g^{2}o: A general Framework for (Hyper) Graph Optimization

Giorgio Grisetti, Rainer Kümmerle, Hauke Strasdat, Kurt Konolige
email: `{grisetti,kuemmerl}@informatik.uni-freiburg.de`
`strasdat@gmail.com konolige@willowgarage.com`

November 6, 2016

In this document we describe a C++ framework for performing the optimization of nonlinear least squares problems that can be embedded as a graph or in a hyper-graph. A hyper-graph is an extension of a graph where an edge can connect multiple nodes and not only two. Several problems in robotics and in computer vision require to find the optimum of an error function with respect of a set of parameters. Examples include, popular applications like SLAM and Bundle adjustment.

In the literature, many approaches have been proposed to address this class of problems. The naive implementation of standard methods, like Levenberg-Marquardt or Gauss-Newton can lead to acceptable results for most applications, when the correct parameterization is chosen. However, to achieve the maximum performances substantial efforts might be required.

g^{2}o stands for General (Hyper) Graph Optimization. The purposes of this framework are the following:

- To provide an easy-to-extend and easy-to-use general library for graph optimization that can be easily applied to different problems,
- To provide people who want to understand SLAM or BA with an easy-to-read implementation that focuses on the relevant details of the problem specification.
- Achieve state-of-the-art performances, while being as general as possible.

In the remainder of this document we will first characterize the (hyper) graph-embeddable problems, and we will give an introduction to their solution via the popular Levenberg-Marquardt or Gauss-Newton algorithms implemented in this library. Subsequently, we will describe the high-level behavior of the library, and the basic structures. Finally, we will introduce how to implement 2D SLAM as a simple example.

This document is not a replacement for the in-line documentation. Instead, it is a digest to help the user/reader to read/browse and extend the code.

Please cite this when using g^{2}o:
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 (ICRA). Shanghai, China, May 2011.

1 (Hyper)Graph-Embeddable Optimization Problems

A least squares minimization problem can be described by the following equation:

$$\mathbf{F}(\mathbf{x}) = \sum_{k\in\mathcal{C}} \underbrace{\mathbf{e}_k(\mathbf{x}_k,\mathbf{z}_k)^T \mathbf{\Omega}_k \mathbf{e}_k(\mathbf{x}_k,\mathbf{z}_k)}_{\mathbf{F}_k} \quad (1)$$

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}}\, \mathbf{F}(\mathbf{x}). \quad (2)$$

Figure 1: This example illustrates how to represent an objective function by a hyper-graph. Here we illustrate a portion of a small SLAM problem [?]. In this example we assume that where the measurement functions are governed by some unknown calibration parameters $\mathbf{K}$. The robot poses are represented by the variables $\mathbf{p}_{1:n}$. These variables are connected by constraints $\mathbf{z}_{ij}$ depicted by the square boxes. The constraints arise, for instance, by matching nearby laser scans *in the laser reference frame.* The relation between a laser match and a robot pose, however, depends on the position of the sensor on the robot, which is modeled by the calibration parameters $\mathbf{K}$. Conversely, subsequent robot poses are connected by binary constraints $\mathbf{u}_k$ arising from odometry measurements. These measurements are made in the frame of the robot mobile base.

Here

- $\mathbf{x} = (\mathbf{x}_1^T, \ldots, \mathbf{x}_n^T)^T$ is a vector of parameters, where each $\mathbf{x}_i$ represents a generic parameter block.
- $\mathbf{x}_k = (\mathbf{x}_{k_1}^T, \ldots, \mathbf{x}_{k_q}^T)^T \subset (\mathbf{x}_1^T, \ldots, \mathbf{x}_n^T)^T$ is the subset of the parameters involved in the k^{th} constraint.
- $\mathbf{z}_k$ and $\mathbf{\Omega}_k$ represent respectively the mean and the information matrix of a constraint relating the parameters in $\mathbf{x}_k$.
- $\mathbf{e}_k(\mathbf{x}_k \mathbf{z}_k)$ is a vector error function that measures how well the parameter blocks in $\mathbf{x}_k$ satisfy the constraint $\mathbf{z}_k$. It is $\mathbf{0}$ when $\mathbf{x}_k$ and $\mathbf{x}_j$ perfectly match the constraint. As an example, if one has a measurement function $\hat{\mathbf{z}}_k = \mathbf{h}_k(\mathbf{x}_k)$ that generates a synthetic measurement $\hat{\mathbf{z}}_k$ given an actual configuration of the nodes in $\mathbf{x}_k$. A straightforward error function would then be $\mathbf{e}(\mathbf{x}_k, \mathbf{z}_k) = \mathbf{h}_k(\mathbf{x}_k) - \mathbf{z}_k$.

For simplicity of notation, in the rest of this paper we will encode the measurement in the indices of the error function:

$$\mathbf{e}_k(\mathbf{x}_k, \mathbf{z}_k) \overset{\text{def.}}{=} \mathbf{e}_k(\mathbf{x}_k) \overset{\text{def.}}{=} \mathbf{e}_k(\mathbf{x}). \tag{3}$$

Note that each parameter block and each error function can span over a different space. A problem in this form can be effectively represented by a directed hyper-graph. A node i of the graph represents the parameter block $\mathbf{x}_i \in \mathbf{x}_k$ and an hyper-edge among the nodes $\mathbf{x}_i \in \mathbf{x}_k$ represents a constraint involving all nodes in $\mathbf{x}_k$. In case the hyper edges have size 2, the hyper-graph becomes an ordinary graph. Figure 1 shows an example of mapping between a hyper-graph and an objective function.

2 Least Squares Optimization

If a good initial guess $\breve{\mathbf{x}}$ of the parameters is known, a numerical solution of Eq. 2 can be obtained by using the popular Gauss-Newton or Levenberg-Marquardt algorithms [?, §15.5]. The idea is to approximate the error function by its first order Taylor expansion around the current initial guess $\breve{\mathbf{x}}$

$$\begin{aligned} \mathbf{e}_k(\breve{\mathbf{x}}_k + \mathbf{\Delta x}_k) &= \mathbf{e}_k(\breve{\mathbf{x}} + \mathbf{\Delta x}) && (4)\\ &\simeq \mathbf{e}_k + \mathbf{J}_k \mathbf{\Delta x}. && (5) \end{aligned}$$

Here $\mathbf{J}_k$ is the Jacobian of $\mathbf{e}_k(\mathbf{x})$ computed in $\breve{\mathbf{x}}$ and $\mathbf{e}_k \overset{\text{def.}}{=} \mathbf{e}_k(\breve{\mathbf{x}})$. Substituting Eq. 5 in the error terms $\mathbf{F}_k$ of Eq. 1, we obtain

$$\begin{aligned} &\mathbf{F}_k(\breve{\mathbf{x}} + \mathbf{\Delta x}) && (6)\\ &= \mathbf{e}_k(\breve{\mathbf{x}} + \mathbf{\Delta x})^T \mathbf{\Omega}_k \mathbf{e}_k(\breve{\mathbf{x}} + \mathbf{\Delta x}) && (7)\\ &\simeq (\mathbf{e}_k + \mathbf{J}_k \mathbf{\Delta x})^T \mathbf{\Omega}_k (\mathbf{e}_k + \mathbf{J}_k \mathbf{\Delta x}) && (8)\\ &= \underbrace{\mathbf{e}_k^T \mathbf{\Omega}_k \mathbf{e}_k}_{\mathrm{c}_k} + 2 \underbrace{\mathbf{e}_k^T \mathbf{\Omega}_k \mathbf{J}_k}_{\mathbf{b}_k} \mathbf{\Delta x} + \mathbf{\Delta x}^T \underbrace{\mathbf{J}_k^T \mathbf{\Omega}_k \mathbf{J}_k}_{\mathbf{H}_k} \mathbf{\Delta x} && (9)\\ &= \mathrm{c}_k + 2\mathbf{b}_k \mathbf{\Delta x} + \mathbf{\Delta x}^T \mathbf{H}_k \mathbf{\Delta x} && (10) \end{aligned}$$

