European Commission

Horizon 2020
European Union funding
for Research & Innovation

memmo

Pinocchio

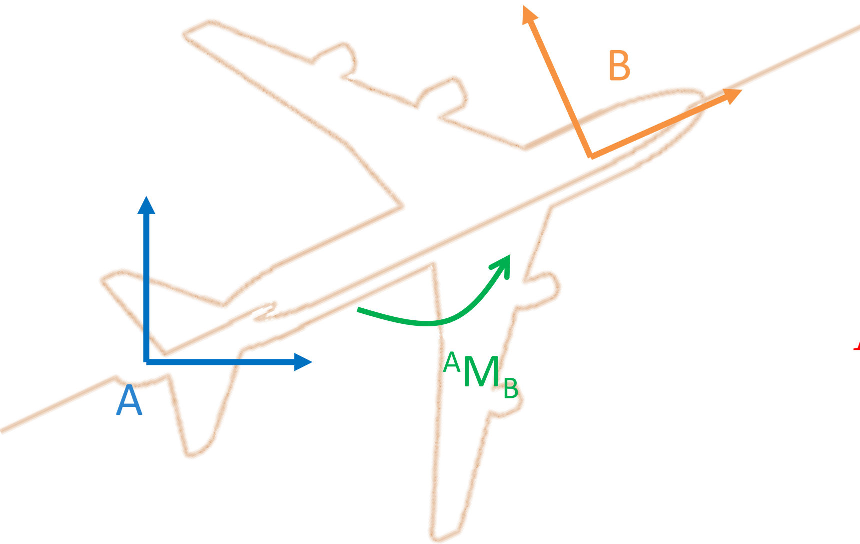
Rigid body derivatives

Nicolas Mansard
(CNRS)

