Applied 3D Rotations For Scientists and Engineers

Olabode Sule

April 21, 2024

1 Introduction

Rotations in 3-dimensions come up in various scientific and engineering applications. For example the motion of a rigid object is $3D$ can be parameterized by the specifying as a function of time a pair $(R(t), T(t))$, where $R(t)$ and $T(t)$ refer to a rotation matrix and a translation matrix respectively. In Odometry for mobile robots we are usually interested in estimating rigid body motion of the robot using data from sensors attached to the robot, among other things this helps with tasks such as localization on a map. In computer vision using cameras, we are often interested in the intrinsic and extrinsic calibration. The intrinsic calibration tell us how to map (x, y, z) positions measured with respect to the camera axis to (u, v) points on the image plane. The extrinsic calibration specifies a rotation matrix and a translation vector that tells us how to transform from the camera coordinate system to the world coordinate system. In computer vision with LIDAR, we might be interested in applying random rotation augmentation to LIDAR point clouds for the purpose of training a deep neural network to perform $3D$ object detection from $3D$ point clouds. In quantum mechanics the representation theory of the rotation group leads to the phenomenon of spin.

The purpose of this article is to demystify several well known basic mathematical facts about 3D rotations, in particular the various ways to represent them. Along the way we will encounter some group theory, topology, quaternions etc. We also present code snippets in python that implement some of the methods we describe. The full source code is available at `hhtps://github.com/bodsul/rot-3D`.

The core of this article only assumes basic familiarity with cross product, linear algebra and matrices, with more advanced topics and other details moved to footnotes. These more advanced pointers can be safely skipped without compromising a practical understanding of the main points.

2 Linear Algebra Foundations

We recall that after a choice of basis $\{e_1, e_2, \dots e_N\}$ on an N-dimensional vector space, a linear transformation $\mathbf{A}$ can be represented by a matrix $A = \{A_{ij}\}$ defined by $Ae_i = \sum_i A_{ji} e_i$. Under a change of basis $e \to Ge$, the matrix representation transforms as $A \to GAG^{-1}$.

We recall that λ is an eigenvalue of a matrix M if there exists a non-zero eigenvector v such that $Mv = \lambda v$. This implies that the determinant $det(M - \lambda) = 0$, and the later constraint is typically how the eigenvalues and eigenvectors are determined. This constraint gives a polynomial whose roots and multiplicities correspond to eigenvalues and their multiplicities.

We recall that the determinant of a matrix usually defined using the usual determinant expansion is the product of its eigenvalues and its trace usually defined as the sum of diagonal entries is the sum of its eigenvalues.

Finally we recall that the determinant and trace of a matrix are basis independent concepts i.e they are properties of the linear transformation (which the matrix represents) on the underlying vector space.

3 $SO(3)$ Matrices

Rotations in $3D$ are linear transformations characterized by the property that they preserve lengths and angles. This implies a matrix M is a rotation iff

$$\begin{aligned} M^t M = 1, and \\ MM^t = 1. \end{aligned} \tag{1}$$

, where the t superscipt represents transpose. We also require

$$det(M) = 1 \tag{2}$$

so that M can be continuously connected to the identity matrix [1]. These are the defining properties of the $SO(3)$ group, and the $SO(3)$ group is precisely the group of $3D$ rotation matrices.

From equ. 1 and 2 one can show [2] that every rotation matrix not equal to the identity has an eigen value with multiplicity one whose value is 1. This implies that every rotation matrix can be characterized by an axis and an angle of rotation about that axis. The axis of rotation is given by the eigenvector whose eigenvalue is 1 and the angle of rotation can be determined as shown later in this section.

Thus we can represent $SO(3)$ by the set of axes of length 2π (parameterized in the closed interval $[-\pi, \pi]$) about the origin in $3D$. We note that this representation is not unique since a rotation by π around an axis n is equivalent to rotation by $-\pi$ around n. We conclude that $SO(3)$ is topologically equivalent to the set of axes of fixed length in $3D$ with antipodal points identified. This is the same thing as a 3-ball with antipodal points identified. We will recover this topological fact algebraically and more precisely when we discuss quaternions below.

