



C O M P A S S F A B

[ robots ']):

and at kwargs[ 'rob

pose(settings[ 'star

.to\_data()

robot

get\_config

\_or\_pose

(settings[

'goal' ] = goal .

*https://u.nu/tokyo20*

## AGENDA

Intro & session goals

COMPAS 101

COMPAS FAB intro

Robotics

ROS primer

Kinematics & Motion planning



C O M P A S



```
    if key in fixed:
        continue
    p = key_xyz[key]
    nbrs = mesh.vertex_neighbours(key, ordered=True)
    c = center_of_mass_polygon([key_xyz[nbr] for nbr in nbrs])
    attr = mesh.vertex[key]
    attr['x'] += d * (c[0] - p[0])
    attr['y'] += d * (c[1] - p[1])
    attr['z'] += d * (c[2] - p[2])

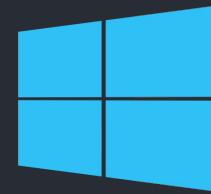
    if callback:
        callback(mesh, k, callback_args)

def smooth_mesh_length(mesh, lmin, lmax, fixed=None, kmax=100):
    if callback:
        if not callable(callback):
            raise Exception('Callback is not callable.')
    fixed = fixed or []
    fixed = set(fixed)
    for k in range(kmax):
```

Getting started

***<https://u.nu/tokyo20>***

⌘ + SPACE



Terminal

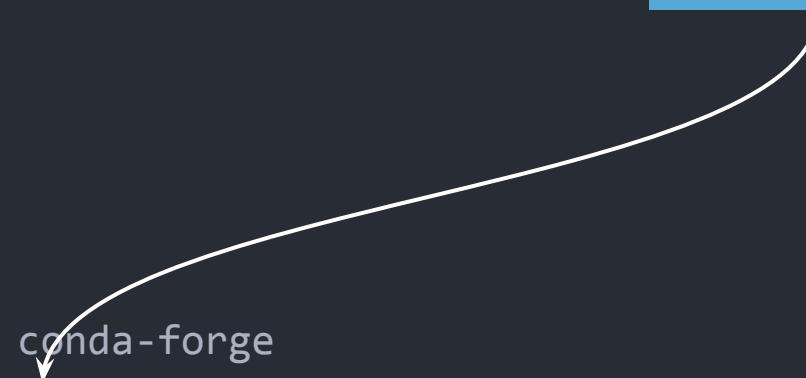
Anaconda Prompt



## Environment

```
(base) conda config --add channels conda-forge
(base) conda create -n tokyo20 python=3.8 compas_fab=0.11 python.app --yes
```

Python version



```
(base) conda config --add channels conda-forge
(base) conda create -n tokyo20 python=3.8 compas_fab=0.11 python.app --yes
```

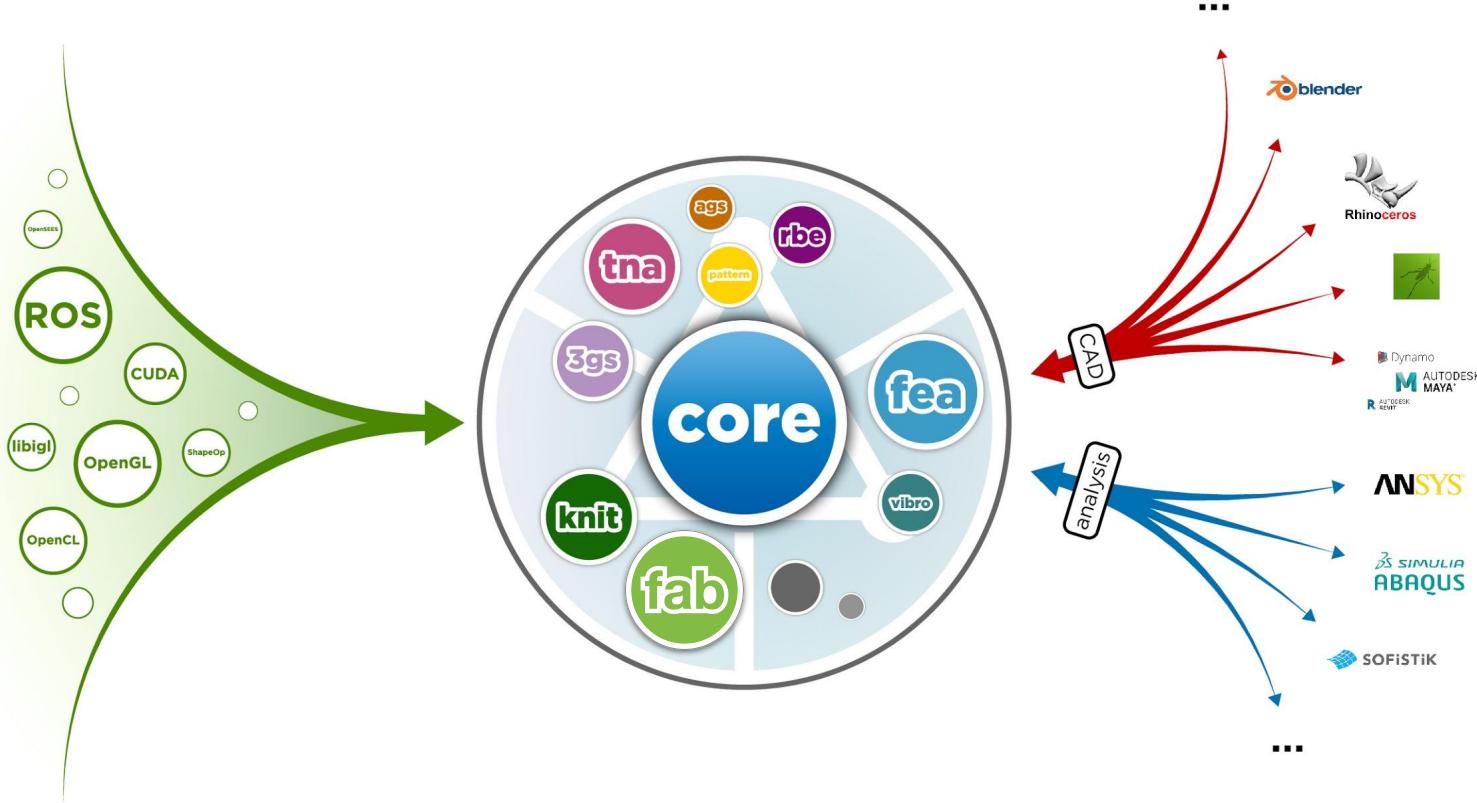
COMPAS FAB

```
(base) conda config --add channels conda-forge  
(base) conda create -n tokyo20 python=3.8 compas_fab=0.11 python.app --yes
```

Only for Mac

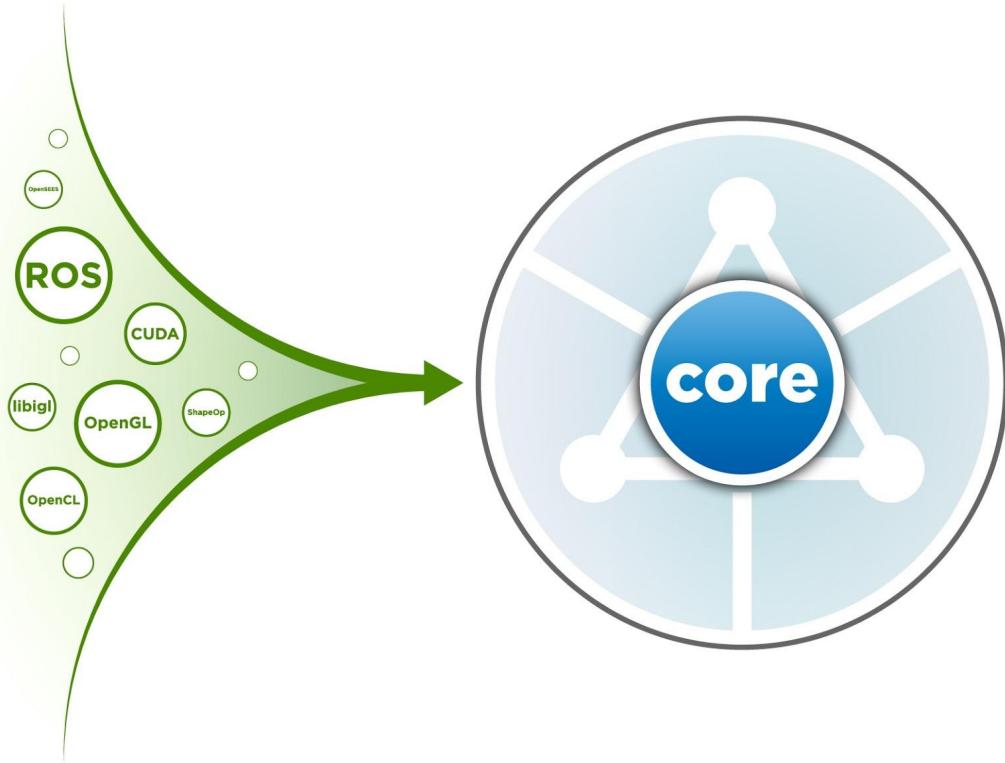


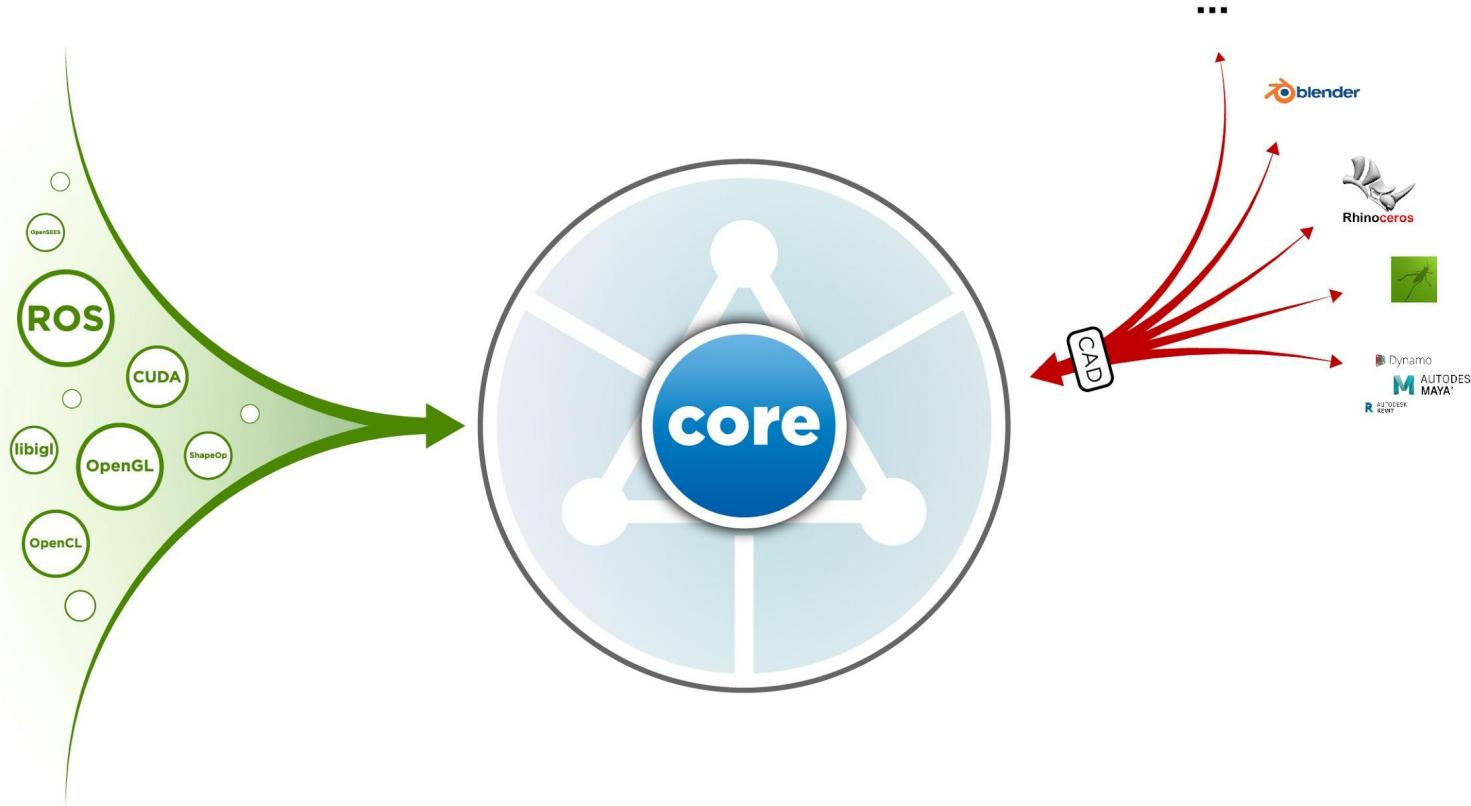
```
(base) conda config --add channels conda-forge
(base) conda create -n tokyo20 python=3.8 compas_fab=0.11 python.app --yes
```

