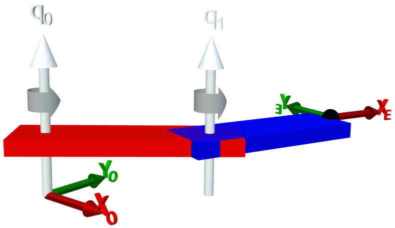
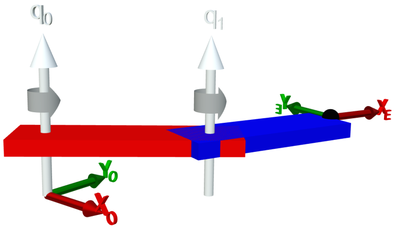
6ELEN018W - Applied Robotics
Lecture 4: Robot Motion - 3D Velocity Kinematics

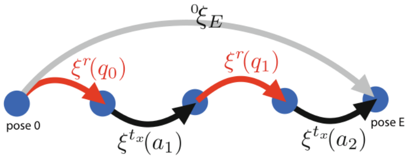
Dr Dimitris C. Dracopoulos

Previously - 2D Pose and Forward Kinematics

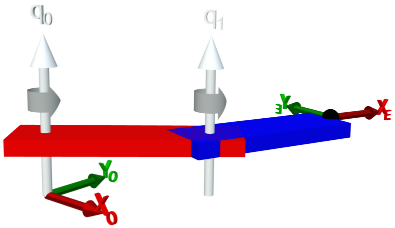


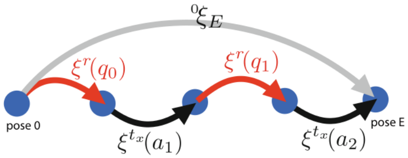
Previously - 2D Pose and Forward Kinematics





Previously - 2D Pose and Forward Kinematics





The pose of the end-effector is:

$$ {}^0\xi_E = \xi^r(q_0) \oplus \xi^{t_x}(a1) \oplus \xi^r(q_1) \oplus \xi^{t_x}(a2) \qquad (1) $$

Previously - 2D Pose and Forward Kinematics (cont'd)

In Python toolbox:

```
>>> a1 = 1
```

Previously - 2D Pose and Forward Kinematics (cont'd)

In Python toolbox:

```
>>> a1 = 1
>>> a2 = 1
```

Previously - 2D Pose and Forward Kinematics (cont'd)

In Python toolbox:

```
>>> a1 = 1
>>> a2 = 1

>>> e = ET2.R()*ET2.tx(a1)*ET2(R)*ET2.tx(a2)
```

Previously - 2D Pose and Forward Kinematics (cont'd)

In Python toolbox:

```
>>> a1 = 1
>>> a2 = 1

>>> e = ET2.R()*ET2.tx(a1)*ET2(R)*ET2.tx(a2)

>>> e.fkine(np.deg2rad([90, 30])).printline()
```

Previously - 2D Pose and Forward Kinematics (cont'd)

In Python toolbox:

```
>>> a1 = 1
>>> a2 = 1

>>> e = ET2.R()*ET2.tx(a1)*ET2(R)*ET2.tx(a2)

>>> e.fkine(np.deg2rad([90, 30])).printline()
```

Equivalently:

```
>>>  T = SE2.Rot(np.deg2rad(90)) * SE2.Tx(a1) \
         * SE2.Rot(np.deg2rad(30)) * SE2.Tx(a2)
```

Previously - 2D Pose and Forward Kinematics (cont'd)

In Python toolbox:

```
>>> a1 = 1
>>> a2 = 1

>>> e = ET2.R()*ET2.tx(a1)*ET2(R)*ET2.tx(a2)

>>> e.fkine(np.deg2rad([90, 30])).println()
```

Equivalently:

```
>>>  T = SE2.Rot(np.deg2rad(90)) * SE2.Tx(a1) \
         * SE2.Rot(np.deg2rad(30)) * SE2.Tx(a2)


>>> T.println()
```

Previously - 2D Pose and Forward Kinematics (cont'd)

In Python toolbox:

```
>>> a1 = 1
>>> a2 = 1

>>> e = ET2.R()*ET2.tx(a1)*ET2(R)*ET2.tx(a2)

>>> e.fkine(np.deg2rad([90, 30])).printline()
```

Equivalently:

```
>>>  T = SE2.Rot(np.deg2rad(90)) * SE2.Tx(a1) \
         * SE2.Rot(np.deg2rad(30)) * SE2.Tx(a2)


>>> T.printline()

>>> e.joints()
```

Pose and Forward Kinematics in 3D

Similar approach with the 2D case, apply successive transformations using the 3D homogeneous transformation matrices of size 4×4.

