Notes for Usage of 3D Lie Group Utility Functions

MT

Creation Date: 2022-May-08
Previous Edit: 2022-May-09

Abstract

This document is to provide some notes on using the Lie group utility functions implemented in the "lie_utils.h" file.

1 Notation

The notation style adopted in this work follows a similar scheme used by Prof. Tim Barfoot's book *State Estimation For Robotics*.

Symbol	Description
$\mathbf{R}$	3 x 3 rotation matrix on SO(3)
$\mathbf{q}$	4 x 1 unit quaternion, where $\mathbf{q} = [q_w \; q_x \; q_y \; q_z]^T$, and q_w is the scalar part of the quaternion
$\boldsymbol{\phi}$	3 x 1 axis-angle representation of rotation
ϕ	scalar rotation angle as $\phi = \lVert\boldsymbol{\phi}\rVert_2$
$\mathbf{a}$	3 x 1 axis of rotation, where $\phi\mathbf{a} = \boldsymbol{\phi}$ and $\mathbf{a}^T\mathbf{a} = 1$
$\mathbf{J}$	3 x 3 left Jacobian of SO(3)
$\mathbf{T}$	4 x 4 transformation matrix on SE(3), where $\mathbf{T} \equiv \begin{bmatrix} \mathbf{R} & \mathbf{J}\boldsymbol{\rho} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$
$\mathbf{t}$	3 x 1 translational vector, and $\mathbf{t} = \mathbf{J}\boldsymbol{\rho}$
$\boldsymbol{\xi}$ or $\boldsymbol{\zeta}$	6 x 1 tangential vector corresponding to the transformation on SE(3), where $\boldsymbol{\xi} = \begin{bmatrix} \boldsymbol{\rho} \\ \boldsymbol{\phi} \end{bmatrix}$
$\mathcal{T}$	6 x 6 adjoint form of transformation matrix, where $\mathcal{T} = \mathrm{Ad}(\mathbf{T}) \equiv \begin{bmatrix} \mathbf{R} & (\mathbf{J}\boldsymbol{\rho})^\wedge\mathbf{R} \\ \mathbf{0}_{3\times3} & \mathbf{R} \end{bmatrix}$
$\mathcal{J}$	6 x 6 left Jacobian of SE(3)
$(\cdot)^\wedge$	skew-symmetric operator (overloaded), $\boldsymbol{a}^\wedge = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}^\wedge = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix}_{3\times3}$ and $\boldsymbol{\xi}^\wedge = \begin{bmatrix} \boldsymbol{\rho} \\ \boldsymbol{\phi} \end{bmatrix}^\wedge = \begin{bmatrix} \boldsymbol{\phi}^\wedge & \boldsymbol{\rho} \\ \mathbf{0}^T & 0 \end{bmatrix}_{4\times4}$
$(\cdot)^\curlywedge$	curly hat operator over SE(3), $\boldsymbol{\xi}^\curlywedge = \begin{bmatrix} \boldsymbol{\rho} \\ \boldsymbol{\phi} \end{bmatrix}^\curlywedge = \begin{bmatrix} \boldsymbol{\phi}^\wedge & \boldsymbol{\rho}^\wedge \\ \mathbf{0}_{3\times3} & \boldsymbol{\phi}^\wedge \end{bmatrix}_{6\times6}$
$\exp(\cdot)$	matrix exponential map (overloaded), where $\mathbf{R} = \exp(\boldsymbol{\phi}^\wedge)$ and $\mathbf{T} = \exp(\boldsymbol{\xi}^\wedge)$ and $\mathcal{T} = \exp(\boldsymbol{\xi}^\curlywedge)$

2 List of functions implemented

SO(3) Utility Functions

- unitQuaternionToAngleAxis()
- angleAxisToUnitQuaternion()
- rotationMatrixArrayToUnitQuaternion()
- vecHat()
- expPhiHat()
- phiToSO3()
- lnVeeToPhi()
- leftJacobianSO3()
- invLeftJacobianSO3()
- eigenRotationMatrixToUnitQuaternion()

SE(3) Utility Functions

- zetaHat()
- expZetaHat()
- zetaToSE3()
- lnVeeToZeta()
- zetaCurlyHat()
- AdjointSE3()
- invAdjointSE3()
- leftJacobianSE3()
- invLeftJacobianSE3()

3 SO(3) Utility Functions

Functions and identities on SO(3).