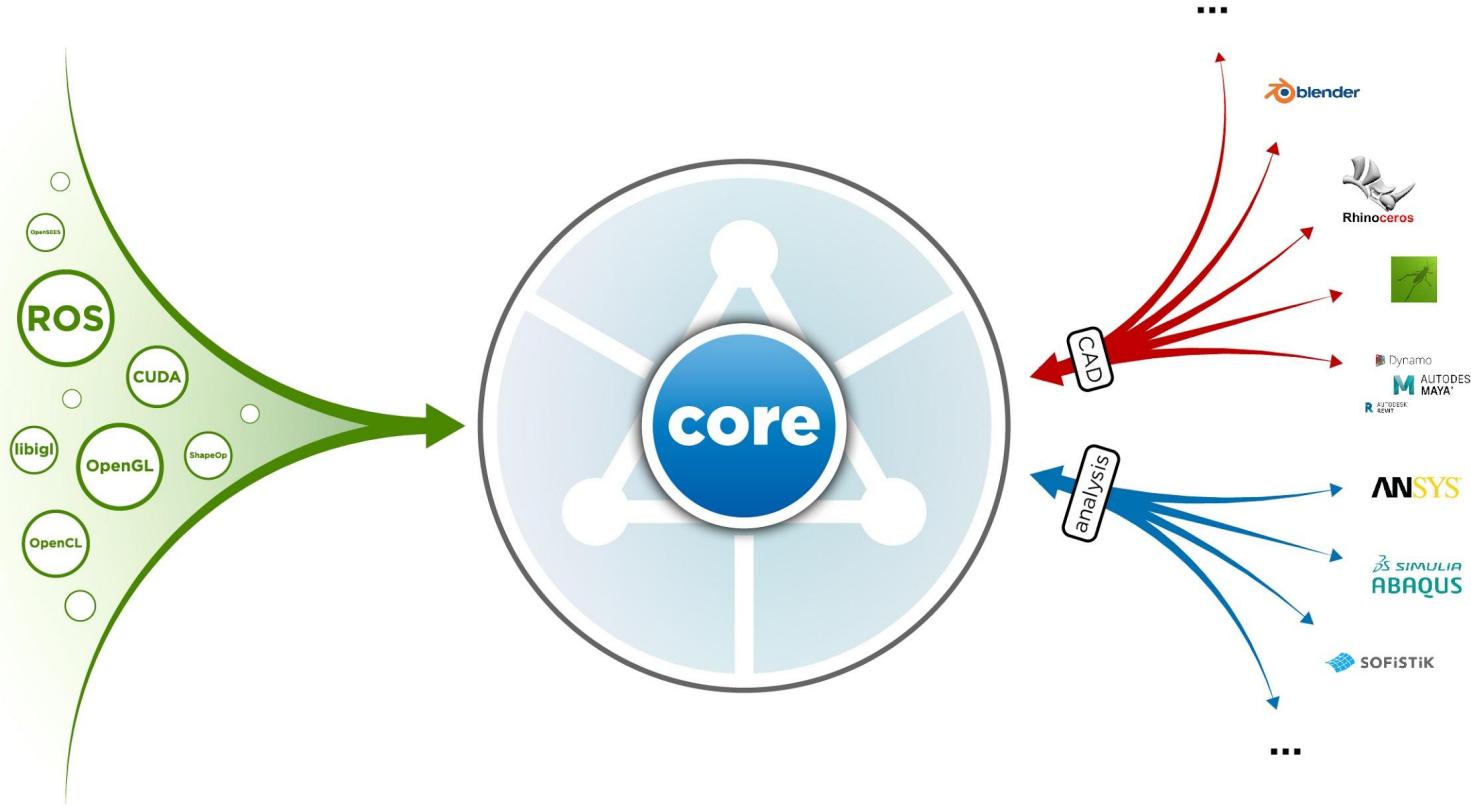


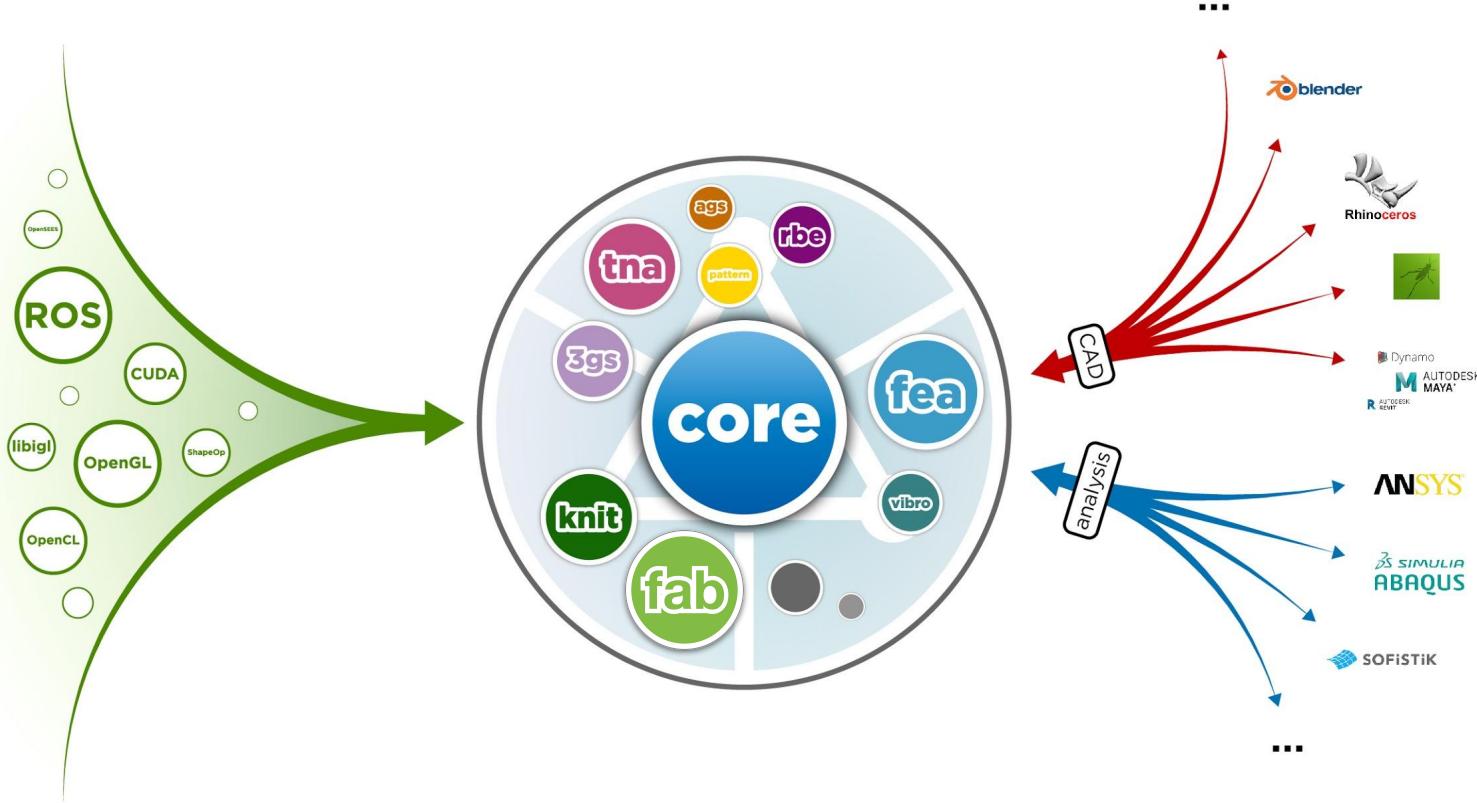


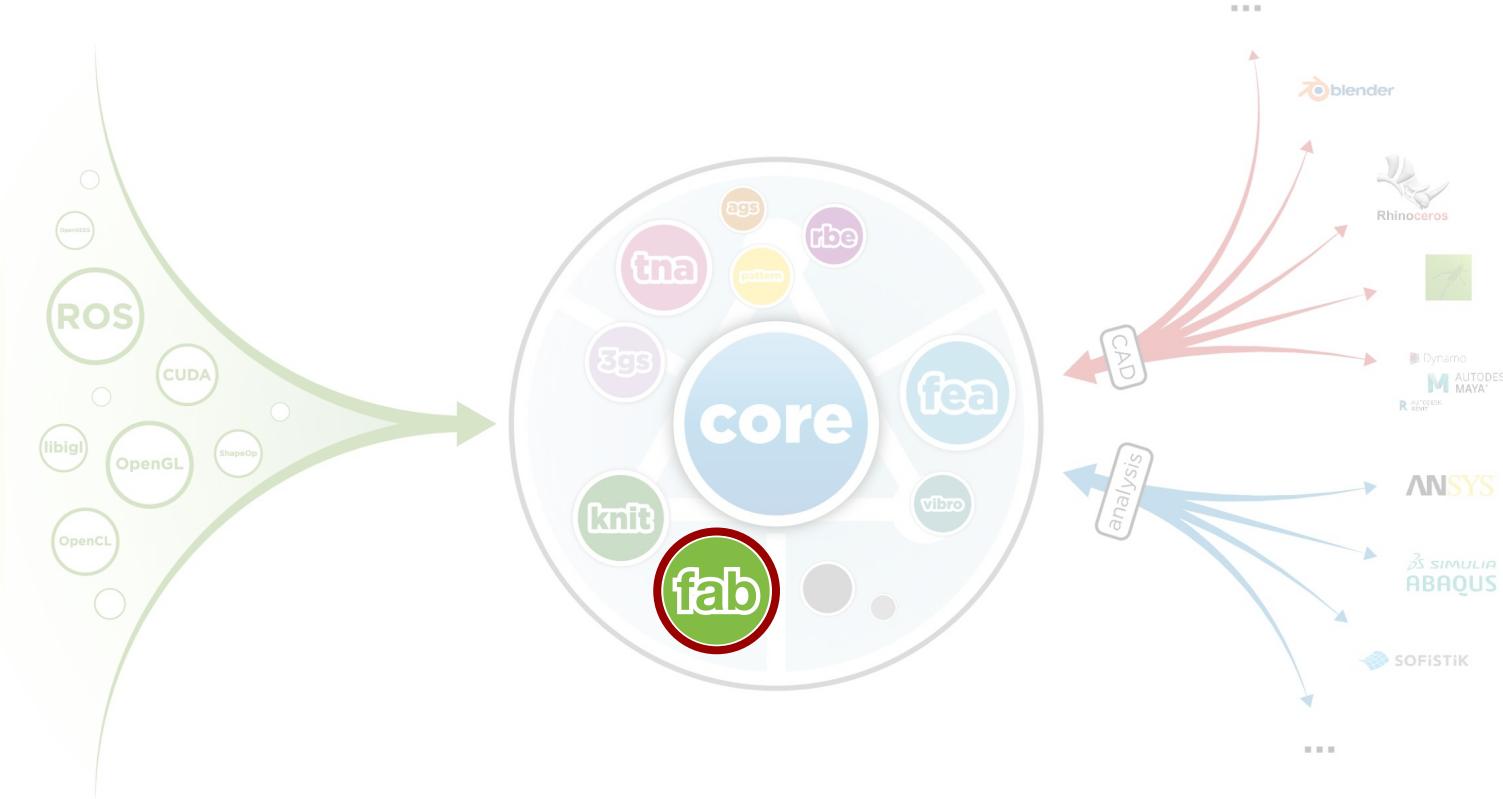
- Open source, Python
- Flexible data structures
- Algorithms & utilities for AEC
- Geometry processing
- CAD independent

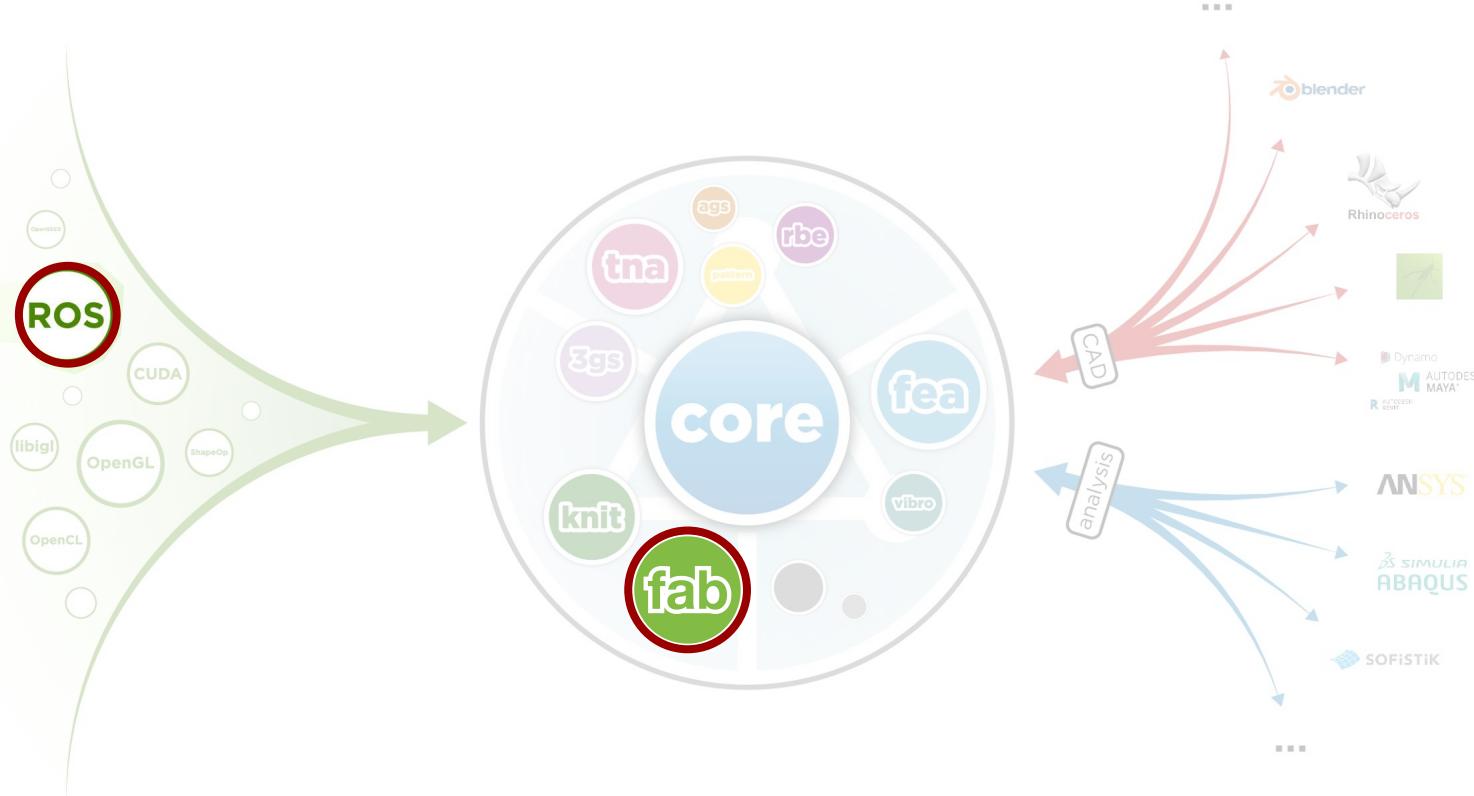












```
arch-hix-dock-453:~ vanmelet$ python
Python 3.6.3 |Anaconda custom (64-bit)| (default, Oct  6 2017, 12:04:38)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import compas
>>> from compas.datastructures import Mesh
>>> mesh = Mesh.from_obj(compas.get('faces.obj'))
>>> mesh.summary()

=====
===
Mesh summary
=====
===
- name: Mesh
- vertices: 36
- edges: 60
- faces: 25
- vertex degree: 2/4
- face degree: 2/4

=====
```

```
===
>>> □
```

