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

memmo

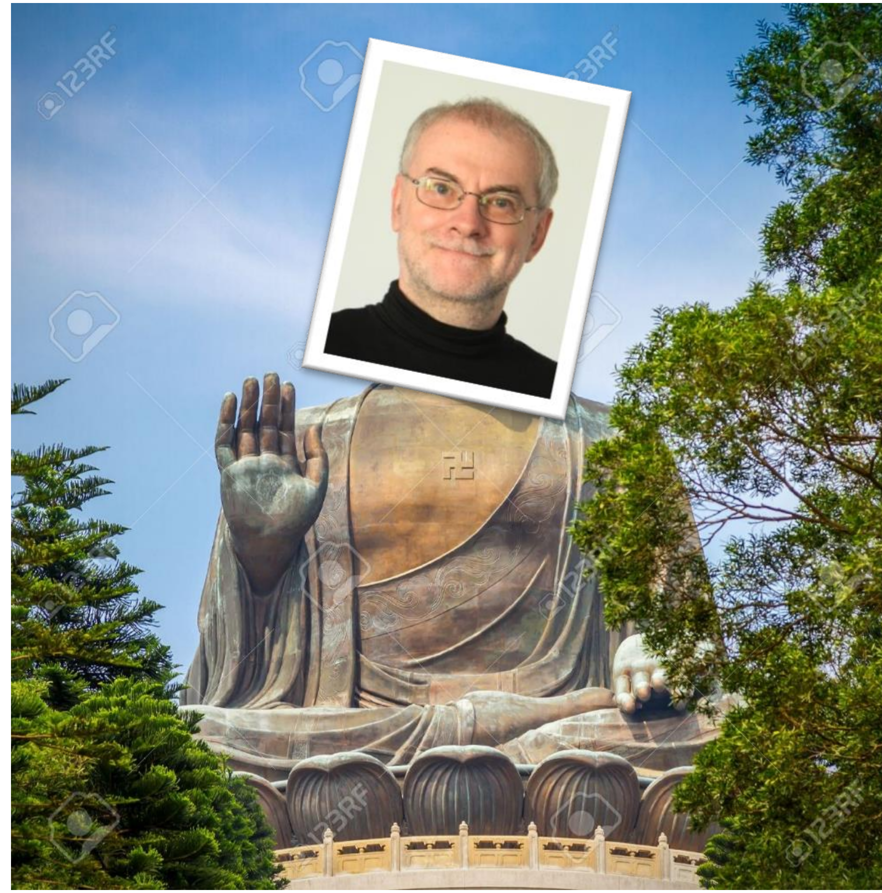
Pinocchio

Fast forward & inverse dynamics

Nicolas Mansard
(CNRS)

Gurus

Justin Carpentier (INRIA)

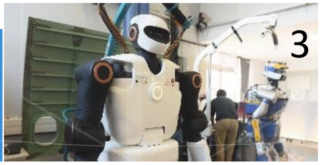


Roy Featherstone (IIT)

WWW Material

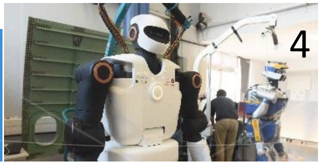
- Web site
  - https://stack-of-tasks.github.io/pinocchio
- Doxygen
  - Documentation tab on github.io
- Tutorials:
  - Practical exercices in the documentation

- Also use the ? In Python



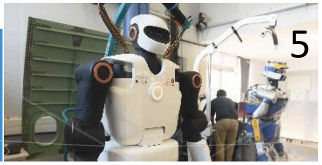
Contributing to Pinocchio

- 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

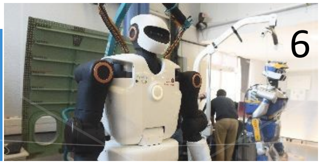
- C++ Library
  - Fast, careful implementation
  - Using curiously recursive template pattern (CRTP)
  - You likely don’t want to develop code there
  - Using it is not so complex (think Eigen)
- Python bindings
  - A 1-to-0.99 map from C++ API to Python API
  - Start by developing in Python
  - Beware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



Modeling and optimizing

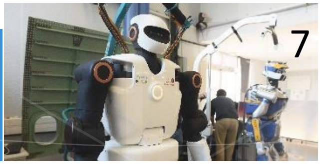
- Pinocchio is a modeling library
  - Not an application
  - Not a solver
  - Some key features directly available