Pose and Forward Kinematics in 3D

Similar approach with the 2D case, apply successive transformations using the 3D homogeneous transformation matrices of size 4×4.

```
>>> a1 =1
```

Pose and Forward Kinematics in 3D

Similar approach with the 2D case, apply successive transformations using the 3D homogeneous transformation matrices of size 4×4.

```
>>> a1 =1

>>> a2 = 1
```

Pose and Forward Kinematics in 3D

Similar approach with the 2D case, apply successive transformations using the 3D homogeneous transformation matrices of size 4×4.

```
>>> a1 =1

>>> a2 = 1

>>> e = ET.Rz() * ET.Ry() \
      * ET.tz(a1) * ET.Ry() * ET.tz(a2) \
      * ET.Rz() * ET.Ry() * ET.Rz()
```

Pose and Forward Kinematics in 3D

Similar approach with the 2D case, apply successive transformations using the 3D homogeneous transformation matrices of size 4×4.

```
>>> a1 =1

>>> a2 = 1

>>> e = ET.Rz() * ET.Ry() \
        * ET.tz(a1) * ET.Ry() * ET.tz(a2) \
        * ET.Rz() * ET.Ry() * ET.Rz()

>>> e.n  # number of joints
```

Pose and Forward Kinematics in 3D

Similar approach with the 2D case, apply successive transformations using the 3D homogeneous transformation matrices of size 4×4.

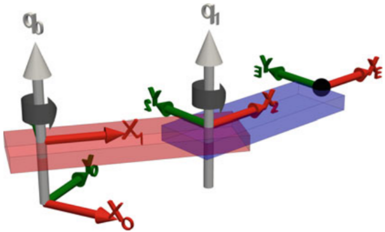
```
>>> a1 =1

>>> a2 = 1

>>> e = ET.Rz() * ET.Ry() \
       * ET.tz(a1) * ET.Ry() * ET.tz(a2) \
       * ET.Rz() * ET.Ry() * ET.Rz()

>>> e.n  # number of joints

>>> e.structure
```

Forward Kinematics as a Chain of Robot Links

A robot can be described as a sequence of links which are attached to joints.

Forward Kinematics as a Chain of Robot Links

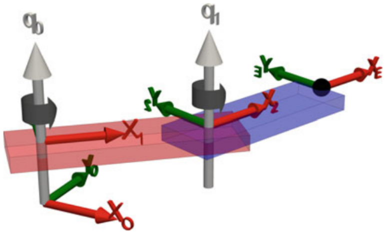
A robot can be described as a sequence of links which are attached to joints.
In 2D:



```
>>> a1=1; a2 =1;
```

Forward Kinematics as a Chain of Robot Links

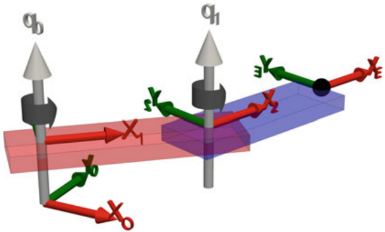
A robot can be described as a sequence of links which are attached to joints.
In 2D:



```
>>> a1=1; a2 =1;

>>> link1 = Link2(ET2.R(), name="link1")
```

Forward Kinematics as a Chain of Robot Links

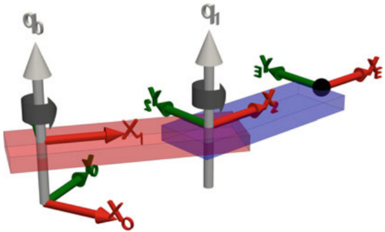
A robot can be described as a sequence of links which are attached to joints.
In 2D:



```
>>> a1=1; a2 =1;

>>> link1 = Link2(ET2.R(), name="link1")
>>> link2 = Link2(ET2.tx(a1)*ET2.R(), name="link2",parent=link1)
```

Forward Kinematics as a Chain of Robot Links

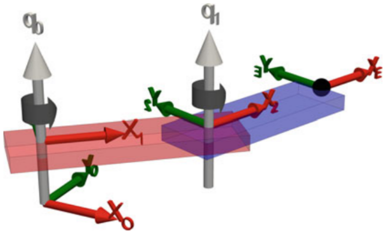
A robot can be described as a sequence of links which are attached to joints.
In 2D:



```
>>> a1=1; a2 =1;

>>> link1 = Link2(ET2.R(), name="link1")
>>> link2 = Link2(ET2.tx(a1)*ET2.R(), name="link2",parent=link1)
>>> link3 = Link2(ET2.tx(a2), name="link3", parent=link2)
```

Forward Kinematics as a Chain of Robot Links

A robot can be described as a sequence of links which are attached to joints.
In 2D:



```
>>> a1=1; a2 =1;

>>> link1 = Link2(ET2.R(), name="link1")
>>> link2 = Link2(ET2.tx(a1)*ET2.R(), name="link2",parent=link1)
>>> link3 = Link2(ET2.tx(a2), name="link3", parent=link2)

>>> robot = ERobot2([link1, link2, link3], name="my_robot")
```

