SLAM Implementation: Bundle Adjustment with g2o

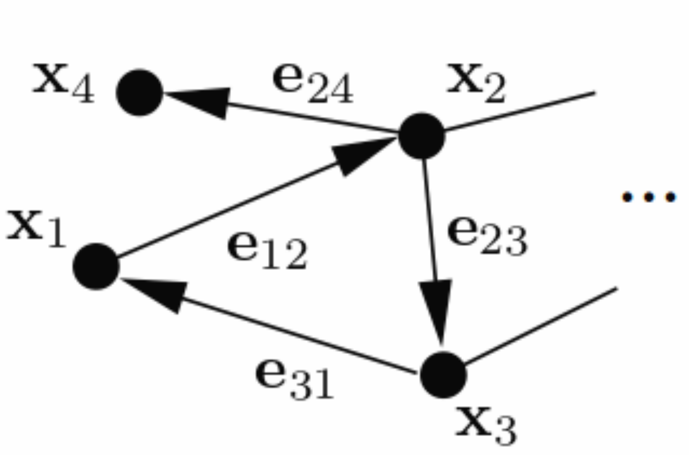
Mar 15, 2016

Contents

Introduction

SLAM problems require a back-end to refine the map and poses constructed in its front-end. The back-end is usually either a filtering framework (like EKF) or graph optimization (i.e. bundle adjustment). Nowadays, graph optimization is much more popular, and has become a state-of-art method.

The general idea of graph optimization is to express the SLAM problem as a graph structure. As the figure below shows, a graph contains two types of elements, nodes (vertices) and constraints (edges). For SLAM problems, a keyframe pose of the robot or a landmark position in the map is denoted as a node, while the observations and geometric model between keyframe and keyframe, keyframe and landmark, or landmark and landmark are denoted as the constraints that connected certain nodes.



$$
\begin{aligned}
\mathbf{F}(\mathbf{x}) = \; & \mathbf{e}_{12}^{\top}\, \boldsymbol{\Omega}_{12}\, \mathbf{e}_{12} \\
& + \mathbf{e}_{23}^{\top}\, \boldsymbol{\Omega}_{23}\, \mathbf{e}_{23} \\
& + \mathbf{e}_{31}^{\top}\, \boldsymbol{\Omega}_{31}\, \mathbf{e}_{31} \\
& + \mathbf{e}_{24}^{\top}\, \boldsymbol{\Omega}_{24}\, \mathbf{e}_{24} \\
& + \dots
\end{aligned}
$$

Fig. 2. This example illustrates how to represent an objective function by a graph.

Given a graph, graph optimization aims to find an optimal estimation of the nodes values which minimize the errors that determined by the constraints. Therefore, SLAM back-end is transformed to be a least squares minimization problem, which can be described by the following equation:

$$
\begin{aligned}
\mathbf{F}(\mathbf{x}) &= \sum_{k \in \mathcal{C}} \underbrace{\mathbf{e}_k(\mathbf{x}_k, \mathbf{z}_k)^T \boldsymbol{\Omega}_k \mathbf{e}_k(\mathbf{x}_k, \mathbf{z}_k)}_{\mathbf{F}_k} \\
\mathbf{x}^* &= \underset{\mathbf{x}}{\operatorname{argmin}}\, \mathbf{F}(\mathbf{x}).
\end{aligned}
$$

g2o

g2o, short for General (Hyper) Graph Optimization [1], is a C++ framework for performing the optimization of nonlinear least squares problems that can be embedded as a graph or in a hyper-graph. It implements many typical vertices and edges as classes that can be directly called and used, like `VertexSE3Expmap` to represent robot poses in SE3 space, `VertexSBAPointXYZ` to represent 3-D points, `EdgeProjectXYZ2UV` to represent observations of 3D points in camera image plane. Also, typical optimization solver algorithms are implemented. With g2o library, what SLAM researcher need to do is defining the nodes and edges in their problems, adding them to the solver provided by g2o, and it will execute all the optimization stuff. g2o is now is a widely used library among SLAM researchers, adopted in many famous SLAM or VO works like ORB_SLAM [2] and SVO [3].

