Optimal Linear Control on the $SO(2)$ Manifold Using Lie Algebras and Auto-Differentiation

Andrew Torgesen[1]

***Abstract*— Many interesting problems in robotics and control entail dealing with extensive usage of rigid body transformations in the formulation of the dynamics of systems and their corresponding controllers. Expressing these transformations adequately can pose a challenge. In particular, rotational transforms cannot be described globally in the language of vector spaces. Thus, control formulations dealing with rotational transforms often have to resort to programmatic "hacks" such as angle wrapping and quaternion normalization to maintain a feasible control strategy. Additionally, deriving the equations of motion of complex systems with rigid body transformations for control often proves to be a cumbersome process. This project briefly reviews the mathematical foundations and applications of Lie Algebras and auto-differentiation to control theory. Lie Algebras are becoming increasingly popular, particularly in applications leveraging computer vision. Auto-differentiation has proved to be a useful and efficient alternative to calculating analytical derivatives for control. These technologies are applied to the formulation and simulation of a linear quadratic regulation (LQR) control strategy on the $SO(2)$ manifold.**

I. Introduction

The fields of robotics and autonomy are characterized by their extensive use of control, perception, and estimation algorithms for systems with many degrees of freedom and coordinate frames. For example, a miniature unmanned air vehicle (UAV) platform usually has at least six degrees of freedom during flight. A UAV is commonly outfitted with an inertial measurement unit (IMU) that is giving measurements in the body-fixed frame, a camera giving measurements in the camera-centric frame, a GPS unit giving measurements in an inertial frame, etc. Moreover, the dynamic equations of motion of the UAV are often derived and expressed in successive frames characterized by Euler angles. For these reasons, a successful implementation of control, perception, and estimation strategies must be able to take into account all of the different rigid body transformations between these frames in a mathematically sound and robust way.

Expressing these rigid body transformations adequately can pose a challenge. In particular, rotational transforms cannot be expressed as members of a vector space. Instead, they are classified as members of a manifold. For example, three-dimensional rotational transforms are all members of the group of all 3-by-3 orthogonal matrices with a determinant of one, known as $SO(3)$. Similarly, all two-dimensional rotational transforms are characterized by the group of all 2-by-2 orthogonal matrices with a determinant of one, known as $SO(2)$. Because rotational transforms in two and three-dimensional space cannot be expressed as members of a vector space, the derivation of control and estimation strategies must often rely on programmatic "hacks" such as angle wrapping and quaternion normalization to keep the mathematics intact and the algorithms stable. As shown in [1], complicated mathematical tools known as coning and sculling integrals have historically been used to deal with this same rotational transform problem.

An additional challenge that arises in robotics problems of this nature is the task of deriving analytic derivatives. As the number of degrees of freedom and distinct frames increases, the complexity of the derivatives of the dynamic and measurement models also increases. Derivative complexity can also be a function of the chosen method for representing rotations, such as quaternions versus Euler angles. The process of deriving cumbersome derivatives for linearization and algorithm implementation often bogs down the development process, distracting from the overarching goal of creating effective and robust algorithms.

This paper provides a tutorial on select methods for dealing with the aforementioned challenges in the context of optimal linear control. Specifically, an optimal linear controller is derived on the $SO(2)$ manifold using Lie Algebras for a simple point mass system constrained to a circle, and auto-differentiation is used to avoid analytically taking derivatives of the nonlinear dynamics. Section II-A gives a brief overview of Lie theory and its application to robotics, and II-B applies the theory to the dynamics of a point mass on the $SO(2)$ manifold, formulated for optimal control. Section III explains the mechanics and use of auto-differentiation, and Section IV gives the background of the optimal linear controller that will be used in the tutorial. Finally, in Section V, the controller is simulated and results are discussed. The tools and methods discussed in this tutorial are equally applicable to problems of higher dimensionality and complexity.

II. Lie Algebras and Dynamics in $SO(2)$

A. Lie Groups and Lie Algebras: Background

As explained in [2], it should be noted that Lie theory is an extremely vast field of mathematics with applications well beyond the scope of this paper. This section will give the *minimum required background* of Lie theory to create an optimal linear controller in $SO(2)$. The material in this section is adapted from [2], [3], [4], and [5].

In robotics and general engineering applications, the property of linearity is an extremely important one. The mathematics for proving stability, controllability, observability, and robustness are well-defined, as are a myriad of algorithms for implementing controllers, observers, etc. when the dynamics

[1]Completed as a project for CS 513 andrew.torgesen@gmail.com

of the system in question are linear. In practice, however, dynamics of a system are almost certainly never *truly* linear. Thus, part of the goal of an engineer is often to find a (linear, if possible) system representation that represents the real system *well enough* to achieve satisfactory performance. If a linear model is not enough to achieve satisfactory performance over the entire configuration space of a system, then it is very common practice to repeatedly linearize a nonlinear model as a system's state evolves. Figure 1 depicts a common version of this process, in which a nonlinear model is re-linearized at each time step k for each new equilibrium point, $x_{eq,k}$ to allow for the use of algorithms which assume linear dynamics:

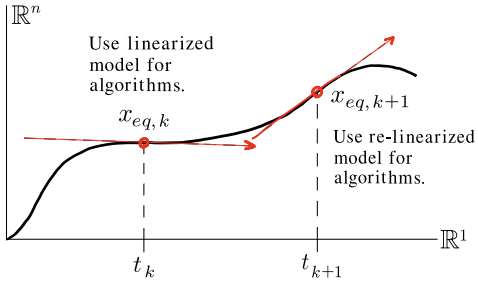

Fig. 1: Representation of repeated linearization about equilibrium point $x_{eq,k}$ for the application of specialized algorithms.

