Overview

- ❑ Standard Methods for **Inverse Kinematics** (IK)
  - Theory
  - *Tutorial on IK*
  - *Assignment on IK*
- ❑ Operational Space Controllers on iCub
  - Cartesian Control (theory)
  - Gaze Control (theory)
  - *Cartesian Control (API + tutorial)*
  - *Gaze Control (API + tutorial)*
  - *Assignment on Operational Space Controllers*
  - ***Experiment with iCub***

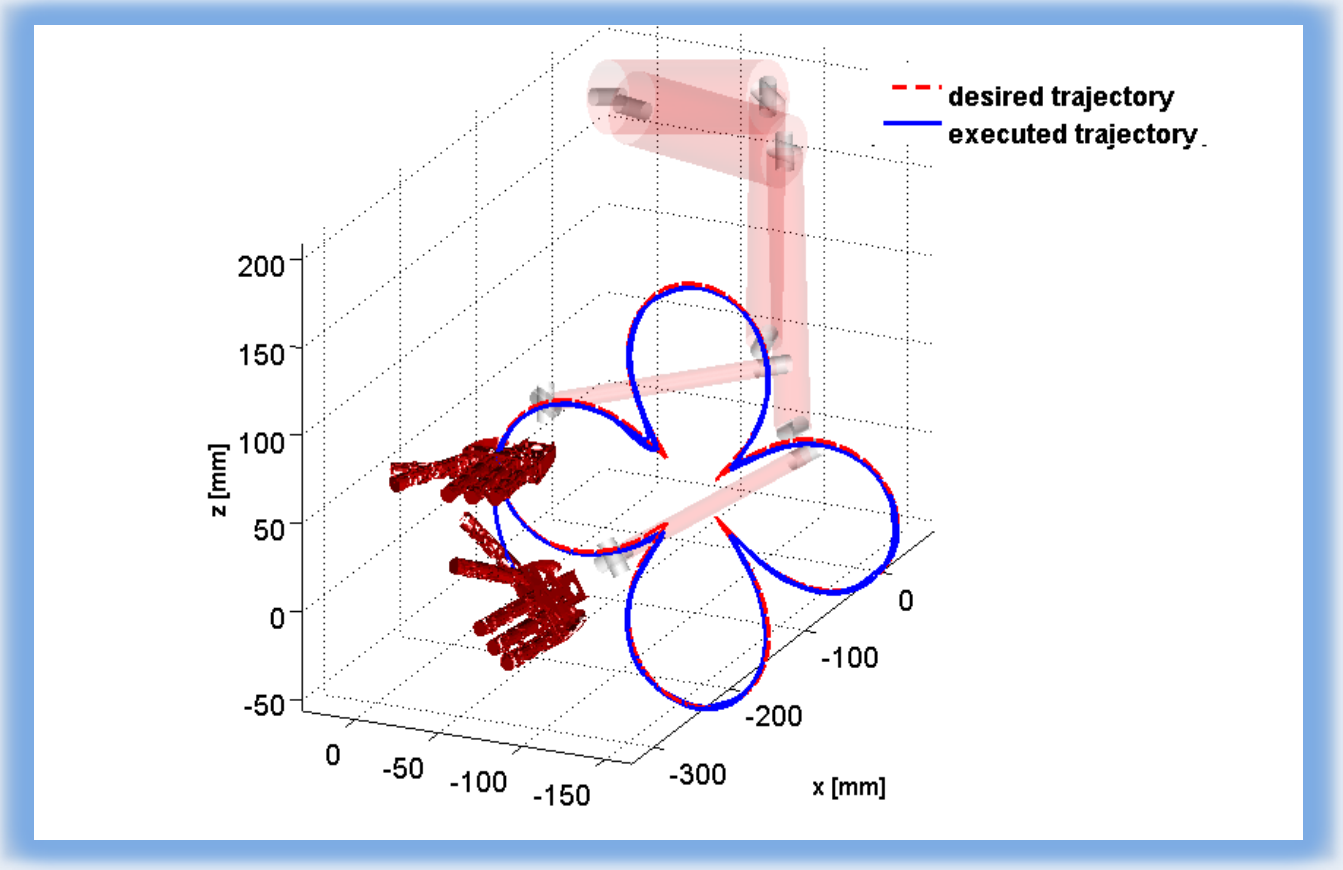
Introduction (1/2)

Study of properties of motion (position, velocity, acceleration) without considering body inertias and internal/external forces

The Problem

$$\begin{cases} \mathbf{x} = \mathbf{f}(\mathbf{q}) \\ \mathbf{q} \in \mathbb{R}^n \\ \mathbf{x} \in \mathbb{R}^6 \end{cases}$$

$$\mathbf{q} \stackrel{?}{=} \mathbf{f}^{-1}(\mathbf{x})$$



Introduction (2/2)

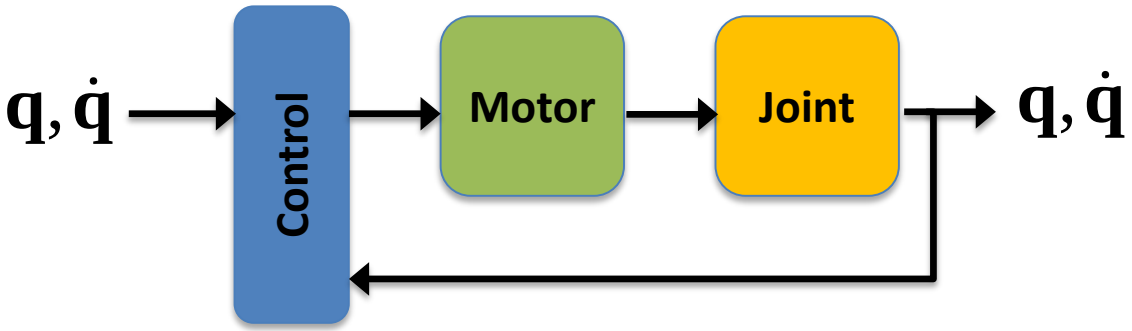
Dynamics – forces, torques, inertias, energy, contact with environment

$$\mathbf{B}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{F}_v\dot{\mathbf{q}} + \mathbf{F}_s\,\mathrm{sgn}(\dot{\mathbf{q}}) + \mathbf{g}(\mathbf{q}) = \boldsymbol{\tau} - \mathbf{J}^T(\mathbf{q})\mathbf{h}_e$$

VS.

Kinematics – pure motion imposed to the manipulator's joints

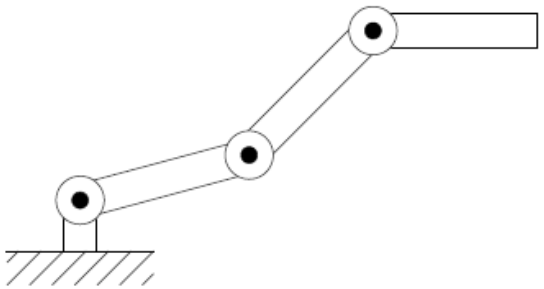
$$\begin{cases} \dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{q}} \\ \mathbf{J} = \partial\mathbf{f}/\partial\mathbf{q} \end{cases}$$

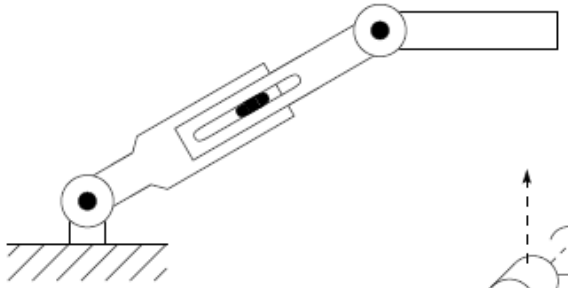


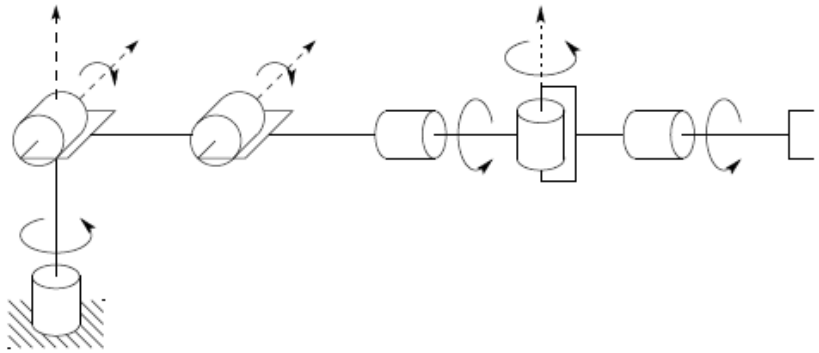
Direct Kinematics (1/3)

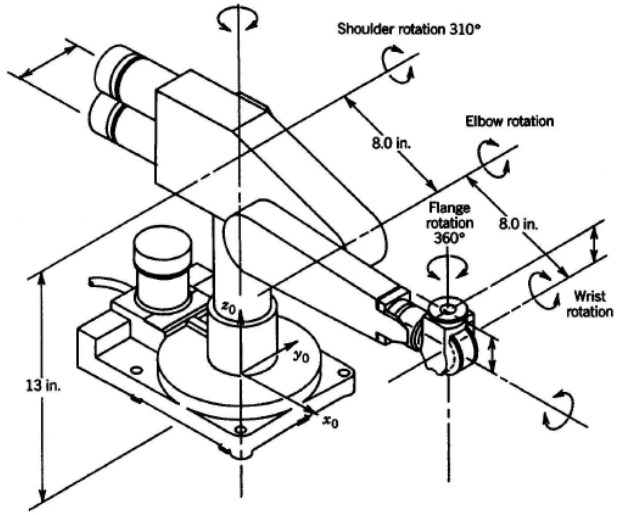
Definitions

- A **serial manipulator** consists of a series of rigid bodies (**links**) connected by means of kinematic pairs or **joints**, forming a **kinematic chain**
- Joints can be: **revolute**, **prismatic**
- Kinematic chains are: **open**, **close**
- Open chains are constrained to a **base** and terminate with the **end-effector** whose pose $\boldsymbol{x}$ is defined in the **Cartesian/Operational Space**
- **DOFs** uniquely determine the posture of the manipulator
- DOFs are associated to joint articulations and constitute the **joint variables** $\boldsymbol{q}$ defined in the **Joint/Configuration Space**
- **Direct Kinematics** aims to compute $\boldsymbol{x}$ as function of $\boldsymbol{q}$









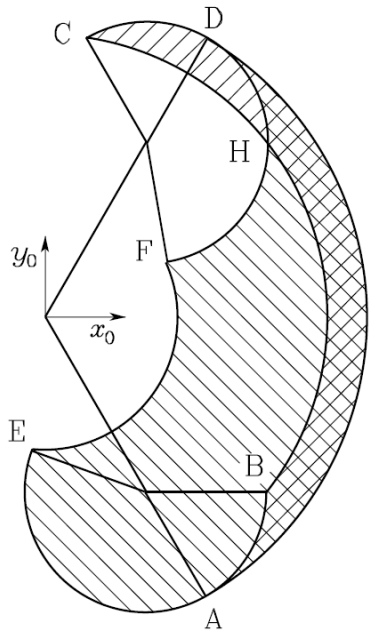
Direct Kinematics (2/3)

