UNIVERSITY OF TORONTO
INSTITUTE FOR AEROSPACE STUDIES

ASRL-2015-SWA001
Rev: 1.0
July 21, 2015

STEAM Engine

Sean Anderson
University of Toronto
Institute for Aerospace Studies
<sean.anderson@mail.utoronto.ca>

1 Introduction

This document serves as an introduction to the Simultaneous Trajectory Estimation and Mapping (STEAM) Engine software. STEAM Engine is an optimization library aimed at solving batch nonlinear optimization problems involving both SO(3)/SE(3) and continuous-time components. This is accomplished by using an iterative Gauss-Newton-style estimator in combination with techniques developed and used by ASRL. With respect to SO(3) and SE(3) components, we make use of the constraint sensitive perturbation schemes discussed in Barfoot and Furgale [1]. STEAM Engine is by no means intended to be the *fastest car on the track*; the intent is simply to be *fast enough* for the types of problems we wish to solve, while being both readable and easy to use by people with a basic background in robotic state estimation.

In this note, we will introduce some terminology, the types of problems we wish to solve, and the methods we use to go about solving them. Note that terms in `ThisFont` reflect classes in the code base. The following sections describe the two major phases involved with setting up and solving an optimization problem with STEAM Engine: i) creating an `OptimizationProblem` and ii) passing the problem to a *solver* (such as the `LevMarqGaussNewtonSolver`).

2 `OptimizationProblem`

The first step in using STEAM Engine is to create an instance of an `OptimizationProblem` which fully defines the problem you wish to solve; this is accomplished by creating and adding the relevant `StateVariable` and `CostTerm` objects. STEAM Engine is setup to solve nonlinear optimization problems of the form:

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} J(\mathbf{x}), \tag{1}$$

where $J(\mathbf{x})$ is the *objective function* or *cost* (composed of many `CostTerms`) which we wish to minimize with respect to our `StateVector`, $\mathbf{x}$.

2.1 `CostTerm`

A typical *objective function*, $J(\mathbf{x})$, is composed of many `CostTerms`, $J_i(\mathbf{x})$:

$$J(\mathbf{x}) := \sum_i J_i(\mathbf{x}), \quad J_i(\mathbf{x}) := \rho(e_i), \qquad e_i := ||\mathbf{e}_i(\mathbf{x})||_{\mathbf{R}_i} = \sqrt{\mathbf{e}_i(\mathbf{x})^T \mathbf{R}_i^{-1} \mathbf{e}_i(\mathbf{x})}. \tag{2}$$

where each `CostTerm`, $J_i(\mathbf{x})$, is subsequently composed of i) an `ErrorEvaluator` (or *error function*), $\mathbf{e}_i(\mathbf{x})$, ii) a `NoiseModel`, $\mathbf{R}_i$, and iii) a `LossFunction`, $\rho(e_i)$.

2.1.1 `ErrorEvaluator`

The purpose of the `ErrorEvaluator` is to use the current value of the state variables, $\mathbf{x}$, to evaluate the function $\mathbf{e}_i(\mathbf{x})$; furthermore, `ErrorEvaluators` must be equipped with the ability to provide Jacobians with respect to relevant state variables, $\mathbf{x}$, given their current value. Note that as the state variables change value between iterations, so does the output of an evaluator; the concept of *evaluators* are a core part of how STEAM Engine functions, and offer a lot of convenience.

The standard `ErrorEvaluator` assumes the form

$$\mathbf{e}_i(\mathbf{x}) := \mathbf{f}_i(\mathbf{x}) + \mathbf{n}_i, \qquad \mathbf{n}_i \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_i), \tag{3}$$

where $\mathbf{f}_i(\cdot)$ is a (non)linear function which returns a vector-space *cost* or *error* and $\mathbf{n}_i$ is zero-mean Gaussian measurement noise with covariance $\mathbf{R}_i$ (set through the `NoiseModel`).

