付録

使用ソースコードを以下に記載する. 実行環境は以下の通りである.

使用言語:	$C++(C++11)$
実行環境:	Mac OS Sierra (Ver 10.12.6)
コンパイラ:	$g++(GCC)$ 5.4.0
コンパイラオプション:	$g++-Wall-g-ggdb-std=c++11$

Table 1: 実行環境

なお, プロットは python を用いて matplotlib で行ったが, プロットを行っているだけなので, 省略する. また関数の中でファイル作成を行っているが, ディレクトリ構成は以下の通りである.

```
sc_controlsystem
├── datafile (Output files produced here)
├── report1
│   ├── Makefile
│   └── report1 (execution file)
├── report2
│   ├── Makefile
│   └── report2 (execution file)
└── src
    ├── Eigen
    ├── unsupported
    ├── kalman_filter.cpp
    ├── kalman_filter.hpp
    ├── non_linear.cpp
    ├── non_linear.hpp
    ├── report1.cpp
    └── report2.cpp
```

Listing 1: report1.cpp (report1 用 main)

```cpp
#include "Eigen/Core"
#include "Eigen/Geometry"
#include "non_linear.hpp"
#include <iostream>

int main() {
  Eigen::Vector3d I;
  I.x() = 1.9; // I_x
  I.y() = 1.6; // I_y
  I.z() = 2.0; // I_z

  Eigen::Vector3d omega_init; //ノミナル角速度
  omega_init.x() = 0.1;
  omega_init.y() = 17 * 0.1047 + 0.1;
  omega_init.z() = 0;

  Eigen::Vector4d qt_init; //初期Quartanion
  qt_init << 1.0, 0, 0, 0;

  // Initialize State
  Eigen::VectorXd omega_qt;
  Eigen::Matrix<double, 3, 3> dcm;
  omega_qt = Eigen::VectorXd::Zero(7);

  omega_qt << qt_init[0], qt_init[1], qt_init[2], qt_init[3], omega_init[0],
      omega_init[1], omega_init[2];

  double SD_Q = 0.0; //外乱トルクの標準偏差
  double SD_R = 0.0; //観測ノイズの標準偏差
  double SEED =
      1; //外乱の乱数生成メルセンヌツイスタ用シード,0の時シードもランダムに生成される

  NonLinear nol(I, SD_Q, SD_R, SEED);

  //ルンゲクッタ用パラメータ
  double dt = 0.01;
  double T_END = 100;

  int counter = 0; // print用カウンター

  for (double t = 0; t < T_END; t += dt) {
    if (!(counter % 100)) {
      std::cout << "time:" << t << std::endl;
    }
    nol.update_vW(); // make disterbuance noise
    nol.propagation(dt, omega_qt, dcm, t, t + dt);
    nol.writefile(t, omega_qt, dcm, "R1");
    counter += 1;
  }
}
```

Listing 2: report2.cpp (report2 用 main)

```
#include "Eigen/Core"
#include "Eigen/Geometry"
#include "iostream"
#include "kalman_filter.hpp"
#include "non_linear.hpp"

int main() {

  // setup for system model

  // settings for satelite attitude
  Eigen::Vector3d I;
  I.x() = 1.9; // I_x
  I.y() = 1.6; // I_y
  I.z() = 2.0; // I_z

  Eigen::Vector3d omega_init; //初期角速度
  Eigen::Vector3d omega_init_error;
  omega_init << 0.1, 17 * 0.1047 + 0.1, 0;
  omega_init_error << 0.01, 0.01, 0.01;

  Eigen::Vector4d qt_init; //初期Quartanion
  Eigen::Vector4d qt_init_errror;
  qt_init << 1.0, 0, 0, 0;
  qt_init_errror << -0.01, 0.08, 0.08, 0.08;

  // initial state
  Eigen::VectorXd vX0_Tru;
  Eigen::VectorXd vX0_Est;
  vX0_Tru = Eigen::VectorXd::Zero(7);
  vX0_Est = Eigen::VectorXd::Zero(7);

  vX0_Tru << qt_init[0], qt_init[1], qt_init[2], qt_init[3], omega_init[0],
      omega_init[1], omega_init[2];
  omega_init += omega_init_error;
  qt_init += qt_init_errror;
  vX0_Est << qt_init[0], qt_init[1], qt_init[2], qt_init[3], omega_init[0],
      omega_init[1], omega_init[2];

  double SD_Q = 0.01; //外乱トルクの標準偏差
  double SD_R = 0.01; //観測ノイズの標準偏差
  double SEED =
      0; //外乱の乱数生成メルセンヌツイスタ用シード,0の時シードもランダムに生成される

  NonLinear nol(I, SD_Q, SD_R, SEED);

  // System Parameters
  int dimI = 3;
  int dimO = 3;
