The DJH Inertial Navigation System Package

David Hanley

October 30, 2017

Abstract

In this document, we cover the details of the Inertial Navigation System: the various classes and their relationship to each other, future work, and related references for inertial navigation.

1 Introduction: System Overview

As shown by Figure 1, the DJH Inertial Navigation System package is composed of C++ classes configured into three related layers. The first layer is the Interface Layer, which is composed of the djh_ins class. This is the only class an individual using the package will interact with. Initializing this function, a user sets a gravity model, IMU model, and chooses the method of INS integration (or preintegration). The djh_ins class is composed of a set of classes that correct IMU measurements using the IMU model and subsequently integrate (or preintegrate) measurements. This set of classes makes up the Integration Layer. The Integrator class within the Integration Layer is also composed of a series of classes for using JPL-style quaternions, using generic rigid body dynamics, and using a given model of gravity. This last set of classes makes up the Model Layer of the package.

This package assumes that Eigen is already installed on the system. See the Wiki page on the DJH INS github for installation instructions and examples of usage.

2 Individual Class Descriptions

Note: italicized variables and data types indicate optional inputs.

2.1 QuatMath

QuatMath is a quaternion math class for the INS package. This class contains quaternion math functions using the JPL quaternion convention. To the best of our knowledge, the Eigen library uses the Hamilton convention for quaternions. So we do not use that functionality.

Attribute: quat: Vector4d

Public JPL Quaternion variable

$$quat = [q_x, q_y, q_z, q_w]^T$$

Constructor: QuatMath(*Vector4d quat_input*)

quat_input = JPL input quaternion of the form $[q_x, q_y, q_z, q_w]^T$

Function: Normalize()

This function ensures that the class' quaternion has a norm of 1.

