Pinocchio Functions

HE

Left upper script is the coordinate frame

In Pinocchio, inertia of link is attached to the joint frame by parallel axis theorem, exhibiting non-zero off-diagonal elements.

$\tau = M(\theta)\ddot{\theta} + C(\theta, \dot{\theta})\dot{\theta} + g(\theta) = M(\theta)\ddot{\theta} + h(\theta, \dot{\theta})$, $M(\theta) = M(\theta)^T, \dot{M}(\theta) - 2C(\theta, \dot{\theta}) = 0$

Action T on SE3, Action Ad_T on se3, Motion action ad_V (adjoint matrix working on se3)

ad_V appears due to : Newton-Euler unified formulation:

$$\mathcal{F}_b = \mathcal{G}_b \dot{\mathcal{V}}_b - \left[\mathrm{ad}_{\mathcal{V}_b}\right]^T \mathcal{G}_b \mathcal{V}_b.$$

, time derivation:

$$\frac{d}{dt}\mathrm{Ad}_{T_{i,i-1}(t)} = -\mathrm{ad}_{V_i} \times \mathrm{Ad}_{T_{i,i-1}(t)}.$$

ϕ : ***Momentum*** in SE3, Power $= V^T * \phi/2$

R : 3D rotation matrix

Y inertia matrix

$V = [v, w]$ se3

α spatial acceleration in se3

Attribute: SE3, Motion, Force, Inertia, Joint

- **SE3 Class:**
  - `T.rotation` and
  - `T.translation` let you access the individual components of the transformation.
  - `T.inverse()` computes the inverse of the transformation. The overloaded
  - `*` operator allows you to compose transformations, as shown with `T * T_inv`.

  - `T.act(point)` applies the transformation to a 3D point.
  - `T.actInv(point)` applies the inverse transformation (useful for recovering the original point).
  - `pin.SE3.Exp(xi)` computes an SE3 transformation from a 6D twist vector using the exponential map.
  - `T_exp.log()` returns the twist vector (logarithm map) corresponding to the transformation.

- **Motion Class:**
  - `pin.Motion(angular, linear)` creates a motion (twist) from its angular and linear velocity components.
  - `M.vector` returns the full 6D vector representation.
  - `M.angular` and `M.linear` provide access to the angular and linear parts, respectively.
  - The `cross()` method computes the spatial cross product between two motion vectors.
  - `act` Action of a motion set on a force object. The input motion set is represented by a 6xN matrix whose each column represent a spatial motion. The output force set is represented by a 6xN matrix whose each column represent a spatial force.
  - `inertiaAction` Action of an Inertia matrix on a set of motions, represented by a 6xN matrix whose columns represent a spatial motion. out = spatial force (momentum)
  - `motionAction` Action of a motion on a set of motions, represented by a 6xN matrix whose columns represent a spatial motion. $out = \mathrm{ad}_{V1}^{-1}V$, adjoint matrix ad_V
  - `se3Action` SE3 action on a set of motions, represented by a 6xN matrix whose column represent a spatial motion. $out = \mathrm{Ad}_G V$.
  - `se3ActionInverse` Inverse SE3 action on a set of motions, represented by a 6xN matrix whose column represent a spatial motion. $out = \mathrm{Ad}_G^{-1}V$, G:ground

- **Force Class:**
  - `pin.Force(torque, force)` constructs a spatial force vector with the given torque and force components.
  - `F.vector` returns the full 6D vector representation.
  - `F.torque` and `F.force` provide access to the torque and force parts, respectively.
  - `se3Action` and `se3ActionInverse` SE3 action on a set of forces, represented by a 6xN matrix whose each column represent a spatial force. $out = \mathrm{Ad}_G F$, $out = \mathrm{Ad}_G^{-1}F$, G:ground
  - `motionAction` Action of a motion on a set of forces, represented by a 6xN matrix whose each column represent a spatial force. $out = \mathrm{ad}_V^{-1}F$, adjoint matrix ad_V

- **Inertia Class:**
  - `.mass()` mass
  - `.lever()` position of CoM representation.
  - `.inertia()` inertia matrix
  - **vtiv:**
    Computes the quadratic form of motion and inertial forces, which is twice the kinetic energy. Implementation details are provided by derived classes.
    $$\text{out} = V^T M V.$$
  - **variation:**
    The first-order derivative of the spatial inertia matrix. Implementation details are provided by derived classes.
    $$\text{out} = \mathrm{ad}_V' \, M \; - \; M \, \mathrm{ad}_V,$$
    which can be seen as the difference between the `vxi` and `ivx` functions.

