Tutorial on Quick and Easy Model Fitting Using the SLoM Framework

Christoph Hertzberg*, René Wagner, and Udo Frese

SFB/TR8 – Spatial Cognition, Reasoning, Action, Interaction
Universität Bremen, Postfach 330 440, 28334 Bremen, Germany
{chtz,rwagner,ufrese}@informatik.uni-bremen.de
http://informatik.uni-bremen.de/agebv

Abstract. In many areas of experimental science ranging from robotics to psychophysical research, to evaluation of spatial sensor-data and surveying, model fitting is a ubiquitous subproblem. Often it is not the actual scientific goal but rather the "necessary evil" of calibrating the equipment. This tutorial introduces methodology and a library allowing to solve model fitting problems easily without requiring the user to have an in-depth understanding of this subject.
After a brief introduction to the theoretical background we guide the reader through using all main features of the SLoM C++ framework based on a stereo camera and inertial measurement unit (IMU) calibration example which is solved with less than 70 lines of non-problem specific code, and provide hints on applying SLoM to other classes of problems.
The reader is only assumed to have a working knowledge of C++ and a basic understanding of statistics and 3D geometry.

Keywords: Model Fitting, Tutorial, Least Squares, Optimization, Manifolds, Calibration, SLAM

1 Least Squares Optimization in a Nutshell

Least squares optimization determines the most likely values of previously unknown (or only vaguely known) model parameters or *variables* from noisy measured data. For this to work, the measured data and the variables need to be linked in a way that can be expressed as a *measurement function*, a function that predicts the measured data given certain values of one or more variables. The error of the predicted data vs. the actually measured data can be used to adjust the variables to minimize the error. If this is done for all variables and all measurements simultaneously this optimization process yields a maximum likelihood solution, i.e., a variable assignment that is most plausible given the measured data as it minimizes the overall error.

As a concrete example, suppose we want to calibrate a digital camera. The variables to be determined are its intrinsic parameters, i.e., a set of numbers

* Corresponding Author

describing its optics, e.g., the focal length. The measurement data consists of pixel coordinates of some markers detected in an image. The marker positions in the world are known. Then, the measurement function is simply a pinhole camera model which calculates at which pixel coordinates in the image each marker should be seen assuming the camera has certain intrinsic parameters and is located at a certain position and orientation (collectively called *pose* and serving as an auxiliary variable in this example). Now, to determine the maximum likelihood parameters all we need to do is plug an initial guess of all variables, the measurement function and measurement data obtained from different viewpoints into a least squares solver.

In mathematical terms, the above can be captured in a concise but self-contained form (previously presented in [21]) as follows. The model parameters (including auxiliary ones) that we want to fit to the measured data are the random variables $x_{1,\dots,n}$. Each of the measurements $M_{1,\dots,m}$ can be represented as a tuple [21]

$$M_i = (z_i, \Sigma_i, f_i, Y_i = \{x_j | \operatorname{dep}(z_i, x_j)\}). \tag{1}$$

The two most important bits here are z_i, the actually measured datum, and f_i, the so-called measurement function, which returns the expected datum $\hat{z}_i$, i.e., the datum we would expect to measure assuming a set of dependent variables Y_i have certain values. Comparing the two yields the error function to be minimized.

The covariance Σ_i describes the uncertainty of the measurement since, as noted above, the measured data is noisy. More specifically, the measurement errors are assumed to adhere to a normal distribution with mean 0 and covariance Σ_i, i.e., [21]

$$f_i(Y_i) \boxminus z_i \sim \mathcal{N}(0,\, \Sigma_i)\,. \tag{2}$$

You will probably wonder what the curious $\boxminus$ is all about and we will get to that later in the tutorial. For now, it will suffice to think of it as the same as a regular vector subtraction.

So far, we have only looked at individual measurements. We can now form the combined problem as follows. We stack all random variables x_i into the vector X and all individual error functions $f_i(Y_i) \boxminus z_i$ into the "big" combined error function F

$$X = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad\qquad F(X) = \begin{bmatrix} f_1(Y_1) \boxminus z_1 \\ \vdots \\ f_m(Y_m) \boxminus z_m \end{bmatrix}, \tag{3}$$

so that we can state the combined least squares problem as

$$\hat{X} = \operatorname*{argmin}_X \tfrac{1}{2} \left\| F(X) \right\|_\Sigma^2 . \tag{4}$$

The curious Σ in (4) denotes the normalization of all measurement errors according to their respective covariance, i.e., uncertainty. One can think of this as a weighting of the errors depending on the measurement precision. The take-home message here, however, is that in (4) we have brought a wide range of problems into a form that a least squares optimizer will understand, i.e., if we can describe