2.1.2 `NoiseModel`

The purpose of the `NoiseModel` is simply to store the *covariance* or *noise* associated with an `ErrorEvaluator`. While a `NoiseModel` can be set by either its covariance ($\mathbf{R}_i$), information ($\mathbf{R}_i^{-1}$), or (upper-triangular) square root information ($\mathbf{U}_i$), we internally store the later, for efficiency in *error whitening*. Note that

$$\mathbf{U}_i^T \mathbf{U}_i = \mathbf{R}_i^{-1}, \tag{4}$$

where $\mathbf{U}_i$ is an upper-triangular matrix found using *cholesky decomposition*. *Error whitening* or the *whitening transformation* refers to the process by which we *normalize* the input vector into a white noise vector (given its 'known' covariance matrix). The resultant *whitened errors* can be assumed to have an identity covariance matrix, noting that

$$\mathbf{e}_i(\mathbf{x})^T \mathbf{R}_i^{-1} \mathbf{e}_i(\mathbf{x}) = \mathbf{e}_i(\mathbf{x})^T \mathbf{U}_i^T \mathbf{U}_i \mathbf{e}_i(\mathbf{x}) = (\mathbf{U}_i \mathbf{e}_i(\mathbf{x}))^T \underbrace{\mathbf{U}_i \mathbf{e}_i(\mathbf{x})}_{\text{whitened error}} . \tag{5}$$

2.1.3 `LossFunctions`

Lastly, the use of `LossFunctions`, $\rho(\cdot)$, provide users the ability to use *M-estimation* or *robust estimation* techniques on a per-`CostTerm` basis. In the context of solving equation (1) using *M-estimation*, we note that it is convenient to rewrite the `CostTerm` in (2) as the equivalent *iteratively reweighted least-squares* (IRLS) term

$$J_i(\mathbf{x}) = \rho(e_i) = \frac{1}{2} w(e_i) e_i^2 = \frac{1}{2} w(e_i)\, \mathbf{e}_i(\mathbf{x})^T \mathbf{R}_i^{-1} \mathbf{e}_i(\mathbf{x}), \tag{6}$$

where we note that a valid `LossFunction`, $\rho(e_i)$, has an associated weighting function, $w(e_i)$. While there are many valid `LossFunctions`, the simplest (and arguably most common) example is the least squares (or L_2) loss function, for which we note that $\rho(e_i) = \frac{1}{2} e_i^2$ and $w(e_i) = 1$. Using a L_2 `LossFunction` for all of the `CostTerms`, we are left with the (hopefully) familiar optimization problem,

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} \frac{1}{2} \sum_i \mathbf{e}_i(\mathbf{x})^T \mathbf{R}_i^{-1} \mathbf{e}_i(\mathbf{x}). \tag{7}$$

2.2 `StateVariable`

The goal of the solver is to minimize the total cost of a collection of `CostTerms` with respect to the values of some collection of `StateVariables`. The state variables are therefore typically the desired output of the optimization problem, and given some initial condition, it is the solvers job to try and find a set of values for the state variables which produces a minimum cost.

Note that while the goal is to find the *globally* minimal cost, we may instead find a *locally* minimal cost (depending on the initial condition). In STEAM Engine, `StateVariable` types are created by defining a *value* class (such as a vector class, transformation matrix class, etc.), a *perturbation* dimension, and a method to update the state variable given a vector of the *perturbation* dimension. Some common types have been implemented and are described in the following subsections.

2.2.1 `VectorSpaceStateVar`

Probably the simplest type of state variable, the vector-space state variable, $\mathbf{v} \in \mathbb{R}^N$, wraps the `Eigen::VectorXd` class, and defines the update method

$$\mathbf{v} \leftarrow \mathbf{v} + \delta\mathbf{v}, \tag{8}$$

where $\delta\mathbf{v}$ is the perturbation (with matching dimension N) the solver will iterative find and use to update the state variable $\mathbf{v}$.

2.2.2 `LieGroupStateVar`

Particularly in robotics, it is common to have rotation and transformation matrices as state variables that describe the orientation or pose of the robot in 3D. Note that rotation matrices, $\mathbf{C}$, belong to the special orthogonal group, $SO(3)$, and transformation matrices, $\mathbf{T}$, belong the special Euclidean group, $SE(3)$.

It is fairly common knowledge how rotation and transformation matrices can be used in practice. However, it is less obvious how we should linearize `CostTerms` with respect to them, solve for an update perturbation, and apply that perturbation to the Lie group state variable. While there exists many parameterizations for rotation matrices, we favour the constraint sensitive perturbation schemes detailed in Barfoot and Furgale [1].

