```

```
      + (-_2dx * q3 + _2dz * q1) * f1
      + (_2dx * q2 - _2dy * q1) * f2;
  s1 += (_2dy * q2 + _2dz * q3) * f0
      + (_2dx * q2 - 2.0f * _2dy * q1 + _2dz * q0) * f1
      + (_2dx * q3 - _2dy * q0 - 2.0f * _2dz * q1) * f2;
  s2 += (-2.0f * _2dx * q2 + _2dy * q1 - _2dz * q0) * f0
      + (_2dx * q1 + _2dz * q3) * f1
      + (_2dx * q0 + _2dy * q3 - 2.0f * _2dz * q2) * f2;
  s3 += (-2.0f * _2dx * q3 + _2dy * q0 + _2dz * q1) * f0
      + (-_2dx * q0 - 2.0f * _2dy * q3 + _2dz * q2) * f1
      + (_2dx * q1 + _2dy * q2) * f2;
}

static inline void compensateMagneticDistortion(
    float q0, float q1, float q2, float q3,
    float mx, float my, float mz,
    float& _2bxy, float& _2bz)
{
  float hx, hy, hz;
  // Reference direction of Earth’s magnetic field (See EQ 46)
  rotateAndScaleVector(q0, -q1, -q2, -q3, mx, my, mz, hx, hy, hz);

  _2bxy = 4.0f * sqrt (hx * hx + hy * hy);
  _2bz = 4.0f * hz;

}


ImuFilter::ImuFilter() :
    q0(1.0), q1(0.0), q2(0.0), q3(0.0),
    w_bx_(0.0), w_by_(0.0), w_bz_(0.0),
    zeta_ (0.0), gain_ (0.0), world_frame_(WorldFrame::ENU)
{
}

ImuFilter::~ImuFilter()
{
}

void ImuFilter::madgwickAHRSupdate(
    float gx, float gy, float gz,
    float ax, float ay, float az,
    float mx, float my, float mz,
    float dt)
{
  float s0, s1, s2, s3;
  float qDot1, qDot2, qDot3, qDot4;
  float _2bz, _2bxy;

  // Use IMU algorithm if magnetometer measurement invalid (avoids NaN in magnetometer norm
alisation)
  if (!std::isfinite(mx) || !std::isfinite(my) || !std::isfinite(mz))
  {
    madgwickAHRSupdateIMU(gx, gy, gz, ax, ay, az, dt);
    return;
  }

  // Compute feedback only if accelerometer measurement valid (avoids NaN in accelerometer
normalisation)
  if (!((ax == 0.0f) && (ay == 0.0f) && (az == 0.0f)))
  {
    // Normalise accelerometer measurement
    normalizeVector(ax, ay, az);

    // Normalise magnetometer measurement
    normalizeVector(mx, my, mz);

    // Compensate for magnetic distortion
```

```
    compensateMagneticDistortion(q0, q1, q2, q3, mx, my, mz, _2bxy, _2bz);

    // Gradient decent algorithm corrective step
    s0 = 0.0;  s1 = 0.0;  s2 = 0.0;  s3 = 0.0;
    switch (world_frame_) {
      case WorldFrame::NED:
        // Gravity: [0, 0, -1]
        addGradientDescentStep(q0, q1, q2, q3, 0.0, 0.0, -2.0, ax, ay, az, s0, s1, s2, s3);

        // Earth magnetic field: = [bxy, 0, bz]
        addGradientDescentStep(q0,q1,q2,q3, _2bxy, 0.0, _2bz, mx, my, mz, s0, s1, s2, s3);
        break;
      case WorldFrame::NWU:
        // Gravity: [0, 0, 1]
        addGradientDescentStep(q0, q1, q2, q3, 0.0, 0.0, 2.0, ax, ay, az, s0, s1, s2, s3);

        // Earth magnetic field: = [bxy, 0, bz]
        addGradientDescentStep(q0,q1,q2,q3, _2bxy, 0.0, _2bz, mx, my, mz, s0, s1, s2, s3);
        break;
      default:
      case WorldFrame::ENU:
        // Gravity: [0, 0, 1]
        addGradientDescentStep(q0, q1, q2, q3, 0.0, 0.0, 2.0, ax, ay, az, s0, s1, s2, s3);