Since every $SO(3)$ matrix is a rotation around some axis, we can get some insight into $SO(3)$ by analyzing the the subgroup of rotations around the z-axis [3]. Matrices M_z in this subgroup have the form

$$M_z = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

Where θ is the angle of rotation measured counterclockwise. From equ. 3 we have $Trace(M_z) = 1 + 2cos(\theta)$. Since the trace is basis independent and every $SO(3)$ matrix can be brought to this form by a change of basis, and also because the determinant is 1, we deduce that the eigenvalues of any $SO(3)$ matrix are of the form $1, \exp(i\theta), \exp(-i\theta)$, where θ is the angle of rotation and i is the complex number i. For the above rotation in 3 around the z-axis, the corresponding normalized eigen values are $\{u, v, w\} = \{[0, 0, 1], \frac{1}{\sqrt{2}}[1, i, 0], \frac{1}{\sqrt{2}}[1, -i, 0]\}$ satisfying

$$v \times w = iu, \tag{4}$$

where $\times$ is the cross product. We will use this cross product property of the eigenvectors below to resolve the ambiguity (described above) when we try to extract the axis and angle of rotation from a general $SO(3)$ matrix.

Given an arbitrary $SO(3)$ matrix not equal to the identity, we can determine its axis (up to a minus sign) and angle of rotation by calculating its eigenvalues and corresponding normalized eigenvectors. Let these be $1, exp(i\theta), exp(-i\theta)$ and u, v, w respectively. We can fix the sign ambiguity as follows, once we pick a normalized eigen vector u with eigen value 1 (note $-u$ is also possible), we assign v and w above by requiring that $v \times w = iu$ as in equ. 4 for a rotation matrix about z-axis. We are using the fact that $SO(3)$ matrices distribute over the cross product i.e $M(a \times b) = Ma \times Mb$, for any $SO(3)$ matrix M, and $3D$ vectors a and b [4]. Finally we can solve for theta by finding the unique theta in $[0, 2\pi]$ that matches the eigen values. This is implemented in python below.

```
def rotation_matrix_to_axis_angle_form(m: np.array)->Tuple[np.array, float]:
    """Given a 3 *3 rotation matrix M, Return a normalized axis of rotation and
    the counterclockwise angle of rotation about that axis"""
    e = l.eig(m)

    one_idx = None
    complex_eig = []
    complex_eig_v = []
    for i, v in enumerate(e.eigenvalues):
        if np.abs(v-1) < TOL:
            one_idx =i
        else:
            complex_eig.append(v)
            complex_eig_v.append(e.eigenvectors[:, i])

    assert one_idx is not None
```

[1] For example in the continuous motion of a rigid body, only matrices that can be continously connected to the identity are relevant. Rotations can be composed together and the inverse or a rotation is also a rotation. Thus rotations form a group. The group of linear transformations that preserve lengths and angles is known as $O(3)$ and includes matrices with determinant -1

[2] See for example `https://www.quora.com/How-do-I-prove-that-1-is-one-of-the-eigen-values-in-a-rotation-matrix`

[3] This is an $SO(2)$ subgroup

[4] this is obvious using the definition of cross product as the area of the parallelogram spanned by a and b and the fact that $SO(3)$ matrices preserve lengths, angles and also areas.

```
    axis = e.eigenvectors[:, one_idx].real

    cross = np.cross(complex_eig_v[0], complex_eig_v[1]).imag
    if l.norm(cross-axis) < TOL:
        id_for_theta = 0
    else:
        id_for_theta = 1

    theta_1 = np.arccos(complex_eig[id_for_theta].real)
    theta_2 = np.pi-theta_1 + np.pi

    for theta in [theta_1, theta_2]:
        if np.abs(np.sin(theta) - complex_eig[id_for_theta].imag) < TOL:
            theta_sol = theta
            break

    return axis, theta_sol
```

