This lecture will be recorded

064-0026-00L: COMPAS II

Introduction to Computational Methods for Digital Fabrication in Architecture

slides + code

*https://***dfab.link/fs2022**

TODAY

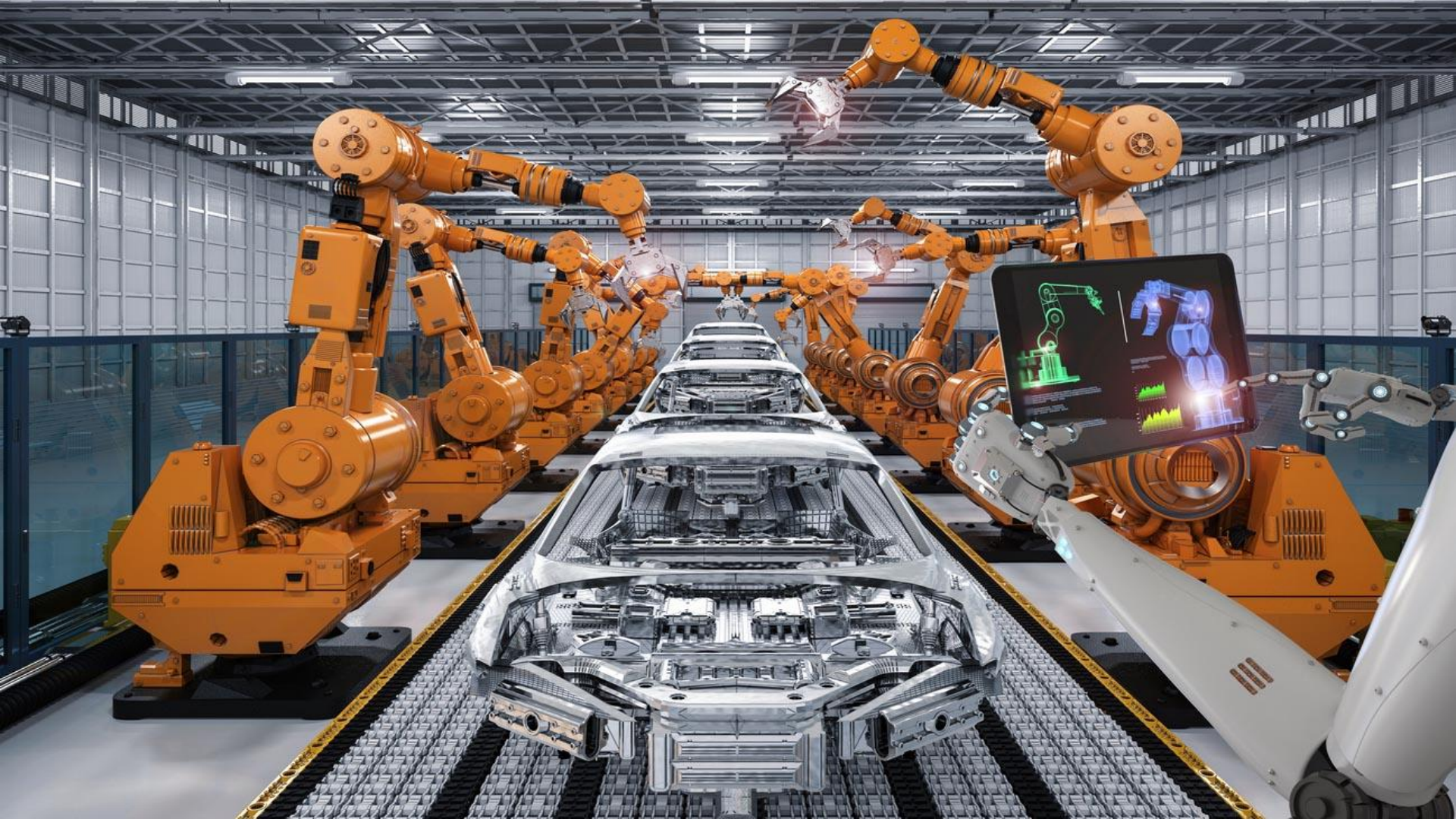
industrial robots

robotic fundamentals

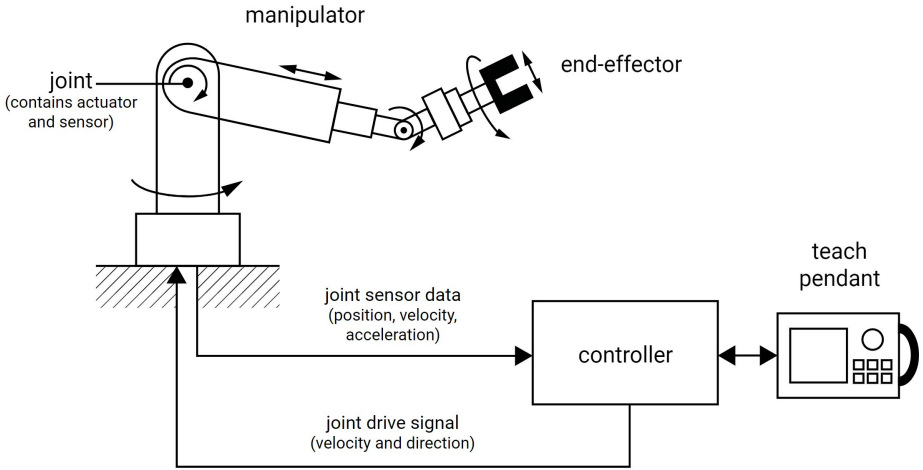
joint space and kinematics

Understand a robot as a **chain of transformations**

```
"""There are several ways to construct a `Frame`.
"""
from compas.geometry import Point
from compas.geometry import Vector
from compas.geometry import Frame
from compas.geometry import Plane

# Frame autocorrects axes to be orthonormal
F = Frame(Point(1, 0, 0), Vector(-0.45, 0.1, 0.3), Vector(1, 0, 0))

F = Frame([1, 0, 0], [-0.45, 0.1, 0.3], [1, 0, 0])

F = Frame.from_points([1, 1, 1], [2, 3, 6], [6, 3, 0])
F = Frame.from_plane(Plane([0, 0, 0], [0.5, 0.2, 0.1]))
F = Frame.from_euler_angles([0.5, 1., 0.2])
F = Frame.worldXY()
```

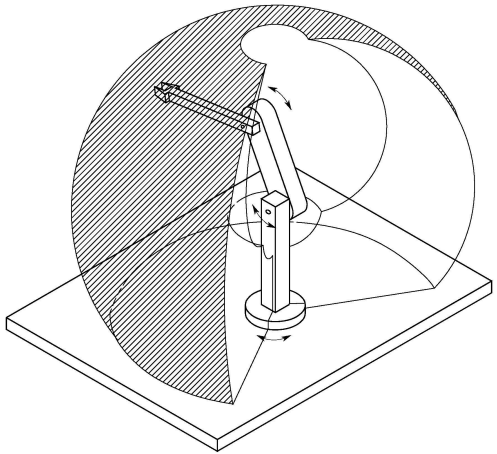


Basic robot components

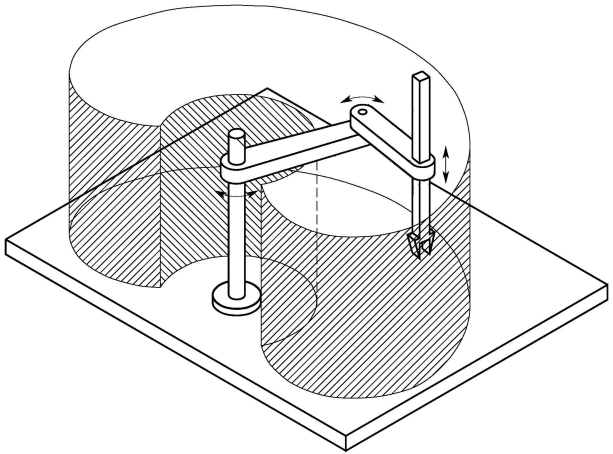


- manipulator
- end-effector
- actuators
- sensors
- controller
- teach pendant

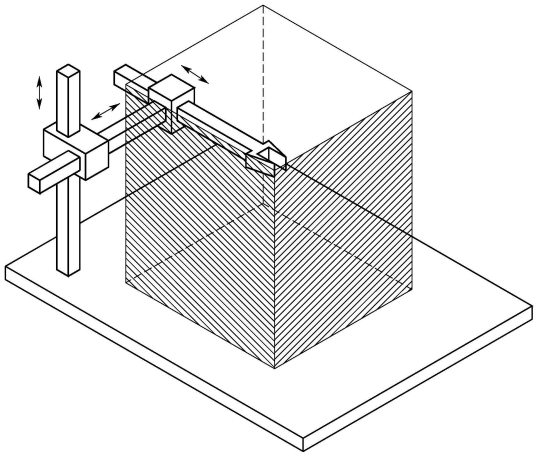
Robot workspace



Anthropomorphic manipulator

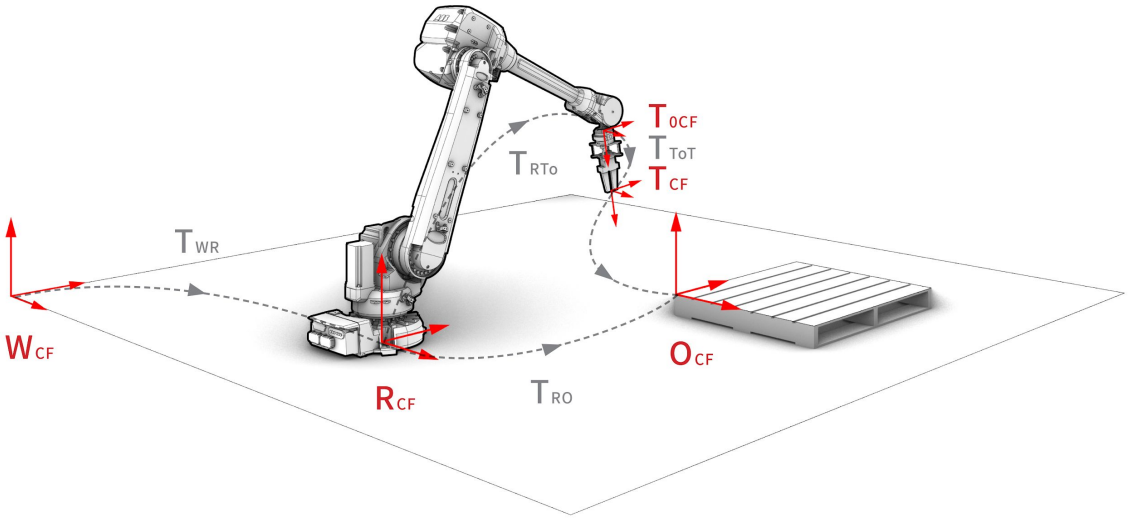


SCARA manipulator



Cartesian manipulator

Robot coordinate frames



World (WCF)

Robot (RCF)

Tool0 (T0CF)

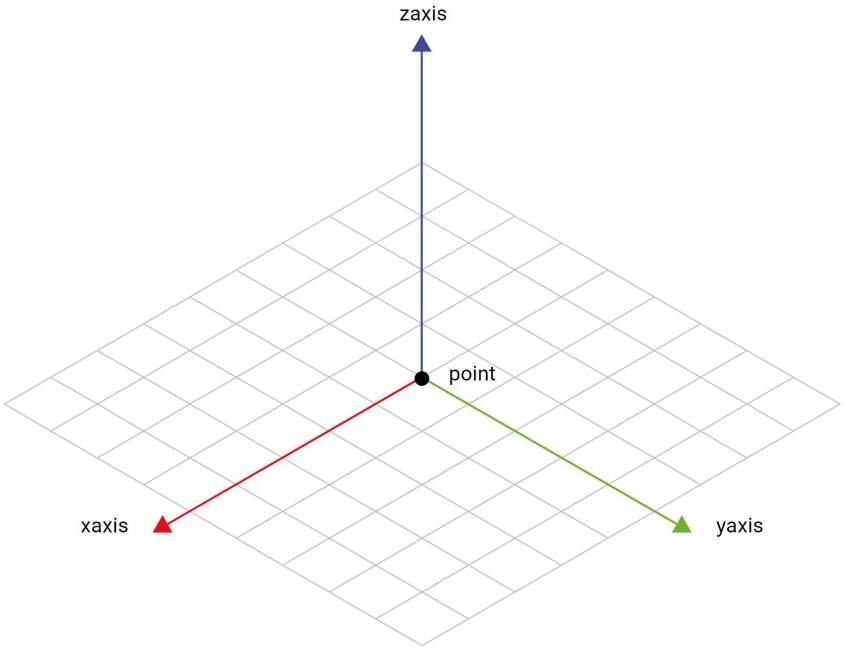
Tool (TCF)

Object (OCF)