One key feature to observe from Figure 1 is the fact that the model state x, the linearization, and the algorithms all operate on and in the space $\mathbb{R}^n$, which is a vector space. Vector spaces possess special properties, such as the ability to be represented uniquely as vectors using a linear combination of basis vectors, as well as the commutative property. The special properties of vector spaces make them desirable to work with mathematically.

Rotations are *not* members of vector spaces. In other words, they cannot be represented as vectors; they cannot be uniquely expressed as linear combinations of basis vectors and they don't commute. To get around the fact that rotations cannot be represented as vectors, they are usually given parameterizations such as Euler angles (roll, pitch, yaw) or quaternions, "stuffed" into a vector, and the pseudo-vector is updated at each time step by algorithms designed to compensate for the resulting irregular properties of the pseudo-vector. In practice, this often works adequately when time steps are very small, but also requires programmatic "hacks" such as angle wrapping and repeated quaternion normalization. Using classical vector calculus methods, rotations and other entities describable not as members of vector spaces but of *manifolds* cannot be analyzed in a mathematically sound manner.

Lie groups and Lie algebras provide the mathematical tools necessary to amend these issues. Lie groups are characterized as *smooth*, *differentiable* manifolds. Smooth, differential manifolds have the special property that *locally* they behave like vector spaces. Lie algebras exploit this property. A Lie algebra is defined as *the tangent space of a corresponding Lie group at the identity*. Figure 2 visualizes this concept:

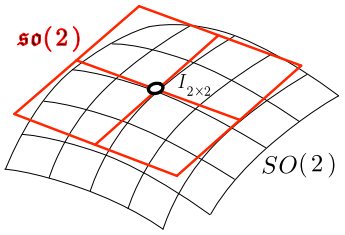

Fig. 2: Representation of the relationship between a Lie group $SO(2)$ and its Lie algebra $\mathfrak{so}(2)$. Figure adapted from [2].

In the figure, $SO(2)$ is the Lie group, and $\mathfrak{so}(2)$ is its corresponding Lie algebra. While $SO(2)$ is a manifold, $\mathfrak{so}(2)$ is a vector space. $\mathfrak{so}(2)$ is derived by computing the tangent space of $SO(2)$ at the identity element of $SO(2)$, which is the 2-by-2 identity matrix, $I_{2\times2}$. It is important to note that the Lie algebra does not have to be calculated at the identity element of the Lie group, though it is convenient to do so because every Lie group is required to have an identity element. Lie groups are any set of mathematical elements (whether they're matrices or something else) characterized by the following properties:

- The set is closed under all composition mapping operations.
- Its group operations are associative.
- The set has a unique identity element.
- Every element in the set has a unique inverse.
- The elements of the set form a differentiable manifold (meaning that we can do calculus with the elements).
- All group composition mappings are smooth and differentiable.
- The inversion mapping is differentiable.

Given a Lie group, its corresponding Lie algebra can be found using what is called a logarithmic mapping. Once the Lie algebra has been found, *all of the math can be done with the algebra since it constitutes a vector space*. That being said, although members of a Lie algebra constitute a vector space (and can thus be added, subtracted, and transformed by derivative operators), these vector spaces usually do not correspond to the Cartesian vector space $\mathbb{R}^n$, so they cannot be immediately manipulated using matrix operators. Thus, linear invertible maps (or isomorphisms) called the *hat* and *vee* operators are used to transform between the two vector spaces. The *hat* operator $(\cdot)^\wedge$ maps elements of $\mathbb{R}^n$ to elements of the vector space of the Lie algebra, and the *vee* operator $(\cdot)^\vee$ maps Lie algebra elements to vectors in $\mathbb{R}^n$. Every element of a Lie group has a corresponding element in its Lie algebra. Thus, once the needed mathematical analysis has been performed with the algebra, the algebra can be transformed back into its corresponding Lie group element using what is called the exponential mapping.

Applications of the above can be found in Section II-B, which uses the mathematics of Lie theory to derive important dynamic relationships for a particle confined to a circle (describable in $SO(2)$) in a mathematically sound way.

B. Dynamic Relations for Optimal Control in $SO(2)$

We will now apply the mathematics of Lie Algebras to the dynamics of a particle confined to a unit circle in two-dimensional space, referred to as $\mathbb{R}^2$. In $\mathbb{R}^2$, rotations are

described by the set of all orthogonal 2-by-2 matrices with a determinant of 1, or the group $SO(2)$. The formal mathematical operations utilized in this section for transforming between Lie group elements and Lie algebra elements are defined in [2], [6], and [7].