- 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

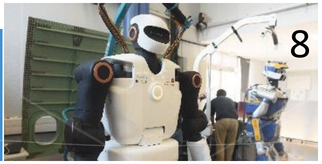
- URDF parser
- Forward kinematics and Jacobians
- Mass, center of mass and gen.inertia matrix
- Forward and inverse dynamics
- Model display (with Gepetto-viewer)
- Collision detection and distances (with HPP-FCL)
- Derivatives of kinematics and dynamics
- Type templatization and code generation



TSID

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

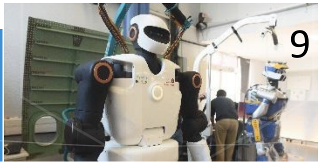
- Formulation of tasks
- Contact models
- QP resolution



Crocoddyl

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

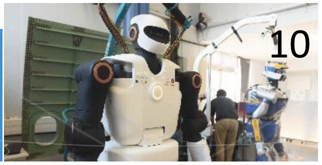
- DDP optimizer
- Task/cost formulation



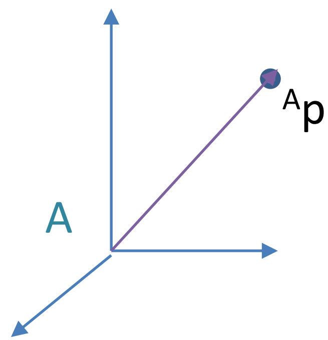
HPP planner

- 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

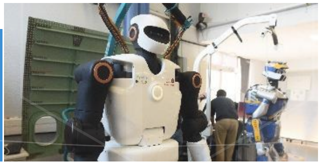


Representing the physical world



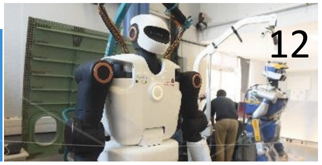
This is a point

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

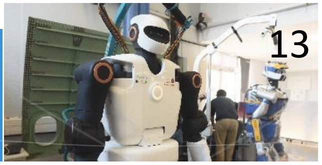


Representing the physical world

- 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



jupyter

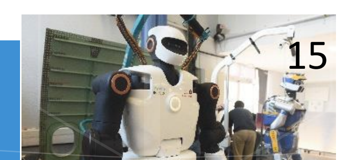
1.1 Load and display

Load a model

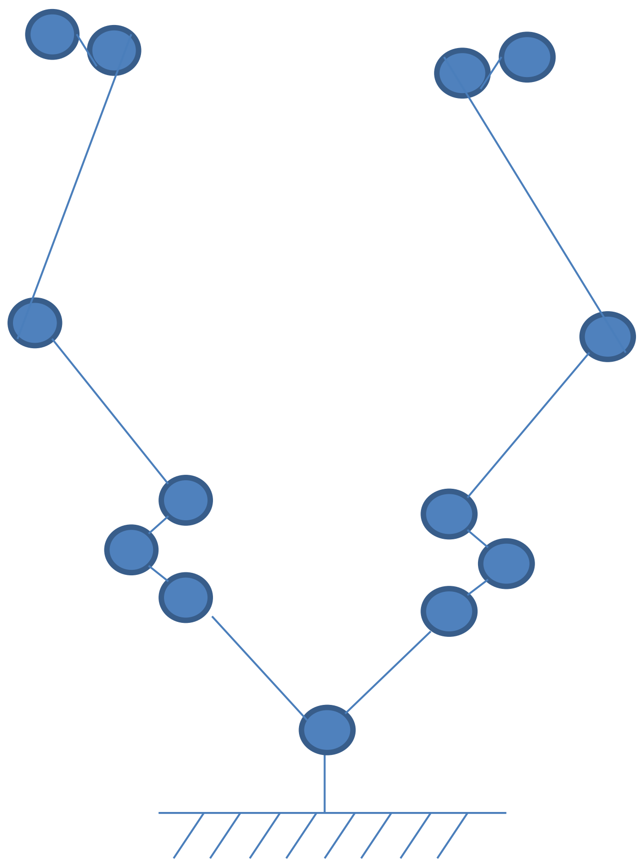
- Pin.buildFromUrdf

- Package example_robot_data
  - A small library of our favorite robots
  - Python scripts to load them easily

```
import example_robot_data as robex
robot =robex.loadTalosArm()
```



Kinematic tree



Wrist 2
Wrist 1

Elbow

Shoulder 3
Shoulder 2
Shoulder 1

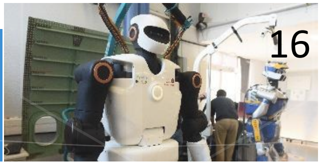
Torso

Universe (joint #0)