3.1 unitQuaternionToAngleAxis()

```
template<typename T>
inline void unitQuaternionToAngleAxis(const T *unit_quaternion, T* angle_axis)
```

This is a template function which takes a 1D input array with four elements as the first argument and assigns values to the 1D output array with three elements at the second argument. The first argument relates to the unit quaternion in the order of $[q_w\ q_x\ q_y\ q_z]^T$, and the second argument is the 3 x 1 angle-axis representation of the rotation as $[\phi_1\ \phi_2\ \phi_3]^T$.

Usage example:

```
    double aa[3]; // initialize an array for the angle-axis
    unitQuaternionToAngleAxis(q, aa); // q stores the unit quaternion values
```

3.2 angleAxisToUnitQuaternion()

```
template<typename T>
inline void angleAxisToUnitQuaternion(const T* angle_axis, T* unit_quaternion)
```

This is a template function which takes a 1D input array with three elements as the first argument and assigns values to the 1D output array with four elements at the second argument. The first argument relates to the 3 x 1 angle-axis representation of a rotation as $[\phi_1\ \phi_2\ \phi_3]^T$, and the second argument is the unit quaternion in the order of $[q_w\ q_x\ q_y\ q_z]^T$.

Usage example:

```
    double q[4]; // initialize an array for the unit quaternion
    angleAxisToUnitQuaternion(aa, q); // aa stores the 3 x 1 angle-axis values
```

3.3 rotationMatrixArrayToUnitQuaternion()

```
template<typename T>
inline void rotationMatrixArrayToUnitQuaternion(const T rot_mat_vec[9], T q[4])
```

This is a template function. The input rotation matrix should be stored in a 1D array with nine elements representing the raveled form of a **row major** 3 x 3 rotation matrix on SO(3). The output unit quaternion will have a format of $[q_w\ q_x\ q_y\ q_z]$, that is the scalar part will be the first element.

Usage example:

```
    double q[4]; // initialize an array for the unit quaternion
    // Note: the R_arr is a 1D array with 9 elements representing the 3x3 rot mat
    rotationMatrixArrayToUnitQuaternion(R_arr, q); // R_arr is in ROW MAJOR order
```

3.4 vecHat()

```
template<typename T>
inline void vecHat(const T vec[3], T vec_hat[9])
```

This is a template function. The input should be a 1D array with three elements. The output matrix will be stored in a 1D array with nine elements in a ROW MAJOR order.

$$\boldsymbol{a}^\wedge = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}^\wedge = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix}_{3\times 3} \Longrightarrow [0\ -a_3\ a_2\ a_3\ 0\ -a_1\ -a_2\ a_1\ 0]$$

Usage example:

```
    double vec_hat[9]; // initialize an array for output
    // vec_hat is in ROW MAJOR order
    vecHat(vec, vec_hat); // vec is a 1D array stores the 3x1 vector values
```

3.5 expPhiHat()

```
void expPhiHat(const Eigen::Matrix3d &phi_hat_mat, Eigen::Matrix3d &R_mat)
```

This function takes 3 x 3 Eigen Matrix as both input and output. The input should be a 3 x 3 Eigen matrix with double precision values, storing the skew-symmetric matrix associated with a 3 x 1 angle-axis vector. The output is the corresponding 3 x 3 rotation matrix for $\phi^\wedge$.

