European Commission

Horizon 2020
European Union funding
for Research & Innovation

memmo

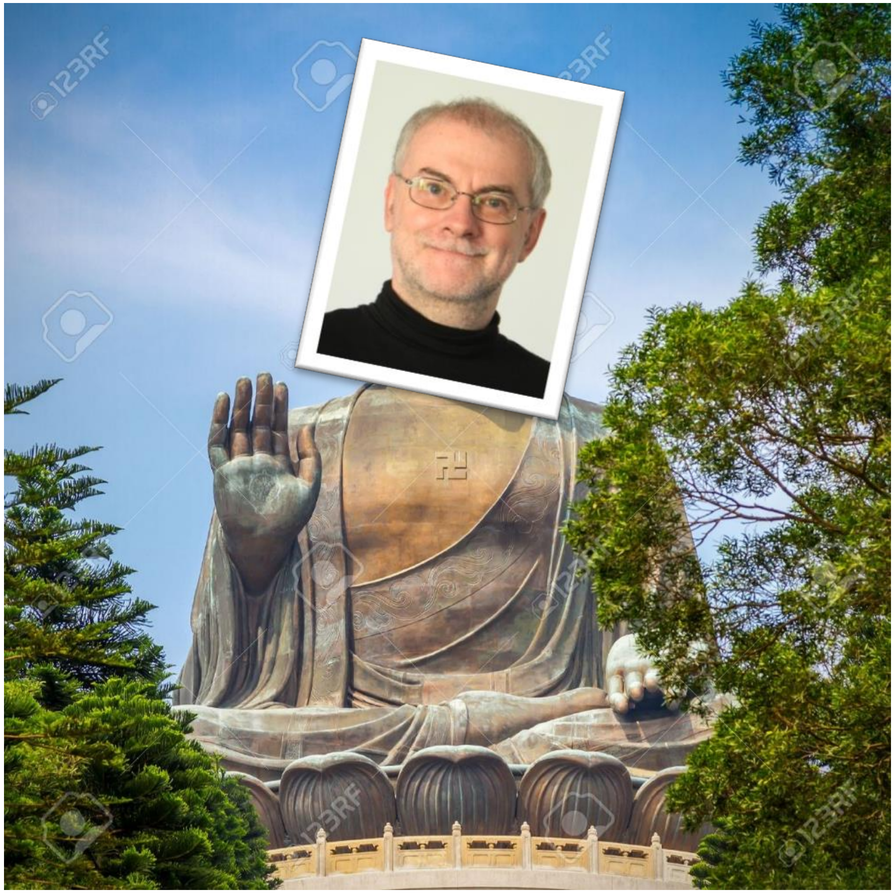
Pinocchio

Fast forward & inverse dynamics

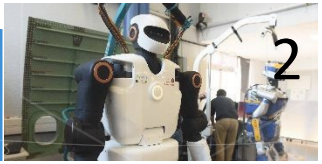
Nicolas Mansard
(CNRS)

Gurus

Justin Carpentier (INRIA)



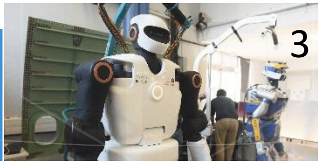
Roy Featherstone (IIT)



WWW Material

- Web site
  - https://stack-of-tasks.github.io/pinocchio
- Doxygen
  - Documentation tab on github.io
- Tutorials:
  - Practical exercices in the documentation

- Also use the ? In Python

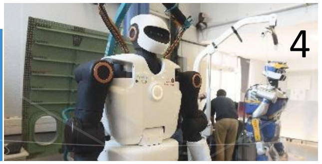


Contributing to Pinocchio

- GitHub project
  - https://github.com/stack-of-tasks/pinocchio

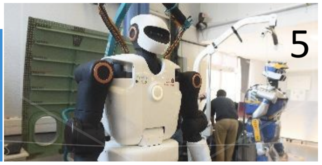
- Post issues for contributing

- We are looking for doc-devs!
  - Feedback some material as a thank-you note
  - In the doc: “examples” is waiting for you



C++ / Python

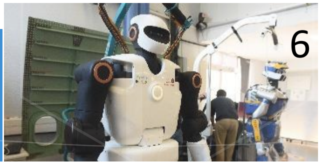
- C++ Library
  - Fast, careful implementation
  - Using curiously recursive template pattern (CRTP)
  - You likely don’t want to develop code there
  - Using it is not so complex (think Eigen)
- Python bindings
  - A 1-to-0.99 map from C++ API to Python API
  - Start by developing in Python
  - Beware of the lack of accuracy … speed is ok



Modeling and optimizing

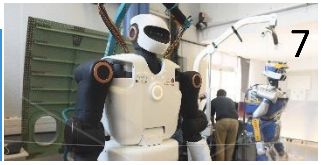
- Pinocchio is a modeling library
  - Not an application
  - Not a solver
  - Some key features directly available

- You don't want the solver inside Pinocchio
  - Inverse dynamics: TSID
  - Planning and contact planning: HPP
  - Optimal control: Crocodyl
  - Optimal estimation, reinforcement learning, inverse kinematics, contact simulation …



List of features

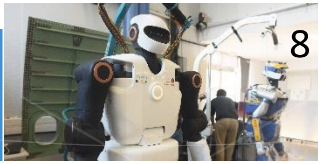
- URDF parser
- Forward kinematics and Jacobians
- Mass, center of mass and gen.inertia matrix
- Forward and inverse dynamics
- Model display (with Gepetto-viewer)
- Collision detection and distances (with HPP-FCL)
- Derivatives of kinematics and dynamics
- Type templatization and code generation



TSID

- Pinocchio for
  - Computing the inertia matrix, jacobians, kinematics

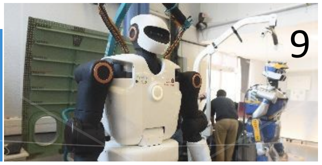
- Formulation of tasks
- Contact models
- QP resolution



HPP planner

- Pinocchio for
  - Geometry, collision (hpp-fcl)
  - Projectors with inverse kinematics
  - Balance constraint with dynamics

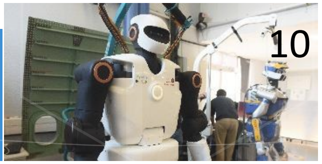
- Pinocchio encapsulated in hpp-Pinocchio
- Stochastic exploration algorithm (RRT)
- Contact checking
- Re-arrangement algorithms



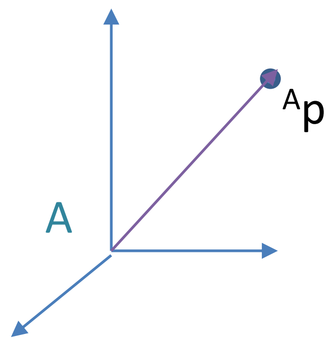
Crocoddyl

- Pinocchio for
  - Kinematics and dynamics
  - And their derivatives
  - Display with Gepetto-viewer

- DDP optimizer
- Task/cost formulation

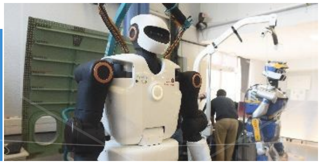


Representing the physical world



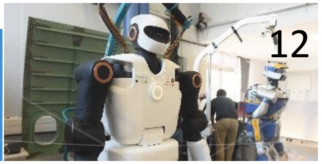
This is a point

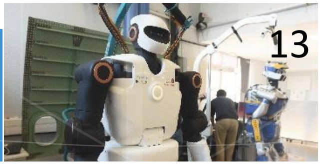
This is not a point
This is the representation of a point



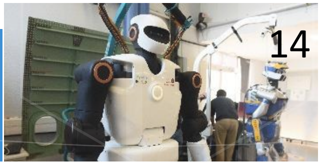
Representing the physical world

- Pinocchio is a model
  - Of course, models are wrong
- The way you represent geometry matters
- Example of SO(3)
  - r is a map from E(3) to E(3)
  - R is a othonormal positive matrix
  - w is a 3D vector
  - q is a quaternion represented as a 4D vector
  - Roll-Pitch-Yaw & other Euler angles should not be used



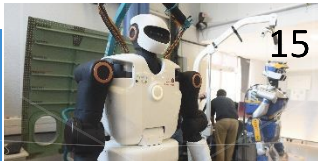


Pinocchio bases



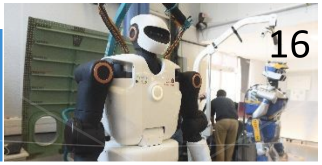
Basics

- Urdf model
- Kinematic tree
- Forward kinematics
- Display
- Spatial algebra



Kinematic tree

- Inside robot model:
  - joints: joint types and indices
  - names: joint names
  - jointPlacements: constant placement wrt parent
  - parents: hierarchy of joints representing the tree
- No bodies
  - masses and geoms are attached as tree decorations
- First joint represent the universe
  - If nq==7 then len(rmodel.joints)==8