2.1 Spatial velocities

Frame Jacobians

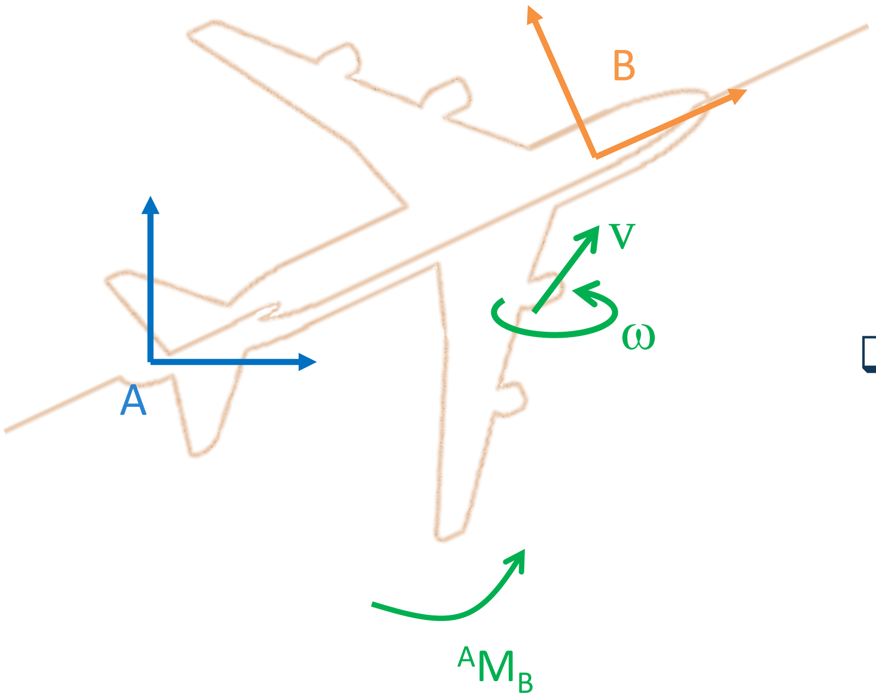
Rigid "spatial" velocities



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

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

Rigid "spatial" velocities

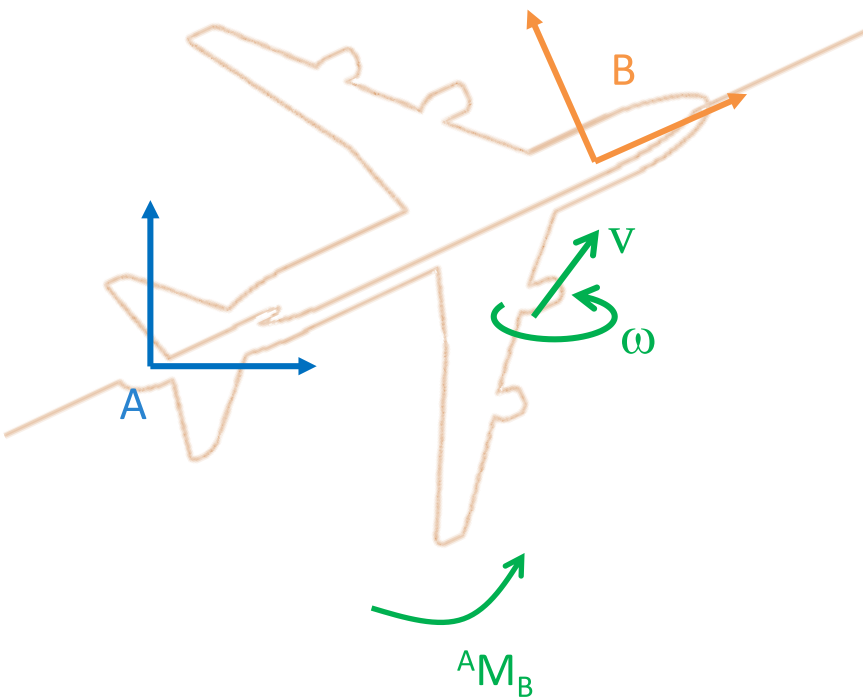


- $\nu=(v,\omega)$ represents the velocity of each point of the object

- Velocity vector field

$$\nu: p \rightarrow v_p$$

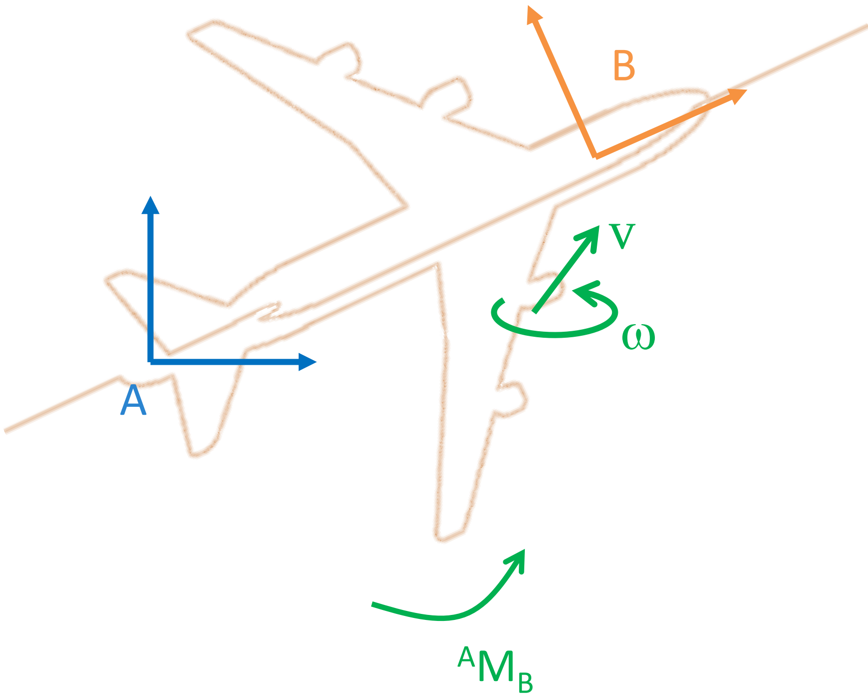
Rigid "spatial" velocities



- Choices to represent

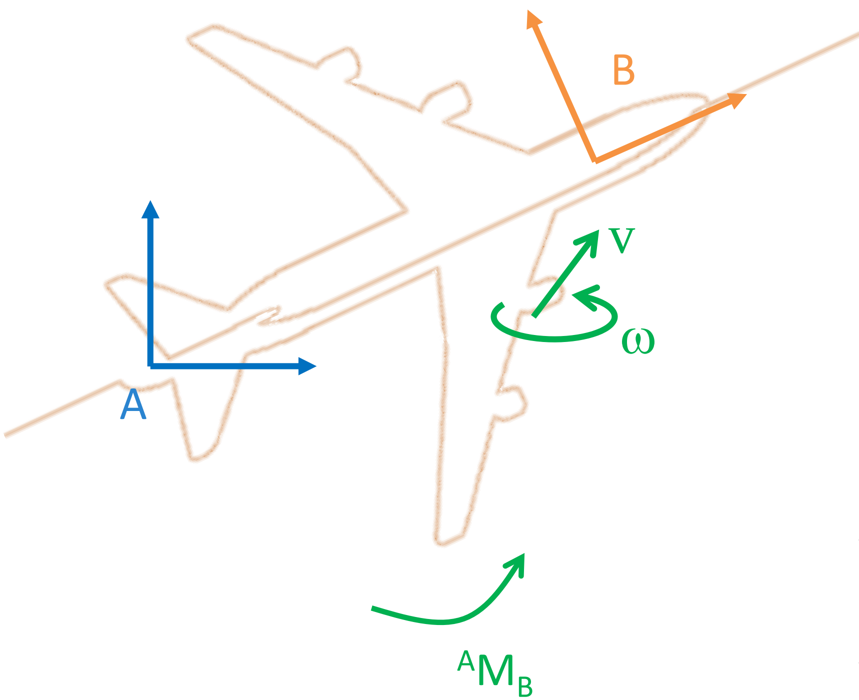
$$\mathbf{v} \quad , \quad \boldsymbol{\omega}$$