  int dimS = 7;

  // Define Initial Vectors and Matrix
  Eigen::VectorXd vz;
  Eigen::MatrixXd Q;
  Eigen::MatrixXd R;
  Eigen::MatrixXd P0;

  // init Vectors and Matrix
  vz = Eigen::VectorXd::Zero(dimO);
  Q = Eigen::MatrixXd::Identity(dimI, dimI);
  R = Eigen::MatrixXd::Identity(dimO, dimO);
  P0 = Eigen::MatrixXd::Identity(dimS, dimS);

  // Q process noise
  Q(0, 0) = SD_Q * SD_Q;
  Q(1, 1) = SD_Q * SD_Q;
  Q(2, 2) = SD_Q * SD_Q;
  // R measurement noise
  R(0, 0) = SD_R * SD_R;
  R(1, 1) = SD_R * SD_R;
  R(2, 2) = SD_R * SD_R;
  // P0
  for (int i = 0; i < dimS; i++) {
    P0(i, i) = SD_R;
  }
```

```

  // simulation用パラメータ
  double dt = 0.01;
  double T_END = 100;
  double obs_interval = 1.0;

  // end model setup

  // make model
  KalmanFilter KFD(dimI, dimO, dimS, vX0_Tru, vX0_Est, Q, R, P0, dt);

  // simulate
  KFD.simulation(nol, T_END, obs_interval);
}
```

Listing 3: non_linear.hpp (衛星シミュレーション用クラス (header))

```
// attitude for spinning satelite
#ifndef NONLINEAR_H
#define NONLINEAR_H

#include "Eigen/Core"
#include "Eigen/Geometry"
#include <cstdio>
#include <string>

class NonLinear {
public:
  NonLinear(Eigen::Vector3d &I, double SD_Q, double SD_R, int SEED) {
    Ix_ = I[0];
    Iy_ = I[1];
    Iz_ = I[2];
    SD_Q_ = SD_Q;
    SD_R_ = SD_R;
    SEED_ = SEED;
  };

  double Ix_, Iy_, Iz_;
  Eigen::Vector3d vV_; //観測ノイズ
  Eigen::Vector3d vW_; //外乱ノイズ

  //  functions for state propagation
  void propagation(double dt, Eigen::VectorXd &omega_qt,
                   Eigen::Matrix<double, 3, 3> &dcm, double t_start,
                   double T_END);
  void cal_omega_qt_dot(Eigen::VectorXd &omega_qt,
                        Eigen::VectorXd &omega_qt_dot);
  void quartanion_to_dcm(Eigen::VectorXd &omega_qt,
                         Eigen::Matrix<double, 3, 3> &dcm_);

  // functions for producing/ deleting noise
  void update_vW(); // disterbuance noise
  void update_vV(); // observation noise
  void del_noise();

  // writing result in files
  void writefile(double t, Eigen::VectorXd &omega_qt,
                 Eigen::Matrix<double, 3, 3> &dcm, std::string fileprefix);

private:
  double SD_Q_, SD_R_; /*ノイズ標準偏差 SD_Q_:外乱トルク SD_R:観測ノイズ*/
  double SEED_; /*メルセンヌツイスタ関数用シード 0-シードも乱数で決定*/
};

#endif
```

Listing 4: non_linear.cpp (衛星シミュレーション用クラス (source))

```
#include "non_linear.hpp"
#include <fstream>
#include <iomanip>
#include <iostream>
#include <random>
#include <string>

/*********************************************************
    state propgation function
      propagation()
      cal_omega_qt_dot()
      quartanion_to_dcm()
**********************************************************/

/* propagation
   @Input
     dt: 時間ステップ
     omega_qt: 伝搬するstate(角速度ω, qt)
     dcm: qtに対応するdcm
     t_start: 積分開始時間
     t_end: 積分終了時間
*/
void NonLinear::propagation(double dt, Eigen::VectorXd &omega_qt,
                            Eigen::Matrix<double, 3, 3> &dcm, double t_start,
                            double T_END) {
  for (double t = t_start; t < T_END; t += dt) { // update
    Eigen::VectorXd k1, k2, k3, k4;
    Eigen::VectorXd omega_qt_2, omega_qt_3, omega_qt_4;
    k1 = Eigen::VectorXd::Zero(7);
    k2 = Eigen::VectorXd::Zero(7);
