iCIR tutorial

충남대학교 Chungnam National University

iCIR Lab
Intelligent Control & Interactive Robotics Lab

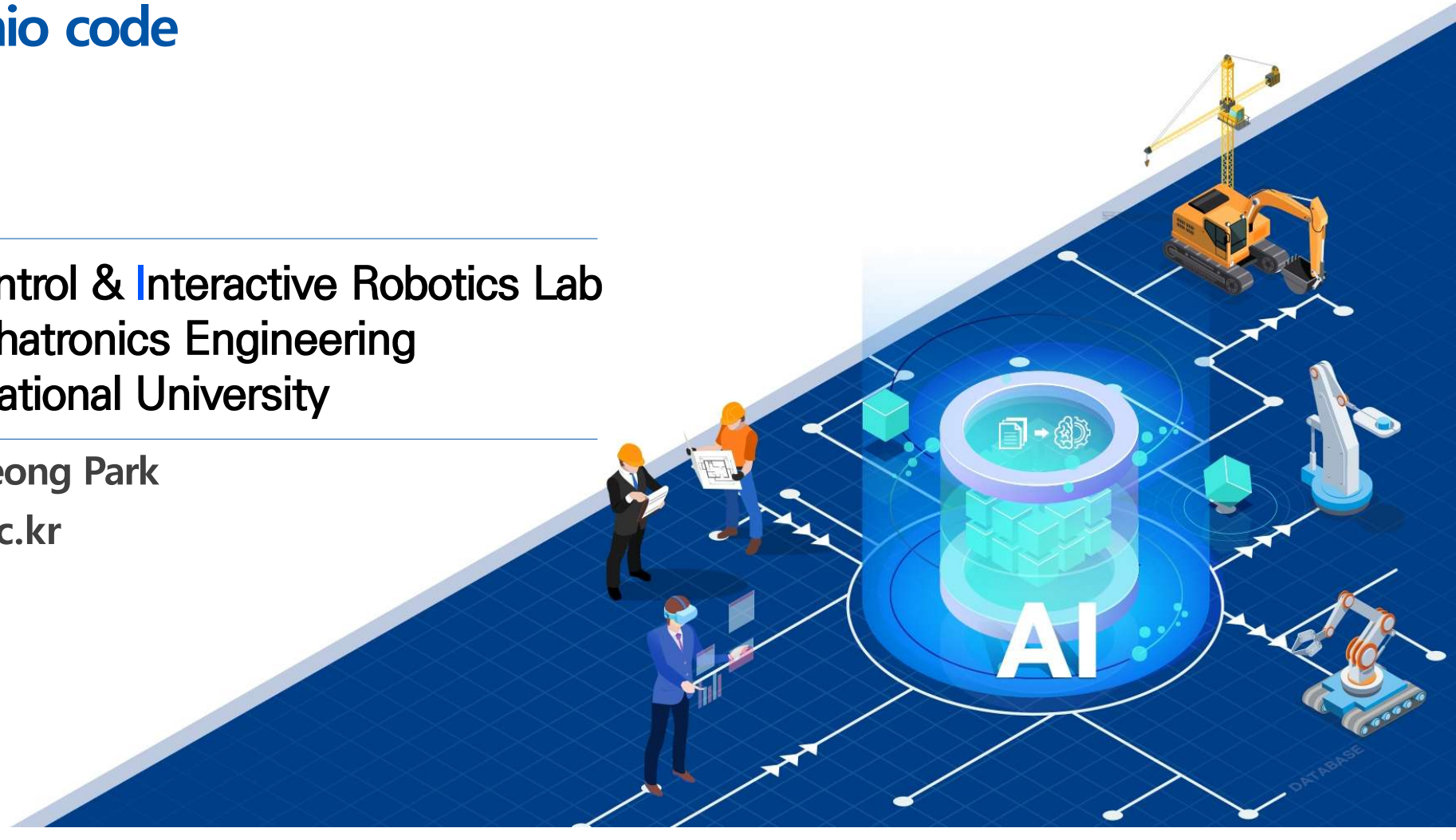
iCIR tutorial

- Pinocchio code

intelligent Control & Interactive Robotics Lab
Dept. of Mechatronics Engineering
Chungnam National University

Professor: Jinseong Park

js.park@cnu.ac.kr



Tutorial ROS package

- Code download

  git clone https://github.com/iCIRLab/icir_tutorial_pinocchio.git

- Code install

  catkin_make &

  echo "source /home/jspark/icir_tutorial_ws/devel/setup.bash" >> ~/.bashrc

- Code run (use “Tab” key)

  roslaunch icir_tutorial_pinocchio icir_tutorial_pinocchio_simulation.launch

Tutorial ROS package

Cmakelist.txt

```
project(icir_tutorial_pinocchio)
...

### library
SET(${PROJECT_NAME}_HEADERS
  utility/urdf_to_pin.hpp
  utility/math_functions.hpp
)
SET(${PROJECT_NAME}_SOURCES
  utility/urdf_to_pin.cpp
  utility/math_functions.cpp
)

ADD_SOURCE_GROUP(${PROJECT_NAME}_HEADERS)
ADD_SOURCE_GROUP(${PROJECT_NAME}_SOURCES)
ADD_LIBRARY(${PROJECT_NAME} SHARED ${${PROJECT_NAME}_SOURCES} ${${PROJECT_NAME}_HEADERS})
TARGET_INCLUDE_DIRECTORIES(${PROJECT_NAME} SYSTEM PUBLIC $<INSTALL_INTERFACE:include> ${EIGEN3_INCLUDE_DIR})
TARGET_LINK_LIBRARIES(${PROJECT_NAME} PUBLIC pinocchio::pinocchio ${pinocchio_LIBRARIES} -lpthread)

### exe
add_executable(${PROJECT_NAME}_simulation src/icir_tutorial_pinocchio_sim.cpp)
target_link_libraries(${PROJECT_NAME}_simulation ${PROJECT_NAME} ${catkin_LIBRARIES} )
TARGET_INCLUDE_DIRECTORIES(${PROJECT_NAME}_simulation SYSTEM PUBLIC $<INSTALL_INTERFACE:include>)
```

Tutorial ROS package

Package.xml

```
<?xml version="1.0"?>
<package format="2">
  <name>icir_tutorial_pinocchio</name>
  <version>0.0.1</version>
  <description>The CNU icir_pinocchio package for tutorial</description>

  <maintainer email="js.park@cnu.ac.kr">Jinseong Park</maintainer>

  <license>MIT</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>eigen</build_depend>

  <depend>controller_interface</ depend >
  <depend>dynamic_reconfigure</ depend >
  <depend>geometry_msgs</depend >
  <depend>hardware_interface</depend >
  <depend>pluginlib</depend >
  <depend>realtime_tools</depend >
  <depend>mujoco_ros</depend >
  <depend>pinocchio</depend >

  <exec_depend>message_runtime</exec_depend>
  <exec_depend>rospy</exec_depend>
  <exec_depend>roscpp</exec_depend>
  <exec_depend>std_msgs</exec_depend>
  <exec_depend>message_generation</exec_depend>
</package>
```