dr\_numpy.py — compas-dev

```

FOLDERS
└── compas-dev
    ├── compas
    ├── build
    ├── data
    ├── dist
    ├── docs
    ├── libs
    └── src
        ├── compas
        │   ├── __init__.py
        │   ├── __pycache__
        │   ├── com
        │   ├── datastructures
        │   ├── files
        │   ├── geometry
        │   ├── interop
        │   └── numerical
        │       ├── __init__.py
        │       ├── __pycache__
        │       ├── alglib
        │       ├── algorithms
        │       ├── descent
        │       ├── devo
        │       ├── dr
        │       ├── drs
        │       ├── fd
        │       ├── ga
        │       ├── lma
        │       ├── mma
        │       ├── pca
        │       ├── solvers
        │       └── topopt
        ├── __init__.py
        ├── linalg.py
        ├── matrices.py
        ├── operators.py
        ├── utilities.py
        ├── plotters
        ├── robots
        ├── rpc
        └── topology
            ├── __init__.py
            ├── __pycache__
            ├── __init__.py
            ├── __pycache__
            ├── COMPAS-app-info
            ├── compas.blender
            ├── compas.ghpython
            ├── compas.hpc
            ├── compas.revit
            ├── compas_rhino
            ├── temp
            └── tests
                ├── bumpversion.cfg
                ├── editconfig
                ├── gitignore
                ├── travis.yml
                └── CONTRIBUTORS.md
                └── LICENSE

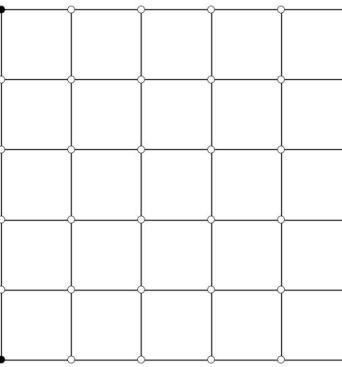
Git branch: master, index: ✓, working: 10x, Line 33, Column 1

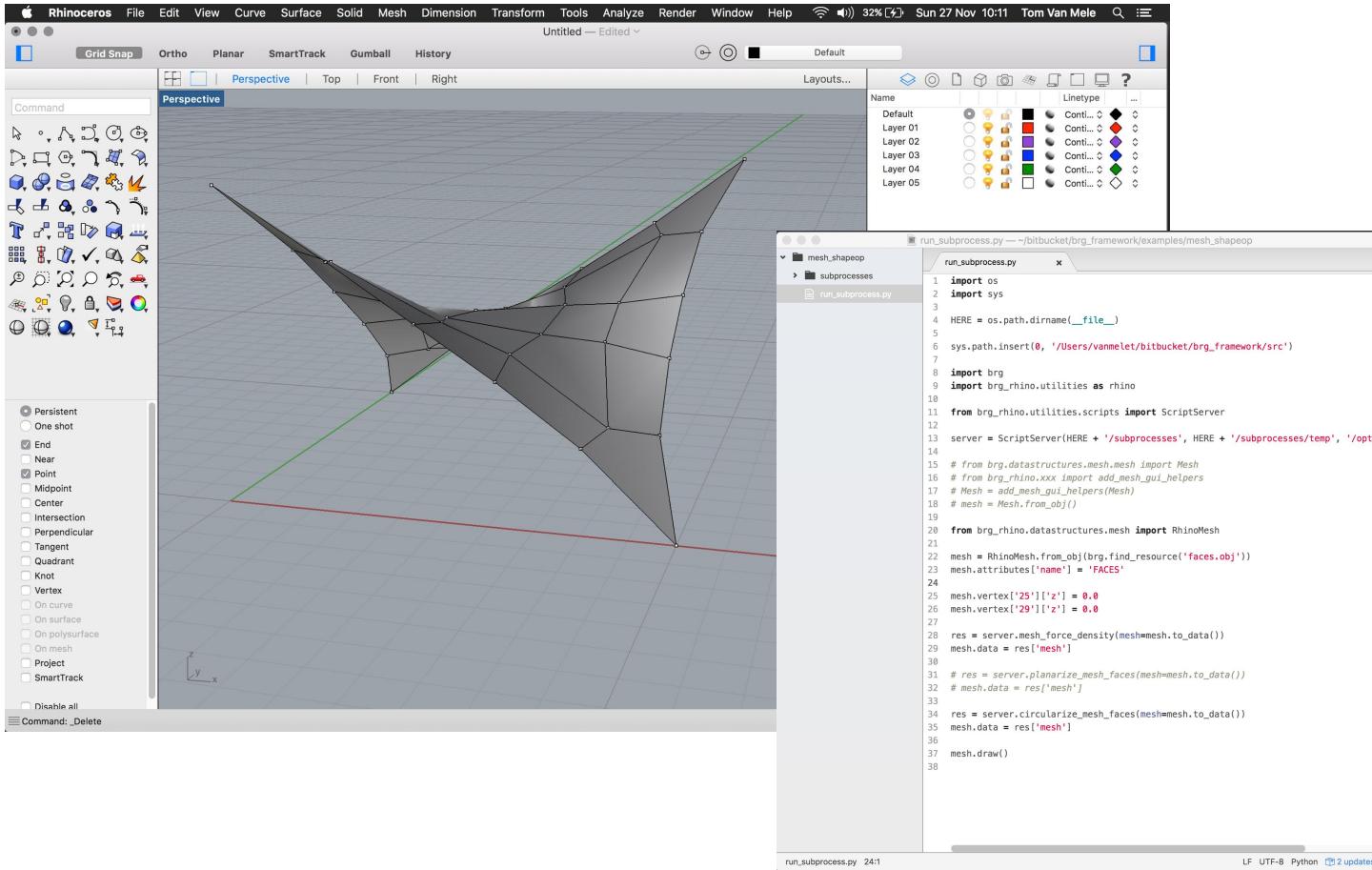
```

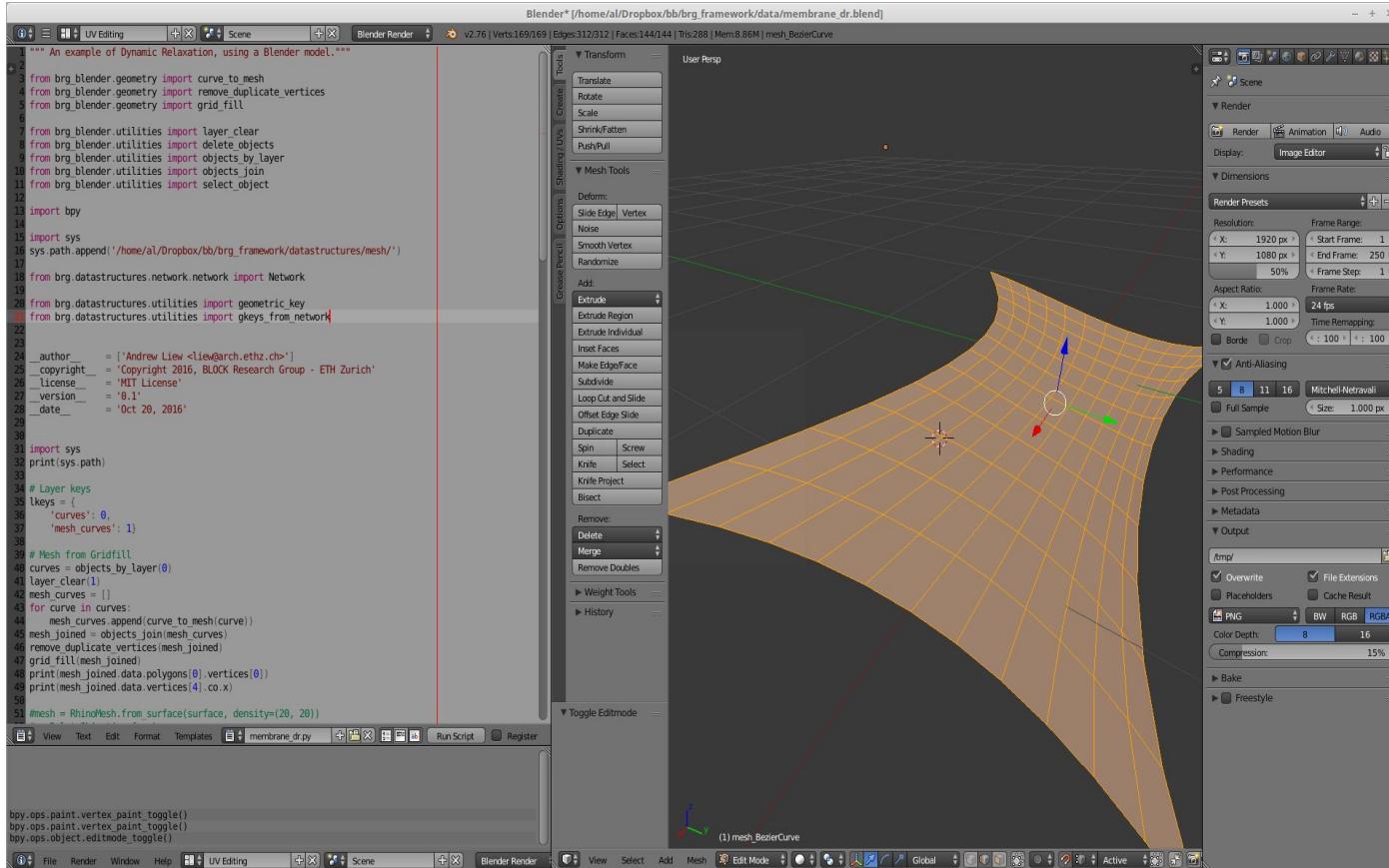
```

dr_numpy.py
1  from __future__ import absolute_import
2  from __future__ import division
3  from __future__ import print_function
4
5  import compas
6
7  try:
8      from numpy import array
9      from numpy import meshgrid
10     from numpy import isnan
11     from numpy import ones
12     from numpy import zeros
13     from scipy.linalg import norm
14     from scipy.sparse import diags
15
16     except ImportError:
17         raise_if_not_ipython()
18
19     from compas.numerical import connectivity_matrix
20     from compas.numerical import normrow
21
22
23     __all__ = ['dr_numpy']
24
25
26     K = [
27         [0.5, 0.5],
28         [0.5, 0.5],
29         [0.5, 0.0, 0.5],
30         [0.0, 0.5, 0.5, 1.0],
31     ]
32
33
34     class Coeff():
35         def __init__(self, c):
36             self.c = c
37             self.a = (1 - c * 0.5) / (1 + c * 0.5)
38             self.b = 0.5 / (1 + self.a)
39
40
41     def dr_numpy(vertices, edges, fixed, loads, ure, fpre, lpre, initE, E, radius,
42                 callback=None, callback_args=None, **kwargs):
43
44         """Implementation of the dynamic relaxation method for form finding and analysis
45         of articulated networks of axial-force members.
46
47         Parameters
48
49             vertices : list
50                 XYZ coordinates of the vertices.
51             edges : list
52                 Connectivity of the vertices.
53             fixed : list
54                 Indices of the fixed vertices.
55             loads : list
56                 XYZ components of the loads on the vertices.
57             qpre : list
58                 Prescribed force densities in the edges.
59             fpre : list
60                 Prescribed forces in the edges.
61             lpre : list
62                 Prescribed lengths of the edges.
63             initE : float
64                 Initial length of the edges.
65             E : float
66                 Stiffness of the edges.
67             radius : list
68                 Radius of the edges.
69             callback : callable, optional
70                 User-defined function that is called at every iteration.
71             callback_args : tuple, optional
72                 Additional arguments passed to the callback.
73
74         Notes
75
76             For more info, see [1]_.
77
78             References
79
80             .. [1] De Lant L., Veenendaal P., Van Hee T., Hollart H. and Block P.,
81                 «Bending incorporated: designing tension structures by integrating
82                 Proceedings of Tensionnet Symposium 2013, Istanbul, Turkey, 2013.
83
84             Examples
85
86             .. plot:: examples/dr_numpy.py
87
88             import compas
89             from compas.datastructures import Network

```







```
# Rhino
from compas.datastructures import Mesh
from compas_rhino.artists import MeshArtist

mesh = Mesh.from_obj('https://u.nu/hypar')

artist = MeshArtist(mesh)

artist.draw_mesh()

# Blender
from compas.datastructures import Mesh
from compas_blender.artists import MeshArtist

mesh = Mesh.from_obj('https://u.nu/hypar')

artist = MeshArtist(mesh)

artist.draw_mesh()
```

```
# Rhino
from compas.datastructures import Mesh
from compas_rhino.artists import MeshArtist

mesh = Mesh.from_obj('https://u.nu/hypar')

artist = MeshArtist(mesh)

artist.draw_mesh()
```

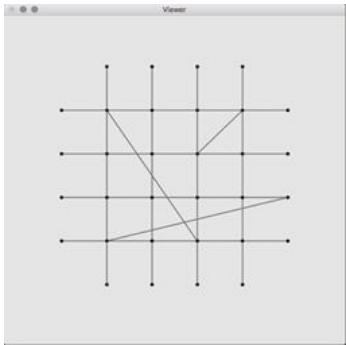
```
# Blender
from compas.datastructures import Mesh
from compas_blender.artists import MeshArtist

mesh = Mesh.from_obj('https://u.nu/hypar')

artist = MeshArtist(mesh)

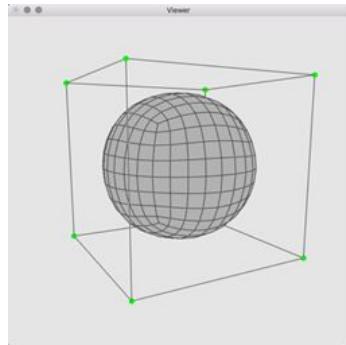
artist.draw_mesh()
```

# Data structures



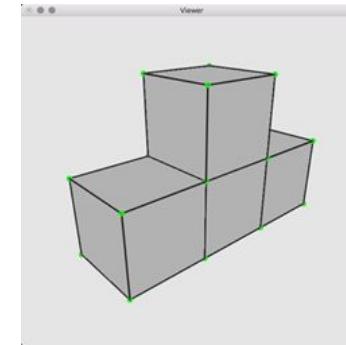
Network

- General networks
- Directed edge graph



Mesh

- Surface meshes
- Half-edge



Volumetric Mesh

- Cellular meshes
- Half-plane

# Algorithms

## Geometry

- Smoothing
- Planarization
- Convex hull
- Triangulation
- Transformations
- Bounding boxes
- Isolines
- ...

## Numerical

- Dynamic relaxation
- Force density
- Topology optimization
- Genetic algorithms
- Principal component analysis
- Differential evolution
- ...

## Topology

- Traversal
- Graph colouring
- Planarity
- Conway operators
- ...