    k3 = Eigen::VectorXd::Zero(7);
    k4 = Eigen::VectorXd::Zero(7);
    omega_qt_2 = Eigen::VectorXd::Zero(7);
    omega_qt_3 = Eigen::VectorXd::Zero(7);
    omega_qt_4 = Eigen::VectorXd::Zero(7);
    cal_omega_qt_dot(omega_qt, k1);

    omega_qt_2 = omega_qt + dt / 2 * k1;
    cal_omega_qt_dot(omega_qt_2, k2);

    omega_qt_3 = omega_qt + dt / 2 * k2;
    cal_omega_qt_dot(omega_qt_3, k3);

    omega_qt_4 = omega_qt + dt * k3;
    cal_omega_qt_dot(omega_qt_4, k4);

    omega_qt = omega_qt + dt / 6 * (k1 + 2 * k2 + 2 * k3 + k4);
    quartanion_to_dcm(omega_qt, dcm);
  }
}

// ω, qtの微分を求める
void NonLinear::cal_omega_qt_dot(Eigen::VectorXd &omega_qt,
                                 Eigen::VectorXd &omega_qt_dot) {
  double q0 = omega_qt(0, 0);
  double q1 = omega_qt(1, 0);
  double q2 = omega_qt(2, 0);
  double q3 = omega_qt(3, 0);
  double ox = omega_qt(4, 0);
  double oy = omega_qt(5, 0);
  double oz = omega_qt(6, 0);

  omega_qt_dot(0, 0) = 0.5 * (-q1 * ox + -q2 * oy + -q3 * oz);
  omega_qt_dot(1, 0) = 0.5 * (q0 * ox + -q3 * oy + q2 * oz);
  omega_qt_dot(2, 0) = 0.5 * (q3 * ox + q0 * oy + -q1 * oz);
  omega_qt_dot(3, 0) = 0.5 * (-q2 * ox + q1 * oy + q0 * oz);
  omega_qt_dot(4, 0) = (Iy_ - Iz_) / Ix_ * oy * oz + vW_[0] / Ix_;
  omega_qt_dot(5, 0) = (Iz_ - Ix_) / Iy_ * oz * ox + vW_[1] / Iy_;
  omega_qt_dot(6, 0) = (Ix_ - Iy_) / Iz_ * ox * oy + vW_[2] / Iz_;
}

// qt->dcm
void NonLinear::quartanion_to_dcm(Eigen::VectorXd &omega_qt,
                                  Eigen::Matrix<double, 3, 3> &dcm) {
  double q0 = omega_qt(0, 0);
```

```
  double q1 = omega_qt(1, 0);
  double q2 = omega_qt(2, 0);
  double q3 = omega_qt(3, 0);
  // first column
  dcm(0, 0) = q0 * q0 + q1 * q1 - q2 * q2 - q3 * q3;
  dcm(1, 0) = 2 * (q1 * q3 + q0 * q3);
  dcm(2, 0) = 2 * (q1 * q3 - q0 * q2);
  // second column
  dcm(0, 1) = 2 * (q1 * q2 - q0 * q3);
  dcm(1, 1) = q0 * q0 - q1 * q1 + q2 * q2 - q3 * q3;
  dcm(2, 1) = 2 * (q2 * q3 + q0 * q1);
  // third column
  dcm(0, 2) = 2 * (q1 * q3 + q0 * q2);
  dcm(1, 2) = 2 * (q2 * q3 - q0 * q1);
  dcm(2, 2) = q0 * q0 - q1 * q1 - q2 * q2 + q3 * q3;
}

/*********************************************************
    noise producing function
      update_vW() : 外乱ノイズ
      update_vV() : 観測ノイズ
      del_noise() : ノイズ消去
**********************************************************/

// 外乱ノイズ生成関数 seed=0の場合, seedもランダム生成
void NonLinear::update_vW() {
  if (SEED_ == 0) {
    std::random_device seed_gen;
    std::mt19937 randMt(seed_gen());
    static std::normal_distribution<> normW(0.0, SD_Q_);
    vW_(0) = normW(randMt);
    vW_(1) = normW(randMt);
    vW_(2) = normW(randMt);
  } else {
    std::mt19937 randMt(SEED_); //メルセンヌツイスタ乱数
    static std::normal_distribution<> normW(0.0, SD_Q_);
    vW_(0) = normW(randMt);
    vW_(1) = normW(randMt);
    vW_(2) = normW(randMt);
  }
}

// 観測ノイズ生成関数, seed=0の時はseedもランダム生成
void NonLinear::update_vV() {
  if (SEED_ == 0) {
    std::random_device seed_gen;
    std::mt19937 randMt(seed_gen());
    static std::normal_distribution<> normW(0.0, SD_Q_);
    vV_(0) = normW(randMt);
    vV_(1) = normW(randMt);
    vV_(2) = normW(randMt);
  } else {
    std::mt19937 randMt(SEED_); //メルセンヌツイスタ乱数