The following are the definitions of the fundamental mathematical operations relevant to manifolds as applied to a scalar quantity $p \in \mathbb{R}^1$, a manifold object $\Phi \in SO(2)$, and a Lie algebra object $\phi \in \mathfrak{so}(2)$. This section will make extensive usage of these definitions:

The hat operator, $(\cdot)^\wedge : \mathbb{R}^1 \to \mathfrak{so}(2)$:

$$p^\wedge = \begin{bmatrix} 0 & -p \\ p & 0 \end{bmatrix} = \phi. \tag{1}$$

The vee operator, $(\cdot)^\vee : \mathfrak{so}(2) \to \mathbb{R}^1$:

$$\phi^\vee = \phi(1,0) = p, \tag{2}$$

where $\phi(i,j)$ denotes the i$^{\text{th}}$ row, j$^{\text{th}}$ column entry of the matrix ϕ.

The exponential mapping, $\exp(\cdot) : \mathfrak{so}(2) \to SO(2)$:

$$\exp(\phi) = \begin{bmatrix} \cos(\phi^\vee) & -\sin(\phi^\vee) \\ \sin(\phi^\vee) & \cos(\phi^\vee) \end{bmatrix} = \Phi. \tag{3}$$

The logarithmic mapping, $\log(\cdot) : SO(2) \to \mathfrak{so}(2)$:

$$\log(\Phi) = \begin{bmatrix} 0 & -\arctan(\frac{\Phi(1,0)}{\Phi(0,0)}) \\ \arctan(\frac{\Phi(1,0)}{\Phi(0,0)}) & 0 \end{bmatrix} = \phi. \tag{4}$$

With these definitions in place, we move to modeling a simple $SO(2)$-defined system for optimal control. When modeling a system for optimal control, it is customary to modify the state x and input u by subtracting off a desired, or reference, trajectory x_{ref}, u_{ref} that maintains the system in equilibrium:

$$\tilde{x} = x - x_{ref} \qquad \tilde{u} = u - u_{ref} \tag{5}$$

Imagine a particle confined to a unit circle. Its state x at any given point in time can be described by an angle on the circle, θ, and angular velocity, ω. However, expressing the angle on the circle with a single value will require finite limits (i.e. $-\pi \to \pi$), leading to the need for angle wrapping throughout the control scheme, as well as other issues. Thus, in this work we express the state instead as the following:

$$x = \begin{bmatrix} \Phi & \omega \end{bmatrix}^T \qquad u = \Gamma \tag{6}$$

where $\Phi \in SO(2)$ is used to describe the position on the circle instead of θ. Γ is the applied torque on the particle. Because Φ is not a member of a vector space, we cannot simply perform the subtraction operation in Equation 5. Instead, we must define the *box-minus* operator $\boxminus$, which subtracts normal vector components in $\mathbb{R}^n$, but does something different for Lie group elements. The box-minus operator takes two Lie group elements and sends them to a Lie algebra element which represents their difference using the logarithmic mapping first, then uses the vee operator to turn that element into a vector in $\mathbb{R}^n$. Thus, we define the difference between x and x_{ref} as

$$\tilde{x} = x \boxminus x_{ref} = \begin{bmatrix} \log(\Phi\Phi_{ref}^{-1})^\vee \\ \omega - \omega_{ref} \end{bmatrix}, \tag{7}$$

where $\Phi\Phi_{ref}^{-1}$ gives a representation of the difference between the current and reference angular states.

The box-minus operator, as well as the exponential mapping and hat operator, will allow for the calculation of the time derivative of $\Phi \in SO(2)$. Keeping in mind that Φ is a 2×2 rotation matrix from a fixed frame $\mathcal{I}$ to a rotating frame $\mathcal{F}(t)$, the definition of the derivative is

$$\begin{aligned}
\dot{\Phi} &= \lim_{\Delta t \to 0} \frac{1}{\Delta t}(\Phi_{t+\Delta t} \boxminus \Phi_t) = \lim_{\Delta t \to 0} \frac{1}{\Delta t} \log(\Phi_{t+\Delta t}\Phi_t^{-1}) \\
&= \lim_{\Delta t \to 0} \frac{1}{\Delta t} \log(\Phi_{\mathcal{I}}^{\mathcal{F}(t+\Delta t)}\Phi_{\mathcal{F}(t)}^{\mathcal{I}}) \\
&= \lim_{\Delta t \to 0} \frac{1}{\Delta t} \log(\Phi_{\mathcal{F}(t)}^{\mathcal{F}(t+\Delta t)}\Phi_{\mathcal{I}}^{\mathcal{F}(t)}\Phi_{\mathcal{F}(t)}^{\mathcal{I}}) \\
&= \lim_{\Delta t \to 0} \frac{1}{\Delta t} \log(\exp(\delta_{\mathcal{F}(t)}^{\mathcal{F}(t+\Delta t)})\Phi_{\mathcal{I}}^{\mathcal{F}(t)}\Phi_{\mathcal{F}(t)}^{\mathcal{I}}) \\
&= \lim_{\Delta t \to 0} \frac{1}{\Delta t} \log(\exp(\delta_{\mathcal{F}(t)}^{\mathcal{F}(t+\Delta t)})) = \lim_{\Delta t \to 0} \frac{1}{\Delta t}\delta_{\mathcal{F}(t)}^{\mathcal{F}(t+\Delta t)} \\
&= \frac{d\delta}{dt} = \frac{d}{dt}\begin{bmatrix} 0 & -\theta \\ \theta & 0 \end{bmatrix} = \begin{bmatrix} 0 & -\omega \\ \omega & 0 \end{bmatrix} \in \mathfrak{so}(2).
\end{aligned} \tag{8}$$