$$\underbrace{\boldsymbol{\phi}^\wedge}_{\text{input}} \longrightarrow \exp(\boldsymbol{\phi}^\wedge) \equiv \cos(\phi)\mathbf{I}_{3\times 3} + (1-\cos(\phi))\mathbf{a}\mathbf{a}^T + \sin(\phi)\mathbf{a}^\wedge = \underbrace{\mathbf{R}_{3\times 3}}_{\text{output}}$$

Usage example:

```
  /* In this case:
    - a 3x1 angle-axis vector has been converted to its corresponding
        3x3 skew-symmetric matrix and saved in variable Phi_hat
  */
  Eigen::Matrix3d R_mat; // initialize a 3x3 Eigen matrix to store the rotation mat
  expPhiHat(Phi_hat, R_mat); // obtain rotation mat from 3x3 skew-symmetric mat
```

3.6 phiToSO3()

```
void phiToSO3(const Eigen::Vector3d &phi_vec, Eigen::Matrix3d &R_mat)
```

This function takes an Eigen::Vector3d as input and assigns values to an Eigen::Matrix3d as output. The input should be a 3 x 1 Eigen vector with double precision values, storing a 3 x 1 angle-axis rotational vector. The output is the corresponding 3 x 3 rotation matrix.

$$\boldsymbol{\phi} \longrightarrow \mathbf{R}$$

Usage example:

```
  /* In this case:
    - an Eigen::Vector3d phi_vec stores the 3 x 1 angle-axis rotational vector
  */
  Eigen::Matrix3d R_mat; // initialize a 3x3 Eigen matrix to store the rotation mat
  phiToSO3(phi_vec, R_mat); // obtain rotation mat from 3x1 rotational vec
```

3.7 lnVeeToPhi()

```
void lnVeeToPhi(const Eigen::Matrix3d &R_mat, Eigen::Vector3d &phi)
```

This function takes an Eigen::Matrix3d as input and assigns values to an Eigen::Vector3d as output. The input should be a 3 x 3 Eigen matrix with double precision values, storing a rotation matrix. The output is the corresponding 3 x 1 angle-axis rotational vector.

$$\ln(\mathbf{R})^{\vee} = \boldsymbol{\phi}$$

Usage example:

```
/* In this case:
  - an Eigen::Matrix3d R_mat stores the 3 x 3 rotation matrix
*/
Eigen::Vector3d phi_vec; // initialize a 3x1 Eigen vector to store output
lnVeeToPhi(R_mat, phi_vec); // obtain rotational vec from 3x3 rotation mat
```

3.8 leftJacobianSO3()

```
void leftJacobianSO3(const Eigen::Vector3d &phi_vec, Eigen::Matrix3d &left_Jac_mat)
```

This function takes an Eigen::Vector3d as input and assigns values to an Eigen::Matrix3d as output. The input should be a 3x1 Eigen vector with double precision values, storing a 3 x 1 angle-axis rotational vector. The output is the corresponding 3 x 3 left Jacobian matrix.

$$\mathbf{J} = \begin{cases} \frac{\sin\phi}{\phi}\mathbf{I} + (1 - \frac{\sin\phi}{\phi})\mathbf{a}\mathbf{a}^T + \frac{1-\cos\phi}{\phi}\mathbf{a}^{\wedge} & \text{for } \phi > \text{ numeric limit} \\ \mathbf{I} + \frac{1}{2}\boldsymbol{\phi}^{\wedge} & \text{otherwise} \end{cases}$$

Usage example:

```
/* In this case:
  - an Eigen::Vector3d phi_vec stores the 3x1 angle-axis rotational vector
*/
Eigen::Matrix3d J_left; // initialize a 3x3 Eigen matrix to store output
leftJacobianSO3(phi_vec, J_left); // obtain left jacobian from the 3x1 rot vec
```

3.9 invLeftJacobianSO3()

```
void invLeftJacobianSO3(const Eigen::Vector3d &phi_vec,Eigen::Matrix3d &inv_Jac_mat)
```