To transform from an axis of rotation u and and counterclockwise rotation angle θ to an $SO(3)$ matrix we compute a right handed orthonormal basis u, v, w (first vector in basis is the axis of rotation) and form the change of basis matrix $T = [u, v, w]$, whose columns are the right handed orthonormal basis. Then $T^t M_z T$ is the $SO(3)$ matrix corresponding to the rotation). This is implemented in python below.

```
def get_vec_orthogonal_to(v: np.array) -> np.array:
    """Given a non zero vector v return a vector orthorgonal to v"""
    res = np.zeros(3)
    for i in range(3):
        if v[i] == 0:
            res[i] = 1
            return res

    return np.array([v[1], -v[0], 0])

def complete_to_right_handed_orthornomal_basis(v: np.array)-> List[np.array]:
    """Given a non-zero 3D vector return an a right handed orthonormal basis whose
                                                first vector is v"""
    v1 = get_vec_orthogonal_to(v)
    v1 = v1/l.norm(v1)
    v2 = np.cross(v, v1)
    return [v, v1, v2]

def axis_angle_to_rot_matrix(v: np.array, theta: float) -> np.array:
    """Given an axis and a counterclockwise angle of rotation theta around that axis,
                                                return the corresponding 3*3 rotation
                                                matrix"""
    v = v/l.norm(v)
    ortho_basis = complete_to_right_handed_orthornomal_basis(v)
    T = np.stack(ortho_basis)
    m =np.array([[1, 0, 0], [0, np.cos(theta), -np.sin(theta)], [0, np.sin(theta), np.
                                                cos(theta)]])
    return reduce(np.dot, [T.transpose(), m, T])
```

Therefore we have learned how to transform an SO(3) matrix to an axis and angle of rotation and vice-versa.

One other thing to note is that $SO(3)$ matrices can be thought of as parameterizing the set of possible right handed [5] orthonormal basis in $3D$. More precisely if we pick one right handed orthonormal basis, all the other right handed orthonormal basis can be obtained as an $SO(3)$ rotation of the one we started form [6].

4 Quaternions and 3D-Rotations

The quaternion algebra $\mathbb{H}$ is the unique associative algebra consisting of a unit (the element 1), and three square roots of -1 that anti-commute with each other and have a cylic product structure wrt.

[5] A more general way to think about handedness in any dimenions is orientations. An orientation on a real vector space is simply an ordered choice of basis. Given another ordered choice of basis, we say they give the same orientation if the deteminant of the invertible linear transformation between them is 1 and a different orientation if the determinant is -1. Thus there are two choices of orientations on a real vector space.

[6] More formally the set of all right handed basis is an $SO(3)$ torsor

each other. More formally [7]. More formally $\mathbb{H}$ is the 4-dimensional algebra of vectors spanned by $1, i, j, k$ i.e $a + bi + cj + dk : a, b, c, d \in \mathbb{R}$ satisfying

$$\begin{aligned} i^2 = j^2 = k^2 = -1 \\ ij = k, jk = i, ki = j \\ ji = -k, kj = -i, ik = -j \end{aligned} \tag{5}$$

Given a quaternion $q = a + bi + cj + dk$. We define its conjugate as $q^* = a - bi - cj - dk$ and the norm as $|q| = qq* = a^2 + b^2 + c^2 + d^2$.

The norm satisfies

$$|q1q2| = |q1||q2|. \tag{6}$$

The set of quaternions of unit norm constrained by $|q| = a^2 + b^2 + c^2 + d^2 = 1$ is a 3-sphere ($3D$-hypersphere).

The $3D$ vector space $\mathbb{I}$ of imaginary quaternions is given by the subset of quaternions of the form $bi + cj + dk$.

The connection of rotations to quaternions comes from the fact that for any imaginary quaternion $q \in \mathbb{I}$ and any unit quaternion r, we have that the quaternion

$$s = rqr^* \in \mathbb{I} \tag{7}$$

(i.e purely imaginary), by the property in equ. 4, we have that $|s| = |q|$ and also the operation defining s in equ. 7 is linear in q.

Thus from pure algebra we find that the linear operation 7, of a unit quaternion r on imaginary quaternions corresponds to $3D$ rotations. We note that the unit quaternion r and $-r$correspond to the same $3D$ rotation since the minus sign cancels in 7.

Thus from pure algebra we find that the above operation of a unit quaternion r on imaginary quaternions corresponds to 3D rotations. We note that the unit quaternion u and -u correspond to the same 3D rotation since the minus sign cancels.