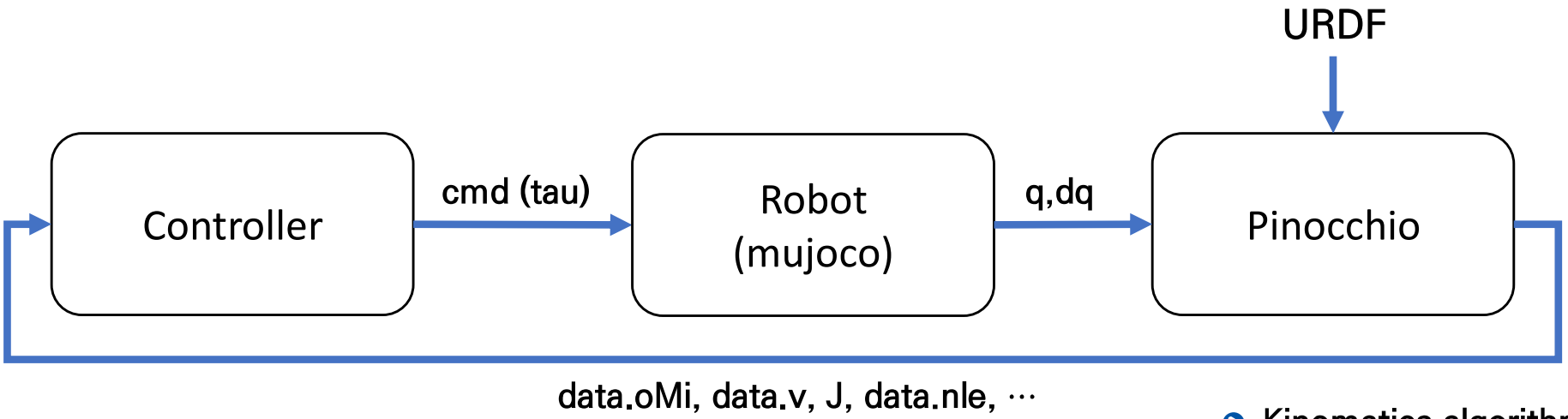
Tutorial ROS package

- launch file & urdf file (mujoco xml, pinocchio urdf, rviz urdf) & robot_state_publisher

```
<launch>
  <arg name="pub_mode" default="false"/>

  <!-- mujoco -->
  <node name="mujoco_ros" pkg="mujoco_ros" type="mujoco_ros" required="true" respawn="false" output="screen">
    <param name="license" type="string" value="$(env HOME)/.mujoco/mjkey.txt" />
    <param name="pub_mode" value="$(arg pub_mode)"/>
    <param name="model_file" type="string" value="$(find icir_gen3_description)/6dof/urdf/kinova_arm_6dof.xml"/>
  </node>

  <!-- controller -->
  <node name="icir_tutorial_pinocchio_simulation" pkg="icir_tutorial_pinocchio" args="-keyboard" type="icir_tutorial_pinocchio_simulation" output="screen">
  </node>

  <!-- for pinocchio -->
  <param name="urdf_path" type="string" value="$(find icir_gen3_description)"/>
  <param name="urdf_name" type="string" value="/6dof/urdf/GEN3-6DOF_VISION_URDF_ARM_V01.urdf"/>

  <!-- robot publish -->
  <param name="robot_description" textfile="$(find icir_gen3_description)/6dof/urdf/GEN3-6DOF_VISION_URDF_ARM_V01.urdf" />
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" respawn="false" output="screen">
        <param name="use_tf_static"     value="false"/>
        <param name="publish_frequency" value="100"/>
        <param name="ignore_timestamp" value="true"/>
        <remap from="joint_states"     to="mujoco_ros/mujoco_ros_interface/joint_states" />
  </node>

  <!-- rqt_gui -->
  <node name="rqt" pkg="rqt_gui" type="rqt_gui" args="--perspective-file $(find icir_tutorial_pinocchio)/rqt/icir_tutorial_pinocchio.perspective"
output="screen"/>

</launch>
```

Pinocchio

- An open-source software framework that implements rigid body dynamics algorithms and their analytical derivatives[1].
- Pinocchio required about 1us evaluating the dynamics on manipulator robots.



- install

  https://stack-of-tasks.github.io/pinocchio/download.html
  https://github.com/iCIRLab/icir_tutorial_pinocchio

- Kinematics algorithms
  - Forward kinematics
  - Kinematic jacobian
- Dynamic algorithms
  - Inverse dynamics
  - Joint-space inertia matrix
  - Forward dynamics

[1]Justin Carpentier, et al., "The Pinocchio C++ library – A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives," SII 2019, 2019.

Dealing with Lie-group geometry

- Pinocchio relies heavily on Lie groups and Lie algebras to handle motions and more specifically rotations.
- SO(3) : Special Orthogonal Group : the set of all rotation matrices is called SO(3)

$$SO(n) = \{R \in \mathbb{R}^{n \times n} : RR^T = I, \det(R) = 1\}$$

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

- SE(3) : Special Euclidean Group : homogeneous transformation matrix

$$T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_1 \\ r_{21} & r_{22} & r_{23} & p_2 \\ r_{31} & r_{32} & r_{33} & p_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - Such a matrix not only represents the configuration of a frame, but also be used to
    - (1) translate and rotate a vector or a frame
    - (2) change the representation of a vector or a frame from coordinates in one frame to coordinates in another frame
  - These operation can be performed by simple linear algebra → major reason

Spatial algebra

- Spatial algebra is a mathematical notation commonly employed in rigid body dynamics to represent and manipulate physical quantities such as velocities, accelerations and forces.

- Spatial velocity, Twist

〈Motion〉

$$\boldsymbol{\nu} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} \omega \\ v \end{bmatrix}$$

- Spatial force, Wrench

〈Force〉

$$\boldsymbol{F} = \begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \\ f_x \\ f_y \\ f_z \end{bmatrix} = \begin{bmatrix} \tau \\ f \end{bmatrix}$$