Name
Type
Parent
Placement

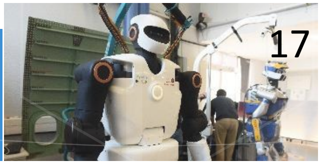
Mass
CoM

Geometries
Op frames



Kinematic tree

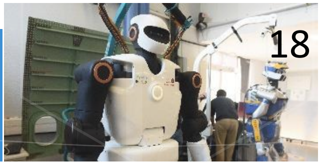
- Inside robot model:
  - joints: joint types and indices
  - names: joint names
  - jointPlacements: constant placement wrt parent
  - parents: hierarchy of joints representing the tree
- No bodies
  - masses and geoms are attached as tree decorations
- First joint represent the universe
  - If nq==7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



Display

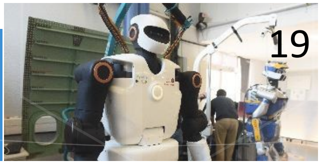
- External display servers
  - Python can create a client to this server
  - Gepetto viewer
  - MeshCat
  - Beta version of a Panda server

- The viewers does not know the kinematic tree
  - Pinocchio must place the bodies
  - `pin.visualize` is doing that for you (not in C++)



Model, data and algorithms

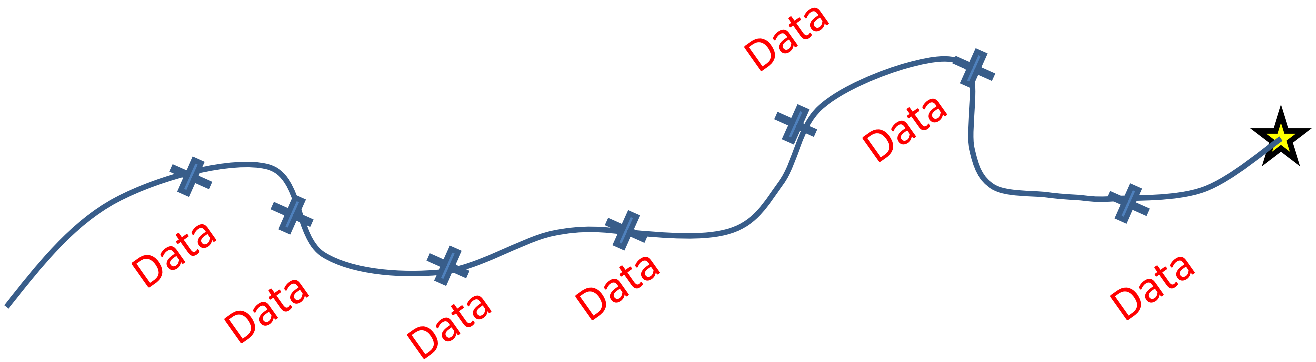
- `pinocchio.Model` should be constant
  - Kinematic tree, joint model, masses, placements …
  - Plain names used here
- `pinocchio.Data` is modified by the algorithms
  - `oMi, v, a`
  - `J, Jcom`
  - `M,`
  - `tau, nle`
- 1 Model, several Data



1.2 Pinocchio's philosophy

(model, data and algos)

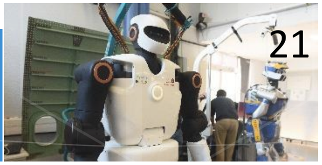
Model, data and algorithms



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

1 model

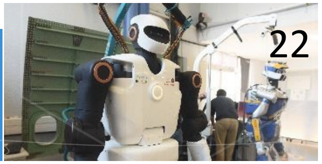
$$\text{s.t.} \quad x_{t+1} = f(x_t, u_t)$$



Model, data and algorithms

- Algorithms:
  - With model and data in input
  - Store final (and some intermediary) results in data
  - Often return the main results

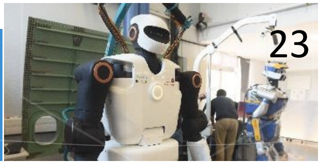
```
pin.randomConfiguration(rmodel)

pin.forwardKinematics(rmodel,rdata,q)
rdata.oMi[jointIndex]
```