The `LieGroupStateVar` implementation assumes that Lie group state variables have a multiplicative update, such as

$$\mathbf{T} \leftarrow \exp(\delta\boldsymbol{\xi}^\wedge)\mathbf{T}, \quad \boldsymbol{\xi} \in \mathbb{R}^6, \tag{9a}$$
$$\mathbf{C} \leftarrow \exp(\delta\boldsymbol{\phi}^\wedge)\mathbf{C}, \quad \boldsymbol{\phi} \in \mathbb{R}^3, \tag{9b}$$

where $\delta\boldsymbol{\xi}$ and $\delta\boldsymbol{\phi}$ are perturbation vectors that would be used to update the matrices $\mathbf{T}$ and $\mathbf{C}$. While STEAM Engine does not tie users to a specific transformation matrix implementation, it does assume that the implementations will have constructors based on the exponential map. If this is not suitable, users can easily implement their own Lie group state variable classes based on other update schemes.

For those interested in more detail on the exponential map, and how uncertainties can be handled, we suggest reading Barfoot and Furgale [1]. Here, we provide a brief overview of the closed-form construction of a transformation matrix,

$$\mathbf{T} := \exp(\boldsymbol{\xi}^\wedge) = \begin{bmatrix} \mathbf{C} & \mathbf{J}\boldsymbol{\rho} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{10}$$

where $\boldsymbol{\xi} \in \mathbb{R}^6$, the $\wedge$ operator turns $\boldsymbol{\xi}$ into a 4×4 member of the *Lie algebra* $\mathfrak{se}(3)$,

$$\boldsymbol{\xi}^\wedge := \begin{bmatrix} \boldsymbol{\rho} \\ \boldsymbol{\phi} \end{bmatrix}^\wedge = \begin{bmatrix} \boldsymbol{\phi}^\wedge & \boldsymbol{\rho} \\ \mathbf{0}^T & 0 \end{bmatrix}, \quad \boldsymbol{\rho}, \boldsymbol{\phi} \in \mathbb{R}^3 \tag{11}$$

and $\wedge$ also turns $\boldsymbol{\phi}$ into a 3×3 member of the *Lie algebra* $\mathfrak{so}(3)$,

$$\boldsymbol{\phi}^\wedge := \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \end{bmatrix}^\wedge := \begin{bmatrix} 0 & -\phi_3 & \phi_2 \\ \phi_3 & 0 & -\phi_1 \\ -\phi_2 & \phi_1 & 0 \end{bmatrix}.$$

The rotation matrix, $\mathbf{C} \in SO(3)$, can be computed using the exponential map,

$$\mathbf{C} := \exp(\boldsymbol{\phi}^\wedge) = \cos\phi\mathbf{1} + (1-\cos\phi)\mathbf{a}\mathbf{a}^T + \sin\phi\mathbf{a}^\wedge, \tag{12}$$

where $\phi = ||\boldsymbol{\phi}||$ is the angle of rotation, and $\mathbf{a} = \boldsymbol{\phi}/\phi$ is the axis of rotation. Lastly, we also have a closed form expression for $\mathbf{J}$, which is the (left) Jacobian of $SO(3)$,

$$\mathbf{J} := \frac{\sin\phi}{\phi}\mathbf{1} + \left(1 - \frac{\sin\phi}{\phi}\right)\mathbf{a}\mathbf{a}^T + \frac{1-\cos\phi}{\phi}\mathbf{a}^\wedge. \tag{13}$$

2.2.3 `LandmarkStateVar`

Landmarks are a very specific type of state used in *vision*-based estimation problems, such as *visual odometry* or *bundle adjustment*. We implement landmarks as their own type of state variable, rather than a `VectorSpaceStateVar`, for two conveniences. First, we internally store landmarks as a homogeneous coordinate point to make multiplication with transformations more efficient. Second, we also provide the ability to denote a *reference frame*, which is a transformation matrix, or pose, that the landmark is estimated with respect to. For example, in the *global* bundle adjustment paradigm we would choose the reference frame to the 'base frame' or first camera frame, where as in the *relative* bundle adjustment paradigm we could set each landmark's reference frame to be the pose of the camera at the first time the landmark was observed.