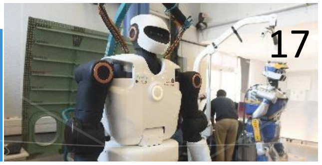
Kinematic tree

jupyter notebook

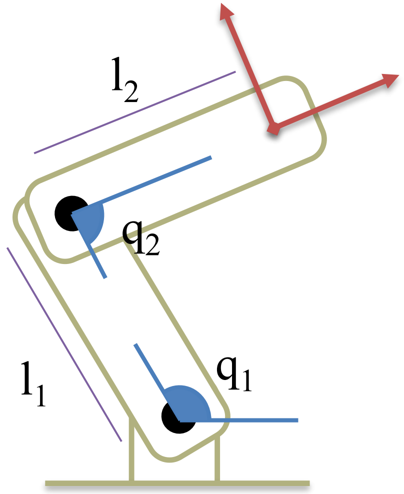
↳ go in ws memmo_pinocchio

open 1. ...

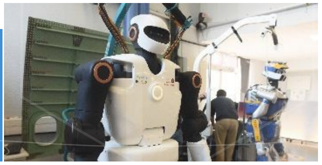
gepetto-gui



Direct geometry

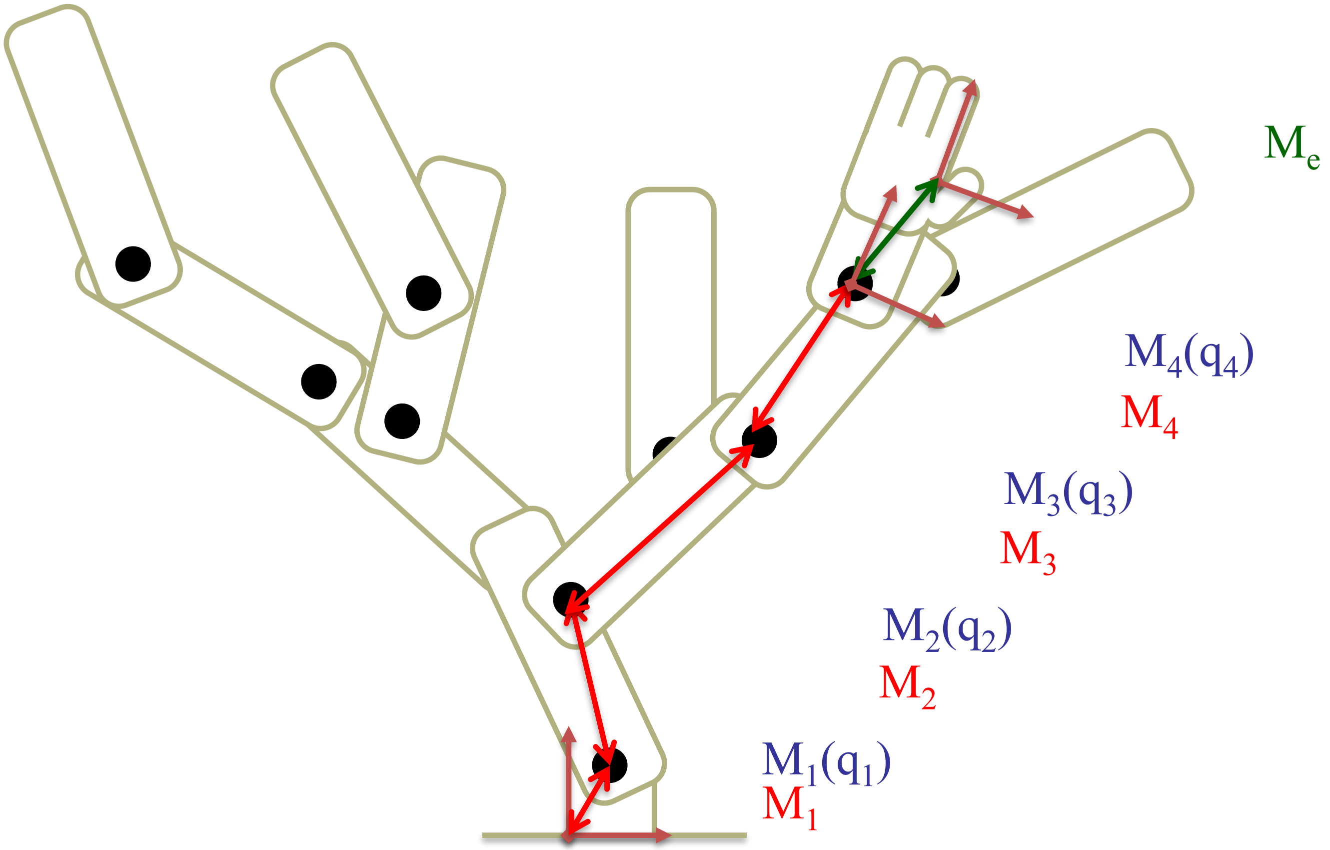


$$M(q) = \begin{bmatrix} l_1 \cos(q_1) + l_2 \cos(q_1+q_2) \\ l_1 \sin(q_1) + l_2 \sin(q_1+q_2) \end{bmatrix}$$



Direct geometry

- The geometric model is a tree of joints and bodies



$$M(q) = M_1 \oplus M_1(q_1) \oplus M_2 \oplus \ \dots \oplus M_4 \oplus M_4(q_4) \ \oplus M_e$$

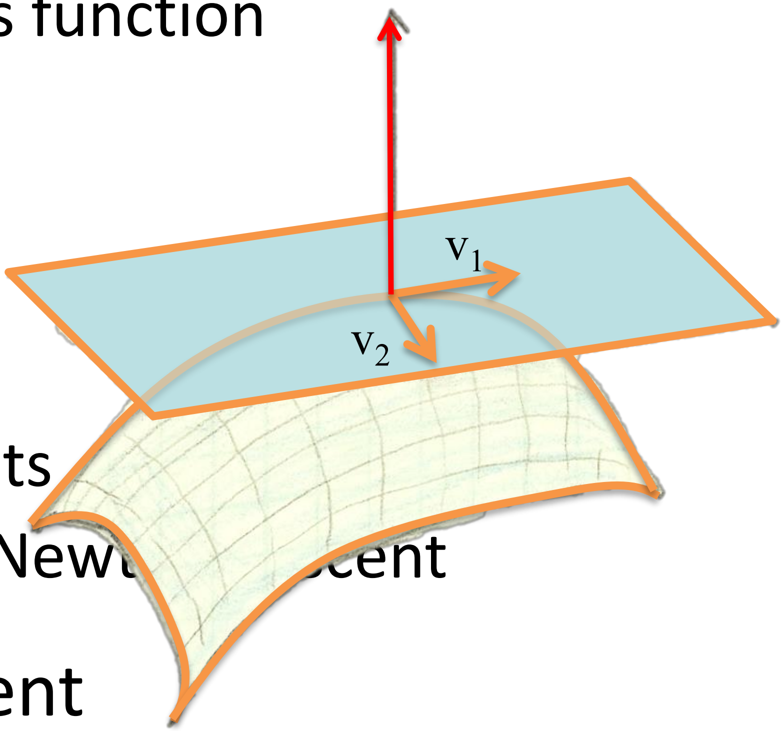
Direct and inverse functions

- Direct geometry

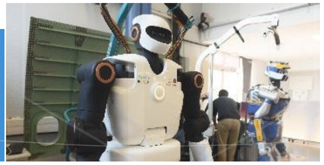
  h: q $\rightarrow h(\text{q})$, C^1 continuous function

- Direct kinematics

  v: $q,\dot{q} \rightarrow v\,(q,\dot{q}) = J(q)\,\dot{q}$



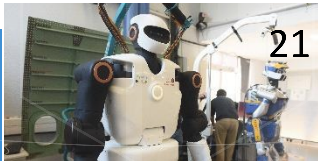
- Inverse geometry
  - Ill defined, singular points
  - Numerical inversion by Newt... cent
- Integration of the descent
  - Robot trajectory
  - Quadratic problem at each step



Display

- Gepetto-viewer is a display server
  - Python can create a client to this server

