VINS online extrinsic calibration

Saturday, January 30, 2021 10:58 AM

extrinsics: from camera frame to IMU frame: ${}_bT_c$

We construct least sequares $Ax=0$, and use SVD to find the smallest singular value.
It's corresponding eigenvector is the solution

To construct least squares:

between time $k, k+1$, we have IMU transformation $q_{b_k b_{k+1}}$, camera transformation $q_{c_k c_{k+1}}$

$$q^{b_k}_{b_{k+1}} \otimes \underline{q^{c_k}_{b_k}} = \underline{q^{c_{k+1}}_{b_{k+1}}} \otimes q^{c_k}_{c_{k+1}} = q^{c_k}_{b_{k+1}} \Big\} \text{ from } c_k \text{ to } b_{k+1} \text{ transformation.}$$

equal

$$\therefore \left([q_{b_k b_{k+1}}]_L - [q_{c_k c_{k+1}}]_R\right) q_{bc} = Q^k_{k+1}\, q_{bc} = 0. \quad (6)$$

$[\cdot]_L$, $[\cdot]_R$ are left/right quaternion multiplication.

Stack the equations at multiple times, and use the robust kernel cost function:

$$\begin{bmatrix} w^0_1 \cdot Q^0_1 \\ w^1_2 \cdot Q^1_2 \\ \vdots \\ w^{N-1}_N\, Q^{N-1}_N \end{bmatrix} q_{bc} = Q_N \cdot q_{bc} = 0.$$

where
$$w^k_{k+1} = \begin{cases} 1 & r^k_{k+1} < \text{threshold} \\ \dfrac{\text{threshold}}{r^k_{k+1}} & \text{otherwise} \end{cases}$$

for rotation residual r:

$$r^k_{k+1} = \operatorname{acos}\left(\left(tr\left(\hat{R}^{-1}_{bc} R^{-1}_{b_k b_{k+1}} \hat{R}_{bc} R_{c_k c_{k+1}}\right) - 1\right) / 2\right)$$

In code:

```
if( initial_ex_rotation.CalibrationExRotation( corres, pre_integrations[frame_count] -> delta_q, calib_ric))
```

corres: feature correspondence between two frames, normalized coordinates, a vector

delta_q: relative rotation from pre integration.

```
bool InitialExRotation::CalibrationExRotation( vector<pair<Vector3d, Vector3d>> corres,
                                               Quaterniond delta_q_imu,
                                               Matrix3d &calib_ric_result)
{
    // record the time we enter this function
    // each time construct one row in Eq (6)
    frame_count ++;

    // use matched features to find essential matrix
    // decompose it to get rotation.
    Rc.push_back( solveRelativeR( corres ));

    Rimu.push_back( delta_q_imu.toRotationMatrix());

    // convert R^{bk}_{bk+1} to R^{ck}_{ck+1}:  R^{ck}_{ck+1} = R^b_c R^{bk}_{bk+1} R^c_b
    Rc_g.push_back(ric.inverse() * delta_q_imu * ric);

    // construct A matrix
    Eigen::MatrixXd A(frame_count * 4, 4);

    A.setZero();

    int sum_ok = 0;
    for (int i=1; i<= frame_count; i++)
    {
        Quaterniond r1(Rc[i]);
        Quaterniond r2(Rc_g[i]);

        // anglarDistance is the relative transformation theta
        // used in angle-axis θu.
        // this is to compute weights for robust kernel.
        double angular_distance = 180 / M_PI * r1.angularDistance(r2);
        double huber = angular_distance > 5.0 ? 5.0 / angular_distance : 1.0;
        ++sum_ok;

        Matrix4d L, R;
        double w = Quaterniond( Rc[i]).w();
        Vector3d q = Quaterniond( Rc[i]).vec();
        L.block<3,3>(0, 0) = w * Matrix3d::Identity() + Utility::skewSymmetric(q);
        L.block<3,1>(0, 3) = q;
        L.block<1,3>(3,0) = - q.transpose();
        L(3,3) = w;

        Quaterniond R_ij(Rimu[i]);
        w = R_ij.w();
        q = R_ij.vec();
        R.block<3,3> (0,0) = w * Matrix3d::Identity() _ Utility::SkewSymmetric(q);
        R.block<3,1>(0,3) = q;
        R.block<1,3>(3,0) = - q.transpose();
        R(3,3) = w;

        A.block<4,4>((i-1) * 4, 0) = huber * ( L - R);
    }

    // use SVD to solve the least squares system.
    JacobiSVD<MatrixXd> svd (A, ComputeFullU | ComputeFullV);

    // quaternion is [x, y, z, w]^T, i.e. [qv qw]^T
    Matrix<double, 4, 1> x = svd.matrixV().col(3);
    Quaterniond estimated_R(x);
    ric = estimated_R.toRotationMatrix().inverse();
    Vector3d ric_cov;
    ric_cov = svd.singularValues().tail<3>();

    // iterate WINDOW_SIZE times, and the second smallest singular value is larger than 0.25
    if (frame_count >= WINDOW_SIZE && ric_cov(1) > 0.25)
    {
        calib_ric_result = ric;
        return true;
    }
    else
        reture false;
}
```