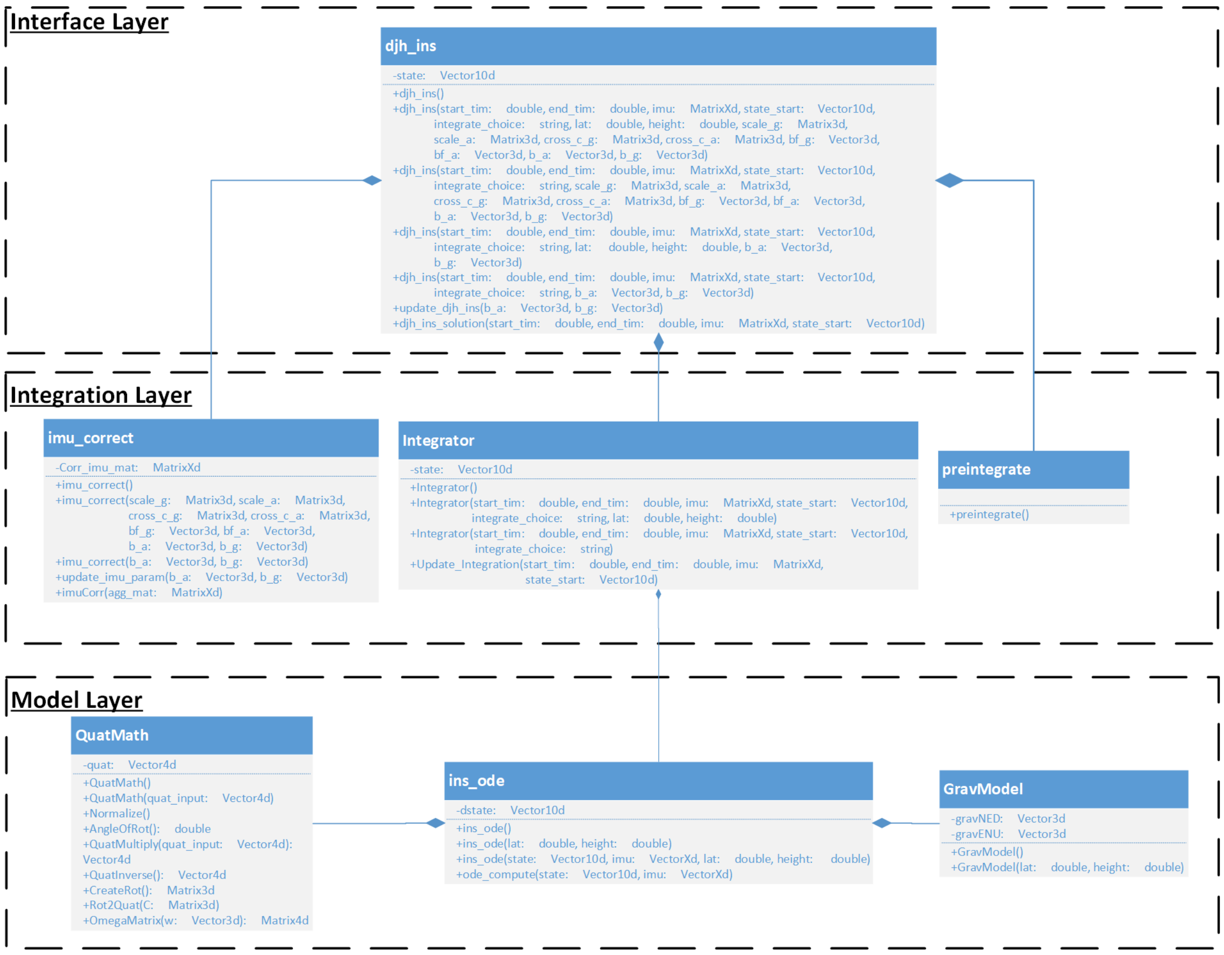
Interface Layer
djh_ins
-state: Vector10d
+djh_ins()
+djh_ins(start_tim: double, end_tim: double, imu: MatrixXd, state_start: Vector10d, integrate_choice: string, lat: double, height: double, scale_g: Matrix3d, scale_a: Matrix3d, cross_c_g: Matrix3d, cross_c_a: Matrix3d, bf_g: Vector3d, bf_a: Vector3d, b_a: Vector3d, b_g: Vector3d)
+djh_ins(start_tim: double, end_tim: double, imu: MatrixXd, state_start: Vector10d, integrate_choice: string, scale_g: Matrix3d, scale_a: Matrix3d, cross_c_g: Matrix3d, cross_c_a: Matrix3d, bf_g: Vector3d, bf_a: Vector3d, b_a: Vector3d, b_g: Vector3d)
+djh_ins(start_tim: double, end_tim: double, imu: MatrixXd, state_start: Vector10d, integrate_choice: string, lat: double, height: double, b_a: Vector3d, b_g: Vector3d)
+djh_ins(start_tim: double, end_tim: double, imu: MatrixXd, state_start: Vector10d, integrate_choice: string, b_a: Vector3d, b_g: Vector3d)
+update_djh_ins(b_a: Vector3d, b_g: Vector3d)
+djh_ins_solution(start_tim: double, end_tim: double, imu: MatrixXd, state_start: Vector10d)
Integration Layer
imu_correct
-Corr_imu_mat: MatrixXd
+imu_correct()
+imu_correct(scale_g: Matrix3d, scale_a: Matrix3d, cross_c_g: Matrix3d, cross_c_a: Matrix3d, bf_g: Vector3d, bf_a: Vector3d, b_a: Vector3d, b_g: Vector3d)
+imu_correct(b_a: Vector3d, b_g: Vector3d)
+update_imu_param(b_a: Vector3d, b_g: Vector3d)
+imuCorr(agg_mat: MatrixXd)
Integrator
-state: Vector10d
+Integrator()
+Integrator(start_tim: double, end_tim: double, imu: MatrixXd, state_start: Vector10d, integrate_choice: string, lat: double, height: double)
+Integrator(start_tim: double, end_tim: double, imu: MatrixXd, state_start: Vector10d, integrate_choice: string)
+Update_Integration(start_tim: double, end_tim: double, imu: MatrixXd, state_start: Vector10d)
preintegrate
+preintegrate()
Model Layer
QuatMath
-quat: Vector4d
+QuatMath()
+QuatMath(quat_input: Vector4d)
+Normalize()
+AngleOfRot(): double
+QuatMultiply(quat_input: Vector4d): Vector4d
+QuatInverse(): Vector4d
+CreateRot(): Matrix3d
+Rot2Quat(C: Matrix3d)
+OmegaMatrix(w: Vector3d): Matrix4d
ins_ode
-dstate: Vector10d
+ins_ode()
+ins_ode(lat: double, height: double)
+ins_ode(state: Vector10d, imu: VectorXd, lat: double, height: double)
+ode_compute(state: Vector10d, imu: VectorXd)
GravModel
-gravNED: Vector3d
-gravENU: Vector3d
+GravModel()
+GravModel(lat: double, height: double)

Figure 1: This UML diagram of the DJH INS package shows that classes in the system are broken into three layers: the interface layer, the integration layer, and the model layer. When using the package, users only use the interface layer and its class.