With this local approximation, we can rewrite the function $\mathbf{F}(\mathbf{x})$ given in Eq. 1 as

$$
\begin{aligned}
\mathbf{F}(\breve{\mathbf{x}} + \mathbf{\Delta x}) &= \sum_{k \in \mathcal{C}} \mathbf{F}_k(\breve{\mathbf{x}} + \mathbf{\Delta x}) && (11)\\
&\simeq \sum_{k \in \mathcal{C}} \mathrm{c}_k + 2\mathbf{b}_k \mathbf{\Delta x} + \mathbf{\Delta x}^T \mathbf{H}_k \mathbf{\Delta x} && (12)\\
&= \mathrm{c} + 2\mathbf{b}^T \mathbf{\Delta x} + \mathbf{\Delta x}^T \mathbf{H} \mathbf{\Delta x}. && (13)
\end{aligned}
$$

The quadratic form in Eq. 13 is obtained from Eq. 12 by setting $\mathrm{c} = \sum \mathrm{c}_k$, $\mathbf{b} = \sum \mathbf{b}_k$ and $\mathbf{H} = \sum \mathbf{H}_k$. It can be minimized in $\mathbf{\Delta x}$ by solving the linear system

$$
\mathbf{H}\,\mathbf{\Delta x}^* = -\mathbf{b}. \qquad (14)
$$

The matrix $\mathbf{H}$ is the information matrix of the system and is sparse by construction, having non-zeros only between blocks connected by a constraint. Its number of non-zero blocks is twice the number of constrains plus the number of nodes. This allows to solve Eq. 14 with efficient approaches like sparse Cholesky factorization or Preconditioned Conjugate Gradients (PCG). An highly efficient implementation of sparse Cholesky factorization can be found in publicly available packages like CSparse [?] or CHOLMOD [?]. The linearized solution is then obtained by adding to the initial guess the computed increments

$$
\mathbf{x}^* = \breve{\mathbf{x}} + \mathbf{\Delta x}^*. \qquad (15)
$$

The popular Gauss-Newton algorithm iterates the linearization in Eq. 13, the solution in Eq. 14 and the update step in Eq. 15. In every iteration, the previous solution is used as linearization point and as initial guess.

The Levenberg-Marquardt (LM) algorithm is a nonlinear variant to Gauss-Newton that introduces a damping factor and backup actions to control the convergence. Instead of solving directly Eq. 14 LM solves a damped version of it

$$
(\mathbf{H} + \lambda \mathbf{I})\,\mathbf{\Delta x}^* = -\mathbf{b}. \qquad (16)
$$

Here λ is a damping factor: the larger λ is the smaller are the $\mathbf{\Delta x}$. This is useful to control the step size in case of non-linear surfaces. The idea behind the LM algorithm is to dynamically control the damping factor. At each iteration the error of the new configuration is monitored. If the new error is lower than the previous one, lambda is decreased for the next iteration. Otherwise, the solution is reverted and lambda is increased. For a more detailed explanation of the LM algorithm implemented in our package we refer to [?].

The procedures described above are a general approach to multivariate function minimization. The general approach, however, assumes that the space of parameters $\mathbf{x}$ is Euclidean, which is not valid for several problems like SLAM or bundle adjustment. This may lead to sub-optimal solutions. In the remainder of this section we discuss first the general solution when the space of the parameters is Euclidean, and subsequently we extend this solution to more general non-euclidean spaces.

3 Considerations about the Structure of the Linearized System

According to Eq. 13, the matrix $\mathbf{H}$ and the vector $\mathbf{b}$ are obtained by summing up a set of matrices and vectors, one for every constraint. If we set $\mathbf{b}_k = \mathbf{J}_k^T \mathbf{\Omega}_k \mathbf{e}_k$ and $\mathbf{H}_k = \mathbf{J}_k^T \mathbf{\Omega}_k \mathbf{J}_k$ we can rewrite $\mathbf{H}$ and $\mathbf{b}$ as

$$
\begin{aligned}
\mathbf{b} &= \sum_{k \in \mathcal{C}} \mathbf{b}_{ij} && (17)\\
\mathbf{H} &= \sum_{k \in \mathcal{C}} \mathbf{H}_{ij}. && (18)
\end{aligned}
$$

Every constraint will contribute to the system with an addend term. The *structure* of this addend depends on the Jacobian of the error function. Since the error function of a constraint depends only on the values of the nodes $\mathbf{x}_i \in \mathbf{x}_k$, the Jacobian in Eq. 5 has the following form:

$$
\mathbf{J}_k = \left(\mathbf{0} \cdots \mathbf{0}\; \mathbf{J}_{k_1}\; \cdots\; \mathbf{J}_{k_i}\; \cdots \mathbf{0}\; \cdots\; \mathbf{J}_{k_q} \mathbf{0} \cdots \mathbf{0}\right). \qquad (19)
$$

Here $\mathbf{J}_{k_i} = \frac{\partial \mathbf{e}(\mathbf{x}_k)}{\partial \mathbf{x}_{k_i}}$ are the derivatives of the error function with respect to the nodes connected by the k^{th} hyper-edge, with respect to the parameter block $\mathbf{x}_{k_i} \in \mathbf{x}_k$.

From Eq. 9 we obtain the following structure for the block matrix $\mathbf{H}_{ij}$:

$$\mathbf{H}_k = \begin{pmatrix} \ddots & & & & & & \\ & \mathbf{J}_{k_1}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_1} & \cdots & \mathbf{J}_{k_1}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_i} & \cdots & \mathbf{J}_{k_1}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_q} & \\ & \vdots & & \vdots & & \vdots & \\ & \mathbf{J}_{k_i}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_1} & \cdots & \mathbf{J}_{k_i}^T \boldsymbol{\Omega}_k \mathbf{B}_{k_i} & \cdots & \mathbf{J}_{k_i}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_q} & \\ & \vdots & & \vdots & & \vdots & \\ & \mathbf{J}_{k_q}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_1} & \cdots & \mathbf{J}_{k_q}^T \boldsymbol{\Omega}_k \mathbf{B}_{k_i} & \cdots & \mathbf{J}_{k_q}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_q} & \\ & & & & & & \ddots \end{pmatrix} \tag{20}$$

$$\mathbf{b}_k = \begin{pmatrix} \vdots \\ \mathbf{J}_{k_1} \boldsymbol{\Omega}_k \mathbf{e}_k \\ \vdots \\ \mathbf{J}_{k_i}^T \boldsymbol{\Omega}_k \mathbf{e}_k \\ \vdots \\ \mathbf{J}_{k_q}^T \boldsymbol{\Omega}_k \mathbf{e}_k \\ \vdots \end{pmatrix} \tag{21}$$

For simplicity of notation we omitted the zero blocks. The reader might notice that the block structure of the matrix $\mathbf{H}$ is the adjacency matrix of the hyper graph. Additionally the Hessian $\mathbf{H}$ is a symmetric matrix, since all the $\mathbf{H}_k$ are symmetric. A single hyper-edge connecting q vertices will introduce q^2 non zero blocks in the Hessian, in correspondence of each pair $\left\langle \mathbf{x}_{k_i}, \mathbf{x}_{k_j} \right\rangle$, of nodes connected.

4 Least Squares on Manifold

To deal with parameter blocks that span over a non-Euclidean spaces, it is common to apply the error minimization on a manifold. A manifold is a mathematical space that is not necessarily Euclidean on a global scale, but can be seen as Euclidean on a local scale [?].