    static std::normal_distribution<> normW(0.0, SD_Q_);
    vV_(0) = normW(randMt);
    vV_(1) = normW(randMt);
    vV_(2) = normW(randMt);
  }
}

// システムノイズ/観測ノイズ消去
void NonLinear::del_noise() {
  vV_(0) = 0;
  vV_(1) = 0;
  vV_(2) = 0;
  vW_(0) = 0;
  vW_(1) = 0;
  vW_(2) = 0;
}

/*********************************************************
    Writing quartanion,omega,dcm data
      writefile()
*********************************************************/

/* @input
     t: 時間
```

```
     omega_qt,dcm: state
     fileprefix: ファイル先頭につける文字
*/

void NonLinear::writefile(double t, Eigen::VectorXd &omega_qt,
                          Eigen::Matrix<double, 3, 3> &dcm,
                          std::string fileprefix) {
  // write data of quartanion
  std::fstream fout_q;
  std::string filename_q;
  if (SD_Q_ > 0) {
    filename_q = "../datafile/" + fileprefix + "qt_error.txt";
  } else {
    filename_q = "../datafile/" + fileprefix + "qt.txt";
  }
  if (t == 0)
    fout_q.open(filename_q, std::ios::out);
  else
    fout_q.open(filename_q, std::ios::app);
  fout_q << t << " ";
  for (int n = 0; n < 4; n++) {
    fout_q << omega_qt(n, 0) << " ";
    if (n == 3) {
      fout_q << std::endl;
    }
  }
  // write data of omega
  std::fstream fout_o;
  std::string filename_o;
  if (SD_Q_ > 0) {
    filename_o = "../datafile/" + fileprefix + "omega_error.txt";
  } else {
    filename_o = "../datafile/" + fileprefix + "omega.txt";
  }
  if (t == 0)
    fout_o.open(filename_o, std::ios::out);
  else
    fout_o.open(filename_o, std::ios::app);
  fout_o << t << " ";
  for (int m = 0; m < 3; m++) {
    fout_o << omega_qt(m + 4, 0) << " ";
    if (m == 2) {
      fout_o << std::endl;
    }
  }
  // write data for dcm
  for (int m = 0; m < 3; m++) {
    std::fstream fout_d;
    std::string filename_d;
    if (SD_Q_ > 0) {
      filename_d = "../datafile/" + fileprefix + "dcm_column" +
                   std::to_string(m) + "_error" + ".txt";
    } else {
      filename_d = "../datafile/" + fileprefix + "dcm_column" +
                   std::to_string(m) + ".txt";
    }
    if (t == 0)
      fout_d.open(filename_d, std::ios::out);
    else
      fout_d.open(filename_d, std::ios::app);
    fout_d << t << " " << dcm(0, m) << " " << dcm(1, m) << " " << dcm(2, m)
           << std::endl;
  }
}
```

Listing 5: kalman_filter.cpp (カルマンフィルタ用クラス (header))

```
/*******************************************************
 * Descreted Kalman filter Program
 *
 * < Linearlized Equation of Stateo>
 *    Δx' = mA * Δx  + mB * (u+vW)
 *    y = mH * Δx + vV
 *
 * <Descreted Linearlized Equation of State>
 *    Δx[k]' = mPhi * Δx[k-1] + mGmm * vW[k-1]
 *
 * <Matrices>
 *    mA - System dynamics matrix(State Matrix) (SxS)
 *    mB - input matrix (SxI)
 *    mH - Output matrix (OxS)
 *    mQ - Process noise covariance (IxI)
 *    mR - Measurement noise covariance (OxO)
 *    mP - Estimate error covariance (SxS)
 *    mK - Kalman Gain
 *
 * <Vectors>
 *    vSysTru - True State Vector
 *    vSysEst - State Vector for Estimated Models
 *    vW - disturbance noise(defined in NonLinear Class)
 *    vV - Observation noise(defined in NonLinear Class)
 *
 *  Non-Linear equation of state is written in class NonLinear.