- **vxi:**
  The left-multiplication semidirect matrix of the six-dimensional motion (left tangent space) for the spatial inertia matrix, used to compute the derivative of the spatial inertia matrix. Implementation details are provided by derived classes.
  $$\text{out} = \mathrm{ad}_V' \, M.$$
- **ivx:**
  The right-multiplication semidirect matrix of the six-dimensional motion (right tangent space) for the spatial inertia matrix, used to compute the derivative of the spatial inertia matrix. Implementation details are provided by derived classes.
  $$\text{out} = M \, \mathrm{ad}_V.$$
- **se3Action:**
  Transforms the spatial inertia matrix from the body coordinate system to the absolute coordinate system. Implementation details are provided by derived classes.
  $$m_2 = m_1, \quad p_2 = R\,p_1 + P, \quad I_2 = R\,I_1\,R^T.$$

Pinocchio Frame Types

- **ReferenceFrame.LOCAL**: The origin is expressed in the local frame, and the linear and angular velocities of the frame are represented in the local frame.
- **ReferenceFrame.WORLD**: The origin coincides with the world frame, and the velocities are projected in the basis of the world frame, representing the spatial and angular velocities of the frame in the world frame.
- **ReferenceFrame.LOCAL_WORLD_ALIGNED**: origin coinciding with the local frame, but the XYZ axes are aligned with the world frame. In this way, the velocities are projected in the basis of the world frame, representing the linear and angular velocities of the frame in the world frame.

Important Pinocchio Functions

- **pinocchio::computeAllTerms**: Computes all the kinematic and dynamic quantities of the robot model, including forward kinematics, Jacobians, and inertia matrices.
- **pinocchio::forwardKinematics**: Computes the positions and orientations of all joints by propagating the kinematic chain from the base using the given joint configuration (and optionally velocities and accelerations).
- **pinocchio::crba**: Implements the Composite Rigid Body Algorithm to compute the joint-space mass (inertia) matrix of the robot, which is used in dynamics calculations.
- **pinocchio::ccrba**: Computes the centroidal dynamics quantities of the robot. In particular, it's used to efficiently compute the momentum (both linear and angular) of the robot and provides the centroidal momentum matrix, which is valuable for tasks related to balance and dynamic motions.
- **pinocchio::nonLinearEffects**: Calculates the combined effects of gravity, Coriolis, and centrifugal forces acting on the robot, given the current configuration and velocity.
- **pinocchio::computeJointJacobians**: Computes the Jacobian matrices for all joints, mapping joint velocities to the spatial velocities of the corresponding bodies.
- **pinocchio::centerOfMass**: Determines the overall center of mass of the robot model by aggregating the contributions of all bodies based on their masses and positions.

- **pinocchio::jacobianCenterOfMass**: Computes the Jacobian of the robot's center of mass with respect to its joint coordinates, useful for tasks like balance and motion planning.
- **pinocchio::kineticEnergy**: Computes the total kinetic energy of the robot given its mass distribution and joint velocity configuration.
- **pinocchio::potentialEnergy**: Calculates the potential energy of the robot, typically due to gravity, based on the configuration and the positions of its centers of mass.

Rigid Transformation Setup

We consider two coordinate frames, $\{A\}$ and $\{B\}$. A point $\mathbf{p}$ expressed in frame $\{B\}$ is denoted by ${}^{B}\mathbf{p}$. Its coordinates in frame $\{A\}$ are given by ${}^{A}\mathbf{p}$. A rigid transform from frame $\{B\}$ to $\{A\}$ can be written:

$$ {}^{A}\mathbf{p} \;=\; {}^{A}\mathbf{R}_B \; {}^{B}\mathbf{p} \;+\; {}^{A}\mathbf{A}_B, $$

where ${}^{A}\mathbf{R}_B \in SO(3)$ is the 3×3 rotation from $\{B\}$ to $\{A\}$, and ${}^{A}\mathbf{A}_B \in \mathbb{R}^3$ is the translation of the origin of $\{B\}$ as seen from $\{A\}$.