2.2.4 Locking a `StateVariable`

The `StateVariableBase` class implements a simple method allowing users to lock/unlock state variables. The expected behaviour of a *locked* state variable is that its value is fixed, and cannot be updated, furthermore, `ErrorEvaluators` *should not* create Jacobians with respect to *locked* state variables when evaluated. The benefit of this feature is that it provides users some reusability with respect to creating `StateVariables` and `CostTerms`. Specifically, we note that `CostTerms` do not need to be recreated when they depend on two or more state variables (e.g. a transformation matrix and landmark variable) and one of them is locked (e.g. the transformation matrix in a sliding-window-filter-style algorithm).

Note that locked state variables can be added to `CostTerms`, but should not be added directly to the `OptimizationProblem` as a state variable. Also, state variables *should not* be locked after having added them to an `OptimizationProblem`. It is quick to recreate an `OptimizationProblem` by reusing the unlocked `StateVariables` and existing `CostTerms`.

2.2.5 Continuous-Time Trajectories

TODO

3 Solvers (derivatives of `SolverBase`)

As a preface, the purpose of this section is to give some intuition behind what the various types of solvers are doing 'behind the curtain'. General users of the STEAM Engine do not need to know the details of the solvers, and simply need to construct an `OptimizationProblem` and pass it to a solver for optimization.

3.1 `SolverBase`

The purpose of the base solver class is to implement the iterative nature of the optimization algorithm and check for stopping criteria. The default stopping criteria, set through the `Params` class are:

1. Maximum iterations (500)

2. Absolute cost threshold ($J_{\text{new}} \leq 0$)
3. Absolute cost change threshold ($|J_{\text{new}} - J_{\text{old}}| \leq 1e-4$)
4. Relative cost change threshold ($\frac{|J_{\text{new}} - J_{\text{old}}|}{J_{\text{old}}} \leq 1e-4$)

The `SolverBase` class is extended by derived classes to implement the linearize, solve and update steps.

3.2 `GaussNewtonSolverBase`

In the current iteration of the STEAM Engine software, the `GaussNewtonSolverBase` class is the sole derivative of `SolverBase` and implements the basic functionalities needed by the Gauss-Newton-style optimization schemes.

Recall the IRLS problem:

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} J, \qquad J = \frac{1}{2}\sum_i w(e_i)\mathbf{e}_i(\mathbf{x})^T\mathbf{R}_i^{-1}\mathbf{e}_i(\mathbf{x}).$$

Taking the standard Gauss-Newton optimization approach to solving this problem requires us to linearize the `ErrorEvaluators` about our best guess of the state, $\bar{\mathbf{x}}$,

$$\mathbf{e}_i(\mathbf{x}) \approx \mathbf{f}_i(\bar{\mathbf{x}}) + \mathbf{F}_i\delta\mathbf{x}, \quad \mathbf{F}_i = \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\bar{\mathbf{x}}}, \tag{14}$$

where $\mathbf{F}_i$ is a set of `Jacobians` (each `Jacobian` being related to a single state variable that impacts the `ErrorEvaluator`). Plugging the linearized error terms into the cost function, taking the derivative of the cost with respect to $\delta\mathbf{x}$, and setting it to zero, reveals that we must solve the following system of linear equations at each iteration of Gauss-Newton:

$$\left(\sum_i w(e_i)\mathbf{F}_i^T\mathbf{R}_i^{-1}\mathbf{F}_i\right)\delta\mathbf{x}^* = \sum_i w(e_i)\mathbf{F}_i^T\mathbf{R}_i^{-1}\mathbf{f}_i(\bar{\mathbf{x}}), \tag{15}$$

where $\delta\mathbf{x}^*$ is the optimal perturbation. Solving for $\delta\mathbf{x}^*$ and using the result to update $\bar{\mathbf{x}}$ according to our update rules (recall (8) and (9)), we can iteratively solve the problem in (1).

Note that the `GaussNewtonSolverBase` class simply implements methods for building the left-hand-side and right-hand-side terms of (15) using block-sparse matrices (note the build step is parallelized with OpenMP, the number of threads to use can be set in the CMakeLists file), and then solving the system for $\delta\mathbf{x}^*$ with a sparse `Eigen` LLT decomposition implementation. It also provides some methods needed by the Levenberg-Marquardt and Powell's Dogleg solvers based on the terms of (15). Derivatives of this class are responsible for actually updating the state, as various algorithms, such as trust-region solvers, have different approaches to applying the update $\delta\mathbf{x}^*$.