Dealing with Lie-group geometry*

Rotations	Rigid-Body Motions
$R \in SO(3) : 3 \times 3$ matrices $R^{\mathrm{T}}R = I, \det R = 1$	$T \in SE(3) : 4 \times 4$ matrices $T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix}$, where $R \in SO(3), p \in \mathbb{R}^3$
$R^{-1} = R^{\mathrm{T}}$	$T^{-1} = \begin{bmatrix} R^{\mathrm{T}} & -R^{\mathrm{T}}p \\ 0 & 1 \end{bmatrix}$
change of coordinate frame: $R_{ab}R_{bc} = R_{ac}, \quad R_{ab}p_b = p_a$	change of coordinate frame: $T_{ab}T_{bc} = T_{ac}, \quad T_{ab}p_b = p_a$
rotating a frame {b}: $R = \mathrm{Rot}(\hat{\omega}, \theta)$ $R_{sb'} = RR_{sb}$: rotate θ about $\hat{\omega}_s = \hat{\omega}$ $R_{sb''} = R_{sb}R$: rotate θ about $\hat{\omega}_b = \hat{\omega}$	displacing a frame {b}: $T = \begin{bmatrix} \mathrm{Rot}(\hat{\omega}, \theta) & p \\ 0 & 1 \end{bmatrix}$ $T_{sb'} = TT_{sb}$: rotate θ about $\hat{\omega}_s = \hat{\omega}$ (moves {b} origin), translate p in {s} $T_{sb''} = T_{sb}T$: translate p in {b}, rotate θ about $\hat{\omega}$ in new body frame
unit rotation axis is $\hat{\omega} \in \mathbb{R}^3$, where $\|\hat{\omega}\| = 1$	"unit" screw axis is $\mathcal{S} = \begin{bmatrix} \omega \\ v \end{bmatrix} \in \mathbb{R}^6$, where either (i) $\|\omega\| = 1$ or (ii) $\omega = 0$ and $\|v\| = 1$
	for a screw axis $\{q, \hat{s}, h\}$ with finite h, $\mathcal{S} = \begin{bmatrix} \omega \\ v \end{bmatrix} = \begin{bmatrix} \hat{s} \\ -\hat{s} \times q + h\hat{s} \end{bmatrix}$
angular velocity is $\omega = \hat{\omega}\dot{\theta}$	twist is $\mathcal{V} = \mathcal{S}\dot{\theta}$
for any 3-vector, e.g., $\omega \in \mathbb{R}^3$, $[\omega] = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \in so(3)$ identities, $\omega, x \in \mathbb{R}^3, R \in SO(3)$: $[\omega] = -[\omega]^{\mathrm{T}}, [\omega]x = -[x]\omega,$ $[\omega][x] = ([x][\omega])^{\mathrm{T}}, R[\omega]R^{\mathrm{T}} = [R\omega]$	for $\mathcal{V} = \begin{bmatrix} \omega \\ v \end{bmatrix} \in \mathbb{R}^6$, $[\mathcal{V}] = \begin{bmatrix} [\omega] & v \\ 0 & 0 \end{bmatrix} \in se(3)$ (the pair (ω, v) can be a twist $\mathcal{V}$ or a "unit" screw axis $\mathcal{S}$, depending on the context)

continued...

Rotations (cont.)	Rigid-Body Motions (cont.)
$\dot{R}R^{-1} = [\omega_s], \quad R^{-1}\dot{R} = [\omega_b]$	$\dot{T}T^{-1} = [\mathcal{V}_s], \quad T^{-1}\dot{T} = [\mathcal{V}_b]$
	$[\mathrm{Ad}_T] = \begin{bmatrix} R & 0 \\ [p]R & R \end{bmatrix} \in \mathbb{R}^{6\times6}$ identities: $[\mathrm{Ad}_T]^{-1} = [\mathrm{Ad}_{T^{-1}}]$, $[\mathrm{Ad}_{T_1}][\mathrm{Ad}_{T_2}] = [\mathrm{Ad}_{T_1T_2}]$
change of coordinate frame: $\hat{\omega}_a = R_{ab}\hat{\omega}_b, \quad \omega_a = R_{ab}\omega_b$	change of coordinate frame: $\mathcal{S}_a = [\mathrm{Ad}_{T_{ab}}]\mathcal{S}_b, \quad \mathcal{V}_a = [\mathrm{Ad}_{T_{ab}}]\mathcal{V}_b$
exp coords for $R \in SO(3)$: $\hat{\omega}\theta \in \mathbb{R}^3$	exp coords for $T \in SE(3)$: $\mathcal{S}\theta \in \mathbb{R}^6$
$\exp : [\hat{\omega}]\theta \in so(3) \to R \in SO(3)$ $R = \mathrm{Rot}(\hat{\omega}, \theta) = e^{[\hat{\omega}]\theta} =$ $I + \sin\theta[\hat{\omega}] + (1 - \cos\theta)[\hat{\omega}]^2$	$\exp : [\mathcal{S}]\theta \in se(3) \to T \in SE(3)$ $T = e^{[\mathcal{S}]\theta} = \begin{bmatrix} e^{[\omega]\theta} & * \\ 0 & 1 \end{bmatrix}$ where $* =$ $(I\theta + (1 - \cos\theta)[\omega] + (\theta - \sin\theta)[\omega]^2)v$
$\log : R \in SO(3) \to [\hat{\omega}]\theta \in so(3)$ algorithm in Section 3.2.3.3	$\log : T \in SE(3) \to [\mathcal{S}]\theta \in se(3)$ algorithm in Section 3.3.3.2
moment change of coord frame: $m_a = R_{ab}m_b$	wrench change of coord frame: $\mathcal{F}_a = (m_a, f_a) = [\mathrm{Ad}_{T_{ba}}]^{\mathrm{T}}\mathcal{F}_b$

*Kevin Lynch and Frank J. Park, "Modern Robotics,"

Model and data

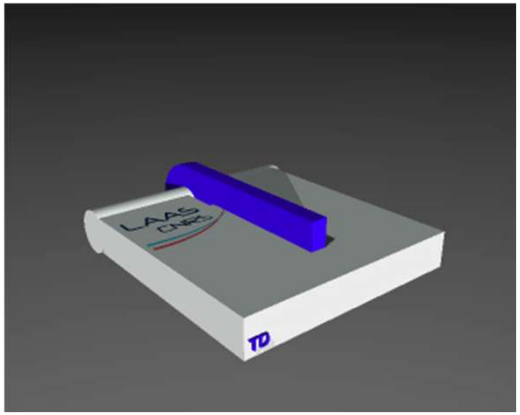
- A fundamental paradigm of Pinocchio is the strict separation between model and data.

- Model : fixed robot structrue
  - Model = Pinocchio.Model()
  - model.name
  - model.joints
  - model.placements
  - model.inertias
  - model.frames

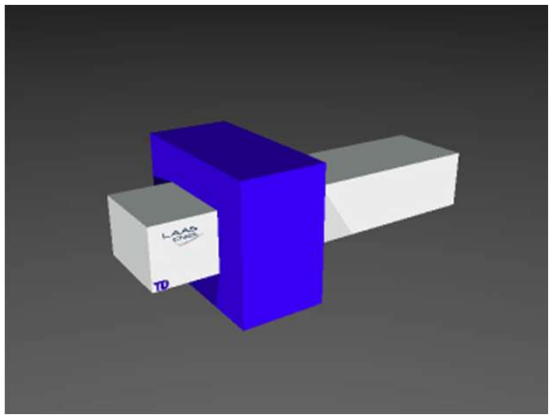
- Data: temporary values computed during algorithms
  - data = Pinocchio.Data(model)
    or data = model.createData()
  - data.joints
  - data.oMi
  - data.v
  - data.a
  - data.f
  - data.M
  - data.nle