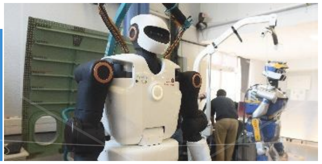
- Gepetto-viewer does not know the kinematic tree
  - Pinocchio must place the bodies
  - RobotWrapper is doing that for you (not in C++)



Spatial algebra

- M: placement in SE3
- ν: "spatial" velocity of SE3
  - $\dot{M} = \nu \times M$
- α: "spatial" acceleration in SE3
  - $\nu \in M^6 = se(3)$
  - $\alpha \in M^6 = se(3)$
  - $\alpha = \dot{\nu}$
- ϕ: "spatial" force in SE3
  - Power $P = <\phi \mid \nu> = {}^A\phi^T \, {}^A\nu \in R$
  - $\eta \in F^6$: momentum
- Y: "spatial" inertia in SE3
  - $\eta = Y \nu$
  - $\phi = Y \alpha$

i am a robot



Placement

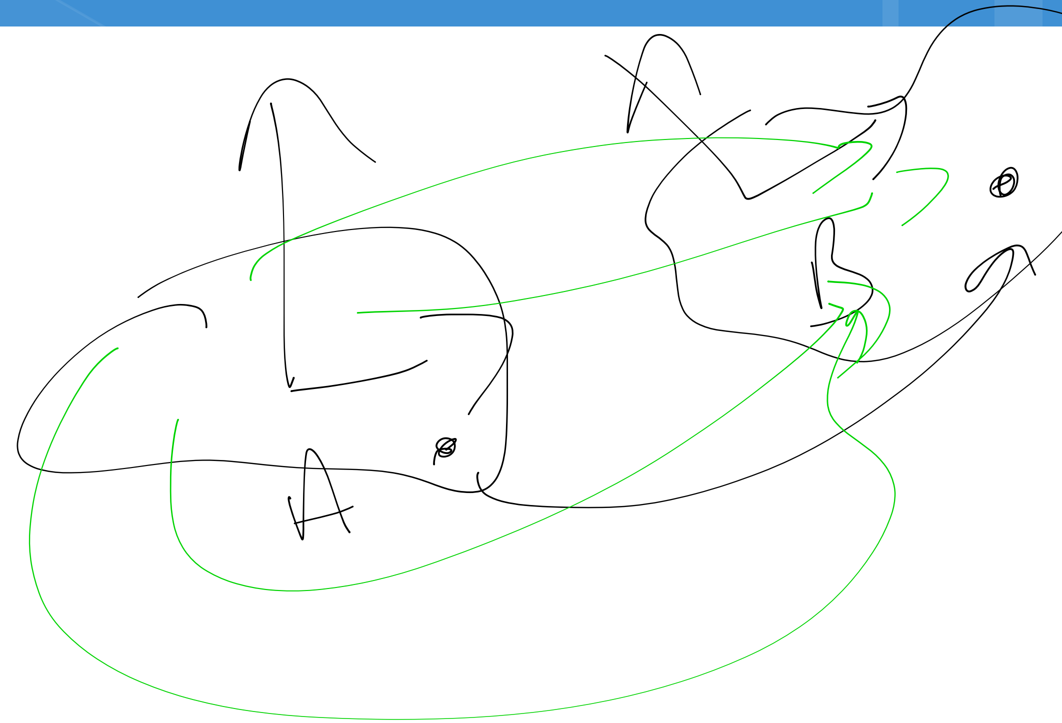
$^A M_B \in SE(3)$

$= \begin{pmatrix} ^A R_B & ^A\vec{AB} \end{pmatrix}$

$^B p \longrightarrow {}^A p$

$^A M_B \, ^B p = {}^A p$

$M_{AB} \, ^B p$

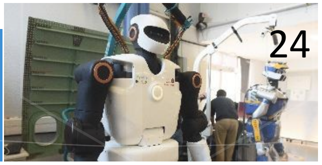


Displacements

$${}^{A}\Pi_{B}(t)$$

$${}^{A}p(t) = {}^{A}\Pi_{B}(t)\,{}^{B}p$$

$${}^{A}\dot{\Pi}_{B}$$



Velocities

$$\dot{M} = \nu \times M$$

$$\uparrow$$

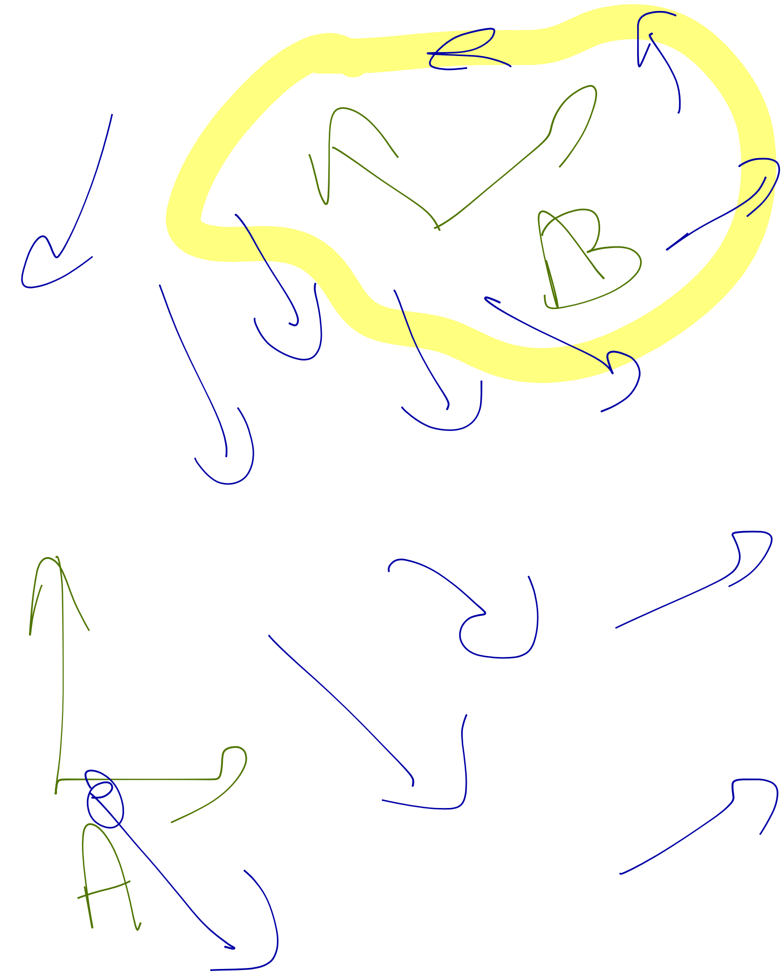
$$\nu \in \mathbb{R}^6 = \begin{pmatrix} v \\ \omega \end{pmatrix}$$

