View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Waterloo's Institutional Repository

On the Manifold: Representing Geometry in C++ for State Estimation

by

Leonid Koppel

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Mechanical and Mechatronics Engineering

Waterloo, Ontario, Canada, 2018

© Leonid Koppel 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Manipulating geometric objects is central to state estimation problems in robotics. Typical algorithms must optimize over non-Euclidean states, such as rigid transformations on the SE(3) manifold, and handle measurements expressed in multiple coordinate frames. Researchers typically rely on C++ libraries for geometric tasks. Commonly used libraries range from linear algebra software such as Eigen to robotics-targeted optimization frameworks such as GTSAM, which provides manifold operations and automatic differentiation of arbitrary expressions. This thesis examines how geometric operations in existing software can be improved, both in runtime performance and in the expression of geometric semantics, to support rapid and error-free development of robotics algorithms.

This thesis presents `wave_geometry`, a C++ manifold geometry library providing representations of objects in affine, Euclidean, and projective spaces, and the Lie groups SO(3) and SE(3). It encompasses the main contributions of this work: an expression template-based automatic differentiation system and compile-time checking of coordinate frame semantics. The library can evaluate Jacobians of geometric expressions in forward and reverse mode with little runtime overhead compared to hand-coded derivatives, and exceeds the performance of existing libraries. While high performance is achieved by taking advantage of compile-time knowledge, the library also provides dynamic expressions which can be composed at runtime.

Coordinate frame conversions are a common source of mistakes in calculations. However, the validity of operations can automatically be checked by tracking the coordinate frames associated with each object. A system of rules for propagating coordinate frame semantics though geometric operations, including manifold operations, is developed. A template-based method for checking coordinate frame semantics at compile time, with no runtime overhead, is presented.

Finally, this thesis demonstrates an application to state estimation, presenting a framework for formulating nonlinear least squares optimization problems as factor graphs. The framework combines `wave_geometry` expressions with the widely used Ceres Solver software, and shows the utility of automatically differentiated geometric expressions.

Acknowledgements

I am grateful for the constant mentorship and encouragement of my supervisor, Dr. Steven L. Waslander. Thank you for the opportunities to work on amazing projects and for the freedom and trust you give your students.

I would like to thank all of my colleagues at Wavelab for their friendship in the office and at the test track. To my roommates, Ben, Jung, Nav, and Sean, thank you for always being there. To my senior colleagues—Ali, Arun, Bek, Carlos, Chris, Jason, Melissa, Nik, Stan—thank you for the advice and answers. Thank you to Ian Colwell, Michael Smart, and Dr. Michał Antkiewicz for the coding companionship. Thank you to Nicholas Charron and Matthew Pitropov for their help submitting this thesis, and to Adeel Akhtar for his Lie group expertise.

Finally, I would like to thank my parents for their endless support and for everything they do for me.

Dedication

To my family,

Mom, Dad, and Olga.

Table of Contents

List of Tables x

List of Figures xi

List of Symbols xiii

1 Introduction 1

1.1 Related work 3

1.1.1 Automatic differentiation 3

1.1.2 Geometry frameworks 4

1.2 Thesis structure 7

2 Background 9

2.1 Notation 9

2.2 Points, vectors, and spaces 10

2.2.1 Points and vectors 10

2.2.2 Affine spaces 11

2.2.3 Affine frames 12

2.2.4 Affine transformations 13

2.3 Homogeneous coordinates 14

2.3.1 Projective spaces 15

2.3.2 Normalization of homogeneous coordinates 15
2.4 Rigid motions 16
2.4.1 Euclidean spaces 16
2.4.2 Distances and orthogonality 17
2.4.3 Rotations and Euclidean motions 18
2.5 Rotations and Euclidean motions 18
2.5.1 The rotations SO(3) 19
2.5.2 Manifolds and charts 20
2.5.3 Lie groups and Lie algebras 21
2.5.4 Exponential and logarithmic maps 22
2.5.5 The tangent space and adjoints 22
2.5.6 Perturbations on a manifold 24
2.5.7 Derivatives on Lie groups 27
2.5.8 Operations on SO(3) 28
2.5.9 Quaternions 29
2.5.10 Perturbations of homogeneous points 30
2.5.11 The Euclidean motions SE(3) 31
2.6 Automatic differentiation 32
2.6.1 Forward mode 33
2.6.2 Reverse mode 34
2.6.3 Block automatic differentiation 35

3 Manifold geometry in C++ 36

3.1 Expression templates for geometry 37
3.1.1 Implementing expression templates 39
3.1.2 Spaces, parametrizations, and storage 41
3.1.3 Evaluating expressions 44
3.2 Automatic differentiation 49

3.2.1 Forward-mode AD 49
3.2.2 Testing for identity 50
3.2.3 Strongly typed forward-mode AD 52
3.2.4 Reverse-mode AD 53
3.3 Composing expressions at runtime 54
3.4 Experimental results 57
3.4.1 Benchmark experiments 57
3.4.2 Rotation Chain 58
3.4.3 IMU Factor 61
3.4.4 Dynamic expressions 61

4 Coordinate frame semantics checking **63**
4.1 Coordinate frame semantics 63
4.1.1 Coordinate-free geometry 63
4.1.2 Expressing semantics through notation 64
4.1.3 Semantics in Code 67
4.1.4 Free and bound transformations 68
4.2 C++ library implementation 69
4.3 Related works 71
4.4 Rules for frame semantics 72

5 Application to state estimation **74**
5.1 State estimation as a least squares problem 74
5.1.1 Uncertain estimates 75
5.1.2 Maximum a Posteriori 77
5.2 Library implementation 81
5.2.1 Design 81

6 Conclusion **85**

References 88

APPENDICES 94

A Mathematical background 95

A.1 Lie group operations . . . 95

A.1.1 Exponential map onto SO(3) . . . 95

A.1.2 Logarithmic map of SO(3) . . . 98

A.1.3 Approximation of the exponential map . . . 99

B Frames, coordinates, systems: a few words 100

C C++ library details 103

C.1 Storing temporary expressions . . . 103

C.1.1 Problems of using `auto` with expressions . . . 103

C.1.2 Storing temporary objects in expressions . . . 105

C.2 Evaluating proxies . . . 106

C.2.1 Optimizations . . . 107

C.2.2 Alternatives . . . 107

D Benchmark methodology 109

List of Tables

1.1 Comparison of C++ Geometry Libraries 6

2.1 Operations on vectors and points in an affine space 12
2.2 Example of forward mode AD 33
2.3 Example of reverse mode AD 34

3.1 Terms describing geometric objects in `wave_geometry` 42
3.2 Time to Evaluate Value and Jacobians of (3.16) 61

4.1 Rules for Semantics of Geometric Operations 73

List of Figures

2.1 Representations of a point and a vector 10
2.2 Rotations as an example of Lie group operations. 25
2.3 Computational graph of $y = 3x_1x_2 + \sin(x_2)$. 33

3.1 Expression tree showing compile-time propagation of types, from bottom to top, for expression (3.2). 38
3.2 Hierarchy of types in `wave_geometry`. 43
3.3 Example of library-inserted conversions. 49
3.4 Computation of the derivative of example (3.2) using forward AD. 55
3.5 Reverse-mode differentiation of example (3.2). 56
3.6 Simplified inheritance diagram for a Dynamic expression. 57
3.7 Example of a dynamically allocated expression graph. 58
3.8 Comparison of time taken to evaluate result and all $N + 1$ Jacobians in a chain of N rotations (3.10). 59
3.9 Time taken to evaluate result and all $N + 1$ Jacobians in a chain of N rotations stored dynamically. 62

4.1 Coordinate-free vector addition. 64
4.2 Vector addition expressed in a coordinate system. 65
4.3 Example of frame descriptor notation for vectors. 66
4.4 Free and bound transformations. 69
4.5 Error message printed by Clang when compiling Listing 4.3. 70

5.1 Simple factor graph for robot localization. 81
5.2 Example of localization problem using our factor graph framework. 82
A.1 Comparison of exponential map functions. 97

List of Symbols

Symbol	Description
a	Font for real scalars
$\mathbf{a}$	Font for real column vectors
$\mathbf{A}$	Font for real matrices
$\boldsymbol{a}$	Font for homogeneous vectors
$\boldsymbol{A}$	Font for homogeneous matrices
$\mathcal{A}$	Font for spaces
$\mathscr{a}, \mathscr{A}$	Font for points on a manifold, other geometric entities independent of representation
AB	Font for Lie groups
$\mathfrak{ab}$	Font for Lie algebras
P, Q, R	Points on an affine space
$\chi(\boldsymbol{x})$	An entity χ parametrized by $\boldsymbol{x}$
$\mathbf{I}_n$	The $n \times n$ identity matrix
$\mathbf{0}_{m\times n}$	The $m \times n$ zero matrix
$\mathbf{J}_{\mathbf{xy}}$	The Jacobian $\partial\mathbf{x}/\partial\mathbf{y}$
$\mathcal{F}_A$	A coordinate frame
${}_A(\cdot)_{BC}$	A physical property of $\mathcal{F}_C$ with respect to $\mathcal{F}_B$, expressed in $\mathcal{F}_A$
$\mathbf{r}_{BC}$	Coordinate-free translation from B to C
${}_A\mathbf{r}_{BC}$	The translation from B to C, expressed in $\mathcal{F}_A$
$\mathbf{R}_{AB}$	The rotation such that ${}_A\mathbf{r}_{BC} = \mathbf{R}_{AB}{}_B\mathbf{r}_{BC}$
$\mathbf{T}_{AB}$	The transformation such that ${}_A\boldsymbol{r}_{AC} = \mathbf{T}_{AB}{}_B\boldsymbol{r}_{BC}$
$\mathbb{A}^n$	n-dimensional affine space
$\mathbb{R}^n$	n-dimensional Euclidean space
$\mathbb{RP}^n$	n-dimensional real projective space
$\mathbb{T}^n$	n-dimensional real oriented projective space
S^n	n-dimensional unit sphere in $\mathbb{R}^{n+1}$
$\mathbb{R}^{m\times n}$	The vector space of real $m \times n$ matrices
G	A Lie group

Symbol	Meaning
$\mathfrak{g}$	A Lie algebra
$SO(3)$	The special orthogonal group
$\mathfrak{so}(3)$	The Lie algebra of $SO(3)$
$SE(3)$	The special Euclidean group
$\mathfrak{se}(3)$	The Lie algebra of $SE(3)$
$\mathbf{C}$	A 3×3 rotation matrix
$\mathbf{q}$	A unit quaternion
$\boldsymbol{\Phi}$	An element of either $SO(3)$ or $SE(3)$
$\boldsymbol{\varphi}$	An element of either $\mathfrak{so}(3)$ or $\mathfrak{se}(3)$
$\mathbf{v}$	An element of $\mathbb{R}^3$, $\mathfrak{so}(3)$, or $\mathfrak{se}(3)$
$\mathcal{N}$	Gaussian distribution
$\approx$	Approximately equal to
$\cong$	Proportional to
$\triangleq$	Definition
$\equiv$	Identity
$:=$	Assignment
$\iff$	If and only if
$\forall$	For all
$\exists$	There exists
$\exists!$	There exists one and only one
$\neg\chi$	Antipode of χ
$\hat{\mathbf{x}}$	Posterior stimate for $\mathbf{x}$
$\check{\mathbf{x}}$	Prior estimate for $\mathbf{x}$

Chapter 1

Introduction

Representation and estimation of geometric states is central to a broad range of problems in robotics and computer vision. Problems such as simultaneous localization and mapping (SLAM) and visual-inertial odometry (VIO) involve estimation of Euclidean motions and of points in projective space. Most approaches rely on optimization over possible states, which requires operations on differentiable manifolds. States are often represented in different coordinate frames, requiring algorithms to keep track of and convert between frames.

Contemporary implementations of robotics algorithms typically use hand-coded, analytically derived derivatives and rely on external C++ libraries, examined in Section 1.1, for numerical and optimization routines. For example, the Eigen (Guennebaud et al., 2010) linear algebra library is often used for matrix operations, sometimes with a specialized library for manifold geometry. These libraries are often used with a separate nonlinear least squares solver, such as Ceres (Agarwal et al., 2010). Other optimization frameworks, such as GTSAM (Dellaert, 2012), provide their own manifold representations, as well as handwritten code for the derivatives of common cost functions. GTSAM also provides automatic differentiation (AD) of arbitrary expressions.

This thesis examines the geometric representations in existing tools and seeks to improve on them. It presents a C++ library for manifold geometry which includes the following features:

- **Manifold operations:** geometric objects such as rotations and poses do not lie on a vector space but on a manifold. The library supports operations on the manifolds of rotations, SO(3), and rigid transformations, SE(3).

- **Geometric semantics:** while geometric objects are stored as an array of coefficients, they must retain their geometric meaning in code. For example, our library makes it impossible to accidentally add a rotation vector to a point, although both are represented as 3D vectors. On the other hand, objects representing 3D rotations are interchangeable regardless of their parametrization.

- **Frame semantics:** a measurement represented in one coordinate frame is not interchangeable with one from another frame. The coordinate frames associated with an object are automatically propagated through operations at compile time.

- **Automatic differentiation:** geometric expressions are differentiable with respect to any variable. Derivatives are *local*, with respect to perturbations in the object's tangent space, and independent of parametrization.

- **Uncertainty representation:** states can be represented with their uncertainty, including covariance between elements of a compound state. Multiple noise models are supported.

- **Optimization:** we present a framework for composing geometric expressions into automatically differentiable cost functions. Combinations of states and cost functions can be represented as a factor graph, which is solved using nonlinear least squares optimization.

- **Flexible storage:** geometric objects can hold parameters of any scalar type, and wrap values provided by external libraries as raw arrays.

- **Extensibility**: the framework is extensible with arbitrary operations, object parametrizations, and geometric spaces.

These features share common goals: type safety, performance, and flexibility. The overarching goal of our library is to make it easier for the programmer to take a step back and observe the mathematical meaning of their computations, without worrying about details such as conversions between parametrizations and minor code optimizations.

While some of these features are offered by existing libraries, none offers all of them. Our main contributions are the automatic differentiation and frame semantics systems. The application of ETs to manifold geometry is itself novel, and this thesis describes several ancillary contributions of our ET implementation, including the storage of temporary objects within expressions, automatic insertion of conversions between geometric representations, dynamic expressions, and the ability to use arbitrary storage types within geometric objects.

This thesis describes the background and implementation of these features, presenting details beyond those offered in Koppel and Waslander (2018). It then describes the targeted application of our library: on-manifold optimization. Here, an additional contribution is a factor graph framework which combines our library's expressions and AD system with Ceres Solver, a widely used tool for optimization introduced in the next section.

1.1 Related work

1.1.1 Automatic differentiation

Automatic differentiation (AD) encompasses a wide body of work in the computer science and machine learning fields. Griewank and Walther (2008) develop the theory of AD, while Verma (2000) and Hoffmann (2016) provide accessible introductions. Baydin et al. (2018) provides a survey of AD implementations.

AD implementations can be divided into several orthogonal groupings: they can use forward or reverse mode, operator overloading or source code transformation, and scalar-valued or matrix-valued operations (Andersson et al., 2012). Most AD libraries in C++, including the popular ADOL-C (Walther and Griewank, 2012), use operator overloading. This method uses AD-aware types for which elementary operations (such as addition and multiplication) are defined to compute derivatives as well as the original function. Aubert et al. (2001), and later Phipps and Pawlowski (2012) (with Sacado) and Hogan (2014) (with Adept), show improvements in efficiency by using expression templates, a technique also used in our library and discussed in Section 3.1. Carpenter et al. (2015) (with the Stan Math Library) shows further improved performance through templates and caching. While we do not intend to compete with these sophisticated, general-purpose AD libraries, we apply some of their techniques to the specific domain of robot state estimation on manifolds.

The above AD implementations use scalar-valued atomic operations, which would consider an operation such as matrix multiplication as a multivariate function of individual coefficients. Andersson et al. (2012) (with CasADi) presents an implementation supporting vector- and matrix-valued operations, which improves efficiency in such cases. Because derivatives of geometric expressions are naturally expressed as matrix operations, our implementation also uses matrix-valued elementary operations.

Today, AD is perhaps most widely used in machine learning, with implementations including Autograd (Maclaurin et al., 2015), PyTorch (Paszke et al., 2017), and Tangent (van Merriënboer et al., 2017). These Python libraries allow differentiation of near-arbitrary

functions with a high level of abstraction and parallel computation on GPUs. Our implementation, while comparatively simple, targets C++ code running on mobile robots with constrained computational budgets, and aims for maximal CPU performance.

Our implementation is closest in principle to Sommer et al. (2013), which defines *block automatic differentiation* on differentiable manifolds and its application to robotics problems. The approach is to build a computation graph of basic operations ("blocks") on manifold elements, and evaluate derivatives using separately-provided optimized code for each operation. This approach is essentially AD with matrix-valued atomic operations, but with a focus on obtaining local Jacobians (defined in Section 2.5.7) of manifold operations.

Sommer et al. (2013) served as the basis[1] for a block AD implementation in GTSAM, which provides reverse-mode differentiation of expression graphs. Our work differs from GTSAM in the use of expression templates. We provide a performance comparison in Section 3.4.

Ceres Solver (Agarwal et al., 2010) provides its own AD implementation which integrates with user-provided cost functions. Because one of our goals is to improve the process of writing cost functions for optimizers such as Ceres, we include it in the performance comparison.

1.1.2 Geometry frameworks

While the library presented in this work can be broadly described as a "geometry library," that term encompasses a huge range of software with diverse applications in computer graphics, 3D modelling, simulation, games, and robotics. At one end of the spectrum are graphics-centred libraries such as OpenGL, Ogre, and VTK. At the other are domain-specific languages for specific robotics applications, such as motion control of industrial robots, as surveyed by Nordmann et al. (2014). While both sets of works contain code for geometric representations, and all embody the geometric theory introduced in Chapter 2, they are typically not suitable for prototyping state estimation algorithms. We limit the scope of our search to open-source C++ libraries commonly used for state representation in mobile robotics research.

Eigen (Guennebaud et al., 2010) is widely used for storage and manipulation of states. Primarily a linear algebra library, it provides an expression template (discussed in Section 3.1) implementation of matrix operations allowing highly optimized, vectorized computation. Similar expression template-based linear algebra libraries include Blaze and Armadillo.

[1] Personal communication with the authors.

Eigen's Geometry module provides transformations and multiple parametrizations of rotations. Although it lacks manifold operations, Eigen is ubiquitous in the fields of computer vision and robotics, and is used internally by most manifold libraries listed here, including ours.

CGAL is a mature and extensive computational geometry library, encompassing areas such as surface reconstruction and higher-dimensional spaces as well as 2D and 3D linear geometry. CGAL offers multiple kernels which determine the numeric types and precision of the calculations used, including exact (non-floating-point) calculations. While the library is unwieldy for our purposes and favours exactness over performance, we take inspiration from its theoretical rigour and its regimented design. For example, points, vectors, and directions are treated as separate primitives. The primitives are represented by different "representation classes" (for example, points can be represented by homogeneous or Cartesian coordinates) and generic code can be written independently of the representation.

Boost.Geometry, part of the widely distributed Boost libraries, is similar, providing generic geometric primitives for arbitrary coordinate systems and precision.

Ceres Solver (Agarwal et al., 2010) is a nonlinear least squares solver which optionally performs AD. Though it does not provide geometric types, it supports on-manifold optimization through its `LocalParameterization`. Ceres' interface uses raw arrays, which can be interpreted as matrices using Eigen's `Map` class. For example, the OKVIS visual-inertial odometry package (Leutenegger et al., 2015) uses Ceres, Eigen, and hand-coded analytic Jacobians.

Sophus implements manifold operations on the Lie groups SO(3) and SE(3), using Eigen internally. Kindr provides manifold operations with a robotics focus, and includes Jacobians of some operations.

The Manifold Toolkit (MTK), introduced by Hertzberg et al. (2013), provides a framework for working on SO(3) and SE(3) as well as arbitrary compound manifolds formed by combining multiple primitives. MTK is built around Eigen, and uses macros to provide the compound manifold feature. While MTK includes Sparse Least Squares on Manifold (SLoM), a library for on-manifold optimization, it uses numerical differentiation to compute Jacobians. Hertzberg et al. (2013) is notable for presenting the theory of on-manifold optimization, including the $\boxplus$ and $\boxminus$ operators which we rely on in Section 2.5.7.

The Kinematics and Dynamics Library (KDL) provides geometric classes, focusing on kinematic chains. While it does not explicitly provide manifold operations, it supports pose interpolation and Jacobians of kinematic chains.

The Mobile Robot Programming Toolkit (MRPT) is a collection of libraries for robotics

Table 1.1: Comparison of C++ Geometry Libraries

Library	Type[a]	Any scalar type[b]	Manifold ops.	Manifold Jacobians	Maps	Expr. Jacobians	Frame checking	URL
Armadillo	LA	✓			✓			arma.sourceforge.net
Boost.Geometry	CG	✓						boost.org
CGAL	CG	✓						cgal.org
Blaze	LA	✓						bitbucket.org/blaze-lib/blaze
Eigen	LA	✓			✓			eigen.tuxfamily.org
g2o	RO		✓	✓				openslam.org/g2o.html
GTSAM	RO		✓	✓		✓		bitbucket.org/gtborg/gtsam
KDL	KC		[p]	[p]			[k]	orocos.org/kdl
Kindr	LG	✓	✓	[p]				github.com/ANYbotics/kindr
MRPT	RO		✓	✓				mrpt.org
MTK	LG	✓	✓					openslam.org/MTK.html
Sophus	LG	✓	✓		✓			github.com/strasdat/Sophus
tf2	SG						[t]	wiki.ros.org/tf2
This work	LG	✓	✓	✓	✓	✓	✓	github.com/wavelab/wave_geometry

[a] LA: linear algebra, CG: computational geometry, LG: Lie geometry, RO: robotics algorithms and optimization, KC: kinematic chains, SG: scene graph
[b] "Any scalar" also indicates a header-only library.
[p] Partial.
[k] The separate Geometric Relations Semantics library provides runtime frame checking for KDL.
[t] tf2 performs runtime frame conversion, but does not check calculations.

applications including SLAM, computer vision, and motion planning. It provides a 3D geometry library, including Jacobians for operations on SE(3).

GTSAM (Dellaert, 2012) and g2o (Kümmerle et al., 2011) are frameworks for nonlinear optimization based on factor graphs. They include their own implementations of manifold geometry, including Jacobians.

Table 1.1 presents a non-exhaustive comparison of these libraries. This comparison is narrowly focused on the features we are targeting—of course, these mature libraries have a wide range of other features and use cases.

In Table 1.1, *any scalar type* refers to support for numeric types other than `float` and `double`. This support makes functions compatible with many general-purpose AD libraries. Here, *any scalar* support also indicates a *header-only* library. While header-only libraries often increase compile time compared to precompiled libraries, they are more flexible and can produce more highly optimized code. *Maps*, sometimes called *views*, allow zero-overhead reuse of raw memory buffers, as demonstrated by Eigen's `Map` class. Maps allow efficient

interfacing with third-party libraries.

Frame checking, the topic of Chapter 4, is the use of software to catch mistakes in geometric calculations by tracking the coordinate frame semantics associated with each object. De Laet et al. (2013b,a) describe this idea and extend KDL with their Geometric Relations Semantics software. DeRose (1989) addresses the same issue but takes a slightly different approach in defining a "coordinate-free abstract data type." We will return to these works in Chapter 4. The most significant difference between these libraries and ours is that they perform checks at runtime, while our goal is zero-overhead checks at compile time.

The tf2 library represents a different class of libraries used by roboticists to track coordinate frames. Part of the Robot Operating System (ROS), the library stores transformations between frames in a data structure resembling a scene graph, which is often found in 3D graphics and simulation libraries (Foote, 2013). tf2 updates this graph dynamically and asynchronously, and computes the transformation between any two frames on request. However, the purpose of tf2 and other scene graphs differs from that of frame checking software: its geometric primitives do not propagate frame information and cannot catch mistakes in the user's subsequent calculations.

Our implementation is inspired by long-existing methods for compile-time dimensional analysis, described by Barton and Nackman (1994). We are not aware of prior works that apply these techniques to coordinate frames.

1.2 Thesis structure

This thesis is organized as follows:

- Chapter 2 provides a theoretical introduction to geometric representations, Lie theory, and automatic differentiation.
- Chapter 3 presents our C++ library, `wave_geometry`. The chapter considers the design of the library, details its implementation in C++, and describes the supported operations. This chapter also presents experimental results measuring runtime performance of our automatic differentiation system.
- Chapter 4 formulates a set of rules for coordinate frame semantics, and demonstrates how semantics checking can be implemented in C++.

- Chapter 5 demonstrates an application of `wave_geometry` to a nonlinear least squares state estimation problem. It presents a factor graph framework combining our geometric expressions and AD system with Ceres Solver.
- Chapter 6 presents conclusions.

Chapter 2

Background

This chapter begins with a summary of the mathematics of translations, rotations, and transformations, and the spaces in which they live: affine spaces, projective spaces, and Lie groups. It focuses on the entities and representations relevant to state estimation and computer vision problems. This chapter draws upon more comprehensive treatments (Hartley and Zisserman, 2004; Gallier, 2011; Förstner and Wrobel, 2016; Barfoot, 2017).

This chapter is organized by type of geometric entity, with each section giving an overview of that entity's theoretical background. Section 2.2 presents points and vectors, and introduces the idea of coordinate-free geometry. Section 2.3 presents homogeneous coordinates and projective spaces. Section 2.4 introduces Euclidean space and rigid motions. Section 2.5 defines manifolds, Lie groups, and Lie algebras, and examines the groups of rotations and rigid transformations. We then consider differentiation: Section 2.5 covers perturbations and derivatives on manifolds and Section 2.6 introduces the principles of automatic differentiation.

2.1 Notation

This thesis works toward a concrete application in which geometric entities are encoded as bits in memory. It is tempting, then, to jump directly to practical concerns: by its nature, a computer program must handle orderings of coefficients, changes of coordinate frames, and finite precision computations. However, as one goal of our library is to make it easier for the programmer to focus on the mathematical meaning of these computations. The library seeks to separate types of geometric entities from their parametrizations, and their semantics from

their value. Concordantly, this chapter strives to distinguish between geometric entities and particular representations and, as Gallier (2011) exhorts, "use coordinate systems only when needed."

Following Förstner and Wrobel (2016), we use calligraphic letters to distinguish geometric entities from representations. For example, a 3D point χ may be expressed as a vector $\mathbf{x} \in \mathbb{R}^3$, or in homogeneous coordinates as $\boldsymbol{x} \in \mathbb{R}^4$. The notation $\chi(\boldsymbol{x})$ means "χ, which is represented by $\boldsymbol{x}$."

The symbol $\mathbf{v}_{BC}$ denotes a (vector) physical quantity of object C with respect to object B. That quantity expressed in a particular coordinate frame $\mathcal{F}_A$ is written ${}_A\mathbf{v}_{BC}$. $\mathbf{R}_{AB}$ denotes a rotation that takes points expressed in $\mathcal{F}_B$ and re-expresses them in $\mathcal{F}_A$. This notation is explained in detail in Chapter 4. The idea of coordinate-free geometry is introduced in the next section.

2.2 Points, vectors, and spaces

This section introduces the geometric background of points, translations, and frames in a coordinate-free manner. It draws from Gallier (2011), Goldman (1985, 2002), and DeRose (1989), attempting to provide a simplified overview without glossing over the difference between affine and Euclidean spaces, as is common. While we mostly work in Euclidean spaces, we first introduce the more general affine spaces.

2.2.1 Points and vectors

While the typical matrix math library may not differentiate between points and vectors, they are distinct geometric entities, as illustrated in Fig. 2.1.

Figure 2.1: Representations of a point P and a vector $\mathbf{v}$.

Per Goldman (1985), "intuitively, points have position but not direction or length; vectors have direction and length but not position." Conflation of points and vectors arises because points can be represented by elements of a vector space, such as $\mathbb{R}^3$.

Vectors are subject to the operations of vector algebra: addition, subtraction, scalar multiplication, dot product, and cross product. Definitions of these operations can be found in Goldman (1985).

2.2.2 Affine spaces

Definition 2.1. A nonempty *affine space* is a pair $(\mathcal{P}, \mathcal{V})$, where $\mathcal{P}$ is a nonempty set of points and $\mathcal{V}$ is a set of vectors forming a vector space, along with an action $+ : \mathcal{P} \times \mathcal{V} \rightarrow \mathcal{P}$ which has the properties (Gallier, 2011):

$$P + \mathbf{0} = P, \qquad \forall P \in \mathcal{P} \tag{2.1}$$

$$(P + \mathbf{u}) + \mathbf{v} = P + (\mathbf{u} + \mathbf{v}), \qquad \forall P \in \mathcal{P},\ \forall \mathbf{u}, \mathbf{v} \in \mathcal{V} \tag{2.2}$$

$$\text{There is a unique } \mathbf{u} \in \mathcal{V} \text{ such that } P + \mathbf{u} = Q, \qquad \forall P, Q \in \mathcal{P}. \tag{2.3}$$

Equivalently, we can define a subtraction operation $- : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{V}$, satisfying

$$\text{There is a unique } \mathbf{u} \in \mathcal{V} \text{ such that } \mathbf{u} = P - Q, \qquad \forall P, Q \in \mathcal{P}. \tag{2.4}$$

In this section, we denote the vector between two points as $P - Q \triangleq \overrightarrow{PQ}$. An alternative definition of the affine space starts with properties of the subtraction operation and defines addition and other operations from there (DeRose, 1989).

Although points can be represented in a vector space, they are better dealt with in an affine space. In an affine space, there is no privileged origin and algebra can be performed in an *intrinsic* or coordinate-free manner.

Definition 2.2. A *coordinate-free* algebra is one whose operations and entities are defined independently of any coordinate system. While we must necessarily choose a coordinate system to compute results, the geometric relationships expressed by coordinate-free operations don't depend on our choice.