Rigid “spatial” velocities

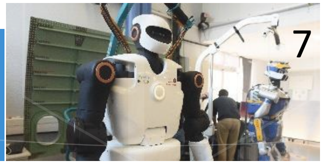


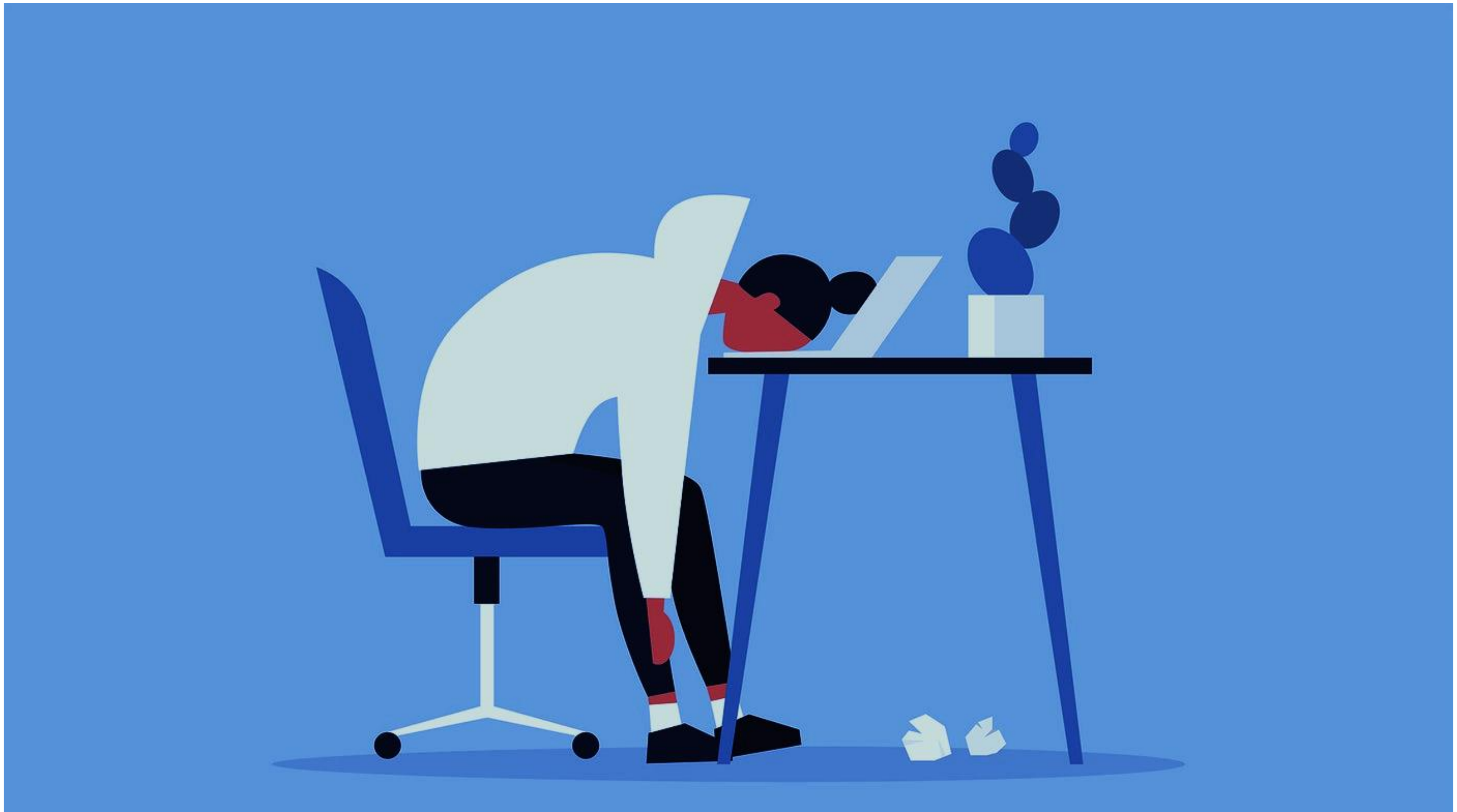
- SE3 action to change velocity expression frame

Rigid "spatial" velocities

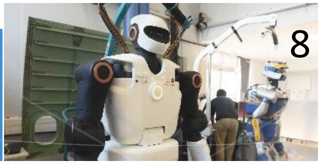



```
nu = pin.Motion.Random()
nu.linear  # 3-array
nu.linear # 3-array

aMb = pin.SE3.Random()
A_nu = aMb.act(B_nu)
A_nuvec = aMb.action @
                  B_nuvec
```