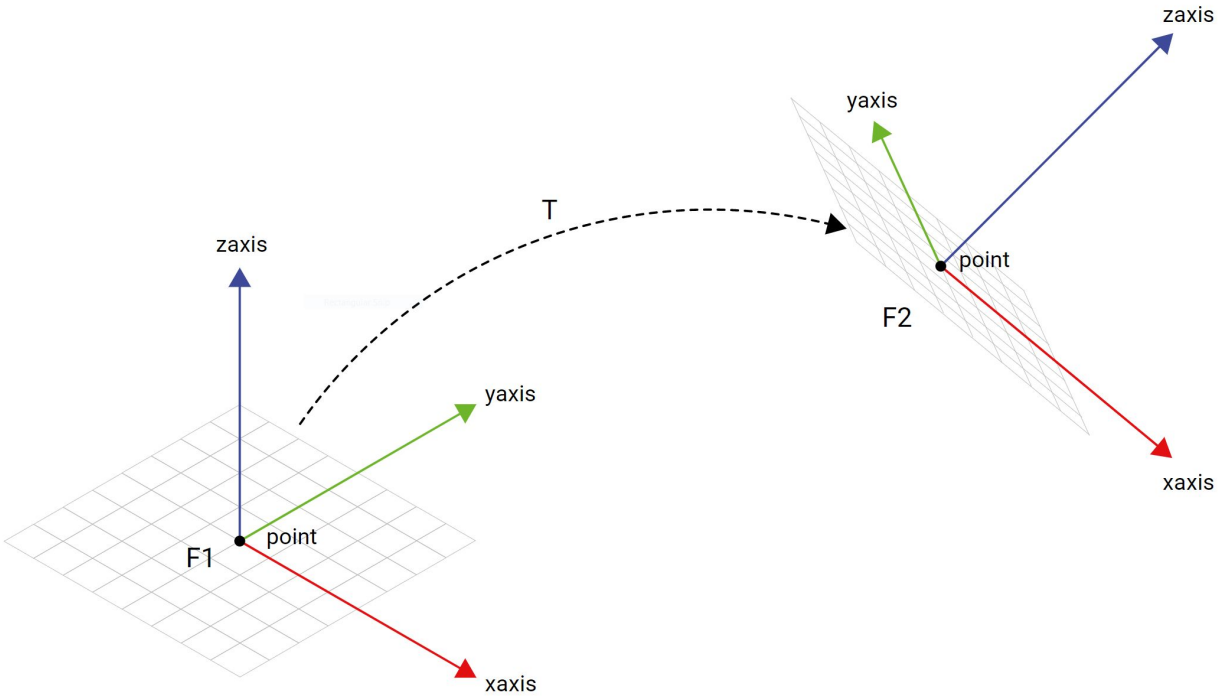
Frame

A frame is defined by a base point and two orthonormal base vectors (xaxis, yaxis), which specify the normal (zaxis).

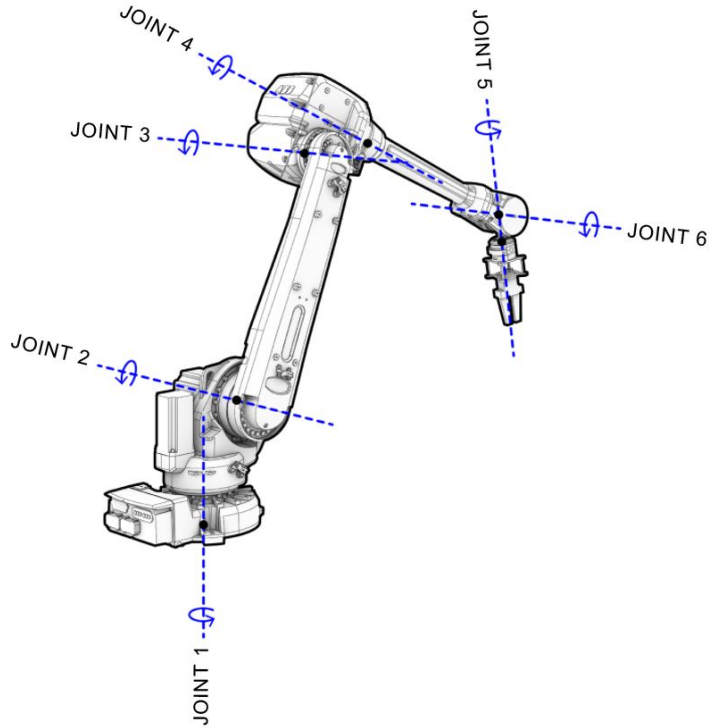
It describes location and orientation in a (right-handed) cartesian coordinate system.



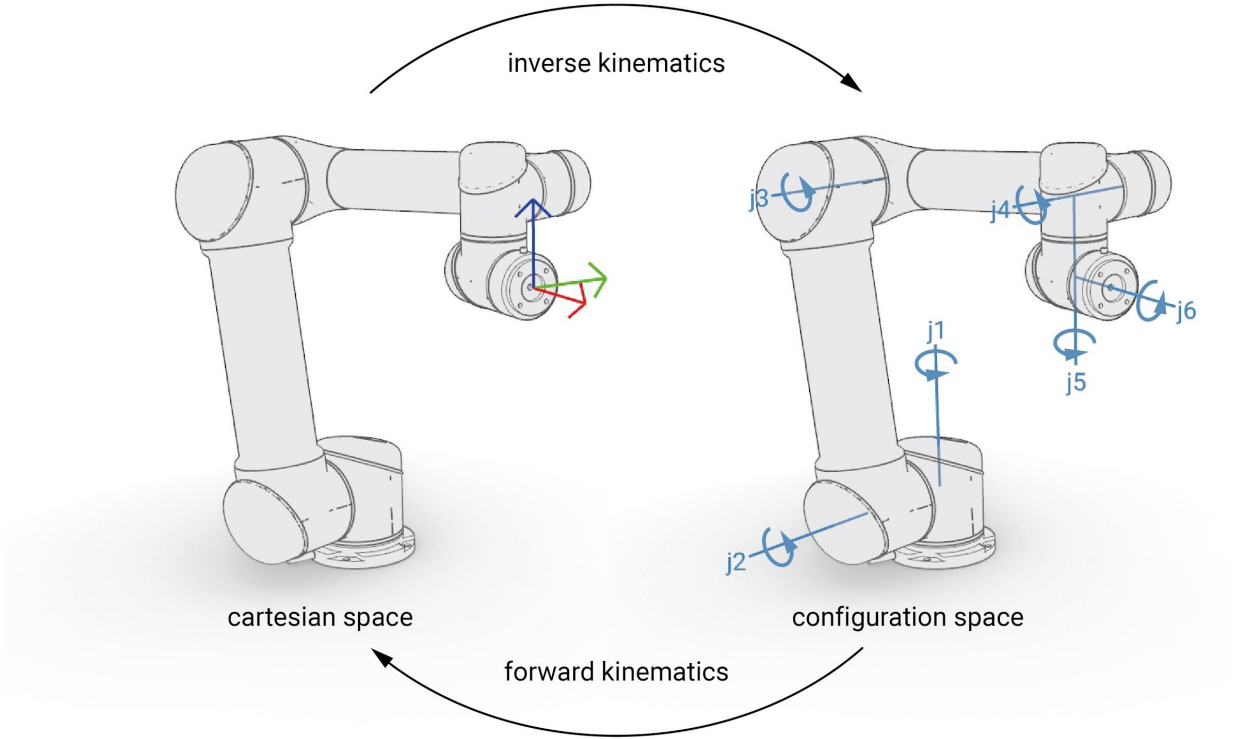
Transformation



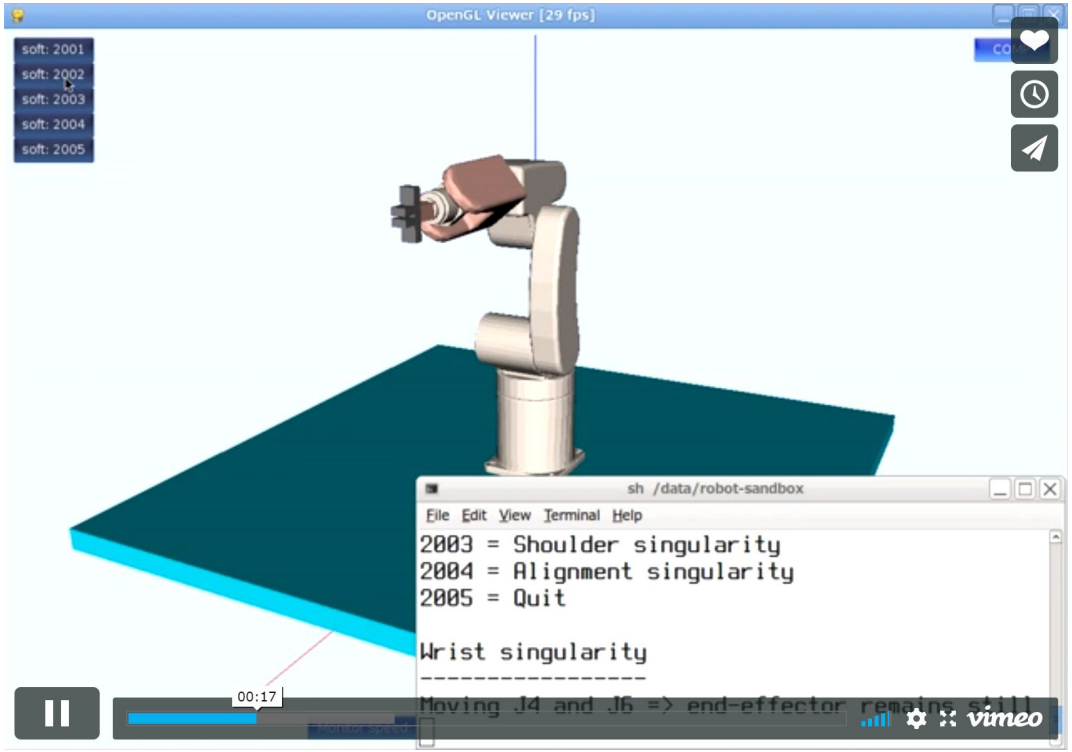
Configuration



Robot positioning

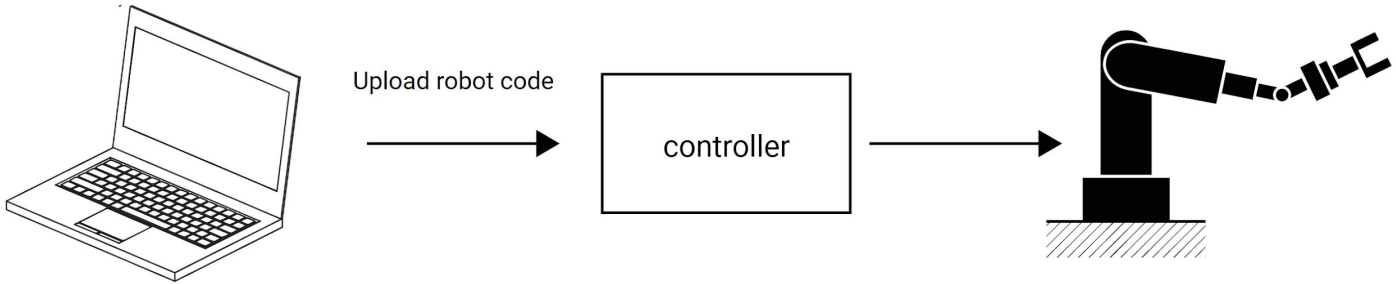


Robot singularities



Robot control

- Offline control
- Online control
  a. real-time
  b. non-real-time

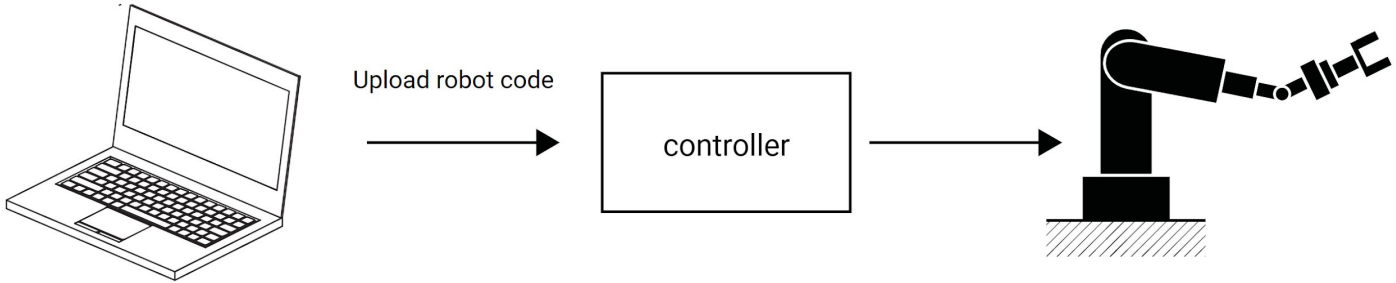


Robot control

- Offline control
- Online control
  a. real-time
  b. non-real-time

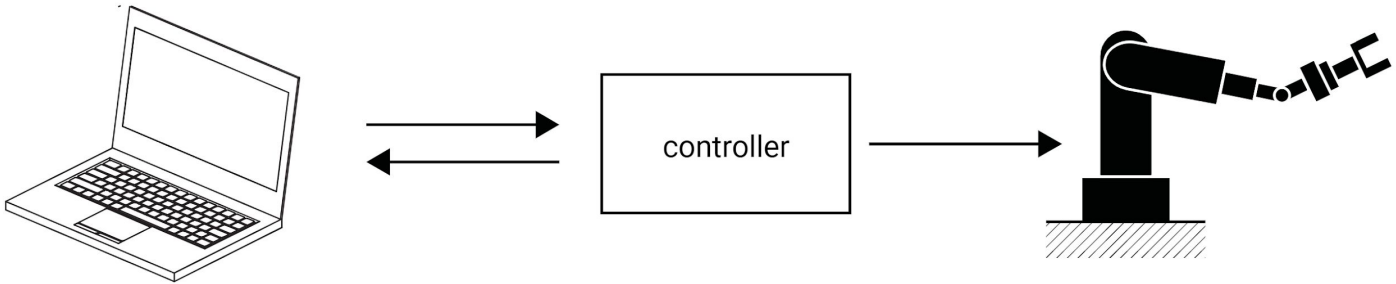
Vendor-specific language and tools:

Vendor	Language	Software
ABB	RAPID	ABB RobotStudio
Staubli	VAL3	Staubli Robotics Suite
UR	URScript	URSim
KUKA	KRL	KUKA WorkVisual

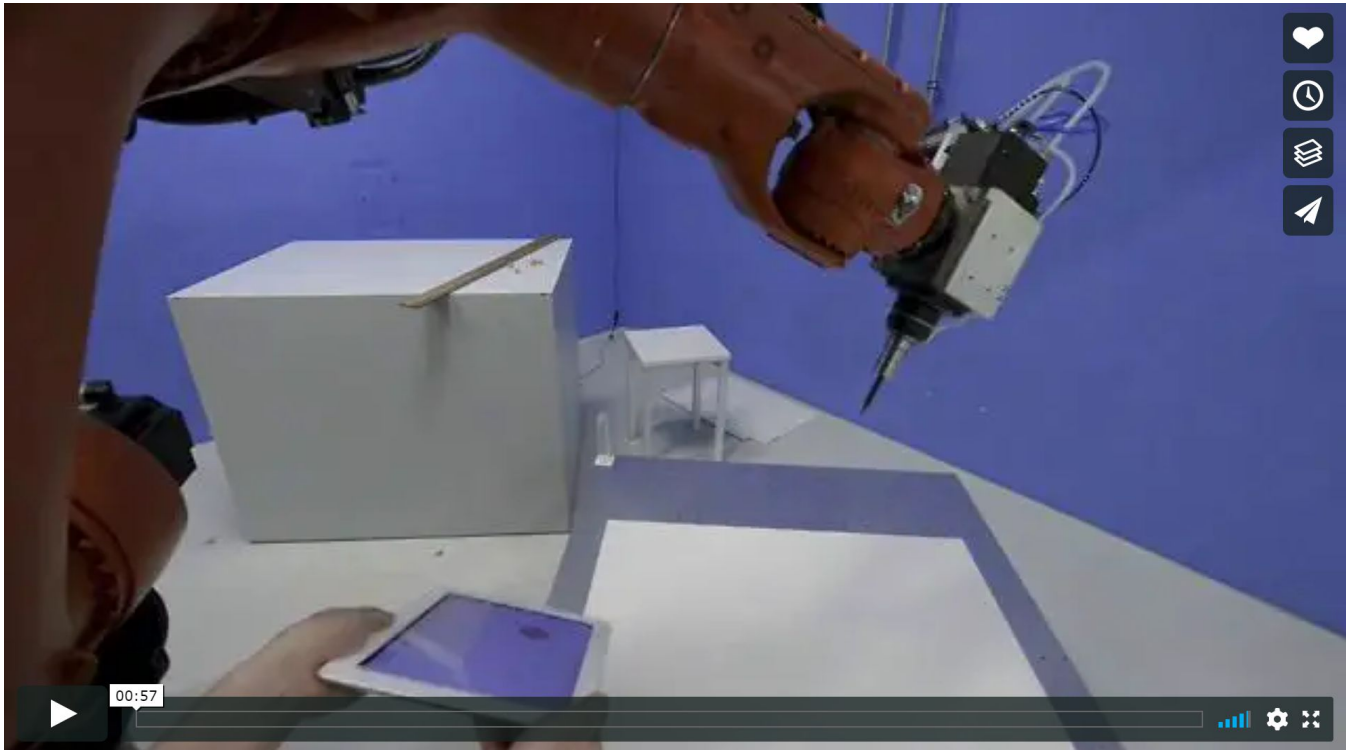


Robot control

- Offline control
- Online control
  a. real-time
  b. non-real-time

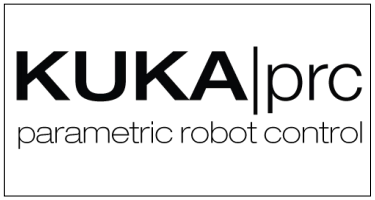


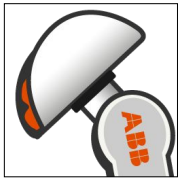
Example: online non-real-time control

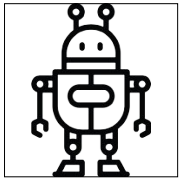


Robot control

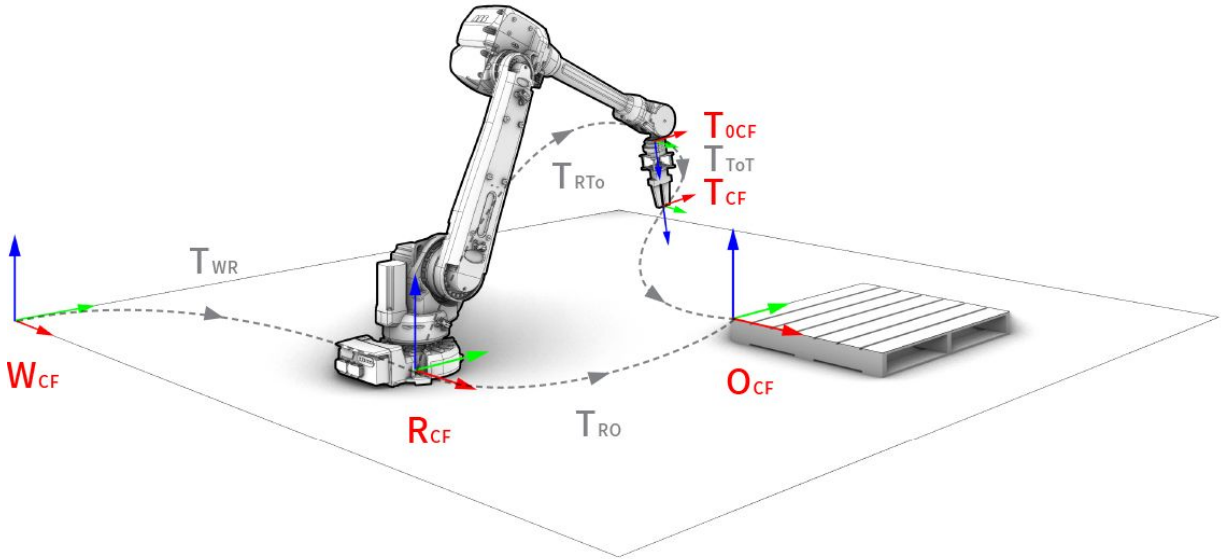
Control robots directly from within parametric design environment (Grasshopper, Dynamo, ...)







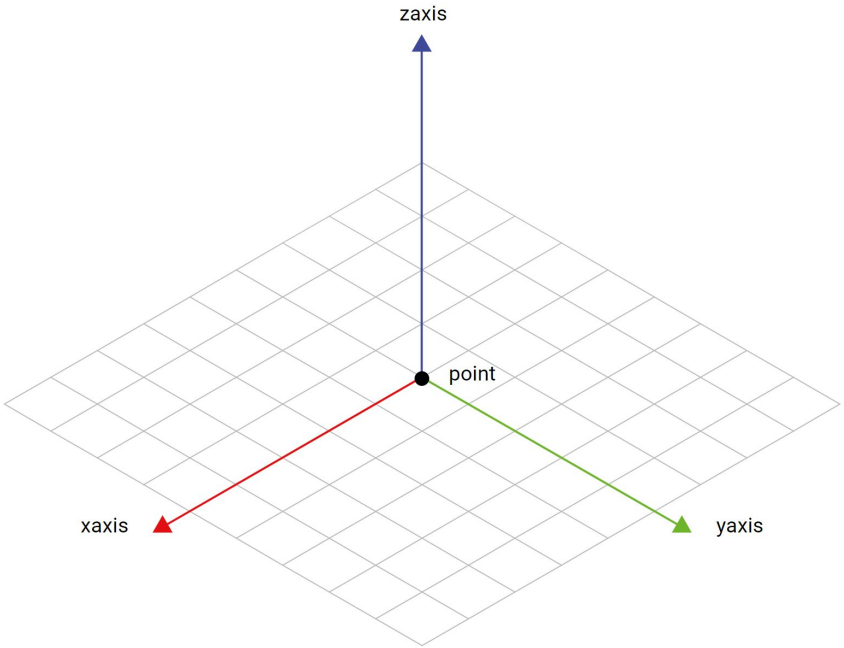
Frame and Transformation



Frame

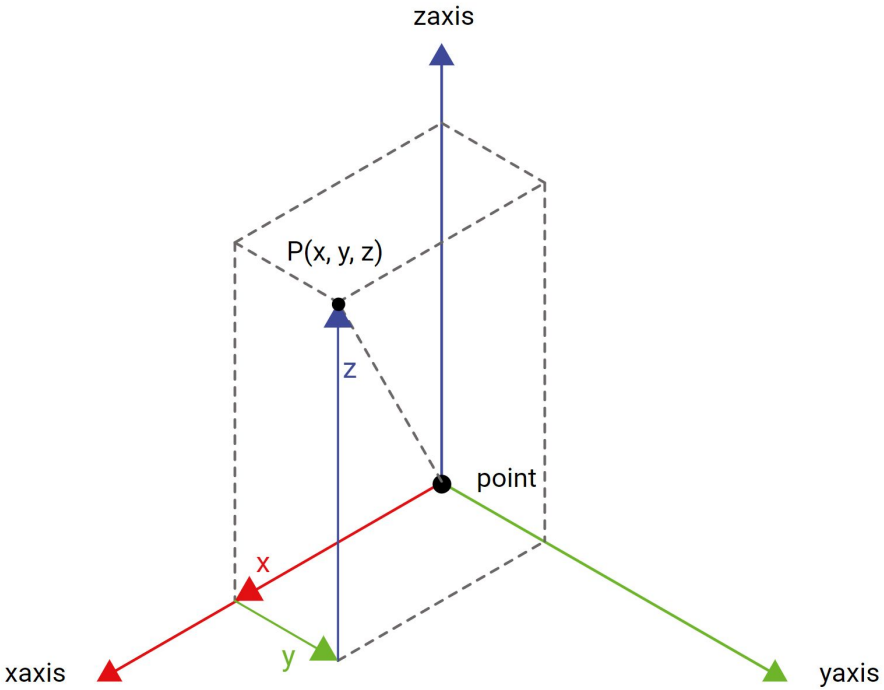
A frame is defined by a base point and two orthonormal base vectors (xaxis, yaxis), which specify the normal (zaxis).

It describes location and orientation in a (right-handed) cartesian coordinate system.