Joints (Supported kinematic models) *

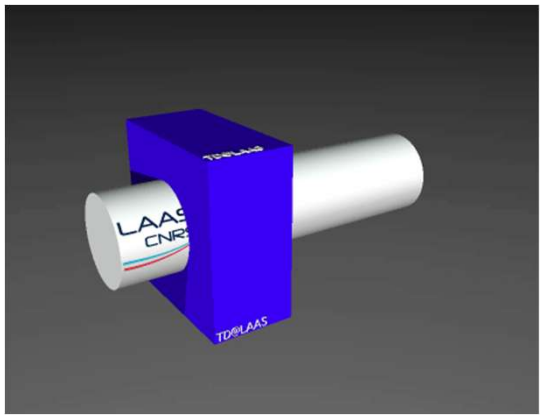
- Revolute joint



- Prismatic joint

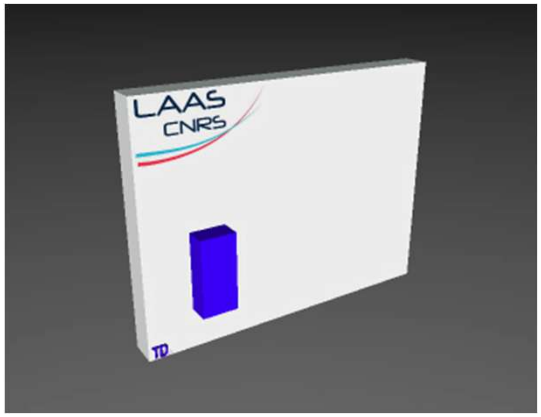


- Cylindrical joint



- Spherical joint

- Planar joint



*https://gepettoweb.laas.fr/doc/stack-of-tasks/pinocchio/devel/doxygen-html/md_doc_2c-maths_2b-joints.html#autotoc_md46

Loading a model

`pinocchio::urdf::buildModel(filename, m_model, m_verbose)`

: loads the robot's URDF and automatically builds its models on defined root joint type (output is m_model)

`model.nq` : gives the dimension of the configuration space

`model.nv` : gives the dimension of the tangent space (velocity space)

`na` : gives the number of actuated joints (not a member of Pinocchio)

Ex) 6dof arm

nq : 6

nv : 6

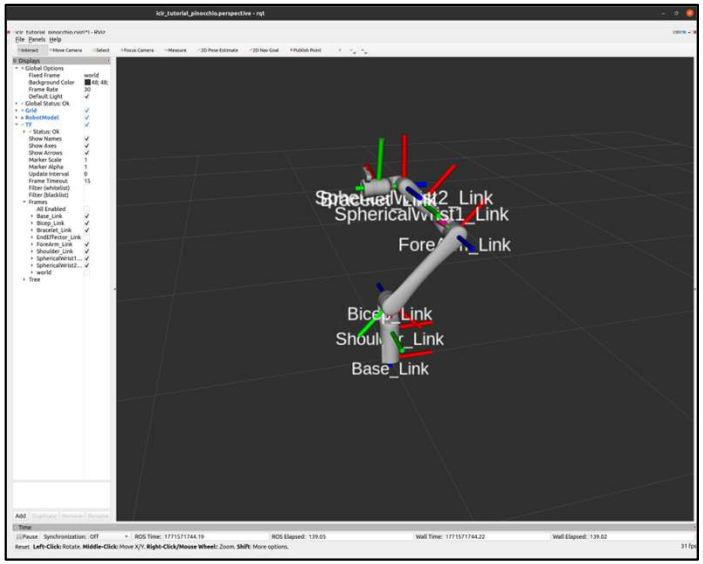
na : 6

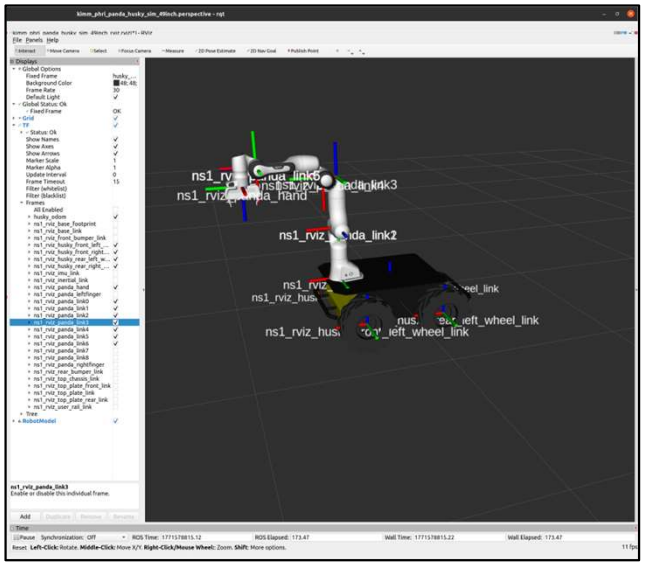
Ex) mobile manipulator with differential drive & 7dof arm

nq : 12 ← 3dof odometry (x,y,theta) + 2 wheel + 7 arm joints

nv : 12

na : 9 ← 2 wheel + 7 arm joints





Compute

`pinocchio::computeAllTerms(m_model, data, q, v)`

: Computes efficiently all the terms needed for dynamic simulation

`pinocchio::forwardKinematics(m_model, data, q, v, a)` → data.oMi, data.v, data.a

: Update the joint placements according to the current joint configuration.

`pinocchio::crba(m_model, data, q, v, convention)` → The joint space inertia matrix with only the upper triangular part computed.

: Computes the upper triangular part of the joint space inertia matrix M

`pinocchio::nonlinearEffects(m_model, data, q, v)` → data.nle

: Computes the non-linear effects (Corriolis, centrifual and gravitationnal effects)

`pinocchio::computeJointJacobians(m_model, data, q)` → data.J

: Computes the full model Jacobian in the world frame

`pinocchio::centerOfMass(m_model, data, q)` → data.com[0] & data.com[i]

: Computes the full model Jacobian in the world frame

`pinocchio::ccrba(m_model, data, q, v)` → data.Ag

: Computes the Centroidal Momentum Matrix

Compute

pinocchio::rnea(m_model, data, q, v, a) → data.tau

: Computes inverse dynamics torques using the Recursive Newton–Euler Algorithm. Used to find the required joint torques to track a desired motion. (inverse dynamics)

$$\tau = M(q)a + C(q,v)v + g(q)$$

pinocchio::aba(m_model, data, q, v, a, tau) → data.ddq

: Computes joint accelerations using the Articulated-Body Algorithm (ABA). Used for simulation, control, and real-time dynamic prediction. (forward dynamics)

$$\ddot{q} = M(q)^{-1}(\tau - c(q,\dot{q}) - g(q))$$

$$\tau = \tau_{control} + \tau_{external}$$

❖ In a MuJoCo–Pinocchio simulation setup, MuJoCo performs forward dynamics to compute the next joint accelerations ($\ddot{q}$), while Pinocchio provides the model-based dynamic quantities required to compute the control inputs.