Matrix Form

The above can be embedded in homogeneous coordinates, yielding a 4×4 matrix:

$$ {}^{A}M_B \;=\; \begin{bmatrix} {}^{A}\mathbf{R}_B & {}^{A}\mathbf{A}_B \\ \mathbf{0} & 1 \end{bmatrix} \;. $$

Then,

$$ {}^{A}\mathbf{p}_h \;=\; {}^{A}M_B \; {}^{B}\mathbf{p}_h, $$

where $\mathbf{p}_h = [p_x \; p_y \; p_z \; 1]^\top$ denotes homogeneous coordinates.

Inverse Transform Going from $\{A\}$ back to $\{B\}$ is given by

$$ {}^{B}\mathbf{p} \;=\; ({}^{A}M_B)^{-1} \; {}^{A}\mathbf{p}. $$

In matrix form,

$$ {}^{B}M_A \;=\; \begin{bmatrix} ({}^{A}\mathbf{R}_B)^\top & -\,({}^{A}\mathbf{R}_B)^\top \, {}^{A}\mathbf{A}_B \\ \mathbf{0} & 1 \end{bmatrix}. $$

(These expressions match the figure's A_{M_B} and B_{M_A} blocks.)

Relation to Pinocchio SE3 Objects

In the `Pinocchio` library, a rigid transform is stored in an `SE3` object. If

$$ \mathbf{R} = {}^{A}\mathbf{R}_B, \quad \mathbf{p} = {}^{A}\mathbf{A}_B, $$

then

```
import pinocchio
M = pinocchio.SE3(R, p)
```

is exactly the matrix ${}^{A}M_B$ in homogeneous form.

2.1 Transforming a 3D Point

To transform a *point* in Pinocchio (3D vector),

$$^{B}\mathbf{p} \;\mapsto\; ^{A}\mathbf{p},$$

you can call `M.act` (for a 3D point). In Python:

```
import pinocchio

# Rotation matrix R and translation p
M = pinocchio.SE3(R, p)

p_B = np.array([x_b, y_b, z_b]) # coordinates in frame B
p_A = M.act(p_B) # coordinates in frame A
```

- **Input:**
  - $\mathbf{R}$ (3x3 rotation), $\mathbf{p}$ (3D translation).
  - $\mathbf{p}_B$ (the point in frame B).
- **Output:** $\mathbf{p}_A$, the point in frame A.

2.2 Inverse Transform

If you have a point $^{A}\mathbf{p}$ in frame $\{A\}$ and wish to express it in $\{B\}$, call:

```
p_B = M.actInv(p_A)
```

which performs $\mathbf{M}^{\top}(\mathbf{p}_A)$.

Featherstone-Style Notation

In spatial-algebra contexts (Featherstone's notation), one often sees:

$$E \;=\; ^{B}\mathbf{R}_A \;=\; (^{A}\mathbf{R}_B)^{\top}, \qquad r \;=\; ^{A}\mathbf{A}_B.$$

Then the homogeneous transforms become

$$^{B}M_A \;=\; \begin{bmatrix} E & -E\,r \\ 0 & 1 \end{bmatrix} \quad , \quad ^{A}M_B \;=\; \begin{bmatrix} E^{\top} & r \\ 0 & 1 \end{bmatrix}.$$

In `Pinocchio`, these correspond to SE3$(E^{\top}, r)$ or SE3$(E, \dots)$ depending on whether you are going from $\{A\} \to \{B\}$ or vice versa.

Summary: The figure's rigid transformations are precisely what `Pinocchio` implements with its `SE3` class. The methods `.act(p)` and `.actInv(p)` (for 3D points) or the analogous `.act(motion)` for 6D motions, perform these frame conversions automatically.

Composition of Transformations

Suppose we have two homogeneous transforms:

$$^{A}M_B \;=\; \begin{bmatrix} ^{A}\mathbf{R}_B & ^{A}\mathbf{A}_B \\ \mathbf{0} & 1 \end{bmatrix} \quad \text{and} \quad ^{B}M_C \;=\; \begin{bmatrix} ^{B}\mathbf{R}_C & ^{B}\mathbf{B}_C \\ \mathbf{0} & 1 \end{bmatrix}.$$

The product of these two matrices corresponds to the transformation from frame $\{C\}$ all the way to frame $\{A\}$. We write:

$$^{A}M_B \, ^{B}M_C \;=\; ^{A}M_C.$$

Resulting Form

Explicitly,

$$ {}^{A}M_B \;\; {}^{B}M_C \;=\; \begin{bmatrix} {}^{A}\mathbf{R}_B \; {}^{B}\mathbf{R}_C & {}^{A}\mathbf{A}_B \;+\; {}^{A}\mathbf{R}_B \; {}^{B}\mathbf{B}_C \\ \mathbf{0} & 1 \end{bmatrix}. $$