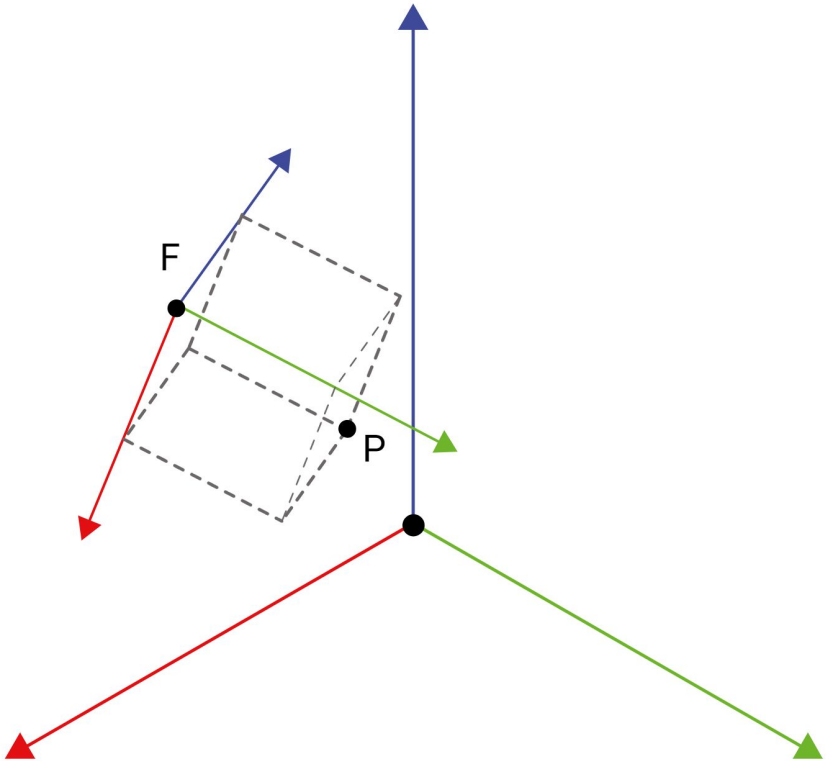
```python
"""There are several ways to construct a `Frame`.
"""
from compas.geometry import Point
from compas.geometry import Vector
from compas.geometry import Frame
from compas.geometry import Plane

# Frame autocorrects axes to be orthonormal
F = Frame(Point(1, 0, 0), Vector(-0.45, 0.1, 0.3), Vector(1, 0, 0))

F = Frame([1, 0, 0], [-0.45, 0.1, 0.3], [1, 0, 0])

F = Frame.from_points([1, 1, 1], [2, 3, 6], [6, 3, 0])
F = Frame.from_plane(Plane([0, 0, 0], [0.5, 0.2, 0.1]))
F = Frame.from_euler_angles([0.5, 1., 0.2])
F = Frame.worldXY()
```


Frame as a cartesian coordinate system



```
"""Example: 'point in frame'
"""
from compas.geometry import *

point = Point(146.00, 150.00, 161.50)
xaxis = Vector(0.9767, 0.0010, -0.214)
yaxis = Vector(0.1002, 0.8818, 0.4609)

F = Frame(point, xaxis, yaxis)  # coordinate system F
P = Point(35., 35., 35.)  # point in F (local coordinates)

P_ = F.to_world_coordinates(P)  # point in global (world) coordinates
print("The point's world coordinates: {!r}".format(P_))

P2 = F.to_local_coordinates(P_)
print("The point's local coordinates: {!r}".format(P2))  # should equal P
print(allclose(P2, P))
```


Frame in frame



```
"""Example: 'frame in frame'
"""
from compas.geometry import *

point = Point(146.00, 150.00, 161.50)
xaxis = Vector(0.9767, 0.0010, -0.214)
yaxis = Vector(0.1002, 0.8818, 0.4609)
F0 = Frame(point, xaxis, yaxis) # coordinate system F0

point = Point(35., 35., 35.)
xaxis = Vector(0.604, 0.430, 0.671)
yaxis = Vector(-0.631, 0.772, 0.074)
f_lcf = Frame(point, xaxis, yaxis) # frame f_lcf in F0 (local coordinates)

# frame in global (world) coordinate system
f_wcf = F0.to_world_coordinates(f_lcf)
print(f_wcf)

f_lcf2 = F0.to_local_coordinates(f_wcf) # world coords back to local coords
print(f_lcf2) # should equal f_lcf
print(f_lcf == f_lcf2)
```

Further information

- https://en.wikipedia.org/wiki/Frame_of_reference
- https://en.wikipedia.org/wiki/Cartesian_coordinate_system

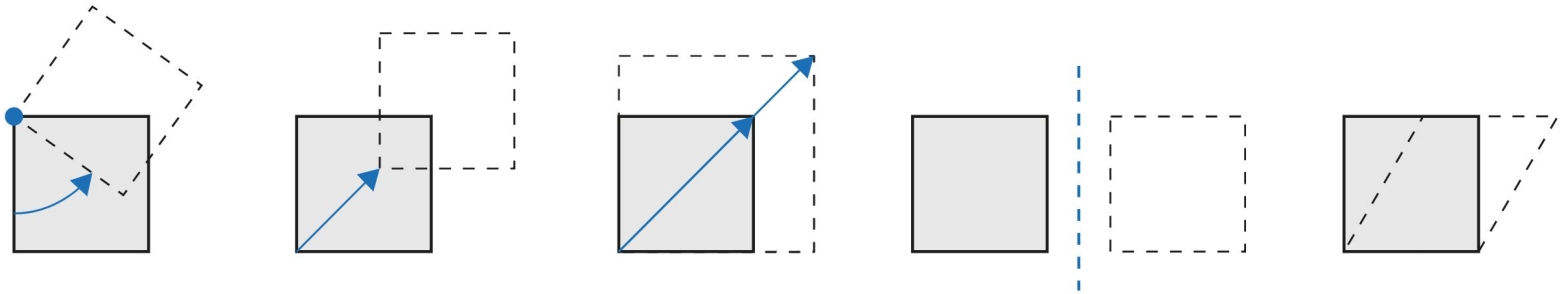
Transformation

Transformations allow you to move, rotate, and scale objects. They are also used to express the coordinates of an object in one frame in the coordinate system of another.

Transformation

Transformations allow you to move, rotate, and scale objects. They are also used to express the coordinates of an object in one frame in the coordinate system of another.

Most transformations preserve the parallel relationship among the parts of the geometry. For example collinear points remain collinear after the transformation. Also points on one plane stay coplanar after transformation. This type of transformation is called an *affine transformation* and concerns transformations such as `Rotation`, `Translation`, `Scale`, `Reflection`, `Shear` and orthogonal and parallel `Projection`. Only perspective `Projection` is not *affine*.



A generic transformation matrix looks like this:

$$\mathbf{T} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Homogenization

This is one of the reasons for distinguishing between points and vectors!

To transform points and vectors, i.e. multiply them with the transformation matrix, we need to homogenize them first. This means representing a 3-vector (x, y, z) as a 4-vector (x, y, z, 1) for points, or (x, y, z, 0) for vectors.

Points:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Vectors:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 0 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

Different types of transformations may take more specific forms:

The `Translation` just uses the first 3 coefficients of the 4th column,

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & a_{14} \\ 0 & 1 & 0 & a_{24} \\ 0 & 0 & 1 & a_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Different types of transformations may take more specific forms:

The `Translation` just uses the first 3 coefficients of the 4th column,

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & a_{14} \\ 0 & 1 & 0 & a_{24} \\ 0 & 0 & 1 & a_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Different types of transformations may take more specific forms:

The `Translation` just uses the first 3 coefficients of the 4th column,

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & a_{14} \\ 0 & 1 & 0 & a_{24} \\ 0 & 0 & 1 & a_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Point:

$$\begin{bmatrix} 1 & 0 & 0 & a_{14} \\ 0 & 1 & 0 & a_{24} \\ 0 & 0 & 1 & a_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + a_{14} \\ y + a_{24} \\ z + a_{34} \\ 1 \end{bmatrix}$$

Different types of transformations may take more specific forms:

The `Translation` just uses the first 3 coefficients of the 4th column,

Vector:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & a_{14} \\ 0 & 1 & 0 & a_{24} \\ 0 & 0 & 1 & a_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & a_{14} \\ 0 & 1 & 0 & a_{24} \\ 0 & 0 & 1 & a_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

`Scale` uses only the first 3 values of the diagonal,

$$\mathbf{T} = \begin{bmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and `Rotation` just uses the upper left 3x3 coefficients.

$$\mathbf{T} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```python
"""Transformation examples
"""
from compas.geometry import *

axis, angle = [0.2, 0.4, 0.1], 0.3
R = Rotation.from_axis_and_angle(axis, angle)
print("Rotation:\n", R)

translation_vector = [5, 3, 1]
T = Translation.from_vector(translation_vector)
print("Translation:\n", T)

scale_factors = [0.1, 0.3, 0.4]
S = Scale.from_factors(scale_factors)
print("Scale:\n", S)

point, normal = [0.3, 0.2, 1], [0.3, 0.1, 1]
R = Reflection.from_plane((point, normal))
print("Reflection:\n", R)
```


Inverse transformation

For most of the transformations we work with, there exists an inverse transformation T^{-1} which "undos" the effect of T.

$T \times T^{-1} = I$, with I as identity matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
"""Example: Transform a point and invert the transformation
"""
from compas.geometry import *
from math import pi

p = Point(3, 4, 5)
T = Rotation.from_axis_and_angle([2, 2, 2], pi/4)

p.transform(T)  # transform Point p with T

Tinv = T.inverse()  # create inverse Transformation to T

p.transform(Tinv)  # transform Point p with inverse Transformation

# check if p has the same values as in the beginning
print(p)  # == (3, 4, 5)

# Btw, what is the result of multiplying T with Tinv?
print(T * Tinv)
```

Concatenation

The concatenation of several Transformations are simple matrix multiplications. But matrices are not commutative ($\mathbf{A}\times\mathbf{B}\neq\mathbf{B}\times\mathbf{A}$), so it is important to consider the order of multiplication.

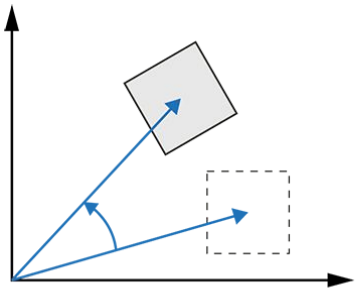
$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

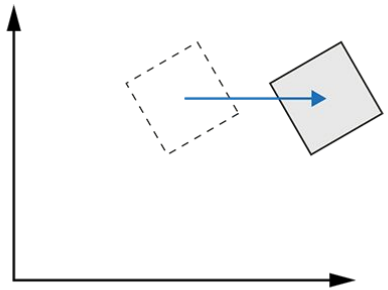
Concatenation

The concatenation of several Transformations are simple matrix multiplications. But matrices are not commutative ($\mathbf{A}\times\mathbf{B}\neq\mathbf{B}\times\mathbf{A}$), so it is important to consider the order of multiplication.

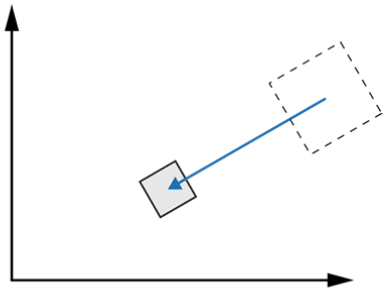
If you transform an object first with transformation $\mathbf{A}$, then with transformation $\mathbf{B}$, followed by transformation $\mathbf{C}$, you get the same result as transforming the object with only one transformation $\mathbf{M}$, which is calculated by $\mathbf{M}=\mathbf{C}\times\mathbf{B}\times\mathbf{A}$. (Transformations applied from right to left.)



Rotation



Translation



Scale

```
"""Example: Multiply transformations
"""
from compas.geometry import *

p = Point(1, 1, 1)
A = Translation.from_vector([1, 2, 3])  # create Translation
B = Rotation.from_axis_and_angle([-0.8, 0.35, 0.5], 2.2)  # create Rotation
C = Scale.from_factors([0.1, 0.3, 0.4])  # create Scale

# Transform p1 one by one
p1 = p.copy()
p1.transform(A)
p1.transform(B)
p1.transform(C)

# Transform with only one concatenated matrix
p2 = p.copy()
p2.transform(C * B * A)
```

```
"""Example: multiplication
"""
import math
from compas.geometry import Translation
from compas.geometry import Rotation
from compas.geometry import Scale

R = Rotation.from_axis_and_angle([0, 0, 1], math.radians(30))
T = Translation.from_vector([2, 0, 0])
S = Scale.from_factors([0.5] * 3)
X1 = S * T * R
X2 = R * T * S

print(X1)
print(X2)
print(X1 == X2)
```

Decompose transformations

Any generic transformation can be decomposed into a product of transformations of the form:

$$\text{Perspective} \times \text{Translation} \times \text{Rotation} \times \text{Shear} \times \text{Scale}$$

```
"""Example: Decompose transformations
"""
from compas.geometry import *

A = Scale.from_factors([0.1, 0.3, 0.4])  # create Scale
B = Rotation.from_axis_and_angle([-0.8, 0.35, 0.5], 2.2)  # create Rotation
C = Translation.from_vector([1, 2, 3])  # create Translation

# Concatenate transformations
M = C * B * A

# A matrix can also be decomposed into its components of Scale,
# Shear, Rotation, Translation and Perspective
Sc, Sh, R, T, P = M.decomposed()
# Check, must be all `True`
print(A == Sc)
print(B == R)
print(C == T)
print(P * T * R * Sh * Sc == M)
```

Question

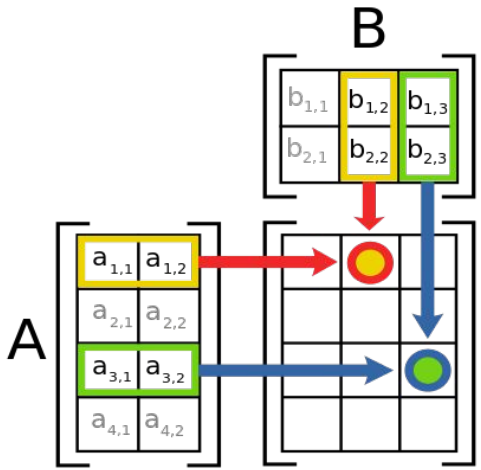
To transform a point, you simply multiply the point with the transformation matrix. But a transformation matrix is 4x4 and vectors and points are 3x1, so how can they be multiplied?