Definitions

$$\mathbf{x} = \begin{bmatrix} \mathbf{p} \\ \phi \end{bmatrix} = \mathbf{f}(\mathbf{q})$$

$$\mathbf{x} \in \mathbb{R}^m$$

$$\mathbf{q} \in \mathbb{R}^n$$

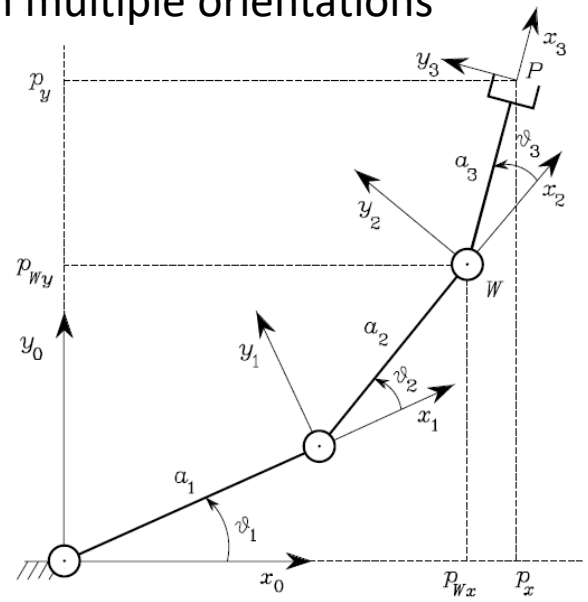


- **Operational Space** is defined by m
- **Joint Space** is defined by n

Manipulator Workspace: region spanned by the end-effector when all the manipulator joints execute all possible motion.
Dependent on manipulator geometry + joint limits

- **Reachable**: contains points reached with at least one orientation
- **Dexterous**: contains points reached with multiple orientations

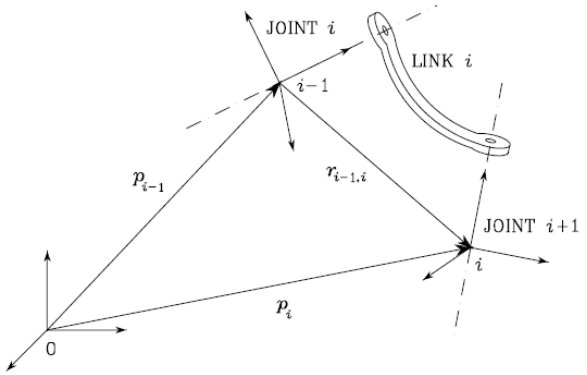
Redundancy: $m < n$
concept relative to the task assigned

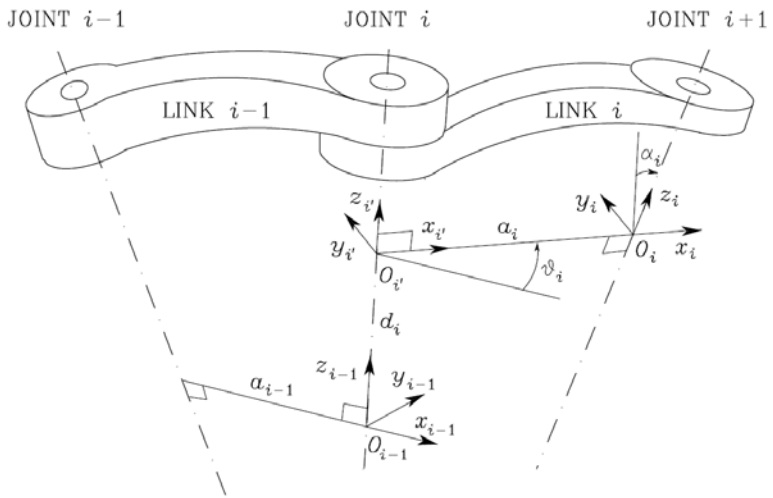


Direct Kinematics (3/3)

Denavit-Hartenberg

$$\mathbf{x} = \mathbf{f}(\mathbf{q}) \quad \mathbf{q} \in \mathbb{R}^n, \ \mathbf{x} \in \mathbb{R}^m$$





DH Parameters

- α_i: twist
- a_i: link length
- d_i: joint offset
- θ_i: joint angle

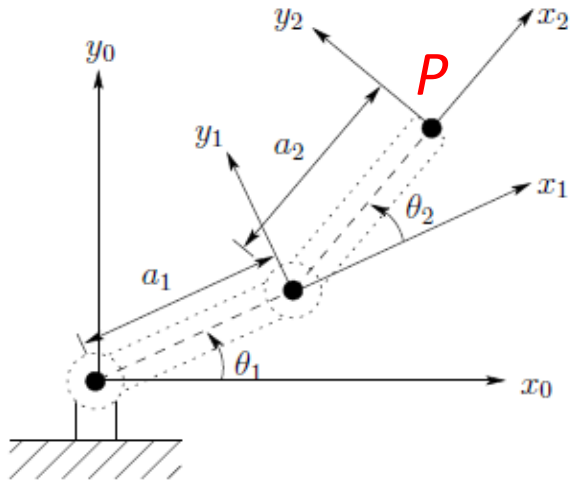
Link	a_i	α_i	d_i	θ_i
1	a_1	0	0	θ_1^*
2	a_2	0	0	θ_2^*

* variable

$$A_1 = \begin{bmatrix} c_1 & -s_1 & 0 & a_1c_1 \\ s_1 & c_1 & 0 & a_1s_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} c_2 & -s_2 & 0 & a_2c_2 \\ s_2 & c_2 & 0 & a_2s_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_2^0 = A_1A_2 = \begin{bmatrix} c_{12} & -s_{12} & 0 & a_1c_1 + a_2c_{12} \\ s_{12} & c_{12} & 0 & a_1s_1 + a_2s_{12} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



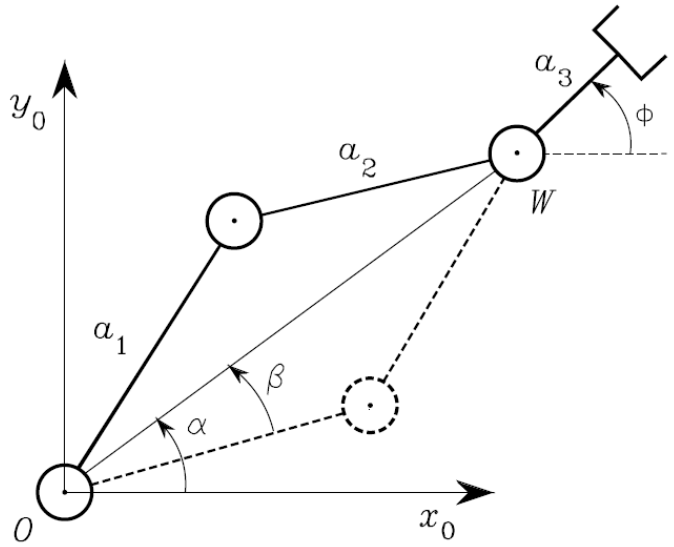
$$P \equiv \begin{cases} x = a_1c_1 + a_2c_{12} \\ y = a_1s_1 + a_2s_{12} \\ \phi = \theta_1 + \theta_2 \end{cases}$$

Inverse Kinematics (1/11)

We know **x,** we cannot control directly the motors, we have to solve the **Inverse Kinematics (IK) problem** beforehand.

$$\mathbf{q} \stackrel{?}{=} \mathbf{f}^{-1}(\mathbf{x})$$

- Nonlinear equations ➔ often no closed-form solutions
- Multiple solutions may exist
- Infinite solutions may exist (redundancy)
- There might be no admissible solutions



Inverse Kinematics (2/11)

We know **x,** we cannot control directly the motors, we have to solve the **Inverse Kinematics (IK) problem** beforehand.

$$\mathbf{q} \stackrel{?}{=} \mathbf{f}^{-1}(\mathbf{x})$$

Iterative Methods

Jacobian Transpose $\dot{\mathbf{q}} = \mathbf{J}^T \mathbf{K} \mathbf{e}, \quad \mathbf{e} = \mathbf{x}_d - \mathbf{x}_e$

Jacobian Pseudoinverse $\dot{\mathbf{q}} = \mathbf{J}^\dagger \mathbf{K} \mathbf{e} + \left(\mathbf{I} - \mathbf{J}^\dagger \mathbf{J}\right) \dot{\mathbf{q}}_0, \quad \mathbf{J}^\dagger = \mathbf{J}^T \left(\mathbf{J}\mathbf{J}^T\right)^{-1}$

Damped Least Squares $\dot{\mathbf{q}} = \mathbf{J}^* \mathbf{K} \mathbf{e}, \quad \mathbf{J}^* = \mathbf{J}^T \left(\mathbf{J}\mathbf{J}^T + k^2 \mathbf{I}\right)^{-1}$

Inverse Kinematics (3/11)

Analytical Jacobian

$$\mathbf{x} = \mathbf{f}(\mathbf{q}) \overset{d/dt}{\Longrightarrow} \dot{\mathbf{x}} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}$$

$$\mathbf{J}(\mathbf{q}) = \begin{bmatrix} \frac{\partial f_1}{\partial q_1} & \cdots & \frac{\partial f_1}{\partial q_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial q_1} & \cdots & \frac{\partial f_m}{\partial q_m} \end{bmatrix}$$

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{\mathbf{p}} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_P \\ \mathbf{J}_\phi \end{bmatrix} \dot{\mathbf{q}}$$

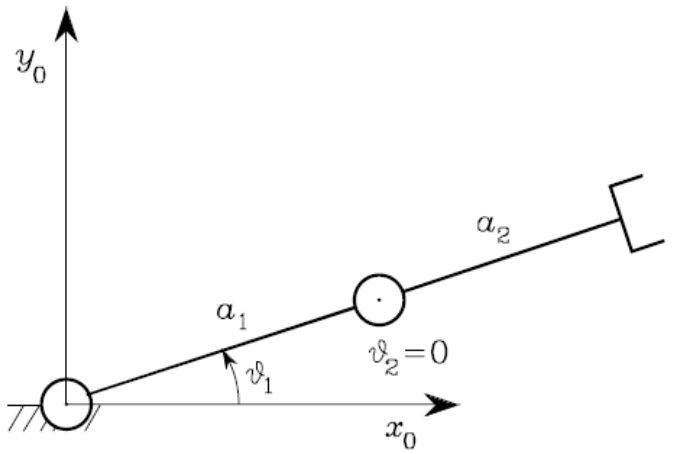
Kinematic Singularities: configurations where the Jacobian degenerates becoming *rank-deficient*

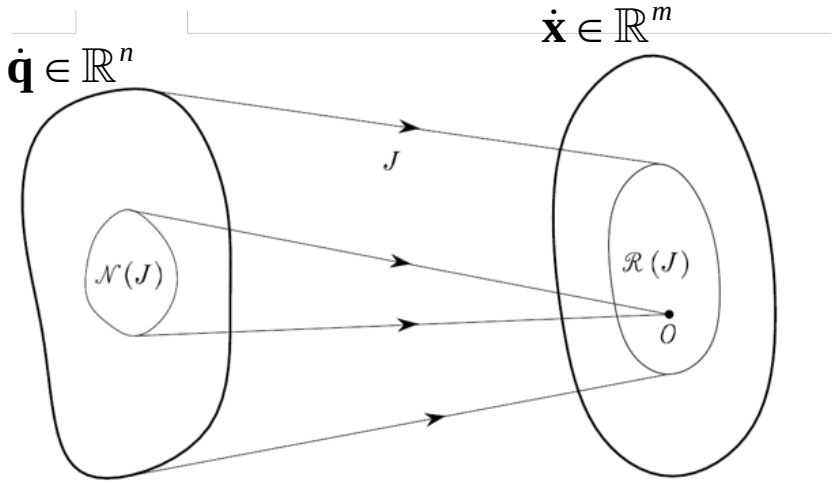
At singularity:

- Structure mobility is reduced (impossible to impose arbitrary motion to the end-effector)
- Infinite solutions to IK may exist

In the singularity's neighborhood:

- Small velocities in the operational space bring about large velocities in the joint space





Inverse Kinematics (4/11)

Jacobian Transpose

We aim to minimize the Cartesian error: $g = \frac{1}{2}\|\mathbf{e}\|^2 = \frac{1}{2}\|\mathbf{x}_d - \mathbf{f}(\mathbf{q})\|^2$

Compute the gradient:

$$\nabla_{\mathbf{q}} g = \frac{1}{2}\nabla_{\mathbf{q}} \langle (\mathbf{x}_d - \mathbf{f}), (\mathbf{x}_d - \mathbf{f}) \rangle = \frac{1}{2} \cdot \left(-2 \langle \nabla_{\mathbf{q}} \mathbf{f}, (\mathbf{x}_d - \mathbf{f}) \rangle \right) =$$

$$= -\langle \mathbf{J}, (\mathbf{x}_d - \mathbf{f}) \rangle = -\mathbf{J}^T (\mathbf{x}_d - \mathbf{f}) = -\mathbf{J}^T \mathbf{e}$$

Gradient descent method for system of nonlinear equations:

$$\dot{\mathbf{q}} = \mathbf{K} \cdot \left(-\nabla_{\mathbf{q}} g \right) = \mathbf{J}^T \mathbf{K} \mathbf{e}$$

we only employ *direct kinematics functions*!