For example, in the context of SLAM problem, each parameter block $\mathbf{x}_i$ consists of a translation vector $\mathbf{t}_i$ and a rotational component α_i. The translation $\mathbf{t}_i$ clearly forms a Euclidean space. In contrast to that, the rotational components α_i span over the non-Euclidean 2D or 3D rotation group $SO(2)$ or $SO(3)$. To avoid singularities, these spaces are usually described in an over-parameterized way, e.g., by rotation matrices or quaternions. Directly applying Eq. 15 to these over-parameterized representations breaks the constraints induced by the over-parameterization. The over-parameterization results in additional degrees of freedom and thus introduces errors in the solution. To overcome this problem, one can use a minimal representation for the rotation (like Euler angles in 3D). This, however, is then subject to singularities.

An alternative idea is to consider the underlying space as a manifold and to define an operator $\boxplus$ that maps a local variation $\mathbf{\Delta x}$ in the Euclidean space to a variation on the manifold, $\mathbf{\Delta x} \mapsto \mathbf{x} \boxplus \mathbf{\Delta x}$. We refer the reader to [?, §1.3] for more mathematical details. With this operator, a new error function can be defined as

$$\breve{\mathbf{e}}_k(\mathbf{\Delta}\tilde{\mathbf{x}}_k) \overset{\text{def.}}{=} \mathbf{e}_k(\breve{\mathbf{x}}_k \boxplus \mathbf{\Delta}\tilde{\mathbf{x}}_k) \tag{22}$$

$$= \mathbf{e}_k(\breve{\mathbf{x}} \boxplus \mathbf{\Delta}\tilde{\mathbf{x}}) \simeq \breve{\mathbf{e}}_k + \tilde{\mathbf{J}}_k \mathbf{\Delta}\tilde{\mathbf{x}}, \tag{23}$$

where $\breve{\mathbf{x}}$ spans over the original over-parameterized space, for instance quaternions. The term $\mathbf{\Delta}\tilde{\mathbf{x}}$ is a small increment around the original position $\breve{\mathbf{x}}$ and is expressed in a minimal representation. A common choice for $SO(3)$ is to use the vector part of the unit quaternion. In more detail, one can represent the increments $\mathbf{\Delta}\tilde{\mathbf{x}}$ as 6D vectors $\mathbf{\Delta}\tilde{\mathbf{x}}^T = (\mathbf{\Delta}\tilde{\mathbf{t}}^T \; \tilde{\mathbf{q}}^T)$, where $\mathbf{\Delta}\tilde{\mathbf{t}}$ denotes the translation and

$\tilde{\mathbf{q}}^T = (\Delta q_x \, \Delta q_y \, \Delta q_z)^T$ is the vector part of the unit quaternion representing the 3D rotation. Conversely, $\breve{\mathbf{x}}^T = (\breve{\mathbf{t}}^T \, \breve{\mathbf{q}}^T)$ uses a quaternion $\breve{\mathbf{q}}$ to encode the rotational part. Thus, the operator $\boxplus$ can be expressed by first converting $\mathbf{\Delta}\tilde{\mathbf{q}}$ to a full quaternion $\mathbf{\Delta q}$ and then applying the transformation $\mathbf{\Delta x}^T = (\mathbf{\Delta t}^T \, \mathbf{\Delta q}^T)$ to $\breve{\mathbf{x}}$. In the equations describing the error minimization, these operations can nicely be encapsulated by the $\boxplus$ operator. The Jacobian $\tilde{\mathbf{J}}_k$ can be expressed by

$$\tilde{\mathbf{J}}_k \quad = \quad \left.\frac{\partial \mathbf{e}_k(\breve{\mathbf{x}} \boxplus \mathbf{\Delta}\tilde{\mathbf{x}})}{\partial \mathbf{\Delta}\tilde{\mathbf{x}}}\right|_{\mathbf{\Delta}\tilde{\mathbf{x}}=\mathbf{0}} . \tag{24}$$

Since in the previous equation $\breve{\mathbf{e}}$ depends only on $\mathbf{\Delta}\tilde{\mathbf{x}}_{k_i} \in \mathbf{\Delta}\tilde{\mathbf{x}}_k$ we can further expand it as follows:

$$\begin{aligned}
\tilde{\mathbf{J}}_k \quad &= \quad \left.\frac{\partial \mathbf{e}_k(\breve{\mathbf{x}} \boxplus \mathbf{\Delta}\tilde{\mathbf{x}})}{\partial \mathbf{\Delta}\tilde{\mathbf{x}}}\right|_{\mathbf{\Delta}\tilde{\mathbf{x}}=\mathbf{0}} && (25)\\
&= \quad \left(\mathbf{0} \cdots \mathbf{0} \, \tilde{\mathbf{J}}_{k_1} \, \cdots \, \tilde{\mathbf{J}}_{k_i} \, \cdots \mathbf{0} \, \cdots \, \tilde{\mathbf{J}}_{k_q} \mathbf{0} \cdots \mathbf{0}\right). && (26)
\end{aligned}$$

With a straightforward extension of notation, we set

$$\tilde{\mathbf{J}}_{k_i} = \left.\frac{\partial \mathbf{e}_k(\breve{\mathbf{x}} \boxplus \mathbf{\Delta}\tilde{\mathbf{x}})}{\partial \mathbf{\Delta}\tilde{\mathbf{x}}_{k_i}}\right|_{\mathbf{\Delta}\tilde{\mathbf{x}}=\mathbf{0}} \tag{27}$$

With a straightforward extension of the notation, we can insert Eq. 23 in Eq. 8 and Eq. 11. This leads to the following increments:

$$\tilde{\mathbf{H}} \, \mathbf{\Delta}\tilde{\mathbf{x}}^* \quad = \quad -\tilde{\mathbf{b}}. \tag{28}$$

Since the increments $\mathbf{\Delta}\tilde{\mathbf{x}}^*$ are computed in the local Euclidean surroundings of the initial guess $\breve{\mathbf{x}}$, they need to be re-mapped into the original redundant space by the $\boxplus$ operator. Accordingly, the update rule of Eq. 15 becomes

$$\mathbf{x}^* \quad = \quad \breve{\mathbf{x}} \boxplus \mathbf{\Delta}\tilde{\mathbf{x}}^*. \tag{29}$$

In summary, formalizing the minimization problem on a manifold consists of first computing a set of increments in a local Euclidean approximation around the initial guess by Eq. 28, and second accumulating the increments in the global non-Euclidean space by Eq. 29. Note that the linear system computed on a manifold representation has the same structure like the linear system computed on an Euclidean space. One can easily derive a manifold version of a graph minimization from a non-manifold version, only by defining an $\boxplus$ operator and its Jacobian $\tilde{\mathbf{J}}_{k_i}$ w.r.t. the corresponding parameter block. In g^{2}o we provide tools for numerically computing the Jacobians on the manifold space. This requires the user to implement the error function and the $\boxplus$ operator only. As a design choice, we do not address the non-manifold case since it is already contained in the manifold one. However, to achieve the maximum performances and accuracy we recommend the user to implement analytic Jacobians, once the system is functioning with the numeric ones.

5 Robust Least Squares

Optionally, the least squares optimization can be robustified. Note, that the error terms in Eq. 1 have the following form:

$$\mathbf{F}_k = \mathbf{e}_k^T \Omega_k \mathbf{e}_k = \rho_2\left(\sqrt{\mathbf{e}_k^T \Omega_k \mathbf{e}_k}\right) \quad \text{with} \quad \rho_2(x) := x^2. \tag{30}$$

Thus, the error vector $\mathbf{e}_k$ has quadratic influence on $\mathbf{F}$, so that a single potential outlier would have major negative impact. In order be more outlier robust, the quadratic error function ρ_2 can be replaced by a more robust cost function which weighs large errors less. In g^{2}o, the Huber cost function ρ_H can be used

$$\rho_H(x) := \begin{cases} x^2 & \text{if } |x| < b \\ 2b|x| - b^2 & \text{else,} \end{cases} \tag{31}$$

Figure 2: Class diagram of g^{2}o.

which is quadratic for small $|x|$ but linear for large $|x|$. Compared to other, even more robust cost functions, the Huber kernel has to advantage that it is still convex and thus does not introduce new local minima in $\mathbf{F}$ [?, pp.616]. In practice, we do not need to modify Eq. 1. Instead, the following scheme is applied. First the error $\mathbf{e}_k$ is computed as usual. Then, $\mathbf{e}_k$ is replaced by a weighted version $w_k \mathbf{e}_k$ such that

$$(w_k \mathbf{e}_k)^T \Omega_k (w_k \mathbf{e}_k) = \rho_H \left(\sqrt{\mathbf{e}_k^T \Omega_k \mathbf{e}_k} \right). \tag{32}$$

Here, the weights w_k are calculated as follows

$$w_k = \frac{\sqrt{\rho_H \left(||\mathbf{e}_k||_\Omega \right)}}{||\mathbf{e}_k||_\Omega} \quad \text{with} \quad ||\mathbf{e}_k||_\Omega := \sqrt{\mathbf{e}_k^T \Omega_k \mathbf{e}_k}. \tag{33}$$

In g^{2}o, the user has fine-grained control and can enable/disable the robust cost function for each edge individually (see Section 6.2.2).

