Openvins feature initialization

Friday, April 3, 2020 4:48 PM

In code ov_core/src/feat/FeatureInitializer.cpp:

```
bool FeatureInitializer::single_triangulation(Feat *feat,
  std::unordered_map<size_t, std::unordered_map<double, ClonePose>> &clonesCAM) {

    int total_meas = 0;
    size_t anchor_most_meas = 0;
    size_t most_meas = 0;
    for (auto const& pair : feat->timestamps) {
        total_meas += (int) pair.second.size();
        if (pair.second.size() > most_meas) {
            anchor_most_meas = pair.first;
            most_meas = pair.second.size();
        }
    }
```

In feature.h, there's a variable

std::unordered_map<size_t, std::vector<double>> timestamps;

size_t is the camera id, and std::vector<double> holds a vector of timestamps correspond to observations.

In FeatureDatabase.h:

```
void update_feature(size_t id, double timestamp, size_t cam_id,
    float u, float v, float u_n, float v_n) {

    std::unique_lock<std::mutex> lck(mtx);
    if (features_idlookup.find(id) != features_idlookup.end()) {
        Feature *feat = features_idlookup[id];
        feat ->uvs[cam_id].emplace_back(Eigen::Vector2f(u,v));
        feat->uvs_norm[cam_id].emplace_back(Eigen::Vector2f(u_n,v_n));
        feat->timestamps[cam_id].emplace_back(timestamp);
        return;
    }
```

emplace_back adds the value to the end of vector.

update_feature is used in track/TrackKLT.cpp

```
void TrackKLT::feed_monocular(double timestamp, cv::Mat &img, size_t cam_id) {
  ...
  for(size_t i=0; i< good_left.size(); i++) {
    cv::Point2f npt_l = undistort_point(good_left.at(i).pt, cam_id);
    database->update_feature(good_ids_left.at(i), timestamp, cam_id,
          good_left.at(i).pt.x, good_left.at(i).pt.y,
          npt_l.x, npt_l.y);
```

In ov_core/src/test_tracking.cpp

```
handle_stereo(double time0, double time1, cv::Mat img0, cv::Mat img1){

   extractor -> fead_stereo(time0, img0, img1, 0, 1);
   extractor -> feed_monocular(time0, img0, 0);
   extractor -> feed_monocular(time0, img1, 1);
```

∴ cam_id means which camera, eg. for monocular cam_id is 0, for stereo cam_id is 0,1, and select the camera that has most observations across time to be anchor.

```
Eigen::MatrixXd A = Eigen::MatrixXd::Zero(2*total_meas, 3);
Eigen::MatrixXd b = Eigen::MatrixXd::Zero(2*total_meas, 1);
size_t c = 0;
ClonePose anchorclone = clonesCAM.at(feat->anchor_cam_id).at(feat->anchor_clone_timestamp);
Eigen::Matrix<double,3,3> &R_GtoA = anchorclone.Rot();
Eigen::Matrix<double,3,1> &p_AinG = anchorclone.pos();
for(auto const& pair : feat -> timestamps) {
   for (size_t m = 0; m < feat->timestamps.at(pair.first).size(); m++) {
      Eigen::Matrix<double,3,3> &R_GtoCi = clonesCAM.at(pair.first).at(feat->timestamps.at(pair.first).at(m)).Rot();
      Eigen::Matrix<double,3,1> &p_CiinG = clonesCAM.at(pair.first).at(feat->timestamps.at(pair.first).at(m)).pos();

      Eigen::Matrix<double,3,3> R_AtoCi;
      R_AtoCi.noalias() = R_GtoCi * R_GtoA.transpose();
      Eigen::Matrix<double,3,1> p_CiinA;
      p_CiinA.noalias() = R_GtoA * (p_CiinG - p_AinG);
```

feature is triangulated in anchor frame A, we can have the transformation from C_i:

$$ {}^{C_i}p_f = {}^{C_i}_{A}R\,({}^{A}p_f - {}^{A}p_{C_i}) \qquad (1) $$

$$ {}^{A}p_f = {}^{C_i}_{A}R^T\,{}^{C_i}p_f + {}^{A}p_{C_i} $$

```
      Eigen::Matrix<double,3,1> b_i;
      b_i << feat->uvs_norm.at(pair.first).at(m)(0), feat->uvs_norm.at(pair.first).at(m)(1), 1;
      b_i << R_AtoCi.transpose() * b_i;
      b_i = b_i / b_i.norm();
      Eigen::Matrix<double,2,3> Bprep = Eigen::Matrix<double,2,3>::Zero();
      Bprep << -b_i(2,0), 0, b_i(0,0), b_i(2,0), -b_i(1,0);

      A.block(2*c, 0, 2,3) = Bprep;
      b.block(2*c, 0, 2, 1).noalias() = Bprep * p_CiinA;
      c++;
   }
}
```

The user can indicate with the noalias() function that there is no aliasing in Eigen, as follows:

matB.noalias() = matA * matA.