There is something very important to note here about Equation 8. The attentive reader may notice that the above result does not quite correspond to the standard definition of the time derivative of a rotation matrix given in [8] as

$$\dot{\Phi} = \begin{bmatrix} 0 & -\omega \\ \omega & 0 \end{bmatrix} \Phi \in SO(2). \tag{9}$$

The discrepancy between Equations 8 and 9 gets back to the definition of a Lie algebra as given in Section II-A: the tangent space of a Lie group evaluated *at the identity*. This is apparent when we see that we arrive at Equation 8 by substituting the identity $SO(2)$ element, $I_{2\times 2}$, for Φ in Equation 9. It may be inferred, then, that a special form of the time derivative of a Lie group element (special in that it returns a member of the Lie algebra, which is a member of a vector space) is obtained by using the box-minus operator in the standard definition of a time derivative.

Finally, we apply the vee operator to transform our derivative result in Equation 8 from $\mathfrak{so}(2)$ to $\mathbb{R}^1$:

$$(\dot{\Phi})^\vee = \omega \in \mathbb{R}^1. \tag{10}$$

We now move forward with Equation 10 as our operating definition of the time derivative of Φ, *as it encodes the same information about the time evolution of* Φ *as Equation 9* and also has the advantage of being encoded as a member of $\mathbb{R}^1$. Indeed, ω can be transformed back into a member of $SO(2)$ at any point using the exponential mapping.

Given Equations 6 and 10, the dynamics of the system $\dot{x}$ are

$$\dot{x} = \begin{bmatrix} \dot{\Phi} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} \omega \\ -\frac{b}{J}\omega + \frac{1}{J}\Gamma \end{bmatrix} \tag{11}$$

where the $\dot{\omega}$ equation comes from the application of Newton's second law to a rotating system with angular velocity damping:

$$J\dot{\omega} + b\omega = \Gamma \tag{12}$$

with particle moment of inertia about the center of the unit circle J and damping coefficient b.

From Equation 11, we can use Euler integration to simulate the system in discrete time on a computer:

$$x(t + \Delta t) = \begin{bmatrix} \Phi(t + \Delta t) \\ \omega(t + \Delta t) \end{bmatrix} = x(t) \boxplus \dot{x}(t)\Delta t \tag{13}$$

where, as with the box-minus operator, the "box-plus" operator $\boxplus$ is a generalized addition operator that adds vector components, but takes Lie algebra objects and sends them to the manifold using the exponential mapping. Applying the definition of the box-plus operator to Equation 13 gives

$$\begin{bmatrix} \Phi(t + \Delta t) \\ \omega(t + \Delta t) \end{bmatrix} = \begin{bmatrix} \exp(\omega(t)\Delta t)\Phi(t) \\ \omega(t) + (\frac{1}{J}\Gamma - \frac{b}{J}\omega(t))\Delta t \end{bmatrix}. \tag{14}$$

These notions of generalized subtraction, addition, differentiation, and simulation provide the mathematical basis for a specialized optimal controller that operates on the $SO(2)$ manifold.

III. Auto-Differentiation

Auto-differentiation, or algorithmic differentiation, is a software tool for automatically calculating derivatives numerically in code. The tool achieves this by exploiting the fact that a derivative is a linear operator, which means that the derivative of a sequence of elementary operations (such as addition, subtraction, multiplication, etc.) is equal to the derivatives of each elementary operation in sequence, joined by the chain rule. Because every mathematical relationship in general robotics applications can be decomposed into elementary operations, auto-differentiation is an effective means for not dealing with the derivation of analytic derivatives.

A common method of applying the chain rule in auto-differentiation, known as the *forward mode*, is given in [9]. To understand the process, let us take an example function of two variables:

$$f(x_1, x_2) = x_1 x_2 + x_2 sin(x_1) \tag{15}$$

We will use the forward mode method of auto-differentiation to evaluate the partial derivative of this function with respect to x_1 (denoted $\partial f/\partial x_1$) at $x_1 = 0.5$, $x_2 = 1.5$. Equation 15 can be represented as a computational graph with edges w_n representing intermediate computed values, as shown below:

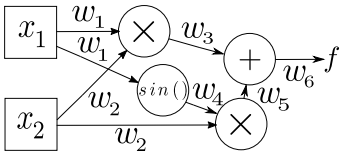

Fig. 3: Computational graph for the equation $f(x_1, x_2) = x_1 x_2 + x_2 sin(x_1)$.