$$ {}^A\dot{M}_B = {}^A\nu \times {}^A M_B$$

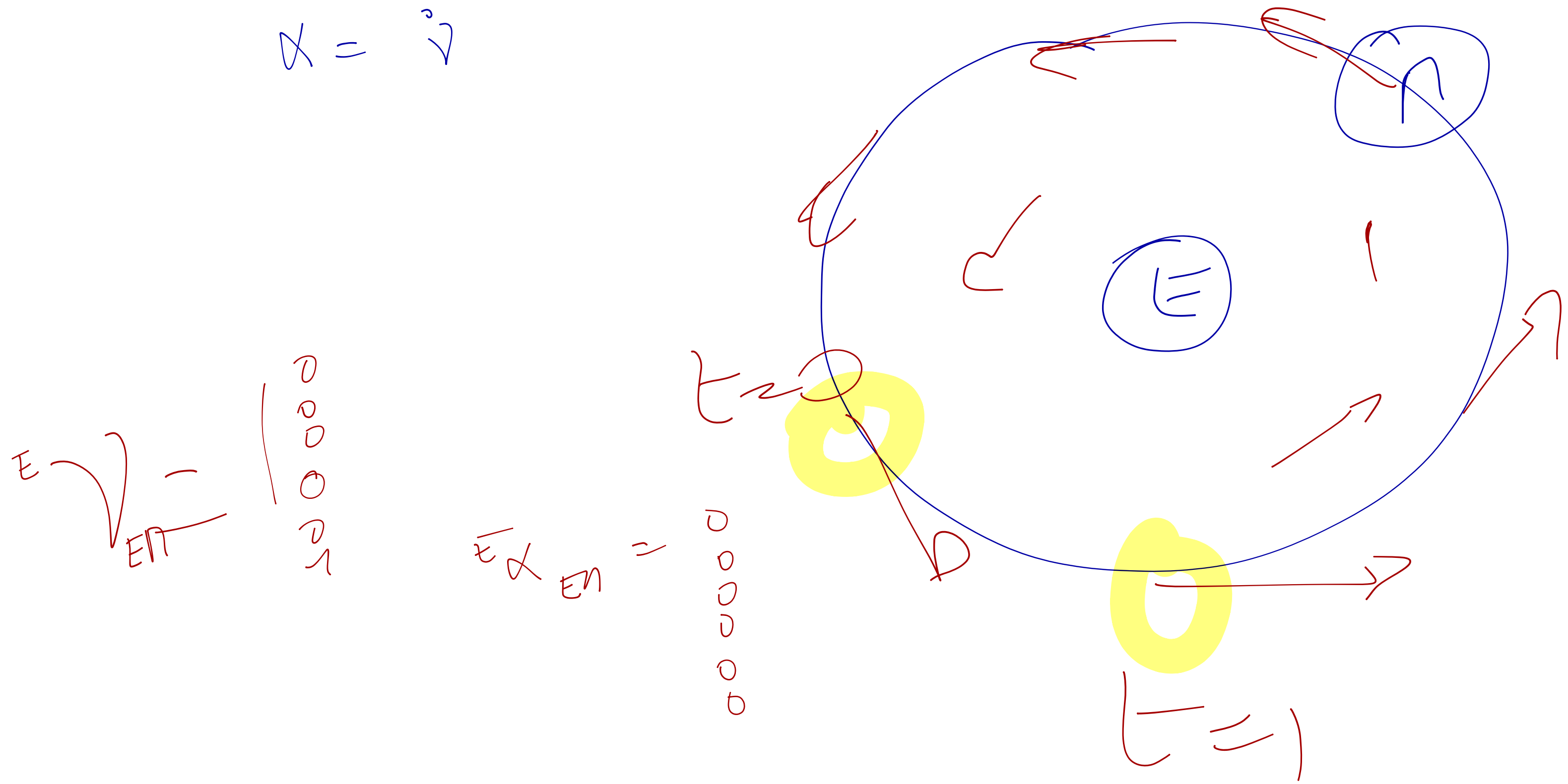
$$ {}^A\dot{M}_B = {}^A M_B \, {}^B\nu \times$$

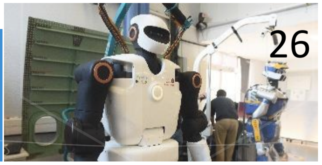
$$ {}^A\nu = \begin{pmatrix} {}^A v_A \\ {}^A\omega \end{pmatrix} \rightarrow \begin{matrix} {}^A v_B \\ {}^A\omega \end{matrix}$$

$$ {}^B\nu = \begin{pmatrix} {}^B v_B \\ {}^B\omega \end{pmatrix}$$



Acceleration

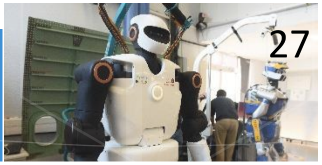




Derivatives

$${}^{A}\frac{d}{dt}\nu = \frac{d}{dt}\,{}^{A}\nu + {}^{A}\nu_{A} \times {}^{A}\nu$$

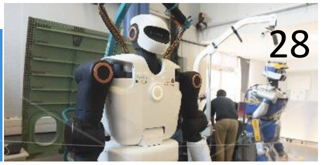
$${}^{A}\frac{d}{dt}\phi = \frac{d}{dt}\,{}^{A}\phi + {}^{A}\nu_{A} \times {}^{A}\phi$$



Inertias

$$ {}^{c}Y = \begin{pmatrix} m I_3 & 0 \\ 0 & I_c \end{pmatrix} $$

$$ Y : \quad x \rightarrow \varphi = Y x $$



$${}^B p \longrightarrow {}^A p = {}^A M_B \, {}^B p$$

$${}^A v \longleftrightarrow {}^B v$$

$${}^A X_B$$

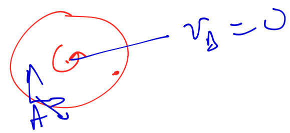
$${}^A v = \begin{pmatrix} {}^A R_B & {}^A AB \times {}^A R_B \\ 0 & {}^A R_B \end{pmatrix} {}^B v$$

$${}^A \varphi^T \, {}^A v = {}^B \varphi^T \, {}^B v$$

$${}^A \varphi = \begin{pmatrix} {}^A R_B & \\ {}^A AB \times {}^A R_B & {}^A R_B \end{pmatrix} {}^B \varphi$$

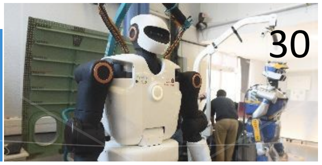
$${}^A X_B^{-1\,T} = {}^A X_B^{-T}$$

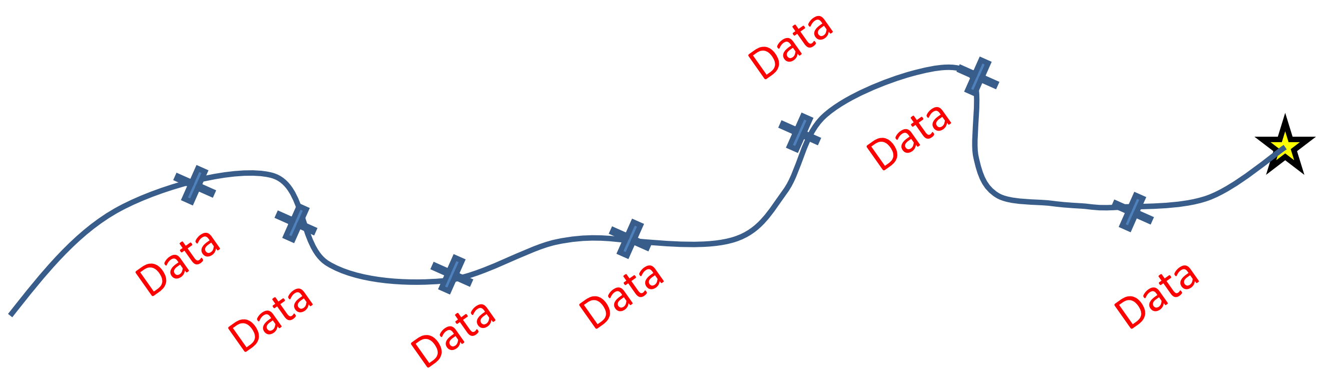
$$v + AB \times \omega \leftarrow \begin{pmatrix} 1 & AB \times \\ & 1 \end{pmatrix} \begin{pmatrix} v \\ \omega \end{pmatrix}$$



Model and data

- Pinocchio.Model should be constant
  - Kinematic tree, joint model, masses, placements …
  - Plain names used here
- Pinocchio.Data is modified by the algorithms
  - oMi, v, a
  - J, Jcom
  - M
  - tau, nle
- 1 Model, several Data

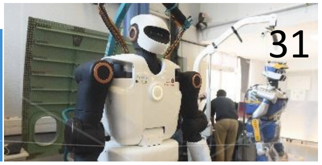


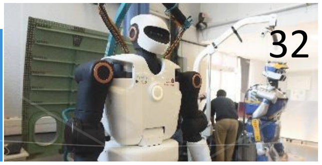


$$\min_{X,U} \; l_T(x_T) + \sum_{t=0}^{T-1} l(x_t, u_t)$$

1 model

$$\text{s.t.} \quad x_{t+1} = f(x_t, u_t)$$





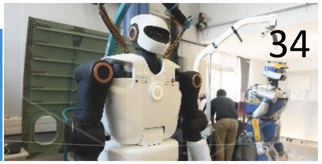
Forward kinematics

Forward kinematics

- pinocchio.forwardKinematics(rmodel,rdata,
  q,vq,aq)

- Compute all the joint placements in data.oMi

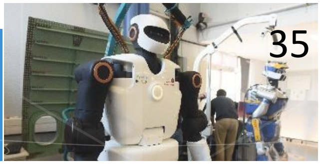
- M = data.oMi[-1] : last placement
- R = M.rotation
- p = M.translation



NumPy Array vs Matrix

- Pinocchio works with NumPY.Matrix
  - R is a matrix
  - p is a 1d matrix: p.shape == (3,1)
  - You can multiply R*p

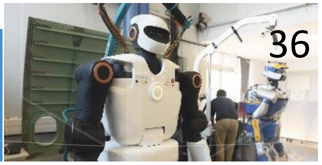
- NumPy works better with Array
  - np.zeros([3,3]) is an array
  - You cannot multiply array
    - Use np.dot   or obtain a coefficient-wise multiplication
- SciPy works with array too