Forward Kinematics as a Chain of Robot Links (cont'd)

Pose of the end-effector for a specific configuration of the joint angles:

```
>>> robot.fkine(np.deg2rad([30, 40])).printline()
```

Forward Kinematics as a Chain of Robot Links (cont'd)

Pose of the end-effector for a specific configuration of the joint angles:

```
>>> robot.fkine(np.deg2rad([30, 40])).printline()
```

Plot at this configuration:

```
robot.plot(np.deg2rad([30, 40]));
```

Forward Kinematics as a Chain of Robot Links (cont'd)

Pose of the end-effector for a specific configuration of the joint angles:

```
>>> robot.fkine(np.deg2rad([30, 40])).printline()
```

Plot at this configuration:

```
robot.plot(np.deg2rad([30, 40]));
```

Animation between an initial and a target configuration:

```
>>> q = np.array([np.linspace(0, pi, 100), \
    np.linspace(0, -2*pi, 100)]);
```

Forward Kinematics as a Chain of Robot Links (cont'd)

Pose of the end-effector for a specific configuration of the joint angles:

```
>>> robot.fkine(np.deg2rad([30, 40])).printline()
```

Plot at this configuration:

```
robot.plot(np.deg2rad([30, 40]));
```

Animation between an initial and a target configuration:

```
>>> q = np.array([np.linspace(0, pi, 100), \
    np.linspace(0, -2*pi, 100)]);

>>> q = q.T;  # take the transpose of q
```

Forward Kinematics as a Chain of Robot Links (cont'd)

Pose of the end-effector for a specific configuration of the joint angles:

```
>>> robot.fkine(np.deg2rad([30, 40])).printline()
```

Plot at this configuration:

```
robot.plot(np.deg2rad([30, 40]));
```

Animation between an initial and a target configuration:

```
>>> q = np.array([np.linspace(0, pi, 100), \
    np.linspace(0, -2*pi, 100)]);

>>> q = q.T;  # take the transpose of q

>> robot.plot(q)
```

Forward Kinematics as a Chain of Robot Links - 3D Case

Rotation about z, rotation about y, translation along z by a_1, rotation about y, translation along z by a_2, rotation about z, rotation about y, rotation about z.

```
e = ET.Rz()*ET.Ry()*ET.tz(a1)*ET.Ry()*ET.tz(a2)*ET.Rz() \
    *ET.Ry()*ET.Rz()*ET.Rx()
```

Pre-defined Robot Models in the Python Robotics Toolbox

```
>>> models.list(type="ETS")
```

Pre-defined Robot Models in the Python Robotics Toolbox

```
>>> models.list(type="ETS")

class              manufacturer           DoF  structure
----------------------------------------------------
Panda          |    Franka Emika            7     RRRRRRR
Frankie        |    Franka Emika, Omron     9     RPRRRRRRR
Puma560        |    Unimation               6     RRRRRR
Planar_Y       |                            6     RRRRRR
GenericSeven|       Jesse's Imagination     7     RRRRRRR
XYPanda        |    Franka Emika            9     PPRRRRRRR
```

Pre-defined Robot Models in the Python Robotics Toolbox

```
>>> models.list(type="ETS")

class               manufacturer          DoF  structure
-------------------------------------------------------------
Panda          |    Franka Emika           7    RRRRRRR
Frankie        |    Franka Emika, Omron    9    RPRRRRRRR
Puma560        |    Unimation              6    RRRRRR
Planar_Y       |                           6    RRRRRR
GenericSeven|       Jesse's Imagination    7    RRRRRRR
XYPanda        |    Franka Emika           9    PPRRRRRRR
```

To create an instance of a Puma560 robot:

Pre-defined Robot Models in the Python Robotics Toolbox

```
>>> models.list(type="ETS")
```

```
class              manufacturer          DoF  structure
-----------------------------------------------------
Panda         |    Franka Emika           7    RRRRRRR
Frankie       |    Franka Emika, Omron    9    RPRRRRRRR
Puma560       |    Unimation              6    RRRRRR
Planar_Y      |                           6    RRRRRR
GenericSeven|      Jesse's Imagination    7    RRRRRRR
XYPanda       |    Franka Emika           9    PPRRRRRRR
```

To create an instance of a Puma560 robot:

```
>>> p560 = models.ETS.Puma560()
```

Pre-defined Robot Models in the Python Robotics Toolbox (cont'd)

A new configuration can be added:

Pre-defined Robot Models in the Python Robotics Toolbox (cont'd)

A new configuration can be added:

```
>>> p560.addconfiguration("my_config", \
        [0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
```

Pre-defined Robot Models in the Python Robotics Toolbox (cont'd)

A new configuration can be added:

```
>>> p560.addconfiguration("my_config", \
        [0.1, 0.2, 0.3, 0.4, 0.5, 0.6])

# and accessed as a dictionary
```