COMPAS **FAB**

# Installation

***<https://u.nu/tokyo20>***

```
(base)      conda activate tokyo20
```

```
(tokyo20)  pip show compas_fab
```

Name: compas-fab

Version: 0.11.0

..

Active  
environment

```
(base)      conda activate tokyo20
```

```
(tokyo20)  pip show compas_fab
```

Name: compas-fab

Version: 0.11.0

..

Check version

```
(base)      conda activate tokyo20
```

```
(tokyo20)  pip show compas_fab
```

```
Name: compas-fab
```

```
Version: 0.11.0
```

```
..
```

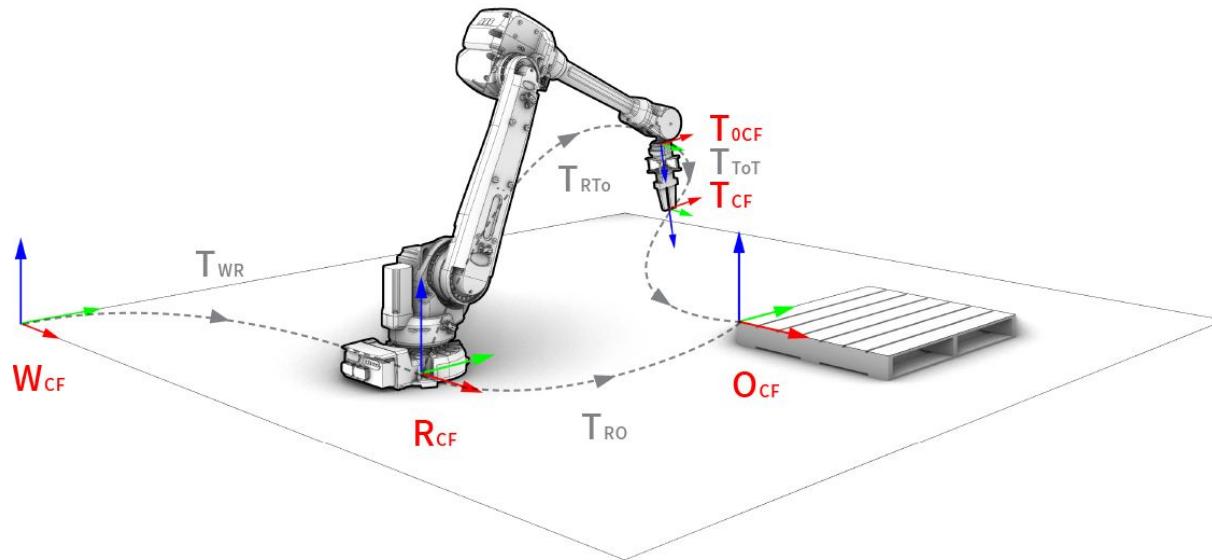


# Robotics

## Fundamentals | Models | Backends

# Robotic **fundamentals**

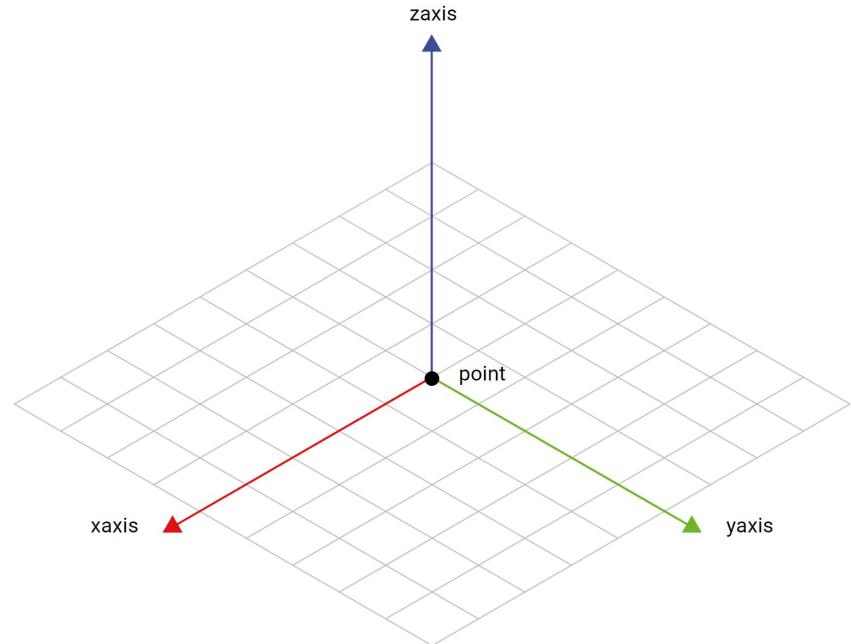
# Frame and Transformation



# 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.



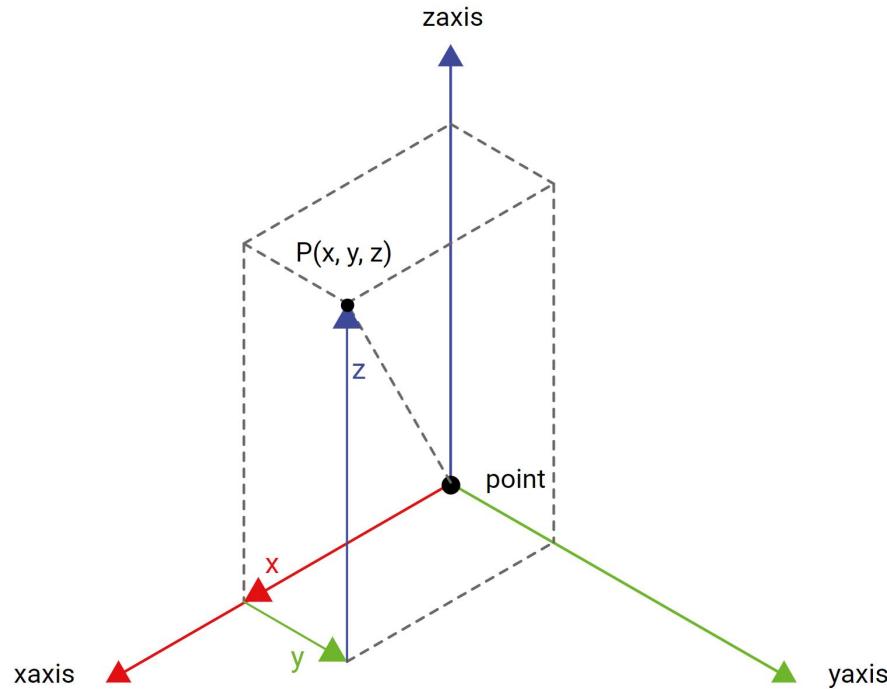


```
"""There are several ways to construct a `Frame`.  
"""  
  
from compas.geometry import Point  
from compas.geometry import Vector  
from compas.geometry import Frame  
from compas.geometry import Plane  
  
# Frame autocorrects axes to be orthonormal  
F = Frame(Point(1, 0, 0), Vector(-0.45, 0.1, 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()
```



```
"""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)

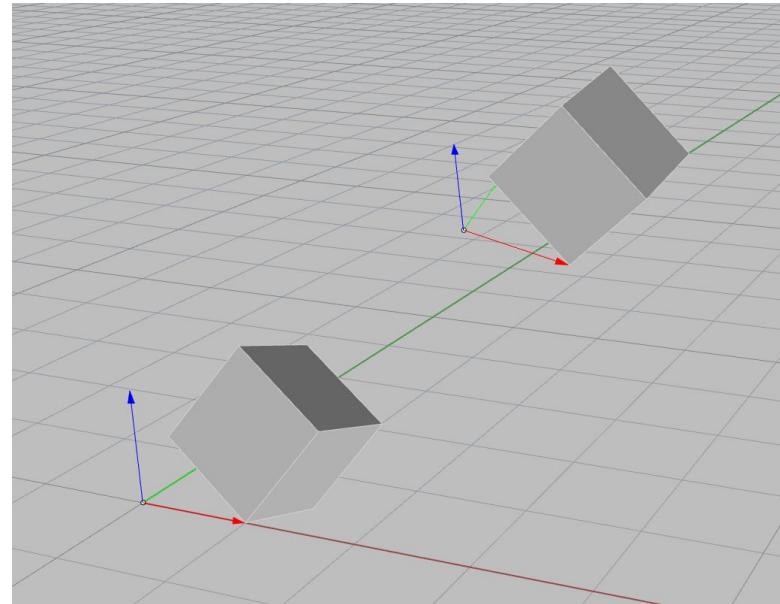
F = Frame(point, xaxis, yaxis) # coordinate system F
P = Point(35., 35., 35.) # point in F (local coordinates)

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

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

# 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.

"""

from compas.geometry import Frame
from compas.geometry import Box

# 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])

# Represent box frame in frame F and construct new box
box_frame_transformed = F.to_world_coords(box.frame)
box_transformed = Box(box_frame_transformed, width, length, height)
print("Box frame transformed:", box_transformed.frame)
```



```
from compas.datastructures import Mesh
from compas_rhino.artists import FrameArtist
from compas_rhino.artists import MeshArtist

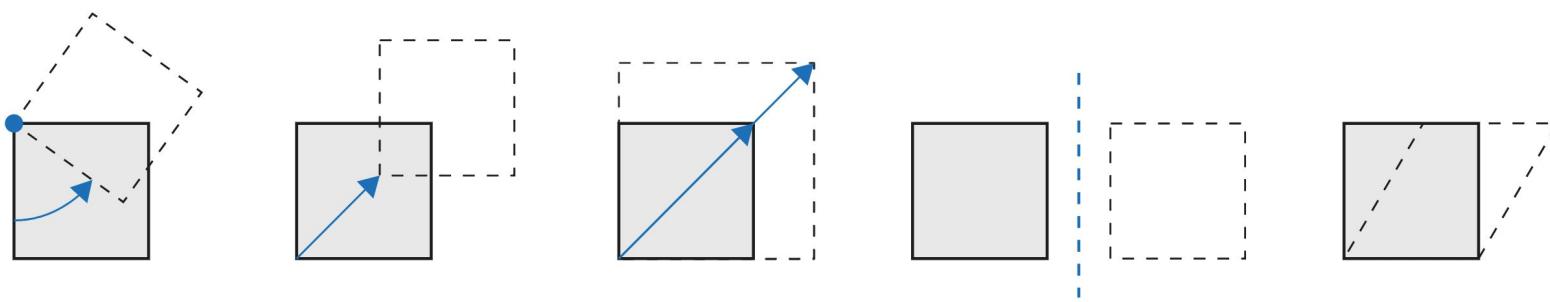
# create artists
artist1 = FrameArtist(Frame.worldXY())
artist2 = MeshArtist(Mesh.from_shape(box))
artist3 = FrameArtist(F)
artist4 = MeshArtist(Mesh.from_shape(box_transformed))

# draw
artist1.draw()
artist2.draw_faces()
artist3.draw()
artist4.draw_faces()
```

# Transformation

Transformations refer to operations such as moving, rotating, and scaling objects. They are stored using 4x4 transformation matrices.

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*.





```
"""Transformation examples
"""

from compas.geometry import *

axis, angle = [0.2, 0.4, 0.1], 0.3
R = Rotation.from_axis_and_angle(axis, angle)

translation_vector = [5, 3, 1]
T = Translation(translation_vector)

scale_factors = [0.1, 0.3, 0.4]
S = Scale(scale_factors)

point, normal = [0.3, 0.2, 1], [0.3, 0.1, 1]
R = Reflection(point, normal)

point, normal = [0, 0, 0], [0, 0, 1]
perspective = [1, 1, 0]
P = Projection.perspective(point, normal, perspective)
```



```
"""Example: Bring a box from the world coordinate system into another coordinate
system.

"""

from compas.geometry import Frame
from compas.geometry import Transformation
from compas.geometry import Box

# 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 frames and apply transformation on box.
T = Transformation.from_frame_to_frame(Frame.worldXY(), F)
box_transformed = box.transformed(T)
print("Box frame transformed", box_transformed.frame)
```

# 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 several ways to construct a `Rotation`.  
"""\n\nimport math\nfrom compas.geometry import Frame\nfrom compas.geometry import Rotation\n\nR = Rotation.from_axis_and_angle([1, 0, 0], math.radians(30))\nR = Rotation.from_axis_and_angle([1, 0, 0], math.radians(30), point=[1, 0, 0])\nR = Rotation.from_basis_vectors([0.68, 0.68, 0.27], [-0.67, 0.73, -0.15])\nR = Rotation.from_frame(Frame([1, 1, 1], [0.68, 0.68, 0.27], [-0.67, 0.73, -0.15]))\nR = Rotation.from_axis_angle_vector([-0.043, -0.254, 0.617])\nR = Rotation.from_quaternion([0.945, -0.021, -0.125, 0.303])\nR = Rotation.from_euler_angles([1.4, 0.5, 2.3], static=True, axes='xyz')\n\nprint(R)
```



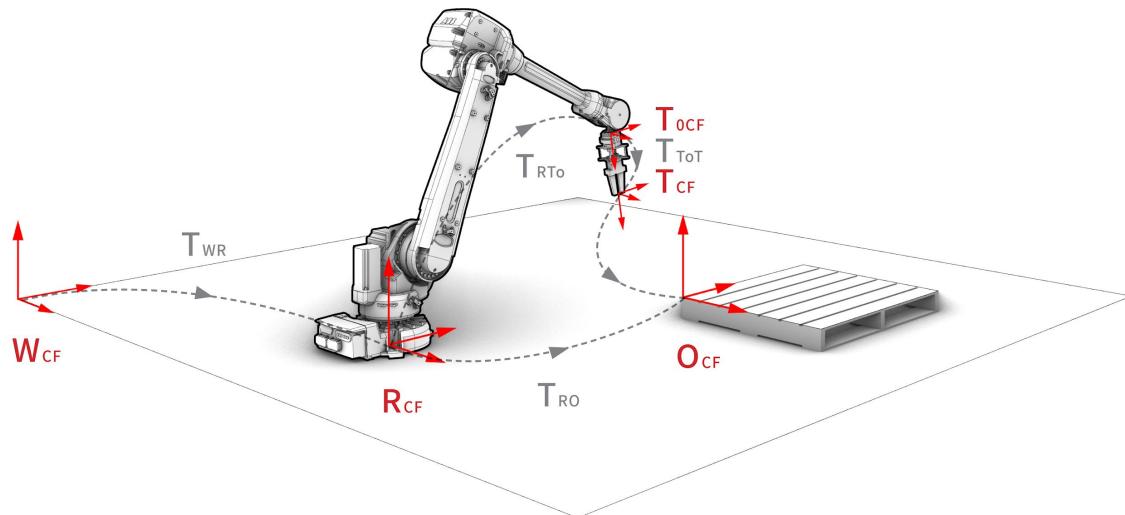
```
"""Example: Different Robot vendors use different conventions to describe TCP
orientation."""
```

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

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
```