At $x_1 = 0.5$, $x_2 = 1.5$, the intermediate signal values w_i take on the following values:

w_i	Value
w_1	0.5
w_2	1.5
$w_3 = w_1 * w_2$	0.75
$w_4 = sin(w_1)$	0.4794
$w_5 = w_2 * w_4$	0.7191
$w_6 = w_5 + w_3$	1.4691

We now use these intermediate signal values to calculate $\partial f/\partial x_1$, which is equal to $\partial w_6/\partial w_1$, derivable using the derivatives of the elementary operations and the chain rule:

w_i	Chain Rule	$\partial w_i/\partial w_1$
w_1	$\frac{\partial w_1}{\partial w_1}$	1
w_2	$\frac{\partial w_2}{\partial w_1}$	0
w_3	$\frac{\partial w_3}{\partial w_1}(\frac{\partial w_1}{\partial w_1}) + \frac{\partial w_3}{\partial w_2}(\frac{\partial w_2}{\partial w_1})$	$w_2(1) + w_1(0) = 1.5$
w_4	$\frac{\partial w_4}{\partial w_1}(\frac{\partial w_1}{\partial w_1})$	$cos(w_1)(1) = 0.8776$
w_5	$\frac{\partial w_5}{\partial w_2}(\frac{\partial w_2}{\partial w_1}) + \frac{\partial w_5}{\partial w_4}(\frac{\partial w_4}{\partial w_1})$	$w_4(0) + w_2(cos(w_1)) = 1.3$
w_6	$\frac{\partial w_6}{\partial w_3}(\frac{\partial w_3}{\partial w_1}) + \frac{\partial w_6}{\partial w_5}(\frac{\partial w_5}{\partial w_1})$	$1(1.5) + 1(1.3164) = 2.816$

The algorithm gives an answer of 2.816, which matches the analytic solution of $\partial f/\partial x_1 = x_2 + x_2 cos(x_1)$. As can be seen from this example, only derivatives of elementary operations and the chain rule are needed to compute derivatives of arbitrarily complex functions. Unlike finite difference methods, auto-differentiation techniques have no truncation error.

Different programming languages feature different libraries for providing automatic differentiation functionality. In Python, the library *autograd* was developed as part of a Ph.D. thesis, found in [10]. Autograd performs operator overloading on elementary operations to implement the forward mode algorithm. For an example of autograd usage, consider the following Python code, which implements the previously given example:

```
from autograd import grad
import autograd.numpy as np

def f(x1, x2):
    return x1*x1 + x2*np.sin(x1)

dfdx1 = grad(f, 0)

slope = dfdx1(0.5, 1.5)
```

The line "`import autograd.numpy as np`" imports the elementary operator overloading. The forward mode auto-differentiation is implemented in the line "`dfdx1 = grad(f, 0)`." The autograd function `grad` takes the function definition `f(x1,x2)` and creates a new function handle `dfdx1`, which is the partial derivative of `f` with respect to the argument index 0 (zero-indexed), giving $\partial f/\partial x_1$. Because autograd creates a new function handle by writing new code, the resulting auto-differentiation runs just as fast as programming the analytic derivative as a function by hand.

IV. LQR Control

We are concerned with the *optimal* control scheme for controlling a linear system. This is tantamount to finding an optimal input $u(t)$ to drive the state $x(t)$ of a linear system to some desired value while minimizing a linear objective function, usually a function of state error and control effort. As insightfully pointed out in [11], such a controller can be thought of as the $\mathcal{H}_2$-optimal controller. Rudolf Emil Kalman was the first individual to solve this problem, as described in [12]. His derived regulation scheme came to be known as the Linear Quadratic Regulator (LQR). While control problems are concerned with driving the states of a system to some commanded value, *regulation* problems are concerned with driving the states of a system to zero. Put succinctly, LQR over a finite time horizon is given in [13] as the control scheme which minimizes the continuous-time cost function

$$C = \frac{1}{2}\int_0^\infty (x^T Q x + u^T R u)dt \tag{16}$$

where $Q \geq 0$, $R > 0$ are positive semi-definite and positive definite matrices, respectively, subject to the constraints

$$\dot{x} = Ax + Bu, x \in \mathbb{R}^n, u \in \mathbb{R}^n, x_0.$$

Compare Equation 16 with the general formulation for an optimal control problem, as given in [14], which is to minimize the continuous-time cost function

$$C = \int_{t_0}^{t_f} L(x(t), u(t), t)dt \tag{17}$$

subject to the constraints

$$\dot{x} = f(x(t), u(t), t).$$

Comparing Equations 16 and 17, one may be convinced that the difference between LQR and generic optimal control is the linearity in the objective function and constraints of the former. If LQR is to be used to regulate a nonlinear system, then a linearized version of the system must be derived.