Pre-defined Robot Models in the Python Robotics Toolbox (cont'd)

A new configuration can be added:

```
>>> p560.addconfiguration("my_config", \
        [0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
```

```
# and accessed as a dictionary
>>> p560.configs["my_config"]
```

Pre-defined Robot Models in the Python Robotics Toolbox (cont'd)

A new configuration can be added:

```
>>> p560.addconfiguration("my_config", \
        [0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
```

```
# and accessed as a dictionary
>>> p560.configs["my_config"]
```

The forward kinematics for a configuration can be computed:

Pre-defined Robot Models in the Python Robotics Toolbox (cont'd)

A new configuration can be added:

```
>>> p560.addconfiguration("my_config", \
        [0.1, 0.2, 0.3, 0.4, 0.5, 0.6])

# and accessed as a dictionary
>>> p560.configs["my_config"]
```

The forward kinematics for a configuration can be computed:

```
>>> p560.fkine(p560.qr)
```

Pre-defined Robot Models in the Python Robotics Toolbox (cont'd)

A new configuration can be added:

```
>>> p560.addconfiguration("my_config", \
        [0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
```

```
# and accessed as a dictionary
>>> p560.configs["my_config"]
```

The forward kinematics for a configuration can be computed:

```
>>> p560.fkine(p560.qr)
# print the pose in compact form
>>> p560.fkine(p560.qr).printline()
```

Pre-defined Robot Models in the Python Robotics Toolbox (cont'd)

A new configuration can be added:

```
>>> p560.addconfiguration("my_config", \
        [0.1, 0.2, 0.3, 0.4, 0.5, 0.6])

# and accessed as a dictionary
>>> p560.configs["my_config"]
```

The forward kinematics for a configuration can be computed:

```
>>> p560.fkine(p560.qr)
# print the pose in compact form
>>> p560.fkine(p560.qr).printline()
```

plotted in a configuration:

Pre-defined Robot Models in the Python Robotics Toolbox (cont'd)

A new configuration can be added:

```
>>> p560.addconfiguration("my_config", \
        [0.1, 0.2, 0.3, 0.4, 0.5, 0.6])

# and accessed as a dictionary
>>> p560.configs["my_config"]
```

The forward kinematics for a configuration can be computed:

```
>>> p560.fkine(p560.qr)
# print the pose in compact form
>>> p560.fkine(p560.qr).printline()
```

plotted in a configuration:

```
>>> p560.plot(p560.qr)
```

Motion in 3D

Previously covered: If the joints move at specific velocities, what is the velocity of the end-effector? (2D case)

- *Rate of change of position*: Speed (velocity): $\boldsymbol{v} = (\dot{x}, \dot{y}, \dot{z})$

Motion in 3D

Previously covered: If the joints move at specific velocities, what is the velocity of the end-effector? (2D case)

- *Rate of change of position*: Speed (velocity): $\boldsymbol{v} = (\dot{x}, \dot{y}, \dot{z})$
- *Rate of change of orientation*: Angular velocity: $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z) = (q_x, q_y, q_z)$

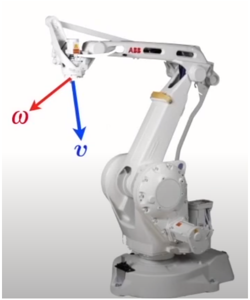
Motion in 3D

Previously covered: If the joints move at specific velocities, what is the velocity of the end-effector? (2D case)

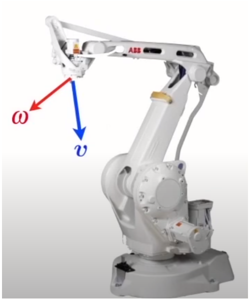
- *Rate of change of position*: Speed (velocity): $\boldsymbol{v} = (\dot{x}, \dot{y}, \dot{z})$
- *Rate of change of orientation*: Angular velocity: $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z) = (q_x, q_y, q_z)$

All of these are with reference to a specific coordinate frame (or simply the *reference coordinate frame*).

Translational and Rotational Motion of a Robot's End-Effector

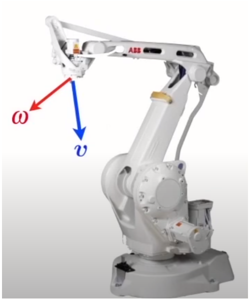


Translational and Rotational Motion of a Robot's End-Effector



The spatial velocity (twist) consists of:

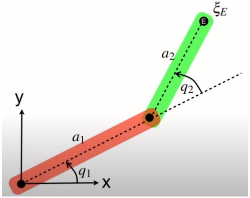
Translational and Rotational Motion of a Robot's End-Effector



The spatial velocity (twist) consists of:

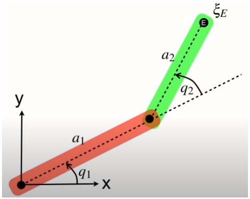
$$\boldsymbol{\nu} = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$$