To see that all $3D$ rotations can be represented using unit quaternions, we use the representation of axis and angles discussed above. Let n be the purely imaginary quaternion corresponding to a unit axis in $3D$, note that n satisfies using quaternion multiplication $n^2 = -1$. Now consider the exponential representation $\exp(n\alpha) = \cos(\alpha) + n\sin(\alpha)$ [8] .We note that this a unit quaternion that satisfies $\exp(n\alpha)q\exp(-n\alpha) = q$ for any imaginary quaternion proportional to n [9]. Thus the unit quaternion $\exp(n\alpha)$ corresponds to a rotation about the n axis and the parameter α allows us to vary the angle. of rotation. The precise correspondence to the actual angle of counterclockwise rotation θ discussed above is $\exp(n\frac{\theta}{2})$. [10].

Thus to transform from a unit quaternion to an $SO(3)$ matrix, we first solve for n and theta to get the exponential representation of the unit quaternion. From the axis and angle in exponential representation, we can form the $SO(3)$ matrix using the method described in the previous section above. A basic python class that implements all of the above is given below.

```
class quaternion:
    def __init__(self, real: float, i: float, j: float, k: float) -> None:
        self.real = real
        self.i = i
        self.j = j
        self.k = k

    def __abs__(self) -> float:
        return self.real*self.real + self.i*self.i + self.j*self.j + self.k*self.k

    def __mul__(self, right):
        real = self.real * right.real - (self.i*right.i + self.j*right.j + self.k*
                                                    right.k)
        i = self.j*right.k - self.k*right.j + self.real * right.i + self.i * right.
                                                    real
        j = self.k*right.i - self.i*right.k+ self.real * right.j + self.j * right.real
```

[7] The quaternions are a fascinating extension of complex numbers and have several connections to other areas including clifford algebras, physics, topology and geometry. See `https://arxiv.org/pdf/math/0105155.pdf` for more advanced and extensive discussion on normed division algebras.

[8] This follows folloes from Euler's identity for the complex number 1 and the fact that $n^2 = -1$

[9] In this case the product commutes

[10] We can proof this by showing that any quaternion orthorgonal to n gets rotated by an angle θ, see for example `https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation`. Our python implementation also verifies that this is correct.

```
        k = self.i*right.j - self.j*right.i + self.real * right.k + self.k * right.
                                                                    real
        return quaternion(real, i, j, k)

    def __str__(self):
        return f'{self.real} + {self.i}i + {self.j}j + {self.k}k'

    def conjugate(self):
        return quaternion(self.real, -self.i, -self.j, -self.k)

    def Rotate(self, v):
        assert abs(abs(self)-1) < TOL

        v_q = quaternion(0, v[0], v[1], v[2])
        v_q_rot = self*v_q*self.conjugate()
        return np.array([v_q_rot.i, v_q_rot.j, v_q_rot.k])

    @classmethod
    def uniformly_random_unit_quaternion(cls):
        p = uniformly_random_point_on_n_sphere(3)
        return quaternion(p[0], p[1], p[2], p[3])

    @classmethod
    def unit_quaternion_to_exp_form(cls, q):
        assert abs(abs(q)-1) < TOL
        half_theta_1 = np.arccos(q.real)
        half_theta_2 = np.pi - half_theta_1 + np.pi
        theta_1 = half_theta_1*2
        theta_2 = half_theta_2/2

        for theta in [theta_1, theta_2]:
            axis = np.array([q.i/np.sin(theta/2), q.j/np.sin(theta/2), q.k/np.sin(
                                                                    theta/2)])
            if abs(l.norm(axis)-1) < TOL:
                return axis, theta

    @classmethod
    def unit_quaternion_from_exp_form(cls, axis, theta):
        assert abs(l.norm(axis)-1) < TOL
        cos_val = np.cos(theta/2)
        sin_val = np.sin(theta/2)
        return quaternion(cos_val, axis[0]*sin_val, axis[1]*sin_val, axis[2]*sin_val)
```

To transform from an $SO(3)$ matrix to a unit quaternion, we compute the axis and angle followed by the exponential representation of the unit quaternion and then we apply the generalization of Euler's formula above. This is implemented below.

```
r = quaternion.uniformly_random_unit_quaternion()
axis, theta = quaternion.unit_quaternion_to_exp_form(r)
m = axis_angle_to_rot_matrix(axis, theta)

axis_, theta_ = rotation_matrix_to_axis_angle_form(m)

r_ = quaternion.unit_quaternion_from_exp_form(axis_, theta_)
print(r)
print(r_)
```

In the above code snippet we also check that these operations are inverses of each other. To fix the sign ambiguity, we can restrict to unit quaternions to have the first non-zero component be positive.