Function: AngleOfRot()

Compute the angle of rotation of the class' quaternion.

Output: the angle of rotation in radians.

Function: QuatMultiply(Vector4d quat_input): Vector4d

Multiply the class quaternion (q) by an input quaternion (p) in the form q*p

quat_input = JPL input quaternion of the form

Output: Output quaternion from multiplication q*p

Function: QuatInverse(): Vector4d

Compute and output the inverse of the class' quaternion.

Output: the inverse of the class' quaternion

Function: CreateRot(): Matrix3d

Output rotation matrix of the class' quaternion

Output: the 3×3 rotation matrix represented by the class' quaternion

Function: Rot2Quat(Matrix3d C)

Assigns the class' quaternion to be equivalent to a rotation matrix input

C = a rotation matrix to be converted to a quaternion

Function: OmegaMatrix(Vector3d w): Matrix 4d

This function computes a 4×4 matrix of angular velocities from a vector of angular velocities. This matrix can be used in a quaternion's equation of motion.

w = a three element vector of angular velocity $\begin{bmatrix} w_x & w_y & w_z \end{bmatrix}^T$.

Output: a 4×4 matrix of angular velocities that can be used in quaternion equations of motion.

2.2 ins_ode

ins_ode is a class expressing the ordinary differential equations for a generic rigid body as used in an inertial navigation system.

Attribute: dstate: Vector10d

State time derivative: $\begin{bmatrix} quat & position & velocity \end{bmatrix}$

Constructor: ins_ode(*Vector10d state, VectorXd imu, double lat, double height*)

state = $\begin{bmatrix} quat & position & velocity \end{bmatrix}$

imu = $\begin{bmatrix} accel & gyros \end{bmatrix}$

lat = latitude of sensor (in radians) for gravity calculation

height = height of sensor (in meters) for gravity calculation

This class can be constructed using no inputs, using only latitude and height, or using all the optional inputs.

Function: ode_compute(*Vector10d state, VectorXd imu*)

This function updates the state derivatives.

2.3 GravModel

Attribute: gravNED: Vector3d

The NED (North, East, Down) gravity vector

Attribute: gravENU: Vector3d

The ENU (East, North, Up) gravity vector

Constructor: GravModel(*double lat, double height*)

lat = latitude (in radians) of current sensor location

height = altitude (in meters) of current sensor location

2.4 preintegrate

Constructor: preintegrate()

2.5 imu_correct

Attribute: Corr_imu_mat: MatrixXd

Aggregated IMU matrix adjusted to eliminate IMU errors.

Matrix is of the form:

$$\begin{bmatrix} time_1 & accel & gyro \\ time_2 & accel & gyro \\ ... & ... & ... \end{bmatrix}$$

Constructor: imu_correct(*Matrix3d scale_g, Matrix3d scale_a, Matrix3d cross_c_g, Matrix3d cross_c_a, Vector3d bf_g, Vector3d bf_a, Vector3d b_a, Vector3d b_g*);

scale_g = scale factor errors for IMU gyroscopes

scale_a = scale factor errors for IMU accelerometers

cross_c_g = cross coupling errors for IMU gyroscopes

cross_c_a = cross coupling errors for IMU accelerometers

bf_g = fixed gyroscope biases

bf_a = fixed accelerometer biases

b_a = accelerometer biases estimated online

b_g = gyroscope biases estimated online

This class can be constructed using no inputs, online estimated accelerometer and gyroscope biases, or using all the optional inputs.