Previously - End-Effector Velocity in a 2-Joint Robot (2D)



$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_1 cos(q_1) + a_2 cos(q_1 + q_2) \\ a_1 sin(q_1) + a_2 sin(q_1 + q_2) \end{pmatrix}$$

Previously - End-Effector Velocity in a 2-Joint Robot (2D)

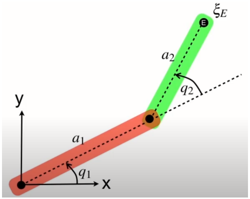


$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_1 cos(q_1) + a_2 cos(q_1 + q_2) \\ a_1 sin(q_1) + a_2 sin(q_1 + q_2) \end{pmatrix}$$

- If joint angles change over time (the robot moves):

$$q_1 = q_1(t), \quad q_2 = q_2(t)$$

Previously - End-Effector Velocity in a 2-Joint Robot (2D)



$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_1 cos(q_1) + a_2 cos(q_1 + q_2) \\ a_1 sin(q_1) + a_2 sin(q_1 + q_2) \end{pmatrix}$$

- If joint angles change over time (the robot moves):

$$q_1 = q_1(t), \quad q_2 = q_2(t)$$

- The velocity of the end-effector can be calculated by computing the derivative (using the chain rule):

$$\dot{x} = -a_1 \dot{q}_1 sin(q1) - a_2(\dot{q}_1 + \dot{q}_2) sin(q1 + q2)$$
$$\dot{y} = a_1 \dot{q}_1 cos(q_1) + a_2(\dot{q}_1 + \dot{q}_2) cos(q1 + q2)$$

Previously - End-Effector Velocity in a 2-Joint Robot (2D)

$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} -a_1 sin(q1) - a_2 sin(q1+q2) - a_2 sin(q1+q2) \\ a_1 cos(q_1) + a_2 cos(q1+q2) a_2 cos(q1+q2) \end{pmatrix} \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \end{pmatrix}$$

Previously - End-Effector Velocity in a 2-Joint Robot (2D)

$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} -a_1 sin(q1) - a_2 sin(q1+q2) - a_2 sin(q1+q2) \\ a_1 cos(q_1) + a_2 cos(q1+q2) a_2 cos(q1+q2) \end{pmatrix} \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \end{pmatrix}$$

The Jacobian $\boldsymbol{J}(\boldsymbol{q})$:

$$\boldsymbol{v} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}}$$

Jacobian Calculation in the Python Robotics Toolbox

```
>>> import sympy
```

Jacobian Calculation in the Python Robotics Toolbox

```
>>> import sympy

>>> a1, a2  = sympy.symbols("a1, a2")
```

Jacobian Calculation in the Python Robotics Toolbox

```
>>> import sympy

>>> a1, a2  = sympy.symbols("a1, a2")

>>> e = ERobot2(ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2))
```

Jacobian Calculation in the Python Robotics Toolbox

```
>>> import sympy

>>> a1, a2  = sympy.symbols("a1, a2")

>>> e = ERobot2(ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2))

>>> q = symbols("q:2") # sympy is already imported
```

Jacobian Calculation in the Python Robotics Toolbox

```
>>> import sympy
```

```
>>> a1, a2  = sympy.symbols("a1, a2")
```

```
>>> e = ERobot2(ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2))
```

```
>>> q = symbols("q:2") # sympy is already imported
```

The forward kinematics are calculated as:

```
>>> TE = e.fkine(q)
```

Jacobian Calculation in the Python Robotics Toolbox

```
>>> import sympy

>>> a1, a2  = sympy.symbols("a1, a2")

>>> e = ERobot2(ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2))

>>> q = symbols("q:2") # sympy is already imported
```

The forward kinematics are calculated as:

```
>>> TE = e.fkine(q)
```

Translation part, i.e location of end-effector $\boldsymbol{p} = (x, y)$:

Jacobian Calculation in the Python Robotics Toolbox

```
>>> import sympy

>>> a1, a2  = sympy.symbols("a1, a2")

>>> e = ERobot2(ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2))

>>> q = symbols("q:2") # sympy is already imported
```

The forward kinematics are calculated as:

```
>>> TE = e.fkine(q)
```

Translation part, i.e location of end-effector $\boldsymbol{p} = (x, y)$:

```
>>> p = TE.t
```

Jacobian Calculation in the Python Robotics Toolbox

```
>>> import sympy

>>> a1, a2  = sympy.symbols("a1, a2")

>>> e = ERobot2(ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2))

>>> q = symbols("q:2") # sympy is already imported
```

The forward kinematics are calculated as:

```
>>> TE = e.fkine(q)
```

Translation part, i.e location of end-effector $\boldsymbol{p} = (x, y)$:

```
>>> p = TE.t
```

The Jacobian is calculated:

Jacobian Calculation in the Python Robotics Toolbox

```
>>> import sympy

>>> a1, a2  = sympy.symbols("a1, a2")

>>> e = ERobot2(ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2))

>>> q = symbols("q:2") # sympy is already imported
```

The forward kinematics are calculated as:

```
>>> TE = e.fkine(q)
```

Translation part, i.e location of end-effector $\boldsymbol{p} = (x, y)$:

```
>>> p = TE.t
```

The Jacobian is calculated:

```
>>> J = Matrix(p).jacobian(q)
```

Jacobian Calculation in the Python Robotics Toolbox

```
>>> import sympy

>>> a1, a2  = sympy.symbols("a1, a2")

>>> e = ERobot2(ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2))

>>> q = symbols("q:2") # sympy is already imported
```

The forward kinematics are calculated as:

```
>>> TE = e.fkine(q)
```

Translation part, i.e location of end-effector $\boldsymbol{p} = (\mathrm{x}, \mathrm{y})$:

```
>>> p = TE.t
```

The Jacobian is calculated:

```
>>> J = Matrix(p).jacobian(q)
```

The velocity of the end-effector is calculated as:

$$\dot{\boldsymbol{p}} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{2}$$

General Form of the Forward Kinematics using the Jacobian

The derivative of the spatial velocity $\boldsymbol{\nu}$ of the end-effector can be written as:

$$\boldsymbol{\nu} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}}$$

where $\boldsymbol{J}(\boldsymbol{q})$ is an $M \times N$ matrix.

- $M = 6$ is the dimension of the task space (3 translational and 3 rotational velocity components)
- N is the number of robot joints

Calculating the Jacobian of Robots in the Python Robotics Toolbox

Call the `jacob0` method on any robot object in the toolbox.

Calculating the Jacobian of Robots in the Python Robotics Toolbox

Call the `jacob0` method on any robot object in the toolbox.

```
>>> p560 = models.ETS.Puma560()
```

Calculating the Jacobian of Robots in the Python Robotics Toolbox

Call the `jacob0` method on any robot object in the toolbox.

```
>>> p560 = models.ETS.Puma560()

>>> p560.jacob0(p560.qr)  # Jacobian for the qr configuration
```

Calculating the Jacobian of Robots in the Python Robotics Toolbox

Call the `jacob0` method on any robot object in the toolbox.

```
>>> p560 = models.ETS.Puma560()

>>> p560.jacob0(p560.qr)  # Jacobian for the qr configuration
```

Calculating the Jacobian of Robots in the Python Robotics Toolbox

Call the `jacob0` method on any robot object in the toolbox.

```
>>> p560 = models.ETS.Puma560()

>>> p560.jacob0(p560.qr)  # Jacobian for the qr configuration
```

- One column per joint.

Calculating the Jacobian of Robots in the Python Robotics Toolbox

Call the jacob0 method on any robot object in the toolbox.

```
>>> p560 = models.ETS.Puma560()

>>> p560.jacob0(p560.qr)  # Jacobian for the qr configuration
```

- One column per joint.

```
>>> p560.teach(p560.qr)
```

Velocity of a *n*-joint Robot Arm

Previous approach does not scale well for more joints. Even for a 6-joint robot it will take too much to do the calculations.

Velocity of a n-joint Robot Arm

Previous approach does not scale well for more joints. Even for a 6-joint robot it will take too much to do the calculations.
How to do this then?

- Relationship between a change of a single joint and the change in the end-effector.

Simple Numerical Calculation of Derivatives

In a numerical simulation, if we take small enough time steps, the derivative can be calculated as:

$$\frac{f(x_{t+1}) - f(x_t)}{\Delta t} \tag{3}$$

Simple Numerical Calculation of Derivatives

In a numerical simulation, if we take small enough time steps, the derivative can be calculated as:

$$\frac{f(x_{t+1}) - f(x_t)}{\Delta t} \tag{3}$$

Forward kinematics:

- An approximation of the forward kinematics changes as a function of changes of a single joint angle.

Simple Numerical Calculation of Derivatives

In a numerical simulation, if we take small enough time steps, the derivative can be calculated as:

$$\frac{f(x_{t+1}) - f(x_t)}{\Delta t} \tag{3}$$

Forward kinematics:

- An approximation of the forward kinematics changes as a function of changes of a single joint angle.
- The mathematical description of this can be a bit difficult, therefore it will be skipped.

Simple Numerical Calculation of Derivatives

In a numerical simulation, if we take small enough time steps, the derivative can be calculated as:

$$\frac{f(x_{t+1}) - f(x_t)}{\Delta t} \tag{3}$$

Forward kinematics:

- An approximation of the forward kinematics changes as a function of changes of a single joint angle.
- The mathematical description of this can be a bit difficult, therefore it will be skipped.
- One way to think about this, is that the total spatial velocity is the sum of the individual components due to a change in each angle (q_1, i.e column 1 of the Jacobian, q_2, i.e column 2 of the Jacobian, etc).