Bundle Adjustment Demo

Code Review

We use one of the example code from g2o to do this bundle adjustment demo. The demo first generalizes some simulated 3D points and keyframe poses, as well the observations of the 3D points in the keyframe camera plane with Gaussian noises, then optimize the graph made up by them. Let's briefly examine the code.

First initialize the optimizer. As the optimization structure may be complicated, we need to assign many solver types mannually:

```
g2o::SparseOptimizer optimizer;
optimizer.setVerbose(false);
g2o::BlockSolver_6_3::LinearSolverType * linearSolver;
if (DENSE) {
linearSolver= new g2o::LinearSolverDense<g2o
    ::BlockSolver_6_3::PoseMatrixType>();
} else {
linearSolver
    = new g2o::LinearSolverCholmod<g2o
    ::BlockSolver_6_3::PoseMatrixType>();
}
g2o::BlockSolver_6_3 * solver_ptr
    = new g2o::BlockSolver_6_3(linearSolver);
g2o::OptimizationAlgorithmLevenberg* solver = new g2o::C
optimizer.setAlgorithm(solver);
```

In this problem, camera intrinsic parameter is also required as part of the measurement constraints, which can be added to the optimizer like:

```
double focal_length= 1000.;
Vector2d principal_point(320., 240.);
vector<g2o::SE3Quat,
    aligned_allocator<g2o::SE3Quat> > true_poses;
g2o::CameraParameters * cam_params
    = new g2o::CameraParameters (focal_length, principal
cam_params->setId(0);
if (!optimizer.addParameter(cam_params)) {
assert(false);
}
```

Then we add all poses as vertices:

```
int vertex_id = 0;
for (size_t i=0; i<15; ++i) {
    Vector3d trans(i*0.04-1.,0,0);

    Eigen:: Quaterniond q;
    q.setIdentity();
    g2o::SE3Quat pose(q,trans);
    g2o::VertexSE3Expmap * v_se3
        = new g2o::VertexSE3Expmap();
    v_se3->setId(vertex_id);
    if (i<2){
        v_se3->setFixed(true);
    }
    v_se3->setEstimate(pose);
    optimizer.addVertex(v_se3);
    true_poses.push_back(pose);
    vertex_id++;
}
```

And map points as well:

```
// Note: code has been simplified for demo convenience
for (size_t i=0; i<true_points.size(); ++i){
    g2o::VertexSBAPointXYZ * v_p
        = new g2o::VertexSBAPointXYZ();
    v_p->setId(point_id);
    v_p->setMarginalized(true);
    v_p->setEstimate(true_points.at(i)
                     + Vector3d(Sample::gaussian(1),
                                Sample::gaussian(1),
                                Sample::gaussian(1)));
    optimizer.addVertex(v_p);
    for (size_t j=0; j<true_poses.size(); ++j){
        // Add edges. See the following passage.
    }
    ++point_id;
}
```

During the process of adding map points vertices, the edges connecting map point vertex and corresponding keyframe vertex are also added:

```
for (size_t j=0; j<true_poses.size(); ++j){
    Vector2d z
        = cam_params->cam_map(true_poses.at(j).map(true_
    double sam = Sample::uniform();
    z += Vector2d(Sample::gaussian(PIXEL_NOISE),
                  Sample::gaussian(PIXEL_NOISE));
    g2o::EdgeProjectXYZ2UV * e
        = new g2o::EdgeProjectXYZ2UV();
    e->setVertex(0, dynamic_cast<g2o::OptimizableGraph::
    e->setVertex(1, dynamic_cast<g2o::OptimizableGraph::
                  (optimizer.vertices().find(j)->second));
    e->setMeasurement(z);
    e->information() = Matrix2d::Identity();
    e->setParameterId(0, 0);
```

```
    optimizer.addEdge(e);
}
```

Please note that for each edge, the two vertices it connects should be specified. Also, an information matrix should be given. In its physical meaning, information matrix represents how reliable this measurement is. Therefore, the more precisely the measurement is made or the more you trust in this measurement, the larger values in the information matrix you can set.