Matrix multiplication

If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times p$ matrix, the matrix product $\mathbf{C}=\mathbf{AB}$ is defined to be a $m \times p$ matrix.

$$\underset{4\times 2 \text{ matrix}}{\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix}} \underset{2\times 3 \text{ matrix}}{\begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}} = \underset{4\times 3 \text{ matrix}}{\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \end{bmatrix}}$$

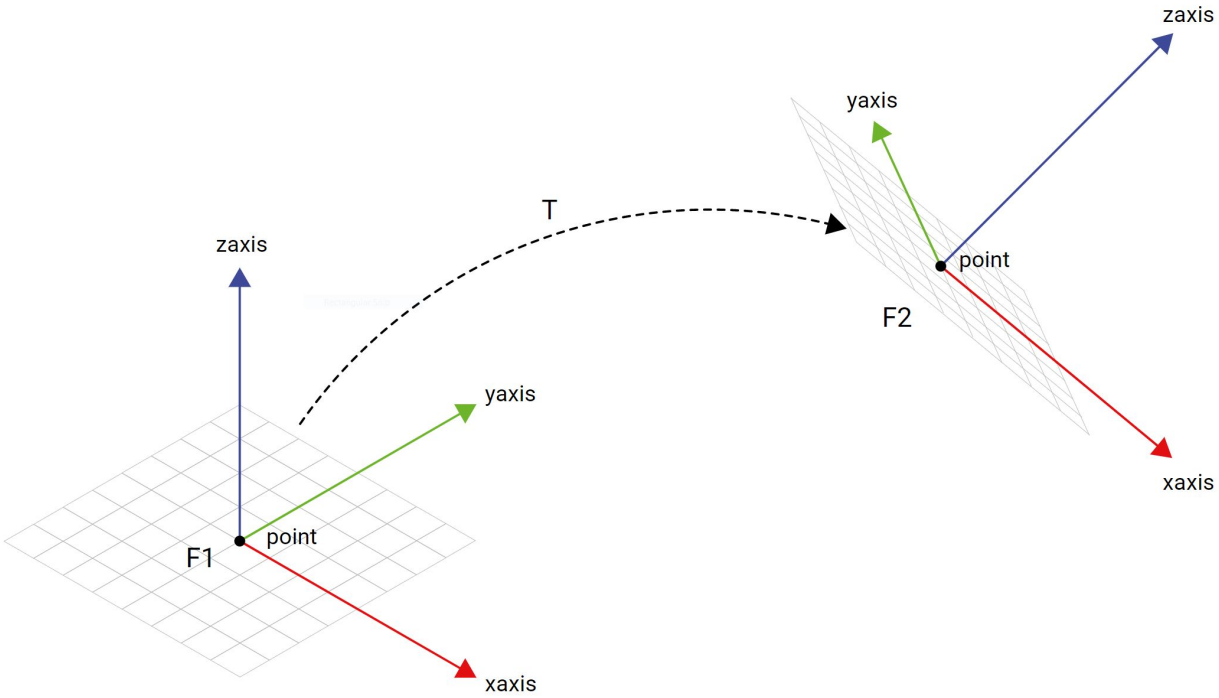
$$x_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$x_{33} = a_{31}b_{13} + a_{32}b_{23}$$



https://en.wikipedia.org/wiki/Matrix_multiplication

```python
"""Example: transform point and vector
"""
from compas.geometry import Point, Vector, Rotation

# create Rotation around point with axis and angle
point = Point(1.0, 0.0, 0.0)
axis, angle = [-0.248, -0.786, -0.566], 2.78
R = Rotation.from_axis_and_angle(axis, angle, point=point)

# apply Transformation to Point
p = Point(1, 1, 1)
p.transform(R)

# apply Transformation to Vector
v = Vector(1, 1, 1)
v.transform(R)

# should not be the same!
print(p)
print(v)
```

Frame and Transformation

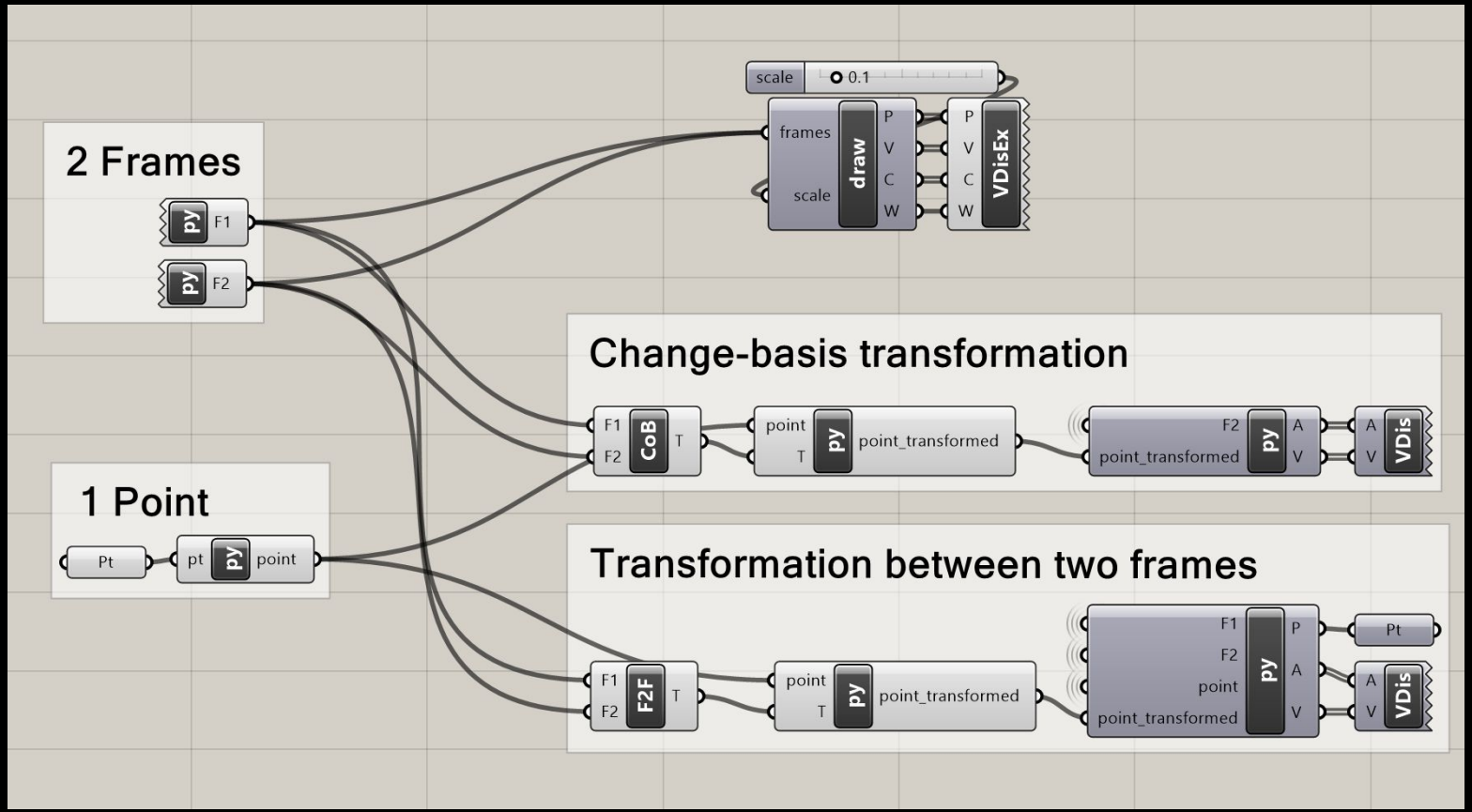


Change-basis transformation and transformation between frames

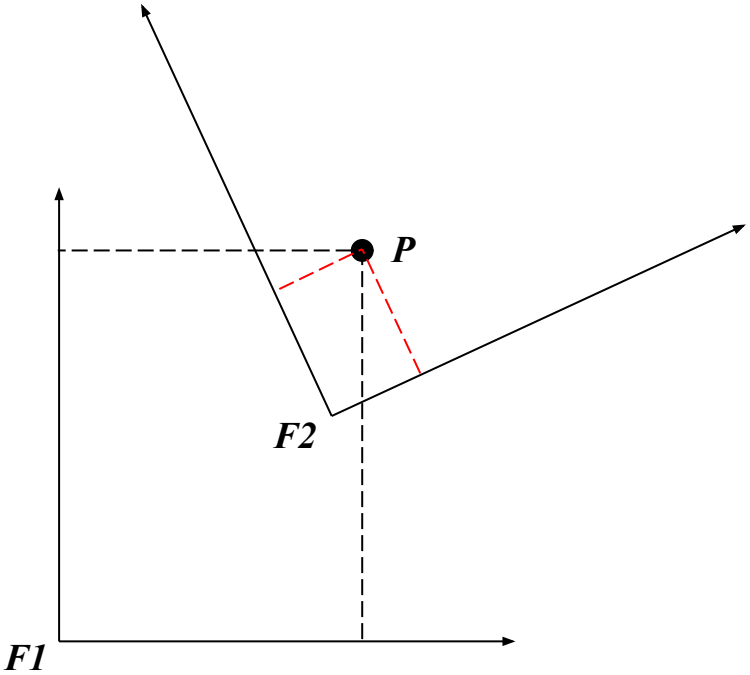
`Transformation.from_change_of_basis(f1, f2)` and

`Transformation.from_frame_to_frame(f1, f2)`

A change-basis transformation allows to remap geometry of one coordinate system in another, i.e. represent the same coordinates in two different frames, whereas the `Transformation` between two frames allows to transform geometry from one coordinate system into the other one.

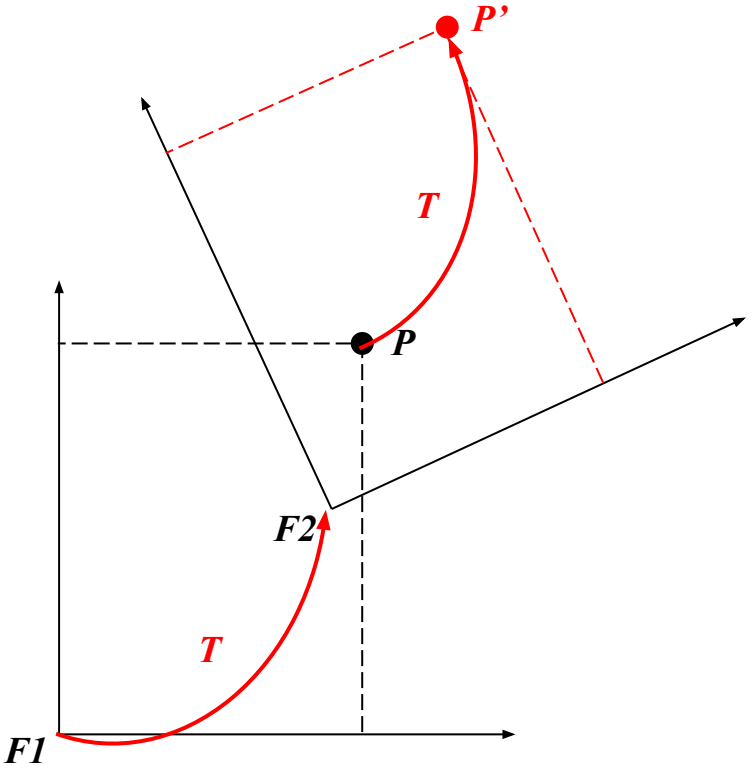

2 Frames
py F1
py F2
1 Point
Pt
pt py point
scale 0.1
frames
scale
draw
P V C W
P V C W
VDisEx
Change-basis transformation
F1 F2 CoB T
point T py point_transformed
F2 point_transformed py A V
A V VDis
Transformation between two frames
F1 F2 F2F T
point T py point_transformed
F1 F2 point point_transformed py P A V
Pt
A V VDis

Change-basis transformation



```python
"""Change-basis transformation.
"""
from compas.geometry import Point
from compas.geometry import Frame
from compas.geometry import Transformation

F1 = Frame.worldXY()
F2 = Frame([1.5, 1, 0], [0.68, 0.68, 0.27], [-0.67, 0.73, -0.15])
P = Point(2, 2, 2)  # local point in F1

# change-basis transformation between two frames F1 and F2.
T = Transformation.from_change_of_basis(F1, F2)

# Represent geometry (point P) in another coordinate frame
print(P.transformed(T))
# You can also use the following
print(F2.to_local_coordinates(P))
```


Transformation between frames



```
"""Transformation between two frames.
"""
from compas.geometry import Point
from compas.geometry import Frame
from compas.geometry import Transformation

F1 = Frame.worldXY()
F2 = Frame([1.5, 1, 0], [0.68, 0.68, 0.27], [-0.67, 0.73, -0.15])
P = Point(2, 2, 2)  # local point in F1

# transformation between 2 frames F1, F2
T = Transformation.from_frame_to_frame(F1, F2)

# Transform geometry (=point P) into another coordinate frame
print(P.transformed(T))
```