Simple Numerical Calculation of Derivatives

In a numerical simulation, if we take small enough time steps, the derivative can be calculated as:

$$\frac{f(x_{t+1}) - f(x_t)}{\Delta t} \tag{3}$$

Forward kinematics:

- An approximation of the forward kinematics changes as a function of changes of a single joint angle.
- The mathematical description of this can be a bit difficult, therefore it will be skipped.
- One way to think about this, is that the total spatial velocity is the sum of the individual components due to a change in each angle (q_1, i.e column 1 of the Jacobian, q_2, i.e column 2 of the Jacobian, etc).
- Use the `jacob0` method of the toolbox instead.

How to achieve a Specific End-Effector Spatial Velocity

What velocities the joints should have in order to achieve a specific end-effector spatial velocity?

How to achieve a Specific End-Effector Spatial Velocity

What velocities the joints should have in order to achieve a specific end-effector spatial velocity?
Forward kinematics:

$$\boldsymbol{\nu} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}}$$

How to achieve a Specific End-Effector Spatial Velocity

What velocities the joints should have in order to achieve a specific end-effector spatial velocity?
Forward kinematics:

$$\boldsymbol{\nu} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}}$$

Inverting the Jacobian:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q})^{-1}\boldsymbol{\nu}$$

How to achieve a Specific End-Effector Spatial Velocity

What velocities the joints should have in order to achieve a specific end-effector spatial velocity?
Forward kinematics:

$$\boldsymbol{\nu} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}}$$

Inverting the Jacobian:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q})^{-1}\boldsymbol{\nu}$$

For a 6-joint robot, $\boldsymbol{J}(\boldsymbol{q})$ is a 6×6 matrix, therefore its inverse can be calculated.

How to achieve a Specific End-Effector Spatial Velocity

What velocities the joints should have in order to achieve a specific end-effector spatial velocity?
Forward kinematics:

$$\boldsymbol{\nu} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}}$$

Inverting the Jacobian:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q})^{-1}\boldsymbol{\nu}$$

For a 6-joint robot, $\boldsymbol{J}(\boldsymbol{q})$ is a 6×6 matrix, therefore its inverse can be calculated.

- Unless the matrix is singular (the determinant is zero), in which case the inverse cannot be calculated!

Example: Inverting the Jacobian matrix for a Puma560 Robot

```
>>> p560 = models.ETS.Puma560()
```

Example: Inverting the Jacobian matrix for a Puma560 Robot

```
>>> p560 = models.ETS.Puma560()

>>> J = p560.jacob0(p560.qr)
```

Example: Inverting the Jacobian matrix for a Puma560 Robot

```
>>> p560 = models.ETS.Puma560()

>>> J = p560.jacob0(p560.qr)

>>> np.linalg.det(J)
```

Example: Inverting the Jacobian matrix for a Puma560 Robot

```
>>> p560 = models.ETS.Puma560()

>>> J = p560.jacob0(p560.qr)

>>> np.linalg.det(J)

>>> J = p560.jacob0(p560.qz)
```

Example: Inverting the Jacobian matrix for a Puma560 Robot

```
>>> p560 = models.ETS.Puma560()

>>> J = p560.jacob0(p560.qr)

>>> np.linalg.det(J)

>>> J = p560.jacob0(p560.qz)

# add a new configuration
>>> p560.addconfiguration("qn", [0, math.pi/4, math.pi, \
                                  0, math.pi/4, 0])
```

Example: Inverting the Jacobian matrix for a Puma560 Robot

```
>>> p560 = models.ETS.Puma560()

>>> J = p560.jacob0(p560.qr)

>>> np.linalg.det(J)

>>> J = p560.jacob0(p560.qz)

# add a new configuration
>>> p560.addconfiguration("qn", [0, math.pi/4, math.pi, \
                                 0, math.pi/4, 0])

>>> J = p560.jacob0(p560.configs["qn"])
```

Example: Inverting the Jacobian matrix for a Puma560 Robot

```
>>> p560 = models.ETS.Puma560()

>>> J = p560.jacob0(p560.qr)

>>> np.linalg.det(J)

>>> J = p560.jacob0(p560.qz)

# add a new configuration
>>> p560.addconfiguration("qn", [0, math.pi/4, math.pi, \
                                  0, math.pi/4, 0])

>>> J = p560.jacob0(p560.configs["qn"])

>>> np.linalg.det(J)
```

Example: Inverting the Jacobian matrix for a Puma560 Robot

```
>>> p560 = models.ETS.Puma560()

>>> J = p560.jacob0(p560.qr)

>>> np.linalg.det(J)

>>> J = p560.jacob0(p560.qz)

# add a new configuration
>>> p560.addconfiguration("qn", [0, math.pi/4, math.pi, \
                                  0, math.pi/4, 0])

>>> J = p560.jacob0(p560.configs["qn"])

>>> np.linalg.det(J)

>>> np.linalg.inv(J)
```

How to Control the Spatial Velocity of an End-Effector?