Inverse Kinematics (5/11)

Jacobian Transpose

Asymptotic stability with Lyapunov direct method

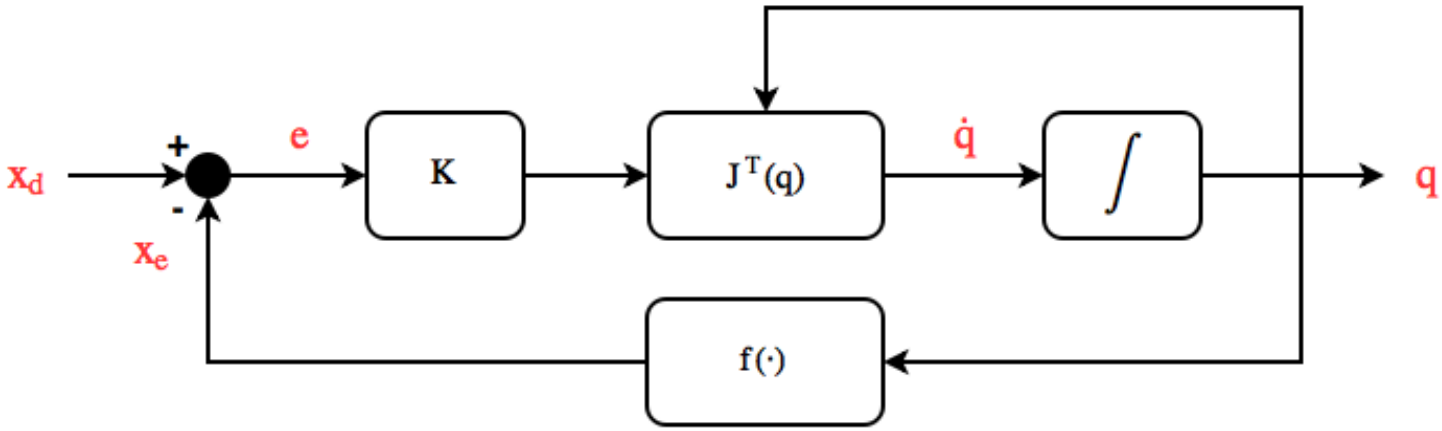
$$\mathbf{e} = \mathbf{x}_d - \mathbf{f}(\mathbf{q})$$

$$\dot{\mathbf{e}} = \dot{\mathbf{x}}_d - \mathbf{J}\dot{\mathbf{q}}$$

$$V(\mathbf{e}) = \frac{1}{2}\mathbf{e}^T\mathbf{K}\mathbf{e}$$

✓ 1. $V(\mathbf{e}) > 0 \;\; \forall \mathbf{e} \neq 0, \; V(\mathbf{0}) = 0$

? 2. $\dot{V}(\mathbf{e}) < 0 \;\; \forall \mathbf{e}$



$$\dot{V} = \mathbf{e}^T\mathbf{K}\dot{\mathbf{x}}_d - \mathbf{e}^T\mathbf{K}\mathbf{J}\dot{\mathbf{q}}$$

$$\dot{\mathbf{q}} = \mathbf{J}^T\mathbf{K}\mathbf{e} \;\Rightarrow\; \dot{V} = \mathbf{e}^T\mathbf{K}\dot{\mathbf{x}}_d - \mathbf{e}^T\mathbf{K}\mathbf{J}\mathbf{J}^T\mathbf{K}\mathbf{e}$$

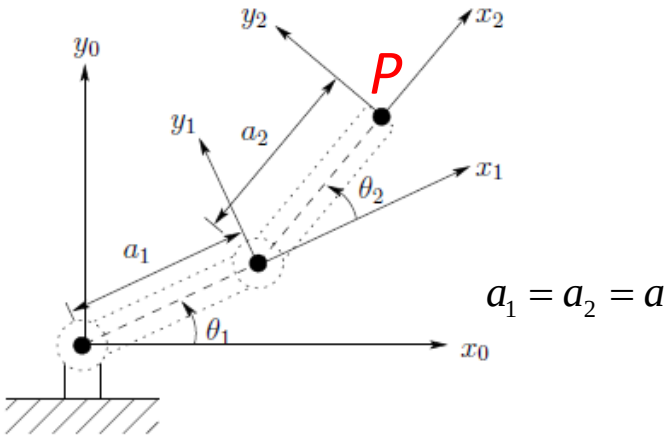
$$\dot{\mathbf{x}}_d = 0 \;\Rightarrow\; \dot{V} = -\mathbf{e}^T\mathbf{K}\mathbf{J}\mathbf{J}^T\mathbf{K}\mathbf{e} < 0 \;\Leftrightarrow\; \mathcal{N}(\mathbf{J}^T) = 0$$

$\dot{V} < 0$ only when $\mathbf{J}^T$ is full rank

Inverse Kinematics (6/11)

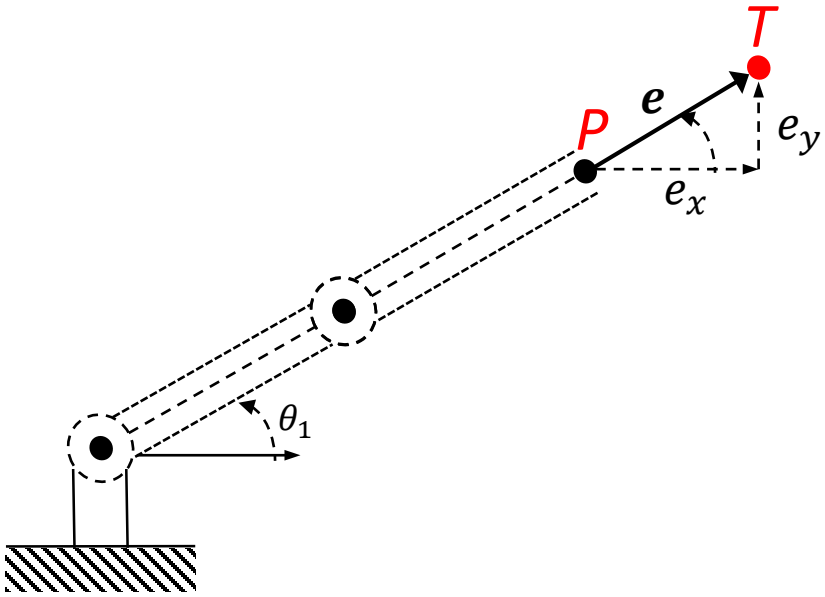
Jacobian Transpose

Null space of the
Jacobian Transpose $\mathcal{N}\left(\mathbf{J}^T\right)$:



$$\mathbf{J}^T\mathbf{e} = 0 \Rightarrow a \cdot \begin{pmatrix} -s_1 - s_{12} & c_1 + c_{12} \\ -s_{12} & c_{12} \end{pmatrix} \begin{pmatrix} e_x \\ e_y \end{pmatrix} = 0 \Rightarrow$$

$$\Rightarrow \begin{cases} \tan\theta_1 = \dfrac{e_y}{e_x} \\ \tan\left(\theta_1 + \theta_2\right) = \dfrac{e_y}{e_x} \end{cases} \Rightarrow \begin{cases} \tan\theta_1 = \dfrac{e_y}{e_x} \\ \theta_2 = 0 \end{cases}$$



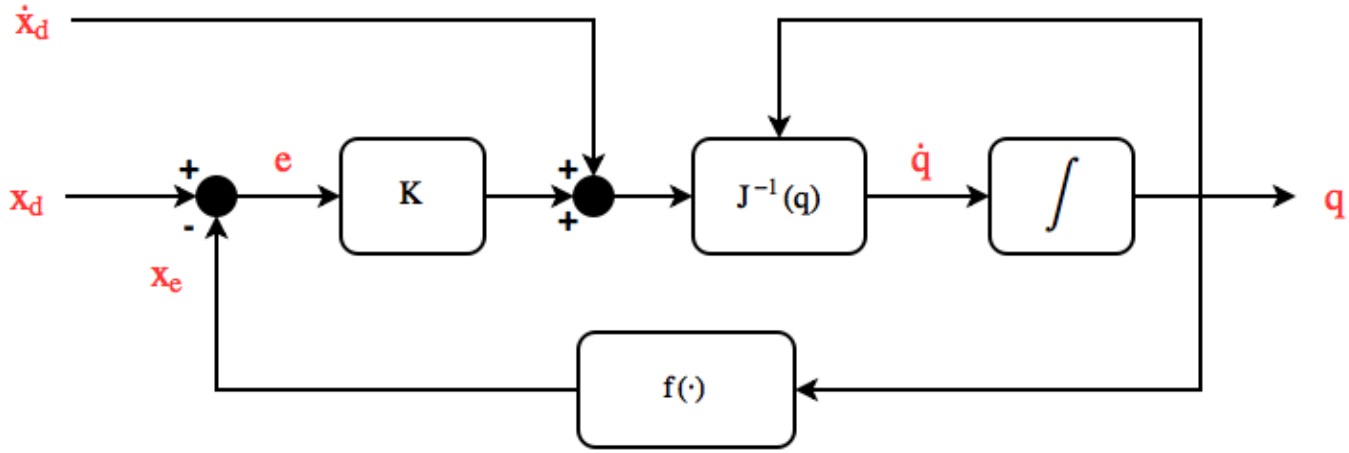
Inverse Kinematics (7/11)

Jacobian (Pseudo-)inverse

$$\mathbf{x} = \mathbf{f}(\mathbf{q}) \Rightarrow \dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{q}}$$