        // Earth magnetic field: = [0, bxy, bz]
        addGradientDescentStep(q0, q1, q2, q3, 0.0, _2bxy, _2bz, mx, my, mz, s0, s1, s2, s3
);
        break;
    }
    normalizeQuaternion(s0, s1, s2, s3);

    // compute gyro drift bias
    compensateGyroDrift(q0, q1, q2, q3, s0, s1, s2, s3, dt, zeta_, w_bx_, w_by_, w_bz_, gx,
 gy, gz);

    orientationChangeFromGyro(q0, q1, q2, q3, gx, gy, gz, qDot1, qDot2, qDot3, qDot4);

    // Apply feedback step
    qDot1 -= gain_ * s0;
    qDot2 -= gain_ * s1;
    qDot3 -= gain_ * s2;
    qDot4 -= gain_ * s3;
  }
  else
  {
    orientationChangeFromGyro(q0, q1, q2, q3, gx, gy, gz, qDot1, qDot2, qDot3, qDot4);
  }

  // Integrate rate of change of quaternion to yield quaternion
  q0 += qDot1 * dt;
  q1 += qDot2 * dt;
  q2 += qDot3 * dt;
  q3 += qDot4 * dt;

  // Normalise quaternion
  normalizeQuaternion(q0, q1, q2, q3);
}

void ImuFilter::madgwickAHRSupdateIMU(
    float gx, float gy, float gz,
    float ax, float ay, float az,
    float dt)
{
  float recipNorm;
  float s0, s1, s2, s3;
  float qDot1, qDot2, qDot3, qDot4;

  // Rate of change of quaternion from gyroscope
```

```
  orientationChangeFromGyro (q0, q1, q2, q3, gx, gy, gz, qDot1, qDot2, qDot3, qDot4);

  // Compute feedback only if accelerometer measurement valid (avoids NaN in accelerometer
normalisation)
  if (!((ax == 0.0f) && (ay == 0.0f) && (az == 0.0f)))
  {
    // Normalise accelerometer measurement
    normalizeVector(ax, ay, az);

    // Gradient decent algorithm corrective step
    s0 = 0.0;  s1 = 0.0;  s2 = 0.0;  s3 = 0.0;
    switch (world_frame_) {
      case WorldFrame::NED:
        // Gravity: [0, 0, -1]
        addGradientDescentStep(q0, q1, q2, q3, 0.0, 0.0, -2.0, ax, ay, az, s0, s1, s2, s3);
        break;
      case WorldFrame::NWU:
        // Gravity: [0, 0, 1]
        addGradientDescentStep(q0, q1, q2, q3, 0.0, 0.0, 2.0, ax, ay, az, s0, s1, s2, s3);
        break;
      default:
      case WorldFrame::ENU:
        // Gravity: [0, 0, 1]
        addGradientDescentStep(q0, q1, q2, q3, 0.0, 0.0, 2.0, ax, ay, az, s0, s1, s2, s3);
        break;
    }

    normalizeQuaternion(s0, s1, s2, s3);

    // Apply feedback step
    qDot1 -= gain_ * s0;
    qDot2 -= gain_ * s1;
    qDot3 -= gain_ * s2;
    qDot4 -= gain_ * s3;
  }

  // Integrate rate of change of quaternion to yield quaternion
  q0 += qDot1 * dt;
  q1 += qDot2 * dt;
  q2 += qDot3 * dt;
  q3 += qDot4 * dt;

  // Normalise quaternion
  normalizeQuaternion (q0, q1, q2, q3);
}


void ImuFilter::getGravity(float& rx, float& ry, float& rz,
    float gravity)
{
    // Estimate gravity vector from current orientation
    switch (world_frame_) {
      case WorldFrame::NED:
        // Gravity: [0, 0, -1]
        rotateAndScaleVector(q0, q1, q2, q3,
            0.0, 0.0, -2.0*gravity,
            rx, ry, rz);
        break;
      case WorldFrame::NWU:
        // Gravity: [0, 0, 1]
        rotateAndScaleVector(q0, q1, q2, q3,
            0.0, 0.0, 2.0*gravity,
            rx, ry, rz);
        break;
      default:
      case WorldFrame::ENU:
        // Gravity: [0, 0, 1]
        rotateAndScaleVector(q0, q1, q2, q3,
```

```
            0.0, 0.0, 2.0*gravity,
            rx, ry, rz);
        break;
    }
}
```