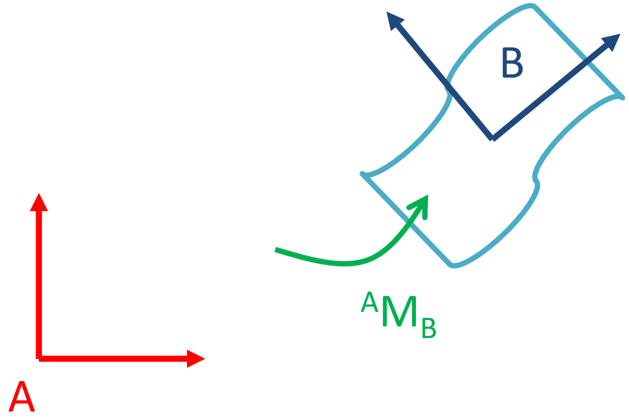
Forward kinematics

- `pin.forwardKinematics(rmodel,rdata, q,vq,aq)`
- `q` -> propagates placements (= forward geometry)
- `vq` -> also propagates velocity (= *differential* kinematics)
- `aq` -> also propagates accelerations (= 2nd order FK)

- Compute all the joint placements in `data.oMi`
- `M = data.oMi[jointIndex]` : placement of <jointIndex>
- `R = M.rotation`
- `p = M.translation`



Placement

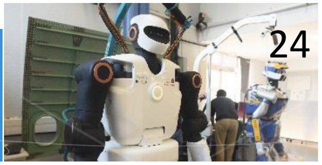


$$^{A}M_{B} = \begin{bmatrix} {}^{A}R_{B} & {}^{A}\overrightarrow{AB} \\ 0 & 1 \end{bmatrix}$$

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

$$^{A}M_{B}\,{}^{B}M_{C} = {}^{A}M_{C}$$

```
aMb.translation   # 3d array
aMb.rotation      # 3x3 array
```

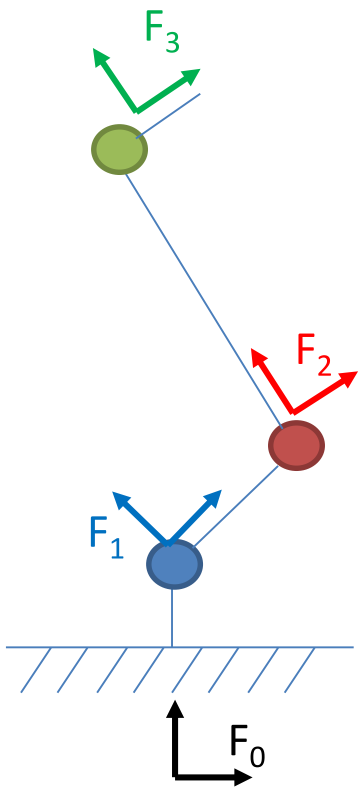


1.3 Cost 3d

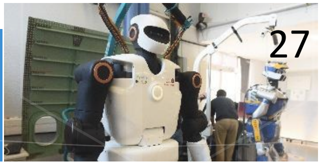
Joint frames
vs
operational
frames

Frames

- One joint = one joint frame
  - Attached to the joint output
  - F_0 is the "universe" world frame

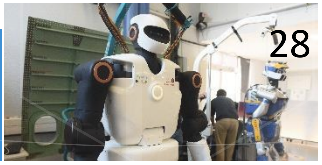


- Operational frames attached to joint frames
  - Name
  - Placement
  - parent



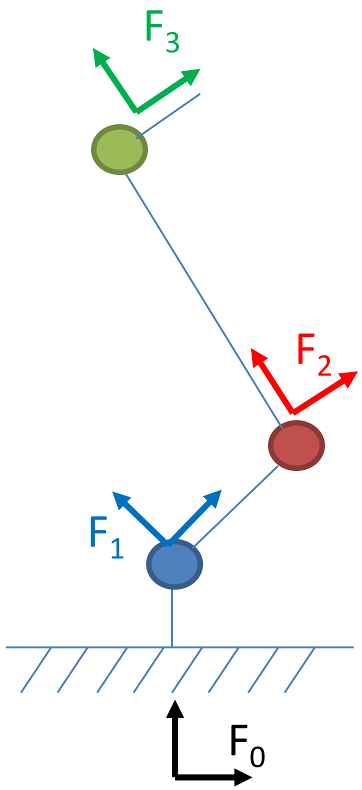
Joint and frames

- 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`



Frames

```
for f in rmodel.frames:
    print(f.name,f.parent)
```

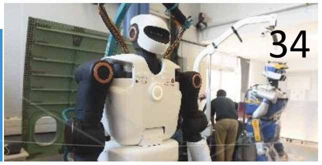



```
pin.frameForwardKinematics(rmodel,
                           rdata,q)
frameIndex = \
rmodel.getFrameId('myname')
rdata.oMf[frameIndex]
```