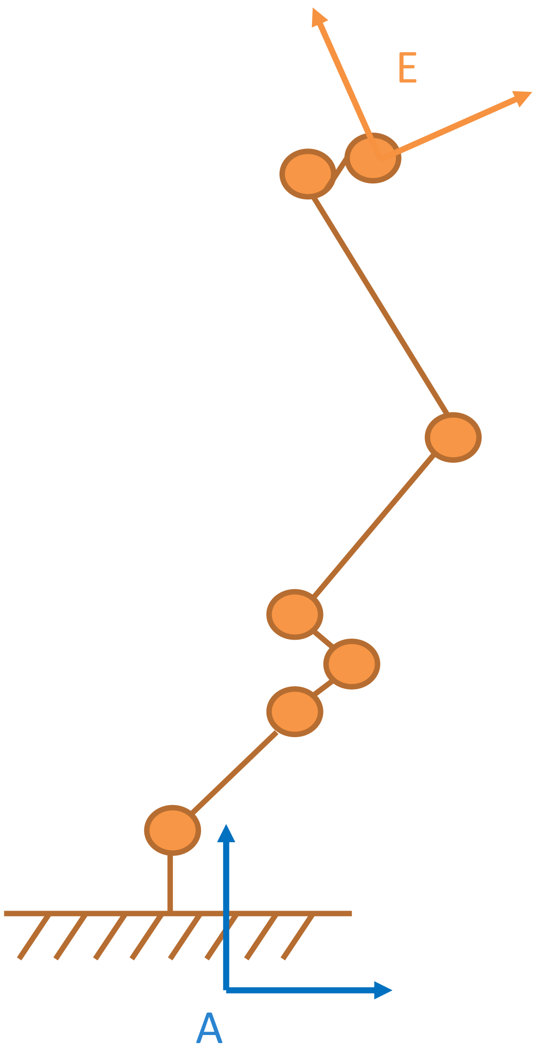


Ready for more?

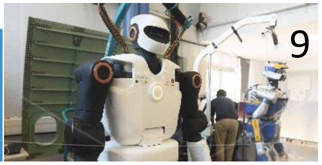


6D jacobians

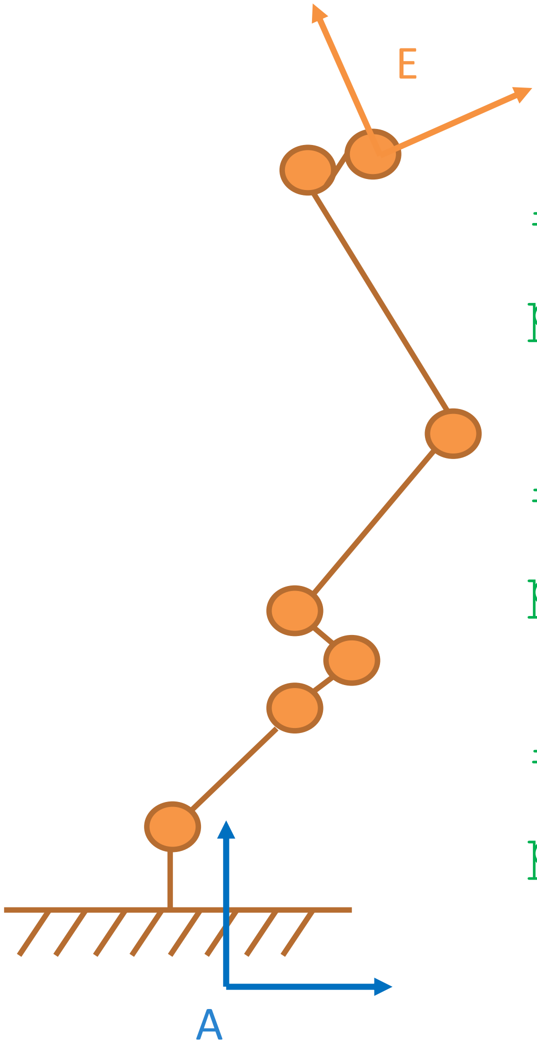
- $^{\heartsuit}\nu_{AE} = {}^{\heartsuit}J_E(q)\ \nu_q$



Explain columns of J

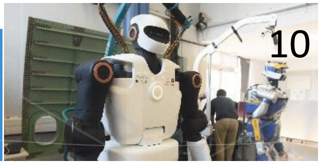


6D jacobians



- $^{\heartsuit}v_{AE} = {^{\heartsuit}J_E}(q)\, v_q$

```
# Precompute
pin.computeJointJacobians(rmodel,
                          rdata,q)
# Joint jacobian
pin.getJointJacobian(rmodel,rdata,
                     jointId,pin.LOCAL)
# Frame jacobian
pin.getFrameJacobian(rmodel,rdata,
                     frameId,pin.WORLD)
```

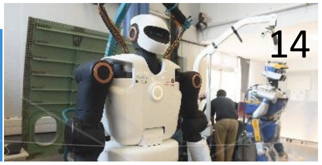


jupyter

2.2 Finite differences

You *MUST* numdiff

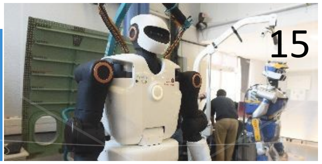
```
def myFunction(x):
      …
def myDerivative(x):
      …

x=random
assert(norm(myDerivative(x)-
             numdiff(myFunction,x))<1e-6)
```



You *MUST* numdiff

- When mathematical programming …
  … start with finite differences (numdiff)