One can show that if $sqs^* = tqt^*$, then $s = \pm t$.

Thus we find that $SO(3)$ is equivalent to unit quaternions with the identification $s = -s$. This is precisely the 3-sphere with antipodal points identified. Just like a 2-sphere with antipodal points identified is topologically equivalent to a 2-disc with antipodal points identified, a 3-sphere with antipodal points identified is topologically equivalent to a 3-ball with antipodal points identified. Therefore we re-connect with what we discovered earlier using axis and angle of rotation that $SO(3)$ is topologically equivalent to a 3-ball with antipodal points identified.

5 Euler Angles and 3D-Rotations

There are many different conventions for Euler angles [11], here we discuss the pitch, yaw, and roll i.e (x-y-z) parameterization.

[11] see https://en.wikipedia.org/wiki/Euler_angles

The corresponding formulas for other Euler angle parameterization can be derived using trigonometry discussed below. The$(x-y-z)$ Euler angle representation represents an $SO(3)$ matrix as a product $M_z M_y M_x$ of rotation around the x-axis by an angle , followed by rotation about the y-axis by an angle followed by rotation about the z-axis.

The formula for transforming from the $(x-y-z)$ Euler angle parameterization to an SO(3) matrix is implemented in python below [12].

```
def x_y_z_euler_angles_to_rotation_matrix(roll, pitch, yaw):
    return np.array([[cos(pitch)*cos(yaw), sin(roll)*sin(pitch)*cos(yaw) - cos(roll)*
                                           sin(yaw), cos(roll)*sin(pitch)*cos(yaw)
                                            + sin(roll)*sin(yaw)],
                     [cos(pitch)*sin(yaw), sin(roll)*sin(pitch)*sin(yaw) + cos(roll)*
                                                    cos(yaw), cos(roll)*
                                                    sin(pitch)*sin(yaw) -
                                                    sin(roll)*cos(yaw)],
                     [-sin(pitch), sin(roll)*cos(pitch), cos(roll)*cos(pitch)]
                     ])
```

This is how we transform from the $(x-y-z)$ Euler angle parameterization to the $SO(3)$ matrix parameterization.

To transform from the $(x-y-z)$ Euler angle parameterization to the unit quaternion parameterization, we simply use the corresponding quaternion exponential form for the individual rotations and use quaternion multiplication, this is implemented in python below:

```
def x_y_z_euler_angles_to_unit_quaternion(roll, pitch, yaw):
    return quaternion(cos(yaw/2), 0, 0, sin(yaw/2))*quaternion(cos(pitch/2), 0, sin(
                                           pitch/2), 0)*quaternion(cos(roll/2),
                                           sin(roll/2), 0, 0)
```

To transform from the $SO(3)$ matrix form to the $(x-y-z)$ Euler angle parameterization we have to solve the trigonometric equations. Details can be found in previous footnote for this form of Euler angles The main thing to note is that when $cos(theta)\neg = +/1$, we have two solutions and we can fix a convention to choose one. When $\cos(theta) = \pm 1$, there is well known Gimbal lock issue, where we have infinitely many solutions for the other two angles. However by fixing one to be 0, (or any other value) we get a unique Euler angle parameterization. We provide the python implementation below.

```
def x_y_z_euler_angles_from_rotation_matrix(m) -> Union[Tuple[list, list, list], Tuple
                                          [float, float, float]]:
    if abs(m[2][0]- 1) > TOL and abs(m[2][0]+1) > TOL:
        pitch = [-arcsin(m[2][0]), np.pi + arcsin(m[2][0])]
        roll = [arctan2(m[2][1]/cos(p), m[2][2]/cos(p)) for p in pitch]
        yaw = [arctan2(m[1][0]/cos(p), m[0][0]/cos(p)) for p in pitch]
    else: # Gimbal lock
        if abs(m[2][0]- 1) < TOL:
            pitch = -np.pi/2
            yaw = 0
            roll = -yaw + arctan2(-m[0][1], -m[0][2])
        else:
            pitch = np.pi/2
            yaw = 0
            roll = yaw + arctan2(m[0][1], m[0][2])

    return roll, pitch, yaw
```