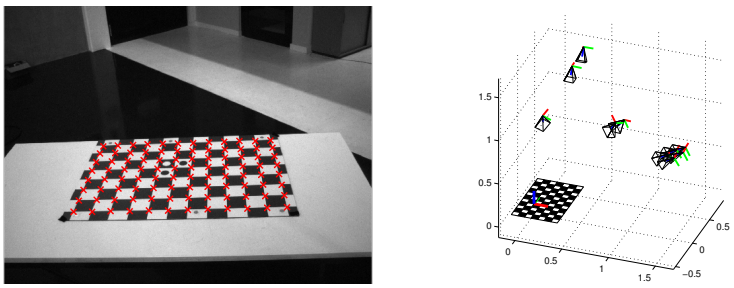

Fig. 1. Results from camera calibration. The left image shows checkerboard corners projected into the image using the estimated parameters. The estimated poses of the cameras w.r.t. the checkerboard are illustrated on the right. Note that for better presentation only the left cameras are displayed. Images are taken from [21].

a model fitting problem as in (1) there is a well-understood black box algorithm that solves it for us. The only other things we need to worry about are that all variables must be observable, i.e., changes of variables always lead to changes in at least one predicted measurement, and that we provide sufficient measurement data to constrain the variables to a unique solution.

Luckily, the above abstraction can be nicely implemented as a software library interface and we will see this in action in the next section.

2 Tutorial

This section will show how to solve model fitting problems using the Sparse Least squares on Manifolds (SLoM) C++-framework. We give actual C++-code, first to be concrete and second because most practical problems are conceptually simple and with SLoM this simplicity carries over to the actual code. SLoM is available as a sub-project of the Manifold ToolKit (MTK) from `http://openslam.org/MTK.html`. MTK uses the Eigen matrix library [9], allowing to write textbook-style matrix expressions. As for the term manifold, for now it will suffice to say that it refers to certain properties all variables have in MTK/SLoM. We will get back to this in §4.

We use the calibration of a stereo camera system with an inertial measurement unit (IMU) as a worked example. The calibration determines all parameters needed to interpret images and inertial measurements spatially. This example is manageable, it is a self-contained realistic application, and shows the main features of SLoM. In particular, such calibration problems tend to have a heterogeneous structure where the SLoM library helps most to avoid complex bookkeeping in the code. The program and an example data set are available from our website `http://www.informatik.uni-bremen.de/agebv/en/pub/hertzbergsc12`. In §3 we discuss the extension to other problems.

Camera calibration involves variables shared by all measurements, e.g., the intrinsic camera parameters which are the same across all calibration images

assuming the camera hardware (its optics in particular) remains the same, and variables that are specific to individual (sets of) measurements, e.g., each calibration image was taken from a different camera pose. Depending on the calibration setup an arbitrary number of measurements can be involved. In this example, the measurements are the coordinates of the checkerboard corners detected in the image (Fig. 1) as well the gravity vector observed by the IMU. The checkerboard geometry is known, so the detected checkerboard corners determine all variables involving the camera, i.e., the intrinsic parameters (optics) and each camera pose. Also, the checkerboard is placed such that it is leveled horizontally, so the cameras observe the direction of up and down from the image whereas the IMU observes it from gravity. Several of these pairs determine the orientation of the IMU relative to the cameras.

We will show how to define variables and measurements, then how to initialize and start the optimization process using the SLoM framework.

2.1 Defining Variables

First of all, the variables which are to be optimized have to be defined. There are some shared variables such as the parameters describing the camera optics, the transformation between left and right camera as well as the 3D orientation between the IMU and the cameras. Another internal parameter is the accelerometer bias of the IMU, which we will assume stays constant during the measurements. Furthermore, we need to estimate the amount of gravity (which is different in different geographical locations).

3D orientations are parameterized by the rotation group $SO(3)$, transformations and poses by the Euclidean group $SE(3)$, both of which are readily implemented by MTK. We declare some `typedef`s for convenience:

```
1  typedef MTK::vect<2> vec2;      // 2D vector
2  typedef MTK::vect<3> vec3;      // 3D vector
3  typedef MTK::SO3< > SO3;        // 3D Orientation
4  typedef MTK::trafo<SO3> SE3;    // 3D Transform
5  typedef MTK::Scalar< > Scalar; // Scalar variable
6  typedef MTK::vect<9> CamIntrinsics; // Camera intrinsics
```

The camera intrinsics consist of a number of scalar values, such as the focal length and lens distortion parameters. We will store them into a single vector (line 6). As `CamIntrinsincs` is essentially a C++ `class`, we can inherit from it and add member functions implementing, e.g., the camera's measurement model:

```
7  struct Camera : public CamIntrinsics {
8    vec2 sensor2image(const vec3& point) const;
9  };
```