1. Choose the spatial velocity $\boldsymbol{\nu} = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$
2. Calculate the required joint velocities:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q})^{-1}\boldsymbol{\nu}$$

3. Move the joints at that speed using the actuators (control motors)

How to Control the Spatial Velocity of an End-Effector?

1. Choose the spatial velocity $\boldsymbol{\nu} = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$
2. Calculate the required joint velocities:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q})^{-1}\boldsymbol{\nu}$$

3. Move the joints at that speed using the actuators (control motors)
4. But after a short time, the angle $\boldsymbol{q}$ have changed, therefore the above calculation is not valid any more!

How to Control the Spatial Velocity of an End-Effector?

1. Choose the spatial velocity $\boldsymbol{\nu} = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$
2. Calculate the required joint velocities:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q})^{-1}\boldsymbol{\nu}$$

3. Move the joints at that speed using the actuators (control motors)
4. But after a short time, the angle $\boldsymbol{q}$ have changed, therefore the above calculation is not valid any more!
5. The Jacobian $\boldsymbol{J}(\boldsymbol{q})$ needs to be re-calculated.

How to Write a Program to Control the Spatial Velocity of the End-Effector

- Choose the spatial velocity $\boldsymbol{\nu} = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$

Repeat for ever:

1. Calculate the required joint velocities:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q_k})^{-1}\boldsymbol{\nu}$$

2. Move the joints at that speed using the actuators (control motors)

How to Write a Program to Control the Spatial Velocity of the End-Effector

- Choose the spatial velocity $\boldsymbol{\nu} = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$

Repeat for ever:

1. Calculate the required joint velocities:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q_k})^{-1}\boldsymbol{\nu}$$

2. Move the joints at that speed using the actuators (control motors)
3. Compute next joint angles: $\boldsymbol{q}_{k+1} = \boldsymbol{q}_k + \Delta_t \dot{\boldsymbol{q}}$
4. $k = k + 1$

Under-Actuated and Over-Actuated Robots

Under-actuated Robots:

- A robot with $N < 6$ joints is *under-actuated*.

Under-Actuated and Over-Actuated Robots

Under-actuated Robots:

- A robot with $N < 6$ joints is *under-actuated*.
- The Jacobian is not a square matrix therefore it cannot be inverted.

Under-Actuated and Over-Actuated Robots

Under-actuated Robots:

- A robot with $N < 6$ joints is *under-actuated*.
- The Jacobian is not a square matrix therefore it cannot be inverted.
- Remove from the spatial velocity components, the ones which cannot be controlled and invert the Jacobian.

Under-Actuated and Over-Actuated Robots

Under-actuated Robots:

- A robot with $N < 6$ joints is *under-actuated*.
- The Jacobian is not a square matrix therefore it cannot be inverted.
- Remove from the spatial velocity components, the ones which cannot be controlled and invert the Jacobian.

Over-actuated Robots:

- A robot with $N > 6$ joints is *under-actuated* (spare joints).

Under-Actuated and Over-Actuated Robots

Under-actuated Robots:

- A robot with $N < 6$ joints is *under-actuated*.
- The Jacobian is not a square matrix therefore it cannot be inverted.
- Remove from the spatial velocity components, the ones which cannot be controlled and invert the Jacobian.

Over-actuated Robots:

- A robot with $N > 6$ joints is *under-actuated* (spare joints).
- The Jacobian is not a square matrix therefore it cannot be inverted.

Under-Actuated and Over-Actuated Robots

Under-actuated Robots:

- A robot with $N < 6$ joints is *under-actuated*.
- The Jacobian is not a square matrix therefore it cannot be inverted.
- Remove from the spatial velocity components, the ones which cannot be controlled and invert the Jacobian.

Over-actuated Robots:

- A robot with $N > 6$ joints is *under-actuated* (spare joints).
- The Jacobian is not a square matrix therefore it cannot be inverted.
- A matrix called *pseudo-inverse* can be computed $\dot{\boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q})^{+}\boldsymbol{\nu}$.

Under-Actuated and Over-Actuated Robots

Under-actuated Robots:

- A robot with $N < 6$ joints is *under-actuated*.
- The Jacobian is not a square matrix therefore it cannot be inverted.
- Remove from the spatial velocity components, the ones which cannot be controlled and invert the Jacobian.

Over-actuated Robots:

- A robot with $N > 6$ joints is *under-actuated* (spare joints).
- The Jacobian is not a square matrix therefore it cannot be inverted.
- A matrix called *pseudo-inverse* can be computed $\dot{\boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q})^{+}\boldsymbol{\nu}$.

$$\boldsymbol{J}^{+} = (\boldsymbol{J}^{T}\boldsymbol{J})^{-1}\boldsymbol{J}^{T}$$