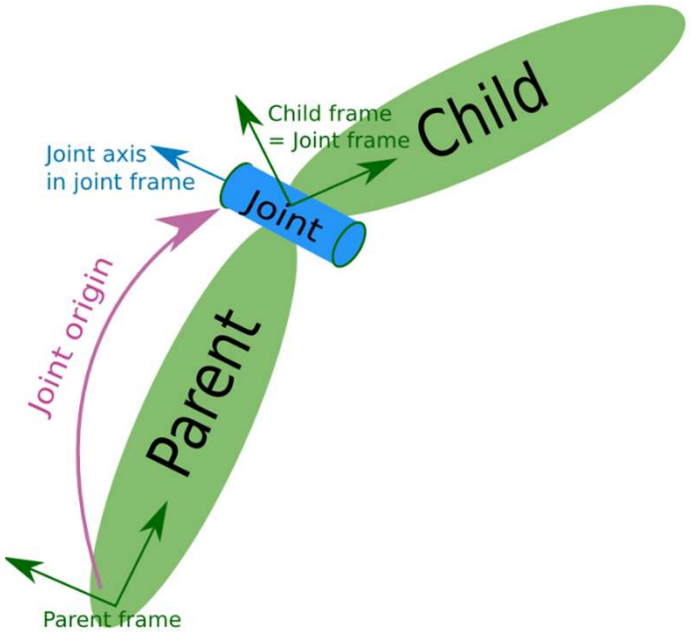
Joint & frame

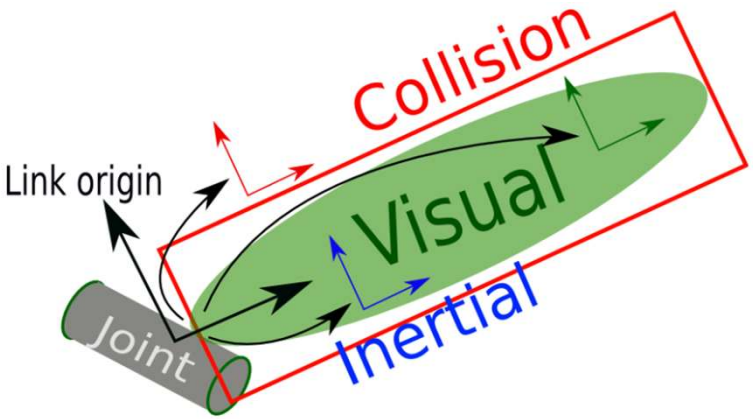
Enumerator	
OP_FRAME	operational frame: user-defined frames that are defined at runtime
JOINT	joint frame: attached to the child body of a joint (a.k.a. child frame)
FIXED_JOINT	fixed joint frame: joint frame but for a fixed joint
BODY	body frame: attached to the collision, inertial or visual properties of a link
SENSOR	sensor frame: defined in a sensor element

JointIndex : The index of a joint in the kinematic tree

FrameIndex : The index of a frame in the model

Joint_frame = child_frame





https://gepettoweb.laas.fr/doc/stack-of-tasks/pinocchio/devel/doxygen-html/namespacepinocchio.html#ad495a9f61af7fff07d7e97979d1ab854

function

data.oMi[Model::JointIndex]

: ⟨SE3⟩ vector of absolute joint placement w.r.t world (joint_frame)

data.v[Model::JointIndex]

: ⟨Motion⟩ vector of spatial joint velocities expressed at the center of the joints (origin of joint_frame)

⟨SE3⟩ $M = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix}$

⟨Motion⟩ $v = \begin{bmatrix} \omega \\ v \end{bmatrix}$ ⟨Force⟩ $\mathbf{f} = \begin{bmatrix} \tau \\ f \end{bmatrix}$

```
SE3 RobotWrapper::framePosition(const Data & data, const Model::FrameIndex index) const
{
  assert(index<m_model.frames.size());
  const Frame & f = m_model.frames[index];
  return data.oMi[f.parent].act(f.placement);
}
```

```
struct Frame {
  std::string name;
  JointIndex parent;
  SE3 placement;
  FrameType type;
};
```

: f.parent is the "JointIndex" of the joint to which the frame is attached

ex) if f=i_{th} child_frame → f.parent is i

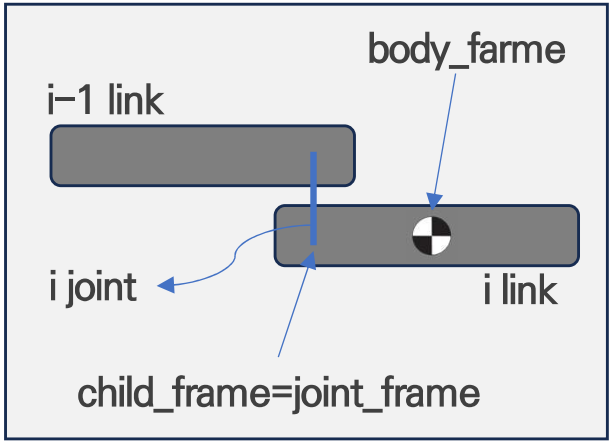
: f.placement is the transformation from the joint frame to this body frame

** Rigid body offset*

: data.oMi[f.parent].act(f.placement) : ${}^{world}T_{frame} = {}^{world}T_{joint} \cdot {}^{joint}T_{frame}$

data.oMi[f.parent] : ${}^{world}T_{joint}$

f.placement : ${}^{joint}T_{frame}$



function

SE3::act(SE3 m2) ← act_impl in se3-tpl.hpp

```
return SE3Tpl(rot * m2.rotation(), translation() + rotation() * m2.translation());
```

$$T = M_1.act(M_2) = M_1 M_2 = \begin{bmatrix} R_{M1}R_{M2} & p_{M1} + R_{M1}p_{M2} \\ 0 & 1 \end{bmatrix}$$

SE3::act(Motion v)

$$v_{out} = \begin{bmatrix} v_{out.ang} \\ v_{out.lin} \end{bmatrix} = M.act(v) = Ad_M v = \begin{bmatrix} R & 0 \\ p \times R & R \end{bmatrix} \begin{bmatrix} v_{ang} \\ v_{lin} \end{bmatrix} = \begin{bmatrix} Rv_{ang} \\ p \times (Rv_{ang}) + Rv_{lin} \end{bmatrix}$$

SE3::act(Force f)

$$f_{out} = \begin{bmatrix} f_{out.ang} \\ f_{out.lin} \end{bmatrix} = M.act(f) = Ad_M^* f = \begin{bmatrix} R & p \times R \\ 0 & R \end{bmatrix} \begin{bmatrix} f_{ang} \\ f_{lin} \end{bmatrix} = \begin{bmatrix} Rf_{ang} + p \times (Rf_{lin}) \\ Rf_{lin} \end{bmatrix}$$

https://github.com/stack-of-tasks/pinocchio/blob/devel/include/pinocchio/spatial/se3-tpl.hpp

function

SE3::actInv(SE3 m2) ← actInv_impl in se3-tpl.hpp

```
return SE3Tpl(rot.transpose() * m2.rotation(), rot.transpose() * (m2.translation() - translation()));
```

$$T = M_1.actInv(M_2) = M_1^{-1}M_2 = \begin{bmatrix} R_{M1}^T R_{M2} & R_{M1}^T(p_{M2} - p_{M1}) \\ 0 & 1 \end{bmatrix}$$

SE3::actInv(Motion v)

$$v_{out} = M.actInv(v) = Ad_M^{-1}v = \begin{bmatrix} R^T & 0 \\ -R^T[p]_\times & R^T \end{bmatrix} \begin{bmatrix} v_{ang} \\ v_{lin} \end{bmatrix} = \begin{bmatrix} R^T v_{ang} \\ -R^T[p]_\times v_{ang} + R^T v_{lin} \end{bmatrix}$$