Next, we combine two `Camera`s and a transformation between them to a single variable defining a stereo camera:

```
10 MTK_BUILD_MANIFOLD(StereoCamera,
11   ((Camera, left))
12   ((Camera, right))
13   ((SE3, left2right))
14 )
```

Here the macro `MTK_BUILD_MANIFOLD` constructs a new compound manifold, named by the first macro parameter. The second parameter is a list containing the sub-components of the manifold. Each of these is given as a pair specifying the type and the name of the entry enclosed in double parentheses. The macro hides all necessities for SLoM to work with the new manifold.

Again, we could inherit from this class to implement the measurement model for the stereo camera. However, in this case we will do it outside this class later.

2.2 Defining Measurements

Our calibration process will involve two kinds of measurements. Visual measurements of both cameras and accelerometer measurements of the IMU.

As we combined the intrinsics of both cameras to a single variable, we do the same for the measurement. Thus each measurement will depend on a `StereoCamera` as well as an `SE3` describing the pose of the left camera. As measurement data it includes the known position of the checkerboard corner on the plate (considered world frame here) and the corner's pixel coordinates in both camera images. The measurement is declared using the following macro:

```
15 SLOM_BUILD_MEASUREMENT(StereoMeasurement, 4,
16   ((StereoCamera, cam))
17   ((SE3, left2world))
18   ,
19   ((vec3, cornerInWorld))
20   ((vec2, leftMeas))
21   ((vec2, rightMeas))
22 )
```

The first parameter is the name of the measurement, then follows its dimensionality (in this case 4 as we measure two 2D feature positions). The third parameter is a list of dependent variables, again in a double-parenthesized list of types and names, and fourth a list of extra user data which is treated by SLoM as arbitrary constant data that is made available to the measurement model (see below) but not otherwise looked at. Note that the dependent variables need to be manifolds, whereas the extra user data can be of arbitrary types.

Next, we implement the measurement model, i.e., a function that, in terms of (1), computes $f_i(Y_i) \boxminus z_i$. By line 23 SLoM automatically generates the necessary function header, requiring the result to be stored in a real-valued vector `ret` of the dimension passed as the second parameter to `SLOM_BUILD_MEASUREMENT` above. It can easily be assigned using the `=` operator or Eigen's comma initializer as is done here.

```
23 SLOM_IMPLEMENT_MEASUREMENT(StereoMeasurement, ret){
24   vec3 cornerInLeft = left2world->inverse() * cornerInWorld;
25   vec3 cornerInRight = cam->left2right * cornerInLeft;
26   ret << cam->left.sensor2image(cornerInLeft) - leftMeas,
27          cam->right.sensor2image(cornerInRight) - rightMeas;
28 }
```

In the implementation of the measurements, variables and user data members can be accessed by name. Variables need to be dereferenced by the `*` or `->` operator. The measurement function transforms the checkerboard coordinates to

the left (line 24) and right (line 25) camera's coordinate system, then applies the appropriate camera projections and subtracts the measured coordinates (lines 26 and 27).

Next, we define the gravitation measurement. This will depend on the accelerometer's bias, the orientation of the accelerometer with respect to the left camera and the position of the left camera in the world. We also assume that we do not know the exact amount of gravitational acceleration, so we add another variable estimating g. Finally, we include the measured acceleration as data member.

```
29 SLOM_BUILD_MEASUREMENT(Gravity, 3,
30   ((vec3, acc_bias))
31   ((SE3, left2world))
32   ((SO3, left2imu))
33   ((Scalar, g))
34   ,
35   ((vec3, acc))
36 )
```

Implementing the measurement requires transforming the local measurement to world coordinates and subtracting the expected gravity:

```
37 SLOM_IMPLEMENT_MEASUREMENT(Gravity, ret){
38   vec3 acc_world = *left2world * left2imu->inverse() * (acc - *acc_bias);
39   ret = (acc_world - *g * vec3::UnitZ());
40 }
```