- Easier to implement
- Less error prone
- Works often just as well, only very slow
- You *NEED* finite differences to check



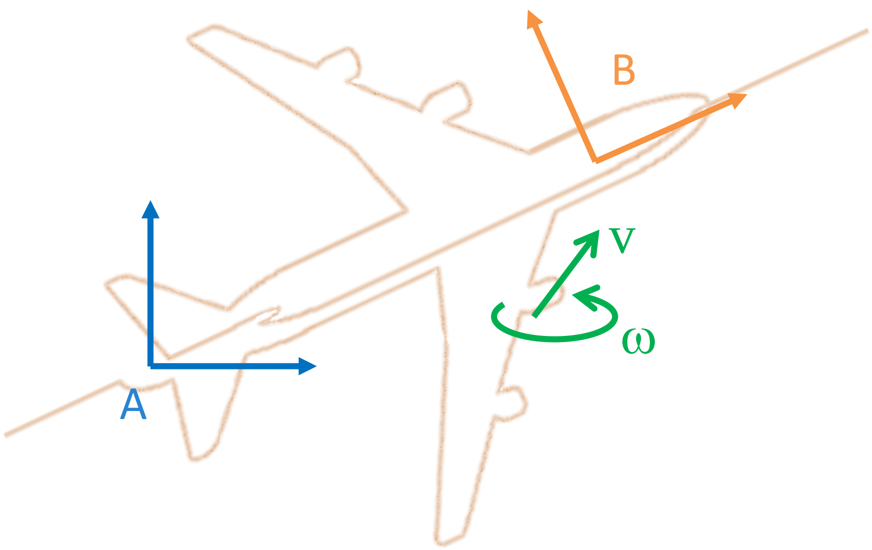
You *MUST* numdiff

2.3 Jacobian in 3d

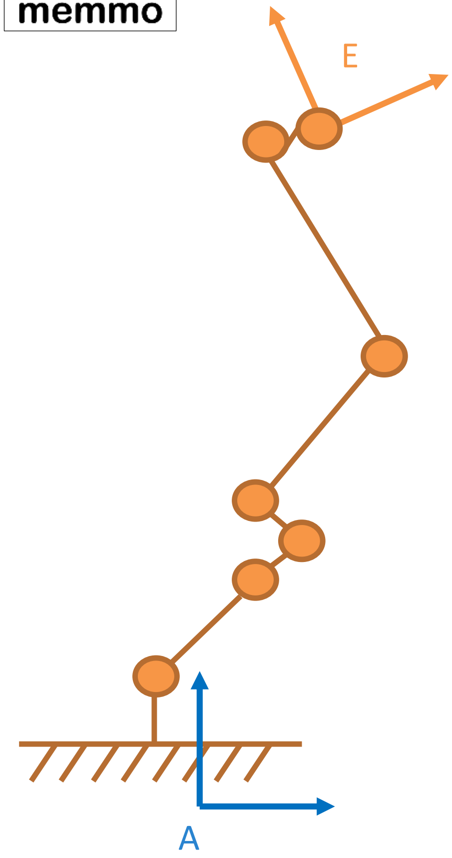
Local world-aligned

- Interpretation of

$$\mathbf{v} \;,\; \boldsymbol{\omega}$$



Linear rows of the jacobian



`pin.getFrameJacobian(…,pin.LOCAL_WORLD_ALIGNED)[3:,:]`

Explain rows of J

2.4 Posture derivatives

Posture cost and derivative

- First implementation

$$c(q) = \| q - qref \|^2$$

$$\nabla c = q - qref$$

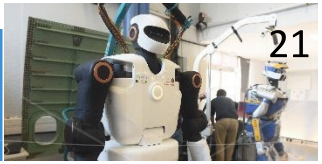
- When q lives in a Lie group

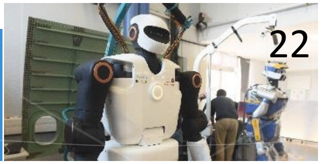
$$c(q) = r(q)^T r(q)$$

$r(q) =$ `pin.difference(rmodel,q,qref)`

$$\nabla c = 2 \nabla r^T r$$

$\nabla r =$ `pin.dDifference(rmodel,q,qref)[0]`





memmo

jupyter

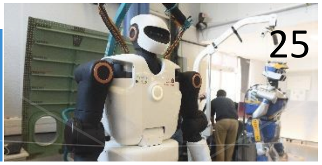
2.5 Derivatives of the gravity

- Gravity cost

$$c(q) = g(q)^T g(q)$$

- Derivatives

```
pin.computeGeneralizedGravityDerivatives
     (rmodel,rdata,q)
```