Cost model

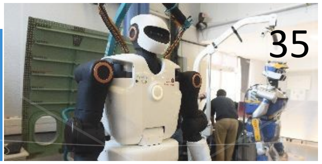
- For this tutorial only (ad-hoc code)
  - … but similar to the organization in crocoddyl

```
class Cost:
    def __init__(self,rmodel,rdata,viz=None):
        self.rmodel = rmodel
        self.rdata  = rdata
        self.viz = viz

    def calc(self,q):
        ### Add the code to recompute your cost here
        cost = 0
        return cost

    def callback(self,q):
        if viz is None: return
        # Display something in viz ...
```

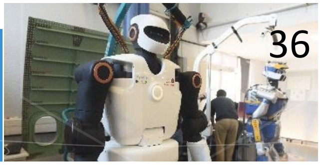
SciPy optimizer

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



SciPy optimizer

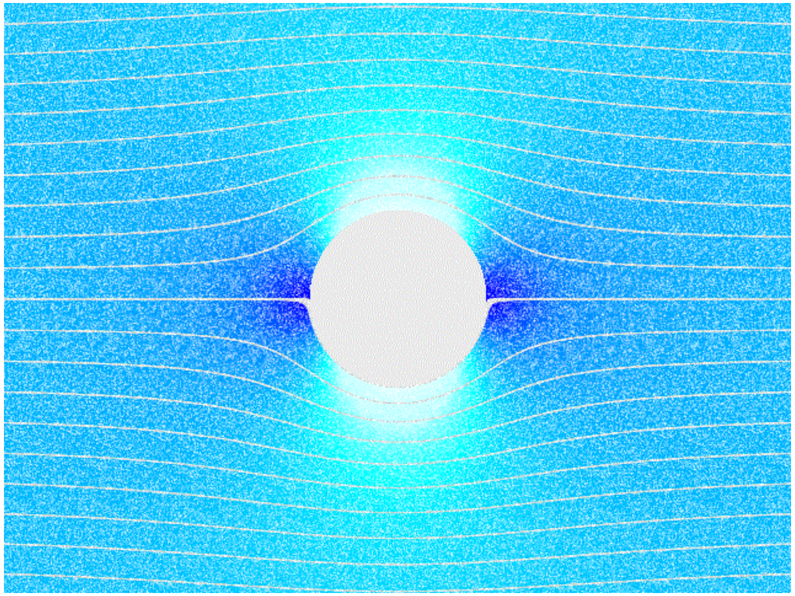
```
class OptimProblem:
    def __init__(self,rmodel):
        # Put your parameters here
        self.rmodel = rmodel
        self.rdata = self.rmodel.createData()
    def cost(self,x): return sum( x**2 )
    def callback(self,x): print(self.cost(x))
pbm = OptimProblem(robot.model)
fmin_slsqp(x0=x0,func=pbm.cost,callback=pbm.callback)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



1.4 Cost 6d

SE(3) and log

Displacement



https://upload.wikimedia.org/wikipedia/commons/b/b8/Inviscid_flow_around_a_cylinder.gif

- Between 2 frames:

  Each point

  moves to another point

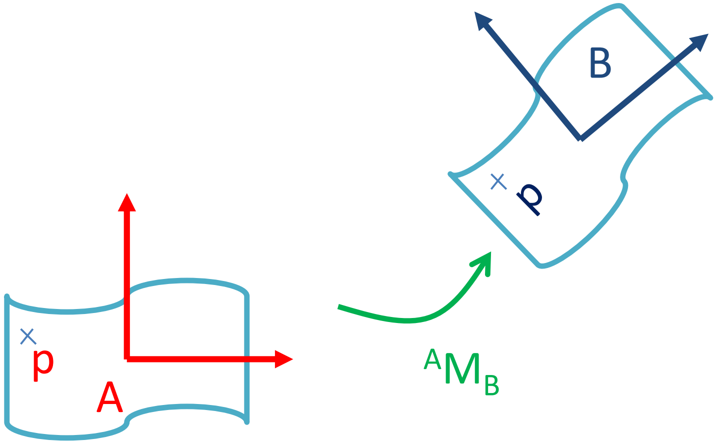
$$m: \quad p \in E^3 \rightarrow m(p)\ E^3$$

Rigid displacement

https://gfycat.com/fr/sleepycleanarcherfish

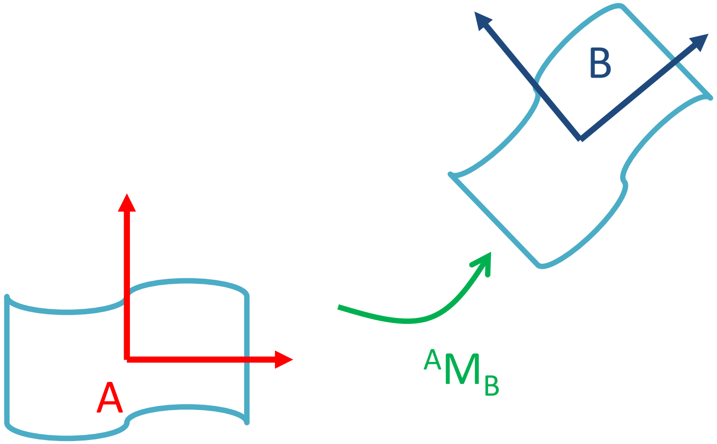
- Between 2 frames:
  - Each point
  - moves to another point
  - Distances are kept
  - Angles are kept

$$m: \quad p \in E^3 \rightarrow m(p)\ E^3$$



- ^AM_B=(^AR_B, ^AAB) represents the motion of all the points of the body

$$^Ap = {}^AM_B \, {}^Bp$$



- Rigid velocities

  = linear + angular velocity

- What is the velocity to transform $\mathcal{F}_A$ into $\mathcal{F}_B$ in 1 second ?

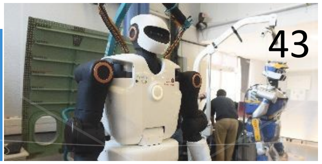
"SE(3) Logarithm" $\log({}^{A}M_{B})$