3.3 `VanillaGaussNewtonSolver`

The *vanilla* Gauss-Newton solver uses the basic method described above to solve for $\delta\mathbf{x}^*$ and *blindly* updates $\bar{\mathbf{x}}$.

3.4 Line Search and Trust Region Solvers

A solver's *update*, or *step*, strategy involves both a *step* direction and a *step* length. In the context of Gauss-Newton-style solvers, there are two major paradigms in step strategy: i) line searches and ii) trust regions.

Line searches typically work on a fixed step direction and then search for a step length that reduces the cost, J. In contrast, trust region methods tend to set a *region size* (usually related to step length), for which it believes the linearization well approximates the nonlinear cost function, and then chooses a step direction; note that the step length is adjusted dynamically as the trust region becomes too conservative, or not conservative enough.

3.4.1 `LineSearchGaussNewtonSolver`

This solver implements a basic backtracking line search. Essentially, the algorithm tries updating the state using the Gauss-Newton step, $\delta\mathbf{x}^*$, and then evaluates the `CostTerms` to see if J_{new} is less than J_{old}. If it is not, then the state is reverted, the step is reduced by the scalar *backtrack multiplier* (default 0.5) and the update is tested again (note the 'direction' is the same). This procedure repeats until the cost is reduced or a maximum number of backtracks are tried (default 10).

3.4.2 `LevMarqGaussNewtonSolver`

The Levenberg-Marquardt trust-region method is one which smoothly transitions between the Gauss-Newton and gradient-descent steps based on how much it *trusts* the step. As an aside, the Gauss-Newton step direction has very strong convergence when the linearization well approximates the nonlinear function, but is not guaranteed to reduce the cost. In contrast, the gradient-descent direction generally has slow convergence, but can be trusted to provide a direction that (locally) reduces the cost. Recalling that (15) has the form, $\mathbf{A}\delta\mathbf{x}^* = \mathbf{b}$, the Levenberg-Marquardt method solves the augmented system of equations:

$$(\mathbf{A} + \lambda \text{diag}(\mathbf{A}))\,\delta\mathbf{x}^* = \mathbf{b}, \tag{16}$$

where λ is the trust-region parameter. As λ tends to zero, we solve for the Gauss-Newton step direction, and as λ increases, we tend towards the gradient-descent direction.

Similar to the line-search methodology, after solving for a step update, $\delta\mathbf{x}^*$, the effect on cost reduction is tested, and depending on the result, the update is either accepted or rejected, and the trust-region parameter, λ, is modified accordingly.

3.4.3 `DoglegGaussNewtonSolver`

The Powell's Dogleg algorithm is an alternative trust-region solver (similar in nature to the Levenberg-Marquardt method) that uses some simple heuristics to choose a step direction and length. In essence, the Dogleg algorithm interpolates the step direction and length from the intersection of the 'trust-region size', λ, with one of two discrete 'paths'. The first 'path' extends from zero along the gradient-descent direction to what is known as the *Cauchy* point:

$$\mathbf{c} = \frac{\mathbf{b}^T\mathbf{b}}{\mathbf{b}^T\mathbf{A}\mathbf{b}}. \tag{17}$$

The second 'path' connects the *Cauchy* point, $\mathbf{c}$, and the Gauss-Newton step, $\delta\mathbf{x}^*$. Depending on the size of λ, we interpolate the step length and direction from one of the two paths, or if λ is greater than the Euclidean norm of the Gauss-Newton step, $\delta\mathbf{x}^*$, then we simply take the Gauss-Newton step.

This method generally performs very well, even in contrast with the more sophisticated Levenberg-Marquardt method. Furthermore, the Powell's Dogleg algorithm is very fast because it does not need to resolve the system of equations when λ changes (unlike Levenberg-Marquardt).

4 `TransformEvaluators`

`TransformEvaluators` are a structure which allows us to abstract multiple operations which ultimately result in a single pose change (and allow us to keep track of how to take Jacobians with respect to any states involved in those operations). This is a major convenience tool for those who are unfamiliar with the *Lie group* maths and are new to taking Jacobians with respect to the se(3) perturbations. The other major benefit of `TransformEvaluators` is that it provides a high level of reusability when it comes to creating `ErrorEvaluators`.