SciPy optimizer

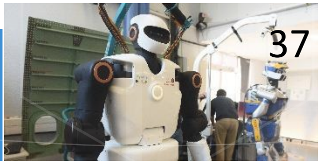
```
from scipy.optimize import fmin_slsqp
fmin_slsqp?


fmin_slsqp(x0 = np.zeros(7),
           func= costFunction,
           f_eqcons = constraintFunction,
           callback = callbackFunction)
```



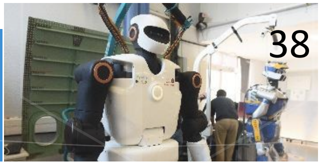
SciPy optimizer

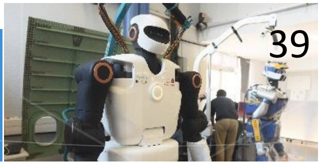
- Make the optimization problem a class:
  - Problem parameters in the __init__
  - Cost method taking x as input
  - Constraint and callback method if need be



SciPy optimizer

```
class OptimProblem:
    def __init__(self,rmodel):
        # Put your parameters here
        self.rmodel = rmodel
        self.rdata = self.rmodel.createData()
    def cost(self,x): return sum( x**2 )
    def callback(self,x): print(self.cost(x))
pbm = OptimProblem(robot.model)
fmin_slsqp(x0=x0,func=pbm.cost,callback=pbm.callback)
```






Frames &+

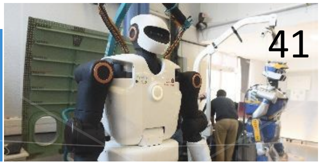
Joint and frames

$^{o}M_f(q) = {}^{o}M_i(q)\,{}^{i}M_f$

fk constant

- Joint frames
  - Skeleton of the kinematic chain
  - Computed by forward kinematics in rdata.oMi

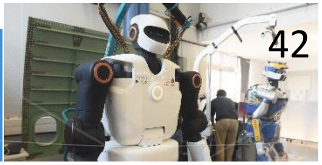
- “Operational” frames
  - Added as decoration to the tree
  - Placed with respect to a joint parent
  - Stored in rmodel.frames
  - Computed by updateFramePlacements in rdata.oMf

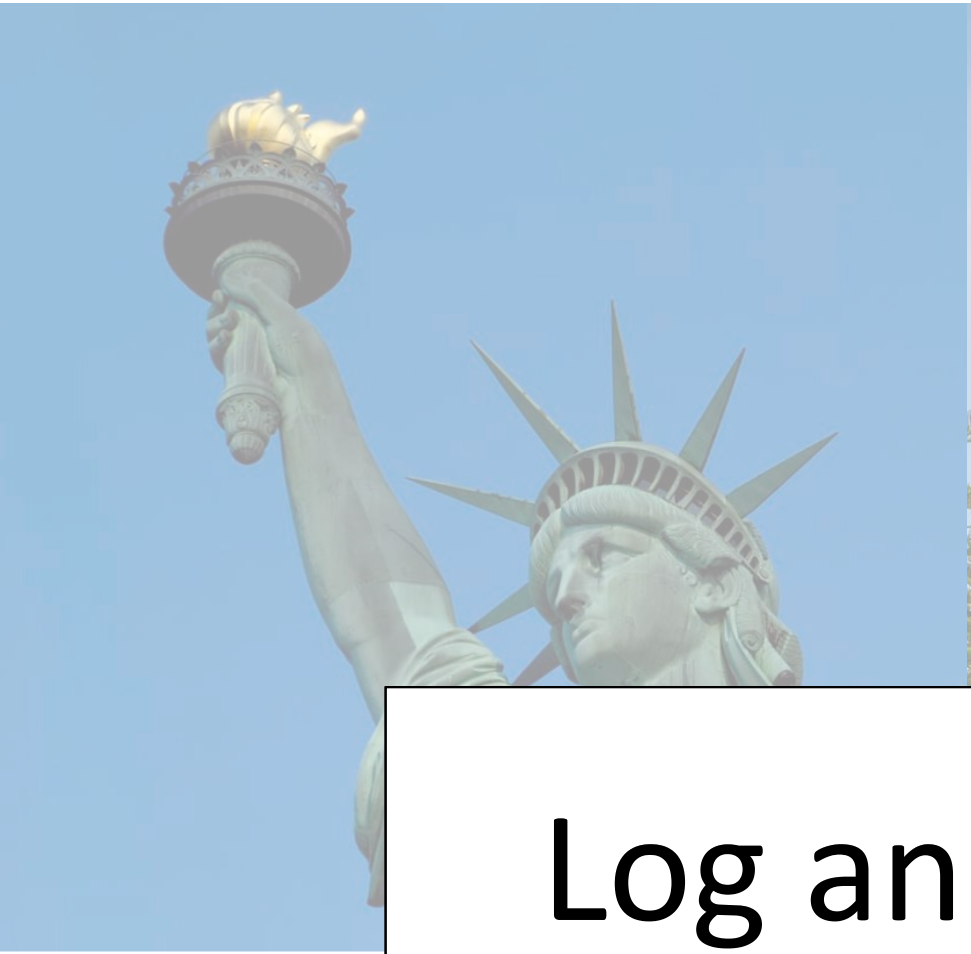


Joint limits

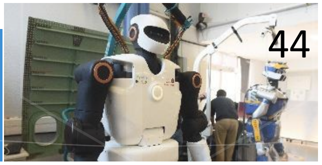
- Parsed from urdf
- In rmodel.lowerPositionLimits and rmodel.upperPositionLimits

- Beware, infinity by default





Log and difference



Position versus placement

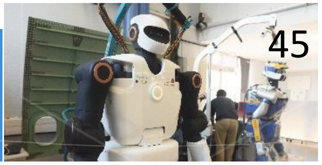
$p(q): \mathbb{R}^{nq} \rightarrow \mathbb{R}^3$

- Difference of positions
  - residuals = p-p*
- Diffence of rotations
  - residuals = $\log_3(R^T R^*)$ $= R^* \ominus R$

$R = {}^0R_i \qquad R^* = {}^0R_*$

$R^T R^* = {}^iR_0 {}^0R_* = {}^iR_*$

- Diffence of placements
  - residuals = $\log_6(M^{-1} M^*)$

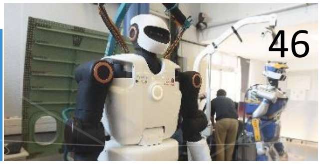


Free flyer joint

- Revolute joint
  - q of dimension one, $v_q = \dot{q}$
- Free flyer

x, y, z, q

$\|q\| = 1$



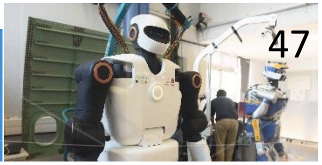
Integrate and differenciate

$$q_{next} = \text{pinocchio.integrate}(q, v_q) \quad \in Q$$

$$q_{next} = q \oplus v_q$$

$$\Delta q = v_q = \text{pinocchio.difference}(q_1, q_2) \quad \in T_{q1}Q$$

$$\Delta q = q_2 \, (-) \, q_1$$

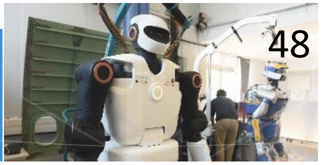


Integrate and differenciate

- On a Matrix Lie Group

$$q \oplus v_q = \text{Matrix}(q)\,\exp(\text{skew}(v_q)) = Q\,\text{Exp}(v_q)$$

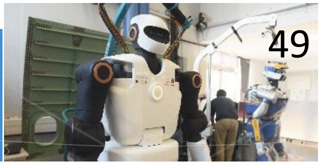
$$q_2\,(-)\,q_1 = \log(\,Q_2^{-1}\,Q_1)$$



Optimization with Q / TQ

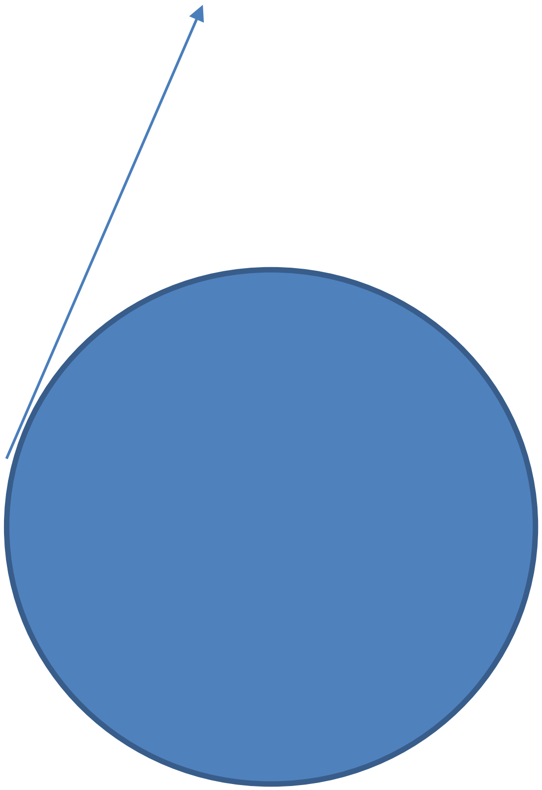
- $q = (x,y,z, \underline{q}, \ldots)$ with $\underline{q}$ unitary

- What is the result with a solver ?