This function takes an Eigen::Vector3d as input and assigns values to an Eigen::Matrix3d as output. The input should be a 3 x 1 Eigen vector with double precision values, storing a 3 x 1 angle-axis rotational vector. The output is the corresponding 3 x 3 **inversion** of the left Jacobian matrix.

$$\mathbf{J}^{-1} = \begin{cases} \frac{\phi}{2}\cot\frac{\phi}{2}\mathbf{I} + (1 - \frac{\phi}{2}\cot\frac{\phi}{2})\mathbf{a}\mathbf{a}^T - \frac{\phi}{2}\mathbf{a}^{\wedge} & \text{for } \sin\phi > \text{ numeric limit} \\ \mathbf{I} - \frac{1}{2}\boldsymbol{\phi}^{\wedge} & \text{otherwise} \end{cases}$$

Usage example:

```
/* In this case:
  - an Eigen::Vector3d phi_vec stores the 3x1 angle-axis rotational vector */
Eigen::Matrix3d inv_J_left; // initialize a 3x3 Eigen matrix to store output
invLeftJacobianSO3(phi_vec, inv_J_left); // obtain inversion of the left jacobian
```

3.10 eigenRotationMatrixToUnitQuaternion()

```
typedef Eigen::Matrix<double, 4, 1> Vector4d;
void eigenRotationMatrixToUnitQuaternion(const Eigen::Matrix3d &R_mat, Vector4d &q)
```

This function takes an Eigen::Matrix3d as input and assigns values to a Vector4d as output. The input should be a 3 x 3 Eigen matrix with double precision values, storing a 3 x 3 rotation matrix. The output is the corresponding 4 x 1 unit quaternion representing the same rotation.

$$\mathbf{R} \longrightarrow \mathbf{q}$$

Usage example:

```
    /* In this case:
      - an Eigen::Matrix3d R_mat stores the 3x3 rotation matrix
    */
    Vector4d q; // initialize a 4x1 Eigen vector to store output
    eigenRotationMatrixToUnitQuaternion(R_mat, q); // obtain quaternion from rot mat
```

3.11 Some Additional Notes

From unit quaternion to angle-axis:

$$k = \begin{cases} \text{atan2}(\sqrt{q_x^2 + q_y^2 + q_z^2}\,,\ q_w) & \text{if } q_w \geqslant 0 \\ \text{atan2}(-\sqrt{q_x^2 + q_y^2 + q_z^2}\,, -q_w) & \text{if } q_w < 0 \end{cases}$$

$$\boldsymbol{\phi} = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \end{bmatrix} = \begin{bmatrix} 2 \times q_x \frac{k}{\sqrt{q_x^2+q_y^2+q_z^2}} \\ 2 \times q_y \frac{k}{\sqrt{q_x^2+q_y^2+q_z^2}} \\ 2 \times q_z \frac{k}{\sqrt{q_x^2+q_y^2+q_z^2}} \end{bmatrix}$$

From angle-axis to unit quaternion:

$$\phi = \|\boldsymbol{\phi}\|_2 = \sqrt{\phi_1^2 + \phi_2^2 + \phi_3^2}\,, \qquad \mathbf{a} = \frac{\boldsymbol{\phi}}{\phi} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

$$\mathbf{q} = \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix} = \begin{bmatrix} \cos(\frac{\phi}{2}) \\ a_1 \sin(\frac{\phi}{2}) \\ a_2 \sin(\frac{\phi}{2}) \\ a_3 \sin(\frac{\phi}{2}) \end{bmatrix}$$

4 SE(3) Utility Functions

Functions and identities on SE(3).