? $\dot{\mathbf{q}} = \mathbf{J}^{-1}\dot{\mathbf{x}}$

non-redundant chain $n = m$



Asymptotic stability with LTI system analysis

$$\mathbf{e} = \mathbf{x}_d - \mathbf{f}(\mathbf{q})$$

$$\dot{\mathbf{e}} = \dot{\mathbf{x}}_d - \mathbf{J}\dot{\mathbf{q}}$$

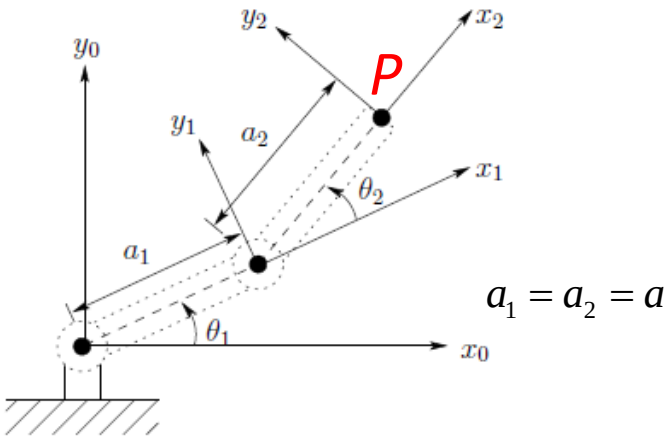
$$\dot{\mathbf{q}} = \mathbf{J}^{-1}(\dot{\mathbf{x}}_d + \mathbf{K}\mathbf{e}) \Rightarrow \dot{\mathbf{e}} + \mathbf{K}\mathbf{e} = 0$$ ✓

$$\dot{\mathbf{x}}_d = 0, \text{ no FF} \Rightarrow \dot{\mathbf{q}} = \mathbf{J}^{-1}\mathbf{K}\mathbf{e}$$

Newton method for system of nonlinear equations

Inverse Kinematics (8/11)

Jacobian (Pseudo-)inverse

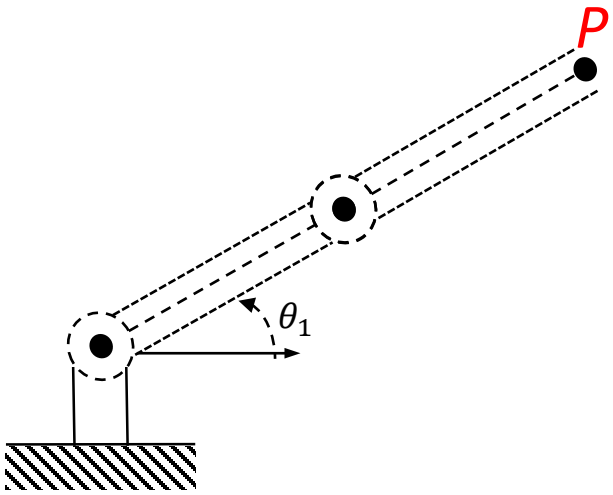


Jacobian singularities:

$$\det \mathbf{J} = 0 \Rightarrow a^2 \det \begin{pmatrix} -s_1 - s_{12} & -s_{12} \\ c_1 + c_{12} & c_{12} \end{pmatrix} = 0 \Rightarrow$$

$$\Rightarrow -c_{12}\left(s_1 + s_{12}\right) + s_{12}\left(c_1 + c_{12}\right) = c_1 s_{12} - s_1 c_{12} = 0 \Rightarrow$$

$$\Rightarrow \tan\theta_1 = \tan\left(\theta_1 + \theta_2\right) \Rightarrow \theta_2 = 0$$



Inverse Kinematics (9/11)

Jacobian (Pseudo-)inverse

non-redundant chain $n > m$

Reformulate the problem as a *linear constrained optimization*

$$\dot{\mathbf{q}}^* = \arg\min_{\dot{\mathbf{q}}} \left(\frac{1}{2} \dot{\mathbf{q}}^T \mathbf{W} \dot{\mathbf{q}} \right)$$

$$\text{s.t. } \dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{q}}$$

Recruit Lagrangian multipliers:

$$\dot{\mathbf{q}}^* = \arg\min_{\dot{\mathbf{q}},\lambda} \left(\frac{1}{2} \dot{\mathbf{q}}^T \mathbf{W} \dot{\mathbf{q}} + \lambda^T \left(\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}} \right) \right)$$

$$\dot{\mathbf{q}}^* = \mathbf{W}^{-1}\mathbf{J}^T \left(\mathbf{J}\mathbf{W}^{-1}\mathbf{J}^T \right)^{-1} \dot{\mathbf{x}}$$

$$\begin{cases} \mathbf{W} = \mathbf{I}_n \\ \mathbf{J}^\dagger = \mathbf{J}^T \left(\mathbf{J}\mathbf{J}^T \right)^{-1} \Rightarrow \dot{\mathbf{q}} = \mathbf{J}^\dagger \dot{\mathbf{x}} \\ \mathbf{J}\mathbf{J}^\dagger = \mathbf{I}_n \end{cases}$$

Inverse Kinematics (10/11)

Secondary Task

non-redundant chain $n > m$

Reformulate the problem as a *linear constrained optimization*

$$\dot{\mathbf{q}}^* = \arg\min_{\dot{\mathbf{q}}} \left(\frac{1}{2} \left(\dot{\mathbf{q}} - \dot{\mathbf{q}}_0 \right)^T \left(\dot{\mathbf{q}} - \dot{\mathbf{q}}_0 \right) \right)$$

$$\text{s.t. } \dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{q}}$$

$$\dot{\mathbf{q}} = \mathbf{J}^\dagger \dot{\mathbf{x}} + \left(\mathbf{I}_n - \mathbf{J}^\dagger \mathbf{J} \right) \dot{\mathbf{q}}_0$$

$$\dot{\mathbf{q}} = \dot{\mathbf{q}}_0 + \mathbf{J}^\dagger \left(\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}}_0 \right)$$

$\left(\mathbf{I}_n - \mathbf{J}^\dagger \mathbf{J} \right)$ Null-space projection operator allows for *internal motions*

How to pick up $\dot{\boldsymbol{q}}_0$:

$$\dot{\mathbf{q}}_0 = k_0 \left(\frac{\partial w(\mathbf{q})}{\partial \mathbf{q}} \right)^T$$

Improve manipulability

$$w(\mathbf{q}) = \sqrt{\det\left(\mathbf{J}(\mathbf{q}) \mathbf{J}^T(\mathbf{q}) \right)}$$

Joint limits avoidance

$$w(\mathbf{q}) = -\frac{1}{2n} \sum_{i=1}^{n} \left(\frac{q_i - \bar{q}_i}{q_{iM} - q_{im}} \right)^2$$

Inverse Kinematics (11/11)

Damped Least Squares

To deal with kinematic singularities

$$\dot{\mathbf{q}}^* = \arg\min_{\dot{\mathbf{q}}} \left(\frac{1}{2}\left(\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}}\right)^T \left(\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}}\right) + \frac{1}{2}k^2 \dot{\mathbf{q}}^T \dot{\mathbf{q}} \right)$$

$$\mathbf{J}^* = \mathbf{J}^T \left(\mathbf{J}\mathbf{J}^T + k^2 \mathbf{I}_n \right)^{-1}$$

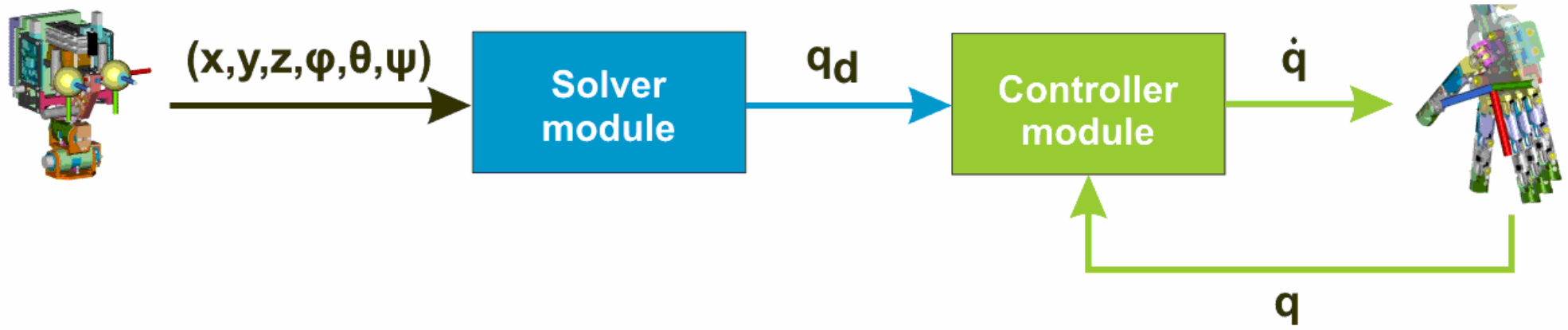
$$\dot{\mathbf{q}} = \mathbf{J}^* \dot{\mathbf{x}}$$

Levenberg-Marquardt method
for system of nonlinear equations

k establishes a synergy between:

- Jacobian (Pseudo-)inverse: $k = 0$
- Jacobian Transpose: $k \gg \max\{|\sigma_i|\}$

The Cartesian Controller (1/9)



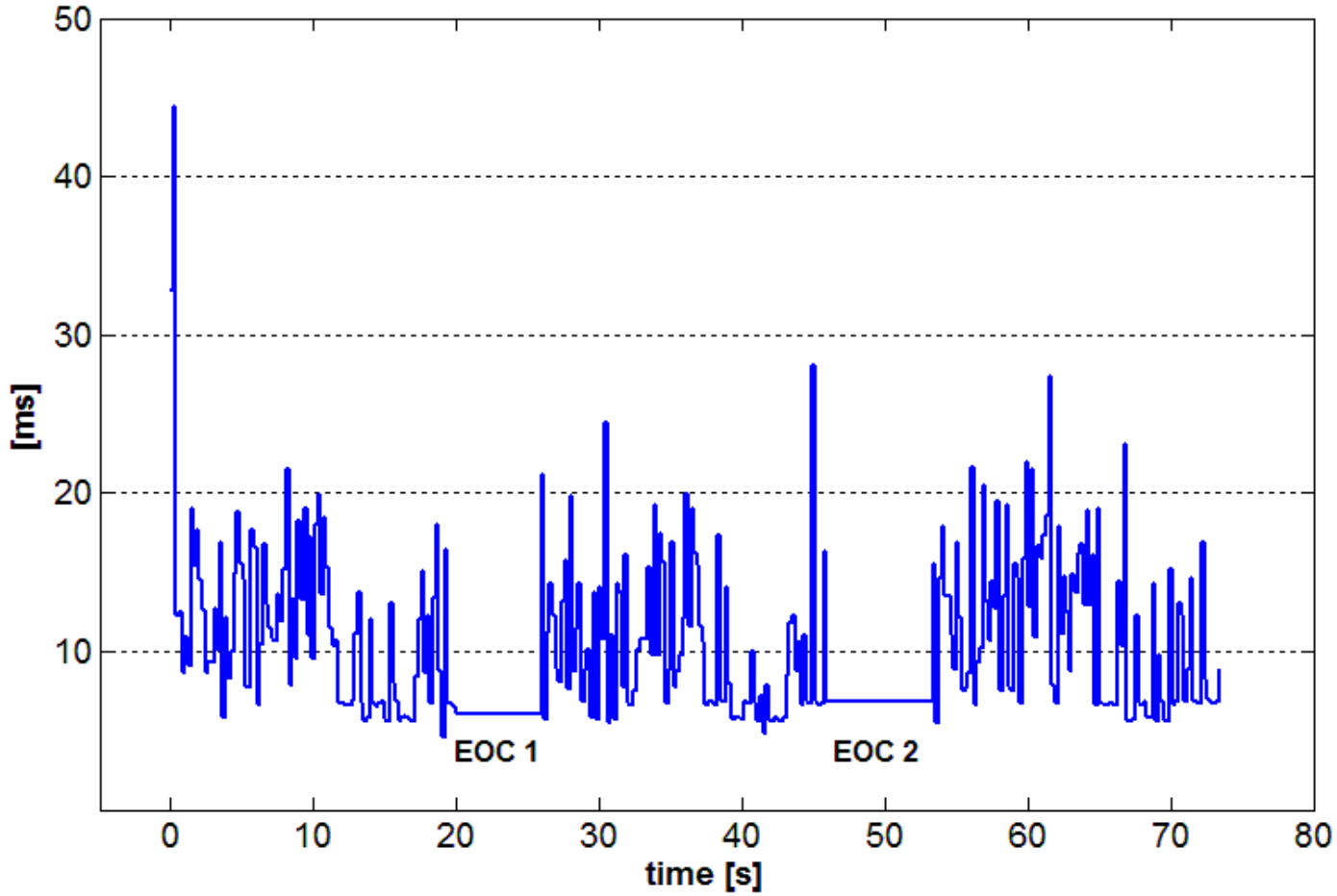
The Cartesian Controller (2/9)

$$\tilde{q}_d = \arg\min_{q \in \mathbb{R}^n} \left(\left\| \alpha_d - K_\alpha(q) \right\|^2 + \lambda \cdot (q_{\text{rest}} - q)^T W (q_{\text{rest}} - q) \right)$$

$$\text{s.t.} \begin{cases} \left\| x_d - K_x(q) \right\|^2 < \varepsilon \\ q_L < q < q_U \\ \text{other obstacles ...} \end{cases}$$