SE3::actInv(Force f)

$$f_{out} = M.actInv(f) = (Ad_M^{-1})^* f = \begin{bmatrix} R^T & -R^T[p]_\times \\ 0 & R^T \end{bmatrix} \begin{bmatrix} f_{ang} \\ f_{lin} \end{bmatrix} = \begin{bmatrix} R^T f_{ang} - R^T[p]_\times f_{lin} \\ R^T f_{lin} \end{bmatrix}$$

https://github.com/stack-of-tasks/pinocchio/blob/devel/include/pinocchio/spatial/se3-tpl.hpp

function

```
Motion RobotWrapper::frameVelocity(const Data & data, const Model::FrameIndex index) const
{
  assert(index<m_model.frames.size());
  const Frame & f = m_model.frames[index];
  return f.placement.actInv(data.v[f.parent]);
}
```

: f.placement is the transformation from the joint frame to this body frame

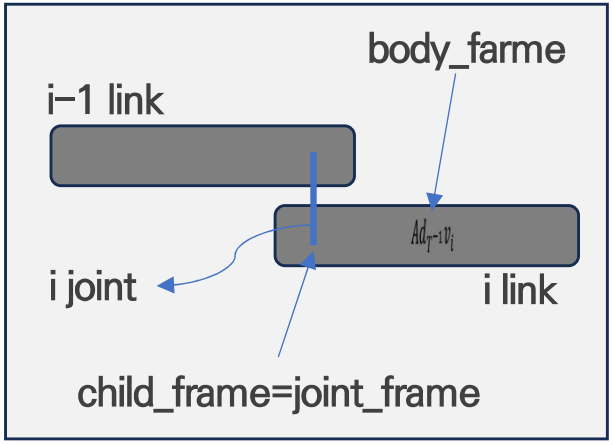
: data.v[f.parent] is the velocity vector of ith joint (f.parent) w.r.t ith joint origin

: f.placement.actInv(data.v[f.parent]) $Ad_{T^{-1}}v_i$

Ad_T : ith joint frame → ith body frame

v_i velocity vector of the ith joint w.r.t ith joint_frame

$Ad_{T^{-1}}v_i$ velocity vector of the ith joint w.r.t. ith body_frame



function

```
Motion RobotWrapper::frameClassicAcceleration(const Data & data, const Model::FrameIndex index) const
{
  assert(index<m_model.frames.size());
  const Frame & f = m_model.frames[index];
  Motion a = f.placement.actInv(data.a[f.parent]);
  Motion v = f.placement.actInv(data.v[f.parent]);
  a.linear() += v.angular().cross(v.linear());
  return a;
}
```

$$a_{classic} = \begin{bmatrix} \alpha_f \\ a_f^{lin} + \omega_f \times v_f^{lin} \end{bmatrix}$$

$$\ddot{x} = J\ddot{q} + \dot{J}\dot{q}$$

$$\ddot{q} = J^{-1}(\ddot{x} - \dot{J}\dot{q})$$

$$\dot{J}\dot{q} = a_{classic}$$

function

```
void RobotWrapper::jacobianWorld(const Data & data, const Model::JointIndex index, Data::Matrix6x & J)
{
    return pinocchio::getJointJacobian(m_model, data, index, pinocchio::WORLD, J) ;
}
```

: compute J of the joint at index, expressed in the world frame, evaluated at the joint's child link frame origin

```
void RobotWrapper::frameJacobianLocal(Data & data, const Model::FrameIndex index, Data::Matrix6x & J)
{
    assert(index<m_model.frames.size());
    J.resize(6, m_model.nv);
    return pinocchio::getFrameJacobian(m_model, data, index, pinocchio::LOCAL, J) ;
}
```

: compute J of the Frame at index, expressed in the LOCAL frame

Joint space position control

- Control each joint to track q_target using a PD controller

```
if (ctrl_mode_ == 1){ // joint task //h home
        if (chg_flag_){
            cubic_.stime = time_;
            cubic_.ftime = time_+ 2.0;
            cubic_.q0 = state_.q;
            cubic_.v0 = state_.v;
            for (int i = 0; i<GEN3_DOF; i++)
            {
                q_target_(i) = Home(i) * M_PI / 180.;
            }
            chg_flag_ = false;
        }
        for (int i=0; i<GEN3_DOF; i++)
        {
            state_.q_des(i) = cubic(time_, cubic_.stime, cubic_.ftime, cubic_.q0(i), q_target_(i), 0.0, 0.0);
            state_.v_des(i) = (state_.q_des(i) - state_.q_des_pre(i)) * SAMPLING_RATE;
            state_.q_des_pre(i) = state_.q_des(i);
            state_.ddq_des(i) = -posture_Kp(i)*(state_.q(i) - state_.q_des(i)) -posture_Kd(i)*(state_.v(i) - state_.v_des(i));
        }
    }
```

```
state_.tau_des = data_.M * state_.ddq_des + data_.nle;
robot_command(); //send torque command state_.tau_des to mujoco
```

Joint space position control

- Control each joint to track q_target using a PD controller

```
if (chg_flag_){                                   ← keyboard input flag
    cubic_.stime = time_;
    cubic_.ftime = time_+ 2.0;                    ← cubic trajectory setting
    cubic_.q0 = state_.q;
    cubic_.v0 = state_.v;
    for (int i = 0; i<GEN3_DOF; i++)
    {
        q_target_(i) = Home(i) * M_PI / 180.;     ← Target setting for each joint
    }
    chg_flag_ = false;                            ← clear keyboard flag
}
```

Home << 0.00, 45.0, 90.0, 0.0, 45.0, -90.0;

Cubic Trajectory:

$$q(t) = q_0 + \hat{q}_0 t + \left(\frac{3(q_f - q_0)}{T^2} - \frac{2q_0 + \hat{q}_f}{T}\right)t^2 - \left(\frac{2(q_f - q_0)}{T^3} + \frac{\hat{q}_0 + \hat{q}_f}{T^2}\right)t^3$$

Joint space position control

- Control each joint to track q_target using a PD controller

```
        for (int i=0; i<GEN3_DOF; i++)
        {
            state_.q_des(i) = cubic(time_, cubic_.stime, cubic_.ftime, cubic_.q0(i), q_target_(i), 0.0, 0.0);
            state_.v_des(i) = (state_.q_des(i) - state_.q_des_pre(i)) * SAMPLING_RATE;
            state_.q_des_pre(i) = state_.q_des(i);
            state_.ddq_des(i) = -posture_Kp(i)*(state_.q(i) - state_.q_des(i)) -posture_Kd(i)*(state_.v(i) - state_.v_des(i));
        }
    }
```

```
double elapsed_time = time - time_0;
double total_time = time_f - time_0;
double total_time2 = total_time * total_time;  // pow(t,2)
double total_time3 = total_time2 * total_time; // pow(t,3)
double total_x = x_f - x_0;

x_t = x_0 + x_dot_0 * elapsed_time
      + (3 * total_x / total_time2 - 2 * x_dot_0 / total_time - x_dot_f / total_time) * elapsed_time * elapsed_time
      + (-2 * total_x / total_time3 + (x_dot_0 + x_dot_f) / total_time2) *elapsed_time * elapsed_time * elapsed_time;
```

Cubic Trajectory:

$$q(t) = q_0 + \hat{q}_0 t + \left(\frac{3(q_f - q_0)}{T^2} - \frac{2q_0 + \hat{q}_f}{T}\right)t^2 - \left(\frac{2(q_f - q_0)}{T^3} + \frac{\hat{q}_0 + \hat{q}_f}{T^2}\right)t^3$$