4.1 Stereo Camera Error Metric

To demonstrate how `TransformEvaluators` can be used, and how they make code reusable, we provide this example. First we will derive the necessary maths, then show how we might 'hardcode' an `ErrorEvaluator`, and finally how it can be done using `TransformEvaluators`.

4.1.1 Nonlinear Measurement

We begin by considering a simple visual measurement model for a stereo-camera,

$$\mathbf{y}_i = \mathbf{h}(\mathbf{T}_{i,0}\mathbf{p}_\ell) + \mathbf{n}_i, \quad \mathbf{n}_i \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_i), \quad \mathbf{p}_\ell = \begin{bmatrix} \boldsymbol{\zeta}_0^{\ell,0} \\ 1 \end{bmatrix}, \tag{18}$$

where $\mathbf{T}_{i,0}$ is the pose of the camera at timestep i with respect to base frame 0, $\mathbf{p}_\ell$ is the homogeneous point coordinate for landmark ℓ, and $\mathbf{h}(\cdot)$ is the nonlinear stereo-camera model that transforms a 3D point into the stereo pixel coordinates:

$$\mathbf{h}(\mathbf{p}) = \begin{bmatrix} u_l \\ v_l \\ u_r \\ v_r \end{bmatrix} = \frac{1}{z}\begin{bmatrix} f_u x \\ f_v y \\ f_u(x-b) \\ f_v y \end{bmatrix} + \begin{bmatrix} c_u \\ c_v \\ c_u \\ c_v \end{bmatrix} \tag{19}$$

where x, y, and z are coordinates of the point $\mathbf{p}$, f_u and f_v are the horizontal and vertical focal lengths, b is the stereo baseline parameter, and c_u and c_v are the cameras horizontal and vertical optical centers.
Formulating this into an error term, we typically write

$$\mathbf{e}_i = \mathbf{y}_i - \mathbf{h}(\mathbf{T}_{i,0}\mathbf{p}_\ell) \tag{20}$$

where $\mathbf{y}_i$ is a physical observation and $\mathbf{e}_i$ is the error based on our estimate of the state variables $\mathbf{T}_{i,0}$ and $\mathbf{p}_\ell$. In order to linearize this error term, we begin by noting that (20) has a composition of two nonlinearities, one for the camera model and one for transformating the landmark into the sensor frame. We separate the two nonlinearities by defining

$$\mathbf{g}_i := \mathbf{T}_{i,0}\mathbf{p}_\ell. \tag{21}$$

Applying our perturbation schemes, we then have

$$\mathbf{g}_i = \exp(\delta\boldsymbol{\xi}_{i,0}^\wedge)\mathbf{T}_{i,0}(\mathbf{p}_\ell + \mathbf{D}\delta\boldsymbol{\zeta}_\ell), \quad \text{where } \mathbf{D} := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}. \tag{22}$$

Noting that when $\delta\boldsymbol{\xi}$ is small,

$$\exp(\delta\boldsymbol{\xi}^\wedge) \approx \mathbf{1} + \delta\boldsymbol{\xi}^\wedge, \tag{23}$$

and using the identity

$$\mathbf{w}^\wedge\mathbf{y} \equiv \mathbf{y}^\odot\mathbf{w}, \quad \mathbf{y}^\odot := \begin{bmatrix} \boldsymbol{\varepsilon} \\ \eta \end{bmatrix}^\odot = \begin{bmatrix} \eta\mathbf{1} & -\boldsymbol{\varepsilon}^\wedge \\ \mathbf{0}^T & \mathbf{0}^T \end{bmatrix}, \tag{24}$$

we are able to expand drop small terms to find that,

$$\mathbf{g}_i \approx (\mathbf{1} + \delta\boldsymbol{\xi}_{i,0}^\wedge)\mathbf{T}_{i,0}(\mathbf{p}_\ell + \mathbf{D}\delta\boldsymbol{\zeta}_\ell) \tag{25}$$

$$= \mathbf{T}_{i,0}\mathbf{p}_\ell + (\mathbf{T}_{i,0}\mathbf{p}_\ell)^\odot\delta\boldsymbol{\xi}_{i,0} + \mathbf{T}_{i,0}\mathbf{D}\delta\boldsymbol{\zeta}_\ell \tag{26}$$