# Robot coordinate frames



World (WCF)

Robot (RCF)

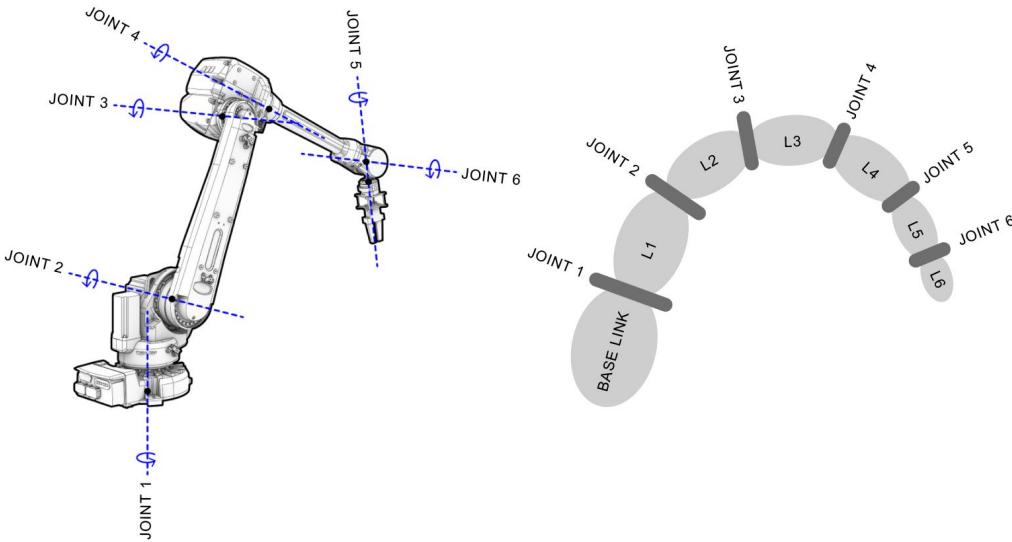
Tool0 (T0CF)

Tool (TCF)

Object (OCF)

Robot **models**

# Robotic fabrication: models



- Links connected by Joints
- Coordinate frames at joints
- Tree structure
- URDF
  - Kinematics & Dynamics
  - Visual representation
  - Collision representation
  - Semantics

# Robotic fabrication: models

compas.robots

```
Robot
Link
Inertial
Visual
Collision
Joint
Origin
Parent
Child
Dynamics
```

```
# Load robot model
from compas.robots import RobotModel

RobotModel.from_urdf_file('ur5.urdf')
```

```
<?xml version="1.0" ?>
<robot name="rfl" xmlns:xacro="http://wiki.ros.org/xacro">
  <link name="rail">
    <inertial>
      <origin xyz="0.0 0.0 0.0"/>
      <mass value="120"/>
      <inertia ixx="-4.98" ixy="-0.0" ixz="2.7" iyy="-5.95" iyz="-0.0" izz="0.828"/>
    </inertial>
    <visual>
      <geometry>
        <mesh filename="package://rfl/meshes/visual/rail.stl"/>
      </geometry>
    </visual>
    <collision>
      <geometry>
        <mesh filename="package://rfl/meshes/collision/rail.stl"/>
      </geometry>
    </collision>
  </link>
  <joint name="gantry_joint" type="prismatic">
    <parent link="rail"/>
    <child link="gantry"/>
    <origin rpy="0 0 0 " xyz="0 0 0"/>
    <axis xyz="1 0 0"/>
    <limit effort="100000" lower="0.0" upper="37.206" velocity="2.618"/>
    <dynamics damping="0.2" friction="0"/>
  </joint>
...

```



```
# Rhino
from compas.robots import *
from compas_fab.rhino import RobotArtist

r = 'ros-industrial/abb'
p = 'abb_irb6600_support'
b = 'kinetic-devel'

github = GithubPackageMeshLoader(r,p,b)
urdf = github.load_urdf('irb6640.urdf')

model = RobotModel.from_urdf_file(urdf)
model.load_geometry(github)

RobotArtist(model).draw_visual()
```

```
# Blender
from compas.robots import *
from compas_fab.blender import RobotArtist

r = 'ros-industrial/abb'
p = 'abb_irb6600_support'
b = 'kinetic-devel'

github = GithubPackageMeshLoader(r,p,b)
urdf = github.load_urdf('irb6640.urdf')

model = RobotModel.from_urdf_file(urdf)
model.load_geometry(github)

RobotArtist(model).draw_visual()
```



```
# Rhino
from compas.robots import *
from compas_fab.rhino import RobotArtist

r = 'ros-industrial/abb'
p = 'abb_irb6600_support'
b = 'kinetic-devel'

github = GithubPackageMeshLoader(r,p,b)
urdf = github.load_urdf('irb6640.urdf')

robot = Robot.from_urdf_file(urdf)
robot.load_geometry(github)

RobotArtist(robot).draw_visual()
```

```
# Blender
from compas.robots import *
from compas_fab.blender import RobotArtist

r = 'ros-industrial/abb'
p = 'abb_irb6600_support'
b = 'kinetic-devel'

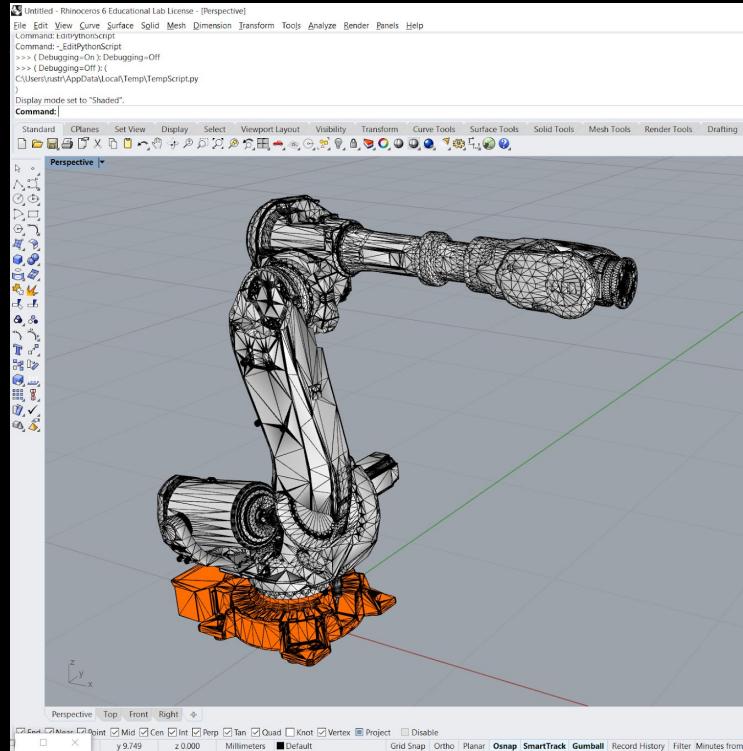
github = GithubPackageMeshLoader(r,p,b)
urdf = github.load_urdf('irb6640.urdf')

robot = Robot.from_urdf_file(urdf)
robot.load_geometry(github)

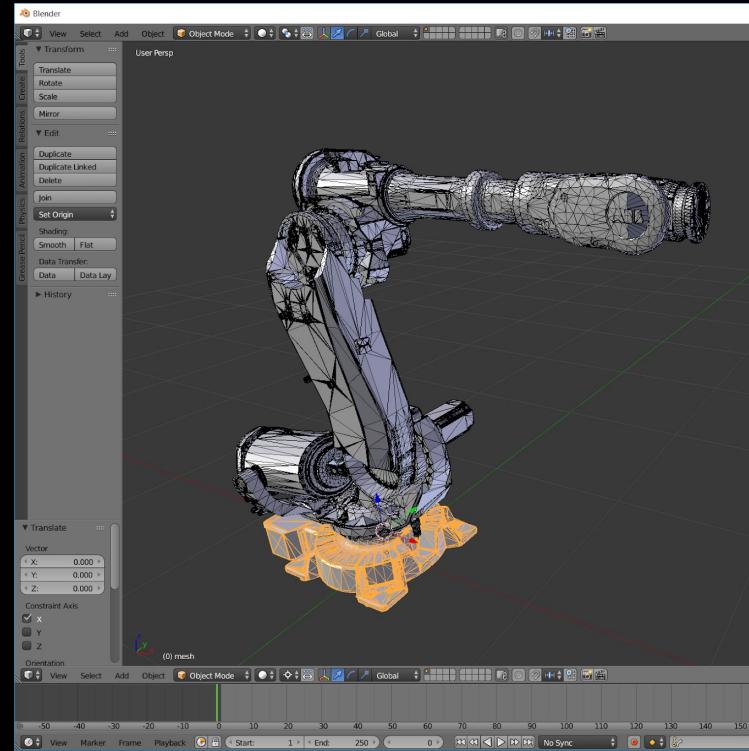
RobotArtist(robot).draw_visual()
```



## # Rhino

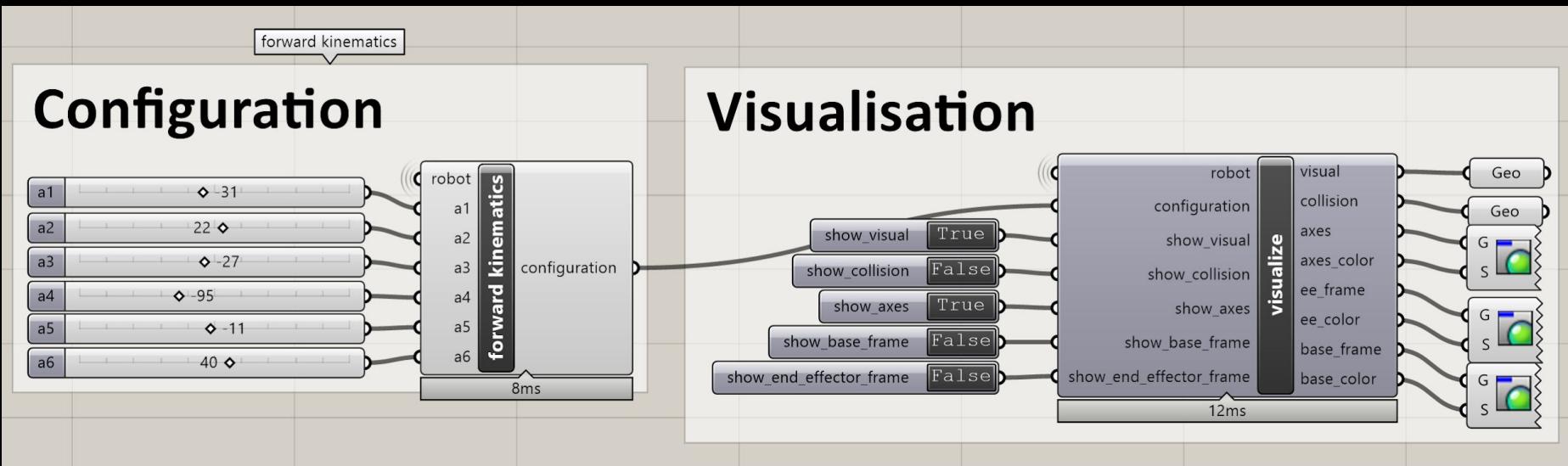


## # Blender





# Grasshopper



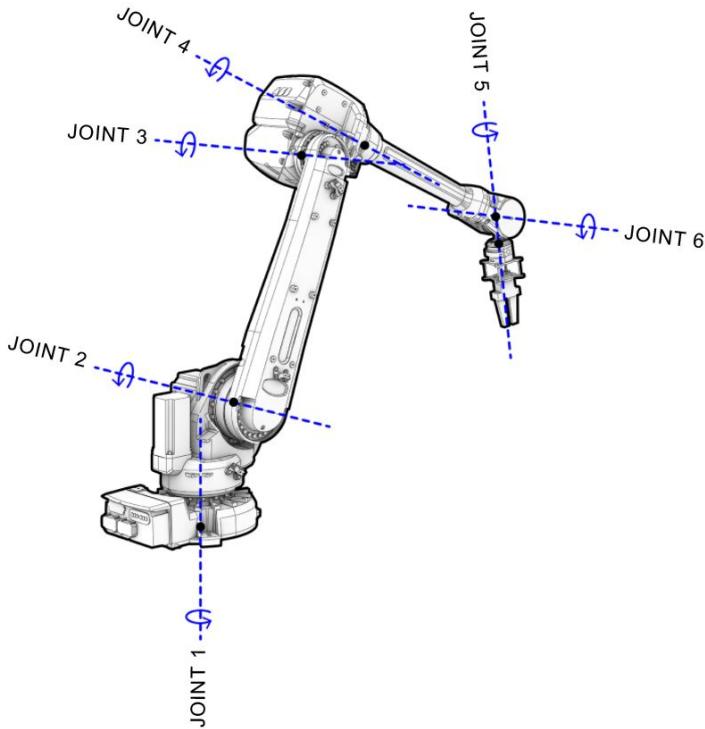


# Loading external models

```
# Set high precision to import meshes defined in meters
compas.PRECISION = '12f'

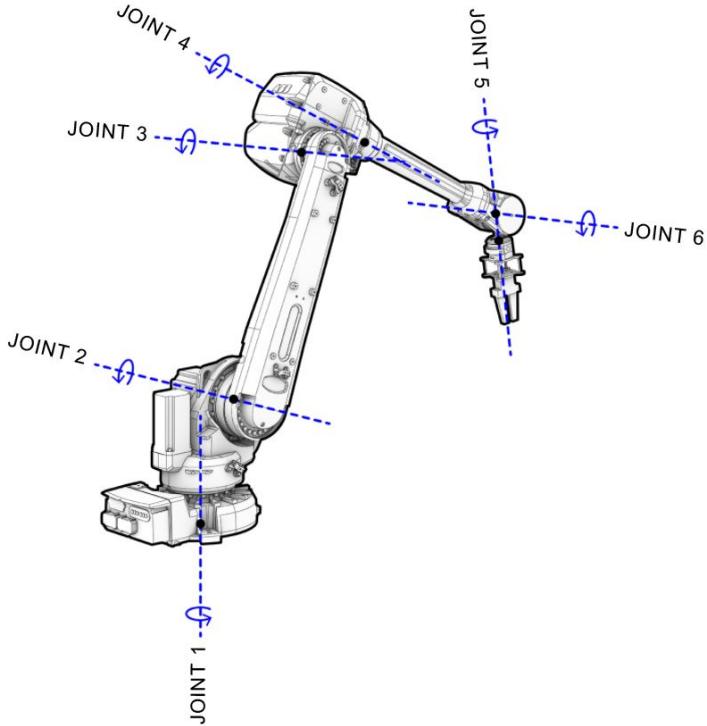
# Load robot and its geometry
with RosClient('localhost') as ros:
    robot = ros.load_robot(load_geometry=True)
    print(robot.model)
```

# Configuration



```
from compas.robots import Joint  
  
print(Joint.REVOLUTE)  
print(Joint.PRISMATIC)  
print(Joint.FIXED)
```

# Configuration



```
from math import pi
from compas_fab.robots import Configuration

values = [1.5, pi]
types = [Joint.PRISMATIC, Joint.REVOLUTE]
config = Configuration(values, types)

config =
Configuration.from_revolute_values([pi/2, 0., 0.,
pi/2, pi, 0])

config =
Configuration.from_prismatic_and_revolute_values(
[8.312], [pi/2, 0., 0., 0., 2*pi, 0.8])
```



# Building your own robot

```
# create robot
robot = RobotModel("robot", links=[], joints=[])

# add links
link0 = robot.add_link("world")
link1 = robot.add_link("link1", visual_mesh=mesh1,)
link2 = robot.add_link("link2", visual_mesh=mesh2,)

# add the joints between the links
robot.add_joint("joint1", Joint.REVOLUTE, link0, link1, origin1, axis1)
robot.add_joint("joint2", Joint.REVOLUTE, link1, link2, origin2, axis2)
```

# Backends

# Robotic fabrication: backends

V-REP

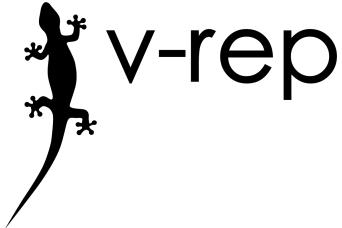
Robot simulator

ROS

Robot Operating  
System

Moveit!