6 Library Overview

From the above sections it should be clear that a graph-optimization problem is entirely defined by:

- The types of the vertices in the graph (that are the parameters blocks $\{\mathbf{x}_i\}$. For each of those one has to specify:
  - the domain $\mathtt{Dom}(\mathbf{x}_i)$ of the internal parameterization,
  - the domain $\mathtt{Dom}(\mathbf{\Delta x}_i)$ of the increments $\mathbf{\Delta x}_i$,
  - $\boxplus : \mathtt{Dom}(\mathbf{x}_i) \times \mathtt{Dom}(\mathbf{\Delta x}_i) \rightarrow \mathtt{Dom}(\mathbf{x}_i)$ that applies the increment $\mathbf{\Delta x}_i$ to the previous solution $\mathbf{x}_i$.
- the error function for every type of hyper-edge $\mathbf{e}_k : \mathtt{Dom}(\mathbf{\Delta x}_{k_1}) \times \mathtt{Dom}(\mathbf{\Delta x}_{k_2}) \times \cdots \times \mathtt{Dom}(\mathbf{\Delta x}_{k_q}) \rightarrow \mathtt{Dom}(\mathbf{z}_k)$ that should be zero when the perturbated estimate $\mathbf{x}_k \boxplus \mathbf{\Delta x}_k$ perfectly satisfies the constraint $\mathbf{z}_k$.

By default the Jacobians are computed numerically by our framework. However to achieve the maximum performances in a specific implementation one can specify the Jacobian of the error functions and of the manifold operators.

In the reminder we will shortly discuss some basic concepts to use and extend g^{2}o. This documentation is by no means complete, but it is intended to help you browsing the automatically generated documentation. To better visualize the interplay of the components of g^{2}o we refer to the class diagram of Figure 2.

6.1 Representation of an Optimization Problem

All in all our system utilizes a generic hyper-graph structure to represent a problem instance (defined in `hyper_graph.h`). This generic hyper graph is specialized to represent an optimization problem by the class `OptimizableGraph`, defined in `optimizable_graph.h`. Within the `OptimizableGraph` the inner classes `OptimizableGraph::Vertex` and `OptimizableGraph::Edge` are used to represent generic hyper edges and hyper vertices. Whereas the specific implementation might be done by directly extending these classes, we provided a template specialization that implements automatically most of the methods that are mandatory for the system to work.

These classes are `BaseVertex` and `BaseUnaryEdge`, `BaseBinaryEdge` and `BaseMultiEdge`.

```
#include "g2o/core/factory.h"

namespace g2o {
  G2O_REGISTER_TYPE_GROUP(slam2d);

  G2O_REGISTER_TYPE(VERTEX_SE2, VertexSE2);
  G2O_REGISTER_TYPE(VERTEX_XY, VertexPointXY);

  // ...
}
```

Listing 1: Registering types by a constructor from a library

`BaseVertex` templatizes the dimension of a parameter block $\mathbf{x}_i$ and of the corresponding manifold $\mathbf{\Delta x}_i$, thus it can use blocks of memory whose layout is known at compile-time (means efficiency). Furthermore, it implements some mapping operators to store the Hessian and the parameter blocks of the linearized problem, and a stack of previous values that can be used to save/restore parts of the graph. The method `oplusImpl(double* v)` that applies the perturbation $\mathbf{\Delta x}_i$ represented by `v`, to the member variable `_estimate` should be implemented. This is the $\boxplus$ operator. Additionally, `setToOriginImpl()` that should set the internal state of the vertex to $\mathbf{0}$ has to specified.

`BaseUnaryEdge` is a template class to model a unary hyper-edge, which can be used to represent a prior. It offers for free the calculation of the Jacobians, via an implementation of the `linearizeOplus` method. It requires to specify the types of the (single) vertex $\mathbf{x}_i$, and type and dimension of the error $\mathbf{e}(\mathbf{x}_k)$ as template parameters. The function `computeError` that stores the result of the error $\mathbf{e}(\mathbf{x}_k)$ in the member `Eigen::Matrix _error` should be implemented.

`BaseBinaryEdge` is a template class that models a binary constraint, namely an error function in the form $\mathbf{e}_k(\mathbf{x}_{k_1}, \mathbf{x}_{k_2})$. It offers the same facilities of `BaseUnaryEdge`, and it requires to specify the following template parameters: the type of the nodes $\mathbf{x}_{k_1}$ and $\mathbf{x}_{k_2}$ and the type and the dimension of the measurement. Again, it implements the numeric Jacobians via a default implementation of the `linearizeOplus` method. Again, the `computeError` should be implemented in a derived class.

`BaseMultiEdge` is a template class that models a multi-vertex constraint in the form of $\mathbf{e}_k(\mathbf{x}_{k_1}, \mathbf{x}_{k_2}, \ldots, \mathbf{x}_{k_q})$. It offers the same facilities of the types above, and it requires to specify only the type and dimension of the measurement as template parameters. The specialized class should take care of resizing the connected vertices to the correct size q. This class relies on a dynamic memory, since too many parameters are unknown, and if you need of an efficient implementation for a specific problem you can program it yourself. Numeric Jacobian comes for free, but you should implement the `computeError` in a derived class, as usual

In short, all you need to do to define a new problem instance is to derive a set of classes from those above listed, one for each type of parameter block and one for each type of (hyper)edge. Always try to derive from the class which does the most work for you. If you want to have a look at a simple example look at `vertex_se2` and `edge_se2`. Those two types define a simple 2D graph SLAM problem, like the one described in many SLAM papers.

Of course, for every type you construct you should define also the `read` and `write` functions to read and write your data to a stream. Finally, once you define a new type, to enable the loading and the saving of the new type you should "register" it to a factory. This is easily done by assigning a string tag to a new type, via the `registerType` function. This should be called once before all files are loaded.

To this end, g^{2}o provides an easy macro to carry out the registration of the class to the factory. See Listing 1 for an example, the full example can be found in `types_slam2d.cpp`. The first parameter given to the macro `G2O_REGISTER_TYPE` specifies the tag under which a certain vertex / edge is known. g^{2}o will use this information while loading files and for saving the current graph into a file. In the example given in Listing 1 we register the tags `VERTEX_SE2` and `EDGE_SE2` with the classes `VertexSE2` and `EdgeSE2`, respectively.

Furthermore, the macro `G2O_REGISTER_TYPE_GROUP` allows to declare a type group. This is necessary if we use the factory to construct the types and we have to enforce that our code is linked to a specific type group. Otherwise the linker may drop our library, since we do not explicitly use any symbol provided by the library containing our type. Declaring the usage of a specific type library and hence enforcing the linking is done by the macro called `G2O_USE_TYPE_GROUP`.

6.2 Construction and Representation of the Linearized Problem

The construction and the solution can be separated into individual steps which are iterated.

- Initialization of the optimization (only before the first iteration).
- Computing the error vector for each constraint.
- Linearize each constraint.
- Build the linear system.
- Updating the Levenberg-Marquardt damping factor.