```
M=pin.SE3.Random()
pin.log(M).vector
```

Position versus placement

- Difference of positions
  - residuals = p-p*
- Diffence of rotations
  - residuals = $\log_3(R^T R^*)$ `pin.log3`
- Diffence of placements
  - residuals = $\log_6(M^{-1} M^*)$ `pin.log6`

`pin.log` (auto-switch based on type)

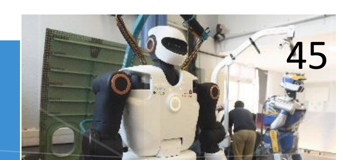


jupyter

1.5 redundancy

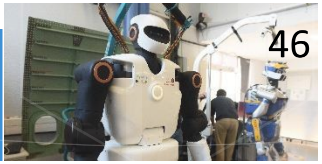
Posture cost

https://youtu.be/sZYBC8Lrmdo



Redundancy

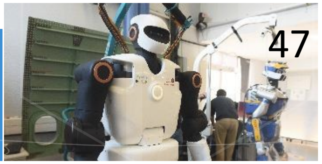
- Same 6D cost
- Hessian is ill-defined



Sum of costs

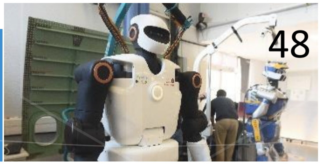
$$c(q) = w_1\, c_1(q) + w_2\, c_2(q)$$

$$\nabla c(q) = w_1\, \nabla c_1(q) + w_2\, \nabla c_2(q)$$



SciPy optimizer

```
from scipy.optimize import fmin_bfgs
fmin_bfgs?


fmin_bfgs(x0 = np.zeros(7),
          func= costFunction,
          fprime = gradFunction,
          callback=callbackFunction)
```

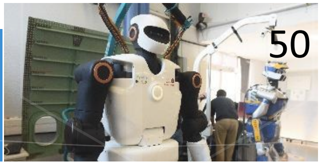


1.6 Dynamics

Whole body dynamics

$$M(q)\, a_q + b(q, v_q) = \tau_q$$

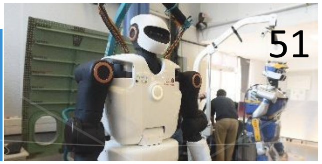
Explain vq,aq,tauq,M,b,



Whole body dynamics

$$M(q)\, a_q + b(q,v_q) = \tau_q$$

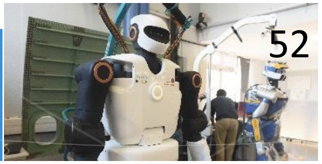
Explain Lagrange d/dv Ldot – d/dq L



Whole body dynamics

$$M(q)\, a_q + b(q, v_q) = \tau_q$$

Explain b and g

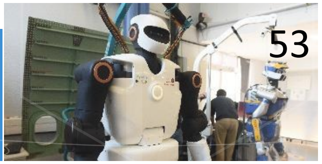


Gravity torque

- Gravity

$$g(q) = b(q, v_q = 0)$$