That is, the block-matrix multiplication yields:

$$ {}^{A}\mathbf{R}_C \;=\; {}^{A}\mathbf{R}_B \, {}^{B}\mathbf{R}_C \quad \text{and} \quad {}^{A}\mathbf{A}_C \;=\; {}^{A}\mathbf{A}_B \;+\; {}^{A}\mathbf{R}_B \; {}^{B}\mathbf{B}_C. $$

Inverse Composition

We might also consider composing $\left({}^{A}M_B\right)^{-1}$ with another transform ${}^{A}M_C$. From the figure:

$$ {}^{A}M_B^{-1} \;\; {}^{A}M_C \;=\; \begin{bmatrix} ({}^{A}\mathbf{R}_B)^T \; {}^{A}\mathbf{R}_C & ({}^{A}\mathbf{R}_B)^T \big({}^{A}\mathbf{A}_C - {}^{A}\mathbf{A}_B\big) \\ \mathbf{0} & 1 \end{bmatrix}. $$

Relation to Pinocchio SE3 Objects

In Pinocchio, each transform ${}^{A}M_B$ (from frame $\{B\}$ to $\{A\}$) is stored in an SE3 object. The composition of two such transforms is simply:

$$ \texttt{M\_AB * M\_BC} = \texttt{M\_AC}. $$

For example, in Python:

```python
import pinocchio as pin

# M_AB: from frame B to frame A
# M_BC: from frame C to frame B
M_AC = M_AB * M_BC # yields from frame C to frame A
```

SE(3) and Motion in Pinocchio

This document explains how Pinocchio handles the transformation of 6D motion vectors in Python. We will describe the relevant functions, their inputs and outputs, and how they correspond to the transformation matrices in Featherstone-style spatial algebra.

Equations

In Pinocchio, a *motion* is represented as a 6D vector, usually referred to as ν. By convention, Pinocchio stores the angular part in the first three components and the linear part in the last three components:

$$ \nu = \begin{bmatrix} \boldsymbol{v} \\ \boldsymbol{\omega} \end{bmatrix}, $$

where $\boldsymbol{\omega} \in \mathbb{R}^3$ is the angular velocity and $\mathbf{v} \in \mathbb{R}^3$ is the linear velocity.

A rigid-body transformation in Pinocchio is an instance of the class `pinocchio.SE3`, which represents an element of the SE(3) group (the space of 3D rotations and translations). Mathematically, an element of SE(3) is defined by:

$$ \mathbf{M} = \begin{pmatrix} \mathbf{R} & \mathbf{p} \\ \mathbf{0} & 1 \end{pmatrix}, $$

where $\mathbf{R} \in \mathrm{SO}(3)$ is a 3×3 rotation matrix and $\mathbf{p} \in \mathbb{R}^3$ is a translation vector.

In spatial algebra, a 6×6 *motion-transform matrix* associated with $\mathbf{M}$ applies to motion vectors and is often written as:

$$\mathbf{X}(\mathbf{M}) = \begin{bmatrix} \mathbf{R} & [\mathbf{p}] \times \mathbf{R} \\ \mathbf{0} & \mathbf{R} \end{bmatrix},$$

where $[\mathbf{p}]$ is the skew-symmetric matrix corresponding to the cross-product by $\mathbf{p}$. In Featherstone's notation, you might see it in block form as:

$$\mathbf{X}(\mathbf{M})^{-1} = \begin{bmatrix} \mathbf{R}^T & -\mathbf{R}^T[\mathbf{p}] \\ \mathbf{0} & \mathbf{R}^T \end{bmatrix}$$

depending on convention (signs can differ based on frames and definitions).

2. Pinocchio Functions

2.1 `SE3` Construction

You first create an SE(3) object (a rigid transform) in Python using:

```
import pinocchio
import numpy as np

R = np.eye(3) # 3x3 rotation matrix
p = np.array([0,0,0])# 3D translation vector
M = pinocchio.SE3(R, p)
```

- **Input**:
  - `R`: A 3×3 `numpy.ndarray` (rotation matrix).
  - `p`: A 3-element `numpy.ndarray` (translation).
- **Output**:
  - `M`: A `pinocchio.SE3` object containing the rotation $\mathbf{R}$ and translation $\mathbf{p}$.