Function: update_imu_param(Vector3d b_a, Vector3d b_g)

Update the IMU error components

b_a = accelerometer biases estimated online

b_g = gyroscope biases estimated online

Function: imuCorr(MatrixXd agg_mat)

imuCorr accepts an aggregated n-by-7 IMU matrix and corrects the IMU measurements for fixed scale, bias, and cross-coupling factors. It then adds a bias estimated online.

agg_mat = Aggregated IMU matrix of the form

$$\begin{bmatrix} time_1 & accel & gyro \\ time_2 & accel & gyro \\ ... & ... & ... \end{bmatrix}$$

2.6 Integrator

Attributes: state: Vector10d

This is the 10 dimensional INS State Estimate

$\begin{bmatrix} quat & position & velocity \end{bmatrix}^T$

Constructor: Integrator(*double start_tim, double end_tim, MatrixXd imu, Vector10d state_start, string integrate_choice, double lat, double height*)

start_tim = start time in seconds (presumably a Unix time)

end_tim = end time in seconds (presumable a Unix time)

imu = IMU measurement matrix of the form

$$\begin{bmatrix} time_1 & accel & gyro \\ time_2 & accel & gyro \\ ... & ... & ... \end{bmatrix}$$

And we assume entries are in chronological order (i.e. time_1 < time_2 < ...)

state_start = starting state prior to integration (10 elements: quaternion, position, velocity)

integration_choice = a string selecting either "Euler", "Heun", "RK4", or "Savage" methods

lat = latitude of the starting point (for gravity model) height = height of the starting point above sea level (for gravity model)

This class can be constructed using no inputs, all the optional inputs, or using start time, end time, imu data, the starting state and the choice of integrator.

Function: Update_Integration(double start_tim, double end_tim, MatrixXd imu, Vector10d state_start)

Update_Integration updates the INS solution given a start and end time, IMU measurements, and a starting state estimate.

start_tim = start time in seconds (presumably a Unix time)

end_tim = end time in seconds (presumably a Unix time)

imu = IMU measurement matrix of the form

$$\begin{bmatrix} time_1 & accel & gyro \\ time_2 & accel & gyro \\ ... & ... & ... \end{bmatrix}$$

And we assume entries are in chronological order (i.e. time_1 < time_2 < ...)

state_start = starting state prior to integration (10 elements: quaternion, position, velocity)

2.7 djh_ins

Attributes: state: Vector10d

This is the 10 dimensional INS State Estimate

$\begin{bmatrix} quat & position & velocity \end{bmatrix}^T$

Constructor: djh_ins(*double start_tim, double end_tim, MatrixXd imu, Vector10d state_start, string integrate_choice, double lat, double height, Matrix3d scale_g, Matrix3d scale_a, Matrix3d cross_c_g, Matrix3d cross_c_a, Vector3d bf_g, Vector3d bf_a, Vector3d b_a, Vector3d b_g*)

start_tim = start time in seconds (presumably a Unix time) end_tim = end time in seconds (presumable a Unix time)

imu = IMU measurement matrix of the form

$$\begin{bmatrix} time_1 & accel & gyro \\ time_2 & accel & gyro \\ ... & ... & ... \end{bmatrix}$$

And we assume entries are in chronological order (i.e. time_1 < time_2 < ...)

state_start = starting state prior to integration (10 elements: quaternion, position, velocity)

integration_choice = a string selecting either "Euler", "Heun", "RK4", or "Savage" methods

lat = latitude of the starting point (for gravity model)

height = height of the starting point above sea level (for gravity model)

scale_g = scale factor errors for IMU gyroscopes

scale_a = scale factor errors for IMU accelerometers

cross_c_g = cross coupling errors for IMU gyroscopes

cross_c_a = cross coupling errors for IMU accelerometers

bf_g = fixed gyroscope biases

bf_a = fixed accelerometer biases

b_a = accelerometer biases estimated online

b_g = gyroscope biases estimated online

This class can be constructed using (1)no inputs, (2)all the optional inputs, (3)using start time, end time, IMU data, starting state, the choice of integrator, the scale factor errors for IMU gyroscopes and accelerometers, the cross coupling errors for IMU gyroscopes and accelerometers, the fixed biases for gyroscopes and acclerometers, and online estimated accelerometer and gyroscope biases, (4)using start time, end time, IMU data, starting state, the choice of integrator, the latitude, the height, and accelerometer and gyroscope biases estimated online, (5)using start time, end time, IMU data, starting state, the choice of integrator, and accelerometer and gyroscope biases estimated online.