$$= \bar{\mathbf{g}}_i + \mathbf{G}_i\delta\mathbf{x}\begin{bmatrix} \delta\boldsymbol{\xi}_{i,0} \\ \delta\boldsymbol{\zeta}_\ell \end{bmatrix}, \tag{27}$$

correct to first order, where

$$\bar{\mathbf{g}}_i := \mathbf{T}_{i,0}\mathbf{p}_\ell, \tag{28a}$$
$$\mathbf{G}_i := \begin{bmatrix} (\mathbf{T}_{i,0}\mathbf{p}_\ell)^\odot & \mathbf{T}_{i,0}\mathbf{D} \end{bmatrix}. \tag{28b}$$

Returning to our measurement error term,

$$\begin{aligned}
\mathbf{e}_i &\approx \mathbf{y}_i - \mathbf{h}\left(\bar{\mathbf{g}}_i + \mathbf{G}_i \begin{bmatrix} \delta\boldsymbol{\xi}_{i,0} \\ \delta\boldsymbol{\zeta}_\ell \end{bmatrix}\right) \\
&\approx \mathbf{y}_i - \mathbf{h}(\bar{\mathbf{g}}_i) - \mathbf{H}_i\mathbf{G}_i \begin{bmatrix} \delta\boldsymbol{\xi}_{i,0} \\ \delta\boldsymbol{\zeta}_\ell \end{bmatrix} \\
&= \bar{\mathbf{e}}_i + \mathbf{F}_i \begin{bmatrix} \delta\boldsymbol{\xi}_{i,0} \\ \delta\boldsymbol{\zeta}_\ell \end{bmatrix}
\end{aligned} \tag{29}$$

correct to first order, where

$$\bar{\mathbf{e}}_i := \mathbf{y}_i - \mathbf{h}(\bar{\mathbf{g}}_i), \tag{30a}$$
$$\mathbf{F}_i := -\mathbf{H}_i\mathbf{G}_i, \qquad \mathbf{H}_i := \left.\frac{\partial \mathbf{h}}{\partial \mathbf{g}}\right|_{\bar{\mathbf{g}}_i}. \tag{30b}$$

4.1.2 Simple `ErrorEvaluator`

In order to use this error term in our code, we create an `ErrorEvaluator`. The constructor for such a class might look like:

```
StereoCameraErrorEval(const Eigen::Vector4d& meas,
                      const CameraIntrinsics::ConstPtr& intrinsics,
                      const se3::TransformStateVar::ConstPtr& T_cam_landmark,
                      const se3::LandmarkStateVar::ConstPtr& landmark);
```

where the `Eigen::Vector4d` holds our measurement, `CameraIntrinsics` is a structure that holds the stereo camera intrinsic parameters, and the `se3::TransformStateVar` and `se3::LandmarkStateVar` are our state variables. The main functionalities we must then implement are the methods:

```
virtual Eigen::VectorXd evaluate() const;
virtual Eigen::VectorXd evaluate(std::vector<Jacobian>* jacs) const;
```

We will focus on the later, as it encompasses the full functionality:

```
Eigen::VectorXd StereoCameraErrorEval::evaluate(std::vector<Jacobian>* jacs) const {

  // Get point in camera frame
  Eigen::Vector4d point_in_c = T_cam_landmark_->getValue() * landmark_->getValue();

  // Get Jacobian for the camera model
  Eigen::Matrix4d H = cameraModelJacobian(point_in_c);

  // Construct diffusion matrix
  Eigen::MatrixXd D = Eigen::Matrix<double,4,3>::Identity();

  // Check and initialize jacobian array
  CHECK_NOTNULL(jacs);
  jacs->clear();
  jacs->reserve(2);

  // Calculate Jacobian w.r.t. transform
  if(!T_cam_landmark_->isLocked()) {
    jacs->push_back(Jacobian(T_cam_landmark_->getKey(),
                             -H * lgmath::se3::point2fs(point_in_c.head<3>())));
  }

  // Calculate Jacobian w.r.t. landmark
  if(!landmark_->isLocked()) {
    jacs->push_back(Jacobian(landmark_->getKey(),
                             -H * T_cam_landmark_->getValue().matrix() * D));
  }

  // Return error (between measurement and point estimate projected in camera frame)
  return meas_ - cameraModel(point_in_c);
}
```