2.3 Insertion of Variables and Measurements

Once all variables and measurements have been defined, we can collect data and insert it into an `Estimator`. For brevity, we omit the process of obtaining the data and finding initial guesses for the camera poses. The templated `VarID` class is a handle to the actual variable, required to declare measurements and needed to obtain their content after optimization.

```
41   Estimator est; // Estimator, responsible for data management & optimization
42   std::vector<vec3> calib_points;  // Known calibration point positions
43
44   // Variables shared by multiple measurements:
45   VarID<StereoCamera> cam = est.insertRV(StereoCamera());
46   VarID<SO3> left2imu     = est.insertRV(SO3());
47   VarID<vec3> acc_bias    = est.insertRV(vec3());
48   VarID<scalar> grav      = est.insertRV(scalar(9.81));
49
50   for(int i=0; i<num_images; ++i){
51     SE3 pose;
52     std::vector<std::pair<vec2, vec2> > point_measurements;
53     // collect image points, get an initial guess for the left camera pose
54     /* LEFT OUT FOR BREVITY */
55
56     // left2world is only local, since we do not need its optimized value
57     VarID<SE3> left2world = est.insertRV(pose);
58     for(int j=0; j<num_points; ++j){
59       est.insertMeasurement(StereoMeasurement(cam, left2world,
60           calib_points[j],
61           point_measurements[j].first, point_measurements[j].second));
62     }
63     vec3 acc;   //insert gravitation measurement:
64     double acc_sigma = 1e-3;
```

```
65      est.insertMeasurement(Gravity(acc_bias, left2world, left2imu, grav, acc),
66                            SLOM::StandardDeviation(acc_sigma));
67    }
```

When inserting measurement, the last parameter (line 66) describes the uncertainty of the measurement. One can choose from multiple ways to represent uncertainty: either as the covariance (`SLOM::Covariance`), as the standard deviation (`SLOM::StandardDeviation`), as the information matrix, i.e., the inverse of the covariance (`SLOM::InvCovariance`) or as the inverse of the standard deviation (`SLOM::InvStandardDeviation`). Each method accepts either a single scalar (as in the example) or a vector describing a diagonal matrix. Covariances can also be passed as full (symmetric) matrix and standard deviations as lower triangular matrix, being the Cholesky factor of the covariance. If the uncertainty parameter is omitted, SLoM assumes the measurement has unit covariance (line 61).

2.4 Optimization and Obtaining the Results

Now that we have inserted all measurements, we can call the `optimize` function of the `Estimator` and read out the optimized values by dereferencing the `VarID` of each variable using the `*` or `->` operator. Note that MTK manifolds overload the streaming operators, so one can easily stream the result into files or to the console.

```
68    for(int i=0; i<100; ++i){
69      est.optimizeStep();
70    }
71    std::cout << "Camera intrinsics " << *cam << "\nleft2imu " << *left2imu
72      << "\nGravity " << *grav << "\nAccelerometer Bias " << *acc_bias << "\n";
```

3 Applying SLoM to Your Own Optimization Problems

Basic data types describing vectors, orientations and transformations are readily implemented. Therefore, SLoM requires no definition of custom manifolds to solve problems such as pose relation and landmark based simultaneous localization and mapping (SLAM). Furthermore, it is easy to combine multiple basic manifolds to combined variables using the `MTK_BUILD_MANIFOLD` macro, which covers more complex problems such as multi-camera bundle adjustment or humanoid robot calibration [21].

With all variables defined, applying SLoM to arbitrary optimization problems boils down to defining custom measurement functions. As shown in the tutorial (e.g., lines 15 to 28) this only requires listing the involved variables and required measurement data, and to implement the actual measurement function.

However, domain knowledge is crucial, i.e., you will first want to understand your problem very well and then expose as much of its structure to SLoM as possible. Usually, this means that whenever possible you want to operate on raw measured data. E.g., if you work with visual markers detected by a camera you want to pass each raw pixel coordinate to SLoM without any pre-processing. It

is the job of the measurement function to predict these raw coordinates using a model that is as accurate as possible, i.e., if you intend to correct for radial distortion, you will want to add this as a parameter to the model and take it into account in the measurement function. You do not want to adjust the measured pixel coordinates by pre-processing. Similarly, you should always pass each individual raw measurement to SLoM – do not combine several measurements into one datum, i.e., use individual pixel coordinates as opposed to a some sort of score value you have pre-computed for an entire image.

Afterwards, acquiring the data for measurements is usually more laborious than feeding the data into SLoM and running the optimizer. The user does not need to care about the tedious task of data management – each variable is identified by a type-safe `VarID`, used to initialize measurements and to access the optimized data.

Most calibration problems require an initial guess not too far from the optimum, which if no explicit (approximating) formula exists must be obtained by measuring by hand or preferably re-using a previous calibration.

Also, especially in calibration problems, care has to be taken that the overall problem does not degenerate. This can happen if too few measurements are acquired or if the system contains unapparent gauge freedoms, i.e., non-determined degrees of freedom, such as a non-determined "free floating" start pose in SLAM. In this case one calls the problem rank-deficient and the standard solver will abort immediately. Another hint for a degenerating problem is if the residual sum of squares (i.e., $\|F(X)\|^2_\Sigma$ from (4)) keeps growing during the optimization process. This can be observed by looking at the results of `est.getRSS()` after each call to `optimizeStep()`.