While addition can be applied to pairs of vectors and between vectors and points, it does not make sense to add two points. Intuitively, we can see the that result of adding two points would depend on the coordinate system. As a simple example, consider adding points represented in $\mathbb{R}^1$: the result of $(0) + (1)$ is equal to the second point, while $(1) + (2)$ is not, although the pairs differ only by a change of origin. Table 2.1 summarizes well-defined and undefined operations in an affine space.

Table 2.1: Operations on vectors and points in an affine space

Operation	Legal expression	Undefined expression
Addition	$\mathbf{w} = \mathbf{u} + \mathbf{v}$	$P + Q$
	$Q = P + \mathbf{u}$	
Subtraction	$\mathbf{w} = \mathbf{u} - \mathbf{v}$	$\mathbf{u} - P$
	$\mathbf{w} = P - Q$	
	$Q = P - \mathbf{u}$	
Scalar multiplication	$\mathbf{w} = c\mathbf{u} = \mathbf{u}c$	cP

P, Q are points; $\mathbf{u}, \mathbf{v}, \mathbf{w}$ are vectors; c is scalar. Adapted from Goldman (1985).

Somewhat surprisingly, although $P + Q$ is undefined, $\frac{P+Q}{2}$ is a meaningful expression. It is an example of an *affine combination*, a linear combination with the added requirement that scalar weights add to 1. Here, it produces the midpoint of P and Q independently of any choice of coordinates, and can be written using the operations we have defined: (Goldman, 2000)

$$\frac{P+Q}{2} = P + \frac{1}{2}(Q - P). \tag{2.5}$$

2.2.3 Affine frames

To obtain numerical results from the relationships we derive, we must eventually introduce coordinates. Because affine spaces have no privileged origin, points are assigned coordinates using *frames*.

Definition 2.3. Given an affine space $(\mathcal{P}, \mathcal{V})$, an *affine frame with origin O* is a collection of points $(O, A_1, \ldots, A_n)$ such that the vectors $(\overrightarrow{OA_1}, \ldots, \overrightarrow{OA_n}) \triangleq (\mathbf{e}_1, \ldots, \mathbf{e}_n)$ are a basis for $\mathcal{V}$. The collection $(O, \mathbf{e}_1, \ldots, \mathbf{e}_n)$ is also called an affine frame with origin O. Then, every point $X \in \mathcal{P}$ can be expressed as:

$$X = O + x_1\mathbf{e}_1 + \ldots + x_n\mathbf{e}_n, \tag{2.6}$$

where $(x_1, \ldots, x_n)$ is a unique family of scalars called the *coordinates of X w.r.t the affine frame* $(O, \mathbf{e}_1, \ldots, \mathbf{e}_n)$. (Gallier, 2011)

For brevity, we call an affine frame simply a *frame* and denote it as $\mathcal{F}_O$, where O may be replaced with the point serving as the frame's origin or another associated object. We will refine the idea and terminology of frames in Chapter 4.

2.2.4 Affine transformations

Definition 2.4. An *affine transformation*, or affine map, is a transformation which maps points to points and preserves affine combinations. That is, for any affine combination of points (with $\sum_i c_i = 1$),

$$\mathcal{A}\left(\sum_i c_i P_i\right) = \sum_i c_i \mathcal{A}(P_i). \tag{2.7}$$

Every affine transformation $\mathcal{A}$ has a unique associated linear map $\vec{\mathcal{A}}$ which maps vectors to vectors,[1] such that

$$\mathcal{A}(Q + \mathbf{v}) = \mathcal{A}(Q) + \vec{\mathcal{A}}(\mathbf{v}) \tag{2.8}$$

for every $Q \in \mathcal{P}$ and every $\mathbf{v} \in \mathcal{V}$. Substituting $Q = O$ and $\mathbf{v} = P - O$, we obtain

$$\mathcal{A}(P) = \mathcal{A}(O) + \vec{\mathcal{A}}(\overrightarrow{OP}), \tag{2.9}$$

showing that "every affine map is determined by the image of any point and a linear map," (Gallier, 2011): that the result of $\mathcal{A}$ on any point P can be obtained by knowing the result of $\mathcal{A}$ on a single point O and how $\vec{\mathcal{A}}$ acts on vectors.

Affine transformations can be expressed in matrix form. Subtracting O from (2.9) and substituting back $\overrightarrow{OP} = \mathbf{v}$ gives

$$\mathcal{A}(P) - O = \vec{\mathcal{A}}(\mathbf{v}) + \mathcal{A}(O) - O, \tag{2.10}$$

$$\overrightarrow{O\mathcal{A}(P)} = \vec{\mathcal{A}}(\mathbf{v}) + \overrightarrow{O\mathcal{A}(O)}. \tag{2.11}$$

Let $\mathcal{F}_O$ be an affine frame with origin O and an arbitrary basis. Then we can define $\mathbf{A} \in \mathbb{R}^{n\times n}$ as the matrix of the linear map associated with $\mathcal{A}$ over the chosen basis, and $\mathbf{x}, \mathbf{b}, \mathbf{y} \in \mathbb{R}^n$ as the the coordinates of $\mathbf{v}$, $\mathcal{A}(O)$ and $\mathcal{A}(P)$ in $\mathcal{F}_O$, respectively. The result is

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b}. \tag{2.12}$$

Note that although $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{b}$ are described as coordinates with respect to a frame, we are still demonstrating coordinate-free geometry: the choice of basis was arbitrary and we used only the operations listed in Table 2.1. Equation (2.12) is valid no matter what frame we choose.

[1] This definition follows Gallier (2011). Goldman (2002) omits $\vec{\mathcal{A}}$ and defines an affine map that works on both points and vectors .

This equation can be converted into a linear form by adding an $(n+1)$th coefficient to the points' coordinate vectors:

$$\begin{bmatrix} \mathbf{y} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}. \tag{2.13}$$

In $\mathbb{R}^3$, the result is the familiar rigid transformation matrix. The extra coefficient is called the *affine coordinate*, and is set to 1 for points and 0 for vectors to enforce that vectors are not affected by translations (Goldman, 2002). These two categories can also be called *bound* and *unbound* vectors (Kelly, 2013), and we will revisit them in Section 4.1.4. The four-coordinate representation can be extended to produce homogeneous coordinates.

2.3 Homogeneous coordinates

A point in $\mathbb{R}^3$ can be expressed using a 4-vector of *homogeneous coordinates*,

$$\boldsymbol{p} = \begin{bmatrix} \lambda x \\ \lambda y \\ \lambda z \\ \lambda \end{bmatrix} \tag{2.14}$$

where $\lambda \in \mathbb{R}$, $\lambda \neq 0$.

More generally, homogeneous coordinates $\boldsymbol{x} \in \mathbb{R}^{n+1}$ can be used to represent any point χ in $\mathbb{R}^n$ including points at infinity (Hartley and Zisserman, 2004, Section 1.1). A point χ can be written as

$$\boldsymbol{x} = \begin{bmatrix} \mathbf{x}_0 \\ x_h \end{bmatrix}, \tag{2.15}$$

where x_h is the *homogeneous part*, and $\mathbf{x}_0$ is the inhomogeneous or *Euclidean part* (Förstner and Wrobel, 2016). Points at infinity have $x_h = 0$.

Homogeneous coordinates are invariant with respect to multiplication by a nonzero scalar: $\boldsymbol{x}$ and $\boldsymbol{y} = \lambda \boldsymbol{x}$ represent the same geometric entity χ.

2.3.1 Projective spaces

The *projective space* $\mathbb{RP}^n$ is defined as (Förstner and Wrobel, 2016, Section 5.3.4) the space of all n-dimensional points χ with homogeneous coordinates $\boldsymbol{x} \in \mathbb{R}^{n+1} \setminus \mathbf{0}$, with the equivalence relation

$$\chi \equiv y \iff \exists \lambda,\ \boldsymbol{x} = \lambda \boldsymbol{y}. \tag{2.16}$$

This equivalence relation can be written without an explicit scalar factor as $\boldsymbol{x} \cong \boldsymbol{y}$, or simply as $\boldsymbol{x} = \boldsymbol{y}$ (Förstner and Wrobel, 2016).

Oriented projected spaces

Under the definition above, $\boldsymbol{x}$ and $-\boldsymbol{x}$ are equivalent. However, it is often useful to distinguish these two as separate points, for example, to determine whether a point is in front of or behind a camera image plane. The *oriented projected space* $\mathbb{T}^n$ is the space of all n-dimensional points χ with homogeneous coordinates $\boldsymbol{x} \in \mathbb{R}^{n+1} \setminus \mathbf{0}$, with the relation

$$\chi \equiv y \iff \exists \lambda > 0,\ \boldsymbol{x} = \lambda \boldsymbol{y}, \tag{2.17}$$

which differs from relation (2.16) by requiring the scaling factor between equivalent points to be positive. Each point $\chi(\boldsymbol{x})$ has a distinct *antipode*, $\neg\chi(-\boldsymbol{x})$.

2.3.2 Normalization of homogeneous coordinates

Because of their scale invariance, homogeneous coordinates are not unique. The Euclidean and spherical normalizations are two choices of constraints commonly used to impose uniqueness (Förstner and Wrobel, 2016, Section 5.8).

Euclidean normalization

A Euclidean normalized point has homogeneous part 1:

$$\boldsymbol{x}^e = \frac{\boldsymbol{x}}{x_h} = \frac{1}{x_h}\begin{bmatrix}\mathbf{x}_0 \\ x_h\end{bmatrix} = \begin{bmatrix}\mathbf{x} \\ 1\end{bmatrix} \tag{2.18}$$

Euclidean normalization is used for obtaining the Euclidean part of the point in inhomogeneous coordinates. It cannot be applied to points at infinity.

Note that we focus on homogeneous points. Homogeneous coordinates can also represent lines and planes, for which Euclidean normalization is slightly different (Förstner and Wrobel, 2016).

Spherical normalization

Spherical normalization sets the homogeneous vector to lie on a unit sphere by requiring $||\boldsymbol{x}|| = 1$. For example, the spherically normalized homogeneous coordinates of a 3D point χ are

$$\boldsymbol{x}^s = \frac{\boldsymbol{x}}{||\boldsymbol{x}||} = \frac{1}{\sqrt{x^2 + y^2 + z^2 + w^2}} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \tag{2.19}$$

Under this constraint, 3D points lie on the the unit sphere S^3 embedded in $\mathbb{R}^4$. This sphere is a closed manifold (see Section 2.5.2), making it useful for optimization.

Spherical normalization has the effect of converting image coordinates to ray directions; when dealing with measurements uncertainty, it converts "from uncertain image coordinates to uncertain image ray directions" (Förstner and Wrobel, 2016, Section 10.2.2).

Note that the spherical normalization is not a unique representation because of the antipodal equivalence relation

$$\boldsymbol{x}^s \cong -\boldsymbol{x}^s. \tag{2.20}$$

In fact, it is a *unit quaternion* representation of $\mathbb{RP}^3$ and closely related to rotations on SO(3), as will be discussed in Section 2.5.9.

2.4 Rigid motions

2.4.1 Euclidean spaces

We have not yet introduced any concept of distance or angle between vectors, because they are not present in affine geometry. We turn to Euclidean geometry to introduce these *metric* notions.

Definition 2.5. A *Euclidean affine space* is an affine space $(\mathcal{P}, \mathcal{V})$ whose vector space $\mathcal{V}$ is a real vector space equipped with an inner product. For our purposes, the vector space is

$\mathbb{R}^n$, whose inner product is the familiar dot product

$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n \tag{2.21}$$

For brevity, we call a Euclidean affine space a *Euclidean space*, although that term formally refers only to $\mathcal{V}$ (Gallier, 2011).

Note on notation

Because measurements of distance and angle are useful in robotics, consider Euclidean spaces the default for the remainder of this work: *space* refers to a Euclidean space, and *frame* refers to a Euclidean frame. Because affine geometry is a generalization of Euclidean geometry, our earlier results are still valid. Analogous definitions can be made where necessary: a *Euclidean frame* has the same definition as an affine frame, but in a Euclidean space.

Following Chirikjian (2011), we will denote n-dimensional Euclidean space simply as $\mathbb{R}^n$ and leave implicit the distinction between points and vectors. For example, it is understood that a "translation on $\mathbb{R}^3$" is a vector that acts on points in 3-dimensional Euclidean space.

2.4.2 Distances and orthogonality

Two metric notions help define the familiar robotics concepts of rotations and poses: distance and orthogonality.

Definition 2.6. The distance between any two points $P, Q \in \mathcal{P}$ is $||\overrightarrow{PQ}||$, the *Euclidean norm* of $\overrightarrow{PQ}$. The Euclidean norm of a vector $\mathbf{x} = [x_1, \ldots, x_n]^\intercal$ is

$$||\mathbf{x}|| = \sqrt{\mathbf{x} \cdot \mathbf{x}} = \sqrt{x_1^2 + \ldots + x_n^2}. \tag{2.22}$$

A function which preserves distance is called an *isometry*.

Definition 2.7. Two vectors $\mathbf{u}$ and $\mathbf{v}$ are *orthogonal* if

$$\mathbf{u} \cdot \mathbf{v} = 0. \tag{2.23}$$

A matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is an *orthogonal matrix* if

$$\mathbf{A}\mathbf{A}^\intercal = \mathbf{A}^\intercal \mathbf{A} = \mathbf{I}_{n \times n}, \tag{2.24}$$

or equivalently if $\mathbf{A}^\mathsf{T} = \mathbf{A}^{-1}$.

2.4.3 Rotations and Euclidean motions

Definition 2.8. *Rigid motions* are affine transformations which preserve the Euclidean norm. That is, they are isometries, preserving the distance between points.

Rigid motions of an n-space form the group $\mathrm{Is}(n)$. Their associated linear maps form the group of orthogonal transformations, $\mathrm{O}(n)$.

Definition 2.9. *Rotations* are elements of $\mathrm{O}(n)$ with $\det(\mathbf{A}) = 1$, where $\det(\mathbf{A})$ is the determinant of the matrix representation of the linear map. They form the Lie group $\mathrm{SO}(n)$, the *special orthogonal group*. Its elements are also called *proper rotations* or *proper orthogonal transformations*. Groups and Lie groups are defined in Section 2.5.3, below.

The other subgroup of $\mathrm{O}(n)$, with $\det(\mathbf{A}) = -1$, holds *improper* rotations, which include a reflection—they do not preserve handedness. Proper rotations are handedness-preserving linear isometries of Euclidean space.

Definition 2.10. *Euclidean motions* are elements of $\mathrm{Is}(n)$ whose associated linear map is a proper rotation. They form the Lie group $\mathrm{SE}(n)$, the *special Euclidean group*, of handedness-preserving isometries of Euclidean space. They are also called *proper rigid transformations*.

2.5 Rotations and Euclidean motions

In robotics, we are mostly concerned with rotations and Euclidean motions of three-dimensional points, comprising the Lie groups SO(3) and SE(3). This section presents a summary of these Lie groups and their Lie algebras based on Chirikjian (2011), Gallier (2011), and Barfoot (2017). This section starts with an introductory description of the Lie group of rotations, SO(3). Sections 2.5.2 to 2.5.5 define relevant concepts in the theory of manifolds, Lie groups, and Lie algebras, and Section 2.5.7 defines derivatives on Lie groups. Sections 2.5.8 and 2.5.11 return to concrete examples, covering operations and derivatives on SO(3) and SE(3).

2.5.1 The rotations SO(3)

SO(3) is the group of proper rotations on $\mathbb{R}^3$, and "is simply the set of valid rotation matrices": (Barfoot, 2017)

$$\mathrm{SO}(3) = \left\{ \mathbf{C} \in \mathbb{R}^{3\times 3} : \mathbf{C}\mathbf{C}^{\mathsf{T}} = \mathbf{I}_3,\ \det(\mathbf{C}) = 1 \right\}. \tag{2.25}$$

Note that rotation matrices are parameter-free geometric objects lying on a three-dimensional manifold, and should not be thought of as having 9 parameters.

SO(3)'s tangent space is described by its *Lie algebra*, $\mathfrak{so}(3)$, the space of skew-symmetric matrices which can be denoted

$$\boldsymbol{\varphi} = \boldsymbol{\phi}^{\times} = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \end{bmatrix}^{\times} = \begin{bmatrix} 0 & -\phi_3 & \phi_2 \\ \phi_3 & 0 & -\phi_1 \\ -\phi_2 & \phi_1 & 0 \end{bmatrix}. \tag{2.26}$$

The cross operator $\times$ converts 3-vectors into matrices such that $\mathbf{a} \times \mathbf{b} = (\mathbf{a}^{\times})\mathbf{b}$. In fact, any element $\boldsymbol{\varphi}$ of $\mathfrak{so}(3)$ can be written as a linear combination of matrices

$$\boldsymbol{\varphi} = \sum_{i=1}^{3} \lambda_i \mathbf{E}_i, \tag{2.27}$$

for $\lambda_i \in \mathbb{R}$ and $\mathbf{E}_i = \mathbf{e}_i^{\times}$, where

$$\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix},\ \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},\ \mathbf{e}_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{2.28}$$

The matrices $\mathbf{E}_i$ form a basis for $\mathfrak{so}(3)$, and are called *generators* of SO(3). The vector $\boldsymbol{\phi} = \boldsymbol{\varphi}^{\vee}$ is called a *rotation vector*, and represents a rotation of angle $\theta = \|\boldsymbol{\phi}\|$ about an axis $\mathbf{a} = \boldsymbol{\phi}/\theta$.

Elements of $\mathfrak{so}(3)$ are related to SO(3) by the *exponential map*, $\exp : \mathfrak{so}(3) \to \mathrm{SO}(3)$. For small rotations, this map is bijective, and its inverse is the logarithmic map, $\log : \mathrm{SO}(3) \to \mathfrak{so}(3)$. In the context of software, elements of $\mathfrak{so}(3)$ are represented directly as rotation vectors, and the cross operator may be omitted: we can write $\exp(\boldsymbol{\phi}) \triangleq \exp\left(\boldsymbol{\phi}^{\times}\right)n \in \mathrm{SO}(3)$.

Before going further, we define several key concepts related to Lie groups.

2.5.2 Manifolds and charts

A manifold is a surface which locally resembles Euclidean space at every point. An example of a manifold is the unit sphere in 3D space, S^2, described as the subset of $\mathbb{R}^3$ satisfying

$$x_1^2 + x_2^2 + x_3^2 = 1. \tag{2.29}$$

Like the surface of the Earth, S^2 locally resembles a Euclidean plane. It is a two-dimensional manifold *embedded* in $\mathbb{R}^3$, meaning there is an injective map $\mathcal{m} : S^2 \to \mathbb{R}^3$ which describes the embedded manifold in the higher-dimensional space. The sphere S^3 embedded in $\mathbb{R}^4$ is another example of a manifold, which will be relevant in Section 2.3. Trivially, Euclidean space is itself a manifold.

A more detailed, but still informal, definition of a manifold follows. Formal definitions can be found in Chirikjian (2009, Chapter 7) and Gallier (2011, Chapter 18).

Definition 2.11. Let χ be any point on an m-dimensional manifold $\mathcal{M}$. We can choose a local neighbourhood $\mathcal{U}$ of χ which is a set of nearby points. For example, given a distance function $\rho : \mathcal{M} \times \mathcal{M} \to \mathbb{R}$, $\mathcal{U}$ can be defined as the set of points $\mathcal{u}$ such that $\rho(\chi, \mathcal{u}) < \epsilon$ for some $\epsilon \in \mathbb{R} > 0$. Let ϕ be an injective map $\phi : \mathcal{U} \to \mathcal{V}$, where $\mathcal{V}$ is an open subset of $\mathbb{R}^m$. A pair $(\mathcal{U}, \phi)$ is a *chart* about χ.

A collection of charts chosen so that each point falls into at least one neighbourhood is called an *atlas*.[2] Where a pair of neighbourhoods $\mathcal{U}_i$ and $\mathcal{U}_j$ overlap, the mapping between charts $\phi_i \circ \phi_j^{-1}$ must be continuous.

A manifold, then, is a topological space with an atlas. If for each overlapping region, $\phi_i \circ \phi_j^{-1}$ is differentiable with respect to any coordinate system chosen for the Euclidean space $\mathbb{R}^m$, $\mathcal{M}$ is a *differentiable manifold.* If each $\phi_i \circ \phi_j^{-1}$ is infinitely differentiable, $\mathcal{M}$ is a *smooth manifold.*

For every point, $\phi(\chi) = \mathbf{q}$, where $\mathbf{q} \in \mathbb{R}^m$ is the vector of *local coordinates* of χ (with respect to that chart). The inverse $\phi^{-1}(\mathbf{q})$ is a *local parametrization* of $\mathcal{M}$ at χ.[3]

In the case of S^2, a chart is intuitively understood as a map, showing a flattened view of part of the globe. Any set of charts spanning the whole globe—for example, a pair of polar projections, each covering one hemisphere—forms an atlas.

[2] Chirikjian defines an atlas as a collection of all possible compatible charts, which other works (such as Norton, 1993) distinguish as a *maximal* or *complete* atlas.

[3] Some works take an inverse view and use ϕ for the local parametrization and ϕ^{-1} for the chart.

2.5.3 Lie groups and Lie algebras

Definition 2.12. A *group* is a nonempty set G along with a binary operation $\circ : G \times G \to G$ with the requirements:

1. There is an identity element $e \in G$ such that $e \circ g = g \circ e = g$, $\forall g \in G$.
2. Every element $g \in G$ has an inverse $g^{-1} \in G$ such that $g^{-1} \circ g = g \circ g^{-1} = e$.
3. The operation is associative: $(g_1 \circ g_2) \circ g_3 = g_1 \circ (g_2 \circ g_3)$, $\forall g_1, g_2, g_3 \in G$.

The operation $\circ$ is the *group operation*, sometimes called *composition*. The operation $(\cdot)^{-1} : G \to G$ is called the *inverse map*.

Definition 2.13. A *Lie Group* is a group whose set G is a manifold, and whose group operation and inverse map are smooth[4].

Because it is a manifold, every element g of G has a *tangent space* $\mathcal{T}_g G$ which is a vector space.

The Lie groups we are interested in are *matrix Lie groups*, for which the set G is a smooth (actually, analytic) manifold, its elements are $n \times n$ matrices, and the group operation is matrix multiplication. It follows that the identity element of any matrix Lie group is the identity matrix $\mathbf{I}_n$. While the matrices can be complex, we consider only real matrices.

Definition 2.14. A (real) *Lie algebra* $\mathfrak{g}$ is a vector space together with an operation $[\cdot, \cdot] : \mathfrak{g} \times \mathfrak{g} \to \mathfrak{g}$ called the *Lie bracket* of $\mathfrak{g}$. The Lie bracket must satisfy certain properties for all $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathfrak{g}$ and $\lambda \in \mathbb{R}$:

Bilinearity:
$$[\mathbf{A} + \mathbf{B}, \mathbf{C}] = [\mathbf{A}, \mathbf{C}] + [\mathbf{B}, \mathbf{C}], \tag{2.30a}$$
$$[\mathbf{A}, \mathbf{B} + \mathbf{C}] = [\mathbf{A}, \mathbf{B}] + [\mathbf{A}, \mathbf{C}], \tag{2.30b}$$
$$[\lambda \mathbf{A}, \mathbf{B}] = [\mathbf{A}, \lambda \mathbf{B}] = \lambda [\mathbf{A}, \mathbf{B}]. \tag{2.30c}$$
Alternativity:
$$[\mathbf{A}, \mathbf{A}] = \mathbf{0}. \tag{2.30d}$$
Jacobi identity:
$$[\mathbf{A}, [\mathbf{B}, \mathbf{C}]] + [\mathbf{C}, [\mathbf{A}, \mathbf{B}]] + [\mathbf{B}, [\mathbf{C}, \mathbf{A}]] = \mathbf{0}. \tag{2.30e}$$

It follows that the Lie bracket is anticommutative:

$$[\mathbf{A}, \mathbf{B}] = -[\mathbf{B}, \mathbf{A}]. \tag{2.31}$$

The cross product on $\mathbb{R}^3$ is a familiar example of a Lie bracket.

[4] Formally, these functions must also be *analytic*, meaning the Taylor series at each point describes the neighbourhood of that point. In practice most smooth functions meet this requirement (Chirikjian, 2011).

2.5.4 Exponential and logarithmic maps

The *exponential map* is a map from a Lie algebra to its Lie group,

$$\exp : \mathfrak{g} \to G. \tag{2.32}$$

The Lie algebra of a Lie Group G is the set of all $\mathbf{X}$ such that $\exp(\mathbf{X}) \in G$.

The *logarithmic map* is a map in the other direction,

$$\log : G \to \mathfrak{g}. \tag{2.33}$$

For Matrix Lie groups, these operations rely on the *matrix exponential*, a function $\exp : \mathbb{R}^{n\times n} \to \mathbb{R}^{n\times n}$. It is defined as the series

$$\exp(\mathbf{A}) \triangleq e^{\mathbf{A}} = \sum_{p=0}^{\infty} \frac{1}{p!}\mathbf{A}^p, \tag{2.34}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\mathbf{A}^0 \triangleq \mathbf{I}_n$. It can be proven that the series converges. For subsets of real matrices, such as rotations, it can be evaluated in closed form. There is also a *matrix logarithm* $\log : \mathbb{R}^{n\times n} \to \mathbb{R}^{n\times n}$, such that

$$\exp(\log(\mathcal{g})) \equiv \mathcal{g}. \tag{2.35}$$

Despite this identity, the logarithmic map is not the inverse of the exponential map in general. The exponential map is well-defined for every element of $\mathfrak{g}$, while the logarithmic map is only defined in a neighbourhood of the identity element of G. For SO(3) and SE(3), both operations are well-defined for all $\mathcal{g}$ but $\log(\exp(\mathbf{A})) \equiv \mathbf{A}$ is only true near identity.

2.5.5 The tangent space and adjoints

The Lie algebra of a Lie group G is the tangent space at its identity element $\mathcal{e}$, denoted $\mathcal{T}_{\mathcal{e}}G$. The tangent space "linearizes" the Lie group. Specifically, $\mathcal{T}_{\mathcal{e}}G$ is the space of tangent vectors to smooth paths in G as they pass through $\mathcal{e}$ (Stillwell, 2008). We can write such a path as

$$\mathcal{g}(t) = e^{t\mathbf{X}} \tag{2.36}$$

where $\mathcal{g}(t) \in G$ for some $t \in \mathbb{R}$ and $\mathbf{X} \in \mathfrak{g}$. The tangent vector is $\frac{\mathrm{d}}{\mathrm{d}t}e^{t\mathbf{X}}\big|_0 = \mathbf{X}$. If we do not already know what $\mathfrak{g}$ is, we can derive it by finding all $\mathbf{X}$ which satisfy (2.36).

Equation (2.36) can be considered interpolation from $\mathcal{e}$ (at $t = 0$) to $\exp(\mathbf{X})$ (at $t = 1$) along a curve on G. For matrix Lie groups, we can approximate $\mathcal{g}(t) \approx \mathbf{I} + t\mathbf{X}$ for small t (Chirikjian, 2011). Stillwell (2008) points out a property of the exponential which helps illustrate its connection to interpolation:

$$\exp(\mathbf{X}) = \lim_{n \to \infty} \left(\mathbf{I} + \frac{\mathbf{X}}{n} \right)^n \tag{2.37}$$

Using a very large n produces an *infinitesimal* element of G, which is iterated n times to produce $\exp(\mathbf{X})$.

As Equation (2.27) shows for SO(3), group elements can be written as a linear combination of basis elements

$$\mathcal{g} = \exp\left(\sum_{i=1}^{m} \lambda_i \mathbf{E}_i \right) \tag{2.38}$$

where m is the dimension of the group. (For a group represented by $n \times n$ matrices, m is the number of degrees of freedom, not the matrix size $n \times n$.) The vectors of coordinates are obtained by the "vee" operator $(\cdot)^\vee : G \to \mathbb{R}^m$:

$$\left(\sum_{i=1}^{m} \lambda_i \mathbf{E}_i \right)^\vee \triangleq \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_m \end{bmatrix} \tag{2.39}$$

The inverse to this operator is the "hat" operator $(\cdot)^\wedge : \mathbb{R}^m \to \mathfrak{g}$ such that $(\boldsymbol{\phi}^\wedge)^\vee = \boldsymbol{\phi}$. Note that these operations are not uniquely defined, but depend on our choice of basis for G. The cross operator for SO(3) is a special case of the hat operator.

What if we want to work in the tangent space of an element other than identity? The answer lies in the *conjugation map*, $c_{\mathcal{h}} : G \to G$ for some $\mathcal{h} \in G$:

$$c_{\mathcal{h}}(\mathcal{g}) = \mathcal{h} \circ \mathcal{g} \circ \mathcal{h}^{-1}. \tag{2.40}$$

Figure 2.2 illustrates the effect of conjugation on rotations. For example, we can apply a perturbation $\exp(t\mathbf{X})$ to some non-identity $\mathcal{h}$ by applying the map $c_{\mathcal{h}}(\exp(t\mathbf{X}))$. The tangent of this map defines the *adjoint*: (Chirikjian, 2011)

$$\mathrm{Ad}(\mathcal{h})\mathbf{X} \triangleq \frac{\mathrm{d}}{\mathrm{d}t} \left(\mathcal{h} \circ e^{t\mathbf{X}} \circ \mathcal{h}^{-1} \right) \bigg|_{t=0} = \mathcal{h} \mathbf{X} \mathcal{h}^{-1}. \tag{2.41}$$

The adjoint $\mathrm{Ad}(\mathcal{h})$, also denoted $\mathrm{Ad}_{\mathcal{h}}$, is a map $\mathrm{Ad}_{\mathcal{h}} : \mathfrak{g} \to \mathfrak{g}$. The Ad operator itself is a homomorphism mapping group elements to $\mathrm{GL}(\mathfrak{g})$, the group of all bijective linear maps on $\mathfrak{g}$; the map $\mathrm{Ad} : G \to \mathrm{GL}(\mathfrak{g})$ is the *adjoint representation* of G.