It is shown in [13] that the closed-form solution (or optimal control $u(t)$) to Equation 16 is given by

$$u(t) = -Kx(t) \tag{18}$$

where K is a m-by-n matrix of gains (m is the dimension of u and n is the dimension of x) given by

$$K = R^{-1}B^T S \tag{19}$$

and S is given by the solution to the algebraic Riccati equation:

$$-SA + A^T S + SBR^{-1}B^T S - Q = 0 \tag{20}$$

Equation 20 can be solved efficiently using modern algorithms, as discussed in [15]. In fact, the LQR design process essentially automates the process of picking optimal full-state feedback gains, given the Q and R weighting matrices in Equation 16. Thus, the task of designing a LQR given the linear (or linearized) time-invariant system $\dot{x} = Ax + Bu$ amounts to choosing Q and R. Q is a n-by-n, positive semi-definite matrix whose values weight the relative cost of errors in the control of x (x is synonymous with error since a regulator tries to drive the state to zero). For example, setting Q equal to the identity matrix equally weights error in each state x_i in the cost function. Likewise, R is a m-by-m positive definite matrix whose values weight the relative cost of the components of the control effort, or $\sum||u_i||^2$. Thus, choosing values for Q and R allows one to define the tradeoff between full-state regulation accuracy and control effort.

V. Simulation and Results

With the tools of $SO(2)$ manifold operators, auto-differentiation, and LQR, we now derive and implement an optimal controller for commanding a particle confined to a unit circle to specified angles θ_c. The effectiveness of the controller is demonstrated in simulation using Python.

A. Error State Dynamics and Controller Design

In accordance with the state formulation in Section II-B, we define the error state between the position and velocity of a particle on a unit circle x and its desired position and velocity x_d:

$$\tilde{x} = x \boxminus x_d \tag{21}$$

Taking the time derivative of Equation 21 removes the box-minus operator, since the time derivatives of the $SO(2)$ elements were shown in Section II-B to be members of $\mathbb{R}^1$:

$$\dot{\tilde{x}} = \dot{x} - \dot{x_d} = \begin{bmatrix}\dot{\tilde{\Phi}} \\ \dot{\tilde{\omega}}\end{bmatrix} = \begin{bmatrix}\omega - \omega_d \\ -\frac{b}{J}\omega + \frac{1}{J}\Gamma - (-\frac{b}{J}\omega_d + \frac{1}{J}\Gamma_d)\end{bmatrix} = \begin{bmatrix}\omega - \omega_d \\ \frac{b}{J}(\omega_d - \omega) + \frac{1}{J}(\Gamma - \Gamma_d)\end{bmatrix}. \tag{22}$$

As discussed in Section IV, the LQR control scheme requires a linear description of a system and its inputs in the form of the matrices A and B, which are linearized with respect to the states and inputs, respectively. Thus, in order

to derive A and B for our particle system, we must describe Equation 22 as a function of the error state, $\tilde{x}$. We do this by first applying the box-minus operator and exponential mapping to obtain an expression relating the current state, the desired state, and the error state:

$$\begin{bmatrix}\tilde{\Phi}\\ \tilde{\omega}\end{bmatrix} = \begin{bmatrix}\log(\Phi_d\Phi^{-1})^\vee\\ \omega-\omega_d\end{bmatrix} \tag{23}$$
$$\rightarrow \begin{bmatrix}\exp(\tilde{\Phi}^\wedge)\\ \tilde{\omega}\end{bmatrix} = \begin{bmatrix}\Phi_d\Phi^{-1}\\ \omega-\omega_d\end{bmatrix}$$
$$\rightarrow \begin{bmatrix}\exp(\tilde{\Phi}^\wedge)\Phi\\ \omega-\tilde{\omega}\end{bmatrix} = \begin{bmatrix}\Phi_d\\ \omega_d\end{bmatrix}.$$

Substituting the results of Equation 23 into Equation 22, we obtain the error state dynamics in terms of only the error state:

$$\begin{aligned}\begin{bmatrix}\dot{\tilde{\Phi}}\\ \dot{\tilde{\omega}}\end{bmatrix} &= \begin{bmatrix}\omega-(\omega-\tilde{\omega})\\ \frac{b}{J}((\omega-\tilde{\omega})-\omega)+\frac{1}{J}(\Gamma-(\Gamma-\tilde{\Gamma}))\end{bmatrix}\\ &= \begin{bmatrix}\tilde{\omega}\\ -\frac{b}{J}\tilde{\omega}+\frac{1}{J}\tilde{\Gamma}\end{bmatrix} = f(x,u).\end{aligned} \tag{24}$$

For the simulation, we choose coefficient values of $J = 1.0$ and $b = 0.1$. Given Equation 24, the state space matrices A and B are given as the Jacobians of the dynamics with respect to the error state and error input:

$$A = \frac{\partial f(x,u)}{\partial x}. \tag{25}$$

$$B = \frac{\partial f(x,u)}{\partial u}. \tag{26}$$

Though Equations 25 and 26 are, in this case, very simple to calculate analytically by hand, we will refrain from doing so in order to demonstrate the effectiveness of auto-differentiation in calculating Jacobians for control applications.