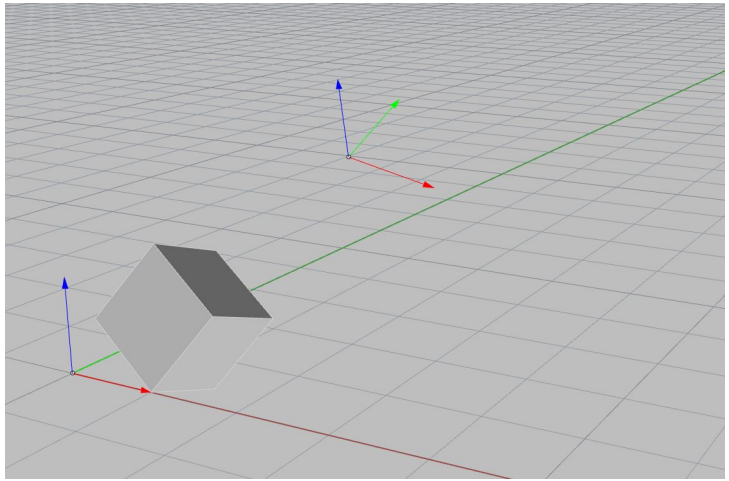
Question

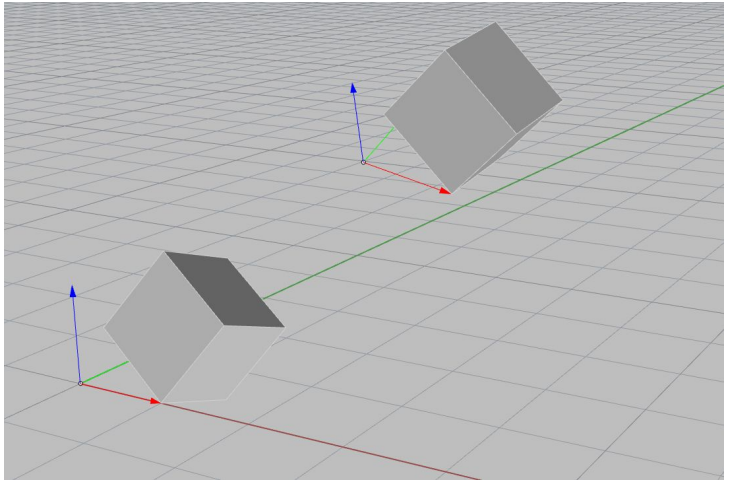
If you have an object at a location in the world other than the origin, how do you bring it back to the `worldXY` frame ?

Example

1. Bring a box from the world coordinate system into another coordinate system

2. Draw frames and boxes in Rhino





```
"""Example: Bring a box from the world coordinate system into another coordinate
system and view in Rhino.
"""
from compas.artists import Artist
from compas.geometry import Box
from compas.geometry import Frame
from compas.geometry import Transformation

# Box in the world coordinate system
frame = Frame([1, 0, 0], [-0.45, 0.1, 0.3], [1, 0, 0])
width, length, height = 1, 1, 1
box = Box(frame, width, length, height)

# Frame F representing a coordinate system
F = Frame([2, 2, 2], [0.978, 0.010, -0.210], [0.090, 0.882, 0.463])

# Get transformation between frame F and world
# T = ...
# Apply transformation on box.
# box_transformed = ...
```


`transform()` vs `transformed()`

- Make a copy of box and transform it (don't change box):

  ```
  box_transformed = box.transformed(T)
  ```

- Transform box (returns `None`):

  ```
  box.transform(T)
  ```

`transform()` vs `transformed()`

- Make a copy of box and transform it (don't change box):

  ```
  box_transformed = box.transformed(T)
  ```

- Transform box (returns `None`):

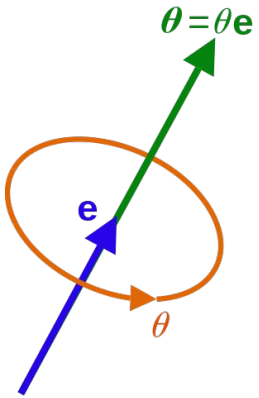
  ```
  box.transform(T)
  ```

Depending on the size of the object, copying takes time. So consider carefully if you really need a copy.

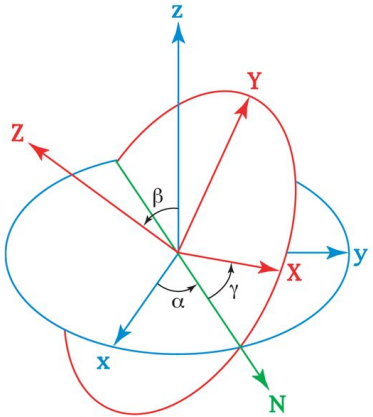
Rotation and orientation

A `Rotation` is a circular movement of an object around a point of rotation. A three-dimensional object can always be rotated around an infinite number of imaginary lines called *rotation axes*.

There are many ways to represent an orientation of an object



Axis–Angle



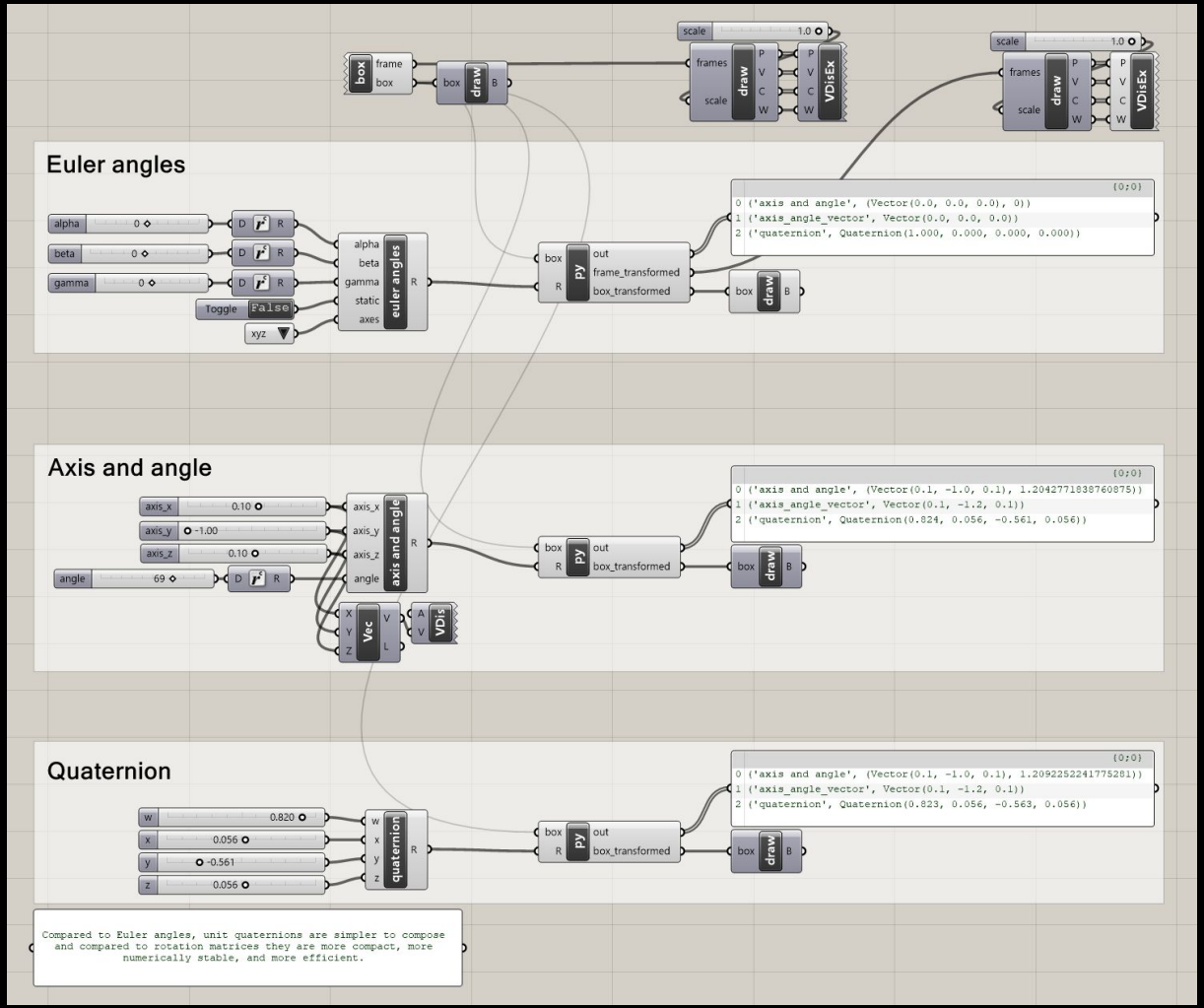
Euler angles

$$w+x\,\mathbf{i}+y\,\mathbf{j}+z\,\mathbf{k}$$

Unit quaternions

```
"""There are several ways to construct a `Rotation`.
"""
import math

from compas.geometry import Frame
from compas.geometry import Rotation

R = Rotation.from_axis_and_angle([1, 0, 0], math.radians(30))
R = Rotation.from_axis_and_angle([1, 0, 0], math.radians(30), point=[1, 0, 0])
R = Rotation.from_basis_vectors([0.68, 0.68, 0.27], [-0.67, 0.73, -0.15])
R = Rotation.from_frame(Frame([1, 1, 1], [0.68, 0.68, 0.27], [-0.67, 0.73, -0.15]))
R = Rotation.from_axis_angle_vector([-0.043, -0.254, 0.617])
R = Rotation.from_quaternion([0.945, -0.021, -0.125, 0.303])
R = Rotation.from_euler_angles([1.4, 0.5, 2.3], static=True, axes='xyz')

print(R)
```

```
"""Example: Different Robot vendors use different conventions to describe TCP
orientation."""

from compas.geometry import Frame
from compas.geometry import Point
from compas.geometry import Vector

point = Point(0.0, 0.0, 63.0)
xaxis = Vector(0.68, 0.68, 0.27)
yaxis = Vector(-0.67, 0.73, -0.15)
F = Frame(point, xaxis, yaxis)

print(F.quaternion)  # ABB
print(F.euler_angles(static=False, axes='xyz'))  # Staubli
print(F.euler_angles(static=False, axes='zyx'))  # KUKA
print(F.axis_angle_vector)  # UR
```

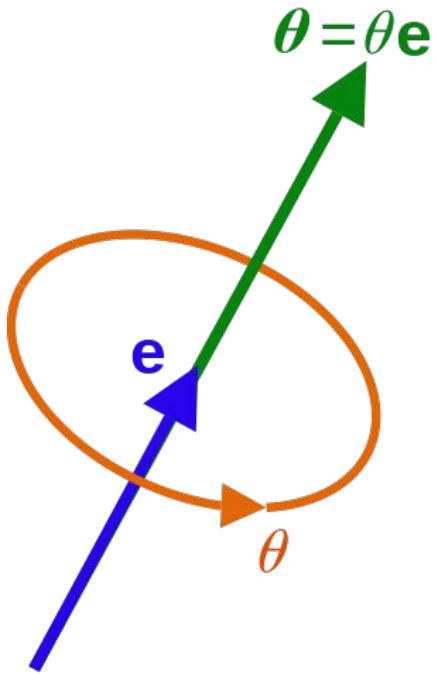

Euler angles
alpha
beta
gamma
Toggle
False
xyz
euler angles
box
frame
draw
scale
frames
VDisEx
py
out
frame_transformed
box_transformed
{0;0}
0 ('axis and angle', (Vector(0.0, 0.0, 0.0), 0))
1 ('axis_angle_vector', Vector(0.0, 0.0, 0.0))
2 ('quaternion', Quaternion(1.000, 0.000, 0.000, 0.000))
Axis and angle
axis_x
axis_y
axis_z
angle
axis and angle
Vec
VDis
{0;0}
0 ('axis and angle', (Vector(0.1, -1.0, 0.1), 1.2042771838760875))
1 ('axis_angle_vector', Vector(0.1, -1.2, 0.1))
2 ('quaternion', Quaternion(0.824, 0.056, -0.561, 0.056))
Quaternion
w
x
y
z
quaternion
{0;0}
0 ('axis and angle', (Vector(0.1, -1.0, 0.1), 1.2092252241775281))
1 ('axis_angle_vector', Vector(0.1, -1.2, 0.1))
2 ('quaternion', Quaternion(0.823, 0.056, -0.563, 0.056))
Compared to Euler angles, unit quaternions are simpler to compose and compared to rotation matrices they are more compact, more numerically stable, and more efficient.

```python
"""Example: Rotations from euler angles, rotate an object based on 3 euler angles.
"""
from compas.geometry import Rotation

# euler angles
alpha, beta, gamma = -0.156, -0.274, 0.785
static, axes = True, 'xyz'

# Version 1: Create Rotation from angles
R1 = Rotation.from_euler_angles([alpha, beta, gamma], static, axes)

# Version 2: Concatenate 3 Rotations
xaxis, yaxis, zaxis = [1, 0, 0], [0, 1, 0], [0, 0, 1]
Rx = Rotation.from_axis_and_angle(xaxis, alpha)
Ry = Rotation.from_axis_and_angle(yaxis, beta)
Rz = Rotation.from_axis_and_angle(zaxis, gamma)

if static:  # check difference between pre- and post-concatenation!
    R2 = Rz * Ry * Rx
else:
    R2 = Rx * Ry * Rz
```