As its definition (2.41) suggests, $\mathrm{Ad}_{\mathcal{h}}$ can be seen as the Jacobian of the conjugation map with respect to the coordinate vector in $\mathbb{R}^m$. After introducing coordinates, adjoint elements are expressed as $m \times m$ matrices which can multiply coordinate vectors in $\mathbb{R}^m$.

There is also an adjoint representation of the Lie algebra, denoted by the lowercase $\mathrm{ad} : \mathfrak{g} \to \mathrm{gl}(\mathfrak{g})$. Its definition also gives the Lie bracket of $\mathfrak{g}$:

$$\mathrm{ad}(\mathbf{X})\mathbf{Y} \triangleq \frac{\mathrm{d}}{\mathrm{d}t}\left(\mathrm{Ad}(e^{t\mathbf{X}})\mathbf{Y}\right)\Big|_{t=0} \tag{2.42}$$

$$\mathrm{ad}(\mathbf{X})\mathbf{Y} \triangleq [\mathbf{X}, \mathbf{Y}] \tag{2.43}$$

for all $\mathbf{X}, \mathbf{Y} \in \mathfrak{g}$. Just as the Lie group and algebra are connected via $\exp(\mathbf{X})$, it can be shown that the adjoints are connected via

$$\mathrm{Ad}\left(e^{t\mathbf{X}}\right) = e^{t\,\mathrm{ad}(\mathbf{X})}. \tag{2.44}$$

While we do not use the adjoint and Lie bracket directly in this work, their properties make it possible to capture the structure of a curved object, the Lie group, with a flat one, its tangent space at the identity (Stillwell, 2008). Thus, we can produce a perturbation in the tangent space, apply it to the identity using the exponential map, and, as illustrated in Fig. 2.2, apply it equivalently to any non-identity element of G.

2.5.6 Perturbations on a manifold

Hertzberg et al. (2013) defines the operators $\boxplus$ ("boxplus") and $\boxminus$ ("boxminus") to express the mapping between a local neighbourhood of an m-dimensional manifold $\mathcal{M}$ and $\mathbb{R}^m$:

$$\boxplus : \mathcal{M} \times \mathbb{R}^m \to \mathcal{M}, \tag{2.45}$$

$$\boxminus : \mathcal{M} \times \mathcal{M} \to \mathbb{R}^m. \tag{2.46}$$

Hertzberg et al. (2013) further defines a $\boxplus$-*manifold* as a collection $(\mathcal{M}, \boxplus, \boxminus, \mathcal{V})$ where $\mathcal{V} \subset \mathbb{R}^m$ is an open neighbourhood of $\mathbf{0}$ and the following axioms hold for all $\mathcal{x}, \mathcal{y} \in \mathcal{M}$ and

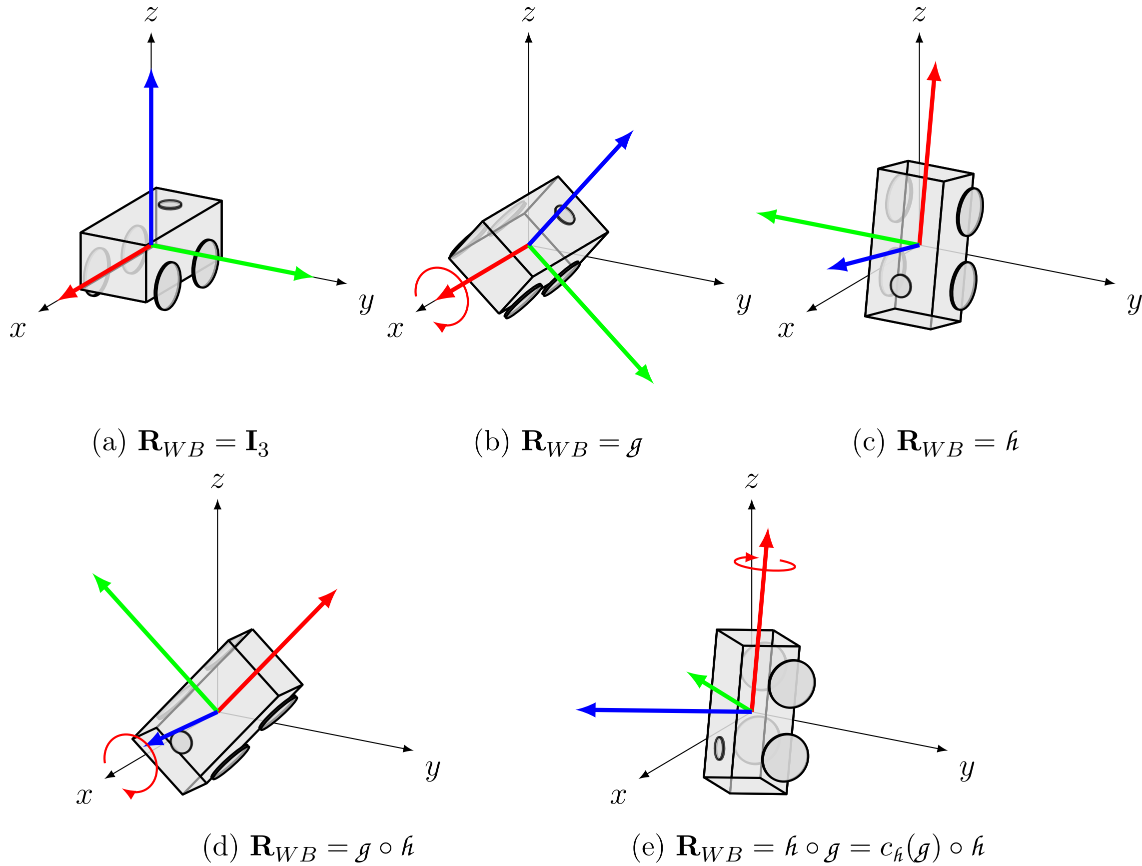


Figure 2.2: Rotations as an example of Lie group operations. To visualize group elements, let SO(3) elements represent $\mathbf{R}_{WB}$, the orientation of a robot body in the world frame, with (a) showing identity. Consider group elements $g, h \in G$ illustrated in (b) and (c). We can compose g and h in either order (d, e) with different results. If we think of g as a perturbation about identity (in this case, a roll about the robot's body x axis) and wish to apply that perturbation to the non-identity element h to obtain result (e), the desired operation is $h \circ g$. To get the same result while perturbing h only from the left, we can equivalently apply the conjugation map $c_h(g) = h \circ g \circ h^{-1}$.

all $\mathbf{a}, \mathbf{b} \in \mathcal{V}$:

$$
\begin{aligned}
\mathcal{X} \boxplus \mathbf{0} &= \mathcal{X}, && \text{(2.47a)}\\
\mathcal{X} \boxplus (\mathcal{Y} \boxminus \mathcal{X}) &= \mathcal{Y}, && \text{(2.47b)}\\
(\mathcal{X} \boxplus \mathbf{a}) \boxminus \mathcal{X}) &= \mathbf{a}, && \text{(2.47c)}\\
(\mathcal{X} \boxplus \mathbf{a}) \boxminus (\mathcal{X} \boxplus \mathbf{b}) &\leq \|\mathbf{a} - \mathbf{b}\|. && \text{(2.47d)}
\end{aligned}
$$

Additionally, $\mathcal{X} \boxplus \mathbf{a}$ must be smooth in $\mathbf{a}$ and $\mathcal{Y} \boxminus \mathcal{X}$ must be smooth in $\mathcal{Y}$. Lie groups meet these requirements (Hertzberg et al., 2013).

$\boxplus$ and $\boxminus$ on Lie groups

For Lie groups, the $\boxplus$ operator adds a perturbation $\mathbf{X} \in \mathfrak{g}$ to a group element $\mathcal{g} \in G$. Following Bloesch et al. (2016) and Barfoot (2017), we adopt the convention of applying the perturbation on the left and define:

$$
\begin{aligned}
\boxplus &: G \times \mathfrak{g} \to G,\\
\mathcal{g} \boxplus \mathbf{X} &\triangleq \exp(\mathbf{X}) \circ \mathcal{g}.
\end{aligned} \tag{2.48}
$$

As an analogue to vector subtraction, the $\boxminus$ ("boxminus") operator obtains the difference between two group elements:

$$
\begin{aligned}
\boxminus &: G \times G \to \mathfrak{g},\\
\mathcal{g} \boxminus \mathcal{h} &\triangleq \log\big(\mathcal{g} \circ \mathcal{h}^{-1}\big).
\end{aligned} \tag{2.49}
$$

$\boxplus$ and $\boxminus$ operators can also be defined for $\mathbb{R}^n$, where they are equivalent to ordinary vector addition and subtraction:

$$
\mathbf{a} \boxplus \mathbf{b} \triangleq \mathbf{a} + \mathbf{b}, \tag{2.50}
$$

$$
\mathbf{a} \boxminus \mathbf{b} \triangleq \mathbf{a} - \mathbf{b}. \tag{2.51}
$$

Compound $\boxplus$-manifolds

A *compound $\boxplus$-manifold*, defined by Hertzberg et al. (2013), is the Cartesian product of multiple $\boxplus$-manifolds. For a $\boxplus$-manifold $\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2$, the $\boxplus$ and $\boxminus$ operators apply the

the original manifolds' operators component-wise:

$$(\mathcal{X}_1, \mathcal{X}_2) \boxplus \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \end{bmatrix} \triangleq (\mathcal{X}_1 \boxplus_1 \mathbf{a}_1, \mathcal{X}_2 \boxplus_2 \mathbf{a}_2) \tag{2.52}$$

$$(\mathcal{Y}_1, \mathcal{Y}_2) \boxminus (\mathcal{X}_1, \mathcal{X}_2) \triangleq \begin{bmatrix} \mathcal{Y}_1 \boxminus_1 \mathcal{X}_1 \\ \mathcal{Y}_2 \boxminus_2 \mathcal{X}_2 \end{bmatrix} \tag{2.53}$$

Compound manifolds describe state models composed of multiple geometric primitives: for example, a robot's position, orientation, and velocity.

2.5.7 Derivatives on Lie groups

The directional derivative of a real-valued function $f : \mathbb{R}^p \to \mathbb{R}$, giving the rate of change of f in the direction $\mathbf{v}$, is defined

$$\nabla_{\mathbf{v}} f(\mathbf{x}) = \lim_{\epsilon \to 0} \frac{f(\mathbf{x} + \epsilon \mathbf{v}) - f(\mathbf{x})}{\epsilon} \tag{2.54}$$

for $\epsilon \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^p$ and unit vector $\mathbf{v} \in \mathbb{R}$. It can also be expressed as a weighted sum of the partial derivatives with respect to each component of $\mathbf{x}$:

$$\nabla_{\mathbf{v}} f(\mathbf{x}) = \sum_{i=1}^{p} v_i \frac{\partial f}{\partial x_i}. \tag{2.55}$$

The directional derivative can be extended to Lie groups by replacing the addition of $\epsilon\mathbf{v}$ with a perturbation by $\exp(\epsilon\mathbf{X})$. Consider a function $f : G \to \mathbb{R}$. Because the perturbation can be applied on either side, we have two possible directional derivatives:

$$\nabla^l_{\mathbf{v}} f(\mathcal{g}) = \lim_{\epsilon \to 0} \frac{f(\exp(\epsilon \mathbf{X}) \circ \mathcal{g}) - f(\mathcal{g})}{\epsilon} \qquad \text{(left Lie derivative)} \tag{2.56}$$

$$\nabla^r_{\mathbf{v}} f(\mathcal{g}) = \lim_{\epsilon \to 0} \frac{f(\mathcal{g} \circ \exp(\epsilon \mathbf{X})) - f(\mathcal{g})}{\epsilon} \qquad \text{(right Lie derivative).} \tag{2.57}$$

We follow the convention of using the left Lie derivative. The analogy to derivatives on vector spaces becomes clear when we apply definition (2.48) of the $\boxplus$ operator:

$$\nabla^l_{\mathbf{v}} f(\mathcal{g}) = \lim_{\epsilon \to 0} \frac{f(\mathcal{g} \boxplus \epsilon \mathbf{X}) - f(\mathcal{g})}{\epsilon}. \tag{2.58}$$

As with the vector derivative (2.55), we can split the Lie derivative (2.58) into a sum of partial derivatives by substituting $\mathbf{X} = \sum_{i=1}^{m} \lambda_i \mathbf{e}_i^{\wedge}$, where $\mathbf{e}_i \triangleq \mathbf{E}_i^{\vee}$ are the basis elements of $\mathbb{R}^m$ corresponding to the basis elements of the Lie algebra. The partial derivative with respect to the ith component is

$$\frac{\partial f(\mathcal{g})}{\partial \lambda_i} = \lim_{\epsilon \to 0} \frac{f(\mathcal{g} \boxplus \epsilon \mathbf{e}_i^{\wedge}) - f(\mathcal{g})}{\epsilon}. \tag{2.59}$$

We can extend this idea to any analytic function $h : G_1 \to G_2$ between an m-dimensional Lie group G_1 and l-dimensional Lie group G_2:

$$\frac{\partial h(\mathcal{g})}{\partial \lambda_i} = \lim_{\epsilon \to 0} \left(\frac{h(\mathcal{g} \boxplus \epsilon \mathbf{e}_i^{\wedge}) \boxminus h(\mathcal{g})}{\epsilon} \right)^{\vee}. \tag{2.60}$$

This partial derivative is a vector in $\mathbb{R}^l$. Concatenating the columns for each component gives the $l \times m$ Jacobian matrix

$$\mathbf{J}_h = \begin{bmatrix} \frac{\partial h(\mathcal{g})}{\partial \lambda_1} & \frac{\partial h(\mathcal{g})}{\partial \lambda_2} & \dots & \frac{\partial h(\mathcal{g})}{\partial \lambda_n} \end{bmatrix}. \tag{2.61}$$

This matrix is also called the *local Jacobian* of h. The *global* Jacobian is the derivative with respect to coefficients of the vector space in which G is embedded (Sommer et al., 2013). For example, if h is a function mapping rotation matrices to rotation quaternions, its local Jacobian will be 3×3, and its global Jacobian will be 4×9: quaternions have 4 coefficients and rotation matrices have 9, although both have 3 degrees of freedom.

As discussed in Section 2.5.6, the $\boxplus$ and $\boxminus$ operators can be defined for manifolds that are not Lie groups. Thus, with appropriate notation changes, equation (2.60) holds for functions of the form $\mathbb{R}^m \to \mathbb{R}^l$, $\mathbb{R}^m \to G$, and $G \to \mathbb{R}^l$.

2.5.8 Operations on SO(3)

Having defined operations of generic Lie groups and Lie algebras, we return to the example of SO(3). The matrices $\mathbf{E}_i$ form a basis for $\mathfrak{so}(3)$, and are called *generators* of SO(3). The vector $\boldsymbol{\phi} = \boldsymbol{\varphi}^{\vee}$ is called a *rotation vector*, and represents a rotation of angle $\theta = ||\boldsymbol{\phi}||$ about an axis $\mathbf{a} = \boldsymbol{\phi}/\theta$. Its exponential map, as derived in Appendix A.1.1, is

$$\exp(\boldsymbol{\phi}^{\times}) = \mathbf{I}_3 + \frac{\sin\theta}{\theta}\boldsymbol{\phi}^{\times} + \frac{1-\cos\theta}{\theta^2}(\boldsymbol{\phi}^{\times})^2, \tag{2.62}$$

where $\boldsymbol{\phi}$ is the rotation vector introduced in Section 2.5.1, with $\theta = ||\boldsymbol{\phi}||$. In fact, it is the vector of components of $\mathfrak{so}(3)$ with respect to our chosen basis. The previously defined $\times$ operator can now be recognized as the special case of the $\wedge$ operator for $\mathfrak{so}(3)$.

The Lie bracket of $\mathfrak{so}(3)$ is $[\boldsymbol{\varphi}_1, \boldsymbol{\varphi}_2] = \boldsymbol{\varphi}_1\boldsymbol{\varphi}_2 - \boldsymbol{\varphi}_2\boldsymbol{\varphi}_1$, which is equivalent to the cross product of rotation vectors: $[\boldsymbol{\varphi}_1, \boldsymbol{\varphi}_2] = (\boldsymbol{\phi}_1 \times \boldsymbol{\phi}_2)^\wedge = \boldsymbol{\phi}_1^\times \boldsymbol{\phi}_2$. The adjoint of $\mathfrak{so}(3)$ is then trivially $\mathrm{ad}_{\boldsymbol{\varphi}_1} = \boldsymbol{\varphi}_1$. Similarly, the adjoint of SO(3) can be shown simply to be $\mathrm{Ad}_{\mathbf{C}} = \mathbf{C}$.

In practice, as suggested by Grassia (1998), it is advantageous not to calculate the exponential map (2.62) but instead to map onto S^3:

$$\mathbf{q}(\boldsymbol{\phi}) = \begin{bmatrix} \frac{\sin\left(\frac{1}{2}\theta\right)}{\theta}\boldsymbol{\phi} & \cos\left(\frac{1}{2}\theta\right) \end{bmatrix}^\mathsf{T}. \tag{2.63}$$

The result is a unit quaternion representation of $\exp\left(\boldsymbol{\phi}^\times\right)$ and can be converted to a rotation matrix if desired. A full discussion, performance comparison, and numerically stable formulations of maps (2.62) and (2.63) are provided in Appendix A.1.1. Quaternions are further discussed in Section 2.5.9.

2.5.9 Quaternions

A quaternion may be written as a column of four coefficients:

$$\mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} \vec{\mathbf{q}} \\ s \end{bmatrix}. \tag{2.64}$$

Rotations on SO(3) can be represented by quaternions of unit length, such that $\mathbf{q}^\mathsf{T}\mathbf{q} = 1$.

It is possible to define conversion of quaternions to rotation matrices, multiplication of quaternions, and rotation of vectors such that

$$\mathbf{C}(\mathbf{q}_1 \circ \mathbf{q}_2) = \mathbf{C}(\mathbf{q}_1) \circ \mathbf{C}(\mathbf{q}_2), \tag{2.65}$$

$$\mathbf{q}(\mathbf{v}) = \mathbf{C}(\mathbf{q})\mathbf{v}, \quad \mathbf{v} \in \mathbb{R}^3. \tag{2.66}$$

The multiplication (2.65) is one possible quaternion multiplication. Conflicting conventions for representation, conversion, and multiplication of quaternions are used in different fields and software packages; this issue is discussed by Shuster (2008), Solà (2016), and Sommer

et al. (2018). Our implementation follows the convention used by Eigen. Full definitions of quaternion operations under that convention can be found in Solà (2016).

In software, quaternions are often preferred over rotation matrices because they are easy to interpolate, more space-efficient, and take fewer floating point operations to compose, although they are slower at rotating vectors (Schneider and Eberly, 2003). However, quaternions are useful for more than representing rotations.

Quaternions, S^3 and $\mathbb{RP}^3$

The group of unit quaternions, SU(2), is isomorphic to the 3-sphere S^3 (Gallier, 2011). Like unit quaternions, S^3 is represented in $\mathbb{R}^4$ by the set of points (x, y, z, w) such that

$$x^2 + y^2 + z^2 + w^2 = 1. \tag{2.67}$$

As described by Gallier (2011), the map SU(2) → SO(3) is a surjective and continuous homomorphism and "SO(3) is homeomorphic to the quotient of the sphere S^3 modulo the antipodal map." In other words, for every element of SO(3), there are two elements of SU(2), with the antipode $\neg\mathbf{q} = -\mathbf{q}$ of any unit quaternion producing an identical rotation:

$$\mathbf{q} \cong -\mathbf{q}, \tag{2.68}$$

$$\mathbf{C}(\mathbf{q}) = \mathbf{C}(-\mathbf{q}). \tag{2.69}$$

This equivalence relation is exactly that of spherically normalized homogeneous coordinates, presented in (2.20). In fact, $\mathbb{RP}^3$ and SO(3) are homeomorphic topological spaces and diffeomorphic manifolds (see also Chirikjian, 2009, Section 7.2). For our purposes, following Stillwell (2008), we can simply say they are the same group.

2.5.10 Perturbations of homogeneous points

Because $\mathbb{RP}^3$ and SO(3) are equivalent, we can treat points in spherically normalized homogeneous coordinates exactly like rotations, and apply the $\boxplus$ and $\boxminus$ operators of definitions (2.48) and (2.49):

$$\boldsymbol{x}^s \boxplus \mathbf{v} = \exp(\mathbf{v}) \circ \boldsymbol{x}^s, \tag{2.70}$$

$$\boldsymbol{y}^s \boxminus \boldsymbol{x}^s = \log\big(\boldsymbol{y}^s \circ (\boldsymbol{x}^s)^{-1}\big). \tag{2.71}$$

Here, $\boldsymbol{x}^s, \boldsymbol{y}^s$ are homogeneous points expressed as unit quaternions. The exponential map is the map (2.63) onto quaternions, with the subtle difference that its argument is not a rotation vector but a perturbation vector $\mathbf{v}$. A small complication is that any vector of length 2π is mapped to the same quaternion $\begin{bmatrix}\mathbf{0} & -1\end{bmatrix}^\mathsf{T}$ and requires renormalization (see Hartley and Zisserman, 2004, Appendix A6.9.2); however, that is not a concern for perturbations with $\|\mathbf{v}\| < \pi$. The logarithmic map is given in Appendix A.1.2. For small perturbations, it is possible to use a linear approximation of the exponential map, given in Appendix A.1.3.

The homogeneous parametrization is preferred for estimation of 3D points because of its well-conditioned behaviour for distant points, including points at infinity (Triggs et al., 1999).

2.5.11 The Euclidean motions SE(3)

The special Euclidean group, of proper rigid motions, is represented by the set of transformation matrices whose linear part is a proper rotation:

$$\mathrm{SE}(3) = \left\{ \mathbf{T} \in \mathbb{R}^{4\times 4} = \begin{bmatrix} \mathbf{C} & \mathbf{t} \\ \mathbf{0}_3^\mathsf{T} & 1 \end{bmatrix} : \mathbf{C} \in \mathrm{SO}(3), \mathbf{t} \in \mathbb{R}^3 \right\}. \tag{2.72}$$

As described in Section 2.2.4, $\mathbf{T}$ acts by matrix multiplication on points represented as vectors with an added affine coordinate:

$$\mathbf{T}\begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{C} & \mathbf{r} \\ \mathbf{0}_3^\mathsf{T} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{C}\mathbf{x} + \mathbf{t} \\ 1 \end{bmatrix} \tag{2.73}$$

The Lie algebra $\mathfrak{se}(3)$ is given by 4×4 matrices

$$\boldsymbol{\xi}^\wedge = \begin{bmatrix} \boldsymbol{\phi} \\ \mathbf{t} \end{bmatrix}^\wedge = \begin{bmatrix} \boldsymbol{\phi}^\times & \boldsymbol{\rho} \\ \mathbf{0}_3^\mathsf{T} & 0 \end{bmatrix}, \tag{2.74}$$

where $\boldsymbol{\phi}, \boldsymbol{\rho} \in \mathbb{R}^3$ are respectively the rotation and translation vectors. Note that we use a different ordering for the component vector $\boldsymbol{\xi}$ than Barfoot (2017).

The exponential map, $\exp : \mathfrak{se}(3) \to \mathrm{SE}(3)$, is

$$\exp(\boldsymbol{\xi}^\wedge) = \begin{bmatrix} \exp(\boldsymbol{\phi}^\wedge) & \mathbf{J}_\phi \boldsymbol{\rho} \\ \mathbf{0}_3^\mathsf{T} & 1 \end{bmatrix}, \tag{2.75}$$

where $\mathbf{J}_{\boldsymbol{\phi}}$ is the Jacobian of the exponential map of SO(3) at $\boldsymbol{\phi}$. The logarithmic map is

$$\log(\mathbf{T}) = \begin{bmatrix} \log(\mathbf{C}) & \mathbf{J}_{\boldsymbol{\phi}}^{-1}\mathbf{t} \\ \mathbf{0}_3^\intercal & 1 \end{bmatrix}. \tag{2.76}$$

The adjoints of SE(3) are (Chirikjian, 2011)

$$\mathcal{T} = \mathrm{Ad}(\mathbf{T}) = \begin{bmatrix} \mathbf{C} & \mathbf{0}_{3\times3} \\ \mathbf{t}^\times\mathbf{C} & \mathbf{C} \end{bmatrix}, \qquad \mathrm{ad}(\boldsymbol{\xi}^\wedge) = \begin{bmatrix} \boldsymbol{\phi}^\times & \mathbf{0}_{3\times3} \\ \boldsymbol{\rho}^\times\mathbf{C} & \boldsymbol{\phi}^\times \end{bmatrix}, \tag{2.77}$$

Since the adjoints depend on the order of the chosen basis, these definitions differ from Barfoot (2017).

Similarly to the derivations for SO(3) in Appendix A, we can obtain derivatives for these operations and use them to optimize over robot position and orientation. Notably although SE(3) elements include a rotation matrix and a translation, these subobjects cannot be independently perturbed. SE(3) is *not* the direct product of the groups SO(3) and $\mathbb{R}^3$—if that were true, its exponential map (2.75) would be a simple combination of $\exp(\boldsymbol{\phi}^\wedge)$ and $\boldsymbol{\rho} \equiv \mathbf{t}$. The group that *is* the direct product of these manifolds is examined by Chirikjian et al. (2018).

2.6 Automatic differentiation

Automatic differentiation (AD) presents the possibility of calculating accurate derivatives of arbitrary functions without an analytical expression. Accessible introductions to AD can be found in Hoffmann (2016), Baydin et al. (2018).

Briefly, AD finds a function's derivatives by applying the chain rule to the sequence of elementary operations used to evaluate the function. To adopt an example from Baydin et al. (2018), consider the real-valued function:

$$y = f(x_1, x_2) = 3x_1x_2 + \sin(x_2), \tag{2.78}$$

The evaluation of this function can be described as a sequence of elementary operations on intermediate values. This *evaluation trace* is shown in Table 2.2. The function can also be drawn as a computational graph, shown in Fig. 2.3.

AD has two main modes: forward and reverse.

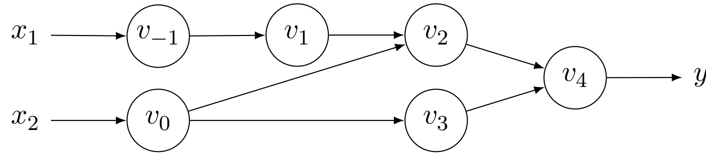


Figure 2.3: Computational graph of $y = 3x_1x_2 + \sin(x_2)$. The intermediate variables v_i are defined in Table 2.2.

Table 2.2: Example of forward mode AD of $y = f(x_1, x_2) = 3x_1x_2 + \sin(x_2)$ evaluated at $x_1 = 5, x_2 = 0.5$. To compute $\frac{\partial y}{\partial x_1}$, the primal trace of the original function evaluation (left) is combined with the tangent trace (right). Adapted from Baydin et al. (2018).

Forward primal trace			Forward tangent trace		
v_{-1}	$= x_1$	$= 5$	$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
v_0	$= x_2$	$= 0.5$	$\dot{v}_0$	$= \dot{x}_2$	$= 0$
v_1	$= 3v_{-1}$	$= (3)(5)$	$\dot{v}_1$	$= \dot{v}_{-1} \cdot 3$	$= 3$
v_2	$= v_1v_0$	$= (15)(0.5)$	$\dot{v}_2$	$= \dot{v}_1v_0 + \dot{v}_0v_1$	$= (3)(0.5) + (0)(15)$
v_3	$= \sin v_0$	$= \sin 0.5$	$\dot{v}_3$	$= \dot{v}_0 \cos v_0$	$= 0 \cos 0.5$
v_4	$= v_2 + v_3$	$= 7.5 + 0.479$	$\dot{v}_4$	$= \dot{v}_2 + \dot{v}_3$	$= 1.5 + 0$
y	$= v_4$	$= 7.979$	$\dot{y}$	$= \dot{v}_4$	$= 1.5$

2.6.1 Forward mode

In forward accumulation mode, we assign a tangent to each intermediate variable in the original function's evaluation trace (the *primal* trace). Each tangent is the partial derivative of an intermediate variable with respect to a single input. These tangents are propagated forward using the chain rule to produce the forward tangent trace, shown in Table 2.2.

A single pass of forward mode AD gives the derivative with respect to a single input. For a function $f : \mathbb{R}^n \to \mathbb{R}^m$, a forward pass thus produces one column of the $m \times n$ Jacobian matrix.

Table 2.3: Example of reverse mode AD of $y = f(x_1, x_2) = 3x_1x_2 + \sin(x_2)$ evaluated at $x_1 = 5, x_2 = 0.5$. After the primal trace (left), the adjoint trace (right) computes both $\frac{\partial f}{\partial x_1}$ and $\frac{\partial f}{\partial x_2}$ in one reverse pass. Note that the adjoint $\bar{v}_0$ is assigned more than once. Adapted from Baydin et al. (2018).

Forward primal trace			Reverse adjoint trace			
v_{-1}	$= x_1$	$= 5$	$\bar{x}_1$	$= \bar{v}_{-1}$	$= \frac{\partial f}{\partial x_1}$	$= 1.5$
v_0	$= x_2$	$= 0.5$	$\bar{x}_0$	$= \bar{v}_0$	$= \frac{\partial f}{\partial x_2}$	$= 15.878$
			$\bar{v}_{-1}$	$= \bar{v}_1 \frac{\partial v_{-1}}{\partial v_1}$	$= \bar{v}_1 \cdot 3$	$= 1.5$
v_1	$= 3v_{-1}$	$= (3)(5)$	$\bar{v}_1$	$= \bar{v}_2 \frac{\partial v_2}{\partial v_1}$	$= \bar{v}_2 v_0$	$= 0.5$
v_2	$= v_1 v_0$	$= (15)(0.5)$	$\bar{v}_0$	$= \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 v_1$	$= 15.878$
v_3	$= \sin v_0$	$= \sin 0.5$	$\bar{v}_2$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4$	$= 1$
v_4	$= v_2 + v_3$	$= 7.5 + 0.479$	$\bar{v}_0$	$= \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \cos v_0$	$= 0.878$
			$\bar{v}_3$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_3}$	$= \bar{v}_4$	$= 1$
y	$= v_4$	$= 7.979$	$\bar{v}_4$	$= \bar{y}$	$= 1$	

2.6.2 Reverse mode

The reverse accumulation mode of AD propagates *adjoints*—derivatives of the output with respect to intermediate variables—backward, from the output toward the inputs.[5] This mode, demonstrated in Table 2.3, simultaneously computes the derivatives of all inputs with respect to one output.