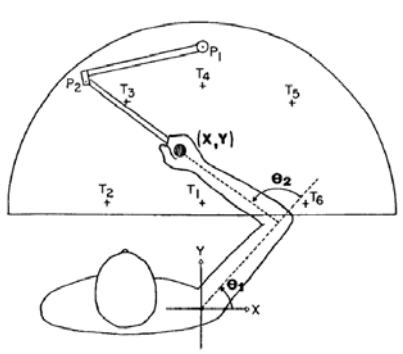
- **Quick convergence**: real-time compliant, < **20 ms**
- **Scalability**: *n* can be high and set on the fly
- **Singularities handling**: no Jacobian inversion
- **Joints bound handling**: no explicit boundary functions
- **Tasks hierarchy**: no use of null space
- **Complex constraints**: intrinsically nonlinear

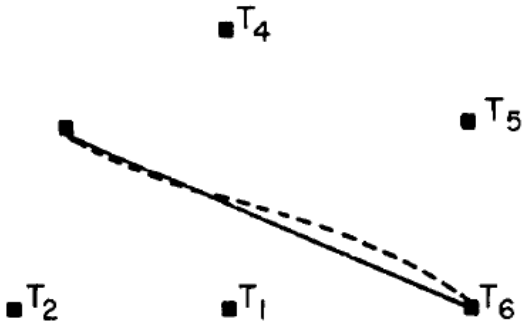
The Cartesian Controller (3/9)

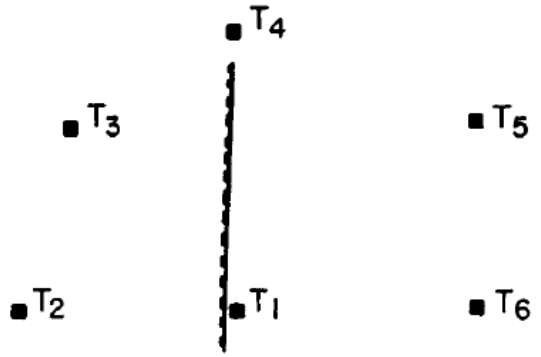


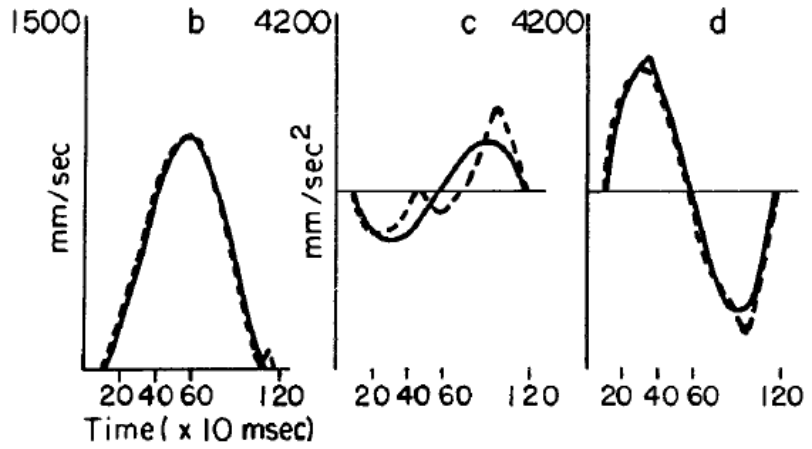
Solver running @ 33 Hz on multicore Intel (R) Xeon 2.27 GHz

The Cartesian Controller (4/9)

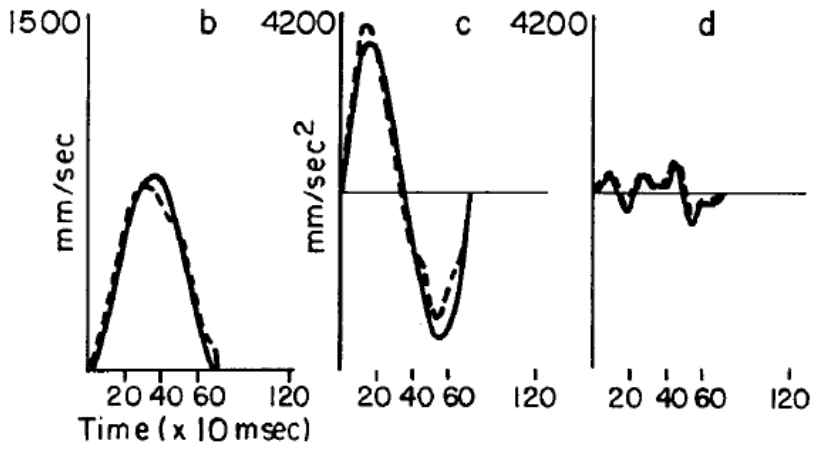






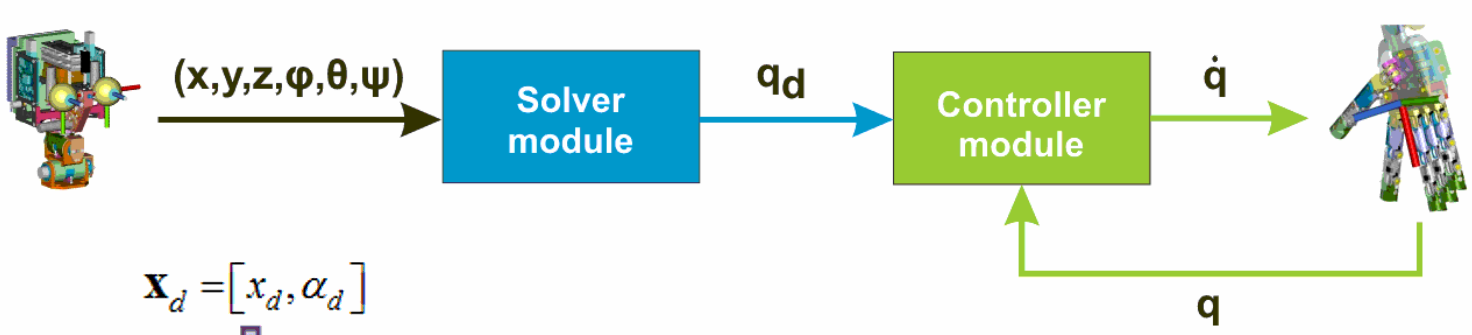


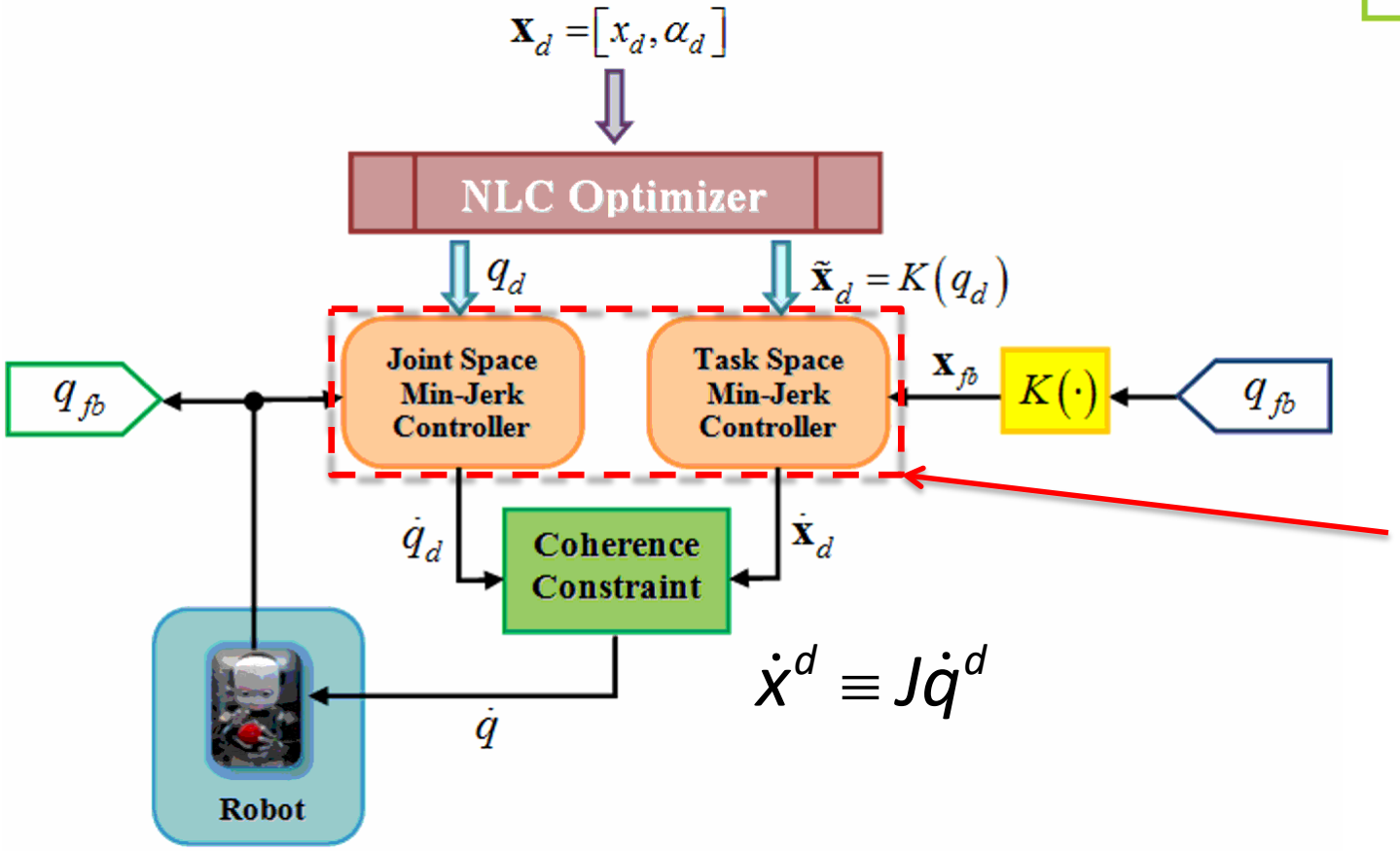
A



B

The Cartesian Controller (5/9)