Besides adding more measurements which constrain the variables better, a possible solution is to fix some variables, e.g.,

```
VarID<vec3> bias = est.insertRV(bias, false); // do not optimize bias
```

then use `bias` as shown in the tutorial and optionally, after optimization "un-fix" it by

```
est.optimizeRV(bias, true); // "un-fix" bias
est.optimizeStep();         // call optimizer again
```

Another solution is to use a more robust optimization algorithm such as Levenberg-Marquardt [15] instead of the default Gauss-Newton:

```
est.changeAlgorithm(new SLOM::LevenbergMarquardt());
```

If for certain variables, there is no measurement depending on it, i.e., the variable is not observable, SLoM automatically raises a warning. This will also make the standard solver fail immediately. Again, this can be circumvented by adding measurements depending on this variable or by fixing this variable (see above).

By alternately inserting variables/measurements and running the optimizer, SLoM is capable to perform online model fitting. In [10], we showed that using this approach it is possible to run full bundle adjustment online for a certain

time. However, with increasing number of variables and measurements, the optimization time increases as well thus losing real-time capability, eventually. SLoM allows to fix variables and remove measurements (e.g., following a sliding window approach) to reduce the computation time by reducing the problem size – at the cost of a less optimal result.

4 What's with those Manifolds?

Although surprising at first, you have already seen the most important property of manifolds – you have hardly taken notice of them at all. This is because we apply a little trick: Locally, a manifold behaves much like an $\mathbb{R}^n$ vector while globally its (topological) structure can be more complex. We can define [11] two encapsulation operators $\boxplus$ ("boxplus") and $\boxminus$ ("boxminus") for a manifold $\mathcal{S}$:

$$\boxplus : \mathcal{S} \times \mathbb{R}^n \to \mathcal{S}, \qquad \boxminus : \mathcal{S} \times \mathcal{S} \to \mathbb{R}^n. \tag{5}$$

Here, $\boxplus$ adds a small perturbation vector to a manifold variable, while $\boxminus$ is its inverse, calculating the difference between two manifold variables (This is the same $\boxminus$ we left unexplained in §1). If the perturbations are small, the manifold suddenly looks like $\mathbb{R}^n$ if you use $\boxplus/\boxminus$ instead of the familiar vector $+/-$.

Why does this matter? In its original form, least squares optimization only works with $\mathbb{R}^n$ variables. However, if we want to deal with real-world spatial data this is insufficient since, most notably, there is no singularity-free representation of 3D orientations with just three parameters – there are always some orientations where a small change in orientation requires a very large change in the representation. Overparameterized (e.g., $\mathbb{R}^4$) representations are not a solution either since the optimizer would try to make use of the extra degrees of freedom which do not actually exist.

We overcome this dilemma by having the least squares optimizer only operate on the local vectorized view established by the $\boxplus$ and $\boxminus$ operators. It does not know about the global structure and still does the right thing as we show in [11] which also gives mathematical details, proofs and experiments.

Although the interested reader is encouraged to read the referenced paper, this is not absolutely necessary even though the above only provides a vague idea of manifolds, $\boxplus$ and $\boxminus$. This is due to another trick: It happens that the Cartesian product of two (and by induction arbitrarily many) manifolds yields another, compound manifold. More importantly, the $\boxplus/\boxminus$ operators of that compound manifold are simply the operators of its components (or sub-manifolds) applied component-wise. This allows `MTK_BUILD_MANIFOLD` to generate compound manifold classes automatically for you if you have implementations of the sub-manifolds. Luckily, MTK comes with implementations of all practically relevant manifold primitives: vectors ($\mathbb{R}^n$), 2D orientations ($SO(2)$), 3D orientations ($SO(3)$) and the (less commonly used) unit sphere (S^2). Thus, based on these, you can build virtually all specialized manifolds you will need without an in-depth understanding of manifolds, $\boxplus$ and $\boxminus$.

5 Related Work

Least squares optimization for model fitting goes back to Gauss [5]. Thus, we will focus on key ideas and readily available tools/libraries here.

The intuition that lead to the use of manifolds in the SLAM community goes back to Triggs et al. [19, p. 6–7] who suggested handling non-vector space states such as orientations by using a global over-representation R with local perturbations δR using a minimal parameterization. The first to explicitly use manifold properties were Ude [20] for least squares optimization and Kraft [13] for Kalman filtering. The extension to arbitrary manifolds and the de-coupling of the optimizer implementation from the variable representation was done in earlier work by the authors [11].

