Installation

```
conda install compas_fab
```

Anaconda/Miniconda must be installed and the conda-forge channel must be added. Use virtual environments to avoid dependency issues.

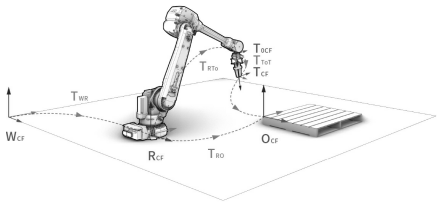
Frames

`Frame.worldXY()`	World XY frame.
`Frame(pt, xaxis, yaxis)`	Create new frame with origin and vectors `x` and `y`.
`Frame(pt1, pt2, pt3)`	Create new frame from 3 points.
`Frame.from_euler_angles(y, p, r)`	Create new frame from euler angles (yaw, pitch, roll).
`frame.point`	Origin of the frame.
`frame.normal`, `frame.zaxis`	Normal of the frame.
`frame.quaternion`	Rotation as quaternion.
`frame.axis_angle_vector`	Rotation as axis-angle vector.
`frame.euler_angles()`	Rotation as euler angles.
`frame.euler_angles(False)`	Rotation as non-static euler angles.

Frames belong to the robotics fundamentals. Each joint contains a frame and there are several frames as coordinate systems involved (eg. `wcf`, `rcf`).

```
from compas.geometry import Frame
```

Frames as cartesian coordinate systems



Coordinate systems

`wcf`	Root coordinate frame of the world.
`rcf`	Robot coordinate frame, usually on the base of the robot.
`t0cf`	Tool0 coordinate frame on the last link of the robot.
`tcf`	Tool coordinate frame on the tool of the robot.
`ocf`	Object coordinate frame, origin of the work area.
`rcf.to_world_coords(pt)`	Transform point in local coordinates of `rcf` to world coordinates.
`rcf.to_local_coords(pt)`	Transform point in world coordinates to local coordinates of `rcf`.

Transformations

`Transformation.from_frame(f)`	Create transformation from world XY to `f` frame.
`Translation([10, 5, 0])`	Create translation along `x=10.0, y=5.0, z=0.0` vector.

Transformations (cont)

`t.inverse()`	Calculate inverse of `t` transformation.
`pt.transform(t)`	In-place transform point `pt` with `t` transformation.
`pt.transform(t * t2 * t3)`	In-place transform point `pt` with `t`, `t2` and `t3` transformations.
`pt2 = pt.transformed(t)`	Return a transformed copy of point `pt` with `t` transformation.

`Transformation` represents a 4x4 transformation matrix. `Translation`, `Scale`, `Reflection`, `Shear` and `Projection` are specialized sub-classes of it.

```
from compas.geometry import Transformation, Translation, Rotation # , ...
```

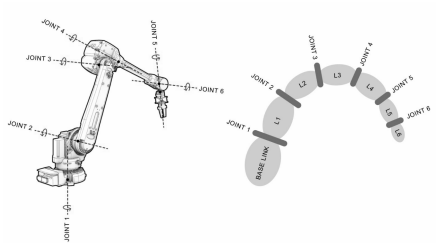
Geometric primitives

`Point(3, 2, 5)` or `[3, 2, 5]`	3D point with `x=3.0, y=2.0, z=5.0` coordinates.
`Vector(1, 0, 0)` or `[1, 0, 0]`	Vector `x=1.0, y=0.0, z=0.0`
`Frame(point, xaxis, yaxis)` or `(point, xaxis, yaxis)`	Frame at `point` origin and `xaxis` and `yaxis` vectors.

In COMPAS, geometric primitives can be created either as objects or as iterables (lists, tuples, etc).

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

Robot models



Parts of robot model

`Link`	Contains visual & collision meshes, inertial info, etc.
`Joint`	Contains parent, child links, origin frame, etc.
`RobotModel`	Root of robot model tree.

Robot models are a tree structure of **links** and **joints** based on `URDF` format.

```
from compas.robots import RobotModel, Link, Joint
```

Joint types and units

`Joint.PRISMATIC`	Meters
`Joint.REVOLUTE`	Radians
`Joint.CONTINUOUS`	Radians
`Joint.FIXED`	-

Configuration examples

```
Configuration([.5, pi], [Joint.PRISMATIC, Joint.REVOLUTE])
Configuration.from_revolute_values([0, 0, pi, 0, 0, 0])
Configuration.from_prismatic_and_revolute_values([8.3], [0.0] * 6)
```

Configuration describes the positions of each of the joints of a robot model in its corresponding unit.

```
from compas_fab.robots import Configuration
```

ROS backend

ROS

Open Source Robotics Foundation

Start ROS backend

Download or create a `docker-compose.yml` file (examples) and start up with:

```
docker-compose up -d
```

Load robot from ROS

```
from compas_fab.backends import RosClient
with RosClient('localhost') as client:
    robot = client.load_robot()
```

If using Docker Toolbox, replace localhost with **192.168.99.100**.

Robotic planning with MoveIt!

MoveIt!

Forward Kinematics (FK)

Input	`Configuration`.
Output	`Frame` in `rcf`.

Calculate end-effector frame in `rcf` for a given configuration.

Code example

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

Inverse Kinematics (IK)

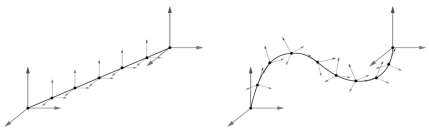
Input	`Frame` in `wcf` and start `Configuration`.
Output	`Configuration`

Calculate possible configuration(s) for a given frame in `wcf`.

Code example

```
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)
```

Path planning



Cartesian vs free-space planning

Plan cartesian path

Input	List of `Frame` in `wcf` and start `Configuration`
Output	`JointTrajectory`

Calculate linear joint trajectory for a given list of waypoints defined by frames.

This might return partial solutions, `fraction` attribute indicates degree of completion, e.g. `trajectory.fraction = 0.5` means 50% of trajectory completed.

Code example

```
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 free-space motion

Input	List of goal `Constraint` and start `Configuration`.
Output	`JointTrajectory`.

Calculate joint trajectory for a given start configuration and a list of goal constraints.

```
from compas_fab.robots import JointTrajectory,
Constraint, JointConstraint, PositionConstraint,
OrientationConstraint
```

Code example

```
pc = robot.constraints_from_frame(f1)
start_config = robot.init_configuration()
trajectory = robot.plan_motion(pc, start_config,
planner_id='RRT')
```

Planning scene operations

```
scene = PlanningScene(robot)
scene.add_collision_mesh(CollisionMesh(mesh,
'floor'))
scene.remove_collision_mesh('floor')
scene.append_collision_mesh(CollisionMesh(mesh,
'brick'))
```

Append will group multiple meshes under the same name, and they can be removed with a single call to remove for that name.

```
from compas_fab.robots import PlanningScene,
CollisionMesh
```

Attach meshes to robot

```
scene = PlanningScene(robot)
gripper = CollisionMesh(mesh, 'gripper')
scene.attach_collision_mesh_to_robot_end_effecto-
r(gripper)
scene.remove_attached_collision_mesh('gripper')
beam = CollisionMesh(mesh, 'beam')
acm = AttachedCollisionMesh(beam, 'ee_link',
['ee_link'])
scene.add_attached_collision_mesh(acm)
```

These operations can be used to attach an end-effector geometry, or to attach an element to the end-effector itself.

```
from compas_fab.robots import PlanningScene,
CollisionMesh, AttachedCollisionMesh
```