Axis-angle representation

The axis–angle representation of a rotation parameterizes a rotation in a three-dimensional Euclidean space by two quantities:

- **a unit vector e** (indicating the direction of an axis of rotation)
- **angle θ** (describing the magnitude of the rotation about the axis)



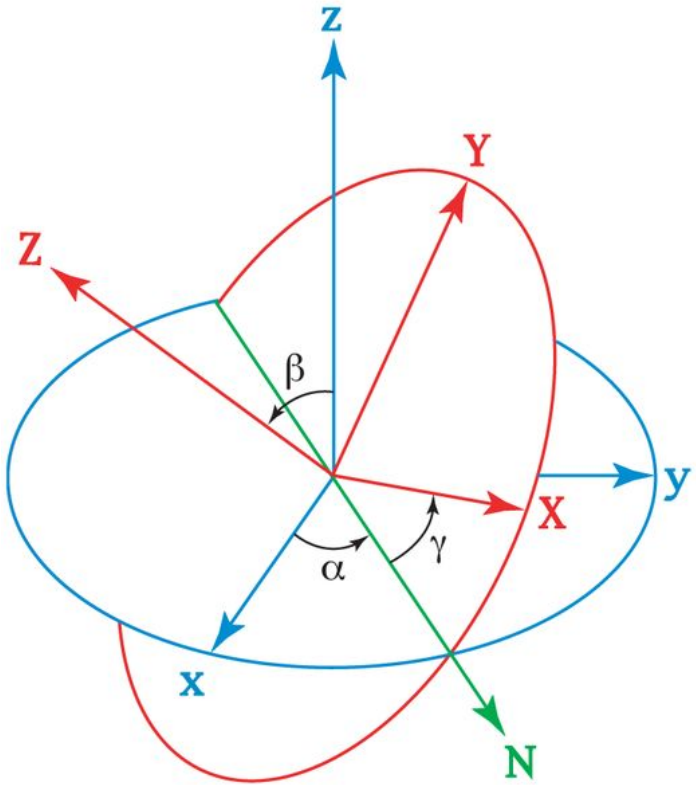
```python
"""Example: Create a rotation from and axis and an angle.
"""
from compas.geometry import Rotation
from compas.geometry import Vector

aav = Vector(-0.043, -0.254, 0.617)
angle, axis = aav.unitized(), aav.length
print(angle, axis)

R = Rotation.from_axis_angle_vector(aav)
axis, angle = R.axis_and_angle
print(axis, angle)
```

Euler angles

The Euler angles are three angles introduced by Leonhard Euler to describe the orientation of a rigid body with respect to a fixed coordinate system.

The three elemental rotations may be

- extrinsic (`static=True`, rotations about the axes xyz of the original coordinate system, which is assumed to remain motionless), or
- intrinsic (`static=False`, rotations about the axes of the rotating coordinate system XYZ, solidary with the moving body, which changes its orientation after each elemental rotation).



Quaternions

In mathematics, the quaternions comprise a number system that extends the complex numbers.

Quaternions are generally represented in the form:

$$w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

where x, y, z, and w are real numbers, and $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ are the fundamental quaternion units.

Quaternions can be added and multiplied in a similar way to complex numbers.

Unit Quaternions

Unit quaternions (those of norm 1) provide a convenient mathematical notation for representing orientations and rotations of objects in three dimensions..

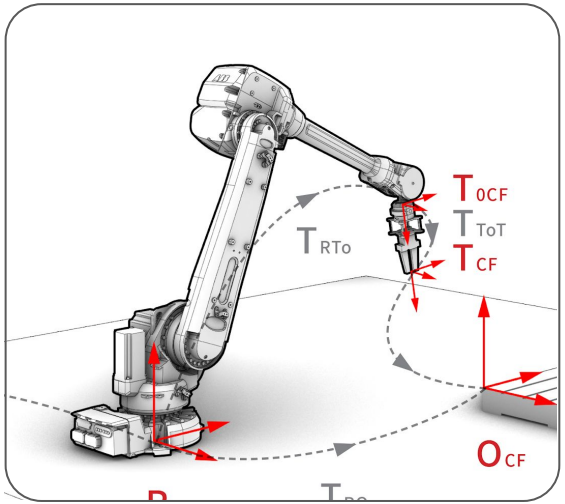
The quaternionic representation is very closely related to the axis-angle representation. A rotation of Θ about the unit vector $v = (v_x, v_y, v_z)$ is given by:

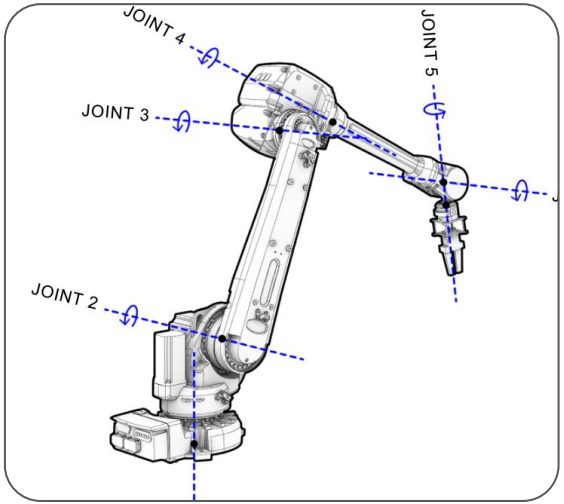
$$\cos\left(\frac{\Theta}{2}\right) + (v_x\mathbf{i} + v_y\mathbf{j} + v_x\mathbf{k})\sin\left(\frac{\Theta}{2}\right)$$

Compared to Euler angles they are simpler to compose and compared to rotation matrices they are more compact, more numerically stable, and more efficient.

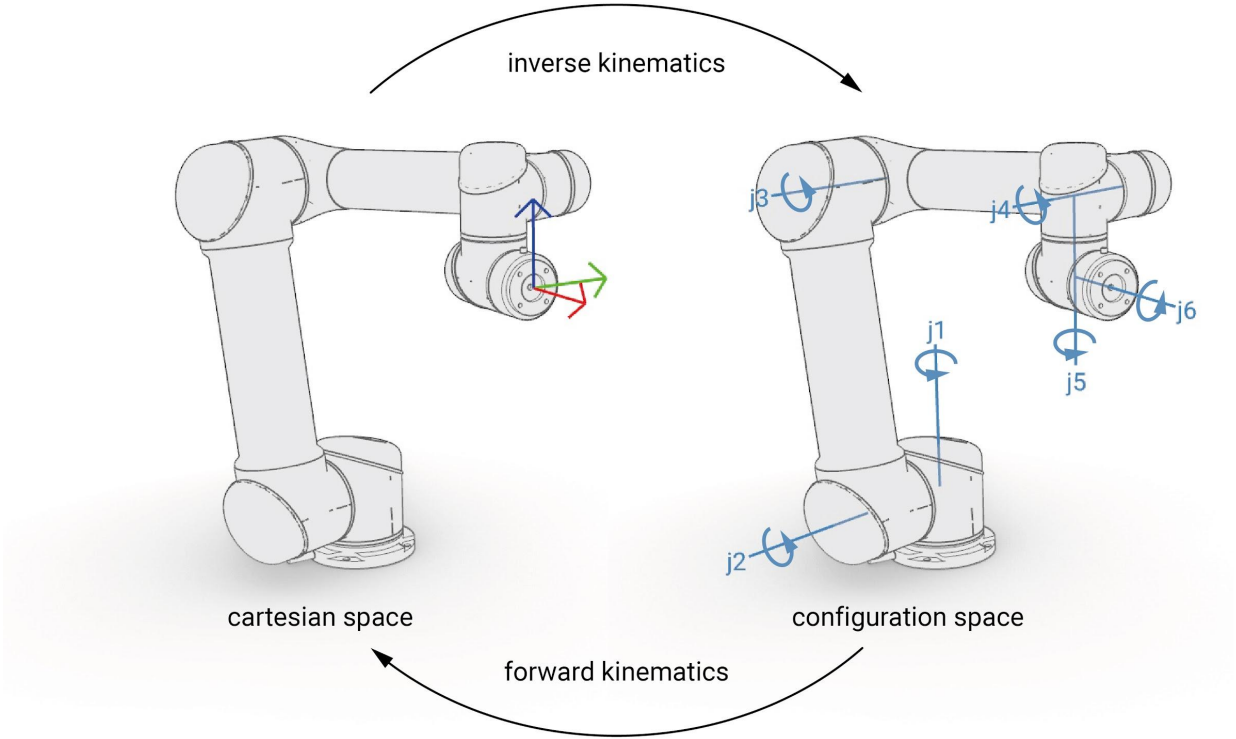
```
"""Example: Create a rotation from a quaternion.
"""
from compas.geometry import Rotation
from compas.geometry import Quaternion

q =(0.918958, -0.020197, -0.151477, 0.363544)
print(q.is_unit)
R = Rotation.from_quaternion(q)
print(R.quaternion == q)
```

Frames vs Configuration

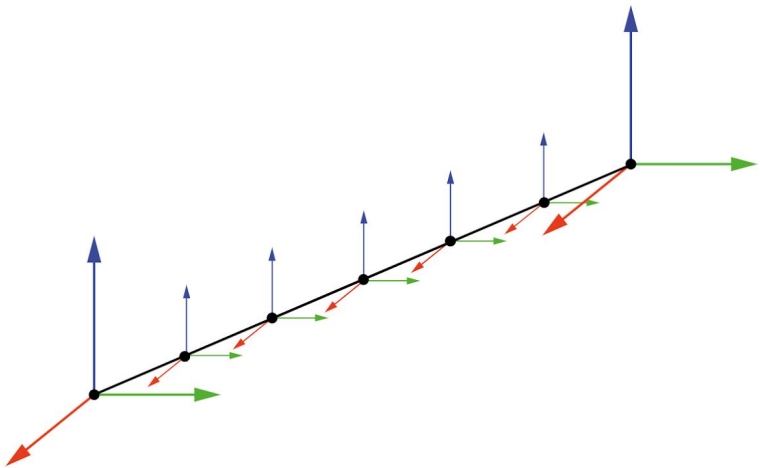


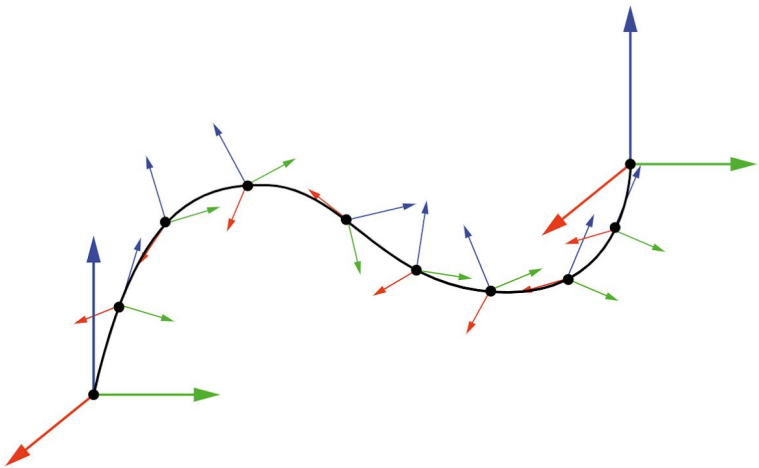


Cartesian vs Joint space



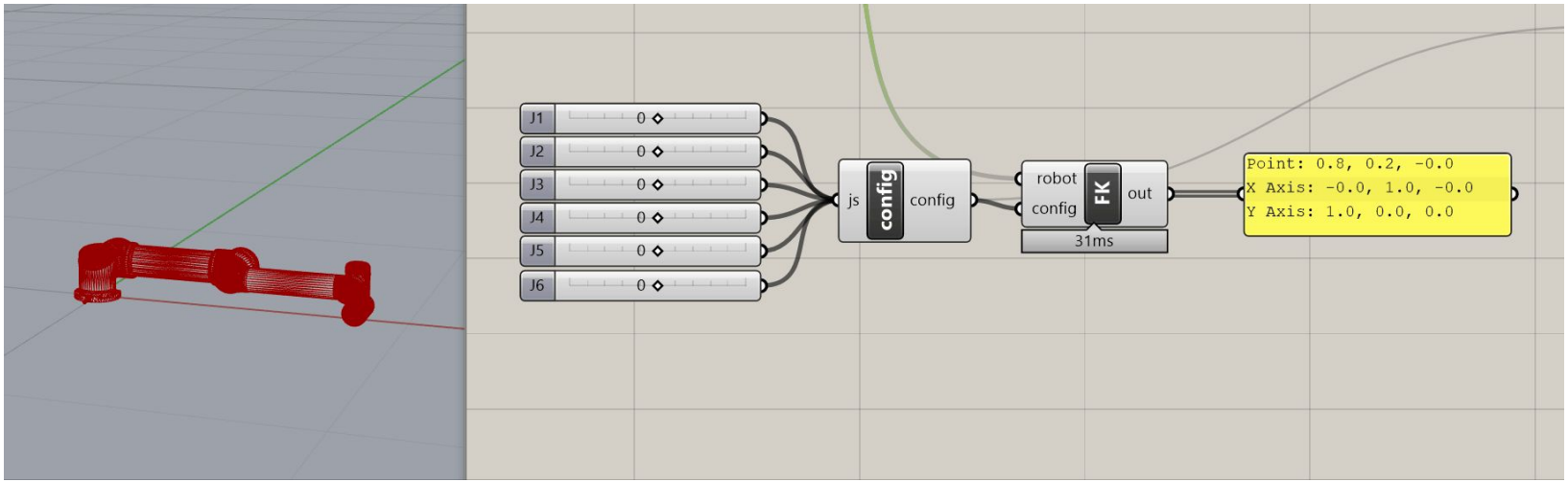
Motion in Joint vs Cartesian space





Forward Kinematics

- **From a configuration, get frame of tip (or other link)**
- Fast to solve for serial manipulators