5.1 Problem Specific Frameworks

There are many tools and libraries for specific problems such as camera calibration [2,4,18], bundle adjustment [16] or pose adjustment [17,8,7,6]. They have in common that they are specialized to a specific task and are not easily extensible to more complex problems such as when the sensor setup is non-standard, sensors are added or measurements need to be combined differently.

5.2 Generic Frameworks

Other than SLoM, to our knowledge, there are currently three C++-frameworks aimed at solving arbitrary optimization problems. Namely g^{2}o [14], optimized for fast batch optimization and iSAM [12], focusing on online or incremental optimization problems and the very recently released Ceres solver [3].

While g^{2}o is slightly faster than SLoM, as it exploits the structure of the problem better, iSAM turned out to be slower than SLoM in a recent contest.[1] Very preliminary tests – re-implementing the examples of Ceres using SLoM – showed that Ceres is about as fast as SLoM. Be aware though that, especially for incremental/online optimization, a fair comparison of computation times is difficult, e.g., due to the fact that there is always a trade-off between precision and computation time. However, a more thorough comparison is beyond the scope of this paper.

When handling manifolds, g^{2}o adapts a similar concept as SLoM, whereas iSAM directly maps its variables to vector spaces, thus causing problems when singularities occur. Ceres adds the concept of "local parameterization" which essentially does the same as SLoM's $\boxplus$ operator, however this is not bound to types but has to be handled individually when registering variables.

Both g^{2}o and iSAM provide basic variable-classes such as vectors, orientations and poses, as well as simple measurements, such as pose relations and basic

[1] For comparisions of computation times see: `http://slameval.willowgarage.com/workshop/talks/2011-RSS-SLAM-Evaluation.pdf`, pp. 17–20.

landmark measurements. This makes them easy to adapt to SLAM problems requiring just these types of measurements. However, if new measurements need to be defined or if variables shall be combined from sub-variables, the user has to implement this from scratch, conforming the internal requirements of the respective framework. As of now, Ceres entirely works on pointers to scalars but provides some convenience functions for often required operations. This requires the user to manually keep track of variable indices. Previous generations of our calibration tools relied on similar techniques and this turned out to be a frequent source of errors which ultimately led to the development of SLoM.

We believe that SLoM's library support for constructing arbitrary compound variable types (manifolds) from primitives and the fact that measurement functions can be directly implemented as C++ functions are the key distinguishing features of SLoM. We try to elaborate this by giving a brief comparison of the different APIs in the appendix.

Beyond the realm of C++, the MTKM framework [21] ports the idea of the SLoM to Matlab and is as such able to solve the same problems. MTKM avoids some syntactic noise which the C++ implementation of SLoM requires. However, due to the poor performance of Matlab's object orientated programming extensions MTKM runs slower by orders of magnitude. We believe that MTKM provides a valuable alternative to users more accustomed to Matlab and not requiring real-time performance (or when working with smaller problem sizes).

6 Conclusions and Future Work

We showed that using SLoM it is possible to solve calibration problems, requiring only basic knowledge of statistics and 3D geometry. Using the same approach most model fitting problems arising in practice can be optimized. By using the $\boxplus/\boxminus$ operators SLoM can solve optimization problems involving manifolds without the user needing to bother about their internal structure.

In future work, we intend to simplify the way measurements are defined even more. Using C++11 features, most importantly variadic templates [1, §14.5.3], it will be possible to define measurements by only defining the actual measurement function:

```
VectorNd measurement(const Var1& v1, const Var2& v2, const Dat1& d1) {
  // do some calculations using v1, v2, ...
  return VectorNd(...);
}
```

and register them to the `Estimator` as such:

```
  VarID<Var1> v1 = est.registerRV(Var1());  // register variables
  VarID<Var2> v2 = est.registerRV(Var2());  // ...
  est.registerMeasurement(measurement, v1, v2, Dat1(...));
```

Furthermore, there is a proof-of-concept implementation allowing the user to easily supply symbolical derivations. By providing derivations of basic functions this will become almost as simple as implementing the actual measurement. Finally, there is on-going research improving SLoM's performance, especially its speed doing online optimization.

Appendix: API-Comparison of SLoM with iSAM and g²o

We will give a short comparison of the user interfaces of SLoM vs. iSAM and g²o. We compare the code required to implement and initialize constraints between a 2D pose and a 2D landmark. We believe that for more complicated examples (such as the calibration example described in this tutorial) the differences will be the same or even more evident. For Ceres we did not find a readily implemented 2D SLAM example, from which we could excerpt code to make a fair comparison.