```
pin.computeGeneralizedGravity \
    (rmodel,
      rdata,q)
```



1.6 weighted gravity

Inverse and forward dynamics

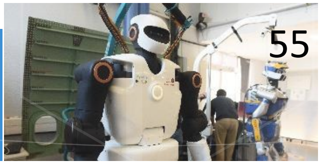
- Inverse dynamics

$$\tau_q = \text{invdyn}(q, v_q, a_q)$$

- Direct / forward dynamics

$$a_q = \text{dirdyn}(q, v_q, \tau_q)$$

Explain control / simu … explicit invdyn / dirdyn equations



Inverse and forward dynamics

- Inverse dynamics

```
tauq = pin.rnea(rmodel,rdata,
                q, vq, aq)
```

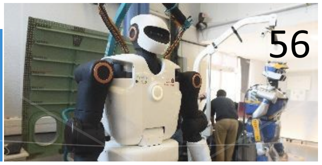
- Direct / forward dynamics

```
aq = pin.aba(rmodel,rdata,
             q, vq, tauq)
```

- Generalized inertia " mass " matrix

```
M = pin.crab(rmodel,rdata,q)
```

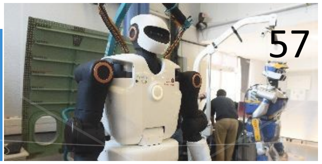
Give timings



Weighted gravity

$$c(q) = g(q)^T M(q)^{-1} g(q)$$

Compute c from rnea and aba

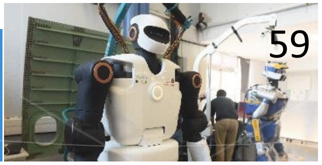


1.8 floating basis

Humanoids and quadrupeds

Free flyer joint

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



Integrate and differenciate

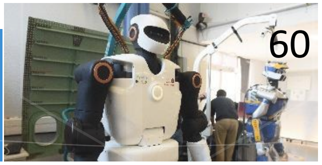
q_{next} = pin.integrate(q, v_q) $\in Q$