Because the adjoints depend on the results of the primal evaluation, a "tape" of intermediate primal values must be recorded in memory during the forward pass. In our implementation, this tape is provided by our use of expression templates and our Evaluator cache structure, described in Section 3.2.

Note that AD is not symbolic differentiation. Symbolic differentiation would give

[5]This usage of *adjoint* is unrelated to the adjoint operator on Lie groups.

derivatives of example (2.78) as equations:

$$\frac{\partial f}{\partial x_1} = 3x_2, \tag{2.79a}$$

$$\frac{\partial f}{\partial x_2} = 3x_1 + \cos x_2. \tag{2.79b}$$

Although the results of forward and reverse AD, shown in Tables 2.2 and 2.3, match these equations, they are obtained purely by propagation of numerical values. At the same time, AD is not numerical differentiation, which is an approximation based on finite differences. AD suffers from neither the accuracy and numerical stability problems of numerical differentiation, nor the complexity issues of symbolic differentiation (Hoffmann, 2016).

2.6.3 Block automatic differentiation

The above example demonstrates AD with scalar-valued atomic operations. To find the local Jacobians described in Section 2.5.7, we use *block automatic differentiation* (BAD), defined by Sommer et al. (2013). In BAD, the inputs, outputs, and primals x, y, v are manifold elements, and the tangents and adjoints $\dot{v}, \bar{v}$ are matrices. Instead of scalar arithmetic, the elementary operations considered under BAD are manifold operations such as composition and inverse, as well as geometric operations such as transformation of a translation vector. Section 3.2 demonstrates BAD applied to geometric expressions in forward and reverse modes.

Note that our use of the term *Jacobian* diverges from Baydin et al. (2018): in robotics, a Jacobian is often broken into multiple matrices, also called Jacobians. Each Jacobian matrix is the partial derivative with respect to a single multidimensional input. Thus, we state that one pass of forward mode AD produces the Jacobian with respect to one (multidimensional) variable, while one pass of reverse mode AD produces multiple Jacobians.

Chapter 3

Manifold geometry in C++

This chapter discusses the design of a C++ library for manifold geometry. Our implementation, `wave_geometry`, is available at https://github.com/wavelab/wave_geometry.

Our library uses expression templates (ETs), introduced in Section 3.1, to represent functions of geometric entities. Using ETs, we can manipulate and differentiate geometric expressions as C++ objects:

Listing 3.1: Example of working with expressions

```
// Construct geometric primitives
wave::RotationMd R = wave::RotationMd::Random();
wave::Translationd v = wave::Translationd::Random();

// Construct expressions
auto expr = R * v;
auto expr2 = 2 * expr;

// Evaluate the result only
wave::Translationd v2 = expr2;

// Evaluate the Jacobian with respect to R
Eigen::Matrix3d JR = expr2.jacobian(R);

// Evaluate the result and Jacobians
auto [v2, JR, Jv] = expr.evalWithJacobians(R, v);
```

This chapter's purpose is to explain how the library makes the user code in Listing 3.1 possible. The *user* in this context is the programmer making calls to the library from their own code. As shown in Listing 3.1, the public interface of the library is defined in the `wave`

namespace. We attempt to make this interface as clean and intuitive as possible, and prefer to return multiple values in a tuple instead of via output parameters. By contrast, *internal* parts of the library described in this chapter are not meant to be called directly by the user and often use unintuitive template metaprogramming techniques. Internals are usually defined in the `wave::internal` namespace. In this work, we typically omit namespaces for brevity.

Section 3.1.1 discusses how our library implements ETs. Section 3.1.2 discusses organization of types in the library. Section 3.1.3 describes expression evaluation using a recursive evaluator structure. Section 3.2 describes automatic differentiation in forward and reverse modes, and discusses how we use compile-time knowledge about the types in an expression to optimize AD evaluation. Section 3.1.3 demonstrates how the library can be extended with new types and operations.

3.1 Expression templates for geometry

Consider the expression

$$\mathbf{r}_2 := \mathbf{T}^{-1}\mathbf{r}_1. \tag{3.1}$$

It may be represented in code as `r2 = T.inverse()* r1`.

In a simple C++ implementation, the call `T.inverse()` would compute the inverse transformation and return it as a temporary object, the multiplication operation would return another object, and that object would be used to assign (or initialize) `r2`. However, this is not the most efficient way to evaluate the equation. There is no need to obtain the matrix $\mathbf{T}^{-1}$; instead, it is cheaper to change the coefficients of $\mathbf{T}$ as needed (using the rotation part in transposed order and subtracting the translation part) to perform the transformation.

Normally, such optimizations would require changes to the calling code—for example, KDL instructs users to call a separate function, `T.Inverse(p1)`, for (3.1). A programming technique called *expression templates* (ETs) allows the compiler to pick the more efficient method without a change in user code. With ET, mathematical expressions are encoded as template arguments. For example, `T.inverse()` might return an object of type `Inverse<` ↪ `RigidTransform>`, a lightweight *expression object* holding a reference to the variable `T`.

Expressions combine to form trees (or, more generally, directed acyclic graphs). To use a concrete example from `wave_geometry`, consider the C++ expression `A_r_AC = inverse`

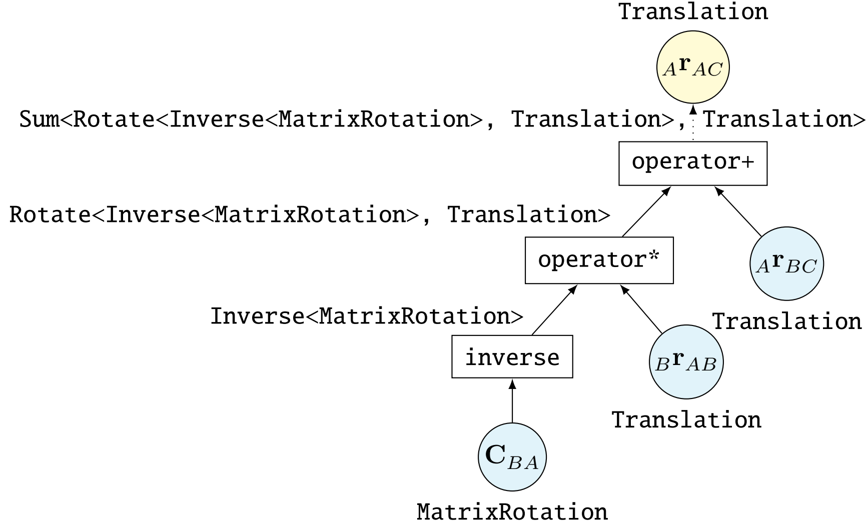


Figure 3.1: Expression tree showing compile-time propagation of types, from bottom to top, for expression (3.2). Leaf nodes are drawn as blue circles, the root node as a yellow circle, and functions as rectangles. Above each function is its return type, which encodes the subtree below. The scalar type (e.g. `double`) is omitted from type names.

↪ `(C_BA)* B_r_AB + A_r_B`, representing

$$_A\mathbf{r}_{AC} := \mathbf{R}_{BA}{}^{-1}\,{}_B\mathbf{r}_{AB} + {}_A\mathbf{r}_{BC}. \tag{3.2}$$

This mathematical expression can be thought of as a tree in which the three variables are leaf nodes, the operations (inverse, rotation, addition) are unary or binary nodes, and the final assigned-to variable is the root node. ETs encode the tree, as illustrated in Fig. 3.1. In this case, the return type of `operator+` holds the entire structure of (3.2). When it is assigned to a `Translationd` object, the expression tree is evaluated.

Expression templates were originally invented to optimize numeric array operations (Vandevoorde and Josuttis, 2002). Early ETs improved performance by eliminating unnecessary temporaries and loops, while relying on the compiler's decision-making to produce efficient low-level code. On their own, such optimizations have relatively little effect on the small, fixed-size vectors and matrices involved in pose representations, for which the compiler is often able to optimize away temporaries (though the optimizations are still desirable, as demonstrated by KDL's `Inverse`). Modern "smart" ET implementations, introduced by Iglberger et al. (2012), use ETs primarily as a parsing mechanism. Once they understand the structure of the mathematical expression, they apply their own optimizations, such as

choosing the order of evaluation of subexpressions and selecting low-level compute kernels. Eigen is one such library.

`wave_geometry` is an ET library built on top of Eigen. It builds its own representation of geometric expressions, and uses Eigen for internal representation and final computation. As discussed in Section 3.2, it also uses the expression-parsing property of ETs as a mechanism for automatic differentiation.

3.1.1 Implementing expression templates

ETs rely on *static polymorphism*: they use a single interface to denote different specific behaviours, which are resolved at compile time (Vandevoorde and Josuttis, 2002). For example, the notation `inverse(C)` can be used whether `C` is a plain matrix or a placeholder representing the result of another calculation. The implementation of statically polymorphic classes is simplified by the curiously recurring template pattern (CRTP), a technique which mixes templates with inheritance (Vandevoorde and Josuttis, 2002, Section 16.3). Its application to ETs is described by Härdtlein et al. (2010).

An expression template using CRTP looks like the example in Listing 3.2. Here, `RotationBase` is a base class template for which the rotation interface is defined, and `Compose` is an ET representing the result of composing two rotations. The `operator*` function template allows us to compose any two rotations by writing `a * b`. The pattern of passing the derived class as a template argument to `RotationBase` lets it provide a common interface while also being able to resolve, at compile time, the exact type of the derived class.

`RotationBase` can be thought of as a *concept*[1], which describes the interface required of rotation objects. Particular rotation types such as `MatrixRotation` and `Quaternion-Rotation`, as well as expressions such as `Inverse<MatrixRotation>`, *model* this concept. For example, all rotation objects can be passed to the `inverse()` and `log()` functions, and can transform vectors. Concepts can refine other concepts: the requirements of Rotation objects are a superset of those on Transform objects.

In principle, there is no reason for models of a concept to be related via inheritance; however, inheritance allows code reuse and simplifies the definition of generic functions which accept only models of a concept (Abrahams and Gurtovoy, 2004). For example, a function composing two `RotationBase` objects is shown in Listing 3.3. CRTP is thus little more than an implementation detail which helps write generic code.

[1] Here, *concept* refers generally to a set of requirements on a type, as defined by Abrahams and Gurtovoy (2004), and not to the concepts language feature planned for C++20.

Listing 3.2: Example of CRTP applied to expression templates

```
template <typename Derived>
class RotationBase {
    // No data members. Methods common to all rotation objects may be defined here.

    // Obtain a reference to the derived object
    const Derived &derived() const & {
        return *static_cast<Derived const *>(this);
    }
}

template <typename Lhs, typename Rhs>
class Compose : public RotationBase<Compose<Lhs, Rhs>> {
    const Lhs &lhs_;
    const Rhs &rhs_;

 public:
    Compose(const Lhs &lhs, const Rhs &rhs) {...}
};

template <typename Lhs, typename Rhs>
Compose<Lhs, Rhs> operator*(const RotationBase<Lhs> &lhs,
                            const RotationBase<Rhs> &rhs) {
    return Compose<Lhs, Rhs>{lhs.derived(), rhs.derived()};
}
```

Listing 3.3: Example of a generic function accepting rotation arguments

```
template <typename Lhs, typename Rhs>
Compose<Lhs, Rhs> operator*(const RotationBase<Lhs> &lhs,
                            const RotationBase<Rhs> &rhs) {
    return Compose<Lhs, Rhs>{lhs.derived(), rhs.derived()};
}
```

Listing 3.4: Example of `wave_geometry` expression template

```
template <typename Lhs, typename Rhs>
struct Compose : internal::base_tmpl_t<Lhs, Rhs, Compose<Lhs, Rhs>>,
                 internal::binary_storage_for<Compose<Lhs, Rhs>> {
 private:
    using Storage = internal::binary_storage_for<Compose<Lhs, Rhs>>;

 public:
    // Inherit constructors from BinaryStorage
    using Storage::Storage;

    static_assert(std::is_same<RightFrameOf<Lhs>, LeftFrameOf<Rhs>>(),
                  "Adjacent frames do not match");
};
```

In reality, our implementation is more complex than the example in Listing 3.2 terms of levels of inheritance and storage of data members. The actual `Compose` template in `wave_geometry` is shown in Listing 3.4.

In Listing 3.4, `internal::base_tmpl_t` is a selector that evaluates to the correct conceptual base class, which might be `RotationBase` or `RigidTransformBase`; the library hierarchy is presented in Section 3.1.2. `binary_storage_for` selects a base class which provides storage and constructors for the `lhs` and `rhs` members; storage details are discussed in Appendix C.1. The `static_assert` is a frame semantics check, discussed in Chapter 4. We note that the expression template itself contains little functionality; its main purpose is to provide a simple tag for the type of geometric expression it represents and to appear in user-facing compiler messages.

3.1.2 Spaces, parametrizations, and storage

We distinguish three orthogonal ideas related to representing a geometric primitive: spaces, parametrizations, and storage. Each object belongs to a vector space or Lie group, which we call a *space*. Each object embodies a representation of this space, such as rotation matrix, quaternion, or angle-axis. Note that representations are not a subset of spaces: three-dimensional points, translations, and rotation vectors can all be represented by vectors in $\mathbb{R}^3$. Finally, each object has *storage*, which describes its representation in computer memory. For example, a rotation matrix may be stored using floats or doubles, in row-major or column-major order, or even in non-contiguous memory. Table 3.1 presents a summary

of these terms.

Table 3.1: Terms describing geometric objects in `wave_geometry`

Term	Examples	Implemented with
Space	SO(3), $\mathfrak{se}(3)$	Base class templates (CRTP)
Representation	Matrix, quaternion	Leaf templates
Storage	Row-major, column-major	Underlying Eigen classes

In `wave_geometry`, spaces are base class templates using CRTP, as described in Section 3.1.1. This pattern allows defining a consistent interface for all objects of the same space. Representations are defined as leaf class templates, such as `MatrixRotation` and `QuaternionRotation`. These classes inherit the interface of `RotationBase`, and do not define their own user-facing methods. However, each leaf class has its own internal evaluation and conversion functions. Figure 3.2 presents the hierarchy of base classes and parametrizations.

The leaves are themselves class templates (not simple classes) to allow for different storage types. As shown is Table 1.1, it is common for classes to be templated on the scalar type, such as `float` or `double`. In `wave_geometry`, the classes are templated on the whole storage type. While a rotation matrix may be stored using a plain `Eigen::Matrix3d`, it may also be stored in any way supported by Eigen's `Map` or `Block` expressions. The template parameter may even be a placeholder Eigen expression such as a `Product`. In a way, `wave_geometry` leaf classes can be seen as type-safe wrappers for the underlying matrix classes.

Nullary expressions

In addition to the leaves shown in Fig. 3.2, there are several nullary ETs which serve as leaves but have no storage. The `Random` ET evaluates to a random value of the given leaf type. `Identity` represents the identity element of a given space, and `Zero` the zero element of a vector space.

The term *nullary* is borrowed from Eigen. Our implementation does not distinguish between nullary and other leaf expressions.

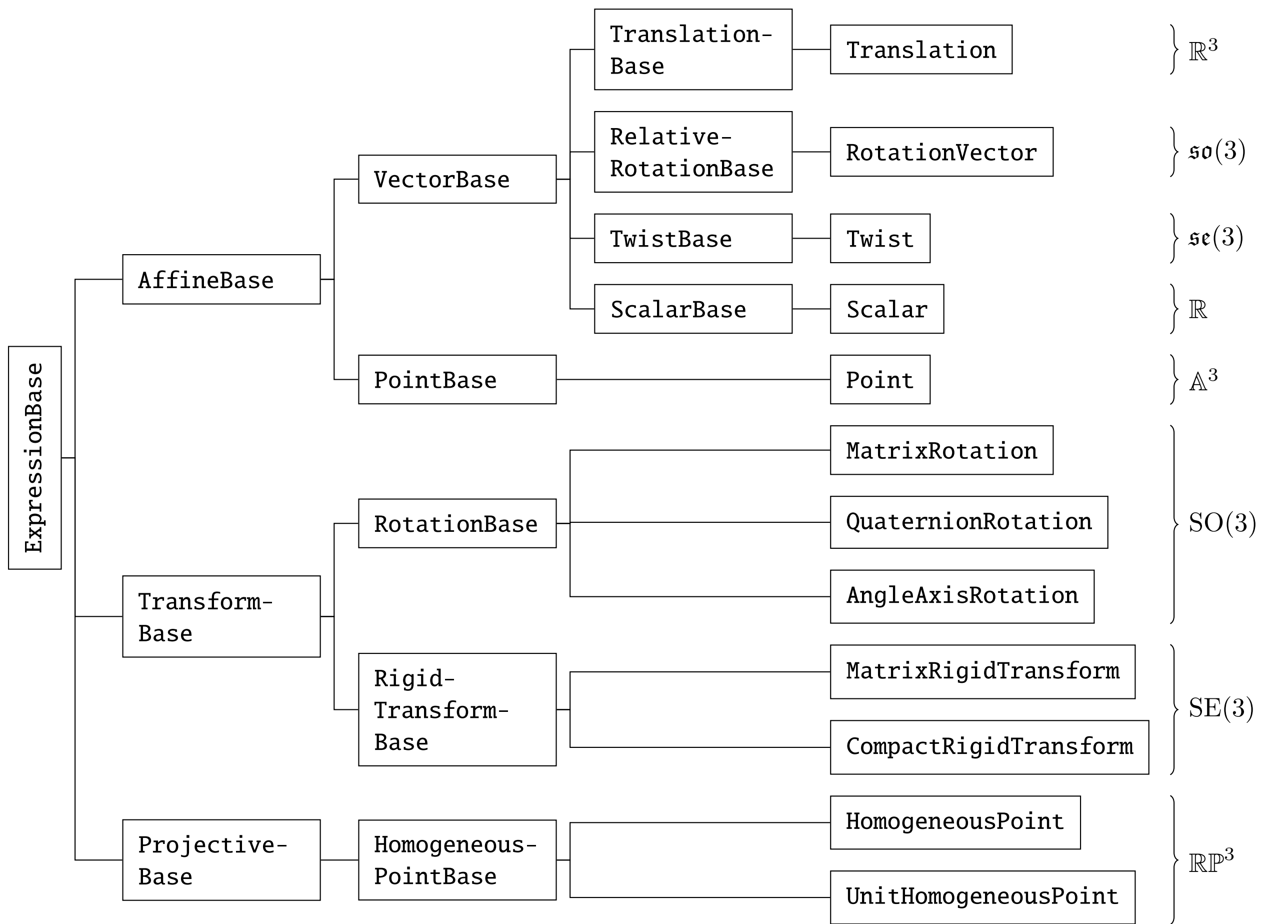


Figure 3.2: Hierarchy of types in `wave_geometry`. Each base class template defines a common interface for a space of geometric objects. Leaves correspond to a particular representation of a geometric object. Storage types are not shown.

3.1.3 Evaluating expressions

Following Eigen,[2] we evaluate expression trees using a recursive *evaluator* structure. The evaluator for each expression caches intermediate results for use by subsequent operations.

Each expression has one or more corresponding *implementation* functions. These functions (actually function templates) define the actual code needed to evaluate the expression for inputs of a particular representation. They are called internally by the evaluator and never directly by the user.

Implementation functions are called using tag dispatch (Meyers, 2014, Item 27). This technique selects an overloaded function definition using an empty tag type, instead of by name or by type of (non-empty) arguments. For example, the implementation of `Compose` for a pair of rotation matrices is:

Listing 3.5: Implementation function for Compose of rotation matrices

```
template <typename Lhs, typename Rhs>
auto evalImpl(expr<Compose>,
              const MatrixRotation<Lhs> &lhs,
              const MatrixRotation<Rhs> &rhs) {
    return plain_eval_t<MatrixRotation<Lhs>>{lhs.value() * rhs.value()};
}
```

Here, `expr<Compose>` is an empty tag used to select this overload of `evalImpl`. In this case, the implementation code is fairly trivial: since Eigen already defines matrix multiplication, we obtain the underlying Eigen objects using `value()`, multiply them, and return a suitable a plain leaf type chosen by the `plain_eval_t` selector.

We define implementation functions as free functions (instead of, for example, as static member functions of `Compose`) to allow extending the library with new types, as described in Section 3.1.3. We use tag dispatch instead of a template specialization to simplify defining implementations with inheritance. The compiler will choose the best viable function overload, including making derived-to-base conversions for one or more arguments if an exact match is not found (ISO, 2017, over.best.ics). For example, the implementation for rigid transformations is:

Listing 3.6: Implementation function for Compose of general transforms

```
template <typename Lhs, typename Rhs>
auto evalImpl(expr<Compose>,
```

[2] Eigen's internal implementation is described at http://eigen.tuxfamily.org/index.php?title=Working_notes_-_Expression_evaluator

```
            const RigidTransformBase<Lhs> &lhs,
            const RigidTransformBase<Rhs> &rhs) -> plain_eval_t<Rhs> {
    plain_eval_t<Rhs> res{};
    res.rotation() = lhs.derived().rotation() * rhs.derived().rotation();
    res.translation() = lhs.derived().rotation() * rhs.derived().translation() +
                        lhs.derived().translation();
    return res;
}
```

Listing 3.7: Example of `wave_geometry` expression template

```
template <typename Lhs, typename Rhs>
auto evalImpl(expr<Compose>,
            const RigidTransformBase<Lhs> &lhs,
            const RigidTransformBase<Rhs> &rhs) -> plain_eval_t<Rhs> {
    plain_eval_t<Rhs> res{};
    res.rotation() = lhs.derived().rotation() * rhs.derived().rotation();
    res.translation() = lhs.derived().rotation() * rhs.derived().translation() +
                        lhs.derived().translation();
    return res;
}
```

The evaluation sequence

Evaluation is invoked when an expression is passed to a constructor or assignment operator of any leaf expression, or when the user explicitly calls `eval()`. In either case, the internal `evaluateTo` function, presented in Listing 3.8, is called.

Listing 3.8: Internal evaluateTo function

```
template <typename Destination, typename Derived>
auto evaluateTo(Derived &&expr) -> Destination {
    // Construct Evaluator tree
    const auto evaluator = prepareEvaluatorTo<Destination>(std::forward<Derived>(expr));

    // Evaluate and apply output functor (e.g. wrap in Framed)
    return prepareOutput(evaluator);
}
```

If called from a constructor or assignment, the template parameter `Destination` is the type of the target class. If called using `eval()`, `Destination` is chosen to be the "natural" result of evaluating the expression tree without extra conversions. As suggested by Listing 3.8, evaluation has several steps:

1. Make any needed transformations to the expression tree.
2. Construct the evaluator tree, which evaluates the expression.
3. Make any needed transformations to the output.

First, the expression tree may be transformed. `wave_geometry` adds conversions between representations if needed, as described below in Section 3.1.3. In future work, optimizing transformations such as rearrangement of operations may be applied here, following the design of Eigen. An evaluator tree, described below, invokes the implementation functions for the transformed expression tree. For some expressions, a final transformation is applied before the result is returned; currently, this step wraps the result with frame semantics as described in Chapter 4.

Note that these structures (the expression tree, the evaluator tree, and so on) are constructs intended to guide the compiler. As observed by Phipps and Pawlowski (2012), much of the task of implementing ET systems lies in coercing the compiler to make favourable optimizations. In practice, the structures do not exist in the final executable when optimizations are enabled.

The evaluator tree

The evaluator itself is a class template whose purpose is to invoke the `evalImpl` for each expression and cache the results. It has different partial specializations for unary, binary, and leaf expressions. The partial specialization for a binary expression is:

Listing 3.9: Evaluator template for binary expressions

```
template <typename Derived>
struct Evaluator<Derived, enable_if_binary_t<Derived>> {
    using EvalType = eval_t<Derived>;
    using LhsEval = Evaluator<typename Derived::LhsDerived>;
    using RhsEval = Evaluator<typename Derived::RhsDerived>;

    inline explicit Evaluator(const Derived &expr)
        : expr{expr},
          lhs_eval{expr.lhs()},
          rhs_eval{expr.rhs()},
          result{
            evalImpl(get_expr_tag_t<Derived>(), this->lhs_eval(), this->rhs_eval())} {}

    const EvalType &operator()() const {
```

```
        return this->result;
    }

public:
    const wave_ref_sel_t<Derived> expr;
    const LhsEval lhs_eval;
    const RhsEval rhs_eval;
    const EvalType result;
};
```

When each evaluator is constructed, it recursively constructs evaluators for its expression's operands. It then calls the implementation function for its expression, passing the cached results of its children and caches the result for its own parent to use. An expression's evaluation is thus simultaneous with the construction of the evaluator tree.

Evaluation can be seen as applying a *fold* to the expression tree, recursively reducing it to single a return value. Fold is an operation which, given a combining function $f : \mathcal{A} \times \mathcal{B} \to \mathcal{B}$ and some initial value, applies f recursively to a sequence of elements of $\mathcal{A}$ to produce a single result in $\mathcal{B}$. We do not attempt to use the formal definition of folds from recursion theory (Hutton, 1999), and instead use the term loosely to provide an intuitive description of expression trees. For our purposes, it is simplest to consider a family of combination functions: for leaf nodes, $f : \mathcal{A} \to \mathcal{B}$, for unary nodes, $f : \mathcal{A} \times \mathcal{B} \to \mathcal{B}$, and for binary nodes, $f : \mathcal{A} \times \mathcal{B} \times \mathcal{B} \to \mathcal{B}$.

The combining function applied by each `Evaluator` is function composition. The evaluator tree combines a tree of expression objects $a \in \mathcal{A}$ into a resulting leaf expression $v \in \mathcal{B}$; in this case, $\mathcal{B} \subset \mathcal{A}$. The combining functions can be written in terms of the `evalImpl` function for a, represented by g_a:

$$\text{Leaf:} \qquad f(a) = g_a(a) \tag{3.3a}$$

$$\text{Unary:} \qquad f(a, v) = g_a(v) \tag{3.3b}$$

$$\text{Binary:} \qquad f(a, v_l, v_r) = g_a(v_l, v_r). \tag{3.3c}$$

Note that while $g_a(a) = a$ for most leaves, there are exceptions such as `Random` expressions. For binary nodes, v_l and v_r are the folded values of the left and right subtrees, respectively.

Conversions

Sometimes, conversions between representations are needed to evaluate an expression. In the simplest case, a user attempts to direct-construct one representation from another:

Listing 3.10: Conversion of output representation

```
MatrixRotationd m{...};
AngleAxisRotationd a{m};
```

In fact, an `AngleAxisRotation` constructor accepting `MatrixRotation` does not exist. The code in Listing 3.10 constructs and evaluates the conversion expression `Convert<AngleAxisRotationd, MatrixRotationd>`. The library also attempts to add conversions between representations whenever no matching implementation function is available. For example, consider the composition of mismatching rotations:

Listing 3.11: Conversion of inputs to an operation

```
AngleAxisRotationd a{...};
MatrixRotationd m{...};
AngleAxisRotationd a2{a * m};
```

Since we implement composition for pairs of rotation matrices but not for angle-axis arguments, a conversion is needed in Listing 3.11. In this case, `wave_geometry` will insert `Convert<MatrixRotationd, AngleAxisRotationd>` before beginning evaluation. Figure 3.3 illustrates how an expression tree is transformed when two conversions are needed.

We use conversion expressions—instead of "normal" constructors or conversion operators, like most C++ libraries—to avoid hidden implicit conversions and let the library track all computations. In future work, the library can use this mechanism to estimate the cost of evaluating an expression tree, and choose the cheapest option where more than one sequence of conversions is possible.

Extending the library

The evaluator implementation described in this section makes it possible extend the library, without modifying the original class definitions, with both new types and new operations. The challenge of allowing such modular extensibility is related to the *expression problem* described by Torgersen (2004). However, the problem we solve is simpler because we deal only with static dispatch: the functions to be called can be resolved at compile-time.

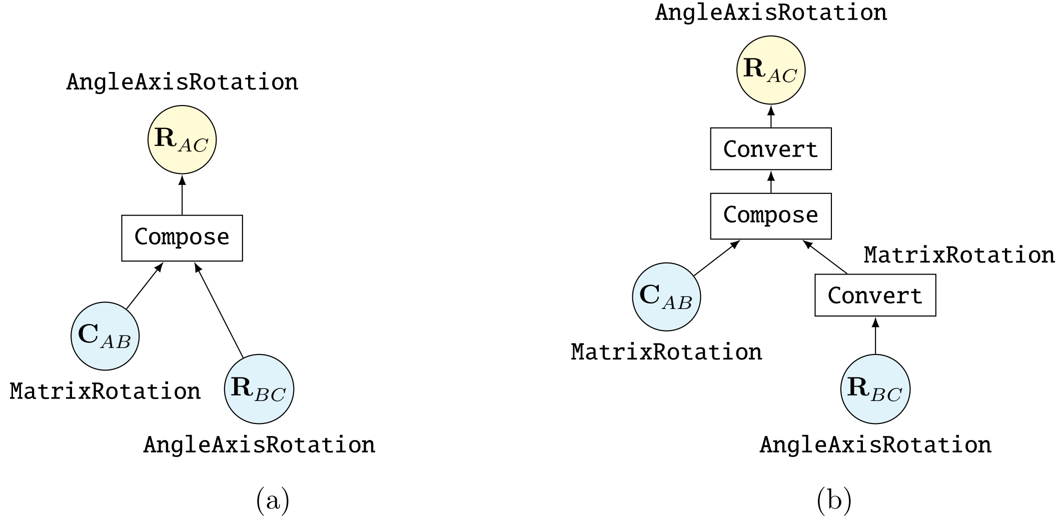


Figure 3.3: Example of library-inserted conversions. The expression tree (a) is not directly evaluable because there is no suitable Compose implementation. Before evaluation, `wave_geometry` inserts conversion expressions (b).