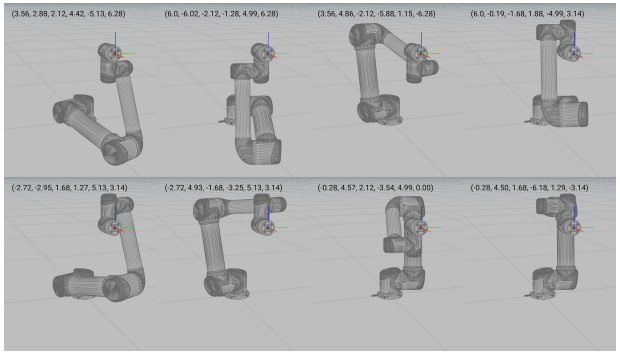


Forward Kinematics

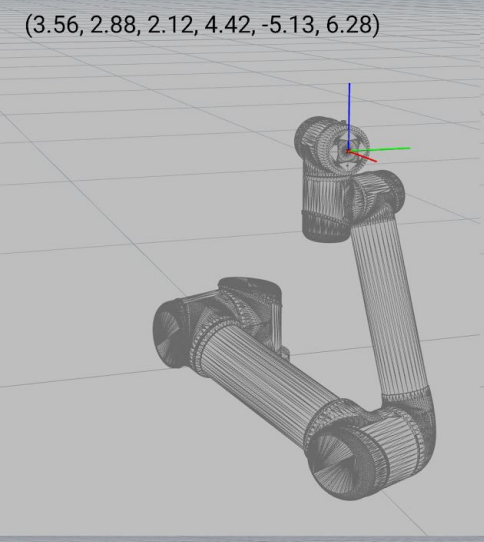
```python
# Create config
config = model.zero_configuration()

# Get FK for tip
print (model.forward_kinematics(config))
# Get FK for base
print (model.forward_kinematics(config, link_name=model.get_base_link_name()))
```

Inverse Kinematics

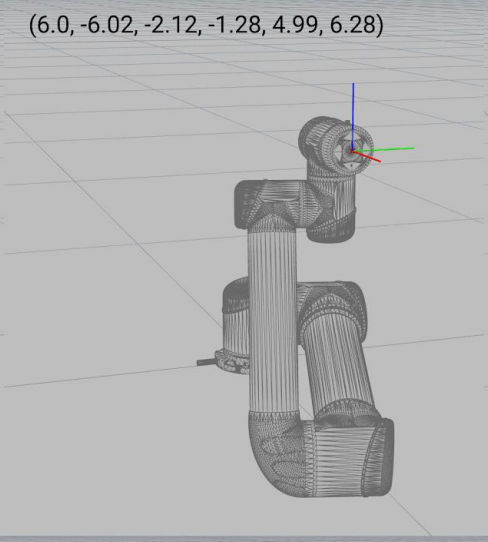
- **From a frame, get a feasible configuration**
- Analytical approach
  - *Pro:* Computes all solutions and determines if no solution exists
  - *Pro:* Very fast to compute
  - *Pro:* No need to define initial guesses
  - *Con:* Deriving equations is difficult or tedious
  - *Con:* Different derivation for different kinematic structures
  - *Con:* Only works for non-redundant robots
- Numerical approach
  - Based on an initial guess, calculate a sequence of configurations such that the error approaches 0.
  - *Pro:* Works on most robots including redundant ones
  - *Con:* Computes only one solution at a time
  - *Con:* Slower to compute
  - *Con:* Does not indicate if no solution exists



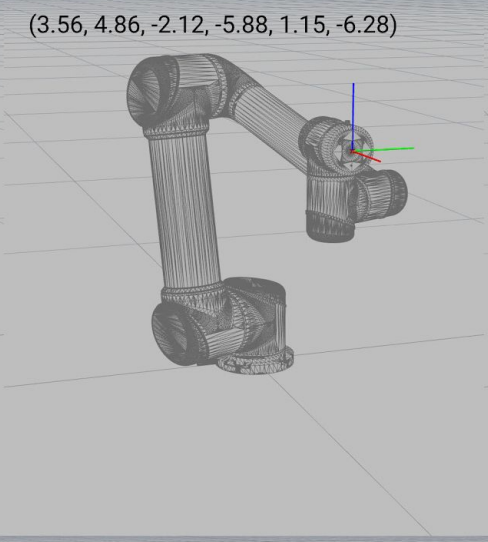
(3.56, 2.88, 2.12, 4.42, -5.13, 6.28)



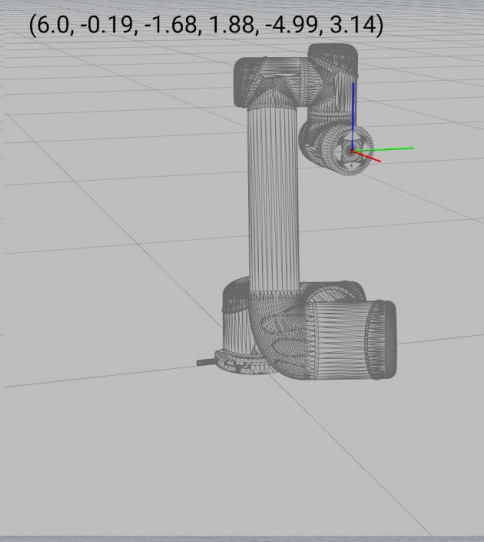
(6.0, -6.02, -2.12, -1.28, 4.99, 6.28)



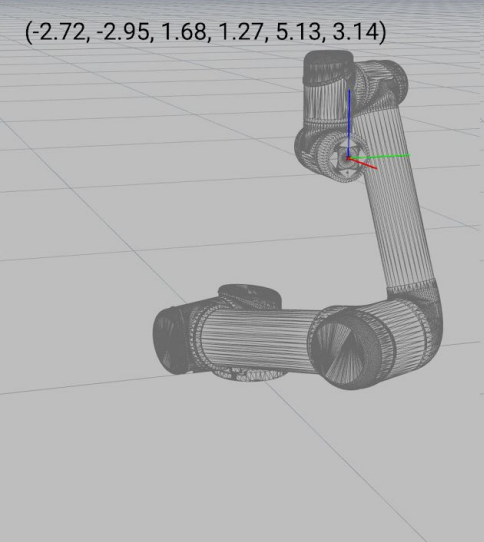
(3.56, 4.86, -2.12, -5.88, 1.15, -6.28)



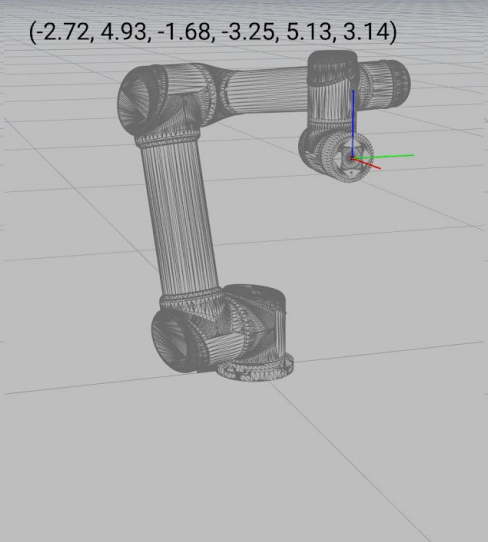
(6.0, -0.19, -1.68, 1.88, -4.99, 3.14)



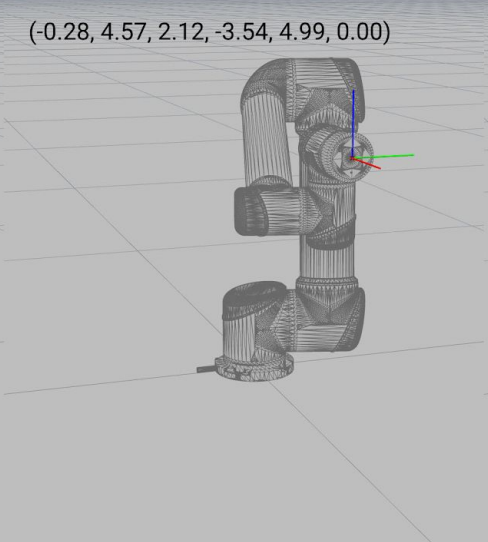
(-2.72, -2.95, 1.68, 1.27, 5.13, 3.14)



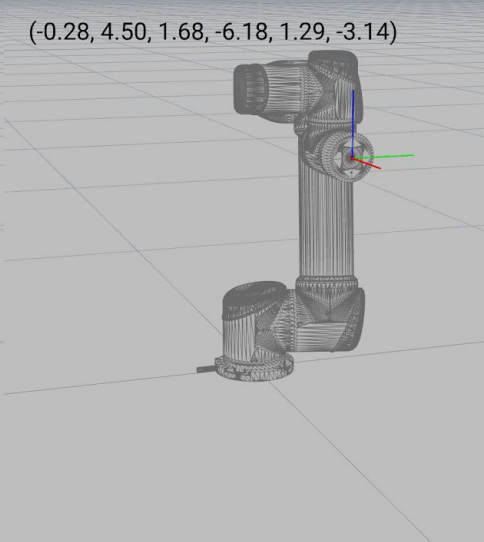
(-2.72, 4.93, -1.68, -3.25, 5.13, 3.14)



(-0.28, 4.57, 2.12, -3.54, 4.99, 0.00)



(-0.28, 4.50, 1.68, -6.18, 1.29, -3.14)



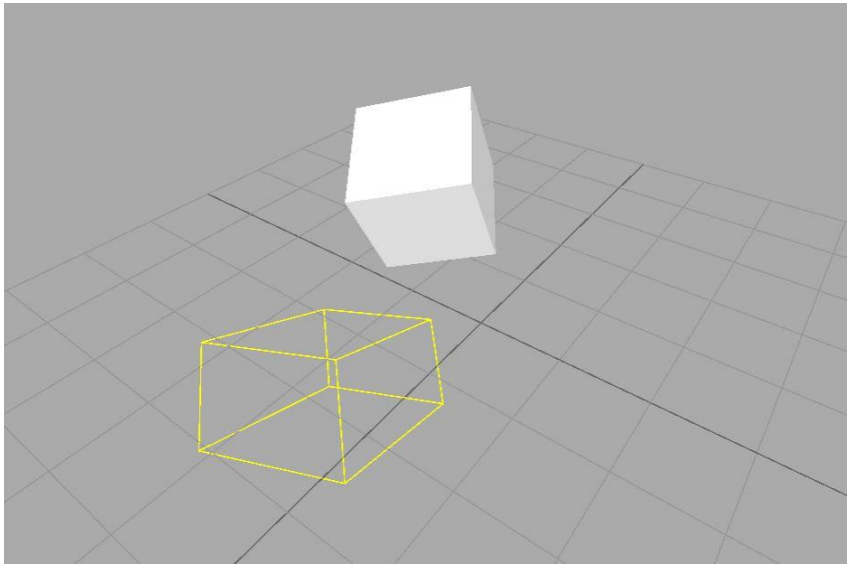
Inverse Kinematics

```
from compas_fab.backends.kinematics.solvers import UR5Kinematics

f = Frame((0.417, 0.191, -0.005), (-0.000, 1.000, 0.00), (1.000, 0.000, 0.000))
solutions = UR5Kinematics().inverse(f)
```

Assignment

Project box to xy-plane

1. Create a box at a certain location with a certain orientation.
2. Create a `Projection` (can be orthogonal, parallel or perspective)
3. Convert the box to a mesh and project the it onto the xy-plane.
4. Use artists to draw the result,



Assignment submission

Next week

- First assignment submission due: Wed 9h March, 9AM.
- Ask for help if needed
  - Slack
  - Forum
- Next week:
  - Robot models

Thanks!