Solution 1: normalized

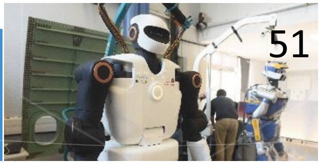
```
def constraint_q(self, x):
        return norm(x[3:7])-1)
```



Solution 2: reparametrize

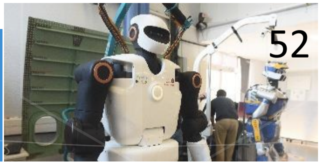
- We represent q
  - as the displacement v_q
  - from a reference configuration q_0

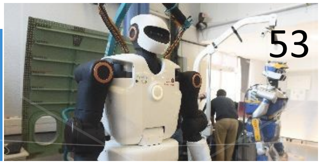
$$q = q_0 \oplus v_q$$



Random configuration

pinocchio.randomConfiguration(rmodel)

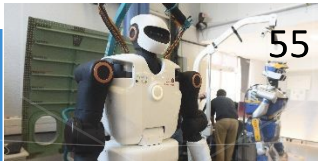


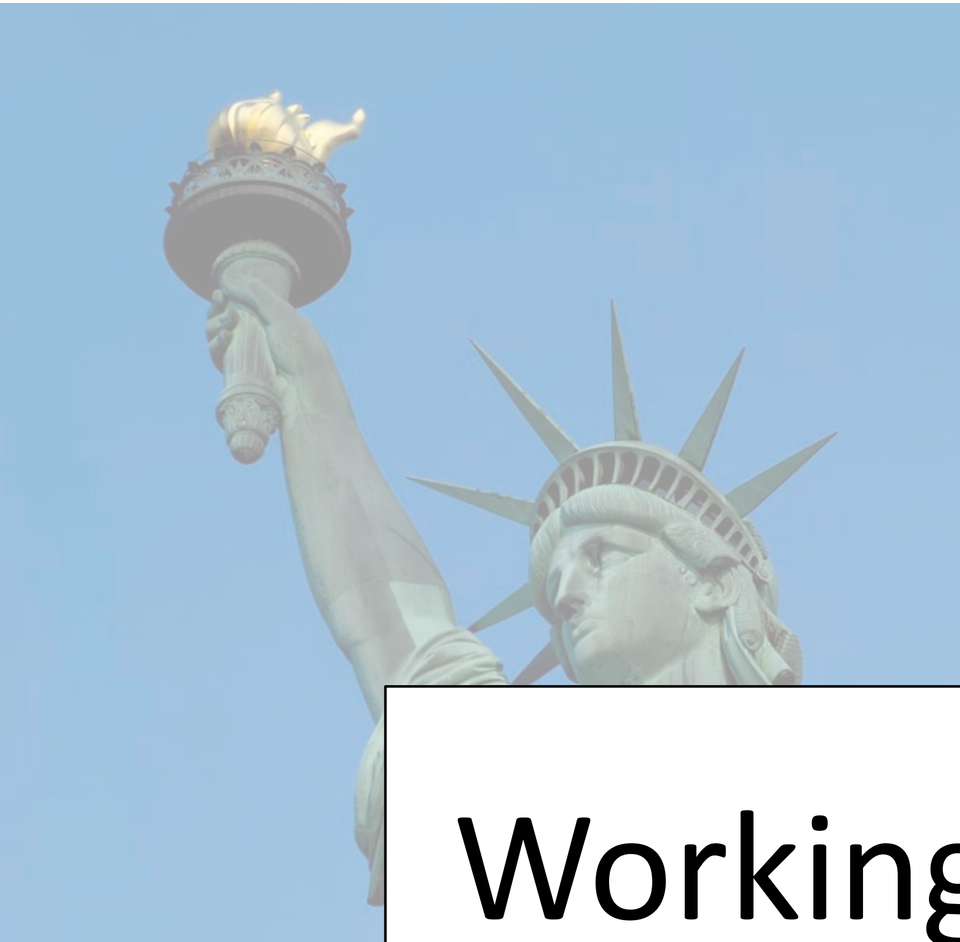


European Commission | Horizon 2020 European Union funding for Research & Innovation

memmo

Part II
Differencial kinematics

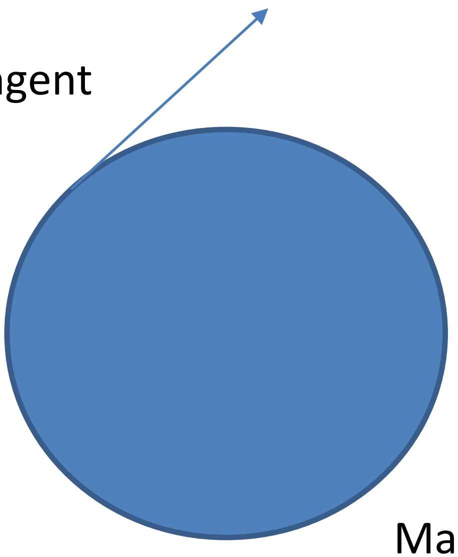




Working in manifolds

Manifold-to-manifold maps

- Function f:
  - From manifold to manifold
  - $M: q \in Q \rightarrow M(q) \in SE3$
- Derivative F_x
  - From tangent to tangent
  - $M_q : v_q \in TQ \rightarrow \nu \in M^6$
- $\nu(q,v_q) = J(q)\, v_q$
  - J: from vector to vector



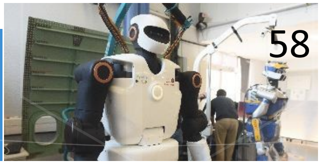
Consequence

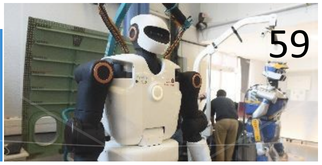
- You should know in which tangent space you work
  - Typically at the local point, or at the origin

$$ {}^{A}v^{i}(q,v_q) = {}^{A}J^{i}(q) \; v_q $$

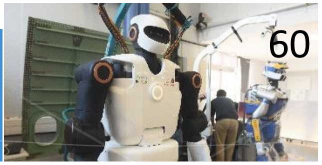
- **In Pinocchio,**

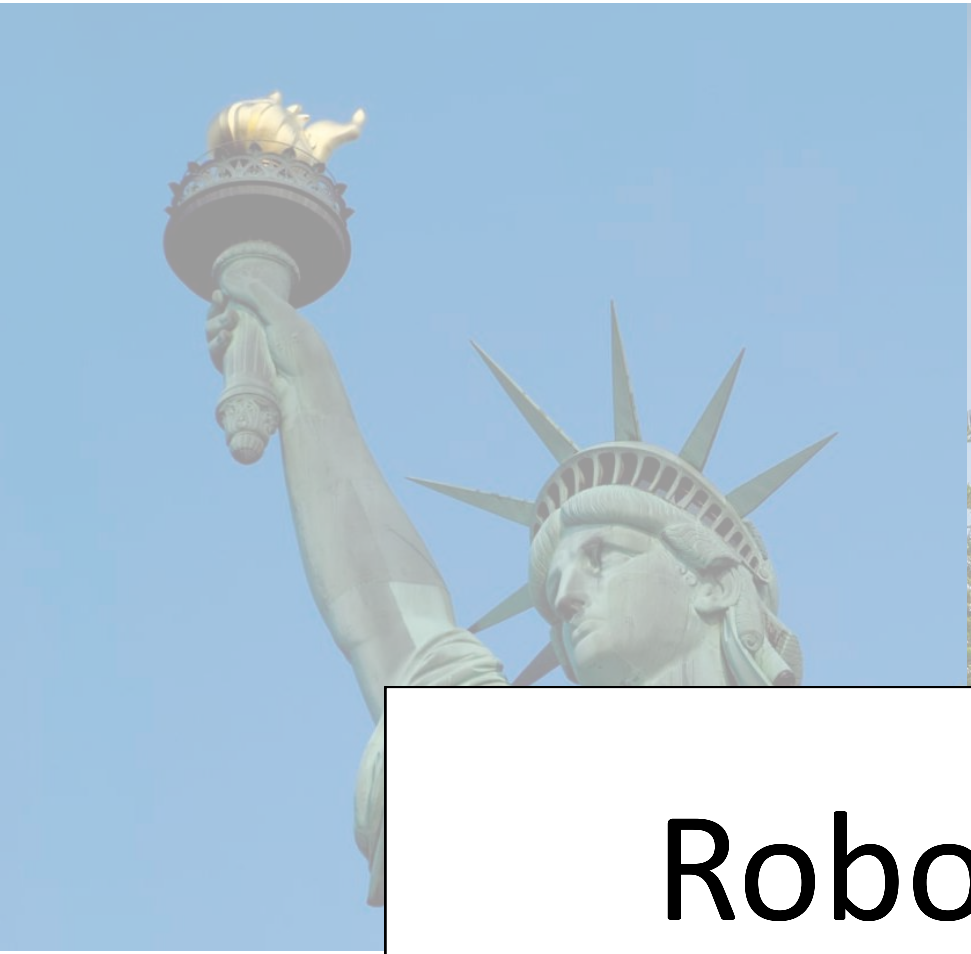
  the velocity are often represented locally
  - Velocity of the free flyer in the frame of the hip