3.2 Automatic differentiation

3.2.1 Forward-mode AD

We previously described evaluating an expression tree as applying a fold operation to the tree (see Section 3.1.3). The same approach lets us calculate the forward-mode derivative $\frac{\partial a}{\partial x}$ of any expression a with respect to a leaf x In this case, the fold operation f'_x reduces a tree of expressions a to a Jacobian matrix $\mathbf{J}_x$. The combining functions apply the chain rule:

$$\text{Leaf:} \qquad f'_x(a) = \frac{\partial g_a}{\partial x} = \begin{cases} \mathbf{I}, & \text{if } a \equiv x \\ \mathbf{0}, & \text{otherwise} \end{cases} \tag{3.4a}$$

$$\text{Unary:} \qquad f'_x(a, v) = \frac{\partial g_a}{\partial g_r}\frac{\partial g_r}{\partial x} = \mathbf{J}(a)v \tag{3.4b}$$

$$\text{Binary:} \qquad f'_x(a, v_l, v_r) = \frac{\partial g_a}{\partial g_r}\frac{\partial g_r}{\partial x} + \frac{\partial g_a}{\partial g_l}\frac{\partial g_l}{\partial x} = \mathbf{J}_l(a)v_l + \mathbf{J}_r(a)v_r. \tag{3.4c}$$

Here, g_r and g_l are the implementation functions of the left and right subtrees of a,

Listing 3.12: Jacobian implementation functions for composition of rotations

```
template <typename Val, typename Lhs, typename Rhs>
decltype(auto) rightJacobianImpl(expr<Compose>,
                                 const Val &,
                                 const MatrixRotation<Lhs> &lhs,
                                 const RotationBase<Rhs> &) {
    return lhs.value();
}

template <typename Val, typename Lhs, typename Rhs>
auto leftJacobianImpl(expr<Compose>,
                      const Val &,
                      const TransformBase<Lhs> &,
                      const TransformBase<Rhs> &) {
    return identity_t<Val>{};
}
```

giving $\partial g_r/\partial x = v$. For any leaf, the identity check $a \equiv x$ determines whether a is the variable with respect to which we are differentiating, which we call the *target* expression. The Jacobian functions, $\mathbf{J}$ for unary expressions and $\mathbf{J}_l$ and $\mathbf{J}_r$ for binary expressions, must be provided separately by the library.

In `wave_geometry`, these Jacobians are provided by the `jacobianImpl`, `leftJacobian-Impl` and `rightJacobianImpl` functions, which are similar to `evalImpl`. These functions are invoked by the forward-mode Jacobian evaluator, illustrated in Fig. 3.4a. The Jacobian evaluator closely resembles the standard `Evaluator` (see Section 3.1.3). However, instead of traversing the original expression tree, the Jacobian evaluator traverses the `Evaluator` tree. This design allows reuse of the `Evaluator`'s cached values: each `jacobianImpl` function is passed the cached values of its expression and of its operands.

For example, the right Jacobian implementation for a composition of rotation matrices, given in Listing 3.12, simply returns a reference to the cached left operand to implement $\partial(\mathbf{C}_l\mathbf{C}_r)/\partial\mathbf{C}_r = \mathbf{C}_l$. The left Jacobian implementation, returning an identity matrix, need only be defined once for all transform expressions.

3.2.2 Testing for identity

We use type information encoded in the expression tree to eliminate unnecessary operations. The identity check used in the leaf derivative (3.4a) can be implemented as a boolean

function:

$$(a \equiv x) = \begin{cases} \operatorname{addr}(a) = \operatorname{addr}(x), & \text{if } \operatorname{type}(a) = \operatorname{type}(x) \\ 0, & \text{otherwise} \end{cases} \tag{3.5}$$

where $\operatorname{addr}(a)$ is the address of the expression object, as given by `&a`, and $\operatorname{type}(a)$ is its type excluding any const or volatile qualifiers. If the two leaves have different types, the Jacobian is known to be zero at compile time, and we avoid traversing that branch of the tree.

This optimization is possible because of C++'s strict aliasing rule (ISO, 2017, basic.lval). An object cannot legally be accessed via a pointer to a different type (with a few exceptions, such as pointers to a base class and `char` types). Therefore references to objects of different types can be assumed not to alias each other, and there is no need to check the address at runtime.

Note that it is possible to produce a standard-compliant form of "aliasing" using proxy objects. For example, two Eigen Maps may write to the same memory region although the Maps themselves have different addresses. Because this situation is impossible to detect in general and is not special to differentiation,[3] we do not consider it.

The corollary is that if two leaves do have the same type, we cannot tell at compile time whether they refer to the same object, as noted by Hogan (2014). Because we cannot predict evaluation flow, cached values must be dense matrices, not arbitrary expressions. We mitigate the runtime cost by wrapping intermediate Jacobians in `std::optional`, which allows efficient return of zero-valued Jacobians as a "null" object, $\emptyset$. The resulting combining functions are:

$$\text{Leaf:} \qquad f'_x(a) = \begin{cases} \mathbf{I}, & \text{if } a \equiv x \\ \emptyset, & \text{otherwise} \end{cases} \tag{3.6a}$$

$$\text{Unary:} \qquad f'_x(a, v) = \begin{cases} \emptyset, & \text{if } v = \emptyset \\ \mathbf{J}(a)v, & \text{otherwise} \end{cases} \tag{3.6b}$$

$$\text{Binary:} \qquad f'_x(a, v_l, v_r) = \begin{cases} \emptyset, & \text{if } v_l = v_r = \emptyset \\ \mathbf{J}_r(a)v_r, & \text{if } v_l = \emptyset, v_r \neq \emptyset \\ \mathbf{J}_l(a)v_l, & \text{if } v_l \neq \emptyset, v_r = \emptyset \\ \mathbf{J}_l(a)v_l + \mathbf{J}_r(a)v_r, & \text{otherwise.} \end{cases} \tag{3.6c}$$

[3] Examples of aliasing are given in Eigen's documentation: https://eigen.tuxfamily.org/dox/group__TopicAliasing.html

These equations, together with compile-time optimizations which prune some checks, describe `wave_geometry`'s `JacobianEvaluator`.

3.2.3 Strongly typed forward-mode AD

The implementation can be more efficient if it is guaranteed that all instances of a type refer to the same object. We call this property *uniqueness* of objects in a expression. Our second forward-mode AD implementation, the *strongly typed forward evaluator* (`TypedJacobianEvaluator`, assumes uniqueness. The identity check (3.5) is simplified to

$$(a \equiv x) = \begin{cases} 1, & \text{if } \mathrm{type}(a) = \mathrm{type}(x) \\ 0, & \text{otherwise.} \end{cases} \tag{3.7}$$

The combining functions used by `TypedJacobianEvaluator` are similar to functions (3.6) but with the conditionals $v = \emptyset$ known at compile time. This knowledge means Jacobian computation can be fully optimized, without converting placeholder expressions to dense matrices and without branches. Figure 3.4b illustrates the effect.

Our `contains_same_type` metafunction determines $v = \emptyset$ at compile time by applying another fold operation: it applies the type identity test $\mathrm{type}(a) = \mathrm{type}(x)$ to every a in a tree, and recursively combines the results with a logical OR.

Normally, the uniqueness property covers only a small subset of expressions, unless variables are "tagged" specifically to differentiate their types. However, the coordinate frame semantics system presented in Chapter 4 conveniently has the same effect, extending the uniqueness property to a large set of physically meaningful expressions.

As Baydin et al. (2018) points out, we can efficiently calculate the Jacobian-vector product $\mathbf{Jx}$ by initializing the forward pass with $\mathbf{x}$ instead of $\mathbf{I}$ in function (3.4a). However, if the Jacobian is available as a placeholder Eigen expression, it would be just as efficient to evaluate the product of that expression and the vector.

Is uniqueness guaranteed?

If *every* object in an expression tree has a unique type (which is true in Fig. 3.4b), the uniqueness property is *guaranteed*. We check the guarantee at compile time using another fold metafunction. This fold returns either a list of unique leaf types or a false value if

duplicate leaves exist in the tree. Its combining functions are:

$$
\begin{aligned}
&\text{Leaf:} && f(a) = \{\text{type}(a)\} && \text{(3.8a)}\\
&\text{Unary:} && f(a, v) = v && \text{(3.8b)}\\
&\text{Binary:} && f(a, v_l, v_r) = \begin{cases} \emptyset, & \text{if } v_l = \emptyset \text{ or } \cup v_r = \emptyset \\ \emptyset, & \text{if } v_l \cap v_r \neq \emptyset \\ v_l \cup v_r, & \text{otherwise.} \end{cases} && \text{(3.8c)}
\end{aligned}
$$

Here, the results v are sets of types, and an expression has unique leaves iff $v \neq \emptyset$. In our implementation, the result type represents the uniqueness value by deriving from either `std::true_type` or `std::false_type`, and holds a type list (which acts as the set v) in the true case. Each expression's `UniqueLeaves` trait holds the result. Our `concat_if_unique` metafunction implements function (3.8c) on type lists.

When `expr.jacobian(x)` is called, `wave_geometry` automatically selects the strongly typed Jacobian evaluator only if uniqueness is guaranteed at compile time (if `expr` has unique leaves). If uniqueness holds but is not guaranteed (when the same leaf appears more than once in `expr`), the user may invoke the strongly typed evaluator directly by calling `evaluateTypedJacobian(expr, x)`. That function asserts uniqueness at runtime in debug builds. However, it is often possible for the user to rearrange the expression so that each variable appears only once.

3.2.4 Reverse-mode AD

In reverse-mode AD, adjoints flow from the root node toward all leaves, as illustrated in Fig. 3.5. Here, assume that the expression is a true tree. We can loosely describe this backward pass as three unfolding operations, which take a node a and an input adjoint w_{in} and calculate the adjoint(s) w to pass on to its child(ren):

$$
\begin{aligned}
&\text{Leaf:} && w(a, w_{in}) = w_{in} && \text{(3.9a)}\\
&\text{Unary:} && w(a, w_{in}) = w_{in}\mathbf{J}(a) && \text{(3.9b)}\\
&\text{Binary:} && w_l(a, w_{in}) = w_{in}\mathbf{J}_l(a) && \\
& && w_r(a, w_{in}) = w_{in}\mathbf{J}_r(a). && \text{(3.9c)}
\end{aligned}
$$

The algorithm starts at the root node with $w_{in} = \mathbf{I}_m$, where m is the dimension of the expression's tangent space. These unfolding operations are performed in the constructor of our reverse evaluator, `ReverseJacobianEvaluator`. The Jacobian of the expression with

respect to a leaf, which is the adjoint that reaches that leaf, is cached at each leaf node of the reverse evaluator tree.

To actually return the Jacobians to the user, we follow the unfold with a fold: the cached Jacobians are packed into tuples, which are recursively concatenated using `std::tuple_cat`.

Our reverse evaluator is strongly typed, and the intermediate adjoints may be placeholder expressions. It is invoked by calling `expr.evalWithJacobians()` with no arguments for an `expr` with unique types. It returns a tuple of Jacobian matrices corresponding to all leaves in the expression in left-to-right order.

Future work would provide a reverse evaluator for cases without the uniqueness guarantee. For an expression that is a non-tree DAG, multiple adjoints reaching a node must be summed. To support complex expressions, implementations must also deal with increasing storage requirements (Baydin et al., 2018).

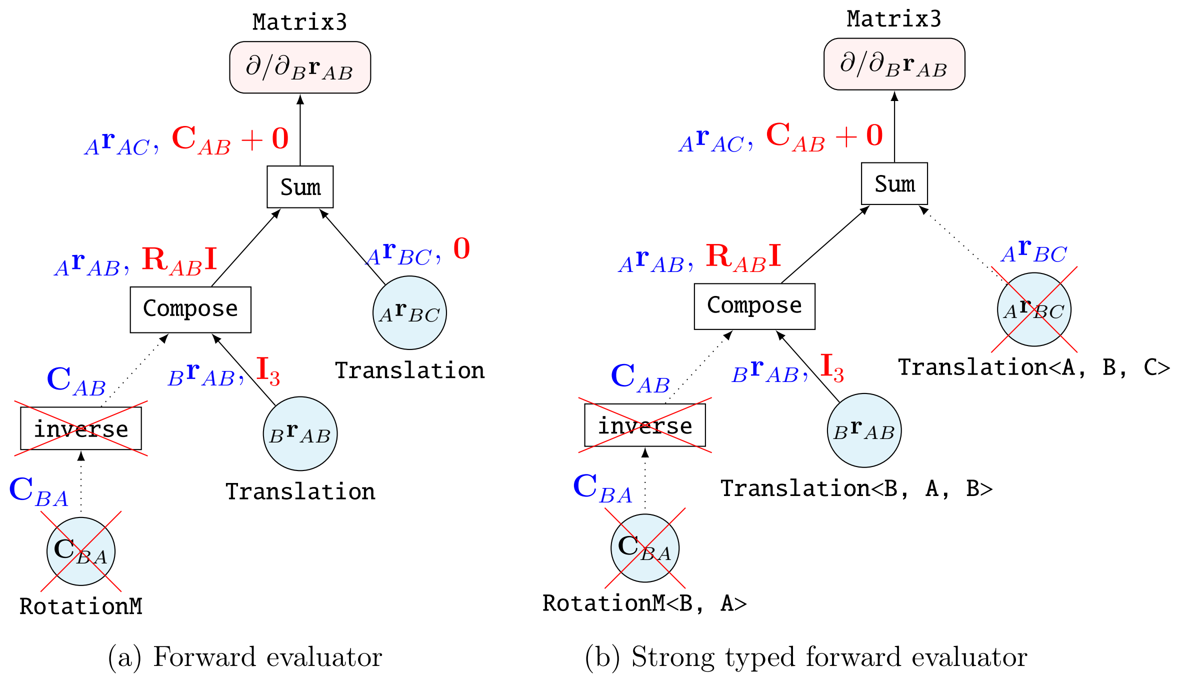
The unfolding operations (3.9) could be modified to compute derivatives only for some target leaves, by limiting the backward propagation to branches which contain the targets. In future work, the `expr.evalWithJacobians(x, ...)` interface could then be extended to choose the fastest evaluator for the given expression and targets.

3.3 Composing expressions at runtime

The compile-time nature of ETs is both a benefit and a drawback. Compared to the dynamic polymorphism provided by virtual functions, ETs allow superior optimizations and extensibility. However, dynamic polymorphism lets us decide the structure of an expression at runtime, make heterogenous vectors of expressions, spread compilation across translation units, or build deep expressions that would exceed compilers' template instantiation limits. We combine the two forms of polymorphism using *type erasure*: "the process of turning a wide variety of types with a common interface into one type with that same interface" (Abrahams and Gurtovoy, 2004).

Our application of the type erasure idiom consists of three parts: the `Proxy<L>` template class, which can hold any expression evaluating to a leaf type `L`; the `DynamicBase<L>` abstract base class, which provides an interface for the expression through virtual functions; and the `Dynamic` expression template, which acts as the "glue" between a concrete expression and the `DynamicBase<L>` interface.

Figure 3.6 illustrates how an instance of the `Dynamic` ET inherits from `DynamicBase` while retaining its concrete type. `Dynamic` is needed because expression types are unrelated



(a) Forward evaluator (b) Strong typed forward evaluator

Figure 3.4: Computation of the derivative of example (3.2) using forward AD. Cached values from the original function evaluation are shown above each node in blue, and derivatives in red. Crossed-out nodes are not traversed at runtime.

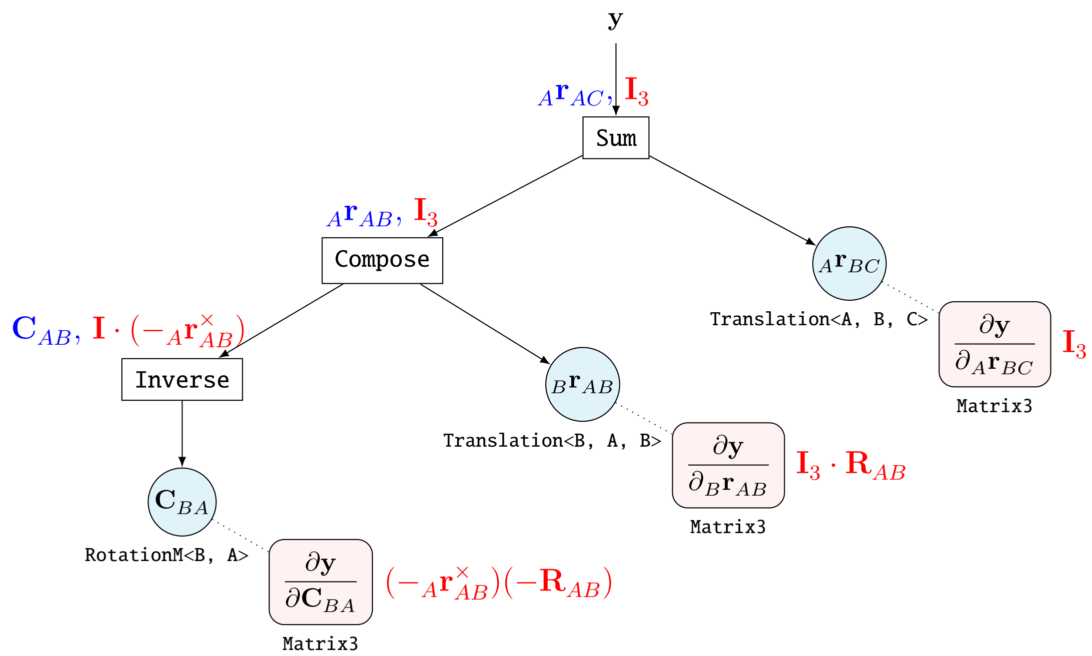


Figure 3.5: Reverse-mode differentiation of example (3.2). In the original function evaluation, values (blue) propagate upward. In reverse AD, adjoints (red) propagate downward to calculate the derivatives with respect to all leaves in one pass.

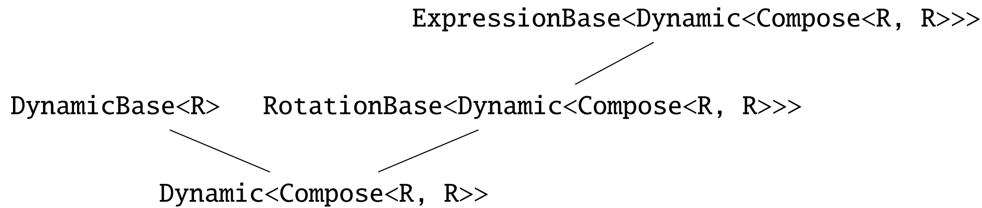


Figure 3.6: Simplified inheritance diagram for a Dynamic expression. Dynamic wraps an abritrary expression of some leaf type. It also inherits the abstract base class DynamicBase<R>, which provides a common interface for all expressions which evaluate to R.

in terms of inheritance, even if they represent the same leaf type: for example, RotationMd inherits from ExpressionBase<RotationMd>, while Inverse<RotationMd> inherits from ExpressionBase<Inverse<RotationMd>>. (While it would be possible to have every instance of ExpressionBase inherit from DynamicBase, that would introduce overhead related to virtual functions even when unneeded.) Note that Dynamic is an otherwise ordinary unary expression; it holds a reference to another expression and is implemented as an identity map.

Users interact with dynamic expressions through the Proxy expression. Proxy expressions are approximately equivalent to GTSAM's Expression class template, except that a proxy may itself contain an arbitrary static expression. Like GTSAM's Expressions, each proxy object holds a smart pointer to a dynamic expression allocated on the heap.

Proxies have pointer semantics: multiple proxies can point to the same dynamic expression, and can be shallowly copied and reassigned at runtime. Each Proxy<L> has the same static interface as the leaf type L, and is itself useable in other expessions. Proxies let us build arbitrary expression graphs at runtime, as illustrated in Fig. 3.7.

Their performance is compared to GTSAM in Section 3.4.4. Details on evaluation of dynamic expressions are given in Appendix C.2.

3.4 Experimental results

3.4.1 Benchmark experiments

We evaluate the runtime of our implementation compared to Ceres, GTSAM, and hand-coded derivatives using Eigen 3.3.4. Clang 5.0 was used with optimization flags -O3 -DNDEBUG -
↪ march=native on an Intel Core i7 Skylake processor. The Google Benchmark library

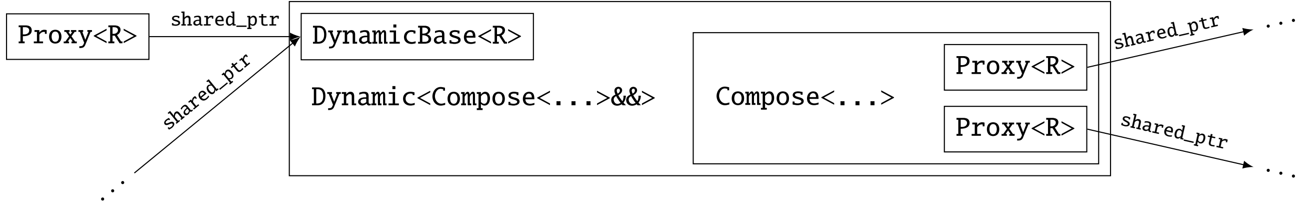


Figure 3.7: Example of a dynamically allocated expression graph. While the **Dynamic<...>** object holds an arbitrary expression, the **Proxy** is only aware of its resulting leaf type, **R**.

was used for timing. Results were averaged over repeated trials on sequences of random rotations.

3.4.2 Rotation Chain

First, we consider an increasingly long chain of rotations

$$\mathbf{r}_2 = \left(\prod_{i=1}^{N} \mathbf{R}_i \right) \mathbf{r}_1 \tag{3.10}$$

where $\mathbf{R}_i$ are rotation matrices. For example, for $N = 3$,

$$\mathbf{r}_2 = \mathbf{R}_1 \mathbf{R}_2 \mathbf{R}_3 \mathbf{r}_1. \tag{3.11}$$

This example is similar to (Sommer et al., 2013, eq. (6)). Applying the chain rule to the derivatives found in Bloesch et al. (2016) gives

$$\partial \mathbf{r}_2 / \partial \mathbf{r}_1 = \mathbf{R}_1 \mathbf{R}_2 \mathbf{R}_3, \tag{3.12}$$
$$\partial \mathbf{r}_2 / \partial \mathbf{R}_1 = -\mathbf{r}_2^{\wedge}, \tag{3.13}$$
$$\partial \mathbf{r}_2 / \partial \mathbf{R}_2 = -\mathbf{r}_2^{\wedge} \mathbf{R}_1, \tag{3.14}$$
$$\partial \mathbf{r}_2 / \partial \mathbf{R}_3 = -\mathbf{r}_2^{\wedge} \mathbf{R}_1 \mathbf{R}_2. \tag{3.15}$$

We use Eigen to hand-code (3.11) to (3.15), reusing intermediate values and evaluating $-\mathbf{r}^{\wedge}\mathbf{R}$ as column-wise cross products for efficiency.

Using Ceres presents a challenge, as explained with example code in Sommer et al. (2013): its `AutoDiffCostFunction` produces global Jacobians. Obtaining a local Jacobian requires the extra calculation of the derivative of the global parametrization with respect

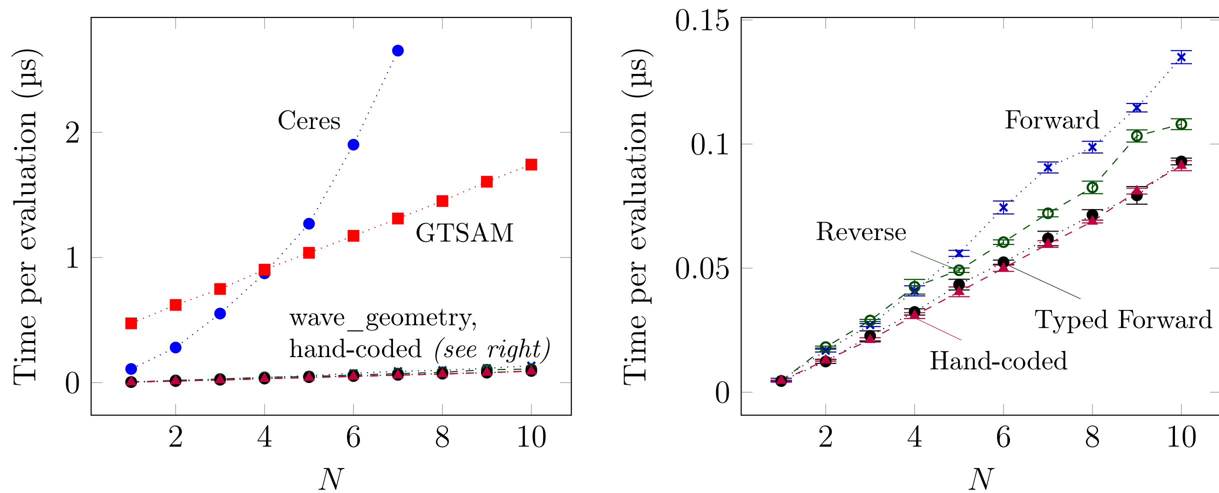


Figure 3.8: Comparison of time taken to evaluate result and all $N+1$ Jacobians in a chain of N rotations (3.10). Results are averaged over many trials. The left plot compares our results to existing libraries. The right plot shows the same data at a larger scale, comparing our three implementations to the hand-coded reference.

to the local. This is so inefficient for rotation matrices that it is not a realistic use case of Ceres, and the results we show for Ceres use quaternions.

Using GTSAM, we differentiate (3.11) as shown in Listing 3.13.

Listing 3.13: Automatic differentiation of a rotation chain in GTSAM

```
// Define expressions for inputs
Expression<Rot3> R1_{'R', 1}, R2_{'R', 2}, R3_{'R', 3};
Expression<Point3> p1_{'p', 1};

// Define expression for the rotation chain
Expression<Point3> p2_ = rotate(R1_ * R2_ * R3_, p1_);

// For each symbol, set a linearization point
Values values{};
values.insert(Symbol{'R', 1}, getRotation());
values.insert(Symbol{'R', 2}, getRotation()});
values.insert(Symbol{'R', 3}, getRotation()});

// Get result and all Jacobians
std::vector<Matrix> jacobians(4);
Point3 p2 = p2_.value(values, jacobians);
```

Listing 3.14 demonstrates the same task in `wave_geometry`, using the forward evaluator combined with coordinate frame semantics (labelled “typed forward” in results). The reverse evaluator is invoked by passing no arguments, as demonstrated in Listing 3.15. While these examples show C++17 syntax, the library can be used in C++11 and above.

Listing 3.14: Automatic differentiation of a rotation chain in `wave_geometry`

```
// Define inputs (with frame semantics)
wave::RotationMd<D, C> R1 = getRotation();
wave::RotationMd<C, B> R2 = getRotation();
wave::RotationMd<B, A> R3 = getRotation();
wave::Translationd<A> r1 = getPoint();

// Define the expression and differentiate
auto expr = R1 * R2 * R3 * r1;
auto[p2, J1, J2, J3, Jr] = expr.evalWithJacobians(R1, R2, R3, r1);
```

Listing 3.15: Reverse-mode AD in `wave_geometry`

```
// Differentiate in reverse mode
auto[p2, J1, J2, J3, Jr] =
   (R1 * R2 * R3 * r1).evalWithJacobians();
```

Figure 3.8 presents the results for N from 1 to 10. For this function, all three `wave_geometry` methods clearly outperform the existing libraries. The time taken by Ceres grows rapidly with N, matching the results of Sommer et al. (2013). GTSAM has a high initial overhead, but scales linearly, at a rate about 14 times that of the hand-coded reference.

Our typed forward evaluator's performance matches the reference, while the reverse evaluator has an average overhead of 24%. This represents an improvement over the $4\times$ slowdown reported in Sommer et al. (2013) for a similar example with quaternions. The typed forward evaluator can outperform the reverse because it naturally exploits the structure of this problem. In the next example, that is not the case.

GTSAM is disadvantaged in this comparison because it is designed for calculating sparse Jacobians of large graphs, not individual expressions. While our approach is faster than GTSAM's runtime tree, it does have a limitation: it requires advance knowledge of function flow, and cannot be used on arbitrary functions with unpredictable branching and loops, or on expressions composed at runtime. A combination of the two methods, using optimized ET-based AD within subtrees of a larger graph, could be a valuable improvement.

Table 3.2: Time to Evaluate Value and Jacobians of (3.16)

Algorithm	Hand-coded	Forward	Typed Forward	Reverse
Mean (ns)	302	642	596	360
Std. dev. (ns)	10	29	27	18

3.4.3 IMU Factor

Next, we evaluate our work on a sample expression simplified from a preintegrated IMU factor (Forster et al., 2017, eq. (45)). Let $\tilde{\mathbf{C}}_{IJ}$ represent a preintegrated measurement of rotation between times i and j, for which $\mathbf{R}_{WI}$ and $\mathbf{R}_{WJ}$ are the estimated orientations in the world frame. Let φ be an unknown small change in bias. The residual of the bias-updated preintegrated measurement is

$$\mathbf{r}_{ij} = \left(\tilde{\mathbf{C}}_{IJ} \boxplus \boldsymbol{\varphi}\right) \boxminus \mathbf{R}_{WI}^{-1}\mathbf{R}_{WJ} \tag{3.16}$$

which can be expressed as

$$\mathbf{r}_{ij} = \log\left(\left(\tilde{\mathbf{C}}_{IJ} \exp(\boldsymbol{\varphi})\right)^{-1} \circ \mathbf{R}_{WI}^{-1}\mathbf{R}_{WJ}\right). \tag{3.17}$$

Each of the four Jacobians of (3.17) contains the derivative of the logarithmic map (Forster et al., 2017, eq. (9)) which, compared to the derivatives of (3.10), is expensive to compute.

Table 3.2 presents the results. As expected when multiple Jacobians rely on an intermediate Jacobian calculation, the reverse evaluator outperforms the forward evaluator in this example, and is approximately 20% slower than the hand-coded reference.