Multi Referential Dynamical Systems

M. Hersch, A.G. Billard, "*Reaching with multi-referential dynamical systems*", Springer-Verlag.

The Cartesian Controller (6/9)

MJ closed-loop:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -\frac{60}{(T-t)^3} & -\frac{36}{(T-t)^2} & -\frac{9}{T-t} \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \ddot{x} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \frac{60}{(T-t)^3} \end{bmatrix} x_d$$

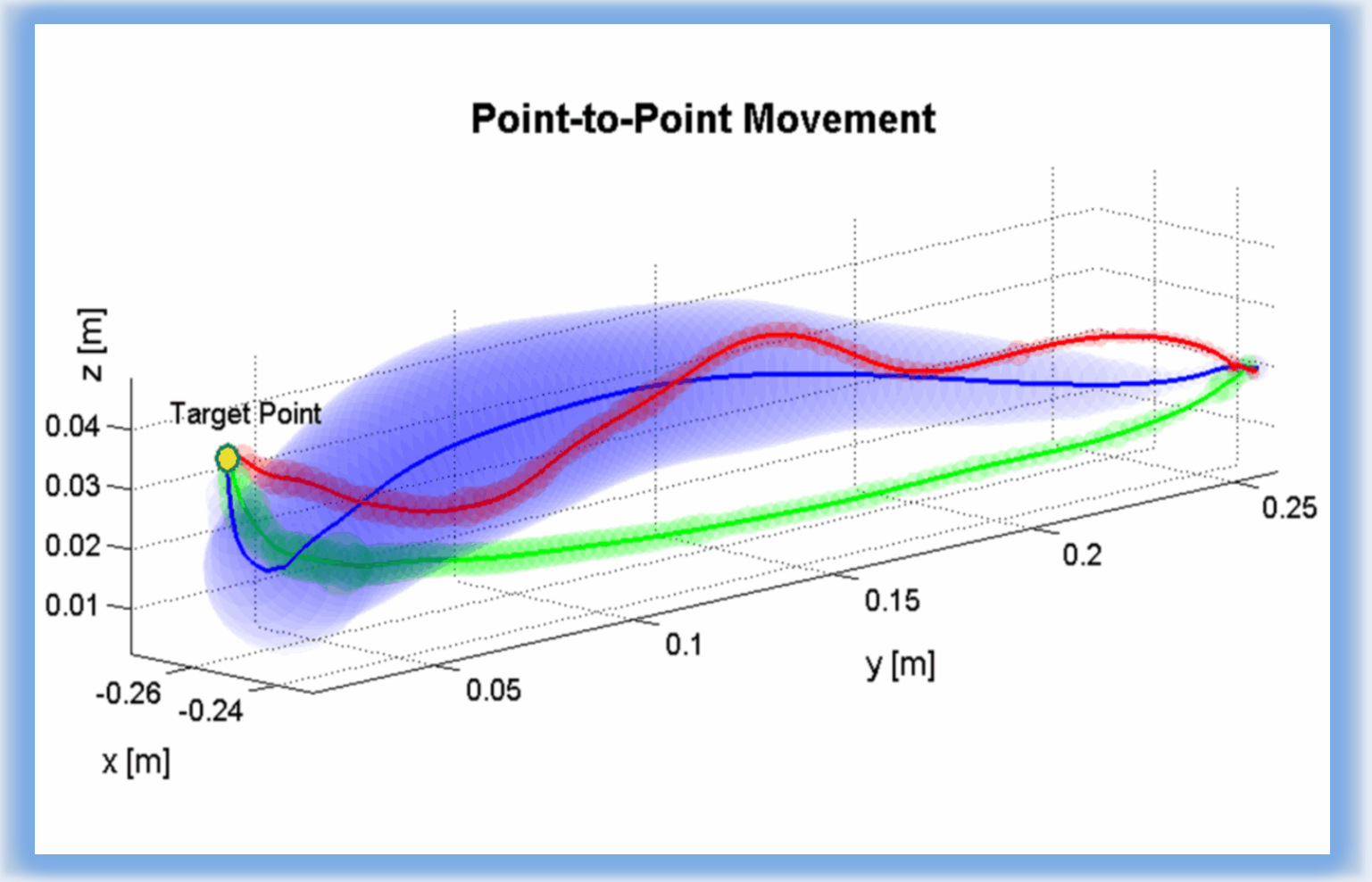
$$x(t) = \left(C_1 + C_2 t + C_3 t^2\right) \cdot \exp(\lambda \cdot t) + x_d$$

$$\lambda^* = \arg\min_{\lambda \in \mathbb{R}} \left(\int_0^\infty \dddot{x}^2(\tau)\, d\tau \right) \quad \text{s.t.} \begin{cases} \lambda < 0 \\ x(1) \geq 1 - \varepsilon \end{cases}$$

LTI min-jerk approximation:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dddot{x} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ \frac{a(\lambda)}{T^3} & \frac{b(\lambda)}{T^2} & \frac{c(\lambda)}{T} \end{bmatrix}}_{A} \begin{bmatrix} x \\ \dot{x} \\ \ddot{x} \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ -\frac{a(\lambda)}{T^3} \end{bmatrix}}_{B} x_d$$

The Cartesian Controller (7/9)

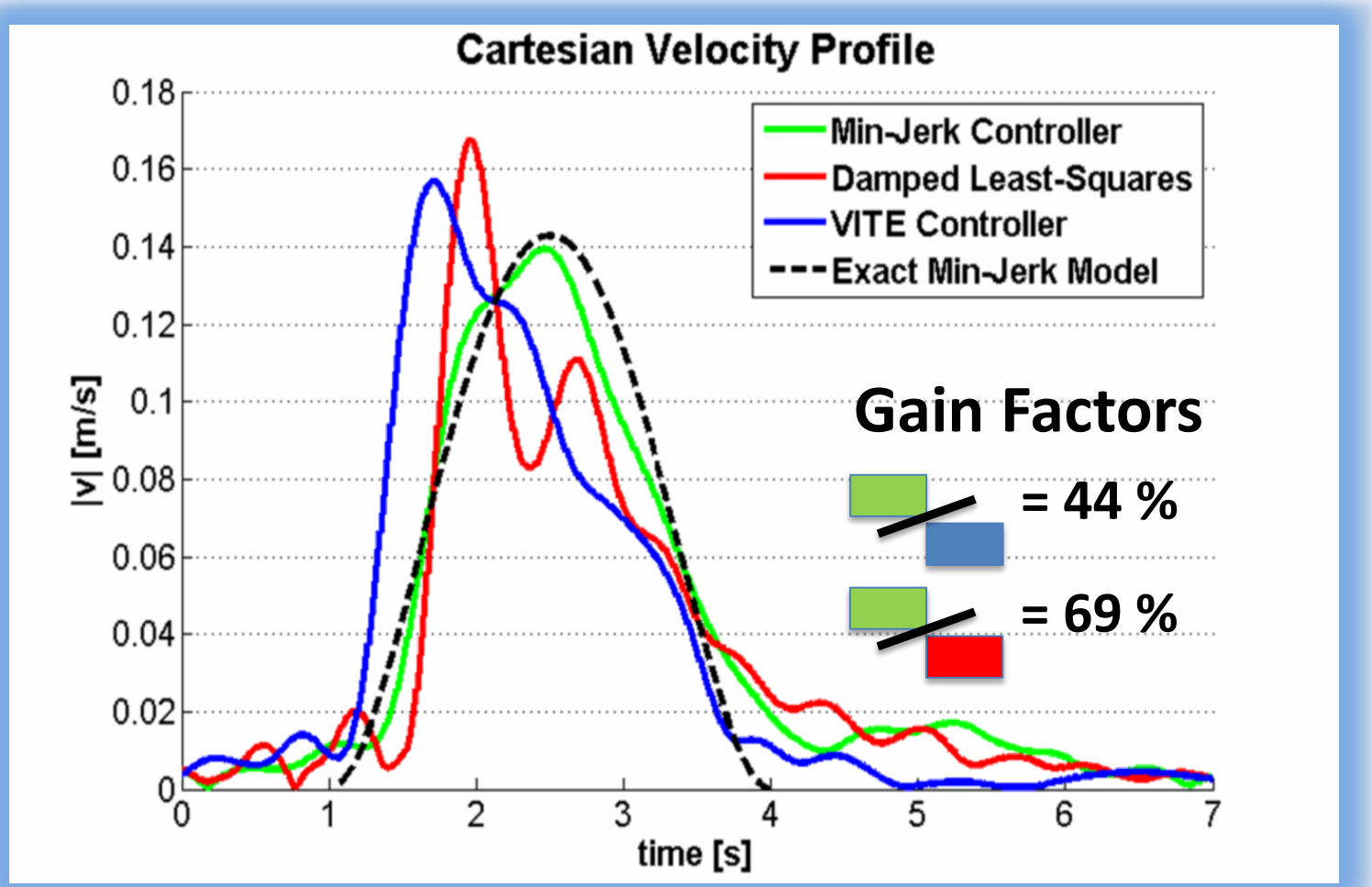


Min-Jerk

DLS

VITE

The Cartesian Controller (8/9)

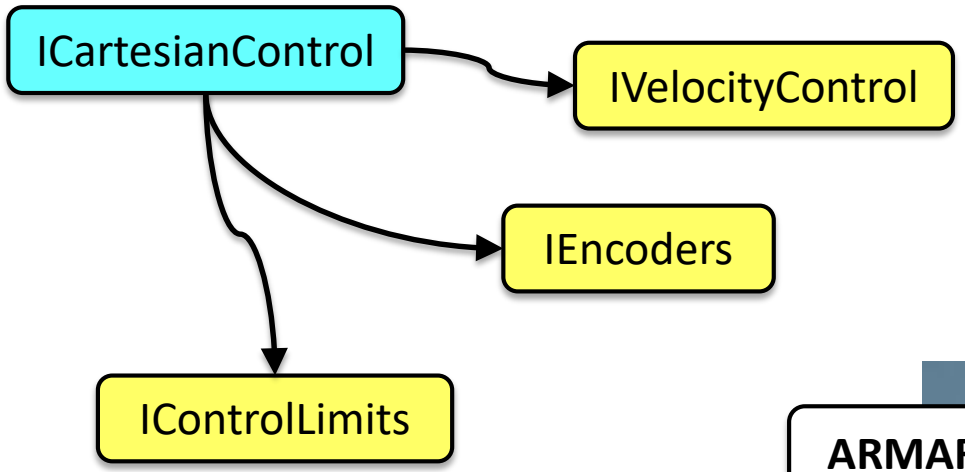


Min-Jerk

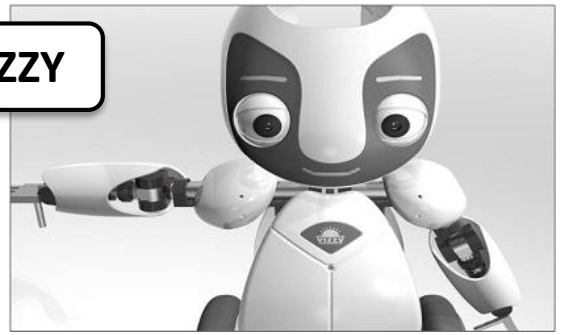
DLS

VITE

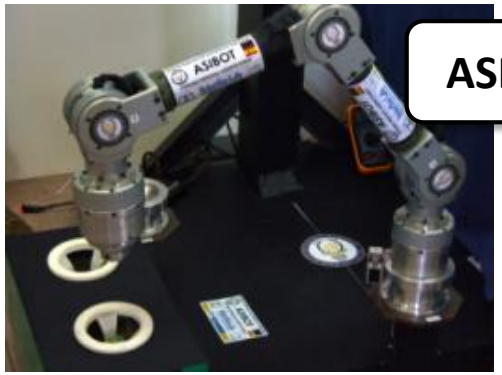
The Cartesian Controller (9/9)