4.1 zetaHat()

```
template<typename T>
inline void zetaHat(const T zeta_vec[6], T zeta_hat[16])
```

This is a template function which takes a 1D input array with six elements as the first argument and assigns values to the 1D output array with 16 elements at the second argument. The first argument relates to the 6D tangential vector of SE(3) in the order of $[\rho_1\, \rho_2\, \rho_3\, \phi_1\, \phi_2\, \phi_3]^T$, and the second argument is the corresponding raveled matrix stored in a **row major** order.

$$\underbrace{\zeta_{6\times1}}_{\text{input}} \longrightarrow \zeta^\wedge = \begin{bmatrix}\rho\\ \phi\end{bmatrix}^\wedge = \begin{bmatrix}\phi^\wedge & \rho\\ \mathbf{0}^T & 0\end{bmatrix}_{4\times4} \longrightarrow \text{store row-by-row into an array}$$

Usage example:

```
    double zeta_hat[16]; // initialize an array to store the output
    zetaHat(zeta_arr, zeta_hat); // zeta_arr stores the input values
```

```
    /* In this case:
      - the zeta vector is stored in Eigen::Matrix<double,6,1> zeta_vec
    */
    double zeta_hat[16]; // initialize an array to store the output
    zetaHat(zeta_vec.data(), zeta_hat); // .data() method to get data as an array
```

4.2 expZetaHat()

```
typedef Eigen::Matrix<double, 4, 4> Matrix4d;
void expZetaHat(const Matrix4d &zeta_hat_mat, Matrix4d &T_mat)
```

This function takes 4 x 4 Eigen Matrix as both input and output. The input should be a 4 x 4 Eigen matrix with double precision values, storing the $\zeta^\wedge$ matrix. The output is the corresponding 4 x 4 transformation matrix, $\mathbf{T}$.

$$\underbrace{\zeta^\wedge}_{\text{input}} \longrightarrow \exp(\zeta^\wedge) = \left\{\begin{array}{lr} \mathbf{I}_{4\times4} + \zeta^\wedge + (\frac{1-\cos\phi}{\phi^2})\,(\zeta^\wedge)^2 + (\frac{\phi-\sin\phi}{\phi^3})\,(\zeta^\wedge)^3 & \text{for } \phi^3 > \text{numeric limit}\\ \mathbf{I}_{4\times4} + \zeta^\wedge & \text{otherwise}\end{array}\right\} = \mathbf{T}$$

Usage example:

```
    /* In this case:
      - a 6x1 zeta vector has been converted to its corresponding
        4x4 zeta_hat matrix and stored in variable Zeta_hat
    */
    Matrix4d T_mat; // initialize a 4x4 Eigen matrix to store the transformation mat
    expZetaHat(Zeta_hat, T_mat); // obtain transformation mat from the input
```

4.3 zetaToSE3()

```
typedef Eigen::Matrix<double, 6, 1> Vector6d;
typedef Eigen::Matrix<double, 4, 4> Matrix4d;
void zetaToSE3(const Vector6d &zeta_vec, Matrix4d &T_mat)
```

This function takes a 6D Eigen vector as input and assigns values to a 4 x 4 Eigen matrix as output. The input should be a 6 x 1 Eigen vector with double precision values, storing the tangential vector of SE(3). The output is the corresponding 4 x 4 transformation matrix.

$$\zeta = \begin{bmatrix} \rho \\ \phi \end{bmatrix} \longrightarrow \mathbf{T}$$

Usage example:

```
  /* In this case:
    - a Vector6d zeta_vec stores the 6x1 input vector
  */
  Matrix4d T_mat; // initialize a 4x4 Eigen matrix to store the transformation mat
  zetaToSE3(zeta_vec, T_mat); // obtain transformation matrix from the input
```

4.4 lnVeeToZeta()

```
typedef Eigen::Matrix<double, 6, 1> Vector6d;
typedef Eigen::Matrix<double, 4, 4> Matrix4d;
void lnVeeToZeta(const Matrix4d &T_mat, Vector6d &zeta)
```

This function takes an Eigen 4 x 4 matrix as input and assigns values to an Eigen 6 x 1 vector as output. The input should be a 4 x 4 Eigen matrix with double precision values, storing the transformation matrix. The output is the corresponding 6 x 1 tangential vector.