2.2 `M.toActionMatrix()`

This function computes the 6×6 adjoint matrix that transforms a motion vector expressed in the *local* frame of `M` to the motion vector expressed in the *world* frame (or parent frame). Formally:

$$\mathbf{X} = M.\text{toActionMatrix}() \in \mathbb{R}^{6\times 6},$$

so that if $\nu_{\text{local}} \in \mathbb{R}^6$ is the motion in `M`'s local coordinate system, then

$$\nu_{\text{world}} = \mathbf{X}\,\nu_{\text{local}}.$$

- **Input**: (none) — it uses the internally stored $\mathbf{R}$ and $\mathbf{p}$ in `M`.
- **Output**: A `numpy.ndarray` of shape $(6, 6)$.

Example usage:

```
A = M.toActionMatrix() # 6x6 numpy array
```

2.3 `M.toActionMatrixInverse()`

This function returns the inverse of the 6×6 action matrix, i.e., it transforms a motion vector from the *world* frame back to the *local* frame:

$$\mathbf{X}^{-1} = M.\text{toActionMatrixInverse}() \quad \text{such that} \quad \nu_{\text{local}} = \mathbf{X}^{-1}\, \nu_{\text{world}}.$$

- **Input**: (none) — again uses `M`.
- **Output**: A `numpy.ndarray` of shape $(6, 6)$.

Example usage:

```
A_inv = M.toActionMatrixInverse()
```

2.4 `M.act(motion)`

Rather than explicitly forming the 6×6 matrix, Pinocchio also provides an operator that acts directly on 6D motion objects. If `v` is an instance of `pinocchio.Motion`, then

$$\text{act}(\mathbf{v}) = M.\text{act}(\mathbf{v}),$$

producing a new `pinocchio.Motion` that is the motion vector expressed in the parent frame. Internally, this is equivalent to multiplying by the 6×6 matrix from `M.toActionMatrix()`.

- **Input**:
  - `v`: A `pinocchio.Motion` object (6D motion).
- **Output**:
  - A new `pinocchio.Motion`, transformed into the coordinate frame of `M`.

Example usage:

```
v_local = pinocchio.Motion(np.array([0,0,0, 1,2,3]))
v_world = M.act(v_local)
```

2.5 `M.actInv(motion)`

Similarly, `M.actInv(motion)` applies the *inverse* transformation to a motion vector. Conceptually:

$$\text{actInv}(\mathbf{v}) = M.\text{actInv}(\mathbf{v}),$$

which is the motion in the local frame if $\mathbf{v}$ was originally expressed in the parent frame. Internally, it is equivalent to multiplying by `M.toActionMatrixInverse()`.

- **Input**:
  - `v`: A `pinocchio.Motion` object (6D motion in the parent/world frame).
- **Output**:
  - A new `pinocchio.Motion`, transformed into `M`'s local frame.

Example usage:

```
v_world = pinocchio.Motion(np.array([0,0,0, 1,2,3]))
v_local = M.actInv(v_world)
```

Example: Computing the 6×6 Matrix and Applying It

Below is a short snippet showing how to create an SE3 transform, get its 6×6 action matrix, and multiply it by a raw 6×1 motion vector:

```
import pinocchio
import numpy as np

# Example rotation and translation
R = np.eye(3)
p = np.array([1.0, 2.0, 3.0])

# Create the SE3 object
M = pinocchio.SE3(R, p)

# A 6D motion vector in M's local frame
v_local = np.array([0.1, 0.2, 0.3, 1.0, 2.0, 3.0]) # [omega, v]

# 1) Compute the 6x6 matrix
A = M.toActionMatrix()

# 2) Transform the motion to the world frame
v_world = A @ v_local

print("The␣6x6␣action␣matrix␣A:\n", A)
print("Motion␣in␣world␣frame:\n", v_world)
```

Alternatively, you can avoid explicit matrix multiplication by using Pinocchio's `Motion` class and the `.act()` method:

```
# Pinocchio motion object
v_local_motion = pinocchio.Motion(v_local) # pass a 6D numpy array

# Apply the transform directly
v_world_motion = M.act(v_local_motion)

print("v_world␣from␣act()␣=", v_world_motion.vector)
```

Spatial Force Vectors

A spatial force (wrench) in frame A can be written as:

$$^{A}\boldsymbol{\phi} \;=\; \begin{bmatrix} ^{A}\mathbf{f} \\ ^{A}\boldsymbol{\tau} \end{bmatrix},$$

where $^{A}\mathbf{f} \in \mathbb{R}^3$ is the linear force component and $^{A}\boldsymbol{\tau} \in \mathbb{R}^3$ is the torque (moment) component, both expressed in the coordinate frame $\{A\}$.