 *  For propagation of state vectors in both True State and
 *  Estimated Model,propagation function in class NonLinear
 *  is used. In addition,system parameters in class NonLinear
 *  might be used to update A,B,H matrices.
 *
 **************************************************************/

#ifndef KFD_H
#define KFD_H
#include "Eigen/Core"
#include "Eigen/Geometry"
#include "non_linear.hpp"
#include <cstdio>
#include <iostream>

class KalmanFilter {

public:
  KalmanFilter(int dimI, int dimO, int dimS, const Eigen::VectorXd &vX0_Tru,
               const Eigen::VectorXd &vX0_EST, const Eigen::MatrixXd &Q,
               const Eigen::MatrixXd &R, const Eigen::MatrixXd &P0,
               const double dt)
      : dimI(dimI), dimO(dimO), dimS(dimS), vSysTru(vX0_Tru), vSysEst(vX0_EST),
        mQ(Q), mR(R), mP(P0), dt(dt) {
    // init Vectors and Matrix
    mA = Eigen::MatrixXd::Zero(dimS, dimS);
    mB = Eigen::MatrixXd::Zero(dimS, dimI);
    mH = Eigen::MatrixXd::Zero(dimO, dimS);
    vz = Eigen::VectorXd::Zero(dimO);

    // init
    mM = mP;
    mGmm = Eigen::MatrixXd::Zero(dimS, dimI);
    mK = Eigen::MatrixXd::Zero(dimS, dimO);
    // display Initial Matrix
    std::cout << "A" << std::endl << mA << std ::endl;
    std::cout << "B" << std::endl << mB << std ::endl;
    std::cout << "H" << std::endl << mH << std ::endl;
    std::cout << "Q" << std::endl << mQ << std ::endl;
    std::cout << "R" << std::endl << mR << std ::endl;
    std::cout << "P" << std::endl << mP << std ::endl;
    std::cout << "M" << std::endl << mM << std ::endl;
    std::cout << "K" << std::endl << mK << std ::endl;
    std::cout << "H" << std::endl << mH << std ::endl;
    std::cout << "Phi" << std::endl << mPhi << std ::endl;
    std::cout << "Gmm" << std::endl << mGmm << std ::endl;
  };

  // System Parameters (class SpinSat)
```

```
  // System Matrices
  int dimI, dimO, dimS; // Number of Input,Output,State
  Eigen::MatrixXd mA;   // State Matrix
  Eigen::MatrixXd mB;   // Input Matrix
  Eigen::MatrixXd mH;   // Output Matrix
  // Matrices for Descreted State Equation
  Eigen::MatrixXd mPhi, mGmm;
  // Vectors for states(vSys = x)
  Eigen::VectorXd vSysTru, vSysEst;
  Eigen::VectorXd vz; // observed Y - estimated Y
  // Matrices for errors
  Eigen::MatrixXd mQ, mR;
  Eigen::MatrixXd mP, mM, mK;
  // Discrete time step
  double dt;

  //@ method

  /* Update functions (need to rewrite when system is changed)*/
  void UpdateMatrixA(NonLinear &nol);
  void UpdateMatrixB(NonLinear &nol);
  void UpdateMatrixH(int col);
  // Calculate Phi,Gmm using Matrix A and Matrix B
  void UpdateMatrixPhi();
  void UpdateMatrixGmm();
  // Calculating State Vector Norms
  double CalcQuaternionNorm(Eigen::VectorXd &vX);
  void NormalizeQuaternion(Eigen::VectorXd &vX);
  double CalcErrNorm(Eigen::VectorXd &vSysTru, Eigen::VectorXd &vSysEst);

  // select column of DCM observed
  int select_column(int seed);

  /* Whole Update Sequence(need to rewrite when system is changed)*/
  void simulation(NonLinear &nol, double T_END, double obs_interval);

  // write files
  void write_file_pe(double t);
  void write_file_k(double t);
};

#endif
```

Listing 6: kalman_filter.cpp (カルマンフィルタ用クラス (source))