VIZZY

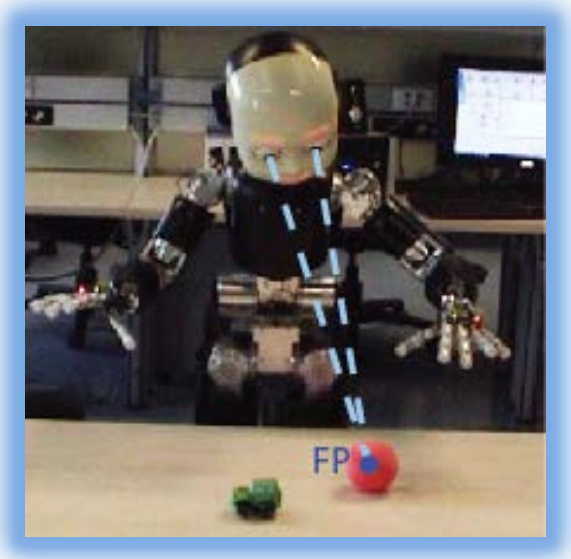


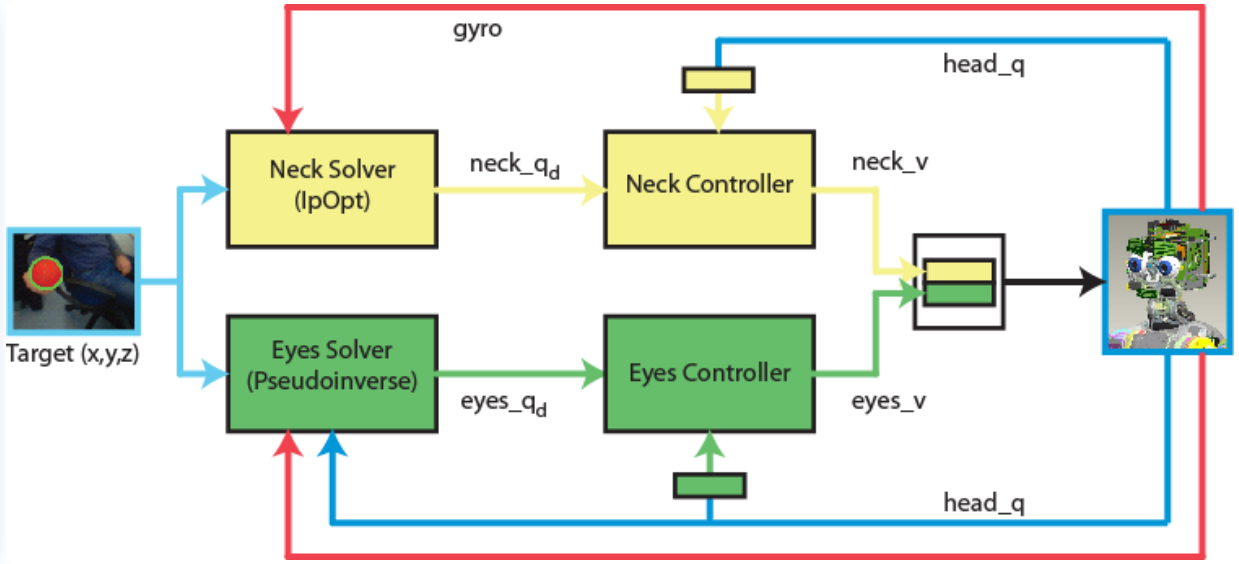
ARMAR III



ASIBOT

The Gaze Controller (1/9)

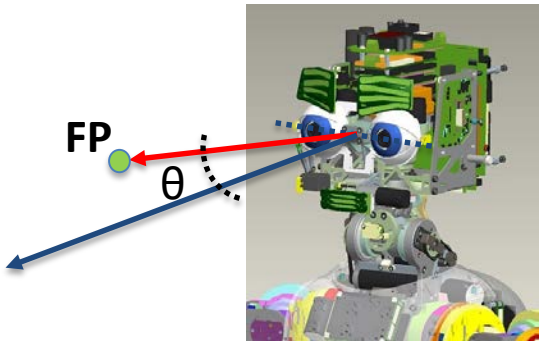




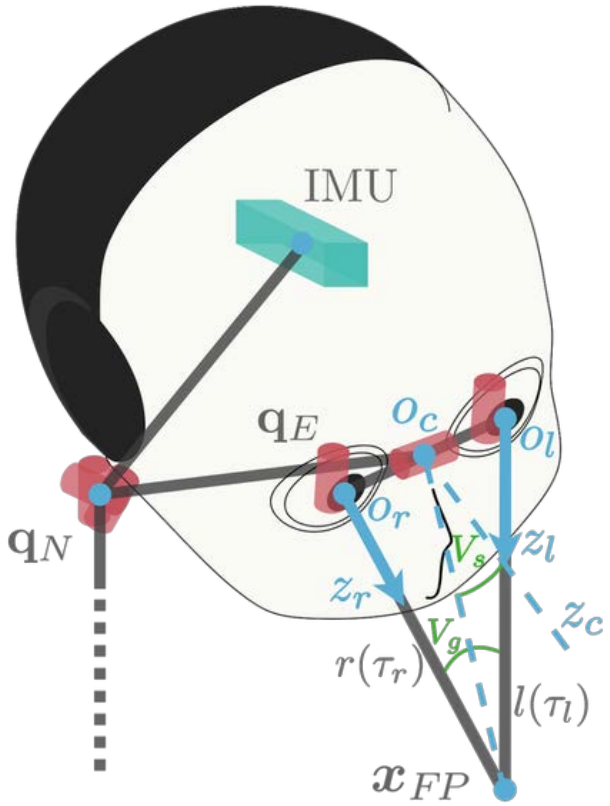
Yet another Cartesian Controller: **reuse ideas** …

Then, apply easy transformations from Cartesian to …

1. Egocentric angular space
2. Image planes (mono and stereo)

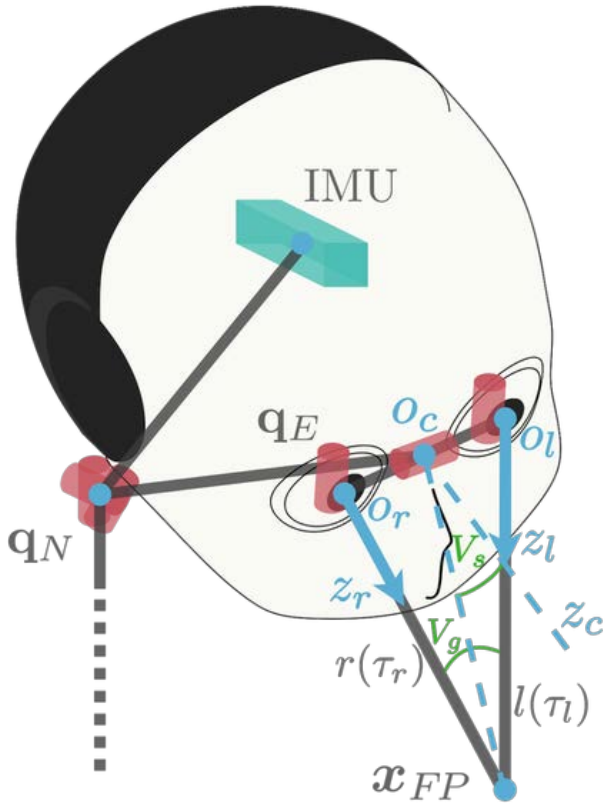


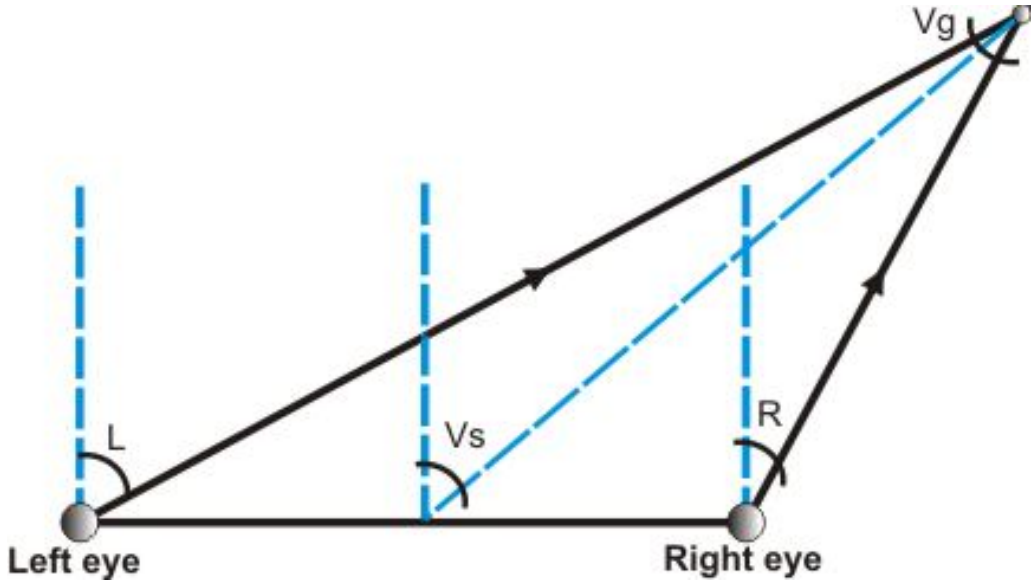
The Gaze Controller (2/9)



Joint #	Part	Joint Name	Range	Unit
0	Neck	**Pitch**	$+/-$	[deg]
1	Neck	**Roll**	$+/-$	[deg]
2	Neck	**Yaw**	$+/-$	[deg]
3	Eyes	**Tilt**	$+/-$	[deg]
4	Eyes	**Version**	$+/-$	[deg]
5	Eyes	**Vergence**	≥ 0	[deg]