2. Transformation of a Force

If we wish to express the same physical force in another coordinate frame $\{B\}$, the spatial force transforms via a 6×6 matrix often denoted by $^{B}X_{A}^{*}$. The relationship is:

$$^{B}\boldsymbol{\phi} \;=\; ^{B}X_{A}^{*}\;\, ^{A}\boldsymbol{\phi}.$$

2.1 Derivation via Duality

One way to see this is by recognizing that the inner product $\boldsymbol{\phi}^\top \boldsymbol{\nu}$ (force dotted with velocity) is invariant under change of reference frame. For spatial motions,

$$^B\boldsymbol{\nu} \;=\; {}^BX_A \;\; {}^A\boldsymbol{\nu},$$

where ${}^BX_A$ is the motion-transform matrix. The invariance of $\boldsymbol{\phi}^\top \boldsymbol{\nu}$ implies:

$$({}^B\boldsymbol{\phi})^\top ({}^B\boldsymbol{\nu}) \;=\; ({}^A\boldsymbol{\phi})^\top ({}^A\boldsymbol{\nu}) \quad\Longrightarrow\quad {}^AX_B^* \;=\; \left({}^AX_B\right)^{-T}.$$

Notice that $\left({}^AX_B\right)^{-T} \neq \left({}^AX_B\right)$

Hence the force transform is the inverse transpose of the motion transform.

3. Block-Matrix Form

Given a homogeneous transform

$${}^AM_B \;=\; \begin{bmatrix} {}^A\mathbf{R}_B & {}^A\mathbf{A}_B \\ \mathbf{0} & 1 \end{bmatrix},$$

the motion-transform matrix is

$${}^AX_B \;=\; \begin{bmatrix} {}^A\mathbf{R}_B & \left({}^A\mathbf{A}_B\right)_\times {}^A\mathbf{R}_B \\ \mathbf{0} & {}^A\mathbf{R}_B \end{bmatrix},$$

and the corresponding *force*-transform matrix becomes

$${}^AX_B^* \;=\; \left({}^AX_B\right)^{-T} \;=\; \begin{bmatrix} {}^A\mathbf{R}_B & \mathbf{0} \\ \left({}^A\mathbf{A}_B\right)_\times {}^A\mathbf{R}_B & {}^A\mathbf{R}_B \end{bmatrix}.$$

Hence,

$${}^B\boldsymbol{\phi} \;=\; {}^BX_A^* \;\; {}^A\boldsymbol{\phi}, \quad \text{with} \quad {}^BX_A^* \;=\; \begin{bmatrix} ({}^A\mathbf{R}_B)^T & \mathbf{0} \\ -\,({}^A\mathbf{R}_B)^T\,({}^A\mathbf{A}_B)_\times & ({}^A\mathbf{R}_B)^T \end{bmatrix}.$$

3.1 Featherstone's (E, r) Notation

Following Featherstone, one sometimes sets

$$E \;=\; {}^B\mathbf{R}_A \;=\; ({}^A\mathbf{R}_B)^T p, \quad r \;=\; {}^A\mathbf{A}_B.$$

Then,

$${}^BX_A^* \;=\; \begin{bmatrix} E & 0 \\ -\,E\,r_\times & E \end{bmatrix}, \qquad {}^AX_B^* \;=\; \begin{bmatrix} E^T & 0 \\ r_\times\,E^T & E^T \end{bmatrix}.$$

Usage in Pinocchio

In Pinocchio's Python API, the `SE3` class also knows how to transform a `Force` object correctly. If `F_A` is a `pinocchio.Force` expressed in frame A, and `M_AB` is the `SE3` transform from B to A, then the force in frame B is:

```
F_B = M_AB.act(F_A)
```

Behind the scenes, Pinocchio applies the force-transform matrix ${}^BX_A^*$. Similarly,

```
F_A2 = M_AB.actInv(F_B)
```

applies the inverse of the dual transform.