Joint space position control

- Control each joint to track q_target using a PD controller

```
for (int i=0; i<GEN3_DOF; i++)
{
    state_.q_des(i) = cubic(time_, cubic_.stime, cubic_.ftime, cubic_.q0(i), q_target_(i), 0.0, 0.0);
    state_.v_des(i) = (state_.q_des(i) - state_.q_des_pre(i)) * SAMPLING_RATE;
    state_.q_des_pre(i) = state_.q_des(i);
    state_.ddq_des(i) = -posture_Kp(i)*(state_.q(i) - state_.q_des(i)) -posture_Kd(i)*(state_.v(i) - state_.v_des(i));
}
```

```
state_.tau_des = data_.M * state_.ddq_des + data_.nle;
robot_command(); //send torque command state_.tau_des to mujoco
```

- robot dynamics

$$\tau = M\ddot{\theta} + c(\theta, \dot{\theta}) + g(\theta)$$

- computed torque control framework

$$\tau_{cmd} = \tilde{M}\ddot{\theta}_{des} + \tilde{c}(\theta, \dot{\theta}) + \tilde{g}(\theta)$$

- PD controller for each joint

$$\ddot{\theta}_{des} = \ddot{\theta}_r - K_P(\theta - \theta_r) - K_D(\dot{\theta} - \dot{\theta}_r)$$

$$\Rightarrow \tau_{cmd} = \tilde{M}\left(\ddot{\theta}_r - K_P(\theta - \theta_r) - K_D(\dot{\theta} - \dot{\theta}_r)\right) + \tilde{c}(\theta, \dot{\theta}) + \tilde{g}(\theta)$$

Task space position control

Control task space to track q_target using a PD controller

```
if (ctrl_mode_ == 3){ // ee task //k ee jog -0.05z
        if (chg_flag_){
            SE3Cubic_.stime = time_;
            SE3Cubic_.duration = 2.0;

            //set init ee pose
            H_ee_ = robot_->position(data_, m_joint_id);
            SE3Cubic_.m_init = H_ee_;

            //set desired ee pose
            H_ee_ref_ = H_ee_;
            H_ee_ref_.translation()(2) -= 0.05;

            chg_flag_ = false;
        }
        sampleEE_ = SE3Cubic(time_, SE3Cubic_.stime, SE3Cubic_.duration, SE3Cubic_.m_init, H_ee_ref_);
        vectorToSE3(sampleEE_, m_M_ref);  //desired trajectory in SE3 [pinocchio::SE3]

        SE3 oMi;
        Motion v_frame;
        robot_->framePosition(data_, m_frame_id, oMi);              //frame position in global frame
        robot_->frameVelocity(data_, m_frame_id, v_frame);          //frame velocity in local frame
        robot_->frameClassicAcceleration(data_, m_frame_id, m_drift); //m_drift in local frame which is identical to J_dot * q_dot
        robot_->frameJacobianLocal(data_, m_frame_id, m_J_local_);   //frame jacobian in local frame

        SE3ToVector(oMi, m_p_);                             // current pos in vector form
        errorInSE3(oMi, m_M_ref, m_p_error);                    // pos erorr represented in local frame, oMi_inv*m_M_ref

        // Transformation from local to world
        m_wMl.translation(oMi.translation());                       // use rotation only for vel&acc transformation
        m_wMl.rotation(oMi.rotation());                     // use rotation only for vel&acc transformation

        m_p_error_vec = m_p_error.toVector();                   // pos err vector in local frame
        m_v_error =  m_wMl.actInv(m_v_ref) - v_frame;               // vel err vector in local frame

        // desired acc in local frame
        m_a_des =   ee_Kp.cwiseProduct(m_p_error_vec)  + ee_Kd.cwiseProduct(m_v_error.toVector()) + m_wMl.actInv(m_a_ref).toVector();

        //transformation from ee to joint
        state_.ddq_des = m_J_local_.completeOrthogonalDecomposition().pseudoInverse() * (m_a_des - m_drift.toVector());
}
```

Task space position control

- Control task space to track q_target using a PD controller

```
if (ctrl_mode_ == 3){ // ee task //k ee jog -0.05z
        if (chg_flag_){
            SE3Cubic_.stime = time_;
            SE3Cubic_.duration = 2.0;

            //set init ee pose
            H_ee_ = robot_->position(data_, m_joint_id);
            SE3Cubic_.m_init = H_ee_;

            //set desired ee pose
            H_ee_ref_ = H_ee_;
            H_ee_ref_.translation()(2) -= 0.05;

            chg_flag_ = false;
        }
```

keyboard input flag

SE3Cubic trajectory setting

Target ee pose setting

clear keyboard flag

$$\begin{bmatrix} & & & p(x) \\ & R & & p(y) \\ & & & p(z) - 0.05 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Task space position control

- Control task space to track q_target using a PD controller

```
sampleEE_ = SE3Cubic(time_, SE3Cubic_.stime, SE3Cubic_.duration, SE3Cubic_.m_init, H_ee_ref_);
vectorToSE3(sampleEE_, m_M_ref);  //desired trajectory in SE3 [pinocchio::SE3]
```

```
Eigen::Matrix3d rot_diff = (m_init.rotation().transpose() * m_goal.rotation()).log();

double a0, a1, a2, a3;
double tau = cubic(m_time, m_stime, m_stime + m_duration, 0, 1, 0, 0);

for (int i = 0; i < 3; i++) {
  a0 = m_init.translation()(i);
  a1 = 0.0; //m_init.vel(i);
  a2 = 3.0 / pow(m_duration, 2) * (m_goal.translation()(i) - m_init.translation()(i));
  a3 = -1.0 * 2.0 / pow(m_duration, 3) * (m_goal.translation()(i) - m_init.translation()(i));

  cubic_tra(i) = a0 + a1 * (m_time - m_stime) + a2 * pow(m_time - m_stime, 2) + a3 * pow(m_time - m_stime, 3);
}

m_cubic.rotation() = m_init.rotation() * (rot_diff * tau).exp();
m_cubic.translation() = cubic_tra;

pos_sample.head<3>() = m_cubic.translation();
pos_sample.tail<9>() = Eigen::Map<const Vector9>(&m_cubic.rotation()(0), 9);
```

Task space position control

- Control task space to track q_target using a PD controller
  - Initial & goal pose

$$T_{init} = \begin{bmatrix} R_{init} & p_{init} \\ 0 & 1 \end{bmatrix} \qquad T_{gaol} = \begin{bmatrix} R_{goal} & p_{goal} \\ 0 & 1 \end{bmatrix}$$

  - Rotational difference (Lie algebra)

$$R_{diff} = R_{init}^{T} R_{goal} \qquad \log(R_{diff}) = [\omega]\theta$$

$$\omega \in \mathbb{R}^3, unit\ rotation\ axis$$

$$\theta \in \mathbb{R}, total\ rotation\ angle$$

$$[\omega] = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \in so(3)(skew\ matrix)$$