One can also use trigonometric manipulations to go directly from the unit quaternion representation to the $(x-y-z)$ Euler angle representation, we omit that here but the enthusiastic reader can try to work that out the details as an exercise. It suffices to say that what we have discussed here enables us to do it it three steps unit quaternion $\rightarrow SO(3)$ matrix$\rightarrow$ Euler angle. This is implemented in python below.

```
def x_y_z_euler_angles_from_unit_quaternion(q):
    m = quaternion_to_rot_matrix(q)
    return x_y_z_euler_angles_from_rotation_matrix(m)
```

6 Pros and Cons Of Different Representations Of 3D-Rotations

The matrix representation is the defining representation and is the most natural. To rotate a 3D vector, simply multiply by the corresponding $SO(3)$ matrix. However it is not the most efficient,

[12] see `https://eecs.qmul.ac.uk/~gslabaugh/publications/euler.pdf` for more details about this form of Euler angles.

multiplying two $SO(3)$ matrices requires 27 multiplication and 18 addition operations. Also storing a general $SO(3)$ matrix requires storing 9 numbers [13].

The axis and angle of rotation representation is the most intuitive but does not come readily given in most applications. Also two rotation matrices in this form with non-parallel axes of rotation are not readily multiplied.

The quaternion representation is somewhat abstract (hopefully this article makes it less so!), but it is quite efficient for doing numerical calculations/simulations on the computer. Multiplying two unit quaternions involves 16 multiplication and 12 addition operations. Storing a unit quaternion involves storing 4 numbers [14].

The different Euler angle representations come up naturally in applications in mechanics, aerospace engineering etc. and are found in many textbooks, but there is the gimbal lock issue and also rotation matrices are not easily multiplied in this representation.

7 SU(2) and 3D-Rotations

This section is for more advanced readers and can be skipped.

Unit quaternions discussed above are actually isomorphic (the same as groups) to the lie group $SU(2)$. This is well known in quantum physics and the classical study of lie groups, and leads to the phenomenon of integer and half integer spin in quantum mechanics.

A lie group is a group that is also a smooth differentiable manifold for which the map induced by group multiplication is also smooth for all group elements. The $SU(2)$ lie group is defined as the group of complex 2×2 matrices satisfying

$$\begin{aligned} U^\dagger U = UU^\dagger = 1 \\ det(U) = 1 \end{aligned} \tag{8}$$

The constraints imposed by the definition implies that $SU(2) = a + b\sigma_i + c\sigma_j + d\sigma_k : a, b, c, d \in \mathbb{R}$, where $a^2 + b^2 + c^2 + d^2 = 1$ [15]. Where $\sigma_i = -i\sigma_x, \sigma_j = -i\sigma_y, \sigma_k = -i\sigma_z$, where $\sigma_x, \sigma_y, \sigma_z$ are the Pauli spin matrices [16].

Incidentally the matrices $\sigma_i, \sigma_y j \sigma_k$ satisfy the quaternion algebra! This essentially proofs the isomorphism of unit quaternions to SU(2).

In terms of SU(2), the imaginary quaternions correspond to the lie algebra of $SU(2)$. A working definition of the Lie algebra of a matrix Lie group is the vector space of matrices that when exponentiated give rise to elements of the matrix lie group [17].

The representation of $SO(3)$ we encountered above using quaternions is precisely the adjoint representation of $SU(2)$ [18]. The 3-Sphere is $SU(2)$ and the 3-sphere with aniti-podal points identified is the phenomenon that $SU(2)$ is the universal cover of $SO(3)$ and it is a double cover [19]. A Lie group and its universal cover [20] have isomorphic Lie algebras.