3.4.4 Dynamic expressions

Figure 3.9 presents results of the rotation chain example (3.10) extended to large N using dynamic expressions (Section 3.3). `wave_geometry` outperforms GTSAM in this test, perhaps because GTSAM has an extra layer of indirection in identifying variables by a user-assigned key instead of a memory address. Neither library is designed for extremely deep expressions: GTSAM exceeds the system default 8192 KB stack size before reaching $N = 2^{11}$, and `wave_geometry` before $N = 2^{14}$. For both libraries, the recursive algorithm could be replaced with an iterative one to remove this limitation. `wave_geometry` takes

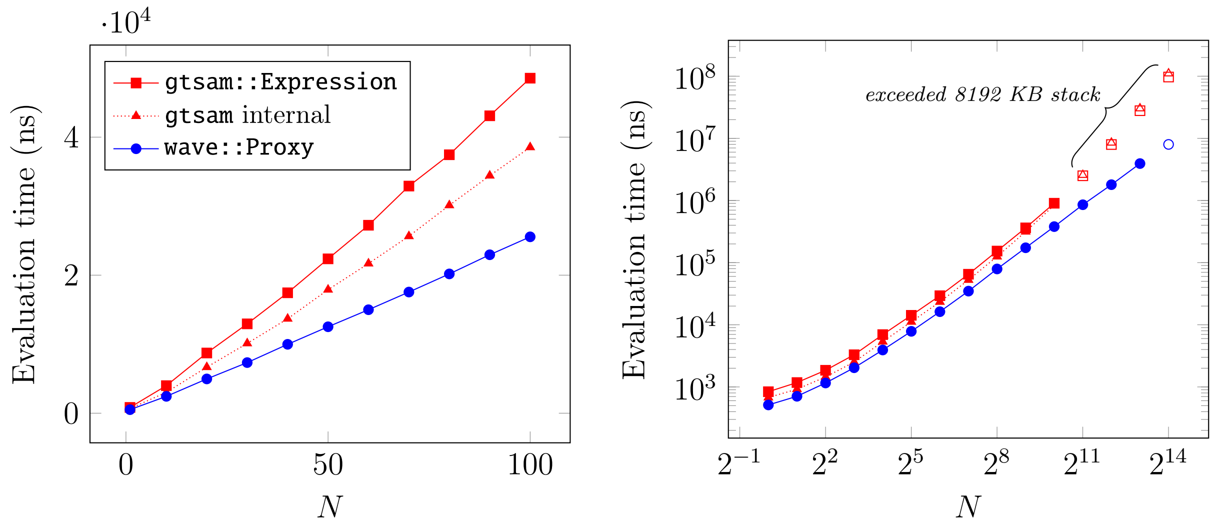


Figure 3.9: Time taken to evaluate result and all $N+1$ Jacobians in a chain of N rotations stored dynamically. Results for small N (left) and large N on logarithmic axes (right). Because GTSAM performs an extra copying step compared to `wave_geometry`, we show results for a comparable internal GTSAM function.

$N \log N$ time for large chains; this complexity is expected because the sorted vectors used to hold leaf addresses have $N \log N$ operations, and were chosen to minimize overhead for small N.

Chapter 4

Coordinate frame semantics checking

Geometric expressions encode *semantics*: information about the meaning of the symbols and their relationships. This section gives an overview of the semantics associated with coordinate frames, describes how checking semantics can detect common coding mistakes at compile time, and presents a system of rules for semantics which extends to manifold operations.

4.1 Coordinate frame semantics

A *coordinate frame* is a coordinate system fixed to a frame of reference. In our work, the coordinate system is always a right-handed Cartesian system, and the frame of reference is associated with a physical object such as a sensor. For a detailed discussion of the terms "reference frame," "coordinate system," and "coordinate frame," see Appendix B.

4.1.1 Coordinate-free geometry

In Section 2.2, we introduced the idea of *coordinate-free* geometry, in which operations are defined independently of the choice of coordinate system. For example, given three affine points A, B, C, the vectors between them satisfy

$$\mathbf{r}_{AB} + \mathbf{r}_{BC} \equiv \mathbf{r}_{AC}. \tag{4.1}$$

This relationship, called Chasles' identity (Gallier, 2011), is illustrated in Fig. 4.1. It holds independently of the coordinate system—if any—in which we choose to express the vectors.

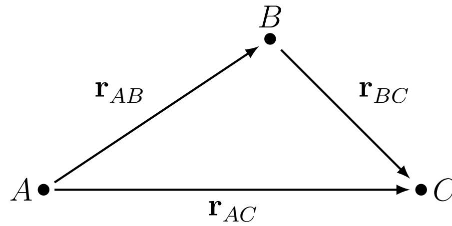


Figure 4.1: Coordinate-free vector addition.

Of course, to compute a numerical result, we need coordinates. In that case, the identity requires all three vectors to be expressed in the same coordinate system. For example, if we associate a coordinate frame with point A,

$$ {}_{A}\mathbf{r}_{AB} + {}_{A}\mathbf{r}_{BC} \equiv {}_{A}\mathbf{r}_{AC}. \tag{4.2} $$

Mixing coordinate frames is invalid, in general:

$$ {}_{A}\mathbf{r}_{AB} + {}_{B}\mathbf{r}_{BC} \not\equiv {}_{A}\mathbf{r}_{AC}. \tag{4.3} $$

Figure 4.2 illustrates this vector sum in two dimensions, in two different coordinate frames.

4.1.2 Expressing semantics through notation

The notation used for translation vectors in (4.2) and (4.3) can describe any physical quantity, as illustrated in Fig. 4.3. Vector quantities are written with three descriptors, in the form

$$ {}_{A}\mathbf{v}_{BC}, \tag{4.4} $$

where $\mathbf{v}$ stands in for any letter denoting a physical quantity; for example, $\mathbf{r}$ for translation or $\boldsymbol{\omega}$ for angular velocity. The right subscript (C) denotes the object which the quantity describes. The middle subscript (B) denotes the object in whose frame of reference the quantity is measured, also called the datum. The left subscript (A) denotes the object whose coordinate system is used to express the quantity.

These definitions, paraphrased from Kelly (2013), intentionally distinguish coordinate systems and frames of reference (see Appendix B). For simplicity, we let all three subscripts identify a coordinate frame (which we can attach to any object of interest) and call them *frame descriptors*.

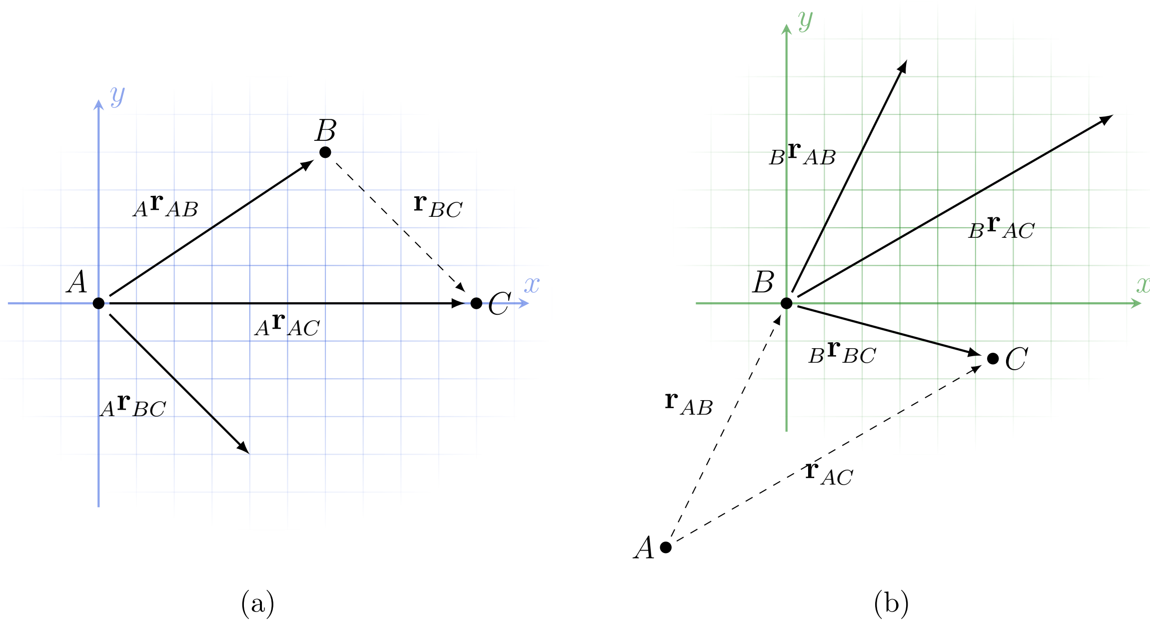


Figure 4.2: Vector addition expressed in a coordinate system. The vectors in the sum (4.1) are illustrated in arbitrarily defined coordinate frames at points A (a) and B (b), drawn with their orientation matching the page. While the coordinate-free result is always $\mathbf{r}_{AC}$, its representation in the chosen coordinate frames is different.

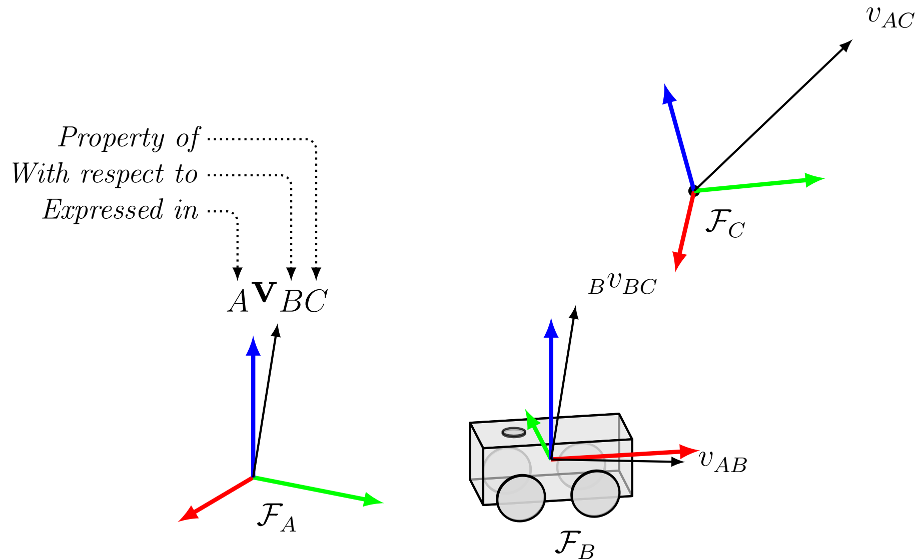


Figure 4.3: Example of frame descriptor notation for vectors. Here, $_A v_{BC}$ denotes the velocity of $\mathcal{F}_C$ as measured with respect to $\mathcal{F}_B$, expressed in the coordinate system of $\mathcal{F}_A$.

Maps (including rotations and rigid transformations) have two descriptors. The rotation $\mathbf{R}_{AB}$ rotates column vectors from $\mathcal{F}_B$ to $\mathcal{F}_A$, such that

$$_A\mathbf{v}_{BC} = \mathbf{R}_{AB\,B}\mathbf{v}_{BC}. \tag{4.5}$$

It can equivalently be interpreted as the rotation of axes from $\mathcal{F}_A$ to $\mathcal{F}_B$.[1]

The subscript notation is described by Furgale (2014), who attributes it to Bremer (2008). It is equivalent to the mixed script notation ($^A\underline{v}_C^B = R_B^{AB}\underline{v}_C^B$) of Kelly (2013). The subscript notation's clear left-to-right sequence is advantageous for representation in code, examined in Section 4.1.3. It is also easy to check for "cancellation" in equations, which is demonstrated in the next section.

For points, we define a shorthand using only two descriptors:

$$_A\mathbf{r}_B \triangleq {_A O} + {_A\mathbf{r}_{AB}}, \tag{4.6}$$

where $_A O$ is the origin of $\mathcal{F}_A$; this detail is needed because points are not vectors (see

[1] These opposite but equivalent interpretations can be called *active* and *passive* (Selig, 2006) or *operator* and *transform* (Kelly, 2013). Because these terms—and the words "to" and "from"—are easily misinterpreted, we do not rely on them to unambiguously describe a map.

Section 2.2). In effect, ${}_A\mathbf{r}_B$ has the same value as ${}_A\mathbf{r}_{AB}$. This notation matches its representation in code, making it slightly more cumbersome than Furgale's ${}_A\mathbf{r} \triangleq {}_A\mathbf{r}_{AP}$.

Using frame descriptors for semantics checking

Consider transforming the position of a landmark L, from a camera frame, $\mathcal{F}_C$, to a robot body frame, $\mathcal{F}_B$. We are given the landmark's position as ${}_C\mathbf{r}_{CL}$ ("the translation from $\mathcal{F}_C$ to $\mathcal{F}_L$, expressed in $\mathcal{F}_C$"). One possible expression for the position in the body frame is

$$ {}_B\mathbf{r}_{BL} = (\mathbf{C}_{CB})^{-1} {}_C\mathbf{r}_{CL} + {}_B\mathbf{r}_{BC}. \tag{4.7}$$

We can check the validity of this equation by propagating frame descriptors through elementary operations and ensuring that adjacent descriptors cancel. Crucially, the semantics of the final result are wholly determined by combinations of elementary operations:

$$\begin{aligned}
(\mathbf{C}_{CB})^{-1} {}_C\mathbf{r}_{CL} + {}_B\mathbf{r}_{BC} &= \mathbf{C}_{B\cancel{C}\cancel{C}}\mathbf{r}_{CL} + {}_B\mathbf{r}_{BC} && (4.8a)\\
&= {}_B\mathbf{r}_{CL} + {}_B\mathbf{r}_{BC} && (4.8b)\\
&= {}_B\mathbf{r}_{B\cancel{C}} + {}_B\mathbf{r}_{\cancel{C}L} && (4.8c)\\
&= {}_B\mathbf{r}_{BL}. && (4.8d)
\end{aligned}$$

Mistakes such as an omitted inverse result in invalid operations, which are indicated by mismatching subscripts:

$$ {}_B\mathbf{r}_{BL} = \mathbf{C}_{CBC}\mathbf{r}_{CL} + {}_B\mathbf{r}_{BC}. \tag{4.9}$$

This analysis is independent of the numeric value and parametrization of the variables. Equation (4.9) is semantically incorrect despite being readily computable.

4.1.3 Semantics in Code

To help avoid mistakes such as (4.9) in code, Furgale (2014) recommends using prefix notation in variable names, as in Listing 4.1. While this convention helps programmers parse the code and spot mistakes, it does not prevent an invalid expression from compiling. To do so, we encode coordinate frame semantics into the *types* of variables, instead of their names, as in Listing 4.2.

Listing 4.1: Frame semantics expressed in variable names

```
Position B_r_BL = C_C_B.inverse() * C_r_CL + B_p_B_C;
```

Listing 4.2: Frame semantics embedded in `wave_geometry` types

```
struct Body;     // Represents the robot frame
struct Camera;   // Represents the camera frame
struct Landmark; // Represents a landmark frame

TranslationFd<Body, Body, Landmark> landmarkToBody(
  const RotationMFd<Body, Camera>& C_cam,
  const TranslationFd<Body, Body, Camera>& p_cam,
  const TranslationFd<Camera, Camera, Landmark>& p) {
    return C_cam * p + p_cam;
}
```

This approach states the meaning of each variable and function at declaration, without cluttering internal code. Invalid operations will cause an error at compile time.

Outside of this toy example, it would of course be simpler to use a single transformation object instead of a separate rotation and translation. We examine transformations in the next section.

4.1.4 Free and bound transformations

Consider the equation which obtains ${}_B\mathbf{r}_{BL}$ from ${}_C\mathbf{r}_{CL}$: it is not only a coordinate transformation, as the coordinate-free meanings of $\mathbf{r}_{BL}$ and $\mathbf{r}_{CL}$ are different. It can be seen as deriving a physical relationship dependent on the relationship of frames B and C. Kelly (2013) describes this equation as a *bound* transformation.

A bound transformation to $\mathcal{F}_B$ treats ${}_C\mathbf{r}_{CL}$ as a *bound* vector, moving its tail to produce ${}_B\mathbf{r}_{BL}$. A *free* transformation treats treats ${}_C\mathbf{r}_{CL}$ as a *free* vector, and merely expresses it in the coordinate system of $\mathcal{F}_B$. For translations, we have the following transformations, illustrated by Fig. 4.4:

$$\mathbf{T}_{BA}^{bound}({}_A\mathbf{r}_{AC}) \triangleq {}_B\mathbf{r}_{BC} = \mathbf{R}_{BA}({}_A\mathbf{r}_{AC}) + {}_B\mathbf{r}_{BA}, \tag{4.10}$$

$$\mathbf{T}_{BA}^{free}({}_A\mathbf{r}_{AC}) \triangleq {}_B\mathbf{r}_{AC} = \mathbf{R}_{BA}({}_A\mathbf{r}_{AC}). \tag{4.11}$$

For other quantities, such as angular velocity measured in a rotating frame, the bound transformation is more complicated; however, the free transformation is always given by equation (4.11).

When designing a geometry library, the question arises: should `T * r` produce the result of $\mathbf{T}^{bound}$ or $\mathbf{T}^{free}$?

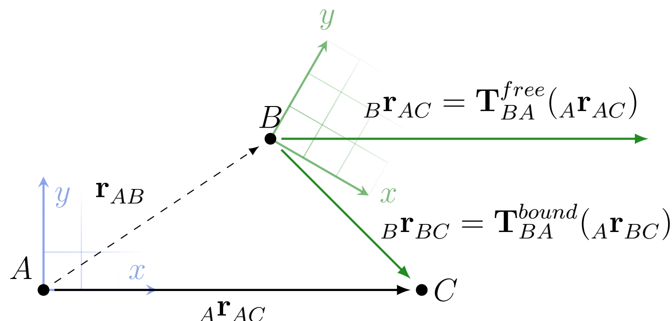


Figure 4.4: Free and bound transformations. A transformation treating $_A\mathbf{r}_{AC}$ as a free vector expresses it in the coordinate system of $\mathcal{F}_B$. A transformation treating $_A\mathbf{r}_{AC}$ as a bound vector performs a change of reference, changing its length and direction.

As mentioned in Section 2.2.4, it is common to represent both points and vectors as 4-vectors and use the affine coordinate to determine the effect of a homogeneous transformation matrix: an affine coordinate of 0 removes the effect of translation, producing a free transformation. Points use an affine coordinate of 1. Kelly (2013) calls these classes displacements (free vectors) and position vectors (bound vectors, which can represent points)

In our library, inhomogeneous points are stored as 3-vectors and rigid transformations are not always stored as 4×4 matrices. Still, a transformation acting on a point has the effect of $\mathbf{T}^{bound}$. Our `Translation` objects are treated as bound vectors, although rotation vectors are treated as free vectors. This choice is somewhat arbitrary but consistent with other libraries which do not differentiate positions and displacements.

What is important is that the frame semantics of the result reflect which transformation was applied: note that the results of (4.10) and (4.11) have different subscripts. With semantics checking in code, it is impossible to mistakenly assign $_B\mathbf{r}_{AC} :\neq \mathbf{T}^{bound}_{BA}(_A\mathbf{r}_{AC})$.

4.2 C++ library implementation

Frame semantics are integrated into `wave_geometry`'s expression template implementation. Using frame semantics is optional, and is done by wrapping leaf types with the `Framed` class template. For example, `Framed<RotationMd, A, B>` (or its alias `RotationMFd<A, B>`) represents the rotation $\mathbf{C}_{AB}$. `Framed` objects have the same interface as their wrapped leaves, modulo the frame semantics rules. For example, both sides of an assignment must

Listing 4.3: Mistaken frame transformation

```cpp
TranslationFd<Body, Body, Landmark> positionInBody(
    RigidTransformMFd<Camera, Body> T,
    TranslationFd<Camera, Camera, Landmark> p) {
    return T * p;
}
```

```
[ 50%] Building CXX object test/CMakeFiles/wrong_frames_example.dir/wrong_frames_exa
mple.cpp.o
In file included from /home/leo/code/wave_geometry/test/wrong_frames_example.cpp:6:
In file included from /home/leo/code/wave_geometry/include/wave/geometry/geometry.hp
p:44:
/home/leo/code/wave_geometry/include/wave/geometry/src/geometry/op/Transform.hpp:21:
5: error: static_assert failed "Adjacent frames do not match"
    static_assert(std::is_same<RightFrameOf<Lhs>, LeftFrameOf<Rhs>>(),
    ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/home/leo/code/wave_geometry/test/wrong_frames_example.cpp:17:14: note: in instantia
tion of template class 'wave::Transform<wave::Framed<wave::MatrixRigidTransform<Eige
n::Matrix<double, 4, 4, 0, 4, 4> >, Camera, Body>, wave::Framed<wave::Translation<Ei
gen::Matrix<double, 3, 1, 0, 3, 1> >, Camera, Camera, Landmark> >' requested here
    return T * p;
             ^
1 error generated.
```

Figure 4.5: Error message printed by Clang when compiling Listing 4.3.

have matching frames. The frame descriptors `A` and `B` themselves are arbitrary type names, which can be declared as `struct`s with no definition. It is left to the user's application code to define appropriately named frames.

Each expression's semantics are encoded in its traits class. Non-leaf expressions propagate the frames of their operands according to the rules in Section 4.4. To simplify the implementation of each expression, frames are stripped from `Framed` objects at the start of the evaluation procedure (see Section 3.1.3). The result of the evaluation is wrapped in `Framed` before output to the user.

Semantics checking is performed in each expression template, as seen in Listing 3.4. C++'s `static_assert` mechanism is used to trigger a compilation error if semantics rules are broken. For example, the error message produced by Listing 4.3, which is a mistaken variation of Listing 4.2, is shown in Fig. 4.5.

In case the user wishes to except a particular operation from the usual frame semantics rules, we provide a `frame_cast` function.

This function, demonstrated in Listing 4.4, explicitly overrides frame semantics of its operand. It can be used in differentiable expressions.

Listing 4.4: Overriding frame semantics with `frame_cast`

```
TranslationFd<A, A, B> a{};
TranslationFd<A, B, C> b{};
// TranslationFd<X, Y, Z> c = a + b; // Error: mismatching frames
TranslationFd<X, Y, Z> c = frame_cast<X, Y, Z>(a + b); // OK
```

4.3 Related works

Our compile-time frame checking is inspired by C++ techniques for dimensional analysis (unit checking), described by Barton and Nackman (1994). This task similarly involves storing semantic operation in template arguments.

De Laet et al. (2013a,b) introduces "semantics checking" for geometric expressions with a similar goal of preventing calculation errors. Their formulation is somewhat more complex than ours and uses a larger set of descriptors because it models rigid bodies. Their Geometric Relations Semantics software is meant to extend existing geometry libraries, essentially by providing wrapper class templates. This software is demonstrated by extending KDL[2].

Unlike our library, De Laet's software stores semantic information inside objects and performs checks at runtime. This design adds time and space overhead, and means semantics violations may be missed until they occur in production.

DeRose (1989) addresses the same problem but takes a *coordinate-free* approach. Instead of checking semantics, DeRose's solution is to abstract away coordinate frames entirely, leaving the programmer to work with coordinate-free objects (such as $\mathbf{v}_{AB}$), at least in intermediate calculations.

DeRose's "coordinate-free abstract data type" in C, updated to C++ by Mann et al. (1997), supports Euclidean and projective operations. In the coordinate-free abstraction, all objects are associated with a "space" but not a coordinate frame. Internally, all objects sharing a space are actually stored in a single coordinate system. The library takes care of transformations when objects are initialized or when coordinates of results are extracted. For these operations, the user provides a Frame object.

The provided implementation[3] also uses runtime objects for frames and spaces, and the transformations themselves impose a runtime overhead. As De Rose describes, removing the programmer's control over the coordinate system for each object can also be a disadvantage.

[2]http://www.orocos.org/wiki/geometric-relations-semantics-wiki

[3]https://web.archive.org/web/20110514024435/http://www.cgl.uwaterloo.ca/software.html

4.4 Rules for frame semantics

Table 4.1 presents a system of rules used for checking and propagating coordinate frame semantics.

While semantics checking can catch common mistakes, it is important to realize it cannot handle every case. For example, it cannot verify that the numeric value of ${}_A\mathbf{r}_{BB}$ is zero. Indeed, ${}_A\mathbf{r}_{BB}$ could reasonably represent a residual obtained by subtracting a measured and estimated vector, or a perturbation to be added to ${}_A\mathbf{r}_{AB}$ at discrete time steps. `wave_geometry` follows the principle that "everything which is not forbidden is allowed", and never requires distinct descriptors.[4] Our choice of rules may change in future work based on use of the library, as it is intended to support best practices without encumbering common tasks.

For example, it is common to update a rotation by integrating an angular velocity at discrete time steps:

$$\mathbf{R}_{AB} \coloneqq \mathbf{R}_{AB} \boxplus (\Delta t \cdot {}_A\omega_{AB})^{\wedge}. \tag{4.24}$$

While the left hand side represents a perturbed or updated version of $\mathbf{R}_{AB}$, explicitly representing the perturbed frame would require $\boxplus$ to introduce a coordinate frame label not present in its operands. We do not force the declaration of a perturbed frame, resulting in the rule (4.20) for $\boxplus$.

Starting with (4.20), we apply (2.48) to obtain

$$\mathbf{\Phi}_{AB} = \exp({}_A\boldsymbol{\varphi}_{AB}) \circ \mathbf{\Phi}_{AB}. \tag{4.25}$$

It follows that (4.22) is $\exp({}_A\boldsymbol{\varphi}_{AB}) = \mathbf{\Phi}_{AA}$. This makes the exponential map semantically non-bijective: it loses coordinate frame information, for which $\log(\exp(\boldsymbol{\varphi})) = \boldsymbol{\varphi}$ does not hold. Consequently, applying (2.49) to (4.21) produces the unsatisfactory result $\log(\mathbf{\Phi}_{AA}) = {}_A\boldsymbol{\varphi}_{AB}$. A solution is to add an extra coordinate frame argument to the logarithmic map, as shown in (4.23).

[4] In fact, all rules in Table 4.1 are satisfied by expressions with all-identical (or all-unset) descriptors, such as ${}_A\mathbf{v}_{AA}$, and using semantics checking in `wave_geometry` is entirely optional.

Table 4.1: Rules for Semantics of Geometric Operations

Operation	Rule	
Sum	${}_D\mathbf{v}_{AC} = {}_D\mathbf{v}_{AB} + {}_D\mathbf{v}_{BC}$ $= {}_D\mathbf{v}_{BC} + {}_D\mathbf{v}_{AB}$	(4.12)
Negative	${}_D\mathbf{v}_{BA} = -{}_D\mathbf{v}_{AB}$	(4.13)
Difference	${}_D\mathbf{v}_{AB} = {}_D\mathbf{v}_{AC} - {}_D\mathbf{v}_{BC}$	(4.14)
Scaling	${}_A\mathbf{v}_{BC} = a({}_A\mathbf{v}_{BC}), \quad a \in \mathbb{R}$	(4.15)
Composition	$\mathbf{\Phi}_{AC} = \mathbf{\Phi}_{AB} \circ \mathbf{\Phi}_{BC}$	(4.16)
Inverse	$\mathbf{\Phi}_{BA} = (\mathbf{\Phi}_{AB})^{-1}$	(4.17)
Rotation	${}_D\mathbf{v}_{BC} = \mathbf{R}_{DA}({}_A\mathbf{v}_{BC})$	(4.18)
Transformation (bound)	${}_A\mathbf{r}_{AC} = \mathbf{T}_{AB}({}_B\mathbf{r}_{BC})$ $= \mathbf{R}_{AB}({}_B\mathbf{r}_{BC}) + {}_A\mathbf{r}_{AB}$	(4.19)
Manifold plus	$\mathbf{\Phi}_{AB} = \mathbf{\Phi}_{AB} \boxplus {}_A\boldsymbol{\varphi}_{AB}$	(4.20)
Manifold minus	${}_A\boldsymbol{\varphi}_{AB} = \mathbf{\Phi}_{AB} \boxminus \mathbf{\Phi}_{AB}$	(4.21)
Exp map	$\mathbf{\Phi}_{AA} = \exp({}_A\boldsymbol{\varphi}_{AB})$	(4.22)
Log map	${}_A\boldsymbol{\varphi}_{AB} = \log_B(\mathbf{\Phi}_{AA})$	(4.23)

Valid operands for each operation are shown on the right hand side, and the result is shown on the left. Repeated frame labels must match.

Chapter 5

Application to state estimation

5.1 State estimation as a least squares problem

Our goal in representing and differentiating geometric expressions is to simplify the writing of state estimation algorithms. As defined by Barfoot (2017),

> The problem of state estimation is to come up with an estimate, $\hat{\mathbf{x}}_k$, of the true state of a system, at one or more timesteps, k, given knowledge of the initial state, $\check{\mathbf{x}}_0$, a sequence of measurements, $\mathbf{y}_{0:K,meas}$, a sequence of inputs, $\mathbf{v}_{1:K}$, as well as knowledge of the system's motion and observation models.

In this section, we review how uncertain measurements are represented, how uncertainty is propagated, and how measurement models are converted into cost functions. While the broad range of estimation and SLAM algorithms is outside the scope of this work, this limited summary is intended to motivate the use of automatically differentiated geometric expressions as cost functions for state estimation.

We then consider a specific instance of the state estimation problem—batch, discrete-time, nonlinear maximum a posteriori estimation with Gaussian assumptions—and formulate it as a general nonlinear least squares problem. For simplicity, we will consider a problem with K measurements $\mathbf{y}_k$ and a prior $\check{\mathbf{x}}_0$, and no inputs $\mathbf{v}$. Nonlinear least squares

5.1.1 Uncertain estimates

State estimates and measurements are uncertain. Typically, we represent discrete-time quantities as Gaussian random variables. The notation

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{5.1}$$

indicates a Gaussian variable $\mathbf{x}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$. Definitions and properties of probability density functions (PDFs) and Gaussian variables can be found in Chirikjian (2009, Chapter 3) and Barfoot (2017, Chapter 2).