Within the following sections we will describe the steps.

6.2.1 Initialization

The class `SparseOptimizer` offers several methods to initialize the underlying data structure. The methods `initializeOptimization()` either takes a subset of vertices or a subset of edges which will be considered for the next optimization runs. Additionally, all vertices and edges can be considered for optimization. We refer to the vertices and edges currently considered as *active* vertices and edges, respectively.

Within the initialization procedure, the optimizer assigns a temporary index to each active vertex. This temporary index corresponds to the block column / row of the vertex in the Hessian. Some of the vertices might need to be kept fixed during the optimization, to resolve arbitrary degrees of freedom (gauge freedom). This can be done by setting the `_fixed` attribute of a vertex.

6.2.2 Compute error

The `computeActiveErrors()` function takes the current estimate of the active vertices and for each active edge calls `computeError()` for computing the current error vector. Using the base edge classes described in Section 6.1 the error should be cached in the member variable `_error`.

If `robustKernel()` is set to true for a particular active edge, `robustifyError()` is called and `_error` is robustified as described in Section 5.

6.2.3 Linearizing the system

Each active edge is linearized by calling its `linearizeOplus()` function. Again the Jacobians can be cached by member variables provided by the templatized base classes described in Section 6.1. If the `linearizeOplus()` function is not re-implemented the Jacobian will be computed numerically as follows:

$$\tilde{\mathbf{J}}_k^{\bullet l} = \frac{1}{2\delta}\left(\mathbf{e}_k(\mathbf{x}_k \boxplus \delta \mathbf{1}_l) - \mathbf{e}_k(\mathbf{x}_k \boxplus -\delta \mathbf{1}_l)\right), \tag{34}$$

where $\delta > 0$ is a small constant (10^{-9} in our implementation) and $\mathbf{1}_l$ is the unit vector along dimension l. Note that we only store and calculate the non-zero entries of $\tilde{\mathbf{J}}_k$ that have not been fixed during the initialization.

6.2.4 Building the system

For each active edge the addend term for Eq. 18 is computed by multiplying the corresponding blocks of the Jacobians and the information matrix of the edge. The addend term is calculated in each edge by calling `constructQuadraticForm()`.

6.2.5 Updating Levenberg-Marquardt

As illustrated in Eq. 16 the Levenberg-Marquardt algorithm requires updates to the linear system. However, only the elements along the main diagonal need to be modified. To this end, the methods `updateLevenbergSystem(double lambda)` and `recoverSystem(double lambda)` of the `Solver` class apply the modifications by respectively adding or subtracting λ along the main diagonal of $\mathbf{H}$.

6.3 Solvers

A central component of these least-squares approaches is the solution of the linear system $\tilde{\mathbf{H}}\,\boldsymbol{\Delta}\tilde{\mathbf{x}}^* = -\tilde{\mathbf{b}}$. To this end there are several approaches available, some of them exploit the known structure of certain problems and perform intermediate reductions of the system, like by applying the Schur complement to a subset of variables. In g^{2}o we do not select any particular solver, but we rely on external libraries. To this end, we decouple these *structural* operations (like the Schur complement) from the solution of the linear system.

The construction of the linear problem from the Jacobian matrices and the error vectors in the hyper-graph elements are controlled by a so-called `Solver` class. To use a specific factorization of the system, the user has to extend the `Solver` class, and to implement the virtual functions. Namely a solver should be able to extract from an hyper-graph the linear system, and to return a solution. This is done in several steps: at the beginning of the optimization the function `initializeStructure` is called, to allocate the necessary memory that will be overwritten in the subsequent operations. This is possible since the structure of the system does not change between iterations. Then the user should provide means to access to the increment vector $\boldsymbol{\Delta}\tilde{\mathbf{x}}$ and $\tilde{\mathbf{b}}$, via the functions `b()` and `x()`. To support Levenberg-Marquardt one should also implement a function to perturb the Hessian with the $\lambda\mathbf{I}$ term. This function is called `setLambda(double lambda)` and needs to be implemented by the specific solver.

We provide a templatized implementation of the solver class, the `BlockSolver<>` that stores the linear system in a `SparseBlockMatrix<>` class. The `BlockSolver<>` implements also the Schur complement, and relies on another abstract class, the `LinearSolver` to solve the system. An implementation of the linear solver does the actual work of solving the (reduced) linear system, and has to implement a few methods. In this release of g^{2}o we provide linear solvers that use respectively preconditioned gradient descent, CSparse, and CHOLMOD.

6.4 Actions

To the extent of g^{2}o, the entities stored in a hyper-graph have a pure mathematical meaning. They either represent variables to be optimized (vertices), or they encode optimization constraints. However, in general these variables are usually related to more "concrete" objects, like laser scans, robot poses, camera parameters and so on. Some variable type may support only a subset of feasible operations. For instance it is possible to "draw" a robot pose, but it is not possible to "draw" the calibration parameters. More in general we cannot know a priori the kind of operations that will be supported by the user types of g^{2}o. However, we want to design a set of tools and of functions that rely on certain operations. These include, for instance viewers, or functions to save/load the graph in a specific format.

A possibility to do this would be to "overload" the base classes of the hyper-graph elements (vertices and edges) with many virtual functions, one for each of the functionality we want to support. This is of course not elegant, because we would need to patch the base classes with the new function every time something new is added. Another possibility would be to make use of the multiple inheritance of C++, and to define an abstract "drawable" object, on which the viewer operates. This solution is a bit better, however we cannot have more than one "drawing" function for each object.

The solution used in g^{2}o consists in creating a library of function objects that operate on the elements (vertices or edges) of the graph. One of these function objects is identified by a function name and by a type on which it operates. These function objects can be registered into an action library. Once these objects are loaded in the action library it is possible to call them on a graph. These functionalities are defined in `hyper_graph_action.h`. It is common to register and create the actions when defining the types for the edges and the vertices. You can see many examples in `types_*/*.h`.

Figure 3: Graphical interface to g^{2}o. The GUI allows to select different suitable optimizers and perform the optimization.

7 g^{2}o Tools

g^{2}o comes with two tools which allow to process data stored in files. The data can be loaded from a file and stored again after processing. In the following we will give a brief introduction to these tools, namely a command line interface and a graphical user interface.

7.1 g^{2}o Command Line Interface

`g2o` is the command line interface included in g^{2}o. It allows to optimize graphs stored in files and save the result back to a file. This allows a fast prototyping of optimization problems, as it is only required to implement the new types or solvers. The g^{2}o distribution includes a data folder which comprises some data files on which `g2o_cli` can be applied.

7.2 g^{2}o Viewer

The Graphical User Interface depicted in Figure 3 allows to visualize the optimization problem. Additionally, the various parameters of the algorithms can be controlled.

7.3 g^{2}o incremental

g^{2}o includes an experimental binary for performing optimization in an incremental fashion, i.e., optimizing after inserting one or several nodes along with their measurements. In this case, g^{2}o performs ranke updates on the Hessian matrix to update the linear system. Please see the `README` in the `g2o_incremental` sub-folder for additional information.

Example for the Manhattan3500 dataset:

```
g2o_incremental -i manhattanOlson3500.g2o
```

7.4 Plug-in Architecture

Both tools support the loading of types and optimization algorithms at run-time from dynamic libraries. This is realized as follows. The tools load from the libs folder all libraries matching "*_types_*" and "_solver_*" to register types and optimization algorithms, respectively. We assume that by loading the libraries the types and the algorithms register via their respective constructors to the system. Listing 1 shows how to register types to the system and Listing 2 is an example, which shows how to register an optimization algorithm via the plug-in architecture.

For loading dynamic library containing types or optimization algorithms, we support two different methods:

- The tools recognize the command line switch `-typeslib` and `-solverlib` to load a specific library.
- You may specify the environment variables `G2O_TYPES_DIR` and `G2O_SOLVER_DIR` which are scanned at start and libraries matching "*_types_*" and "_solver_*" are automatically loaded.