Function: update_djh_ins(Vector3d b_a, Vector3d b_g)

Update IMU error components

b_a = accelerometer biases estimated online

b_g = gyroscope biases estimated online

Function: djh_ins_solution(double start_tim, double end_tim, MatrixXd imu, Vector10d state_start)

Correct IMU data and generate updated state estimate using integration

start_tim = start time in seconds (presumably a Unix time)

end_tim = end time in seconds (presumably a Unix time)

imu = IMU measurement matrix of the form

$$\begin{bmatrix} time_1 & accel & gyro \\ time_2 & accel & gyro \\ ... & ... & ... \end{bmatrix}$$

And we assume entries are in chronological order (i.e. time_1 < time_2 < ...)

state_start = starting state prior to integration (10 elements: quaternion, position, velocity)

3 Future Work

- Include IMU integration described by Paul G. Savage [1, 2]
- Include IMU preintegration computation as described by Kevin Eckenhoff, Patrick Geneva, and Guoquan Huang [3]
- Provide as unit tests, the scenarios for testing INS systems described by Paul G. Savage [4]

4 Related References

General information about numerical integration methods can be found in [5]. For an overview of inertial measurement units, the technology involved, and a look at inertial navigation systems see the work of Titterton and Weston [6]. Paul G. Savage's work on inertial navigation has been and appears to remain the current state of the art in the field [1, 2, 4, 7, 8]. For information about JPL-style quaternions see [9–12]. Information about preintegration (and its application to visual-inertial odometry) is described in [3, 13, 14].

References

[1] P. G. Savage, "Strapdown inertial navigation integration algorithm design part 1: Attitude algorithms," *Journal of Guidane, Control, and Dynamics*, vol. 21, no. 1, pp. 19–28, Jan 1998.

[2] ——, "Strapdown inertial navigation integration algorithm design part 2: Velocity and position algorithms," *Journal of Guidane, Control, and Dynamics*, vol. 21, no. 2, pp. 208–221, March 1998.

[3] K. Eckenhoff, P. Geneva, and G. Huang, "High-Accuracy Preintegration for Visual-Inertial Navigation," in *Proceedings of Workshop on the Algorithmic Foundations of Robotics*, San Francisco, CA, Dec 2016.

[4] P. G. Savage, "Performance analysis of strapdown systems," The Research and Technology Organization of NATO, Tech. Rep. RTO-EN-SET-116(2009), 2009.

[5] M. T. Heath, *Scientific Computing: An Introductory Survey*, 2nd ed. Boston, MA: McGraw-Hill Higher Education, 2002.

[6] D. Titterton and J. Weston, *Strapdown Inertial Navigation Technology*, 2nd ed., ser. Radar, Sonar, Navigation and Avioncs Series. London, UK: The Institution of Engineering and Technology, 2004, vol. 17.

[7] P. G. Savage, *Strapdown Analytics: Part 1*, 2nd ed. Maple Plain, MN: Strapdown Associates, Inc., 2007.

[8] ——, *Strapdown Analytics: Part 2*, 2nd ed. Maple Plain, MN: Strapdown Associates, Inc., 2007.

[9] T. Barfoot, J. R. Forbes, and P. T. Furgale, "Pose estimation using linearized rotations and quaternion algebra," *Acta Astronautica*, vol. 68, pp. 101–112, 2011.

[10] N. Trawny and S. I. Roumeliotis, "Indirect kalman filter for 3d attitude estimation: A tutorial for quaternion algebra," Multiple Autonomous Robotic Systems Laboratory, Department of Computer Science, University of Minnesota, 4-192 EE/CS Building, 200 Union St. S.E., Minneapolis, MN 55455, Tech. Rep. 2005-002, Rev. 57, March 2005.

[11] "Quaternions white paper," NASA Navigation and Ancillary Information Facility (NAIF), Tech. Rep., Nov 2003.

[12] M. D. Shuster, "A Survey of Attitude Representations," *Journal of the Astonautical Sciences*, vol. 41, no. 4, pp. 439–517, Oct 1993.

[13] C. Forster, L. Carlone, F. Dellaert, and D. Scaramuzza, "On-Manifold Preintegration for Real-Time Visual–Inertial Odometry," *IEEE Transactions on Robotics*, vol. 33, no. 1, pp. 1–21, Feb 2017.

[14] ——, "IMU Preintegration on Manifold for Efficient Visual-Inertial Maximum-a-Posteriori Estimation," in *Proceedings of Robotics: Science and Systems*, Rome, Italy, July 2015.