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

Δq= vq= pin.difference(q_1, q_2) $\in T_{q1}Q$

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

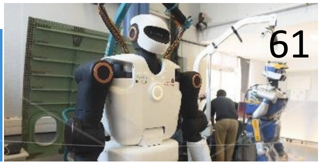
q = pin.normalize(rand(nq)) $\in Q$



Optimization with Q / TQ

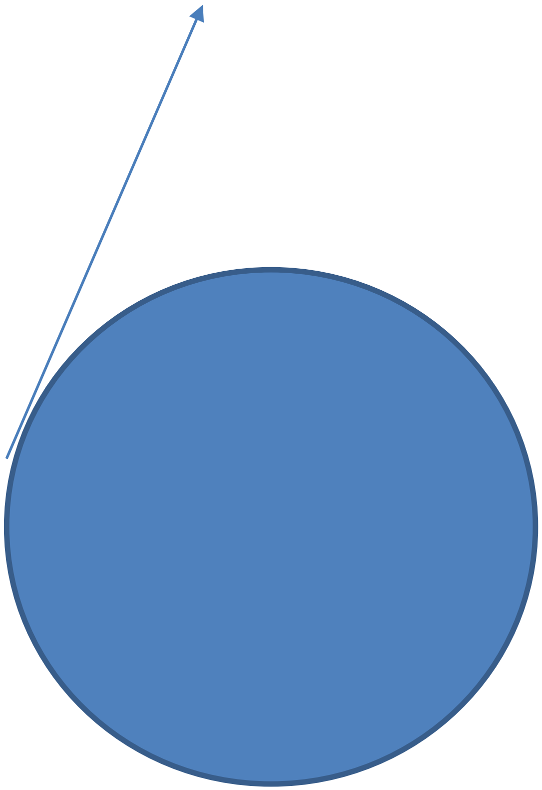
- $q = (x, y, z, \underline{q}, \ldots)$ with $\underline{q}$ unitary

- What is the result with a solver ?



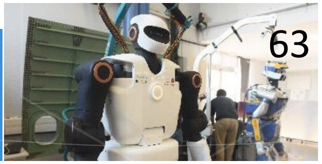
Solution 1: normalized

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



Solution 2: normalize

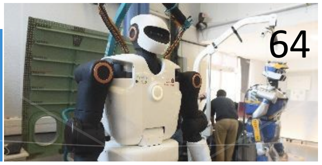
```
def cost(q):
    q = pin.normalize(rmodel,q)
    … # compute the cost
```

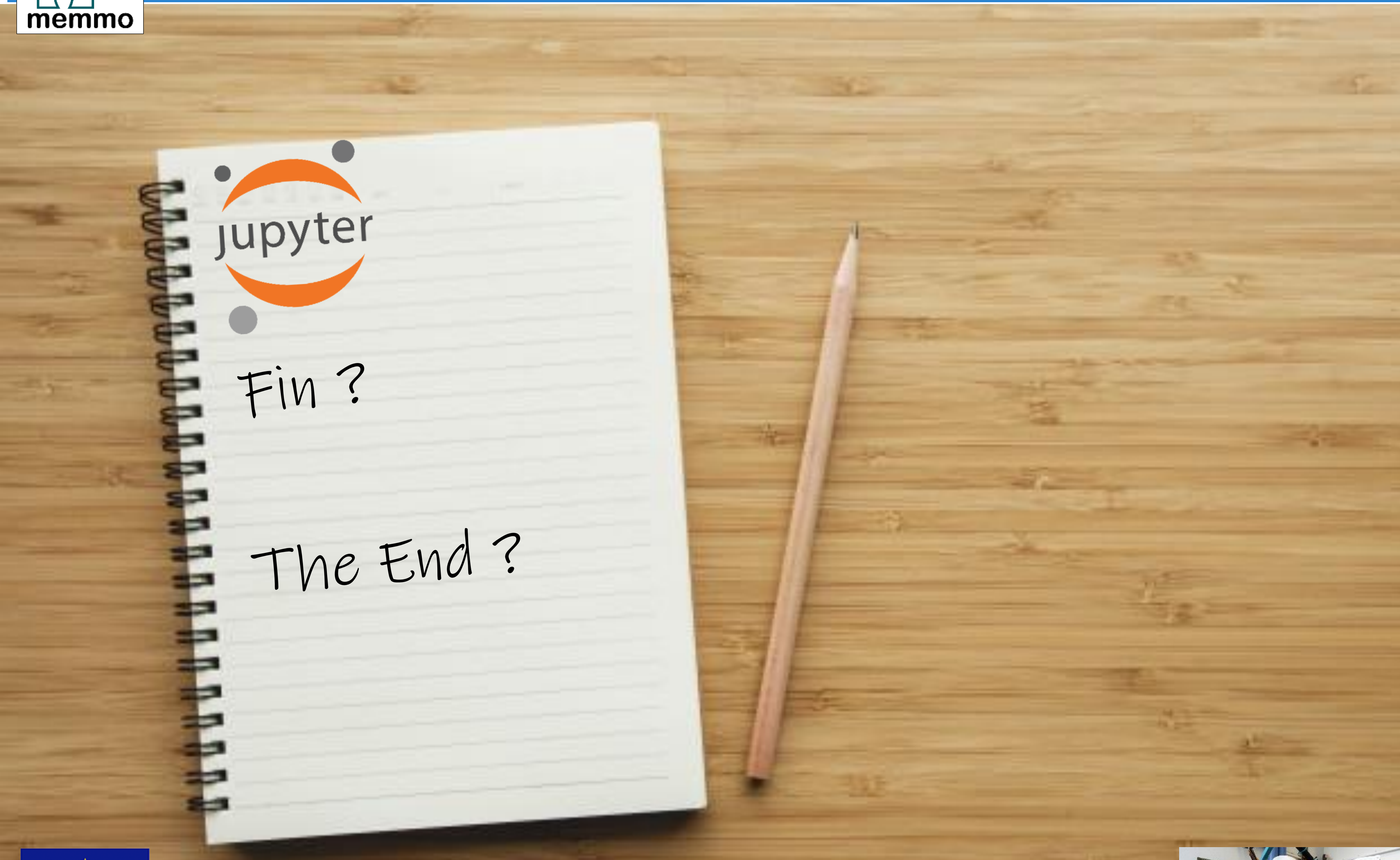



Solution 3: reparametrize

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

$$q = q_0 \oplus v_q$$




jupyter
Fin ?
The End ?

Exercice

- With humanoid Talos or quadruped Solo

- Choose contact location
  - 3d contacts for the quadruped or the humanoid hand
  - 6d contacts for the humanoid feet

- From a random configuration …
  - Optimize the 3d/6d costs + posture cost
  - Project the feet in contact