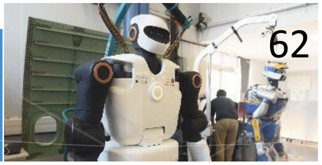


Finite differences





Robot jacobian

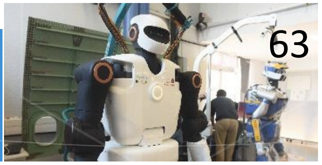


Joint jacobian

- Computed by two steps:
  - computeJointJacobians(rmodel,rdata,q)
  - getJointJacobian(rmodel,rdata,IDX,LOCAL/GLOBAL)
- From local to global

$${}^{o}J\, v_q = {}^{o}v \qquad {}^{o}X_i\, {}^{i}v = {}^{o}v$$

$${}^{i}J\, v_q = {}^{i}v \qquad {}^{o}X_i\, {}^{i}J = {}^{o}J$$



Frame jacobian

- Just add the additional displacement

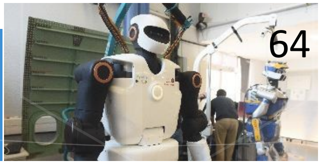
$^{i}X_{f}$ $\quad$ ^{i}J $\quad$ $^{f}J = {}^{i}X_{f}^{-1}\,{}^{i}J$

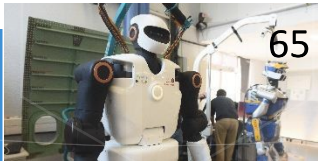
- 4 steps

  ComputeJointJacobian

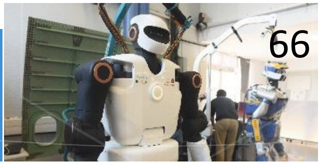
  updateFramePlacements

  getFrameJacobian





Cost jacobian



Chain rule

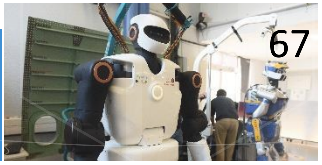
Cost(q) = log(M(q))

Cost = log o M

Cost_q = log_M M_q

M_q ?

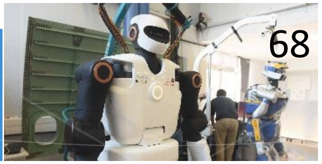
log_M

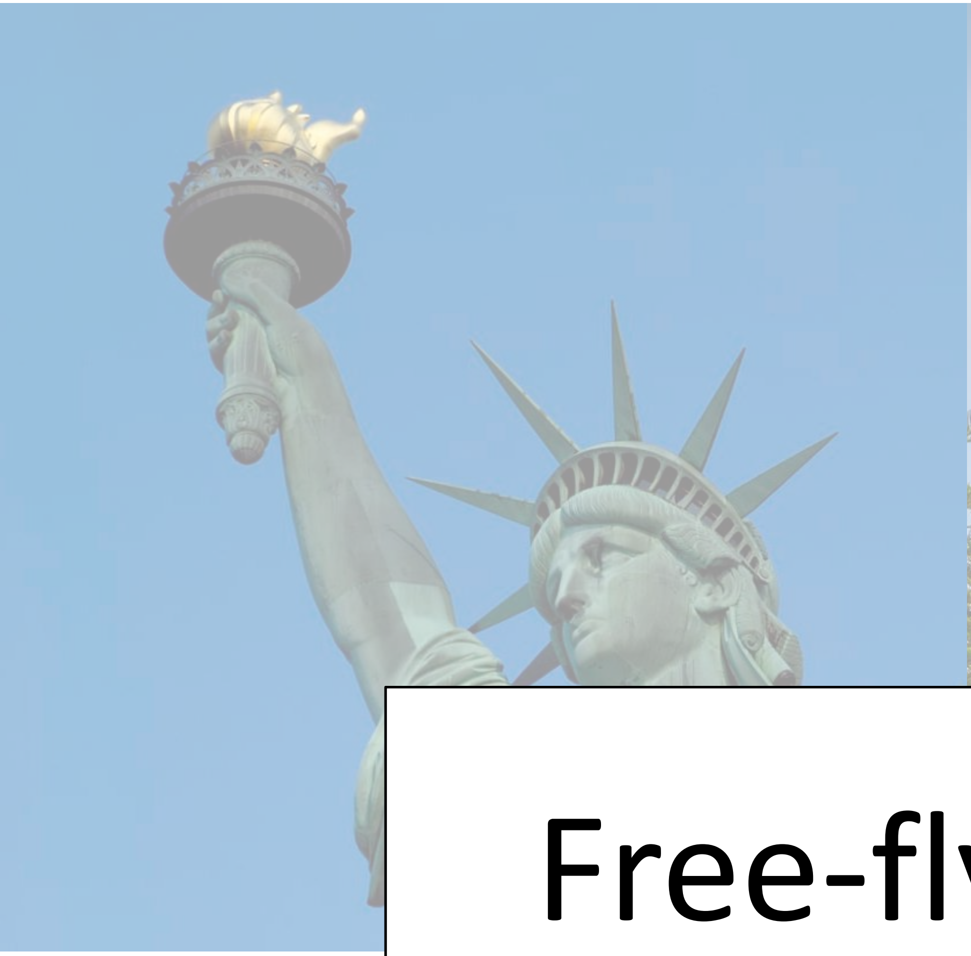


Log jacobian

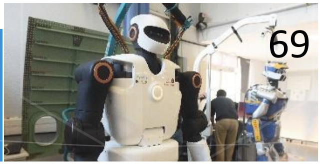
- Computed in pinocchio

- Pinocchio.Jlog



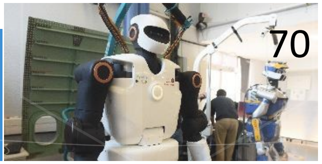


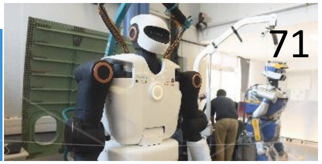
Free-flyer reparam



- Recall $q = q_0 \oplus v_q = r(v_q)$
- $c(v_q) = \log(M(r(v_q)))$

- Chain rule …
  - $r(v) = \text{integrate}(q_0, v)$
  - R_v = dIntegrate_dv (q_0,v)
  - Not implemented yet in Pinocchio
  - But it is the inverse of dDiffence which is implemented



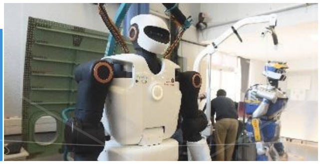


Part III
Dynamics

Dynamics of articulated bodies

- Dynamic equation of the robot

$$M(q)\dot{v}_q + c\left(q, v_q\right) + g(q) = \tau_q$$

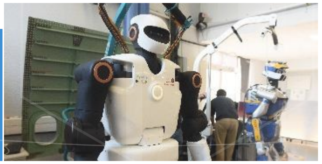


Dynamics of articulated bodies

- Dynamic equation of the robot

$$M(q)\dot{v}_q + c\left(q, v_q\right) + g(q) = \tau_q$$

- Actuation of the robot
  - Fixed manipulator: $\tau_q = \tau_m$
  - Floating robot: $\tau_q = \begin{bmatrix} 0 \\ \tau_m \end{bmatrix} = S^T \tau_m$
  - Robot in contact: : $\tau_q = S^T \tau_m + J^T \phi$



An intuition of M?