Note that the three libraries use different nomenclatures for the same concepts. What is called "variables" and "measurements" in SLoM is called "nodes" and "factors" in iSAM and "vertices" and "edges" in g²o.

We start looking at the definition of measurements. Listing 1 shows how a factor is defined in iSAM. Compared to that Listing 5 does the same for SLoM, but avoids much boiler-plate code by the use of macros. In g²o (Listing 3) some initialization requirements are automatically done by the `BaseBinaryEdge` base class, but most of the initialization is done manually when inserting the edge (Listing 4). In contrast, iSAM (Listing 2) and SLoM (Listing 6) essentially provide initialization in a single statement. Uncertainty information must be provided as inverse covariance in g²o and inverse standard deviation (or "square root information matrix" as they call it) in iSAM. SLoM is more flexible at this point and even allows to omit passing uncertainty information if the measurement already has unit-covariance. We give a more detailed discussion for each code fragment in its caption.

Listing 1. Defining a 2D landmark measurement in iSAM. Notice line 9 requires manually registering nodes to the factor, and lines 12 and 13 require manually converting the generic nodes to the respective types. The implementation of the actual measurement function starts at line 14. Code excerpted from `https://svn.csail.mit.edu/isam/include/isam/slam2d.h`, LGPL v2.1.

```
1  class Pose2d_Point2d_Factor : public FactorT<Point2d> {
2    Pose2d_Node* _pose;
3    Point2d_Node* _point;
4  public:
5    Pose2d_Point2d_Factor(Pose2d_Node* pose, Point2d_Node* point,
6        const Point2d& measure, const Noise& noise)
7      : FactorT<Point2d>("Pose2d_Point2d_Factor", 2, noise, measure),
8        _pose(pose), _point(point) {
9      _nodes.resize(2); _nodes[0] = pose; _nodes[1] = point;
10   }
11   Eigen::VectorXd basic_error(Selector s = LINPOINT) const {
12     Pose2d po(_nodes[0]->vector(s));
13     Point2d pt(_nodes[1]->vector(s));
14     Point2d p = po.transform_to(pt);
15     Eigen::VectorXd predicted = p.vector();
16     return (predicted - _measure.vector());
17   }
18 };
```

Listing 2. Insert a 2D landmark observation using iSAM. Uncertainty information is passed as square root information matrix `noise2`. Code excerpted from `https://svn.csail.mit.edu/isam/examples/example.cpp`, LGPL v2.1.

```
19 Pose2d_Point2d_Factor* measurement =
20   new Pose2d_Point2d_Factor(pose_nodes[1], new_point_node, measure, noise2);
21 slam.add_factor(measurement);
```

Listing 3. Defining a 2D landmark measurement in g^{2}o (excerpt from tutorial code `https://svn.openslam.org/data/svn/g2o/trunk/g2o/examples/tutorial_slam2d/edge_se2_pointxy.h`, LGPL v3). The `BaseBinaryEdge` class implements most boiler-plate code required for implementing binary edges (i.e., between two vertices). Still lines 8 and 9 require manually converting the generic vertices to the respective types. Only the last line defines the actual measurement function.

```
1  class EdgeSE2PointXY
2    : public BaseBinaryEdge<2, Vector2d, VertexSE2, VertexPointXY>
3  {
4  public:
5    EdgeSE2PointXY() : BaseBinaryEdge<2,Vector2d,VertexSE2,VertexPointXY>(){}
6    void computeError()
7    {
8      const VertexSE2* v1 = static_cast<const VertexSE2*>(_vertices[0]);
9      const VertexPointXY* l2=static_cast<const VertexPointXY*>(_vertices[1]);
10     _error = (v1->estimate().inverse() * l2->estimate()) - _measurement;
11   }
12 };
```

Listing 4. Insert a 2D landmark observation using g^{2}o. Note that `optimizer.vertex()` does not do type checking, therefore possible errors are detected only at runtime. The `setMeasurement` function is typesafe, however it allows to pass only a single observation object, thus more complex observations need to be combined manually. Code excerpted from `https://svn.openslam.org/data/svn/g2o/trunk/g2o/examples/tutorial_slam2d/tutorial_slam2d.cpp`, LGPL v3.

```
13   EdgeSE2PointXY* landmarkObservation = new EdgeSE2PointXY;
14   landmarkObservation->vertices()[0] = optimizer.vertex(p.id);
15   landmarkObservation->vertices()[1] = optimizer.vertex(l->id);
16   landmarkObservation->setMeasurement(observation);
17   landmarkObservation->setInverseMeasurement(-1.*observation);
18   landmarkObservation->setInformation(information);
19   optimizer.addEdge(landmarkObservation);
```