  - Timing Scaling (cubic)

$$\tau(t) \in [0,1]$$

  - Rotation Interpolation

$$R(\mathrm{t}) = \mathrm{R}_{init} \cdot \mathrm{e}^{([\omega]\theta\tau(t))}$$

$$\Rightarrow \quad T(t) = \begin{bmatrix} R_{init} \cdot \mathrm{e}^{([\omega]\theta\tau(t))} & p(t) \\ 0 & 1 \end{bmatrix}$$

  - Position cubic Interpolation

$$p(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

*Position → Vector space $\mathbb{R}^3$ cubic interpolation Timing Scaling (cubic)
*Rotation → SO(3) manifold exponential map interpolation

Task space position control

- Control task space to track q_target using a PD controller

```
SE3 oMi;
Motion v_frame;
robot_->framePosition(data_, m_frame_id, oMi);              //frame position in global frame
robot_->frameVelocity(data_, m_frame_id, v_frame);          //frame velocity in local frame
robot_->frameClassicAcceleration(data_, m_frame_id, m_drift); //m_drift in local frame which is identical to J_dot * q_dot
robot_->frameJacobianLocal(data_, m_frame_id, m_J_local_);   //frame jacobian in local frame

SE3ToVector(oMi, m_p_);                                  // current pos in vector form
errorInSE3(oMi, m_M_ref, m_p_error);                     // pos erorr represented in local frame, oMi_inv*m_M_ref

// Transformation from local to world
m_wMl.translation(oMi.translation());                    // use rotation only for vel&acc transformation
m_wMl.rotation(oMi.rotation());                          // use rotation only for vel&acc transformation
```

- m_frame is the ith joint_frame
- Frame position in global coordinate (global → ith jointIndex) oMi
- Frame velocity in local coordinate (ith joint_frame velocity w.r.t ith joint_frame) V_frame
- Frame acceleration in local coordinate (ith joint_frame acceleration w.r.t ith joint_frame) m_drift
- Frame jacobian in local coordinate (ith joint_frame jacobian w.r.t ith joint_frame) J_local

Task space position control

- Control task space to track q_target using a PD controller

```
SE3 oMi;
Motion v_frame;
robot_->framePosition(data_, m_frame_id, oMi);              //frame position in global frame
robot_->frameVelocity(data_, m_frame_id, v_frame);          //frame velocity in local frame
robot_->frameClassicAcceleration(data_, m_frame_id, m_drift); //m_drift in local frame which is identical to J_dot * q_dot
robot_->frameJacobianLocal(data_, m_frame_id, m_J_local_);   //frame jacobian in local frame

SE3ToVector(oMi, m_p_);                                  // current pos in vector form
errorInSE3(oMi, m_M_ref, m_p_error);                     // pos erorr represented in local frame, oMi_inv*m_M_ref

// Transformation from local to world
m_wMl.translation(oMi.translation());                    // use rotation only for vel&acc transformation
m_wMl.rotation(oMi.rotation());                          // use rotation only for vel&acc transformation
```

```
void errorInSE3 (const pinocchio::SE3 & M,
                 const pinocchio::SE3 & Mdes,
                 pinocchio::Motion & error)
 {
   // error = pinocchio::log6(Mdes.inverse() * M);
   // pinocchio::SE3 M_err = Mdes.inverse() * M;
   pinocchio::SE3 M_err = M.inverse() * Mdes;
   error.linear() = M_err.translation();
   error.angular() = pinocchio::log3(M_err.rotation());
 }
```

$T_{err} = T_{current}^{T} T_{des}$

$p_{err} = T_{err}.p()$

$R_{err} = \log(T_{err}.R)$

Task space position control

- Control task space to track q_target using a PD controller

```
m_p_error_vec = m_p_error.toVector();                    // pos err vector in local frame
m_v_error =  m_wMl.actInv(m_v_ref) - v_frame;            // vel err vector in local frame

// desired acc in local frame
m_a_des =   ee_Kp.cwiseProduct(m_p_error_vec)  + ee_Kd.cwiseProduct(m_v_error.toVector()) + m_wMl.actInv(m_a_ref).toVector();

//transformation from ee to joint
state_.ddq_des = m_J_local_.completeOrthogonalDecomposition().pseudoInverse() * (m_a_des - m_drift.toVector());
```

```
state_.tau_des = data_.M * state_.ddq_des + data_.nle;
robot_command(); //send torque command state_.tau_des to mujoco
```

- reference velocity and acceleration in local frame `m_wMl.actInv(m_v_ref)` `m_wMl.actInv(m_a_ref)`
- task space controller in local frame

$$a_{des} = a_r - K_P(x - x_r) - K_D(\dot{x} - \dot{x}_r)$$

- Transformation from task space to joint space

$$\ddot{\theta}_{des} = J^{+}(a_{des} - \dot{J}\dot{\theta})$$

- computed torque control framework

$$\tau_{cmd} = \tilde{M}\ddot{\theta}_{des} + \tilde{c}(\theta, \dot{\theta}) + \tilde{g}(\theta)$$

Task space impedance control

Impedance control in task space for human following control with K=0

```
if (ctrl_mode_ == 10){ // impedance control //v
        if (chg_flag_){
            SE3Cubic_.stime = time_;
            SE3Cubic_.duration = 2.0;

            //set init ee pose
            H_ee_ = robot_->position(data_, m_joint_id);
            SE3Cubic_.m_init = H_ee_;

            //set desired ee pose
            H_ee_ref_ = H_ee_;
            chg_flag_ = false;
        }

        //to make K(x-xd)=0, put xd=x
        H_ee_ref_.translation() = robot_->position(data_, m_joint_id).translation();
        m_M_ref = H_ee_ref_;

        // SE3ToVector(H_ee_ref_, sampleEE_);
        // vectorToSE3(sampleEE_, m_M_ref);  //desired trajectory in SE3 [pinocchio::SE3]

        SE3 oMi;
        Motion v_frame;
        robot_->framePosition(data_, m_frame_id, oMi);              //frame position in global frame
        robot_->frameVelocity(data_, m_frame_id, v_frame);          //frame velocity in local frame
        robot_->frameClassicAcceleration(data_, m_frame_id, m_drift); //m_drift in local frame which is identical to J_dot * q_dot
        robot_->frameJacobianLocal(data_, m_frame_id, m_J_local_);   //frame jacobian in local frame

        SE3ToVector(oMi, m_p_);                             // current pos in vector form
        errorInSE3(oMi, m_M_ref, m_p_error);                    // pos erorr represented in local frame, oMi_inv*m_M_ref

        // Transformation from local to world
        m_wMl.rotation(oMi.rotation());                     // use rotation only for vel&acc transformation

        m_p_error_vec = m_p_error.toVector();                   // pos err vector in local frame
        m_v_error =  m_wMl.actInv(m_v_ref) - v_frame;               // vel err vector in local frame

        // desired acc in local frame
        m_a_des =   ee_Kp.cwiseProduct(m_p_error_vec)  + ee_Kd.cwiseProduct(m_v_error.toVector())  + m_wMl.actInv(m_a_ref).toVector();

        //transformation from ee to joint
        state_.ddq_des = m_J_local_.completeOrthogonalDecomposition().pseudoInverse() * (m_a_des - m_drift.toVector());
}
```