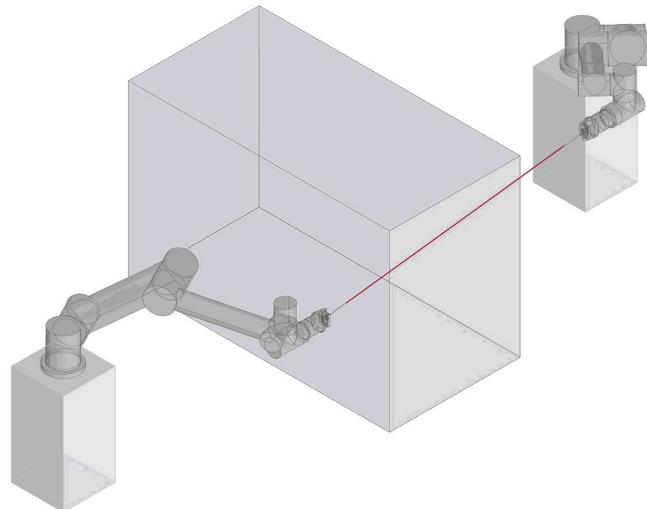
Motion Planning  
Framework



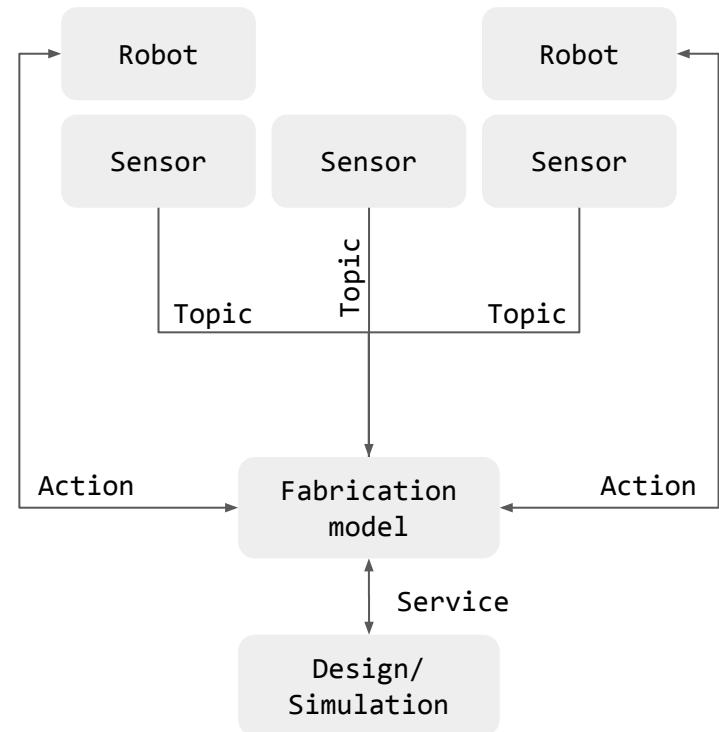
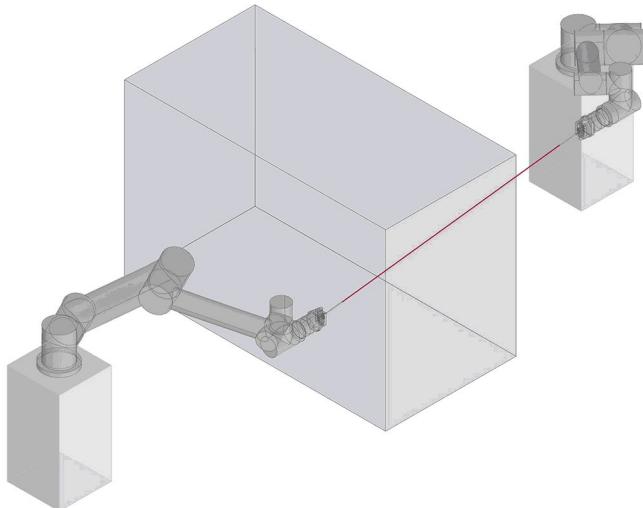
 ROS

Open Source Robotics Foundation

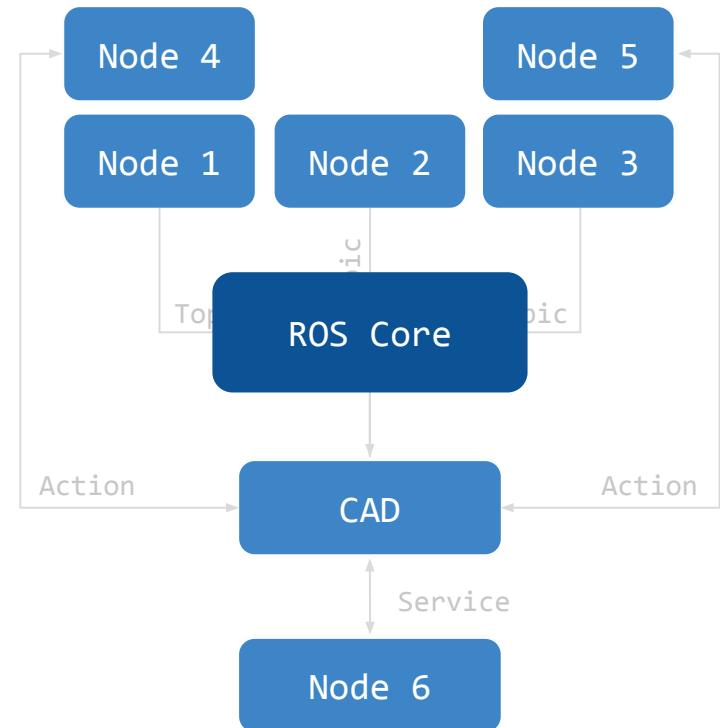
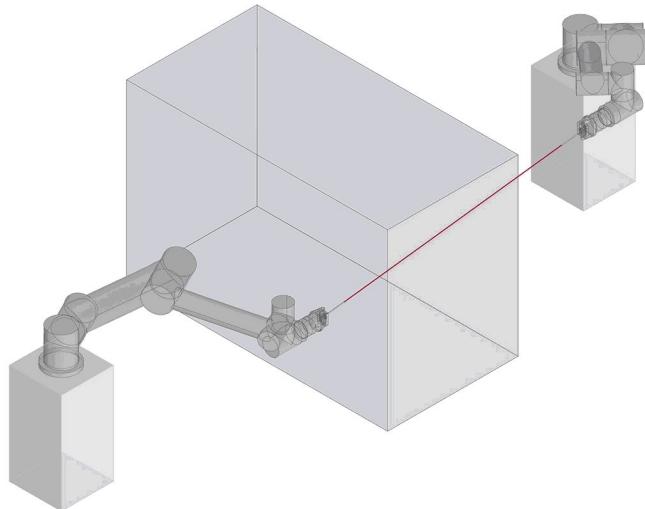
# Fabrication processes



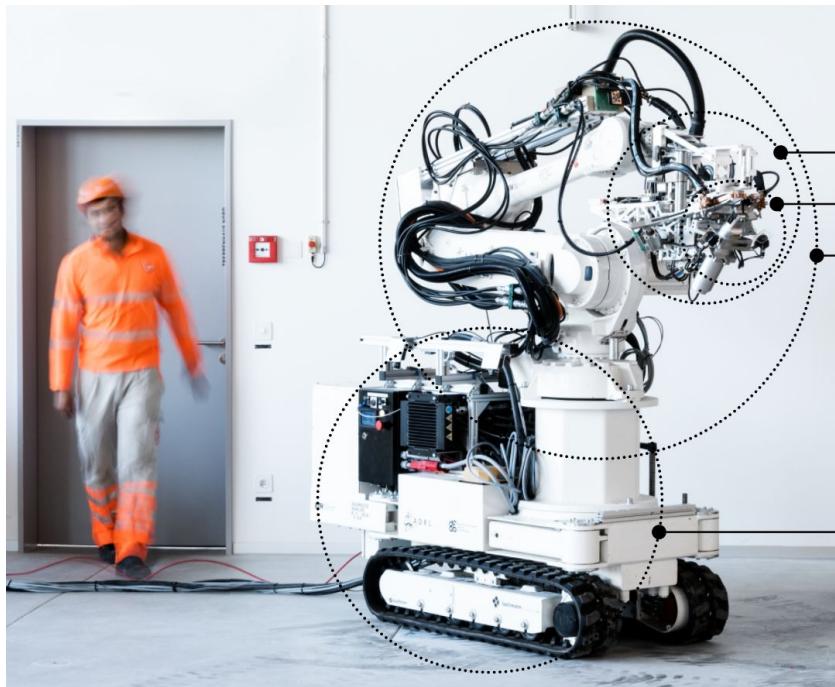
# Fabrication processes



# Fabrication processes



# What is ROS?



- ROS Node 1: End Effector
- ROS Node 2: Localization
- ROS Node 3: Robot Arm
- ROS Node 4: Mobile Base

# What is ROS?

## plumbing

- Process management
- Interprocess communication
- Device drivers

## tools

- Simulation
- Visualization
- Graphical UI
- Data logging

## capabilities

- Control
- Planning
- Perception
- Mapping
- Manipulation

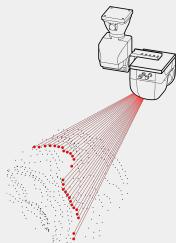
## ecosystem

- Package organization
- Software distribution
- Community
- Documentation
- Tutorials

# ROS Concepts

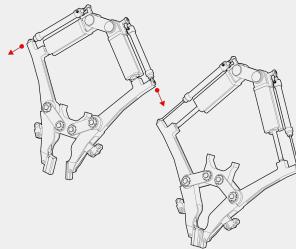
## Topics

- Nodes communicate over topics
- **Publish/subscribe** model
- **One-way** data stream
- e.g: *robot states, sensor data*



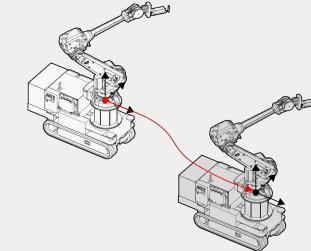
## Services

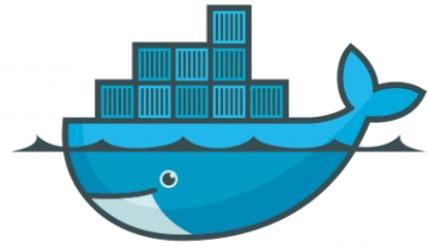
- **Blocking** call per request
- Request/response model
- Short trigger or calculation
- e.g: *calculate path, open gripper*



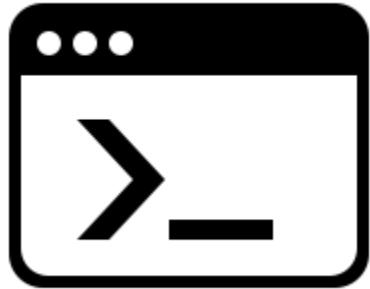
## Actions

- **Non-blocking** requests
- Goal-oriented and cancellable
- Implemented with topics
- e.g: *navigation, motion execution*

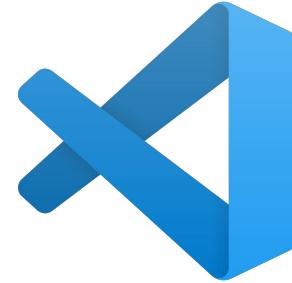




*docker*



`docker-compose up -d`



Right-click → Compose Up

# docker/docker-compose.yml

<http://localhost:8080/vnc.html?resize=scale&autoconnect=true>

<http://192.168.99.100:8080/vnc.html?resize=scale&autoconnect=true>



# Verify connection to ROS

```
from compas_fab.backends import RosClient  
  
with RosClient('localhost') as client:  
    print('Connected:', client.is_connected)
```



# Verify connection to ROS

```
from compas_fab.backends import RosClient  
  
with RosClient('192.168.99.100') as client:  
    print('Connected:', client.is_connected)
```

# Publish to topics



## Publish to topic: **connect**

```
import time

from roslibpy import Topic
from compas_fab.backends import RosClient

with RosClient('localhost') as client:
    talker = Topic(client, '/messages', 'std_msgs/String')
    talker.advertise()

    while client.is_connected:
        print('Sending...')
        talker.publish({'data': 'Hello'})
        time.sleep(1)
```



## Publish to topic: **advertise**

```
import time

from roslibpy import Topic
from compas_fab.backends import RosClient

with RosClient('localhost') as client:
    talker = Topic(client, '/messages', 'std_msgs/String')
    talker.advertise()

    while client.is_connected:
        print('Sending...')
        talker.publish({'data': 'Hello'})
        time.sleep(1)
```



# Publish to topic: **publish**

```
import time

from roslibpy import Topic
from compas_fab.backends import RosClient

with RosClient('localhost') as client:
    talker = Topic(client, '/messages', 'std_msgs/String')
    talker.advertise()

    while client.is_connected:
        print('Sending...')
        talker.publish({'data': 'Hello'})
        time.sleep(1)
```

# Subscribe to topics



## Subscribe to topic: **connect**

```
import time

from roslibpy import Topic
from compas_fab.backends import RosClient

def receive_message(message):
    print('Heard talking: ' + message['data'])

with RosClient('localhost') as client:
    listener = Topic(client, '/messages', 'std_msgs/String')
    listener.subscribe(receive_message)

    while client.is_connected:
        time.sleep(1)
```



## Subscribe to topic: **subscribe**

```
import time\n\nfrom roslibpy import Topic\nfrom compas_fab.backends import RosClient\n\n\ndef receive_message(message):\n    print('Heard talking: ' + message['data'])\n\nwith RosClient('localhost') as client:\n    listener = Topic(client, '/messages', 'std_msgs/String')\n    listener.subscribe(receive_message)\n\nwhile client.is_connected:\n    time.sleep(1)
```

Robot **planning**

## **compas.robots.RobotModel**

Describes the kinematics, linkage geometry and dynamics of a robot cell.