8 2D SLAM: An Example

SLAM is a well known problem in robotics and this acronym stands for "Simultaneous Localization And Mapping". The problem can be stated as follows: given a moving robot equipped with some sensors, we want to estimate both the map and the pose of the robot in the environment from the sensor measurements. Usually, the sensors can be classified in exteroceptive or proprioceptive. An exteroceptive sensor is a device that measures quantities relative to the environment where the robot moves. Examples of these sensors can be cameras that acquire an image of the world at a particular location, laser scanners

```
class PCGSolverCreator : public AbstractOptimizationAlgorithmCreator
{
  public:
    PCGSolverCreator(const OptimizationAlgorithmProperty& p) :
        AbstractOptimizationAlgorithmCreator(p) {}
    virtual OptimizationAlgorithm* construct()
    {
      // create the optimization algorithm
      // see g2o/solver_pcg/solver_pcg.cpp for the details
    }
};

G2O_REGISTER_OPTIMIZATION_LIBRARY(pcg);

G2O_REGISTER_OPTIMIZATION_ALGORITHM(gn_pcg, new PCGSolverCreator(
    OptimizationAlgorithmProperty("gn_pcg", "Gauss-Newton: PCG solver using block-Jacobi
     pre-conditioner (variable blocksize)", "PCG", false, Eigen::Dynamic, Eigen::Dynamic
    )));

G2O_REGISTER_OPTIMIZATION_ALGORITHM(gn_pcg3_2, new PCGSolverCreator(
    OptimizationAlgorithmProperty("gn_pcg3_2", "Gauss-Newton: PCG solver using block-
    Jacobi pre-conditioner (fixed blocksize)", "PCG", true, 3, 2)));

// ...
```

Listing 2: Registering solvers by a constructor from a library

that measure a set of distances around the robot or accelerometers in presence of gravity that measure the gravity vector or GPS that derive a pose estimate by observing the constellation of known satellites. In contrast, proprioceptive sensors measure the change of the robot's state (the position), relative to the previous robot position. Example include odometers, that measure the relative movement of the robot between two time steps or gyroscopes. In traditional approaches to SLAM, like EKF these two sensors play a substantially different role in the system. The proprioceptive measurements are used to evolve a set of state variables, while the exteroceptive measurements are used to correct these estimates, by feeding back the measurement errors. This is not the case of smoothing methods (like the ones that can be implemented with g^{2}o), where all measurements are treated in a substantially similar manner.

A complete solution to SLAM is typically rather complex and involves processing raw sensor data and determining correspondences between previously seen parts of the environment and actual measurements (data association). Describing a complete solution to the problem is out of the scope of this document. However, in the reminder we will present a simplified but meaningful version of the problem that contains all the relevant elements and that is well suited to be implemented with g^{2}o.

The scenario is a robot moving on a plane. The robot is equipped with an odometry sensor that is able to measure the relative movement of the robot between two time frames and of a "landmark" sensor that is able to measure the position of some environment landmarks nearby the robot *in the robot reference frame*. One could implement this landmark detector, for instance, by extracting corners from a laser scan or by detecting the position of relevant features from a stereo image pair. A simplification that we make in this section is that the landmarks are uniquely identifiable. In other words whenever the robot sees a landmark, it can tell if it is a new one or if it has already seen it and when.

Clearly both odometers and landmark sensors are affected by noise. In principle, if the odometry would not be affected by noise one could reconstruct the trajectory of the robot simply by chaining the odometry measurements. However, this is not the case and integrating the odometry leads to an increasing positioning error that becomes evident when the robot reenters a known region. In a similar way, if the robot would have unlimited perception range, it could acquire all the map in one shot and the position could be retrieved by simple geometric constructions. Again this is not the case and the robot perceives the position of the landmarks that are located within a maximum range. These measurements are affected by a noise, that usually increases with the distance of a landmark from the robot.

In the remainder of this section we will walk through all essential steps that are required to characterize a problem within g^{2}o. These are:

- identification of the state variables $\mathbf{x}_i$ and of their domain,

Figure 4: Graphical representation of a SLAM process. The vertices of the graph, depicted with circular nodes, denote either robot poses $\mathbf{x}_*^{\mathrm{s}}$ or landmarks $\mathbf{x}_*^{\mathrm{l}}$. The measurement of a landmark from a robot pose is captured by a constraints $\mathbf{z}_*^{\mathrm{l}}$ and odometry measurements connecting subsequent robot poses are modeled by the constraints $\mathbf{z}_*^{\mathrm{s}}$.

- characterization of the constraints and identification of the graph structure,
- choice of the parameterization for the increments $\boldsymbol{\Delta}\tilde{\mathbf{x}}_i$, and definition of the $\boxplus$ operator.
- construction of the error functions $e_k(\mathbf{x}_k)$.

8.1 Identification of the State Variables

Figure 4 illustrates a fragment of a SLAM graph. The robot positions are denoted by the nodes $\mathbf{x}_t^{\mathrm{s}}$, while the landmarks are denoted by the nodes $\mathbf{x}_i^{\mathrm{l}}$. We assume that our landmark sensor is able to detect only the 2D pose of a landmark, but not its orientation. In other words the landmarks "live" in $\Re^2$. Conversely, the robot poses are parameterized by the robot location $x-y$ on the plane and its orientation θ, thus they belong to the group of 2D transformations $SE(2)$. More formally, the nodes of a 2D SLAM graph are of two types

- Robot positions $\mathbf{x}_t^{\mathrm{s}} = (x_t^{\mathrm{s}}\ y_t^{\mathrm{s}}\ \theta_t^{\mathrm{s}})^T \in SE(2)$
- Landmark positions $\mathbf{x}_i^{\mathrm{l}} = (x_i^{\mathrm{l}}\ y_i^{\mathrm{l}})^T \in \Re^2$

8.2 Modeling of the Constraints

Two subsequent robot positions $\mathbf{x}_t^{\mathrm{s}}$ and $\mathbf{x}_{t+1}^{\mathrm{s}}$ are related by an odometry measurement, that represent the relative motion that brings the robot from $\mathbf{x}_t^{\mathrm{s}}$ to $\mathbf{x}_{t+1}^{\mathrm{s}}$ *measured* by the odometry. This measurement will be typically slightly different from the *real* transformation between the two pose because of the noise affecting the sensors. Being an odometry measurement an euclidean transformation it is also a member of $SE(2)$ group. Assuming the noise affecting the measurement being white and Gaussian, it can be modeled by an 3×3 symmetric positive definite information matrix. In real applications the entries of this matrix depend on the motion of the robot. i.e., the bigger the movement is the larger the uncertainty will be. Thus an odometry edge between the nodes $\mathbf{x}_t^{\mathrm{s}}$ and $\mathbf{x}_{t+1}^{\mathrm{s}}$ consists of these two entities:

- $\mathbf{z}_{t,t+1}^{\mathrm{s}} \in SE(2)$ that represents the motion between the nodes and
- $\boldsymbol{\Omega}_{t,t+1}^{\mathrm{s}} \in \Re^{3\times 3}$ that represents the inverse covariance of the measurement, and thus is symmetric and positive definite.

If the robot senses a landmark $\mathbf{x}_i^{\mathrm{l}}$ from the location $\mathbf{x}_t^{\mathrm{s}}$, the corresponding measurement will be modeled by an edge going from the robot pose to the landmark. A measurement about the landmark consists in a point in the $x-y$ plane, perceived in the robot frame. Thus a landmark measurement lives in $\Re^2$ as the landmarks do. Again, under white Gaussian noise assumption, the noise can be modeled by its inverse covariance. Accordingly, an edge between a robot pose and a landmark is parametrized in this way:

- $\mathbf{z}_{t,i}^{\mathrm{l}} \in \Re^2$ that represents position of the landmark in the frame expressed by $\mathbf{x}_t^{\mathrm{s}}$ and
- $\boldsymbol{\Omega}_{t,i}^{\mathrm{l}} \in \Re^{2\times 2}$ that represents the inverse covariance of the measurement and is SPD.