As explained in Section IV, an optimal input u (corresponding to values for Γ through time) for driving the particle to desired states x_d is given by $u = -K\tilde{x}$, where K is a gain matrix automatically calculated as a function of A, B, and cost matrices Q and R. For this simulation, we choose the following values for the cost matrices:

$$Q = \begin{bmatrix}10 & 0\\ 0 & 1\end{bmatrix} \qquad R = 0.1 \tag{27}$$

Interpreted in terms of a cost function, the chosen matrices Q and R communicate that ten times as much importance is given to removing error in Φ than removing error in ω, and ten times as much importance is given to removing error in ω than experiencing control effort in Γ. In the simulation, the optimal K matrix with respect to these design criteria is calculated using the Python function `scipy.linalg.solve_continuous_are` to solve the algebraic Riccati equation for the matrix S, then applying Equation 19:

$$\begin{aligned}K &= R^{-1}B^TS\\ &= \begin{bmatrix}10 & 5.378\end{bmatrix}.\end{aligned} \tag{28}$$

Given the optimal control gain matrix, we now simultaneously simulate and control the $SO(2)$ particle system according to Algorithm 1, which utilizes auto-differentiation, the optimal gain matrix from LQR, and manifold operators:

Algorithm 1 $SO(2)$ LQR Reference Tracker

Input: Q, R cost matrices, $\theta_d \in \mathbb{R}^1$ reference angle command
Output: $\theta(t)$, $\omega(t)$, $\Gamma(t)$

Initialize A and B matrices by algorithmically differentiating Equation 24
Initialize K (assume constant) using Equation 28
Initialize x: $\Phi(0) = I_{2\times 2} \in SO(2)$, $\omega(0) = 0$
Initialize $\Gamma_d = 0$, $\omega_d = 0$
while $t \leq 5$ **do**
  Compute Control:
  $\Phi_d = \exp(\theta_d^\wedge)$
  $\tilde{x} = \begin{bmatrix}\log(\Phi\Phi_d^{-1})^\vee\\ \omega-\omega_d\end{bmatrix}$
  $\tilde{\Gamma} = -K\tilde{x}$
  $\Gamma = \tilde{\Gamma} + \Gamma_d$
  Propagate Dynamics for Simulation:
  $\begin{bmatrix}\Phi(t+\Delta t)\\ \omega(t+\Delta t)\end{bmatrix} = \begin{bmatrix}\exp(\omega(t)\Delta t)\Phi(t)\\ \omega(t)+(\frac{1}{J}\Gamma-\frac{b}{J}\omega(t))\Delta t\end{bmatrix}$
  Store State Values:
  Store $\theta(t) = \log(\Phi)^\vee$
  Store $\omega(t) = \omega$
  Store $\Gamma(t) = \Gamma$
end while

The following section gives results of a Python simulation of Algorithm 1.

B. Results and Discussion

Figures 4 and 5 depict the results of two separate simulations of algorithm 1 in Python. Figure 4 demonstrates the ability of the controller to track reference commands in a timely manner and with zero steady-state error. It is also apparent from the figure that the LQR control scheme successfully drives the *error state* to zero. Again, it is important that the desired trajectory x_d, u_d correspond to equilibrium for the system (in this case, $\theta = \theta_d$, $\omega = \omega_d = 0$, $\Gamma = \Gamma_d = 0$).

While Figure 4 demonstrates the effectiveness of the controller and the auto-differentiation engine, it does not necessarily demonstrate the advantage of formulating the state as $x = \begin{bmatrix}\Phi & \omega\end{bmatrix}^T$ as opposed to $x = \begin{bmatrix}\theta & \omega\end{bmatrix}^T$, as would normally be customary. Indeed, a properly formulated LQR control scheme with respect to a θ-defined state would produce the exact same plot, provided that the dynamics of the system for simulation were formulated properly.

One important advantage of the manifold state representation is demonstrated in Figure 5, in which the commanded angle jumps suddenly from just below $\theta_d = \pi$, the upper limit of the angle definition, to just above $\theta_d = -\pi$, the

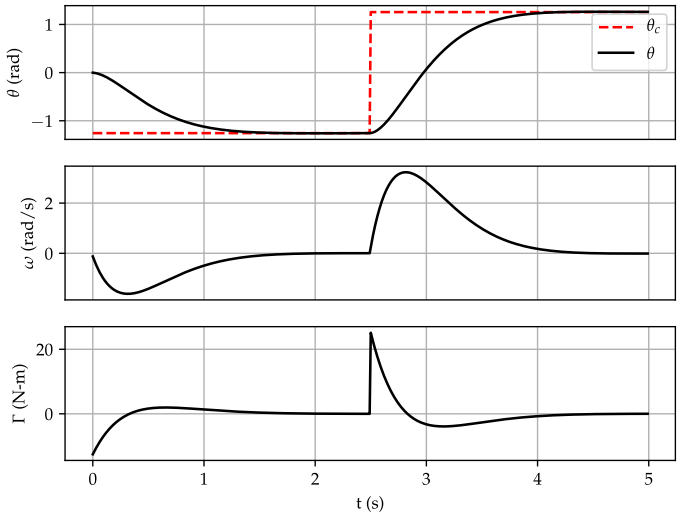

Fig. 4: Simulation of a particle confined to the unit circle, guided to reference angles θ_c by an LQR control scheme. The two commanded angles are $-2\pi/5$ and $2\pi/5$.