2.6 Derivatives of the dynamics

Whole body dynamics

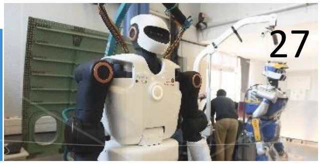
$$M(q)\, a_q + b(q,v_q) = \tau_q$$

- Inverse dynamics

$$\tau_q = \mathrm{rnea}(q, v_q, a_q) = M(q)\, a_q + b(q,v_q)$$

- Direct dynamics

$$a_q = \mathrm{aba}(q, v_q, a_q) = M(q)^{-1} (\tau_q - b(q,v_q))$$



RNEA derivatives

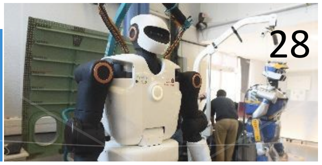
- Inverse dynamics

$$\tau_q = \mathrm{rnea}(q, v_q, a_q) = M(q)\, a_q + b(q, v_q)$$

- Derivatives

```
pin.computeRNEADerivatives
               (rmodel,rdata,q,vq,aq)
```

- $\partial\tau_q / \partial q$ `rdata.dtau_dq`
- $\partial\tau_q / \partial v_q$ `rdata.dtau_dv`
- $\partial\tau_q / \partial a_q$???



ABA derivatives

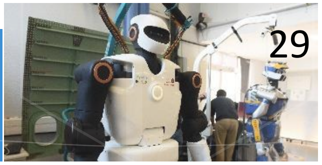
- Direct dynamics

$$a_q = \mathrm{aba}(q, v_q, a_q) = M(q)^{-1} (\tau_q - b(q,v_q))$$

- Derivatives

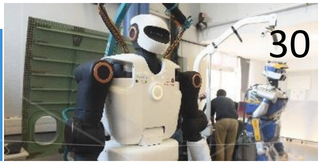
```
pin.computeABADerivatives
            (rmodel,rdata,q,vq,aq)
```

- $\partial a_q / \partial q$ `rdata.ddq_dq`
- $\partial a_q / \partial v_q$ `rdata.ddq_dv`
- $\partial a_q / \partial \tau_q$ `rdata.Minv`



Weighted gravity

- Cost function: $c(q)=g(q)^T M^{-1} g(q)$
- Derivatives



2.7 Derivations in Lie groups

Tangent applications and coefficient-wise jacobians

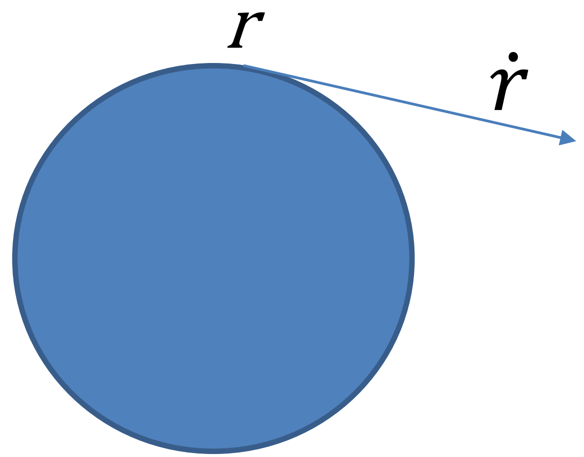
Tangent space

- Representation of SO(3)

$$r \in SO(3) \cong R \in \mathbb{R}^{3x3}$$
$$\cong q \in \mathbb{H} \cong \mathbb{R}^{4x4}$$

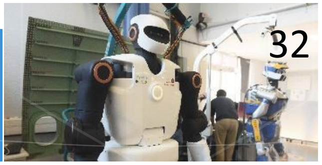
- Representation of SO(3) derivatives

$$\dot{r} \in \mathfrak{so}(3) = \mathbb{R}^{3}$$



- Function of a rotation

$$f(r) = f(R) = f(q)$$



Tangent application

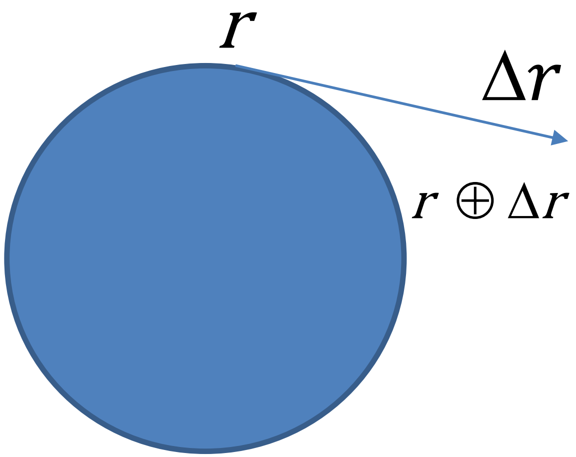
- Thinking "finite differences"

$$r \in SO(3)\ ,\ \Delta r \in \mathfrak{so}(3)$$

$$f(r \oplus \Delta r) - f(r) \approx F_r \Delta r$$

- Tangent application $\oplus$

$$\frac{d f(r \oplus \Delta r)}{d\,\Delta r} := T_r f$$



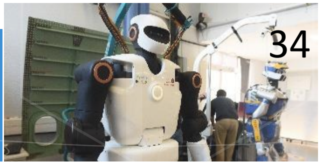
Coefficient-wise derivative

- If we choose a specific representation…

f = f(q)