## **compas\_fab.robots.Robot**

Integrates a robot model, additional semantic information for planning, a backend client and artist instance.

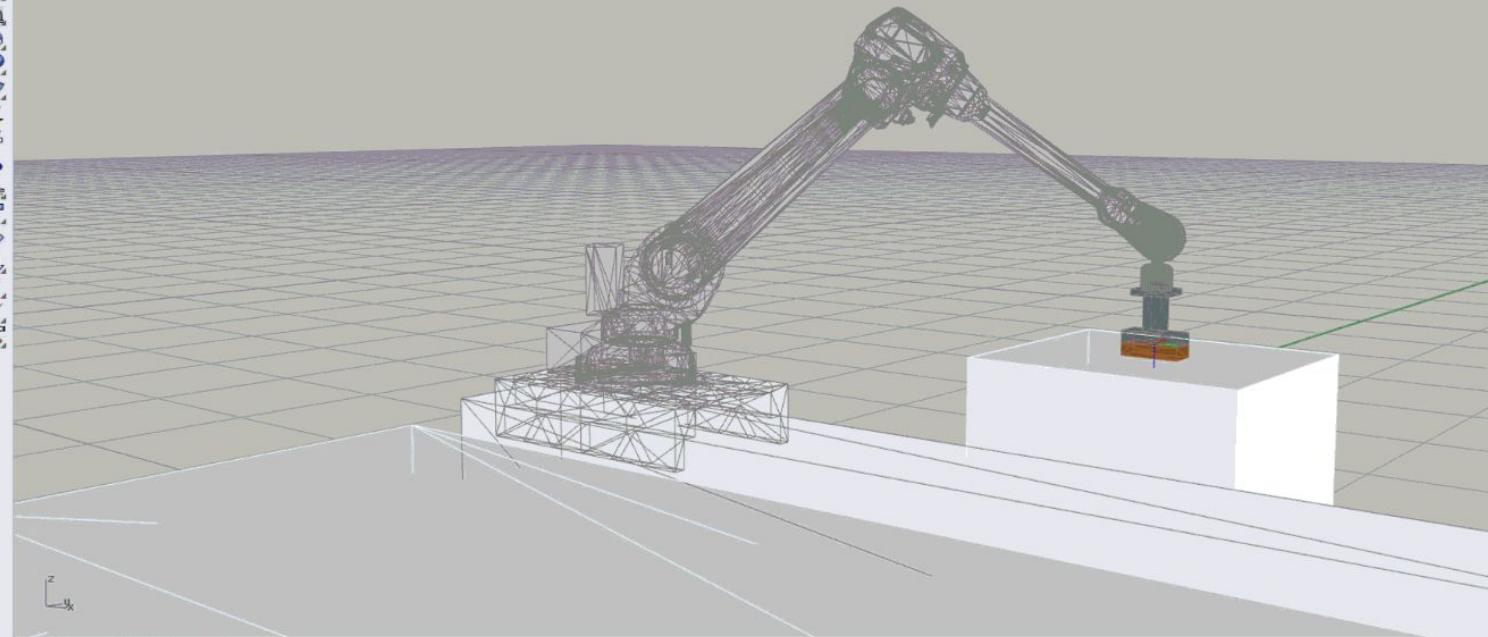
# Planning

➤ **MoveIt!**

Kinematics

Motion planning

Planning scene



Perspective Top Front Right

 End  Near  Point  Mid  Cen  Int  Perp  Tan  Quad  Knot  Vertex  Project  Disable

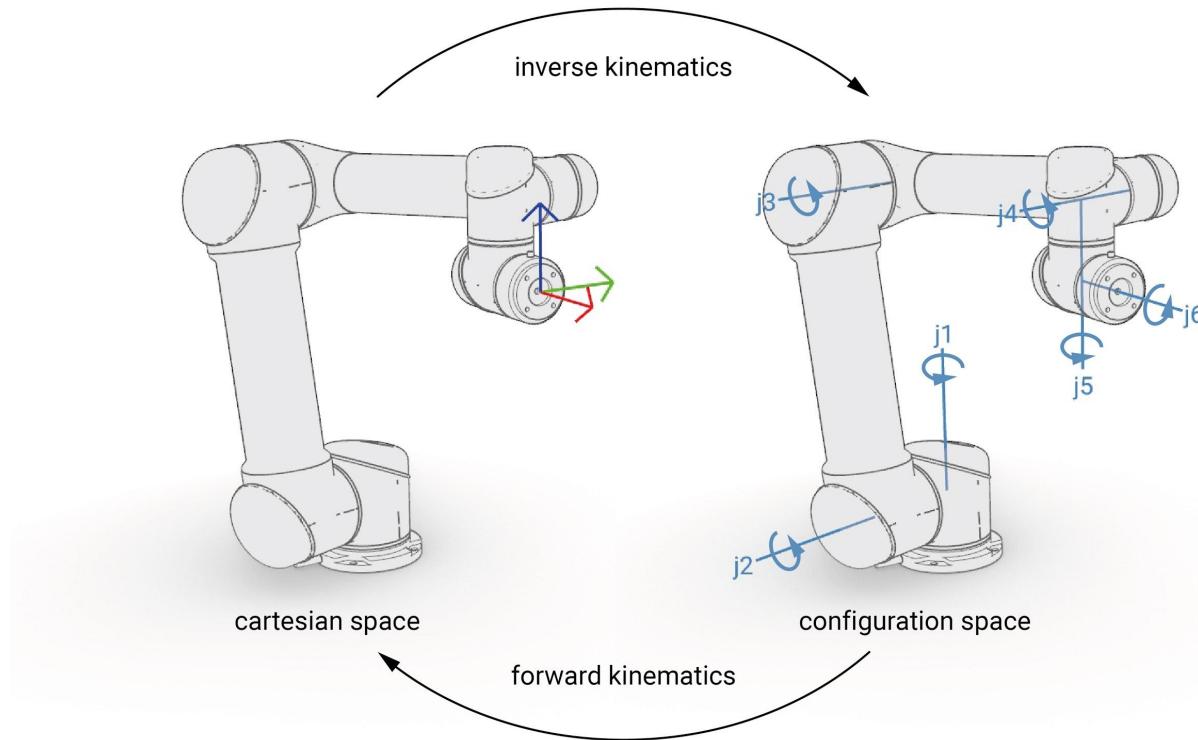
CPlane x 15.11 y -15.12 z 0.00 Meters Default Grid Snap Ortho Planar Osnap SmartTrack Gumball Record History Filter Memory use: 880 MB



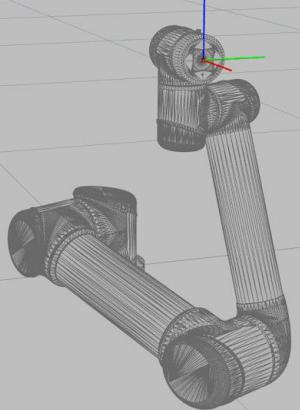
Viewport	
Title	Perspective
Width	1669
Height	937
Projection	Perspective
Camera	
Lens Length	50.0
Rotation	0.0
X Location	4.642
Y Location	-4.352
Z Location	1.970
Location	Place...
Target	
X Target	0.837
Y Target	0.248
Z Target	1.342
Location	Place...

Wallpaper	
Filename	(none)
Show	<input checked="" type="checkbox"/>
Gray	<input checked="" type="checkbox"/>

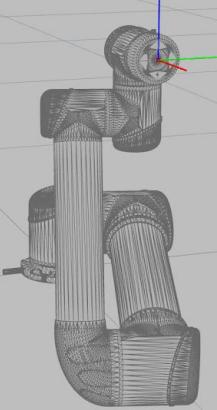
# Kinematics



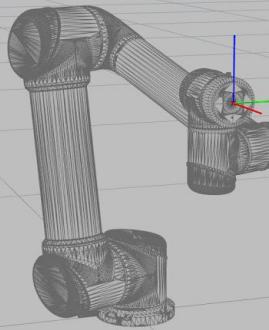
(3.56, 2.88, 2.12, 4.42, -5.13, 6.28)



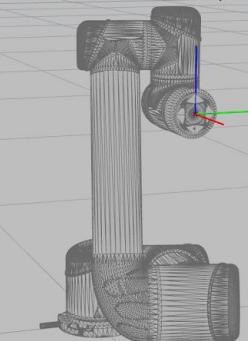
(6.0, -6.02, -2.12, -1.28, 4.99, 6.28)



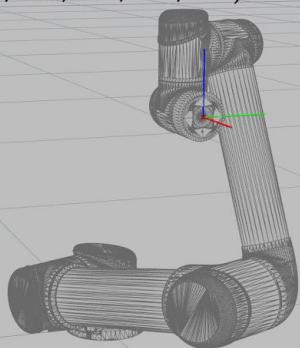
(3.56, 4.86, -2.12, -5.88, 1.15, -6.28)



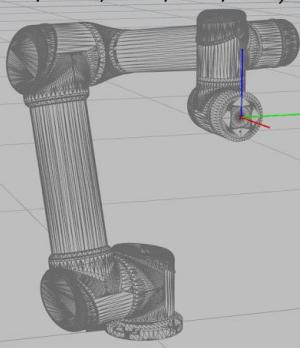
(6.0, -0.19, -1.68, 1.88, -4.99, 3.14)



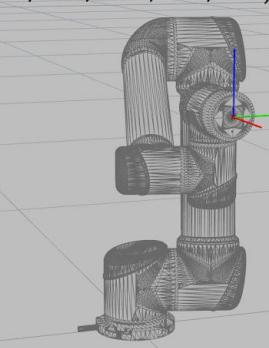
(-2.72, -2.95, 1.68, 1.27, 5.13, 3.14)



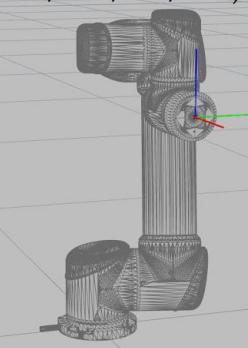
(-2.72, 4.93, -1.68, -3.25, 5.13, 3.14)



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



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





# Forward kinematics

For a given configuration, calculate the frame of the end-effector expressed in the world coordinate frame.

- **Input:** Configuration, i.e. `compas_fab.robots.Configuration()`
- **Output:** Frame in WCF, i.e. `compas.geometry.Frame()`
- **Usage:**

```
config = Configuration.from_revolute_values([0, 0, 0, 3.14, 0, 0])
frame_wcf = robot.forward_kinematics(config)
```



# Inverse kinematics

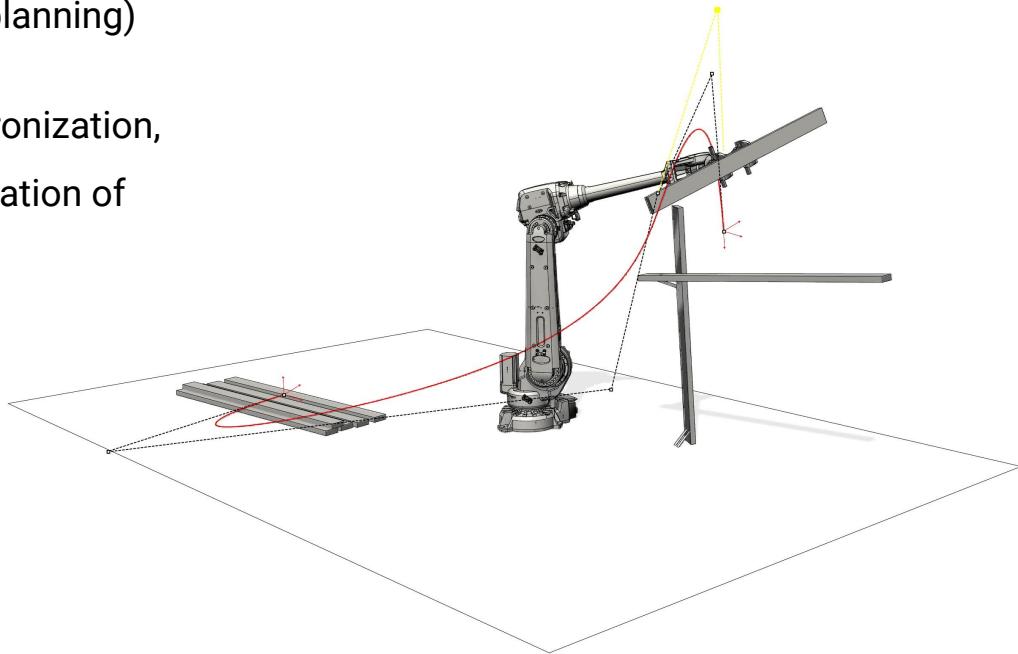
For a given end-effector frame expressed in world coordinate frame, find the corresponding robot configuration(s) to reach it.