For a Euclidean random variable $\mathbf{x} \in \mathbb{R}^n$, we simply have $\boldsymbol{\mu} \in \mathbb{R}^n$ and $\boldsymbol{\Sigma} \in \mathbb{R}^{n\times n}$. We consider propagating this random variable through linear and nonlinear functions before making the extension to states lying on a manifold.

Propagating covariance

Consider the Gaussian random variable

$$\mathbf{x} \in \mathbb{R}^n \sim \mathcal{N}(\boldsymbol{\mu}_x, \boldsymbol{\Sigma}_{xx}). \tag{5.2}$$

The result of applying a linear map

$$\mathbf{y} = \mathbf{G}\mathbf{x}, \tag{5.3}$$

where $\mathbf{G} \in \mathbb{R}^{m\times n}$, is a Gaussian variable $\mathbf{y} \in \mathbb{R}^m$ with the properties

$$\mathbf{y} \sim \mathcal{N}(\mathbf{G}\boldsymbol{\mu}_x, \mathbf{G}\boldsymbol{\Sigma}_{xx}\mathbf{G}^\mathsf{T}). \tag{5.4}$$

Passing the variable through a stochastic nonlinear function, however, does not always allow a closed-form solution for the covariance. Consider a nonlinear map $\mathbf{y} = g(\mathbf{x})$ which introduces purely additive zero-mean Gaussian noise with covariance $\mathbf{R}$:

$$p(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(g(\mathbf{x}), \mathbf{R}). \tag{5.5}$$

For example, if $\mathbf{g}$ represents a sensor measurement model, $\mathbf{R}$ is the uncertainty associated with measurement noise. Propagating covariance through $g(\mathbf{x})$ requires the evaluation of the PDF

$$p(\mathbf{y}) = \int_{\mathbb{R}^n} p(\mathbf{y} \mid \mathbf{x}) p(\mathbf{x}) d\mathbf{x}. \tag{5.6}$$

The most common general solution is *linearization*. We use the first-order approximation

$$g(\mathbf{x}) \approx g(\boldsymbol{\mu}_x) + \mathbf{G}(\mathbf{x} - \boldsymbol{\mu}_x), \tag{5.7}$$

where $\mathbf{G}$ is the Jacobian of g with respect to $\mathbf{x}$:

$$\mathbf{G} = \left.\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}\right|_{\mathbf{x}=\boldsymbol{\mu}_x}. \tag{5.8}$$

The linearized result (derived in Barfoot, 2017, Section 2.2.8) is

$$\mathbf{y} \sim \mathcal{N}(g(\boldsymbol{\mu}_x), \mathbf{R} + \mathbf{G}\boldsymbol{\Sigma}_{xx}\mathbf{G}^{\mathsf{T}}). \tag{5.9}$$

Random variables on a manifold

There is no single definition of mean and covariance on manifolds, which are not in general equipped with distance functions. For Lie Groups, the mean and covariance can be expressed using the logarithmic map. The $m \times m$ covariance matrix of a PDF $p(\mathcal{g})$ on an m-dimensional Lie group G is (Chirikjian, 2011, eq. 20.51)

$$\boldsymbol{\Sigma}_p = \int_G \log(\mathcal{g})^{\vee}[\log(\mathcal{g})^{\vee}]^{\mathsf{T}} p(\mathcal{g}) d\mathcal{g}. \tag{5.10}$$

The mean can be defined as the solution μ to (Barfoot, 2017, eq. 7.272)

$$\int_G \log\left(\mathcal{g}\mu^{-1}\right)^{\vee} p(\mathcal{g}) d\mathcal{g} = \mathbf{0}. \tag{5.11}$$

However, it is not immediately clear how to define a PDF on a Lie group. As noted by Barfoot (2017, Section 7.3), a Gaussian random variable $\mathbf{x} \in \mathbb{R}^n \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ can be equivalently expressed as

$$\mathbf{x} = \boldsymbol{\mu} + \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}), \tag{5.12}$$

but Lie groups do not have an addition operation.

Barfoot's approach is to express the PDF in the Lie algebra (which, as a vector space, has well-defined mean and covariance) and carry it to the Lie group using the exponential map. We replace the vector addition operation $+$ in (5.12) with a perturbation in the tangent space. For example, applying a perturbation on the left (by convention), we can

express a random variable on SO(3) as

$$\mathbf{C} = \exp(\boldsymbol{\epsilon}^{\vee})\bar{\mathbf{C}}, \tag{5.13}$$

where $\bar{\mathbf{C}} \in \mathrm{SO}(3)$ is a noise-free nominal rotation and $\boldsymbol{\epsilon} \in \mathbb{R}^3$ is a "small" random variable in a vector space, for which we can easily define a PDF. If we let $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$, it can be shown that $\bar{\mathbf{C}}$ is the mean rotation according to definition (5.11).

The reason $\boldsymbol{\epsilon}$ must be "small" is that the exponential map is not injective: infinitely many values of $\boldsymbol{\epsilon}$ produce the same $\mathbf{C}$. Thus the statistical properties of the PDF induced onto the Lie group only match those of $\boldsymbol{\epsilon}$ if all of the probability mass is restricted to the neighbourhood where the exponential map is bijective. For a Gaussian $\boldsymbol{\epsilon}$, whose PDF extends to infinity, this approach is a reasonable approximation when most of the probability mass lies in the bijective region (for SO(3), the region $\|\boldsymbol{\epsilon}\| < \pi$). This issue is discussed in Hertzberg et al. (2013, Appendix A.9).

Hertzberg et al. (2013) takes a similar approach to defining random variables on $\boxplus$-manifolds (which are more general than Lie groups). Generalizing (5.12) and (5.13), we can write

$$\chi = \mu \boxplus \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}). \tag{5.14}$$

To a first order approximation, this variable can be propagated through a nonlinear map with the resulting distribution given by (5.9). There are a number of approaches to higher-order approximations (see Barfoot, 2017), but they are outside the scope of this work.

5.1.2 Maximum a Posteriori

This section presents a brief summary of one kind of state estimation, maximum a posteriori estimation with a nonlinear, Gaussian model. It is based on Barfoot (2017) and Dellaert et al. (2017).

The *maximum a posteriori* (MAP) estimate is the solution to the problem

$$\hat{\mathbf{x}}_{MAP} = \underset{\mathbf{x}}{\operatorname{argmax}}\, p(\mathbf{x}|\mathbf{y}). \tag{5.15}$$

That is, it maximizes the posterior probability density of the state $\mathbf{x}$ given the measurements $\mathbf{y}$. Bayes' law gives

$$\hat{\mathbf{x}}_{MAP} = \underset{\mathbf{x}}{\operatorname{argmax}}\, \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{y})}, \tag{5.16}$$

where we can discard the denominator because it does not affect the maximization over $\mathbf{x}$. Equivalently (because log is a monotonic function), we can minimize the negative log likelihood:

$$\hat{\mathbf{x}}_{MAP} = \underset{\mathbf{x}}{\operatorname{argmin}} \Big(-\log p(\mathbf{y}|\mathbf{x}) - \log p(\mathbf{x}) \Big). \tag{5.17}$$

Here, $p(\mathbf{y}|\mathbf{x})$ is the PDF of the observed measurements given a state, and $p(\mathbf{x})$ is a prior probability distribution. If we assume that measurement noise is uncorrelated, we can factor $p(\mathbf{y}|\mathbf{x})$ in terms of individual measurements:

$$p(\mathbf{y}|\mathbf{x}) = \prod_{k=1}^{K} p(\mathbf{y}_k|\mathbf{x}_k), \tag{5.18}$$

$$\log p(\mathbf{y}|\mathbf{x}) = \sum_{k=1}^{K} \log p(\mathbf{y}_k|\mathbf{x}_k). \tag{5.19}$$

If we further assume additive Gaussian noise on measurements, we can write each as a *factor* in the form

$$f_i(\mathbf{x}_i) \propto \exp\left(-\frac{1}{2} \|\mathbf{h}_i(\mathbf{x}_i) - \mathbf{y}_i\|^2_{\boldsymbol{\Sigma}_i} \right), \tag{5.20}$$

where $\mathbf{h}_i(\mathbf{x}_i)$ is the measurement prediction function corresponding to $\mathbf{y}_i$, $\boldsymbol{\Sigma}_i$ is the measurement covariance, and $\|\cdot\|^2_{\boldsymbol{\Sigma}}$ is the Mahalanobis distance defined as

$$\|\boldsymbol{\theta} - \boldsymbol{\mu}\|^2_{\boldsymbol{\Sigma}} \triangleq (\boldsymbol{\theta} - \boldsymbol{\mu})^{\mathsf{T}} \boldsymbol{\Sigma}^{\mathsf{T}} (\boldsymbol{\theta} - \boldsymbol{\mu}). \tag{5.21}$$

In this formulation, there is no inherent difference between measurements and priors. We can equally write Gaussian priors as factors of the form (5.20), substituting the identity function for $\mathbf{h}_i$, and consider just one set of factors f_i. Then, we can write the original MAP problem as (Dellaert et al., 2017, eq. 2.4)

$$\hat{\mathbf{x}}_{MAP} = \underset{\mathbf{x}}{\operatorname{argmin}} \sum_i \|\mathbf{h}_i(\mathbf{x}_i) - \mathbf{y}_i\|^2_{\boldsymbol{\Sigma}_i}. \tag{5.22}$$

The MAP estimation problem is now in the form of a *nonlinear least squares* (NLS) problem. Given a sufficiently close initial estimate of $\mathbf{x}$, a global minimum can be found using iterative methods such as Gauss-Newton or Levenberg-Marquardt (Dellaert et al., 2017).

By applying the $\boxplus$-method of Hertzberg et al. (2013), we can extend equation (5.22) to

$\boxplus$-manifolds:

$$\hat{\mathbf{x}}_{MAP} = \underset{\mathbf{x}}{\operatorname{argmin}} \sum_i \|\mathbf{h}_i(\mathbf{x}_i) \boxminus \mathbf{y}_i\|^2_{\mathbf{\Sigma}_i}. \tag{5.23}$$

The states being estimated can now be manifold elements. This formulation is known as *on-manifold optimization*. We continue to apply this method in the next section.

Linearization

NLS optimization algorithms iteratively solve linear approximations of the objective function (5.22). We take a linear approximation of each function about a linearization point $\mathbf{x}_i^0$:

$$h_i(\mathbf{x}_i) \approx h_i(\mathbf{x}_i^0) \boxplus \mathbf{H}_i \boldsymbol{\delta}_i, \tag{5.24}$$

where $\mathbf{H}_i$ is the *measurement Jacobian*

$$\mathbf{H}_i = \left. \frac{\partial h_i(\mathbf{x})}{\partial \mathbf{x}_i} \right|_{\mathbf{x}_i^0} \tag{5.25}$$

and $\boldsymbol{\delta}_i$ is the *state update vector*,

$$\boldsymbol{\delta}_i \triangleq \mathbf{x}_i \boxminus \mathbf{x}_i^0. \tag{5.26}$$

Each iteration then solves the linear least squares problem for the state update vector:

$$\hat{\boldsymbol{\delta}} = \underset{\boldsymbol{\delta}}{\operatorname{argmin}} \sum_i \left\| \left(\mathbf{h}_i(\mathbf{x}_i^0) \boxplus \mathbf{H}_i \boldsymbol{\delta}_i\right) \boxminus \mathbf{y}_i \right\|^2_{\mathbf{\Sigma}_i} \tag{5.27}$$

$$\approx \underset{\boldsymbol{\delta}}{\operatorname{argmin}} \sum_i \left\| \mathbf{H}_i \boldsymbol{\delta}_i - \left(\mathbf{y}_i \boxminus \mathbf{h}_i(\mathbf{x}_i^0)\right) \right\|^2_{\mathbf{\Sigma}_i}, \tag{5.28}$$

where $\mathbf{y}_i \boxminus \mathbf{h}_i(\mathbf{x}_i^0)$ is the *prediction error*.

Whitening

When solving least squares problems it is simplest to work with variables whose covariance is identity. To obtain these *whitened* variables we use a property of the Mahalanobis norm,

$$\|\mathbf{x}\|^2_{\mathbf{\Sigma}} = \left\| \mathbf{\Sigma}^{-1/2} \mathbf{x} \right\|^2_2, \tag{5.29}$$

to obtain

$$\mathbf{A}_i = \mathbf{\Sigma}^{-1/2}\mathbf{H}_i \tag{5.30}$$

$$\mathbf{b}_i = \mathbf{\Sigma}^{-1/2}\left(\mathbf{y}_i \boxminus \mathbf{h}_i(\mathbf{x}_i^0)\right). \tag{5.31}$$

After whitening, the least squares problem in the standard form

$$\hat{\boldsymbol{\delta}} = \underset{\boldsymbol{\delta}}{\operatorname{argmin}} \|\mathbf{A}\boldsymbol{\delta} - \mathbf{b}\|_2^2. \tag{5.32}$$

The resulting Jacobian $\mathbf{A}$ is large but with a sparse block structure which depends on the structure of the estimation problem. There are a number of strategies to solve such systems (see Triggs et al., 1999; Dellaert et al., 2017; Barfoot, 2017) that lie outside the scope of this work. Our framework is designed to use an external optimizer such as Ceres Solver, which can solve NLS problems of the form

$$\underset{\mathbf{x}}{\operatorname{argmin}} \sum_i \|\mathbf{f}_i(\mathbf{x}_i)\|, \tag{5.33}$$

where $f_i(\mathbf{x}_i)$ is a *cost function*[1] returning a residual vector. This solver performs linearization (5.24), but does not perform whitening nor calculation of prediction error. Thus, our implementation's task is to:

- Represent an estimation problems as a factor graph.
- Provide factors with differentiable nonlinear measurement functions $\mathbf{h}_i$ and measurements $\mathbf{y}_i$.
- Calculate whitened Jacobians (5.30) and residuals (5.31)
- Provide an interface to the solver to obtain these Jacobians and residuals at chosen linearization points.

In the next section, we present our implementation and demonstrate optimization of an example batch estimation problem.

[1]In this work, *cost function* refers to a function whose squared norm is a single term in the objective function. This definition is used by Ceres, but conflicts with Barfoot (2017).

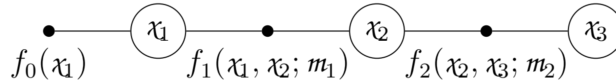


Figure 5.1: Simple factor graph for robot localization. Adapted from Dellaert (2012).

5.2 Library implementation

Estimation problems formulated as a sum over factors (5.23) can be formulated as a factor graph. A factor graph is a bipartite graph with two kinds of nodes: *variables*, which are the unknown states we wish to estimate, and *factors*, which represent probabilistic relationships between variables (Dellaert et al., 2017). Edges connect each factor to an arbitrary number of variables, and each variable to an arbitrary number of factors.

We implement a factor graph framework similar to that provided by GTSAM, which is described in Dellaert (2012). The main difference is that our framework is designed to use a third-party solver such as Ceres Solver in the back end, and uses cost functions defined as `wave_geometry` expressions.

Figure 5.1 shows a simple factor graph used as an example by Dellaert (2012). This graph models robot motion as a Markov chain, with three variables x_1, x_2, x_3 representing robot poses at different timesteps. There is one unary factor which encodes a prior on x_1, and two binary factors connecting the poses with measurements m_1 and m_2. We can recreate this example with `wave_geometry` as shown in Listing 5.1. For simplicity, we replace the 2D poses of the original example with points, and use measurements of the translation between them.

Figure 5.2 shows a more complicated localization problem optimized using our factor graph framework. In a synthetic dataset, a simulated robot flies in a constant-speed spiral path on a sphere. 256 poses are connected by noisy motion constraints, and have noisy range measurements to known landmarks. The problem is formulated using our framework and automatically differentiated measurement functions, and solved using Ceres Solver.

5.2.1 Design

The example in Listing 5.1 shows our interface is similar to GTSAM, but with minor differences. The greatest difference is that we build the factor graph using variables explicitly provided by the user, and use those variables to hold the solution. For example, the graph in Fig. 5.1 can be evaluated simply by calling:

Listing 5.1: Creation of the factor graph in Fig. 5.1

```
// Prepare variables for each pose. Initial values are initialized to zero by default.
auto x1 = std::make_shared<FactorVariable<Pointd>>();
auto x2 = std::make_shared<FactorVariable<Pointd>>();
auto x3 = std::make_shared<FactorVariable<Pointd>>();

// Add a Gaussian prior on x1
auto prior_mean = Pointd{0.0, 0.0, 0.0};
auto prior_noise= DiagonalNoise<Pointd>::FromStdDev(0.3, 0.3, 0.3);
graph.addPrior(prior_mean, prior_noise, x1);

// Add two difference factors (using the same measurement for both)
auto meas_mean = Translationd{2.0, 0.0, 0.0};
auto meas_noise = DiagonalNoise<Translationd>{0.2, 0.2, 0.1};
auto meas = Uncertain<Translationd, DiagonalNoise>{meas_mean, meas_noise};
graph.addFactor<factors::Difference>(meas, x1, x2);
graph.addFactor<factors::Difference>(meas, x2, x3);
```

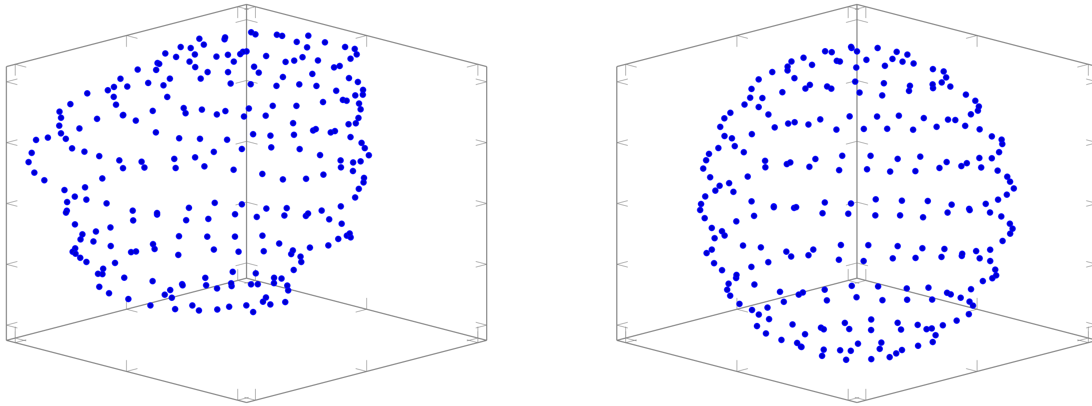


Figure 5.2: Example of localization problem using our factor graph framework. 256 poses are initialized from noisy motion constraints (left) and optimized as a batch NLS problem using motion constraints and noisy range measurements to known landmarks to produce improved estimates (right).

```
graph.evaluate()
```

After evaluation, `x1`, `x2`, and `x3` will contain the estimated values. GTSAM takes a functional approach to representing graphs, and requires values to be passed in separately. Accordingly, its factors are connected to not specific C++ variables, as in our example, but to numeric indices associated with each variable. Keeping track of the variables, their assigned indices, and their presence in the graph requires additional bookkeeping to be done by the user. While GTSAM's design has advantages, our more straightforward approach is intended to reduce this unnecessary coupling between the graph and separately stored values.

Noise models

Our `Uncertain` class template represents a value with associated noise. Like GTSAM, we offer different noise models: For example, `DiagonalNoise` represents a diagonal covariance matrix, and `FullNoise` a dense covariance matrix.

The case of zero noise—when perfect knowledge of a variable is available—requires special handling, because it implies infinite information and is not compatible with the whitening approach of equation (5.31). For this case we provide the method `addPerfectPrior`, which excludes a variable from optimization by setting its parameter block constant in the optimizer. Such "perfect" priors are used, for example, to constrain the first pose of a robot trajectory to the origin.

Cost functions

`wave_geometry` expressions can be used to define the measurement functions used by factors. For example, the `Distance` function is defined as the following functor:

Listing 5.2: `wave_geometry` expression used in a cost functor

```cpp
struct Distance {
    template <typename T, typename U>
    auto operator()(const TransformBase<T> &a,
                    const TransformBase<U> &b) const {
        return (a.derived() - b.derived()).norm();
    }
};
```

This functor returns a differentiable `wave_geometry` expression of its inputs, and is used by our framework to evaluate both the cost and Jacobian associated with a factor.

Ceres integration

Our factor graph framework includes an interface to Ceres as the back-end optimizer.[2] For every added factor, this interface adds parameter blocks to the internal `ceres::Problem`, and generates a `ceres::SizedCostFunction` for each factor.

There are several challenges to achieving integration with Ceres. First, Ceres handles parameter blocks as raw pointers to arrays of `double`, and passes these pointers back to the user-provided cost function. This system is essentially a form type erasure. Our solution is an automatically generated cost function which wraps the user-provided cost functor, and wraps the Ceres-provided raw arrays with strongly typed `wave_geometry` objects. Here, the ability of `wave_geometry` objects to use maps as storage is essential. Thus, although the functor in Listing 5.2 must templated, it can use `wave_geometry` types with the usual interface.

Second, Ceres requires global Jacobians, as described by Sommer et al. (2013). Therefore, our automatically generated cost function must "lift" the local Jacobians to the dimensions of the parameter block. We also generate a `ceres::LocalParameterization` for each manifold object, which allows the solver to perform $\boxplus$ operations on overparameterized variables.

The template-based mechanism that generates the interface to Ceres while hiding the implementation details from the user is the main contribution of our factor graph framework. This framework also demonstrates an application of AD-enabled geometric expressions for state estimation.

[2] Ceres integration is not part of the core `wave_geometry` library. It is planned for inclusion in the wider libwave collection of libraries (`https://github.com/wavelab/libwave`).

Chapter 6

Conclusion

Robotics and computer vision researchers rely on software libraries to develop and demonstrate their algorithms. This thesis asks: how can these libraries be improved? In this work, we discuss a range of topics related to representing and computing geometric relationships for the purpose of state estimation. Our main contributions are embodied in `wave_geometry`, a C++ library for manifold geometry.

We explore fundamental geometric objects and the spaces they inhabit, and discuss how they can be rigorously but conveniently expressed in code. While geometry is often performed using software for linear algebra, a number of modern libraries provide representations and operations on manifolds. We compare commonly used libraries, and consider how their geometric representations can be made more powerful. In Chapter 3, we apply expression templates to geometric objects, allowing improved performance and extended functionality. We also present a concept-based system of representing geometric types, allowing different representations of the same geometric object to be used interchangeably with automatic conversions where needed.

The use of ETs allows an efficient implementation of block automatic differentiation, also presented in Chapter 3. AD can save time and prevent mistakes in deriving and coding Jacobians of geometric functions. Arbitrary `wave_geometry` expressions can be differentiated in forward or reverse mode, with optimizations taking advantage of compile-time type information. We show that our AD system approaches, and in some cases matches, the speed of hand-coded derivatives, and exceeds the speed of AD in the widely used Ceres and GTSAM libraries.

The downside of our ET-based approach to AD is that expressions are static: their structure must be known at compile time. To lift this restriction, we provide dynamic ex-

pressions which can be composed at runtime. These dynamic expressions are an application of type erasure, and can hold arbitrary static expressions. They provide roughly equivalent functionality to the dynamically composed expressions available in GTSAM, and are also differentiable. While they are an order of magnitude slower than our static expressions, a comparison shows slightly better differentiation performance than GTSAM.

Another common source of frustration and mistakes in the development of robotics algorithms is coordinate frames conversions. In Chapter 4, we discuss the coordinate frame semantics associated with geometric objects and how they can be used to check the validity of expressions. These frame semantics can be tracked in code, and previous works have introduced software checking them at runtime. We present a template-based method, integrated into `wave_geometry`, which checks frame semantics at compile time and without runtime overhead. We also present a set of rules for propagating coordinate frame semantics through elementary geometric operations, including manifold operations.

Finally, we show how `wave_geometry` expressions can be applied to state estimation as a nonlinear least squares optimization problem. Estimation problems involve optimizing over possible states, constrained by multiple measurements and priors, and are often formulated as a factor graph. Optimization algorithms work with linearized versions of measurement models, and require Jacobians. We present a framework which combines `wave_geometry`'s expressions and AD system with the Ceres optimizer, allowing rapid development of automatically differentiated factor graphs.

In summary, the main contributions of this thesis are:

- A new C++ library providing geometric operations on affine, Euclidean, and projective spaces, and on manifolds.
- The application of expression templates to geometric expressions, and an extensible framework for defining geometric spaces, representations, and operations.
- A block automatic differentiation system allowing forward and reverse mode AD of geometric expressions. This system achieves high performance through optimizations leveraging compile-time type information, and also extends to expressions composed at runtime.
- A method for propagating and checking coordinate frame semantics at compile time in C++. This method relies on a system of rules for coordinate frame semantics of manifold operations that we present.

- A framework for integrating `wave_geometry` automatically differentiated expressions with Ceres Solver.

The library presented in this thesis is available at github.com/wavelab/wave_geometry. As it continues to evolve, we hope that it will contribute to the development of efficient and correct algorithms for robotics, and to the representation of geometry in software in general.

References

Abrahams, D. and Gurtovoy, A. (2004). *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, Boston.

Agarwal, S., Mierle, K., et al. (2010). Ceres Solver. http://ceres-solver.org.

Andersson, J., Åkesson, J., and Diehl, M. (2012). CasADi: A symbolic package for automatic differentiation and optimal control. In *Recent Advances in Algorithmic Differentiation*, pages 297–307. Springer.

Aubert, P., Di Césaré, N., and Pironneau, O. (2001). Automatic differentiation in C++ using expression templates and application to a flow control problem. *Computing and Visualization in Science*, 3(4):197–208.

Barfoot, T. D. (2017). *State Estimation for Robotics*. Cambridge University Press.

Barton, J. J. and Nackman, L. R. (1994). *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley.

Bauchau, O. A. and Trainelli, L. (2003). The vectorial parameterization of rotation. *Nonlinear Dynamics*, 32(1):71–92.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43.

Bloesch, M. et al. (2016). A primer on the differential calculus of 3d orientations. *arXiv preprint arXiv:1606.05285*.

Bremer, H. (2008). *Elastic Multibody Dynamics*. Springer.

Carpenter, B., Hoffman, M. D., Brubaker, M., Lee, D., Li, P., and Betancourt, M. (2015). The Stan math library: Reverse-mode automatic differentiation in C++. *arXiv preprint arXiv:1509.07164*.

Chirikjian, G. S. (2009). *Stochastic Models, Information Theory, and Lie Groups, Volume 1: Classical Results and Geometric Methods. Applied and Numerical Harmonic Analysis*. Birkhäuser, Boston.

Chirikjian, G. S. (2011). *Stochastic Models, Information Theory, and Lie Groups, Volume 2: Analytic Methods and Modern Applications*, volume 2. Birkhäuser, Boston.

Chirikjian, G. S., Mahony, R., Ruan, S., and Trumpf, J. (2018). Pose changes from a different point of view. *Journal of Mechanisms and Robotics*, 10(2):021008–021008–12.

Dai, J. S. (2015). Euler–rodrigues formula variations, quaternion conjugation and intrinsic connections. *Mechanism and Machine Theory*, 92:144–152.

De Laet, T., Bellens, S., Bruyninckx, H., and De Schutter, J. (2013a). Geometric relations between rigid bodies (part 2): From semantics to software. *IEEE Robotics and Automation Magazine*, 20(2):91–102.

De Laet, T., Bellens, S., Smits, R., Aertbelien, E., Bruyninckx, H., and De Schutter, J. (2013b). Geometric relations between rigid bodies (part 1): Semantics for standardization. *IEEE Robotics and Automation Magazine*, 20(1):84–93.

Dellaert, F. (2012). Factor graphs and GTSAM: A hands-on introduction. Technical Report GT-RIM-CP&R-2012-002, Georgia Institute of Technology.

Dellaert, F., Kaess, M., et al. (2017). Factor graphs for robot perception. *Foundations and Trends in Robotics*, 6(1-2):1–139.

DeRose, T. D. (1989). A coordinate-free approach to geometric programming. In *Theory and practice of geometric modeling*, pages 291–305. Springer.

Falcou, J., Gottschling, P., and Sutter, H. (2017). Implicit evaluation of "auto" variables. Technical Report P0672R0, C++ Evolution Working Group. https://wg21.link/P0672.

Foote, T. (2013). tf: The transform library. In *IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6. IEEE.

Forster, C., Carlone, L., Dellaert, F., and Scaramuzza, D. (2017). On-manifold preintegration for real-time visual–inertial odometry. *IEEE Transactions on Robotics*, 33(1):1–21.

Förstner, W. and Wrobel, B. P. (2016). *Photogrammetric Computer Vision*. Springer, Cham, Switzerland.

Furgale, P. T. (2011). *Extensions to the visual odometry pipeline for the exploration of planetary surfaces*. PhD thesis, University of Toronto.

Furgale, P. T. (2014). Representing robot pose: The good, the bad, and the ugly. Presented at workshop on Lessons Learned from Building Complex Systems, *IEEE International Conference on Robotics and Automation (ICRA)*. http://paulfurgale.info/news/2014/6/9/representing-robot-pose-the-good-the-bad-and-the-ugly.

Gallier, J. (2011). *Geometric Methods and Applications: For Computer Science and Engineering*. Springer Science & Business Media, New York, second edition.

Goldman, R. (2000). The ambient spaces of computer graphics and geometric modeling. *IEEE Computer Graphics and Applications*, 20(2):76–84.

Goldman, R. (2002). On the algebraic and geometric foundations of computer graphics. *ACM Transactions on Graphics*, 21(1):52–86.

Goldman, R. N. (1985). Illicit expressions in vector algebra. *ACM Transactions on Graphics*, 4(3):223–243.

Grassia, F. S. (1998). Practical parameterization of rotations using the exponential map. *Journal of Graphics Tools*, 3(3):29–48.

Griewank, A. and Walther, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, second edition.

Guennebaud, G., Jacob, B., et al. (2010). Eigen v3. http://eigen.tuxfamily.org.