8.3 Choice of the Parameterization for the Increments

So far, we defined most of the elements necessary to implement a 2D SLAM algorithm with g^{2}o. Namely we characterized the domains of the variables and the domains of the measurements. What remains to do is to define the error functions for the two kinds of edges in our system and to determine a (possibly smart) parameterization for the increments.

The landmark positions are parameterized in $\Re^2$, which is already an Euclidean space. Thus the increments $\mathbf{\Delta\tilde{x}}^{\mathrm{l}}_i$ can live in the same space and the $\boxplus$ operator can be safely chosen as the vector sum:

$$\mathbf{x}^{\mathrm{l}}_i \boxplus \mathbf{\Delta\tilde{x}}^{\mathrm{l}}_i \quad \doteq \quad \mathbf{x}^{\mathrm{l}}_i + \mathbf{\Delta\tilde{x}}^{\mathrm{l}}_i \tag{35}$$

The poses, conversely, live in the non euclidean space $SE(2)$. This space admits many parameterizations. Examples include: rotation matrix $\mathbf{R}(\theta)$ and translation vector $(x\ y)^T$ or angle θ and translation vector $(x\ y)^T$.

As a parameterization for the increments, we choose a minimal one, that is translation vector and angle. Having chosen this parameterization, we need to define the $\boxplus$ operator between a pose and a pose increment. One possible choice would be to treat the three scalar parameters x, y and θ of a pose as if they were a vector, and define the $\boxplus$ as the vector sum. There are many reasons why this is a poor choice. One of them is that the angles are not euclidean, and one would need to re-normalize them after every addition.

A better choice is to define the $\boxplus$ between a pose and a pose increment as the motion composition operator. Namely, given a robot pose $\mathbf{x}^s_t = (x\ y\ \theta)^T$ and an increment $\mathbf{\Delta\tilde{x}}^s_t = (\Delta x\ \Delta y\ \Delta\theta)^T$, the operator can be defined as follows:

$$\mathbf{x}^{\mathrm{s}}_t \boxplus \mathbf{\Delta\tilde{x}}^{\mathrm{s}}_t \quad \doteq \quad \begin{pmatrix} x + \Delta x \cos\theta - \Delta y \sin\theta \\ y + \Delta x \sin\theta + \Delta y \cos\theta \\ \mathrm{normAngle}(\theta + \Delta\theta) \end{pmatrix} \tag{36}$$

$$= \mathbf{x}^{\mathrm{s}}_t \oplus \mathbf{\Delta\tilde{x}}^{\mathrm{s}}_t. \tag{37}$$

In the previous equation we introduced the motion composition operator $\oplus$ Similarly to $\oplus$ there is the $\ominus$ operator that performs the opposite operation and is defined as follows:

$$\mathbf{x}^{\mathrm{s}}_a \ominus \mathbf{x}^{\mathrm{s}}_b \quad \doteq \quad \begin{pmatrix} (x_a - x_b)\cos\theta_b + (y_a - y_b)\sin\theta_b \\ -(x_a - x_b)\sin\theta_b + (y_a - y_b)\cos\theta_b \\ \mathrm{normAngle}(\theta_b - \theta_a) \end{pmatrix} \tag{38}$$

8.4 Design of the Error Functions

The last step in formalizing the problem is to design error functions $\mathbf{e}(bx_k)$ that are "reasonable". A common way do is to define a so-called *measurement* function $\mathbf{h}_k(\mathbf{x}_k)$ that "predicts" a measurement $\hat{\mathbf{z}}_k$, given the knowledge of the vertices in the set $\mathbf{x}_k$. Defining this function is usually rather easy, and can be done by directly implementing the error model. Subsequently, the error vector can be computed as the vector difference between the prediction $\hat{\mathbf{z}}_k$ and the real measurement. This is a general approach to construct error functions and it works when the space of the errors is locally euclidean around the origin of the measurement. If this is not the case one might want to replace the vector difference with some other operator which is more "regular".

We will now construct the error functions for the edges connecting a robot pose $\mathbf{x}^{\mathrm{s}}_t$ and a landmark $\mathbf{x}^{\mathrm{l}}_i$. The first step is to construct a measurement prediction $\mathbf{h}^{\mathrm{l}}_{t,i}(\mathbf{x}^{\mathrm{s}}_t, \mathbf{x}^{\mathrm{l}}_i)$ that computes a "virtual measurement". This virtual measurement is the position of the landmark $\mathbf{x}^{\mathrm{l}}_i$, seen from the robot position $\mathbf{x}^{\mathrm{s}}_t$. The equation for $\mathbf{h}^{\mathrm{l}}_{t,i}(\cdot)$is the following:

$$\mathbf{h}^{\mathrm{l}}_{t,i}(\mathbf{x}^{\mathrm{s}}_t, \mathbf{x}^{\mathrm{l}}_i) \quad \doteq \quad \begin{pmatrix} (x^{\mathrm{s}}_t - x_i)\cos\theta^{\mathrm{s}}_t + (y^{\mathrm{s}}_t - y_i)\sin\theta^{\mathrm{s}}_t \\ -(x^{\mathrm{s}}_t - x_i)\sin\theta^{\mathrm{s}}_t + (y^{\mathrm{s}}_t - y_i)\cos\theta^{\mathrm{s}}_t \end{pmatrix} \tag{39}$$

Since the landmarks live in an Euclidean space, it is reasonable to compute the error function as the normal vector difference. This leads to the following definition for the error functions of the landmarks.

$$\mathbf{e}^{\mathrm{l}}_{t,i}(\mathbf{x}^{\mathrm{s}}_t, \mathbf{x}^{\mathrm{l}}_i) \quad \doteq \quad \mathbf{z}_{t,i} - \mathbf{h}^{\mathrm{l}}_{t,i}(\mathbf{x}^{\mathrm{s}}_t, \mathbf{x}^{\mathrm{l}}_i). \tag{40}$$

In a similar way, we can define the error functions of an odometry edge connecting two robot poses $\mathbf{x}^{\mathrm{s}}_t$ and $\mathbf{x}^{\mathrm{s}}_{t+1}$. As stated before, an odometry measurement lives in $SE(2)$. By using the $\oplus$ operator we can write a synthetic measurement function:

$$\mathbf{h}^{\mathrm{s}}_{t,t+1}(\mathbf{x}^{\mathrm{s}}_t, \mathbf{x}^{\mathrm{s}}_{t+1}) \quad \doteq \quad \mathbf{x}^{\mathrm{s}}_{t+1} \ominus \mathbf{x}^{\mathrm{s}}_t. \tag{41}$$

In short this function returns the motion that brings the robot from $\mathbf{x}^{\mathrm{s}}_t$ to $\mathbf{x}^{\mathrm{s}}_{t+1}$, that is the "ideal" odometry. Once again the error can be obtained as a difference between the measurement and the prediction. However, since our measurements do not live in an euclidean space we can use the $\ominus$ instead of the vector difference.

$$\mathbf{e}^{\mathrm{s}}_{t,t+1}(\mathbf{x}^{\mathrm{s}}_t, \mathbf{x}^{\mathrm{s}}_{t+1}) \doteq \mathbf{z}_{t,t+1} \ominus \mathbf{h}^{\mathrm{s}}_{t,t+1}(\mathbf{x}^{\mathrm{s}}_t, \mathbf{x}^{\mathrm{s}}_{t+1}). \tag{42}$$

8.5 Putting things together

Here we summarize the relevant parts of the previous problem definition, and we get ready for the implementation.