- What is df/dq_0?
  - What is dq_0?
  - $\frac{\partial f}{\partial q_0} \approx \frac{f(q+[\varepsilon,0,0,0])+f(q)}{\varepsilon}$

- Let's call this quantiy "coefficient-wise" derivative



Tangent vs coefficient-wise

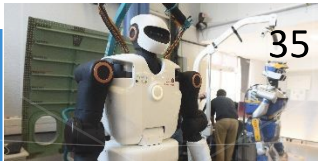
- Both matrices are linked by

$$\frac{df(q\oplus\Delta q)}{d\,\Delta q} = \frac{df}{d\,q}\,\frac{d(q\oplus\Delta q)}{d\,\Delta q}$$

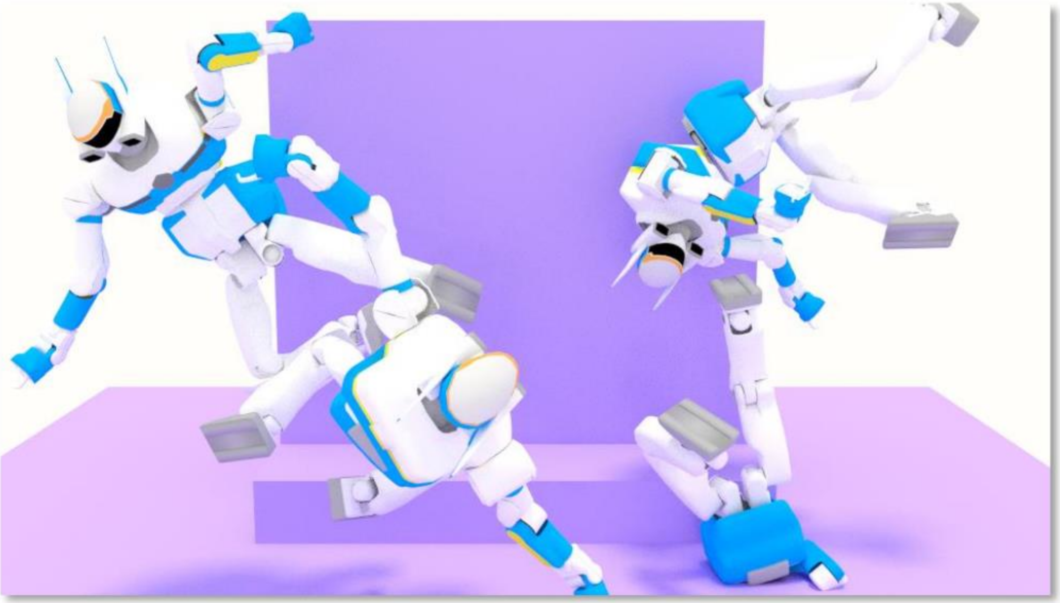
Tangent application

Coefficient-wise derivatives of f

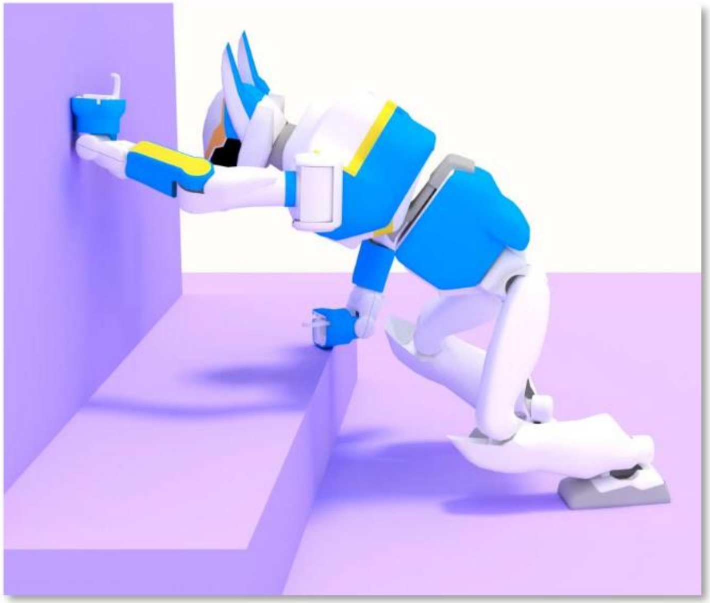
Coefficient-wise derivatives of `pin.integrate`