Listing 5. Declare and implement a 2D landmark observation using SLoM. Note that all variables can be referenced by name and are strongly typed.

```
1 SLOM_BUILD_MEASUREMENT(LM_observation, 2, ((Pose, pose)) ((vec2, landmark)),
2     ((vec2, coords ))
3 )
4 SLOM_IMPLEMENT_MEASUREMENT(LM_observation, ret) {
5   ret = ( pose->inverse() * (*landmark) ) - coords;
6 }
```

Listing 6. Insert a 2D landmark observation using SLoM. The measurement is constructed by an auto-generated constructor. All arguments of the constructor are strongly typed, leading to compile-time errors if wrong types are passed. Passing covariance information is optional (by default unit-covariance is assumed).

```
7    est.insertMeasurement(
8        LM_observation( poseID, landmark[id], observation),
9        SLOM::InvCovariance(information)
10   );
```

References

1. Working draft, standard for programming language C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2798.pdf`, 2008.
2. OpenCV. `http://opencv.willowgarage.com`, 2012.
3. S. Agarwal and K. Mierle. Ceres solver. `http://code.google.com/p/ceres-solver/`, 2012.
4. J.-Y. Bouguet. Camera calibration toolbox for Matlab. `http://www.vision.caltech.edu/bouguetj/calib_doc/`.
5. C. F. Gauss. Theoria combinationis observationum erroribus minimis obnoxiae. *Commentationes societatis regiae scientiarum Gottingensis recentiores*, 5:6–93, 1821.
6. M. K. Grimes, D. Anguelov, and Y. LeCun. Hybrid hessians for flexible optimization of pose graphs. In *Proc. International Conference on Intelligent Robots and Systems (IROS)*, 2010.
7. G. Grisetti, R. Kümmerle, C. Stachniss, U. Frese, and C. Hertzberg. Hierarchical optimization on manifolds for online 2d and 3d mapping. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2010.
8. G. Grisetti, C. Stachniss, Slawomir Grzonka, and Wolfram Burgard. A tree parameterization for efficiently computing maximum likelihood maps using gradient descent. In *Robotics Science and Systems*, 2007.
9. G. Guennebaud, B. Jacob, et al. Eigen 3.1. `http://eigen.tuxfamily.org`, 2012.
10. C. Hertzberg, R. Wagner, O. Birbach, T. Hammer, and U. Frese. Experiences in building a visual SLAM system from open source components. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Shanghai, China*, pages 2644–2651, May 2011.
11. C. Hertzberg, R. Wagner, U. Frese, and L. Schröder. Integrating generic sensor fusion algorithms with sound state representations through encapsulation of manifolds. *Information Fusion*, 2011. **in press**.
12. M. Kaess, A. Ranganathan, and F. Dellaert. iSAM: Incremental smoothing and mapping. *IEEE Trans. on Robotics (TRO)*, 24(6):1365–1378, Dec 2008.
13. E. Kraft. A quaternion-based unscented Kalman filter for orientation tracking. In *Proceedings of the Sixth International Conference of Information Fusion*, volume 1, pages 47–54, 2003.
14. R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. g2o: A general framework for graph optimization. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2011.
15. Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *The Quarterly of Applied Mathematics*, (2):164–168, 1944.

16. M.I.A. Lourakis and A.A. Argyros. The design and implementation of a generic sparse bundle adjustment software package based on the Levenberg-Marquardt algorithm. Technical Report 340, Institute of Computer Science - FORTH, Heraklion, Greece, Aug. 2004. Available from `http://www.ics.forth.gr/~lourakis/sba`.
17. E. Olson, J. Leonard, and S. Teller. Fast iterative optimization of pose graphs with poor initial estimates. In *Proceedings 2006 IEEE International Conference on Robotics and Automation*, pages 2262–2269, 2006.
18. T. Svoboda, D. Martinec, and T. Pajdla. A convenient multi-camera self-calibration for virtual environments. *PRESENCE: Teleoperators and Virtual Environments*, 14(4):407–422, August 2005.
19. W. Triggs, P. McLauchlan, R. Hartley, and A. Fitzgibbon. Bundle adjustment – A modern synthesis. In W. Triggs, A. Zisserman, and R. Szeliski, editors, *Vision Algorithms: Theory and Practice*, LNCS, pages 298–375. Springer Verlag, 2000.
20. A. Ude. Nonlinear least squares optimisation of unit quaternion functions for pose estimation from corresponding features. In *Proc. of the Int. Conf. on Pattern Recognition*, 1998.
21. R. Wagner, O. Birbach, and U. Frese. Rapid development of manifold-based graph optimization systems for multi-sensor calibration and SLAM. In *Proceeding of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011), San Francisco, California*, 2011.