```
#include "Eigen/LU"
#include "unsupported/Eigen/MatrixFunctions"
#include <fstream>
#include <iomanip>
#include <iostream>
#include <random>
#include <stdexcept>

#include "kalman_filter.hpp"

/*******************************************************
    Update Functions for A,B,H
      void UpdateMatrixA(NonLinear &nol)
      void UpdateMatrixB(NonLinear &nol)
      void UpdateMatrixH(int col)
*******************************************************/

void KalmanFilter::UpdateMatrixA(NonLinear &nol) {
  double Ix = nol.Ix_;
  double Iy = nol.Iy_;
  double Iz = nol.Iz_;
  double q0 = vSysEst(0);
  double q1 = vSysEst(1);
  double q2 = vSysEst(2);
  double q3 = vSysEst(3);
  double wx = vSysEst(4);
  double wy = vSysEst(5);
  double wz = vSysEst(6);
  mA(0, 0) = 0.0;       mA(0, 1) = -0.5 * wx; mA(0, 2) = -0.5 * wy; mA(0, 3) = -0.5 * wz;
  mA(1, 0) = +0.5 * wx; mA(1, 1) = 0.0;       mA(1, 2) = +0.5 * wz; mA(1, 3) = -0.5 * wy;
```

```
  mA(2, 0) = +0.5 * wy; mA(2, 1) = -0.5 * wz; mA(2, 2) = 0.0;        mA(2, 3) = +0.5 * wx;
  mA(3, 0) = +0.5 * wz; mA(3, 1) = +0.5 * wy; mA(3, 2) = -0.5 * wx; mA(3, 3) = 0.0;

  mA(4, 0) = 0.0; mA(4, 1) = 0.0; mA(4, 2) = 0.0; mA(4, 3) = 0.0;
  mA(5, 0) = 0.0; mA(5, 1) = 0.0; mA(5, 2) = 0.0; mA(5, 3) = 0.0;
  mA(6, 0) = 0.0; mA(6, 1) = 0.0; mA(6, 2) = 0.0; mA(6, 3) = 0.0;

  mA(0, 4) = -0.5 * q1; mA(0, 5) = -0.5 * q2; mA(0, 6) = -0.5 * q3;
  mA(1, 4) = +0.5 * q0; mA(1, 5) = -0.5 * q3; mA(1, 6) = +0.5 * q2;
  mA(2, 4) = +0.5 * q3; mA(2, 5) = +0.5 * q0; mA(2, 6) = -0.5 * q1;
  mA(3, 4) = -0.5 * q2; mA(3, 5) = +0.5 * q1; mA(3, 6) = +0.5 * q0;

  mA(4, 4) = 0.0; mA(4, 5) = wz * (Iy - Iz) / Ix; mA(4, 6) = wy * (Iy - Iz) / Ix;
  mA(5, 4) = wz * (Iz - Ix) / Iy; mA(5, 5) = 0.0; mA(5, 6) = wx * (Iz - Ix) / Iy;
  mA(6, 4) = wy * (Ix - Iy) / Iz; mA(6, 5) = wx * (Ix - Iy) / Iz; mA(6, 6) = 0.0;
}

void KalmanFilter::UpdateMatrixB(NonLinear &nol) {
  mB(4, 0) = 1.0 / nol.Ix_;
  mB(5, 1) = 1.0 / nol.Iy_;
  mB(6, 2) = 1.0 / nol.Iz_;
}

/*
    @Input: col: column of DCM to observe
*/

void KalmanFilter::UpdateMatrixH(int col) {
  double q0 = vSysEst(0) * 2.0;
  double q1 = vSysEst(1) * 2.0;
  double q2 = vSysEst(2) * 2.0;
  double q3 = vSysEst(3) * 2.0;

  for (int i = 0; i < 3; i++) {
    mH(i, 4) = 0.0; mH(i, 5) = 0.0; mH(i, 6) = 0.0;
  }
  if (col == 0) {
    mH(0, 0) = +q0; mH(0, 1) = +q1; mH(0, 2) = -q2; mH(0, 3) = -q3;
    mH(1, 0) = +q3; mH(1, 1) = +q2; mH(1, 2) = +q1; mH(1, 3) = +q0;
    mH(2, 0) = -q2; mH(2, 1) = +q3; mH(2, 2) = -q0; mH(2, 3) = +q1;
  } else if (col == 1) {
    mH(0, 0) = -q3; mH(0, 1) = +q2; mH(0, 2) = +q1; mH(0, 3) = -q0;
    mH(1, 0) = +q0; mH(1, 1) = -q1; mH(1, 2) = +q2; mH(1, 3) = -q3;
    mH(2, 0) = +q1; mH(2, 1) = +q0; mH(2, 2) = +q3; mH(2, 3) = +q2;
  } else if (col == 2) {
    mH(0, 0) = +q2; mH(0, 1) = +q3; mH(0, 2) = +q0; mH(0, 3) = +q1;
    mH(1, 0) = -q1; mH(1, 1) = -q0; mH(1, 2) = +q3; mH(1, 3) = +q2;
    mH(2, 0) = +q0; mH(2, 1) = -q1; mH(2, 2) = -q2; mH(2, 3) = +q3;
  } else {
    std::cout << "DCM COL INDEX ERROR" << std::endl;
  }
}

/**************************************************
    Update Kalman-filter Matrices
      void UpdateMatrixPhi()
      void UpdateMatrixGmm()