Posture generator

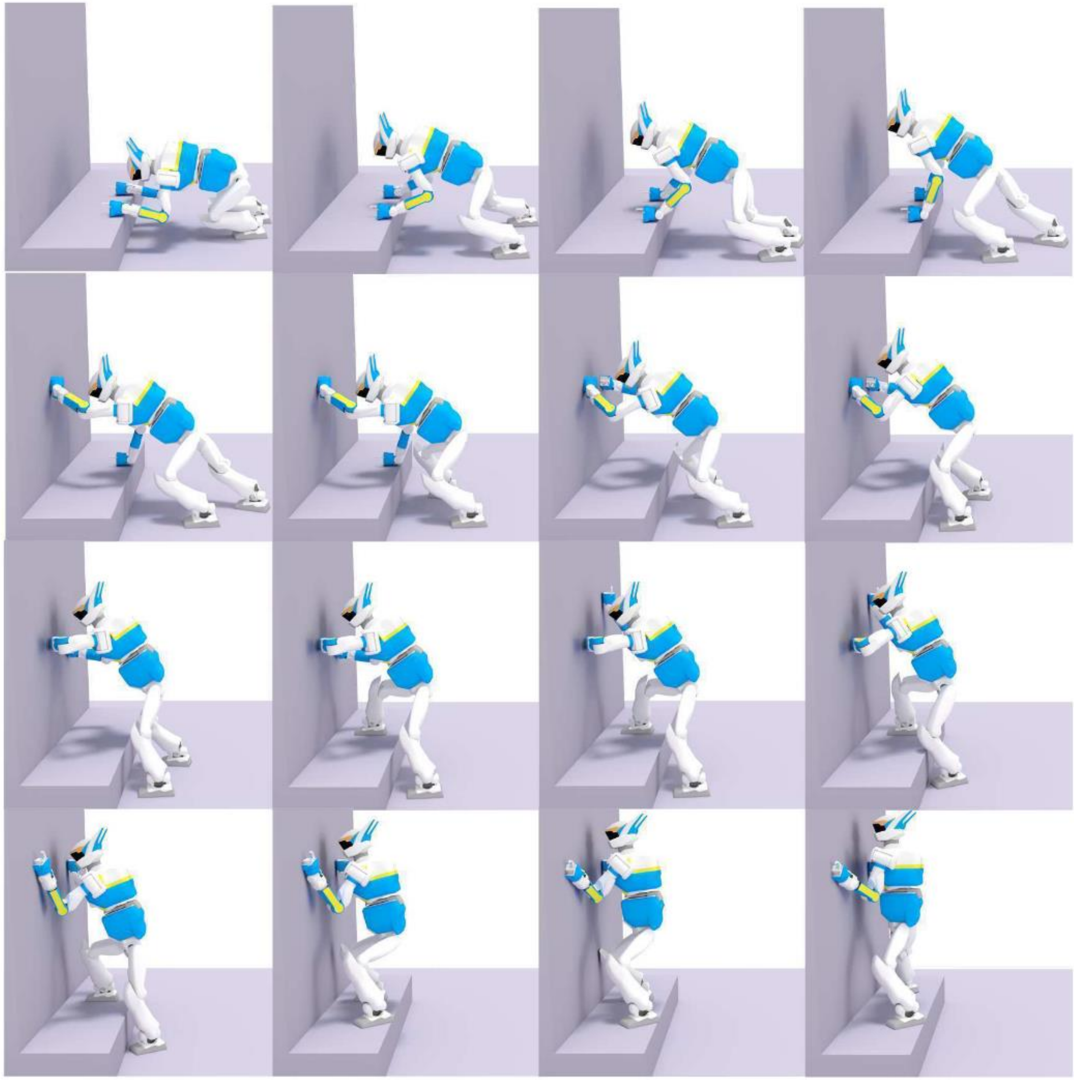


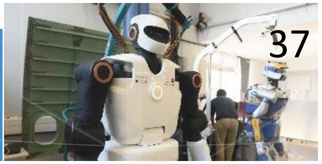
Sample a configuration …



… project it into contact

Contact planner





Python prototype

```python
    def searchContactPosture(self, numberOfContacts, qguess=None, ntrial=100):
        """
        Search (randomly) a contact configuration with a given number of contacts.
        - number of contacts: between 1 and len(self.contactCandidates).
        - qguess is a configuration, of dim rmodel.nq. If None, a random configuration is first
        sampled.
        - ntrial: number of random trial.

        If successfull, set self.success to True, and store the contact posture found in self.qcontact.
        If not sucessfull, set self.success to False.
        Returns self.success.
        """
        assert 0 < numberOfContacts < len(self.contactCandidates)
        if qguess is None:
            qguess = pin.randomConfiguration(self.rmodel)

        for itrial in range(ntrial):
            limbs = random.sample(list(self.contactCandidates.values()), numberOfContacts)
            stones = random.sample(self.terrain, numberOfContacts)

            constraints = [Constraint(self.rmodel, limb.frameIndex, stone, limb.contactType)
                           for limb, stone in zip(limbs, stones)]

            self.postureGenerator.run(qguess, constraints)
            if self.postureGenerator.sucess:
                # We found a candidate posture, let' s check that it is acceptable.
                qcandidate = self.postureGenerator.qopt.copy()

                # computeCollisions check collision among the collision pairs.
                collision = pin.computeCollisions(self.rmodel, self.rdata,
                                                  self.collision_model, self.collision_data, qcandidate, True)

                if not collision:
                    self.qcontact = qcandidate
                    self.success = True
                    return True

        self.success = False
        return False
```