In classical mechanics, in the Lagrangian formalism symmetries of the action give rise to conserved quantities. The conserved quantities for rotational symmetry is angular momentum. In quantum mechanics observables are represented by operators on a Hilbert space and the Hilbert space forms a representation for the operator algebra of observables. The operator algebra of angular momentum in a finite dimensional complex vector space gives rise to the phenomenon of spin in quantum mechanics. The operator algebra of angular momentum is precisely the Lie algebra of SO(3) which as we learned above is isomorphic to the Lie algebra of SU(2). The universal cover of a Lie group is interesting because it is the unique Lie group whose representations are in one to one correspondence to its Lie

[13] Only 3 are actually independent.

[14] We can actually store just 3 numbers and use the unit property to compute the 4th one when needed.

[15] the interested reader can prove this using the constraints imposed by the definition of $SU(2)$

[16] see `https://en.wikipedia.org/wiki/Pauli_matrices`.

[17] In differential geometry we can also define the Lie algebra as the space of left invariant vector fields under pull back by the inverse of the multiplication after the push forward by multiplication. This usual Lie algebra is recovered as the Lie bracket of left invariant vector fields, the dimension of the Lie algebra is equal to the dimension of the Lie group manifold and it can be identified with the tangent space at the identity, and we can recover the exponential definition for matrix lie groups using the exponential map for Riemannian manifolds under the unique left invariant metric known as the killing form on the Lie group

[18] The adjoint representation is the derivative of the smooth map gpg^{-1} of points p on the Lie group manifold at the identity. It is equivalent to conjugation for matrix Lie groups

[19] formally $SO(3) = \frac{SU(2)}{\mathbb{Z}_2}$, which is true as a quotient of groups and also as a quotient of topological spaces.

[20] The universal cover of a connected Lie group$G$ is a connected and simply connected Lie group H such that $G = \frac{H}{D}$ where D is a discrete normal subgroup of H. In this case the fundamental group which is the homotopy group of non-contractible loops at a point in H, $\pi_1(H) \simeq D$

algebra. [21]. Integer spin particles transform under integer spin representations of $SU(3)$ which descend to representations of $SO(3)$ under the double covering map. Half integer spin elementary particles transform under half integer spin representations of $SU(2)$ which do not descend to representations of $SO(3)$ [22].

8 Random $3D$ Rotations

As a final bonus point, we will learn how to generate uniformly random rotations of $3D$ vectors which can be useful for $3D$ point cloud data augmentation and other applications. We will use the quaternion parameterization. First we generate generate uniformly random points on the 3-sphere. To do this we note that in $4D$ spherical polar coordinates (r, Ω) where omega represents the $4D$ polar angles, if we sample $4D$ points $x = (x_1, x_2, x_3, x_4)$ where x_i are sampled independently from any spherically symmetric distribution, then $\frac{x}{|x|}$ gives uniformly distributed points on the 3-sphere [23]. Thus we can sample from a Gaussian distribution with mean 0 and covariance 1 and normalize the corresponding $4D$ points to get random points uniformly distributed on a 3-sphere. This descends to uniformly distributed points on the 3-sphere with antipodal points identified which as we learned above is the same as $SO(3)$ in the unit quaternion parameterization. Finally we can use the uniformly randomly generated unit quaternion to rotate $3D$ vectors. This is implemented in python within the quaternion class above.

9 Summary

In summary we discussed some properties of rotations in $3D$ and various forms of parameterizing them. We also discussed how to convert back and forth from various parameterizations. That is all for now. Thanks for reading!

[21] non-contractible loops can be an obstruction to exponentiating a representation of the algebra to a representation of the group

[22] actually for elementary particles the relevant group is the Lorentz group, but the universal cover of the Lorentz group is $\mathbb{SL}(2, \mathbb{C})$ whose Lie algebra is isomorphic to the direct sum of two copies of the Lie algebra of $SU(2)$.

[23] Normalizing the 4D vector whose coordinates are generated independently from the same spherically symmetric distribution corresponds to marginalizing over the radial coordinate r, this amounts to integrating the probaility density over r, by spherical symmetry the density does not depend on r, therefore the resulting probability distribution on the 3-sphere does not depend on the $4D$ polar angles and we have a uniform distribution