This allows Eigen to evaluate the matrix product matA * matA directly into matB.

Eigen block: block of size (p,q) starting at (i,j).

matrix.block(i,j,p,q)

The measurement in current frame is: unknown bearing ${}^{C_i}b$ and depth ${}^{C_i}z$, we can map the feature in current frame via

$$ {}^{C_i}p_f = z_f\,{}^{C_i}b_f = {}^{C_i}z_f \begin{bmatrix} u_n \\ v_n \\ 1 \end{bmatrix} $$

where u_n, v_n are undistorted normalized coordinates.

The bearing can be warped into the anchor frame by substituting eq. (1):

$$ {}^{A}p_f = {}^{C_i}_{A}R^T\,{}^{C_i}p_f + {}^{A}p_{C_i} $$

$$ = {}^{C_i}_{A}R^T z_f\,{}^{C_i}b_f + {}^{A}p_{C_i} $$

$$ = z_f\,{}^{A}b_f + {}^{A}p_{C_i} $$

To remove the need to estimate the extra degree of freedom of depth z_f, we define

$$ {}^{A}n_1 = [\,\overset{-1}{-{}^{A}b_f(3)} \quad 0 \quad \overset{u_n}{{}^{A}b_f(1)}\,]^T $$

$$ {}^{A}n_2 = [\,0 \quad \overset{1}{{}^{A}b_f(3)} \quad \overset{-v_n}{-{}^{A}b_f(2)}\,]^T $$

these are perpendicular with vector ${}^{A}b_f$, ∴

$$ {}^{A}n_1^T\,{}^{A}b_f = 0 $$

$$ {}^{A}n_2^T\,{}^{A}b_f = 0 $$

We can then multiply the eqs to form two eqs only depend on 3 dof unknown ${}^{A}p_f$:

$$ \begin{bmatrix} {}^{A}n_1^T \\ {}^{A}n_2^T \end{bmatrix} {}^{A}p_f = \underbrace{\begin{bmatrix} {}^{A}n_1^T \\ {}^{A}n_2^T \end{bmatrix} {}^{A}z_f\,b_f}_{\text{zero}} + \begin{bmatrix} {}^{A}n_1^T \\ {}^{A}n_2^T \end{bmatrix} {}^{A}p_{C_i} $$

$$ \begin{bmatrix} {}^{A}n_1^T \\ {}^{A}n_2^T \end{bmatrix} {}^{A}p_f = \begin{bmatrix} {}^{A}n_1^T \\ {}^{A}n_2^T \end{bmatrix} {}^{A}p_{C_i} $$

Bperp in code

By stacking all measurements:

$$ \underset{A}{\begin{bmatrix} \vdots \\ {}^{A}n_1^T \\ {}^{A}n_2^T \\ \vdots \end{bmatrix}} \underset{\text{unknown}}{{}^{A}p_f} = \underbrace{\begin{bmatrix} \vdots \\ {}^{A}n_1^T \\ {}^{A}n_2^T \\ \vdots \end{bmatrix} {}^{A}p_{C_i}}_{b} $$

```
Eigen::MatrixXd p_f = A.colPivHouseholderQr().solve(b);
```

Since each pixel measurement provides two constraints, as long as m>1, we have enough constraints to triangulate.

In practice, more views of feature the better, so we choose at least five views.

We also check that the triangulated feature is valid and in front of the camera and not too far.

The condition number of the above linear system is used to reject features that are sensitive to noise.

```
Eigen::JacobiSVD<Eigen::MatrixXd> svd(A, Eigen::ComputeThinU | Eigen::ComputeThinV);
Eigen::MatrixXd singularValues;
singularValues.resize(svd.singularValues().rows(), 1);
singularValues = svd.singularValues();
double condA = singularValues(0,0) / singularValues(singularValues.rows()-1, 0);

if(std::abs(condA) > _options.max_cond_number ||
    p_f(2,0) < _options.min_dist ||
    p_f(2,0) > _options.max_dist ||
    std::isnan(p_f.norm())) {
      return false;
}

feat -> p_FinA = p_f;
feat -> p_FinG = R_GtoA.transpose() * feat -> p_FinA + p_AinG;
return true;
}
```