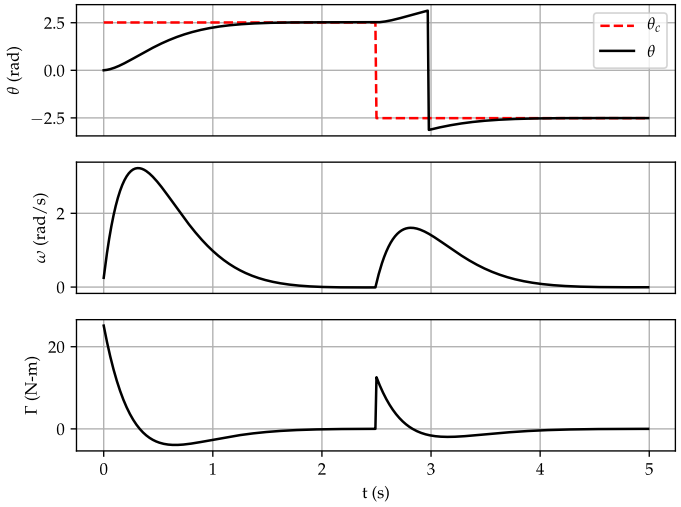

Fig. 5: Simulation of a particle confined to the unit circle, guided to reference angles θ_c by an LQR control scheme. The two commanded angles are $4\pi/5$ and $-4\pi/5$.

lower limit of the angle definition. The manifold controller successfully recognizes that a counter-clockwise path constitutes the shortest path from the first commanded angle to the second commanded angle, *despite the fact that the particle must traverse beyond the domain of the angle definition*. A traditional controller would not behave this way; a traditional controller would guide the particle the *long way around* to maintain the system state within the arbitrarily defined limits $-\pi \leq \theta \leq \pi$. Such behavior could conceivably lead to instability in the case of overshoot. Additional logic loops for angle-wrapping and related computation would be required to correct this undesirable behavior in a reliable way.

The ability of the $SO(2)$ controller to ignore arbitrarily defined angular limits points to one of the principle advantages of computing on the manifold. Namely, *computing on the manifold allows one to leverage algorithms and mathematical operations that require operands to be part of a vector space while also accurately representing quantities that are not adequately represented as vectors*.

VI. Conclusion

While the scope of this tutorial was limited to a relatively simple system in the two-dimensional plane, the applications of its subject matter are wide-ranging and impactful. The discussed tools are scalable to problems of much greater complexity. LQR is famous for having been applied widely through robotics applications with full-state feedback control, as well as for its optimality when applied to linear systems. Similarly, auto-differentiation boasts a wide array of applications, from gradient-based optimization problems of many variables to modeling applications with many state variables and complex parameterizations. Lie group theory is becoming increasingly popular in the world of robotics as the increasing scope of control and estimation problems–particularly involving computer vision–merit the use of a more sound mathematics for differential geometry. This tutorial provides a primer on these important tools with the hope that the reader may be inspired to consider their applicability to additional challenging and interesting computational problems.

References

[1] K. Roscoe, "Equivalency between strapdown inertial navigation coning and sculling integrals/algorithms," *Journal of Guidance Control and Dynamics - J GUID CONTROL DYNAM*, vol. 24, pp. 201–205, 03 2001.

[2] J. Solà, J. Deray, and D. Atchuthan, "A micro lie theory for state estimation in robotics," *CoRR*, vol. abs/1812.01537, 2018.

[3] J. M. Selig, *Lie Groups and Lie Algebras in Robotics*. South End University.

[4] D. Saracino, *Abstract Algebra: A First Course, Second Edition*. Waveland Press, 2008.

[5] Q. Xu and D. Ma, "Applications of Lie groups and Lie algebra to computer vision: A brief survey," in *2012 International Conference on Systems and Informatics (ICSAI2012)*, pp. 2024–2029, IEEE, may 2012.

[6] Z. Zhang, A. Sarlette, and Z. Ling, *Integral Control on Lie Groups*. 2015.

[7] S. Berkane and A. Tayebi, "Some optimization aspects on the lie group so(3)," *International Federation of Automatic Control*, vol. 48, no. 3, p. 11171121, 2015.

[8] S. Zhao, "Time derivative of rotation matrices: A tutorial," *CoRR*, vol. abs/1609.06088, 2016.

[9] U. Naumann, *The Art of Differentiating Computer Programs, An Introduction to Algorithmic Differentiation*. 2012.

[10] D. Maclaurin, *Modeling, Inference and Optimization with Composable Differentiable Procedures*. PhD thesis, 2016.

[11] L. Lublin, S. Grocott, and M. Athans, *H2 (LQG) and H Control*, pp. 18–1. 12 2010.

[12] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME Journal of Basic Engineering*, vol. 82, no. 1, p. 3545, 1960.

[13] R. M. Murray, "Lqr control," *Lectures on Control and Dynamical Systems at the California Institute of Technology*.

[14] J. T. Betts, "Practical methods for optimal control using nonlinear programming," *SIAM*, 2001.

[15] P. Lancaster and L. Rodman, *Algebraic Riccati equations*. Oxford University Press, 1995.