**************************************************/

void KalmanFilter::UpdateMatrixPhi() {
  Eigen::MatrixXd mTemp = dt * mA;
  mPhi = mTemp.exp();
}

void KalmanFilter::UpdateMatrixGmm() {
  static Eigen::MatrixXd mI = Eigen::MatrixXd::Identity(dimS, dimS);
  Eigen::MatrixXd mTemp = dt * mA;
  mGmm = mA.inverse() * (mTemp.exp() - mI) * mB;
}

/*******************************************************
    Fuctions for simulation
      void KalmanFilter::simulation(NonLinear &nol, double T_END,
                                    double obs_interval)
      double KalmanFilter::CalcQuaternionNorm(Eigen::VectorXd &vX)
      void KalmanFilter::NormalizeQuaternion(Eigen::VectorXd &vX)
      double KalmanFilter::CalcErrNorm(Eigen::VectorXd &vSysTru,
```

```
                                           Eigen::VectorXd &vSysEst)
       int KalmanFilter::select_column(int seed)
**********************************************************/

/*
    @input
      nol: NonLinear Class for non-linear model propagation
      T_END: simulation time
      obs_interval: observation interval
*/

void KalmanFilter::simulation(NonLinear &nol, double T_END,
                              double obs_interval) {
  Eigen::Matrix<double, 3, 3> dcm_Tru;
  Eigen::Matrix<double, 3, 3> dcm_Est;
  Eigen::VectorXd vY_Tru; // one column of dcm_Tru + error
  Eigen::VectorXd vY_Est; // one column of

  vY_Tru = Eigen::VectorXd::Zero(dimO);
  vY_Est = Eigen::VectorXd::Zero(dimO);
  double time_obs = 0.0; // counter

  for (double t = 0; t < T_END + dt; t += dt) {
    nol.writefile(t, vSysTru, dcm_Tru, "R2True");
    nol.writefile(t, vSysEst, dcm_Est, "R2Est");
    // 真値計算
    nol.update_vW(); // make disturbance noise
    nol.propagation(dt, vSysTru, dcm_Tru, t, t + dt);
    // 推定値計算
    nol.del_noise();
    nol.propagation(dt, vSysEst, dcm_Est, t, t + dt); // estimate with no noise

    // Matrix更新
    UpdateMatrixA(nol);
    UpdateMatrixB(nol);
    UpdateMatrixPhi();
    UpdateMatrixGmm();
    mM = mPhi * mP * mPhi.transpose() + mGmm * mQ * mGmm.transpose();
    mP = mM;
    if (time_obs >= obs_interval) {
      std::cout << "observe time: " << t << std::endl;
      std::cout << "  vSysTru:" << vSysTru.transpose() << std::endl;
      std::cout << "  vSysEst:" << vSysEst.transpose() << std::endl;
      time_obs = 0.0;
      int seed = 0;

      int dcm_col = select_column(seed); // select column to observe
      std::cout << "  column observed:" << dcm_col << std::endl;
      UpdateMatrixH(dcm_col);
      nol.update_vV();                          // make observation noise
      vY_Tru = dcm_Tru.col(dcm_col) + nol.vV_; // real
      nol.del_noise();
      vY_Est = dcm_Est.col(dcm_col) + nol.vV_; // estimation
      vz = vY_Tru - vY_Est;
      std::cout << "  vz:" << vz.transpose() << std::endl;
      double err = CalcErrNorm(vSysTru, vSysEst);
      std::cout << "  Error Norm(before):" << err << std::endl;

      Eigen::MatrixXd mTemp; // temporary defined matrix used for calculation
      mTemp = Eigen::MatrixXd::Zero(3, 3);
      mTemp = mH * mM * mH.transpose() + mR;
      mP = mM - mM * mH.transpose() * mTemp.inverse() * mH * mM;
      mK = mP * mH.transpose() * mR.inverse();