Variable	**Symbol**	**Domain**	**Dimension**	**Parameterization of $\mathbf{\Delta x}$**	**$\boxplus$ operator**
Robot pose	$\mathbf{x}^{\mathrm{s}}_t$	$SE(2)$	3	$(\Delta x\ \Delta y\ \Delta\theta)$	$\mathbf{x}^{\mathrm{s}}_t \oplus \mathbf{\Delta x}^{\mathrm{s}}_t$
Landmark pose	$\mathbf{x}^{\mathrm{l}}_i$	$\Re^2$	2	$(\Delta x\ \Delta y)$	$\mathbf{x}^{\mathrm{l}}_i + \mathbf{\Delta x}^{\mathrm{l}}_i$

Measurement	**Symbol**	**Domain**	**Dimension**	**Set $\mathbf{x}_k$ of variables involved**	**error function**
Odometry	$= \mathbf{z}^{\mathrm{s}}_{t,t+1}$	$SE(2)$	3	$\{\mathbf{x}^{\mathrm{s}}_t, \mathbf{x}^{\mathrm{s}}_{t+1}\}$	Eq. 42
Landmark	$= \mathbf{z}^{\mathrm{l}}_{t,i}$	$\Re^2$	2	$\{\mathbf{x}^{\mathrm{s}}_t, \mathbf{x}^{\mathrm{l}}_i\}$	Eq. 40

The first thing we are going to do is to implement a class that represents elements of the $SE(2)$ group. We represent these elements internally by using the rotation matrix and the translation vector representation, via the types defined in `Eigen::Geometry`. Thus we define an `operator*(...)` that implements the motion composition operator $\oplus$, and an `inverse()` function that returns the inverse of a transformation. For convenience we also implement an `operator *` that transforms 2D points. To convert the elements from and to a minimal representation that utilizes an $Eigen :: Vector3d$ we define the methods `fromVector(...)` and `toVector(...)`. The constructor initializes this class as a point in the origin oriented at 0 degrees. Note that having a separate class for a group is not mandatory in g^{2}o, but makes the code much more readable and reusable. The corresponding C++ class is reported in Listing 3

Once we defined our nice $SE(2)$ group we are ready to implement the vertices. To this end we extend the `BaseVertex<>` class, and we derive the classes `VertexSE2` to represent a robot pose and `VertexPointXY` to represent a point landmark in the plane. The class definition is reported in Listing 4 The pose-vertex extends a template specialization of `BaseVertex<>`. We should say to g^{2}o that the internal type has dimension 3 and that the estimate is of type `SE2`. This means that the member `_estimate` of a `VertexSE2` is of type `SE2`. Then all we need to do is to redefine the methods `setToOriginImpl()` that resets the estimate to a known configuration, and the method `\oplusImpl(double*)`. The method should apply an increment, expressed in the increment parameterization (that is a vector $(\Delta x\ \Delta y\ \Delta\theta)^t$ to the current estimate. To do this, we first convert the vector passed as argument into an `SE(2)`, then we multiply this increment at the right of the previous estimate. After that we should implement the read and write functions to a stream, but this is straight-forward and you can look it up yourself in the code.

The next step is to implement a vertex to describe a landmark position. Since the landmarks are parameterized in $\Re^2$, we do not need to define any group for that, and we use directly the vector classes defined in Eigen. This class is reported in Listing 5.

Now we are done with the vertices. We should go for the edges. Since both edges are binary edges we, extend the class `BaseBinaryEdge<>`. To represent an odometry edge (see Listing 6) that connects two `VertexSE2`, we need to extend `BaseBinaryEdge<>`, specialized with the types of the connected vertices (the order matters), where the measurement itself is represented by an `SE2` that has dimension 3. The second template parameter is the one used for the member variable `_measurement`. The second step is to construct an error function, by redefining the `computeError()` method. The `computeError()` should put the error vector in a member variable `error` that has type `Eigen::Vector<double,3>`. Here the 3 comes from the template parameter that specifies the dimension. Again, the read and write functions can be looked up in the code. Now we are done with this edge. The Jacobians are computed numerically by g^{2}o. However, if you want to speed up the execution of your code after everything works, you are warmly invited to redefine the `linearizeOplus` method.

The last thing that remains to do is to define a class to represent a landmark measurement. This

```
class SE2 {
  public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
    SE2():_R(0),_t(0,0){}

    SE2(double x, double y, double theta):_R(theta),_t(x,y){}

    SE2 operator * (const SE2& tr2) const{
      SE2 result(*this);
      result._t += _R*tr2._t;
      result._R.angle()+= tr2._R.angle();
      result._R.angle()=normalize_theta(result._R.angle());
      return result;
    }

    Vector2d operator * (const Vector2d& v2) const{
      Vector2d result(*this);
      result._t = _t + _R*tr2._t;
      return result;
    }

    SE2 inverse() const{
      SE2 ret;
      ret._R=_R.inverse();
      ret._R.angle()=normalize_theta(ret._R.angle());
      ret._t=ret._R*(_t*-1.);
      return ret;
    }

    void fromVector (const Vector3d& v){
      *this=SE2(v[0], v[1], v[2]);
    }

    Vector3d toVector() const {
      Vector3d ret;
      for (int i=0; i<3; i++){
        ret(i)=(*this)[i];
      }
      return ret;
    }

  protected:
    Rotation2Dd _R;
    Vector2d _t;
};
```

Listing 3: Helper class to represent $SE(2)$.

```
class VertexSE2 : public BaseVertex<3, SE2>
{
  public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
    VertexSE2();

    virtual void setToOriginImpl() {
      _estimate=SE2();
    }

    virtual void oplusImpl(double* update)
    {
      SE2 up(update[0], update[1], update[2]);
      _estimate = _estimate * up;
    }

    virtual bool read(std::istream& is);
    virtual bool write(std::ostream& os) const;

};
```

Listing 4: Vertex representing a 2D robot pose

```
class VertexPointXY : public BaseVertex<2, Eigen::Vector2d>
{
  public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
      VertexPointXY();

    virtual void setToOriginImpl() {
      _estimate.setZero();
    }

    virtual void oplusImpl(double* update)
    {
      _estimate[0] += update[0];
      _estimate[1] += update[1];
    }

    virtual bool read(std::istream& is);
    virtual bool write(std::ostream& os) const;
};
```

Listing 5: Vertex representing a 2D landmark

```
class EdgeSE2 : public BaseBinaryEdge<3, SE2, VertexSE2, VertexSE2>
{
  public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
      EdgeSE2();

    void computeError()
    {
      const VertexSE2* v1 = static_cast<const VertexSE2*>(_vertices[0]);
      const VertexSE2* v2 = static_cast<const VertexSE2*>(_vertices[1]);
      SE2 delta = _measurement.inverse() * (v1->estimate().inverse()*v2->estimate());
      _error = delta.toVector();
    }
    virtual bool read(std::istream& is);
    virtual bool write(std::ostream& os) const;
};
```

Listing 6: Edge connecting two robot poses, for example, the odometry of the robot.

```
class EdgeSE2PointXY : public BaseBinaryEdge<2, Eigen::Vector2d, VertexSE2,
    VertexPointXY>
{
  public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
      EdgeSE2PointXY();

    void computeError()
    {
      const VertexSE2* v1 = static_cast<const VertexSE2*>(_vertices[0]);
      const VertexPointXY* l2 = static_cast<const VertexPointXY*>(_vertices[1]);
      _error = (v1->estimate().inverse() * l2->estimate()) - _measurement;
    }

    virtual bool read(std::istream& is);
    virtual bool write(std::ostream& os) const;

};
```

Listing 7: Edge connecting a robot poses and a landmark.

is shown in Listing 7. Again, we extend a specialization of `BaseBinaryEdge<>`, and we tell the system that it connects a `VertexSE2` with a `VertexPointXY`, that the measurement is represented by an `Eigen::Vector2d` that has dimension 2.

The final step we should do to make our system operational, is to register the types to let g^{2}o know that there are new types ready. However, if you intend to manually construct your graph without doing any i/o operation on disk, this step is not even necessary. Have fun!

You may find the full code of this 2D SLAM example in the folder `examples/tutorials_slam2d`.