$$\ln(\mathbf{T})^{\vee} = \zeta$$

Usage example:

```
  /* In this case:
    - a Matrix4d T_mat stores the 4x4 transformation matrix
  */
  Vector6d zeta_vec; // initialize a 6x1 Eigen vector to store output
  lnVeeToZeta(T_mat, zeta_vec); // obtain output from 4x4 transformation mat
```

4.5 zetaCurlyHat()

```
template<typename T>
inline void zetaCurlyHat(const T zeta_arr[6], T zeta_curly_hat[36])
```

This is a template function which takes a 1D input array with six elements as the first argument and assigns values to the 1D output array with 36 elements at the second argument. The first argument relates to the 6D tangential vector of SE(3) in the order of $[\rho_1\ \rho_2\ \rho_3\ \phi_1\ \phi_2\ \phi_3]^T$, and the second argument is the corresponding raveled matrix stored in a **row major** order.

$$\zeta_{6\times1} \longrightarrow \zeta^{\curlywedge} = \begin{bmatrix} \rho \\ \phi \end{bmatrix}^{\curlywedge} = \begin{bmatrix} \phi^{\wedge} & \rho^{\wedge} \\ \mathbf{0}_{3\times3} & \phi^{\wedge} \end{bmatrix}_{6\times6} \longrightarrow \text{store row-by-row into an array}$$

Usage example:

```
  double zeta_curlyHat[36]; // initialize an array to store the output
  zetaCurlyHat(zeta_arr, zeta_curlyHat); // zeta_arr is the input 1D arr with 6 elements
```

4.6 AdjointSE3()

```
typedef Eigen::Matrix<double, 6, 6> Matrix6d;
typedef Eigen::Matrix<double, 4, 4> Matrix4d;
void AdjointSE3(const Matrix4d &T_mat, Matrix6d &Adj_T_mat)
```

This function takes an Eigen 4 x 4 matrix as input and assigns values to an Eigen 6 x 6 matrix as output. The input should be a 4 x 4 Eigen matrix with double precision values, storing the transformation matrix. The output is the corresponding 6 x 6 adjoint matrix.

$$\underbrace{\mathbf{T}_{4\times4}}_{\text{input}} \longrightarrow \mathrm{Ad}(\mathbf{T}) \equiv \mathrm{Ad}\left(\begin{bmatrix} \mathbf{R} & \mathbf{J}\rho \\ \mathbf{0}^T & 1 \end{bmatrix}\right) \equiv \begin{bmatrix} \mathbf{R} & (\mathbf{J}\rho)^\wedge \mathbf{R} \\ \mathbf{0}_{3\times3} & \mathbf{R} \end{bmatrix} \equiv \underbrace{\mathcal{T}_{6\times6}}_{\text{output}}$$

Usage example:

```
/* In this case:
  - a Matrix4d T_mat stores the 4x4 transformation matrix
*/
Matrix6d Adj_T_mat; // initialize a 6x6 Eigen matrix to store output
AdjointSE3(T_mat, Adj_T_mat); // obtain output from 4x4 transformation mat
```

4.7 invAdjointSE3()

```
typedef Eigen::Matrix<double, 6, 6> Matrix6d;
typedef Eigen::Matrix<double, 4, 4> Matrix4d;
void invAdjointSE3(const Matrix4d &T_mat, Matrix6d &inv_Adj_mat)
```

This function takes an Eigen 4 x 4 matrix as input and assigns values to an Eigen 6 x 6 matrix as output. The input should be a 4 x 4 Eigen matrix with double precision values, storing the transformation matrix. The output is the corresponding 6 x 6 inversion of the adjoint matrix.

$$\underbrace{\mathbf{T}_{4\times4}}_{\text{input}} \longrightarrow \mathrm{Ad}(\mathbf{T}^{-1}) \equiv \begin{bmatrix} \mathbf{R}^T & -\mathbf{R}^T(\mathbf{J}\rho)^\wedge \\ \mathbf{0}_{3\times3} & \mathbf{R}^T \end{bmatrix} \equiv \underbrace{\mathcal{T}^{-1}}_{\text{output}}$$