Analyzing this code, we encounter a few new concepts to STEAM Engine that are not previously explained. Firstly, in order to evaluate a state variable, we call `getValue()`, as seen in the line that evaluates the point in the sensor frame, `point_in_c`. Next, we see the process of creating and returning `Jacobians` with respect to unlocked state variables. Note that if a state variable is locked, STEAM Engine expects that you will not spend the time calculating and returning `Jacobian` terms for it. The two components of a `Jacobian` are a `StateKey` (found by calling `getKey()`) that associate the `Jacobian` to a specific instance of a `StateVariable`, and the actual matrix value at evaluation time. Lastly, we return the error vector.

4.1.3 Why use `TransformEvaluators`

The primary reason to use `TransformEvaluators` is code reusability. Consider our previous example, but now we have additional sensors (such as an IMU) with extrinsic calibrations. Instead of our state variables, $\mathbf{T}_{i,0}$, representing camera poses, they now represent vehicle poses, and we must use the extrinsic calibration between the camera and vehicle, $\mathbf{T}_{c,v}$, to related measured points to our state variables. To implement this, we would have to modify our previous class to now accept an extrinsic calibration, and make sure that all our `Jacobians` are still correct. If we instead had a constructor that just accepted a `TransformEvaluator`:

```
StereoCameraErrorEval(const Eigen::Vector4d& meas,
                      const CameraIntrinsics::ConstPtr& intrinsics,
                      const se3::TransformEvaluator::ConstPtr& T_cam_landmark,
                      const se3::LandmarkStateVar::ConstPtr& landmark);
```

the code becomes very extensible. Recall, `TransformEvaluators` are a structure which allows us to abstract multiple operations which ultimately result in a single pose change. `TransformEvaluators` can be used to encapsulate a transformation state variable

```
steam::se3::TransformEvaluator::Ptr pose_v_0 = steam::se3::TransformStateEvaluator::MakeShared(T_i0);
```

a fixed transformation, such as the extrinsic calibration,

```
steam::se3::TransformEvaluator::Ptr pose_c_v = steam::se3::FixedTransformEvaluator::MakeShared(T_cv);
```

and finally compose them using the provided `TranformEvalOperations`,

```
steam::se3::TransformEvaluator::Ptr T_cam_landmark = steam::se3::compose(pose_c_v, pose_v_0);
```

The resultant transformation matrix evaluator still has ties to the underlying state variables, and similar to an `ErrorEvaluator`, knows how to provide `Jacobians` with respect to any state variable transformation matrices involved in its evaluation.

The available operations (some of which can be chained), with known Jacobians, are:

$$\mathbf{T}_1\mathbf{T}_2, \quad \mathbf{T}^{-1}, \quad \mathbf{T}\mathbf{p}, \quad \ln(\mathbf{T})^\vee. \tag{31}$$

Using these operations, it becomes trivial to implement new error metrics. For example, implementing a relative transformation error metric is as simple as:

1. Creating `TransformEvaluators` for the state variables, say $\mathbf{T}_1$ and $\mathbf{T}_2$
2. Creating a `TransformEvaluator` for the fixed measurement $\bar{\mathbf{T}}_{21}$
3. Using the inverse operation on $\mathbf{T}_2$
4. Using the compose operation on $\mathbf{T}_1$ with $\mathbf{T}_2^{-1}$ to create $\mathbf{T}_{12}$
5. Using the compose operation on $\bar{\mathbf{T}}_{21}$ with $\mathbf{T}_{12}$
6. Using the logarithmic map operation $\ln(\bar{\mathbf{T}}_{21}\mathbf{T}_{12})^\vee$, which returns a 6×1 vector

5 More Examples

To try out and start understanding the STEAM Engine functionality, take a look at the samples:

- TrustRegionExample.cpp
- SimplePoseGraphRelax.cpp
- SpherePoseGraphRelax.cpp
- SimpleBundleAdjustment.cpp
- SimpleBundleAdjustmentRelLand.cpp

References

[1] Barfoot, T. D. and Furgale, P. T., "Associating Uncertainty with Three-Dimensional Poses for use in Estimation Problems," *IEEE Transactions on Robotics*, 2014.