RNEA algorithm

$M \dot{v}_q + b(q, v_q) = \tau_q$ $\longrightarrow$ $\mathrm{RNEA}(q, v_q, \dot{v}_q)$

$NQ = 30 \longrightarrow t = 2.5\ \mu s$

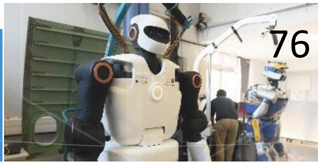
$M^{-1}(\tau_q - b(q, v_q)) = \dot{v}_q$ $\longrightarrow$ $\mathrm{ABA}(q, v_q, \tau_q)$

$10\ \mu s$

M : CRBA $2\ \mu s$

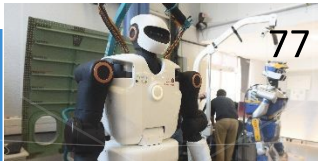
$b = \mathrm{rnea}(q, v_q, 0)$ $\longrightarrow$ computeAllTerms( ) $\longrightarrow$ b, M

$5\ \mu s$



Other algorithms

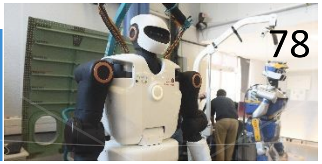
- CRBA
- ABA
- ComputeAllTerms

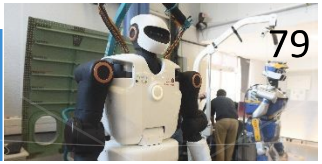


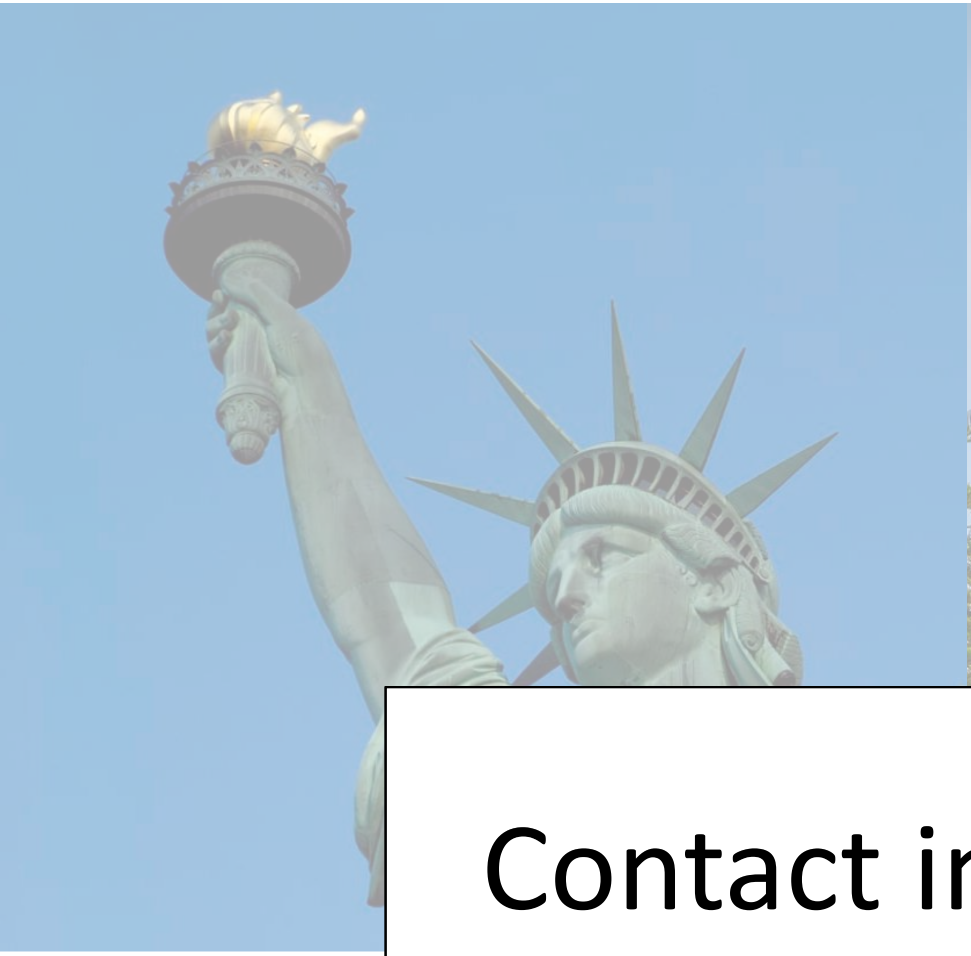
RNEA with forces

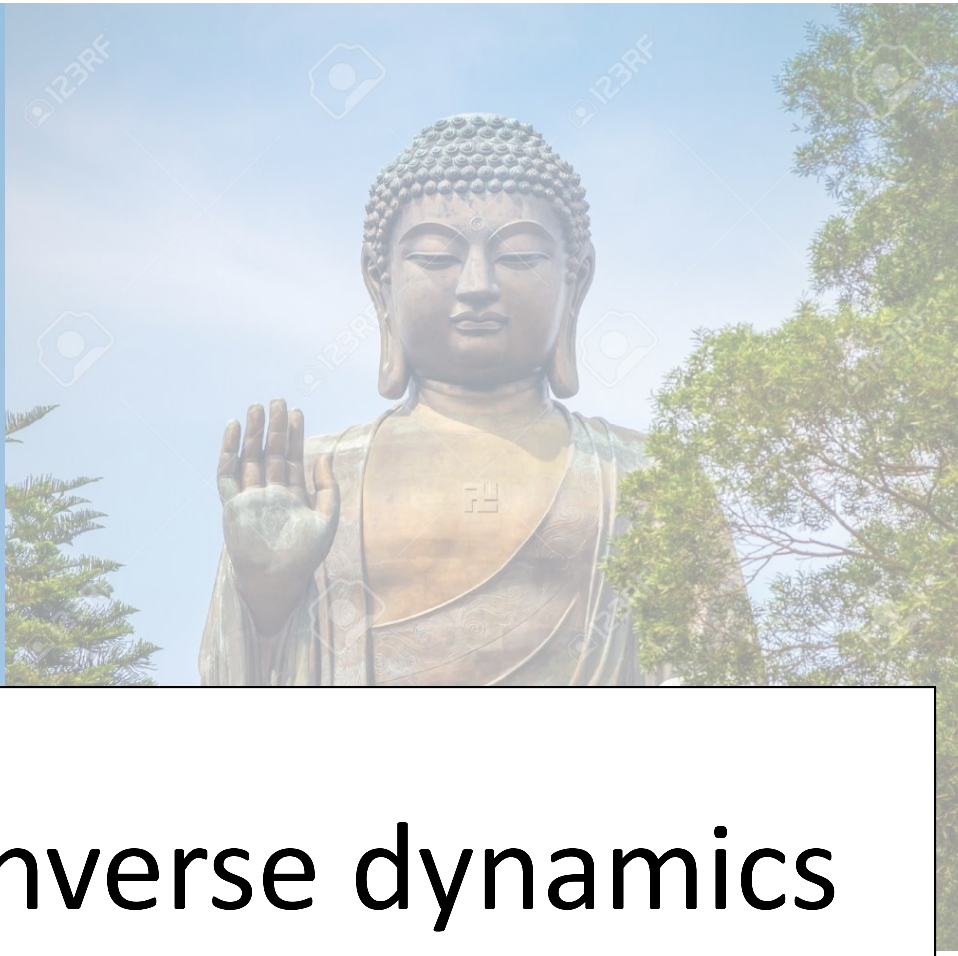
$$M \dot{v}_q + b(q, v_q) + J^T \varphi \rightarrow \text{rnea}(q, v_q, \dot{v}_q, [\varphi_0 \dots \varphi_N])$$

${}^i\varphi$ ← only for joints

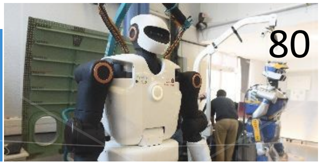






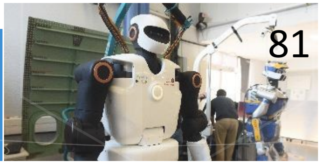


Contact inverse dynamics



Optimization problem

$$\min_{\tau,\varphi} \left\| M\dot{v}_q + b(q, v_q) - \tau - J^T\varphi \right\|^2$$



Optimization problem

- OptimProblem class
- With a x2var function that makes the dispatch

- It is a linear problem: we should not use NLP

- See TSID tomorrow