Usage example:

```
/* In this case:
  - a Matrix4d T_mat stores the 4x4 transformation matrix
*/
Matrix4d inv_Adj_mat; // initialize a 6x6 Eigen matrix to store output
invAdjointSE3(T_mat, inv_Adj_mat); // obtain output from 4x4 transformation mat
```

4.8 leftJacobianSE3()

```
typedef Eigen::Matrix<double, 6, 1> Vector6d;
typedef Eigen::Matrix<double, 6, 6> Matrix6d;
void leftJacobianSE3(const Vector6d &zeta_vec, Matrix6d &left_Jac_mat)
```

This function takes a 6D Eigen vector as input and assigns values to a 6 x 6 Eigen matrix as output. The first input argument relates to the 6 x 1 tangential vector of SE(3) in the order of $[\rho_1\ \rho_2\ \rho_3\ \phi_1\ \phi_2\ \phi_3]^T$ The output is the corresponding 6 x 6 left Jacobian matrix.

$$\mathcal{J} = \begin{cases} \mathbf{I} + (\frac{4-\phi\sin\phi-4\cos\phi}{2\phi^2})\,\zeta^{\curlywedge} + (\frac{4\phi-5\sin\phi+\phi\cos\phi}{2\phi^3})\,(\zeta^{\curlywedge})^2 + (\frac{2-\phi\sin\phi-2\cos\phi}{2\phi^4})\,(\zeta^{\curlywedge})^3 + (\frac{2\phi-3\sin\phi+\phi\cos\phi}{2\phi^5})\,(\zeta^{\curlywedge})^4 \\ \mathbf{I} + \frac{1}{2}\zeta^{\curlywedge} \qquad \text{for } \phi^5 < \text{ numeric limit} \end{cases}$$

Usage example:

```
/* In this case:
  - a Vector6d zeta_vec stores the 6x1 tangential vector
*/
Matrix6d J_left; // initialize a 6x6 Eigen matrix to store output
leftJacobianSE3(zeta_vec, J_left); // obtain left jacobian from the input
```

4.9 invLeftJacobianSE3()

```
typedef Eigen::Matrix<double, 6, 1> Vector6d;
typedef Eigen::Matrix<double, 6, 6> Matrix6d;
void invLeftJacobianSE3(const Vector6d &zeta_vec, Matrix6d &inv_Jac_mat)
```

This function takes a 6D Eigen vector as input and assigns values to a 6 x 6 Eigen matrix as output. The first input argument relates to the 6 x 1 tangential vector of SE(3) in the order of $[\rho_1\ \rho_2\ \rho_3\ \phi_1\ \phi_2\ \phi_3]^T$ The output is the corresponding inversion of the 6 x 6 left Jacobian matrix.

$$\underbrace{\zeta_{6\times1}}_{\text{input}} \longrightarrow \mathcal{J} \equiv \begin{bmatrix} \mathbf{J} & \mathbf{Q} \\ \mathbf{0}_{3\times3} & \mathbf{J} \end{bmatrix} \longrightarrow \begin{bmatrix} \mathbf{J}^{-1} & -\mathbf{J}^{-1}\,\mathbf{Q}\,\mathbf{J}^{-1} \\ \mathbf{0}_{3\times3} & \mathbf{J}^{-1} \end{bmatrix} \equiv \underbrace{\mathcal{J}^{-1}}_{\text{output}}$$

Usage example:

```
/* In this case:
  - a Vector6d zeta_vec stores the 6x1 tangential vector
*/
Matrix6d inv_J_left; // initialize a 6x6 Eigen matrix to store output
invLeftJacobianSE3(zeta_vec, inv_J_left); // get inversion of the left jacobian
```