      vSysEst = vSysEst + mK * vz; // update estimation
      err = CalcErrNorm(vSysTru, vSysEst);
      std::cout << "  Error Norm(after):" << err << std::endl;
      std::cout << "  Kalman Gain norm:" << mK.norm() << std::endl;
      for (int i = 0; i < dimS; i++) {
        std::cout << "  P" << i << ":" << sqrt(mP(i, i)) << " ";
        if (i == dimS - 1)
          std::cout << std::endl;
      }
      std::cout << std::endl;
      NormalizeQuaternion(vSysEst);
      write_file_k(t);
    }
```

```
    write_file_pe(t); // write mP,mK
    time_obs += dt;
  }
}

/* @Input
    vX: System states Vector
*/
double KalmanFilter::CalcQuaternionNorm(Eigen::VectorXd &vX) {
  double result = 0;
  for (int i = 0; i < 4; i++) {
    result += vX(i) * vX(i);
  }
  return sqrt(result);
}

/* @Input
    vX: System states Vector
*/
void KalmanFilter::NormalizeQuaternion(Eigen::VectorXd &vX) {
  double norm = CalcQuaternionNorm(vX);
  for (int i = 0; i < 4; i++) {
    vX(i) = vX(i) / norm;
  }
}

/* @Input
    vSysTru: System states Vector (True(propagated with errors))
    vSysEst: System states Vector (Estimated(propagated without errors))
*/
double KalmanFilter::CalcErrNorm(Eigen::VectorXd &vSysTru,
                                 Eigen::VectorXd &vSysEst) {
  Eigen::VectorXd vD = vSysTru - vSysEst;
  return vD.norm();
}

// select column od DCM (for observation(updating Matrix H))
int KalmanFilter::select_column(int seed) {
  int column = 0;
  if (seed == 0) {
    std::random_device seed_gen;
    std::mt19937 randMt(seed_gen());
    std::uniform_int_distribution<int> rand_dist(0, 2);
    column = rand_dist(randMt);
  } else {
    std::mt19937 randMt(seed);
    std::uniform_int_distribution<int> rand_dist(0, 2);
    column = rand_dist(randMt);
  }
  return column;
}

/**********************************************************
    Functions for writing result (Matrix P,K)
      write_file_pe()
      write_file_k()
***********************************************************/

void KalmanFilter::write_file_pe(double t) {
  // files to write result
  // Estimate error covariance
  std::fstream fout_p;
  std::string filename_p;
  filename_p = "../datafile/R2p.txt";
  // errors
  std::fstream fout_e;
  std::string filename_e;
  filename_e = "../datafile/R2e.txt";

  if (t < 2 * dt) {
    fout_p.open(filename_p, std::ios::out);
    fout_e.open(filename_e, std::ios::out);

  } else {
    fout_p.open(filename_p, std::ios::app);
    fout_e.open(filename_e, std::ios::app);
  }
```

```
  fout_p << t << " ";
  fout_e << t << " ";
  for (int i = 0; i < dimS; i++) {
    fout_p << sqrt(mP(i, i)) << " ";
    if (i == dimS - 1)
      fout_p << std::endl;
  }
  Eigen::VectorXd vD = vSysTru - vSysEst;
  for (int i = 0; i < dimS; i++) {
    fout_e << vD(i) << " ";
    if (i == dimS - 1)
      fout_e << std::endl;
  }
}

void KalmanFilter::write_file_k(double t) {
  // files to write result
  // kalman gain
  std::fstream fout_k;
  std::string filename_k;
  filename_k = "../datafile/R2k.txt";

  if (t < 2) {
    fout_k.open(filename_k, std::ios::out);

  } else {
    fout_k.open(filename_k, std::ios::app);
  }
  double norm = mK.norm();
  fout_k << t << " " << norm << std::endl;
}
```