- **Input:** Frame in WCF and start config, i.e. `compas.geometry.Frame()`
- **Output:** Configuration, i.e. `compas_fab.robots.Configuration()`
- **Usage:**

```
frame_wcf = Frame([0.3, 0.1, 0.5], [1, 0, 0], [0, 1, 0])
start_config = robot.init_configuration()
config = robot.inverse_kinematics(frame_wcf, start_config)
```

# Motion planning

- Collision checking (= path planning)
- Trajectory checking (synchronization, consider speed and acceleration of moving objects)



# Motion planning

## Collision checks

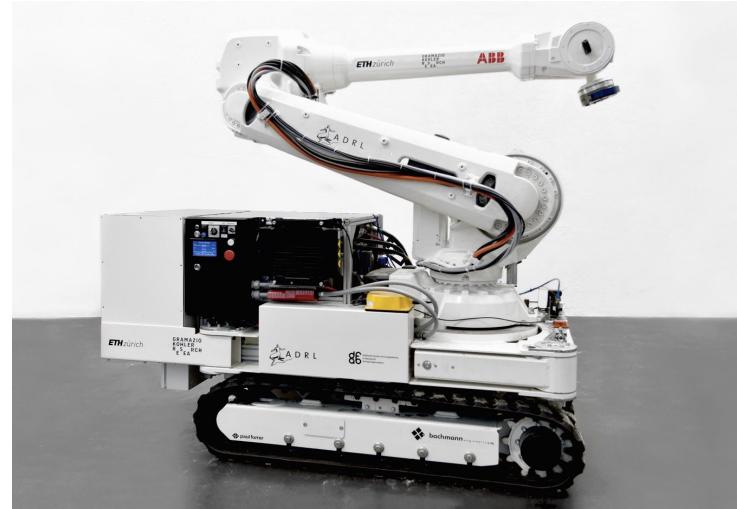
- Intricate positions (spatial assembly)
- Multiple robots working closely



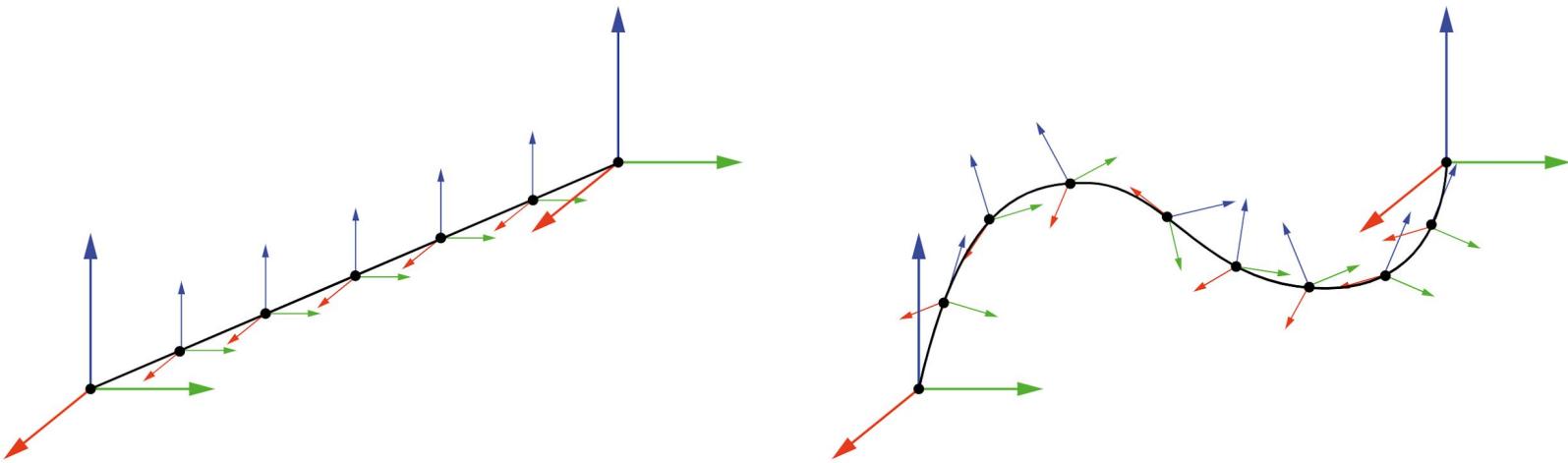
# Motion planning

## Trajectory checks

- Synchronisation
- Continuous processes



# Cartesian motion vs free-space motion





# Plan cartesian motion

For a given list of waypoints, calculate a linear trajectory in task space.

- **Input:** List of frame in WCF and start config, i.e. `compas.geometry.Frame()`
- **Output:** Joint trajectory, i.e. `compas_fab.robots.JointTrajectory()`
- **Usage:**

```
f1 = Frame([0.3, 0.1, 0.5], [1, 0, 0], [0, 1, 0])
f2 = Frame([0.6, 0.1, 0.4], [1, 0, 0], [0, 1, 0])
start_config = robot.init_configuration()

trajectory = robot.plan_cartesian_motion([f1, f2], start_config)
```



# Plan kinematic motion

For a given start configuration and a goal (defined as constraints), calculate a joint trajectory in joint space.

- **Input:** Goal constraints and start cfg, e.g. `compas_fab.robots.PositionConstraint()`
- **Output:** Joint trajectory, i.e. `compas_fab.robots.JointTrajectory()`
- **Usage:**

```
pc = robot.constraints_from_frame(...)  
start_config = robot.init_configuration()
```

```
trajectory = robot.plan_motion(pc, start_config, planner_id='RRT')
```

# Defining constraints

- **JointConstraint**

Constrains the value of a joint to be within a certain bound.

- **OrientationConstraint**

Constrains a link to be within a certain orientation.

- **PositionConstraint**

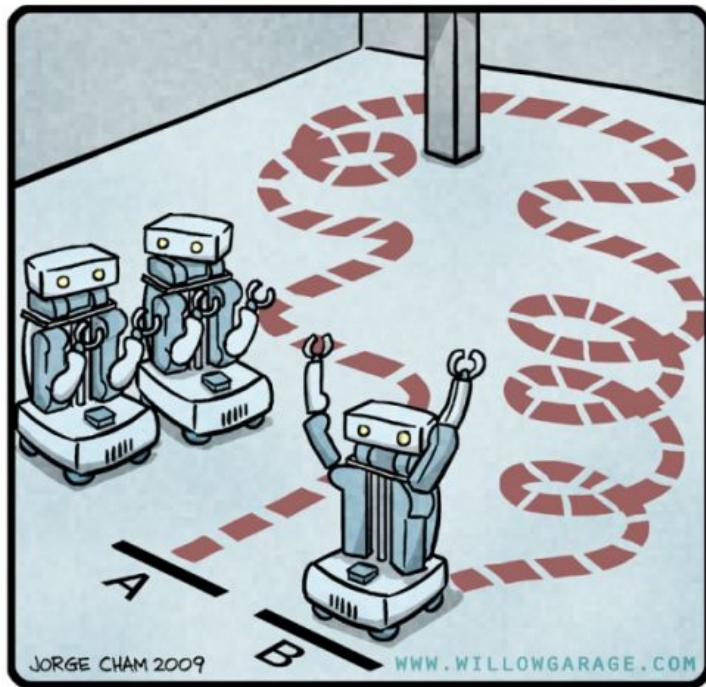
Constrains a link to be within a certain bounding volume.

```
frame = Frame([0.4, 0.3, 0.4], [0, 1, 0], [0, 0, 1])
tolerance_position = 0.001
tolerance_axes = [math.radians(1)] * 3

start_configuration = Configuration.from_revolute_values([-0.042, 4.295, 0, -3.327, 4.755, 0.])
group = robot.main_group_name

# create goal constraints from frame
goal_constraints = robot.constraints_from_frame(frame,
                                                 Tolerance_position,
                                                 Tolerance_axes,
                                                 group)
```

# R.O.B.O.T. Comics



"HIS PATH-PLANNING MAY BE  
SUB-OPTIMAL, BUT IT'S GOT FLAIR."



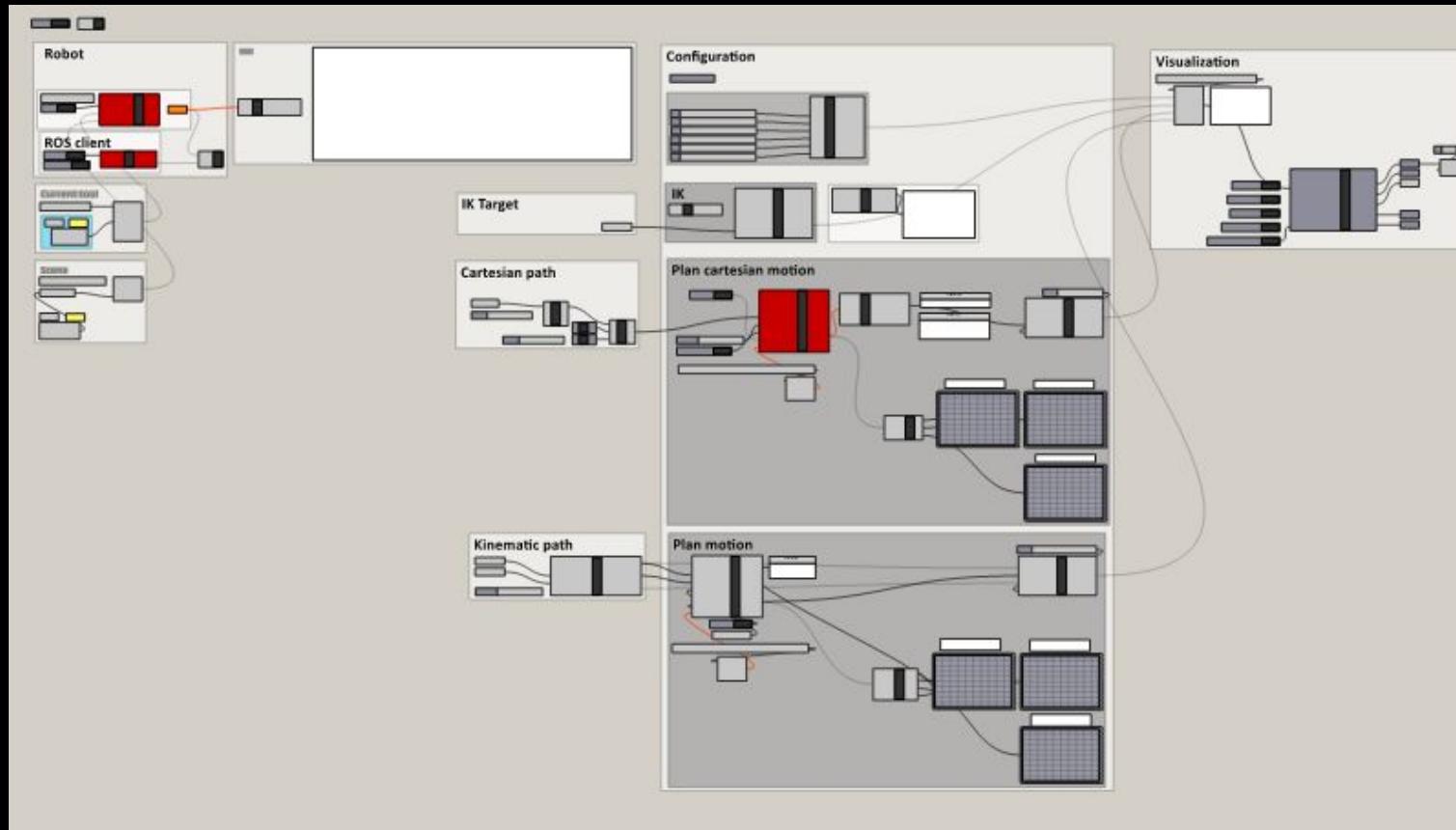
# Planning scene operations

Add/append/remove collision meshes (i.e. obstacles) and add/remove attached collision meshes (i.e. meshes attached to the end effector).

- **Usage:**

```
scene = compas_fab.robots.PlanningScene(robot)
scene.add_collision_mesh(..)
scene.remove_collision_mesh(..)
scene.append_collision_mesh(..)

scene.add_attached_collision_mesh(..)
scene.remove_attached_collision_mesh(..)
scene.attach_collision_mesh_to_robot_end_effector(..)
```



Robot **control**

# Control

➤ **MoveIt!**



C O M P A S   R R C

Offline control

Online real-time control

Online non-real-time control

# Trajectory execution

**High level interface**, that hands over execution to underlying drivers.

Allows to execute a previously computed joint trajectory.

- **Input:** Joint trajectory, i.e. `compas_fab.robots.JointTrajectory()`
- **Output:** Future result, i.e. `compas_fab.backends.CancellableFutureResult()`
- **Usage:**

```
wait = robot.client.execute_joint_trajectory(trajectory)
```

```
wait.result()
```



# Joint trajectory action

Similar to MoveIt! Executor, but lower level. It bypasses MoveIt! pipeline, but serves the same purpose and has almost identical interface.

- **Input:** Joint trajectory, i.e. `compas_fab.robots.JointTrajectory()`
- **Output:** Future result, i.e. `compas_fab.backends.CancellableFutureResult()`
- **Usage:**

```
wait = robot.client.follow_joint_trajectory(trajectory)
```

```
wait.result()
```



# ROS-I Drivers

Different drivers implement different control strategies:

- UR: real-time control
- Franka Emika: real-time control
- ABB: non-realtime downloading driver
- Staubli: non-real-time streaming driver

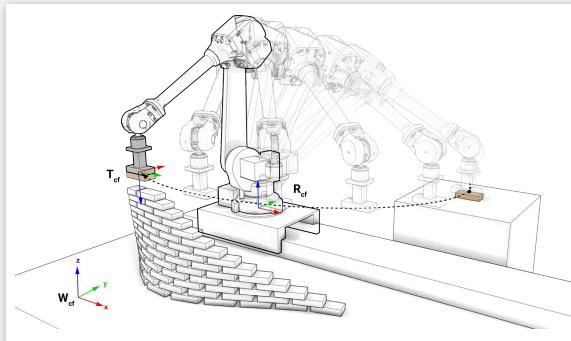
Control: ROS-Industrial drivers

# ROS-I Drivers

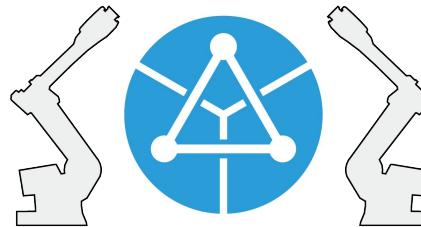


Some drivers' features are **not** accessible over a unified interface.

Use **RosClient** and **Topics & Services** to use them.



compas\_fab



compas\_rrc

## FAB 0.12

New backend extensibility  
PyBullet integration  
More backends

## RRC 1.0

Cyclic instruction support  
New execution levels  
More compact protocol

# Thanks!