Finally, we can do the optimization:

```
optimizer.initializeOptimization();
cout << "Performing full BA:" << endl;
optimizer.optimize(10);
```

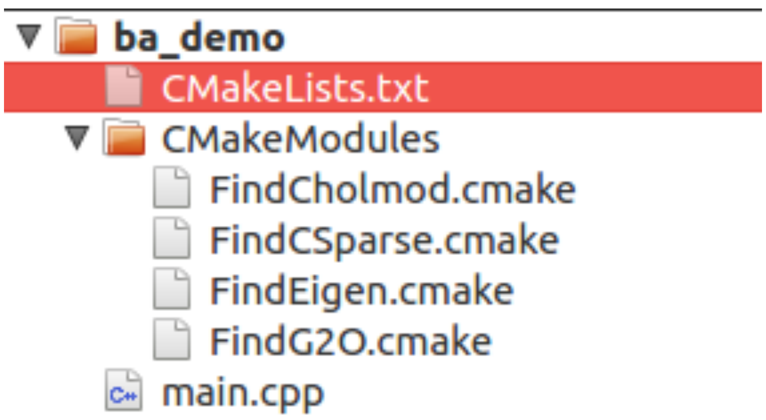
Build the Demo

To build this project, we must install g2o and include and link to it. The installation routine of g2o is given in its project homepage. To include and link to g2o, we use cmake here, with a CMakeLists.txt like:

```
project(ba_demo)
cmake_minimum_required(VERSION 2.8)
LIST(APPEND CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/CM
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x -Wa
FIND_PACKAGE(Eigen REQUIRED)
FIND_PACKAGE(CSparse REQUIRED)
FIND_PACKAGE(Cholmod REQUIRED)
FIND_PACKAGE(G2O REQUIRED)
include_directories(
  ${CMAKE_CURRENT_SOURCE_DIR}
  ${EIGEN_INCLUDE_DIRS}
  ${CSPARSE_INCLUDE_DIR}
  ${Cholmod_INCLUDE_DIR}
```

```
    ${G2O_INCLUDE_DIR}
    /usr/include/suitesparse
)
LIST(APPEND G2O_LIBS
    cxsparse
    cholmod
    g2o_cli g2o_ext_freeglut_minimal g2o_simulator
    g2o_solver_slam2d_linear g2o_types_icp g2o_types_slam2
    g2o_core g2o_interface g2o_solver_csparse g2o_solver_s
    g2o_types_sba g2o_types_slam3d g2o_csparse_extension
    g2o_opengl_helper g2o_solver_dense g2o_stuff
    g2o_types_sclam2d g2o_parser g2o_solver_pcg
    g2o_types_data g2o_types_sim3
)
aux_source_directory(. DIR_SRCS)
add_executable(ba_demo ${DIR_SRCS})
target_link_libraries(ba_demo
${G2O_LIBS}
)
```

Do not forget to put the required `.cmake` files in a sub-directory of the project like:



Run the Demo

Run the program, and we can get:

```
PIXEL_NOISE: 1
OUTLIER_RATIO: 0
ROBUST_KERNEL: 0
STRUCTURE_ONLY: 0
DENSE: 0

Performing full BA:
iteration= 0      chi2= 132333131.273130   time= 0.128667
iteration= 1      chi2= 5627495.269566     time= 0.108231
iteration= 2      chi2= 216111.580556      time= 0.106446
iteration= 3      chi2= 41147.168515       time= 0.106624
iteration= 4      chi2= 15799.044991       time= 0.106301
iteration= 5      chi2= 10624.163303       time= 0.106286
iteration= 6      chi2= 9855.835927        time= 0.106119
iteration= 7      chi2= 9601.932859        time= 0.10637
iteration= 8      chi2= 9172.279560        time= 0.200304
iteration= 9      chi2= 8892.451941        time= 0.200096

Point error before optimisation (inliers only): 1.74126
Point error after optimisation (inliers only): 0.423081
```

It can be seen from the output that g2o did reduce the point errors.

References

[1] Rainer Kuemmerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard g2o: A General Framework for Graph Optimization IEEE International Conference on Robotics and Automation (ICRA), 2011 [PDF]

[2] ORB_SLAM

[3] SVO

Tags: SLAM robotics

Comment by Github Issues