Halliday, D., Resnick, R., and Walker, J. (2011). *Fundamentals of Physics*. John Wiley & Sons, "Hoboken, New Jersey".

Härdtlein, J., Pflaum, C., Linke, A., and Wolters, C. H. (2010). Advanced expression templates programming. *Computing and Visualization in Science*, 13(2):59–68.

Hartley, R. and Zisserman, A. (2004). *Multiple View Geometry in Computer Vision*. Cambridge University Press, second edition.

Hertzberg, C., Wagner, R., Frese, U., and Schröder, L. (2013). Integrating generic sensor fusion algorithms with sound state representations through encapsulation of manifolds. *Information Fusion*, 14(1):57–77.

Hoffmann, P. H. W. (2016). A hitchhiker's guide to automatic differentiation. *Numerical Algorithms*, 72(3):775–811.

Hogan, R. J. (2014). Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Transactions on Mathematical Software*, 40(4):26:1–26:16.

Hutton, G. (1999). A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372.

Iglberger, K., Hager, G., Treibig, J., and Rüde, U. (2012). Expression templates revisited: a performance analysis of current methodologies. *SIAM Journal on Scientific Computing*, 34(2):C42–C69.

ISO (2017). *ISO/IEC 14882:2017: Programming Language C++*. International Organization for Standardization, Geneva, Switzerland. [Working draft]. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/.

Kelly, A. (2013). *Mobile Robotics: Mathematics, Models, and Methods*. Cambridge University Press.

Koppel, L. and Waslander, S. L. (2018). Manifold geometry with fast automatic derivatives and coordinate frame semantics checking in C++. In *15th Conference on Computer and Robot Vision (CRV)*. To be published. Preprint: arXiv:1805.01810.

Korn, G. A. and Korn, T. M. (2000). *Mathematical Handbook for Scientists and Engineers - Definitions, Theorems, and Formulas for Reference and Review*. Dover Publications, Mineola, New York.

Kümmerle, R., Grisetti, G., Strasdat, H., Konolige, K., and Burgard, W. (2011). g2o: A general framework for graph optimization. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3607–3613.

Lee, J. (2008). Representing rotations and orientations in geometric computing. *IEEE Computer Graphics and Applications*, 28(2):75–83.

Leutenegger, S., Lynen, S., Bosse, M., Siegwart, R., and Furgale, P. (2015). Keyframe-based visual–inertial odometry using nonlinear optimization. *International Journal of Robotics Research*, 34(3):314–334.

Maclaurin, D., Duvenaud, D., and Adams, R. P. (2015). Autograd: Effortless gradients in numpy. ICML Workshop.

Mann, S., Litke, N., and DeRose, T. (1997). A coordinate free geometry adt. Technical Report CS-97-15, University of Waterloo.

Meyers, S. (2014). *Effective Modern C++*. O'Reilly Media, Sebastopol, CA.

Nehmzow, U. (2003). *Mobile Robotics: a Practical Introduction*. Springer, London.

Nordmann, A., Hochgeschwender, N., and Wrede, S. (2014). A survey on domain-specific languages in robotics. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 195–206. Springer.

Norton, J. D. (1993). General covariance and the foundations of general relativity: eight decades of dispute. *Reports on Progress in Physics*, 56(7):791.

Paszke, A. et al. (2017). Automatic differentiation in pytorch. NIPS Workshop.

Phipps, E. and Pawlowski, R. (2012). Efficient expression templates for operator overloading-based automatic differentiation. In *Recent Advances in Algorithmic Differentiation*, pages 309–319. Springer.

Schneider, P. J. and Eberly, D. H. (2003). *Geometric Tools for Computer Graphics*. Morgan Kaufmann, "San Francisco".

Selig, J. M. (2006). Active versus passive transformations in robotics. *IEEE Robotics & Automation Magazine*, 13(1):79–84.

Shuster, M. D. (2008). The nature of the quaternion. *The Journal of the Astronautical Sciences*, 56(3):359–373.

Soechting, J. and Flanders, M. (1992). Moving in three-dimensional space: frames of reference, vectors, and coordinate systems. *Annual Review of Neuroscience*, 15(1):167–191.

Solà, J. (2016). Quaternion kinematics for the error-state kf. Technical Report IRI-TR-16-02, Institut de Robòtica i Informàtica Industrial.

Sommer, H., Gilitschenski, I., Bloesch, M., Weiss, S., Siegwart, R., and Nieto, J. (2018). Why and how to avoid the flipped quaternion multiplication. *Aerospace*, 5(3).

Sommer, H., Pradalier, C., and Furgale, P. (2013). Automatic differentiation on differentiable manifolds as a tool for robotics. In *16th Int. Symp. Robot. Res. (ISRR)*.

Spring, K. W. (1986). Euler parameters and the use of quaternion algebra in the manipulation of finite rotations: a review. *Mechanism and machine theory*, 21(5):365–373.

Stillwell, J. (2008). *Naive Lie Theory*. Springer, New York.

Stoer, J. and Bulirsch, R. (2002). *Introduction to Numerical Analysis*. Springer Science & Business Media, New York, second edition.

Torgersen, M. (2004). The expression problem revisited. In *European Conference on Object-Oriented Programming*, pages 123–146. Springer.

Triggs, B., McLauchlan, P. F., Hartley, R. I., and Fitzgibbon, A. W. (1999). Bundle adjustment—a modern synthesis. In *International workshop on vision algorithms*, pages 298–372. Springer.

van Merriënboer, B., Wiltschko, A. B., and Moldovan, D. (2017). Tangent: Automatic differentiation using source code transformation in python. *arXiv preprint arXiv:1711.02712*.

Vandevoorde, D. and Josuttis, N. M. (2002). *C++ Templates: The Complete Guide*. Addison-Wesley.

Verma, A. (2000). An introduction to automatic differentiation. *Current Science*, 78(7):804–807.

Walther, A. and Griewank, A. (2012). Getting started with ADOL-C. In *Combinatorial Scientific Computing*, pages 181–202. CRC Press.

APPENDICES

Appendix A

Mathematical background

A.1 Lie group operations

A.1.1 Exponential map onto SO(3)

Let $\boldsymbol{\phi} \in \mathbb{R}^3$ be a rotation vector. The exponential map of $\boldsymbol{\phi}^\wedge \in \mathfrak{so}(3)$ is the matrix exponential

$$\exp(\boldsymbol{\phi}^\wedge) = \sum_{p=0}^{\infty} \frac{1}{p!}(\boldsymbol{\phi}^\wedge)^p. \tag{A.1}$$

As outlined in (Gallier, 2011, Section 8.10), it can be shown that $(\boldsymbol{\phi}^\wedge)^3 = -\theta^2\boldsymbol{\phi}^\wedge$, where $\theta = ||\boldsymbol{\phi}||$. Substituting,

$$\exp(\boldsymbol{\phi}^\wedge) = \mathbf{I}_3 + \frac{\boldsymbol{\phi}^\wedge}{1!} + \frac{(\boldsymbol{\phi}^\wedge)^2}{2!} - \frac{\theta^2\boldsymbol{\phi}^\wedge}{3!} - \frac{\theta^2(\boldsymbol{\phi}^\wedge)^2}{4!} + \frac{\theta^4\boldsymbol{\phi}^\wedge}{5!} + \dots \tag{A.2}$$

Separating the odd and even terms,

$$\exp(\boldsymbol{\phi}^\wedge) = \mathbf{I}_3 + \left(1 - \frac{\theta^2}{3!} + \frac{\theta^4}{5!} + \dots\right)\boldsymbol{\phi}^\wedge + \left(\frac{1}{2!} - \frac{\theta^2}{4!} + \frac{\theta^4}{6!} + \dots\right)(\boldsymbol{\phi}^\wedge)^2. \tag{A.3}$$

Using the series $\sin\theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} \dots$ and $\cos\theta = 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} \dots$ we obtain *Rodrigues'*

formula,

$$\exp(\boldsymbol{\phi}^\wedge) = \mathbf{I}_3 + \frac{\sin\theta}{\theta}\boldsymbol{\phi}^\wedge + \frac{1-\cos\theta}{\theta^2}(\boldsymbol{\phi}^\wedge)^2 \tag{A.4}$$

When θ is small, the term $1-\cos\theta$ amplifies numerical error due to *cancellation* (Stoer and Bulirsch, 2002). Therefore we replace $1-\cos\theta \equiv 2\sin^2(\frac{1}{2}\theta)$ to obtain

$$\boxed{\exp(\boldsymbol{\phi}^\wedge) = \mathbf{I}_3 + \frac{\sin\theta}{\theta}\boldsymbol{\phi}^\wedge + \frac{2\sin^2(\frac{1}{2}\theta)}{\theta^2}(\boldsymbol{\phi}^\wedge)^2}. \tag{A.5}$$

For $\theta^2 \le \epsilon$, where ϵ is machine precision, we use only the linear terms of the expansion (A.3):

$$\boxed{\exp(\boldsymbol{\phi}^\wedge) \approx \mathbf{I}_3 + \boldsymbol{\phi}^\wedge}. \tag{A.6}$$

A geometric derivation of the formula (not using the matrix exponential) and discussion of its variants can be found in Dai (2015).

Exponential map onto quaternions

While Rodrigues' formula is a compact way of describing the exponential map, it may not be the best way to compute it. Grassia (1998) proposes instead mapping onto S^3, then converting that quaternion into a matrix. The unit quaternion corresponding to a rotation vector is given by *Euler–Rodrigues parameters* (also called Euler parameters) (Spring, 1986; Bauchau and Trainelli, 2003)

$$\begin{aligned}\mathbf{q}(\boldsymbol{\phi}) &= \begin{bmatrix}\sin\left(\frac{1}{2}\theta\right)\hat{\boldsymbol{\phi}} & \cos\left(\frac{1}{2}\theta\right)\end{bmatrix}^\mathsf{T} \\ &\triangleq \begin{bmatrix}\vec{\mathbf{q}} & s\end{bmatrix}^\mathsf{T} \triangleq \begin{bmatrix}x & y & z & w\end{bmatrix}^\mathsf{T},\end{aligned} \tag{A.7}$$

where $\hat{\boldsymbol{\phi}}$ is the unit vector $\hat{\boldsymbol{\phi}} = \boldsymbol{\phi}/\theta$, or the zero vector if $\theta = 0$. This equation is the exponential map $\exp : \mathbb{R}^3 \to S^3$. For numerical stability, it is computed in the form (Grassia, 1998)

$$\boxed{\mathbf{q}(\boldsymbol{\phi}) = \begin{bmatrix}\frac{\sin\left(\frac{1}{2}\theta\right)}{\theta}\boldsymbol{\phi} & \cos\left(\frac{1}{2}\theta\right)\end{bmatrix}^\mathsf{T}}. \tag{A.8}$$

Again, we use a Taylor expansion for small θ. Grassia (1998) suggests the first two terms,

$$\frac{\sin\left(\frac{1}{2}\theta\right)}{\theta} \approx \frac{1}{2} + \frac{\theta^2}{48}, \tag{A.9}$$

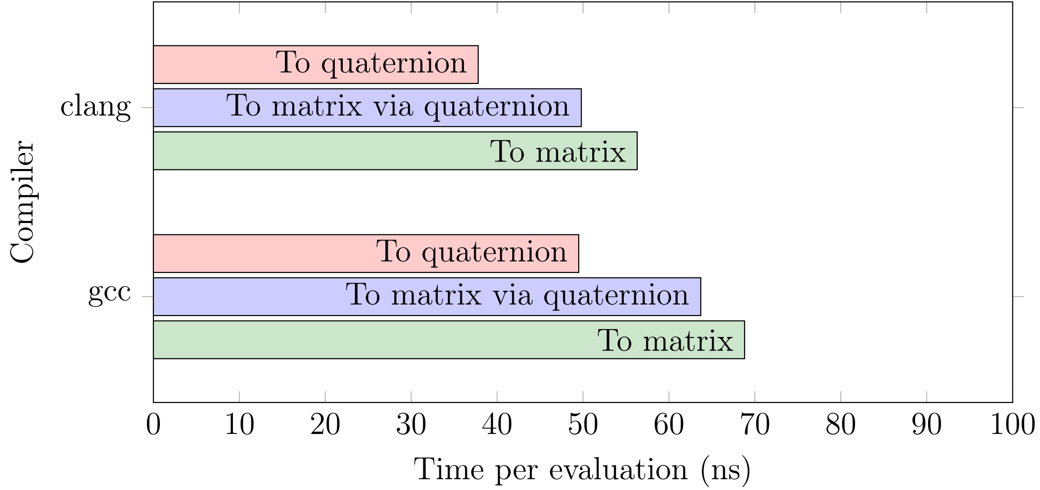


Figure A.1: Performance of exponential map functions, comparing the approaches: map to quaternion (A.8); map to quaternion, then convert to matrix (A.12); and map to matrix (A.5). The input is a list of random double-precision rotation vectors with norm 0.1. This comparison shows mapping to a quaternion is at least as fast when using either common compiler.

if $\theta \leq \sqrt[4]{\epsilon}$. Thus for small θ, we use

$$\boxed{\mathbf{q}(\boldsymbol{\phi}) \approx \left[\left(\frac{1}{2} + \frac{\theta^2}{48}\right)\boldsymbol{\phi} \quad 1 - \frac{\theta^2}{8}\right]^{\mathsf{T}}}. \tag{A.10}$$

The quaternion can then be converted to a matrix using (Spring, 1986)

$$\mathbf{C}(\mathbf{q}) = (s^2 - \vec{\mathbf{q}}^{\mathsf{T}}\vec{\mathbf{q}})\mathbf{I}_3 + 2\vec{\mathbf{q}}\vec{\mathbf{q}}^{\mathsf{T}} + 2s\vec{\mathbf{q}}^{\times} \tag{A.11}$$

$$= \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2wz & 2xz + 2wy \\ 2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx \\ 2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 \end{bmatrix}. \tag{A.12}$$

We run a microbenchmark to demonstrate that code mapping to a matrix via the map (A.8) onto S^3 is, in practice, slightly faster than code implementing the direct map (A.5). Figure A.1 presents the results. Methodology is described in Appendix D. Source code is available at `https://github.com/wavelab/wave_geometry`.

Note that we show results only for rotation vectors in the "large"-angle regime ($\theta = 0.1$);

it can be expected that Taylor series approximations are faster for either method. A linear approximation of (A.8) is given in Appendix A.1.3.

Of course, these results merely show that of two C++ functions written with Eigen, the quaternion version is more amenable to compiler optimization on our test system. A carefully tuned routine written in assembly might outperform both. However, these results support Grassia's claims and are relevant to a library choosing a single C++ implementation of the exponential map.

A.1.2 Logarithmic map of SO(3)

Logarithmic map of quaternions

The conversion from quaternion to rotation vector can be obtained by inverting the exponential map (A.8), starting with the angle θ:

$$\tan\frac{\theta}{2} = \frac{\sin\left(\frac{1}{2}\theta\right)}{\cos\left(\frac{1}{2}\theta\right)} = \frac{\|\vec{\mathbf{q}}\|}{s} \tag{A.13}$$

$$\theta(\mathbf{q}) = 2\,\mathrm{atan}\,\frac{\|\vec{\mathbf{q}}\|}{s}, \tag{A.14}$$

where we use tan instead of cos for the superior numerical properties of `atan2`. We also ensure that θ is in the range $[-\pi, \pi]$:

$$\theta(\mathbf{q}) = 2\,\mathrm{sgn}(s)\,\mathrm{atan2}(\|\vec{\mathbf{q}}\|, |s|). \tag{A.15}$$

The rotation vector is then:

$$\boxed{\boldsymbol{\phi}(\mathbf{q}) = 2\,\mathrm{sgn}(s)\,\mathrm{atan2}(\|\vec{\mathbf{q}}\|, |s|)\frac{\vec{\mathbf{q}}}{\|\vec{\mathbf{q}}\|}}. \tag{A.16}$$

For small angles ($\|\vec{\mathbf{q}}\| \approx 0, s \approx 1$), we use the first-order approximation

$$\boldsymbol{\phi}(\mathbf{q}) \approx 2\vec{\mathbf{q}}. \tag{A.17}$$

A.1.3 Approximation of the exponential map

Instead of calculating the full exponential map (A.8), it is possible to use a linear approximation. Furgale (2011) suggests the linearization

$$\boldsymbol{x}^s \boxplus \mathbf{v} \approx \left(\mathbf{q}(\mathbf{0}) + \left.\frac{\partial \mathbf{q}(\mathbf{v})}{\partial \mathbf{v}}\right|_{\mathbf{v}=\mathbf{0}} \mathbf{v} \right) \circ \boldsymbol{x}^s, \tag{A.18}$$

where

$$\mathbf{q}(\mathbf{0}) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \left.\frac{\partial \mathbf{q}(\mathbf{v})}{\partial \mathbf{v}}\right|_{\mathbf{v}=\mathbf{0}} = \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 \end{bmatrix}. \tag{A.19}$$

Approximations will be added in a future version of `wave_geometry`.

Appendix B

Frames, coordinates, systems: a few words

What is a "frame of reference", a "coordinate system", and a "coordinate frame"? Although often used interchangably, these terms have distinct meanings.

Terminology related to frames is inconsistent in literature. Hartley and Zisserman (2004) uses terms such as "world coordinate frame" and "world coordinate system" seemingly interchangeably. Barfoot (2017) and Chirikjian et al. (2018) use "reference frames" in the same sense that Furgale (2011) uses "coordinate frames." Nehmzow (2003) conflates terms while defining a "navigational frame of reference":

> In the simplest and most easily understood case, this frame of reference is a Cartesian co-ordinate system.

Robotics papers whose use of frames is incidental, rather than definitional (for example, Leutenegger et al., 2015; Forster et al., 2017) tend to use "coordinate frame", "reference frame," and "frame of reference" interchangeably. These works do not necessarily suffer from lack of clarity, since their meaning can be inferred from context.

On the other hand, "reference" is sometimes used as an adjective to distinguish one particular frame from others, if only temporarily. Bremer (2008) often refers to "*the* reference frame" (emphasis added) of a given system. Lee (2008), which uses all of the combinations "coordinate frame," "coordinate system," "reference frame," "reference coordinate frame," and "reference coordinate system," appears to do the same.

Still, there is a difference between frames of reference and coordinate systems. The idea of frames of reference is central to physics, of which Norton (1993) writes:

> In traditional developments of special and general relativity it has been customary not to distinguish between two quite distinct ideas. The first is the notion of a coordinate system, understood simply as the smooth, invertible assignment of four numbers to events in spacetime neighbourhoods. The second, the frame of reference, refers to an idealized physical system used to assign such numbers.

This distinction has consensus across fields. The introductory physics textbook of Halliday et al. (2011) states, "for our purposes, a reference frame is the physical object to which we attach our coordinate system." Soechting and Flanders (1992), in a summary for neuroscientists, describes a "frame of reference" as fixed to a physical object, such as a person or a train, and a "coordinate system" as something defined "within the frame of reference by choosing a set of base vectors," and gives examples of Cartesian and spherical coordinate systems.

These three works do not use the term "coordinate frame" (except for one stray appearance in Norton). However, "coordinate frame" is widely used, as in Hartley and Zisserman (2004); Bremer (2008); De Laet et al. (2013b); Foote (2013), to describe a construct—often tied to a physical object, e.g. a "camera coordinate frame"—in which quantities are expressed numerically.

Kelly (2013) gives the most definitive distinction between the three terms as used in robotics. Of coordinate systems and reference frames, Kelly writes,

> ...these are not only not the same thing; in fact, they have nothing to do with each other.

Specifically, "coordinate systems are conventions for the representation of physical quantities", while "a reference frame is a state of motion, and it is convenient to associate it with a real physical body with such a state of motion."

Kelly defines "conceptual embedded sets of axes," fixed to physical bodies such as robot sensors and manipulators, as *coordinate frames*. Because they are associated with a physical object, these axes have a state of motion; simultaneously, they form the basis of a coordinate system. Therefore, a coordinate frame "possesses the properties of both a frame of reference and a coordinate system."

Coordinates themselves can be defined as "an ordered set of numbers" used to represent a point in some space (Korn and Korn, 2000).

We can form the following summary: A *frame of reference* is a state of motion from which measurements can be made. Typically, it is tied to the motion of a physical object.

A *reference frame* is synonymous, although "reference" is sometimes used as an identifier of a particular frame or coordinate system. A *coordinate system* associates coordinates with points in a mathematical space; this space is often, but not necessarily, chosen to model a physical system. A *coordinate frame* is a coordinate system fixed to a frame of reference.

Appendix C

C++ library details

This appendix discusses several technical implementation details of `wave_geometry`. It assumes familiarity with C++ language topics such as value categories and object lifetimes.

C.1 Storing temporary expressions

Expression template libraries face a new challenge since the introduction of the `auto` type specifier. Meyers (2014, Item 6) describes the problem, summarizing: "As a general rule, 'invisible' proxy classes don't play well with auto. Objects of such classes are often not designed to live longer than a single statement, so creating variables of those types tends to violate fundamental library design assumptions." Eigen warns against using the `auto` keyword without care, and provides several examples of problems.[1]

C.1.1 Problems of using `auto` with expressions

Adapting Eigen's examples to `wave_geometry` types, potential categories of problems are:

1. **Redundant evaluation:** Consider an automatically-deduced expression `C`, which is not a `RotationMd` object but a `Compose<RotationMd, RotationMd>` expression holding references to rotation matrices `A` and `B`. The expression is then used repeatedly:

[1] https://eigen.tuxfamily.org/dox/TopicPitfalls.html

```
RotationMd A, B;
auto C = A * B;
for (...) { Translationd w = C * v; ... }
```

The product `C = A * B` will be needlessly evaluated on every loop iteration, unless the compiler manages to optimize it out.

2. **Change in value:** If the values of `A` and `B` change between iterations in the above example, the evaluated value of `C` will also change. In some cases this may be desired; in others, the user may have assumed `C`'s value is fixed.

3. **Undefined behaviour:** Consider explicilty evaluating a subexpression while building a larger expression,

   ```
   auto C = inverse((A+B).eval());
   ```

   or constructing a temporary object while building an expression:

   ```
   auto C = A + RelativeRotationd{0., 0., 0.1};
   ```

   In both cases, the code will compile but will produce undefined behaviour when `C` is later used. The problem is that expression objects store references to other objects even when, as above, the referents are temporary objects whose lifetime ends with the line of code shown. The resulting reference is commonly known as a *dangling reference* or *dangling pointer*, and its use falls under "Expired Pointer Dereference" in the Common Weakness Enumeration.[2]

A proposal to allow classes to declare the deduced type of `auto` variables (Falcou et al., 2017)[3] may prevent these problems in a future language standard; until then, avoidance relies on the user's vigilance. This proposal also does not significantly help `wave_geometry` since many of our *intended* use cases rely on the expression remaining alive as a non-temporary proxy object.

Problems of the third category, causing undefined behaviour, are what Meyers (2014) warns about and are the most dangerous. In our experience, dangling references are one of the most insidious types of errors in C++, often hard to identify even with tools such as AddressSanitizer and Valgrind. However, it is possible for a library author to prevent many such errors by ensuring all expressions *are* "designed to live longer than a single statement."

[2] https://cwe.mitre.org/data/definitions/825.html
[3] https://wg21.link/P0672

C.1.2 Storing temporary objects in expressions

Our solution to the third category of problems above is to store temporary objects inside expression objects by value, instead of by reference. The solution is done at library level and does not require user knowledge. It consists of two parts: storing by value in expression objects and specializing functions for rvalue reference arguments.

Store-by-value in expression objects

wave_geometry's storage type selector is an extension of Eigen's ref_selector, a template metaprogram which selects the storage type for the operands of an expression. In Eigen, lightweight expression objects, which are cheap to copy, are saved by value, while objects with their own storage, such as matrices, are stored by reference.

wave_geometry takes a different approach. ETs are instantiated with const and reference qualifiers in their template parameters. An ET instantiated like Inverse<Matrix> Inverse<Matrix&&> stores a Matrix by value; an Inverse<const Matrix&> stores a matrix by const reference. Even lightweight expressions can be stored by const reference.

Specializing functions for rvalue reference arguments

The correct template parameters are chosen by the user-facing functions which return expressions. This is done by overloading the usual function and operator definitions, of the form

```
template <typename Derived>
Inverse<Derived> inverse(const RotationBase<Derived>& rhs);
```

with a function that accepts rvalue references,

```
template <typename Derived>
Inverse<arg_t<Derived>> inverse(RotationBase<Derived>&& rhs);
```

where arg_t<Derived> selects ether Derived&& or Derived, depending on whether the argument is a leaf expression. We add && for leaves only to signal to users that there is a difference from Eigen.

This step is combinatorial; for binary functions, four overloads are needed in total:

[3]Eigen does have a mechanism, the NestByValue expression, for storing by value, but it is unused (see http://eigen.tuxfamily.org/index.php?title=Working_notes_-_Expression_evaluator).

```
template <typename L, typename R>
Compose<const L&, const R&> operator*(const RotationBase<L>& lhs, const RotationBase<L
    ↪ >& rhs);
template <typename L, typename R>
Compose<arg_t<L>, const R&>, operator*(RotationBase<L>&& lhs, const RotationBase<L>&
    ↪ rhs);
template <typename L, typename R>
Compose<const L&, arg_t<R>> operator*(RotationBase<L>& lhs, RotationBase<L>&& rhs);
template <typename L, typename R>
Compose<arg_t<L>, arg_t<R>> operator*(RotationBase<L>&& lhs, RotationBase<L>&& rhs);
```

It would be possible to achieve the same effect with fewer overloads by templating the entire parameter type and using forwarding references (`template <typename T> inverse ↪ (T&& rhs)`). However, functions overloaded on forwarding references are "the greediest functions in C++" Meyers (2014, Item 26) and would require additional SFINAE-based constraints to achieve polymorphism. One reason we choose to use built-in static dispatch is that it produces more readable compiler messages.

In `wave_geometry`, the extra overloads are generated by macros: see `WAVE_OVERLOAD_ ↪ FUNCTION_FOR_RVALUES` and similar macros.

We have not yet evaluated the performance of this solution, because preventing dangling reference errors is its first priority. It is possible an improvement could be made by storing temporary values in separately allocated memory using something like `std::shared_ptr`, instead of inside the expression object itself. A further caveat is that this solution prevents only errors arising from using temporaries in expressions, as discussed in Appendix C.1.1. It cannot prevent dangling reference errors in general, such as when an expression object referencing objects on the stack is returned from a function.

C.2 Evaluating proxies

To evaluate and differentiate dynamic expressions (introduced in Section 3.3), evaluators for proxy types must go through the virtual methods of `DynamicBase`. For example, evaluation uses the `dynEvaluate` function, declared in `DynamicBase` as

```
virtual auto dynEvaluate() const -> EvalType;
```

This function's definition inside the derived `Dynamic` class constructs the appropriate `Evaluator` for its concrete type. Forward-mode AD uses the function

```
virtual auto dynJacobian(const void *target_ptr) const
  -> Eigen::Matrix<Scalar, Eigen::Dynamic, Eigen::Dynamic>;
```

Here, we can see some of the costs of dynamic expression graphs: because virtual functions cannot be templated, the Jacobian matrix must be dynamically sized. The performance penalty is twofold: first, the compiler cannot apply optimizations based on knowledge of expression types, such as eliding multiplication by identity; second, memory must be allocated for every such matrix. The overhead of many small allocations can be a significant proportion of the cost of differentiating an expression graph.

C.2.1 Optimizations

For reverse-mode AD, we apply several optimizations.

- Traverse the expression graph in advance, obtaining the address and tangent space dimension (number of columns in the Jacobian matrix) of each leaf. Note the height of the Jacobians is the dimension of the root node, which is known.
- Pre-allocate a single matrix large enough to store all Jacobians as submatrices, and create a map of leaf address to column index in the large matrix. This is a simple, specialized alternative to using a memory pool allocator.
- Provide multiple virtual functions accepting fixed-size adjoint matrices. Since (within the derived `Dynamic` expression) the width m of the adjoint matrix is already known, only several additional overloads are needed to cover most cases: for example, $1 \times m$, $2 \times m$, $3 \times m$, and $6 \times m$ adjoints.
- Store the `Evaluator` for `Dynamic` expressions inside the `Dynamic` object itself to avoid additional allocations.

Variations of the first three of optimizations are also used by GTSAM.

C.2.2 Alternatives

Virtual functions are not the only approach for attaining dynamic polymorphism. Type erasure libraries[4] replace virtual functions with their own implementations of vtables,

and offer value semantics (removing the need for a pointer-holding proxy object) and improved runtime performance. Using such a library could improve `wave_geometry`'s initial implementation.

[4] Examples of type erasure libraries are Boost.TypeErasure, Poly, and Dyno.

Appendix D

Benchmark methodology

For results in Figure 3.8 and Table 3.2, benchmarks were compiled with Clang 5.0 as specified.

For other results, benchmarks were compiled by Clang 7.0.0 (and, where specified, GCC 8.2.0) with options `-std=c++17 -O3 -DNDEBUG -march=native` on an Intel i7-8550U processor. We used the following versions of libraries under test and dependencies: Ceres 1.14, GTSAM commit `c1b14f08f`, Eigen 3.3.4, Boost 1.65.1, libstdc++ 6.0.25. GTSAM was compiled with its bundled Eigen 3.2.10 because of stability issues when using the newer version.

To reduce the effects of system load, benchmarks were given an isolated CPU: the `isolcpus`, `nohz_full`, and `rcu_nocbs` parameters were set on a tickless kernel (Linux 4.15 built with `CONFIG_NO_HZ_FULL`). Turbo boost was disabled.

We used the Google Benchmark library[1] for timing. This library times each benchmark over a large number of iterations (at least 0.5 s in duration) and divides the measurement by the number of iterations. We report the mean of 3 runs.

With this configuration, the standard deviation between runs of the same test is small (typically under 1 %), thus we do not show error bars. Note that the purpose of these benchmarks is to compare different pieces of code on the same typical system. Performance may vary on different compilers, processor architectures, hardware, or real-world invocations of the code when integrated into a larger program.

[1] https://github.com/google/benchmark