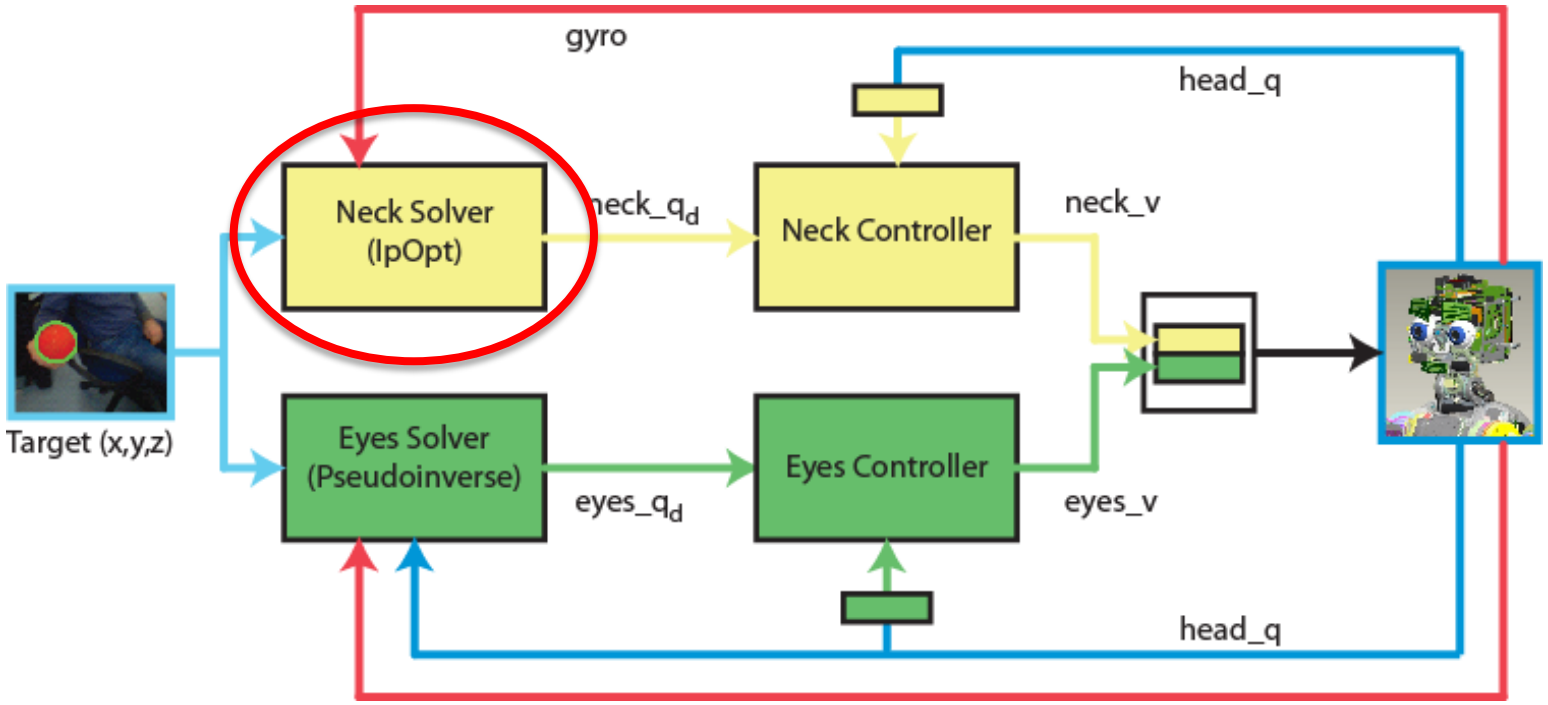
The Gaze Controller (3/9)

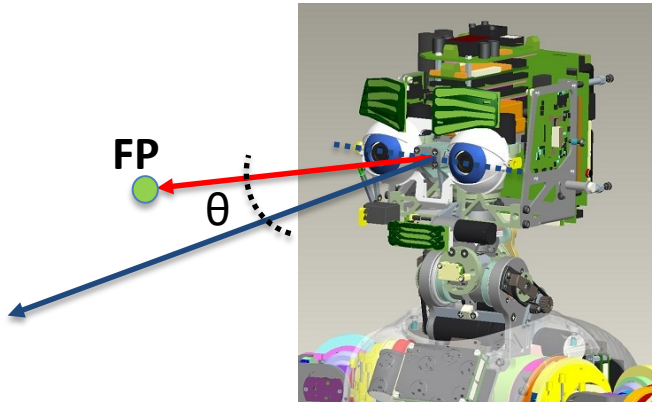




$$\begin{cases} V_g = L - R \\ V_s \approx (L + R)/2 \end{cases}$$

The Gaze Controller (4/9)

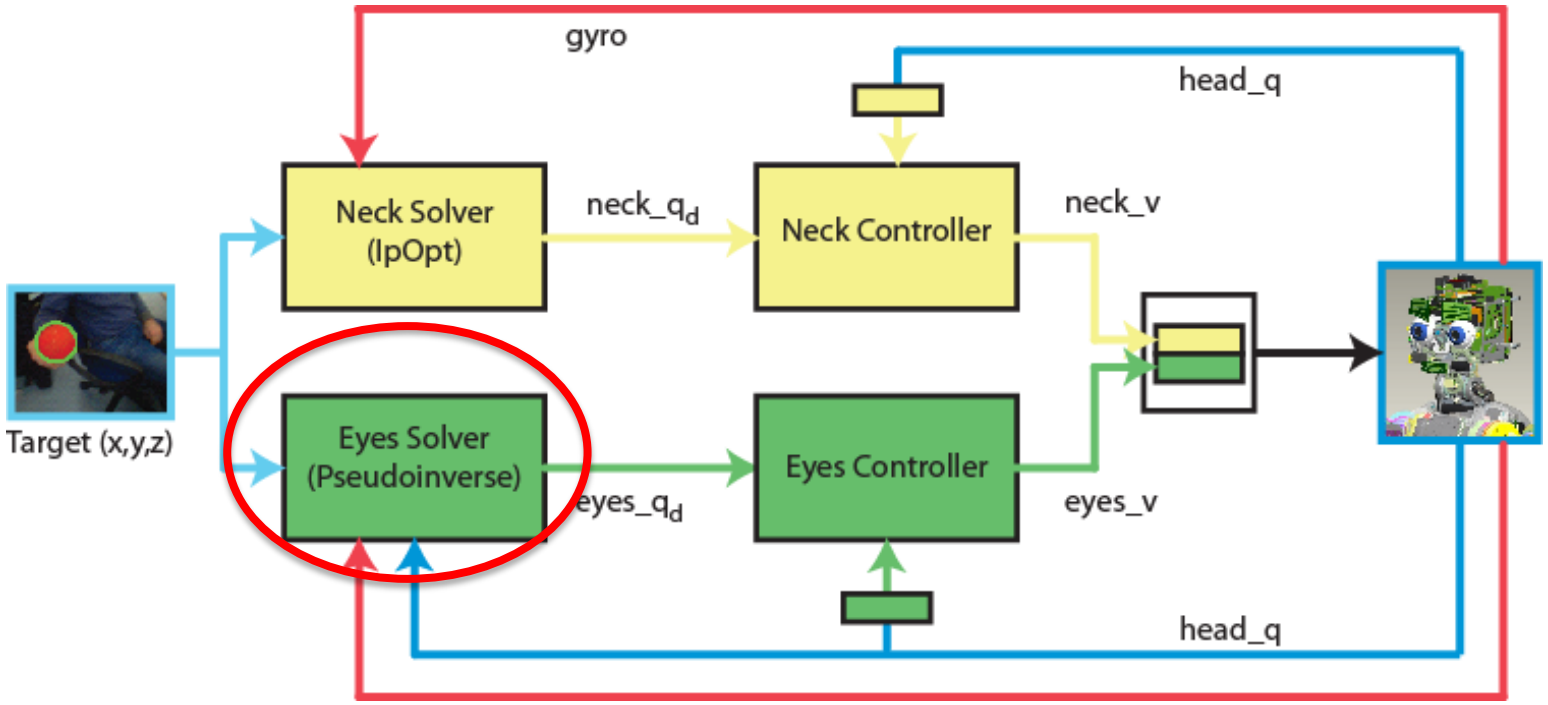




$$q^*_{\text{neck}} = \arg \min_{q_{\text{neck}} \in \mathbb{R}^3} \left\| q_{\text{rest}} - q_{\text{neck}} \right\|^2$$

$$\text{s.t.} \begin{cases} \cos\left(\theta\left(q_{\text{neck}}\right)\right) > 1 - \varepsilon \\ q_{\text{neck}_L} < q_{\text{neck}} < q_{\text{neck}_U} \end{cases}$$

The Gaze Controller (5/9)

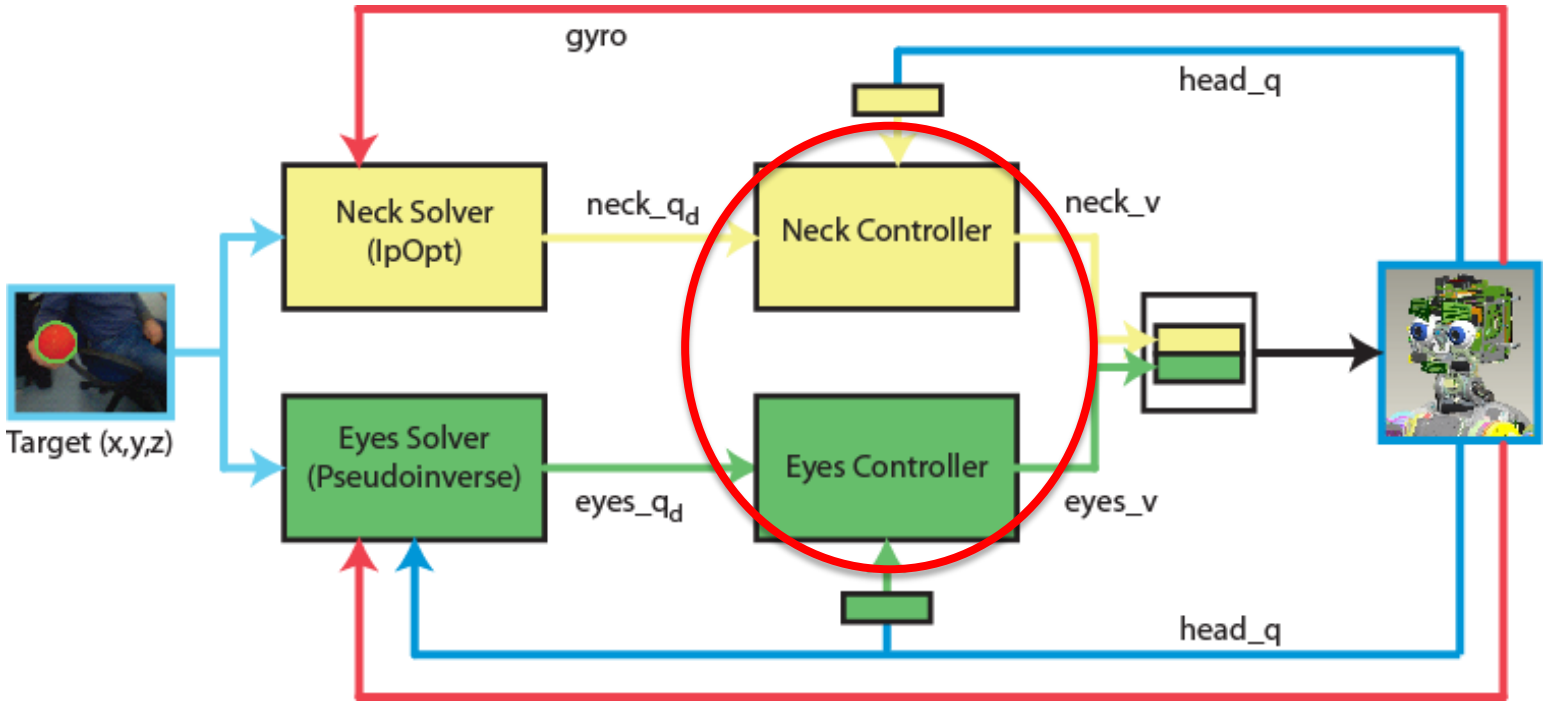


$$q^*_{\text{eyes}} = \arg\min_{q_{\text{eyes}} \in \mathbb{R}^3} \left\| FP_d - K_{FP}\left(q_{\text{eyes}}\right) \right\|^2$$

$$q_{\text{eyes}_{t+1}} = q_{\text{eyes}_t} + \Delta T \left(G \cdot J^{\#} \cdot \left(FP_d - K_{FP}\left(q_{\text{eyes}_t}\right) \right) - \dot{q}_c \right)$$

Gyro

The Gaze Controller (6/9)



Retain Controllers Laws

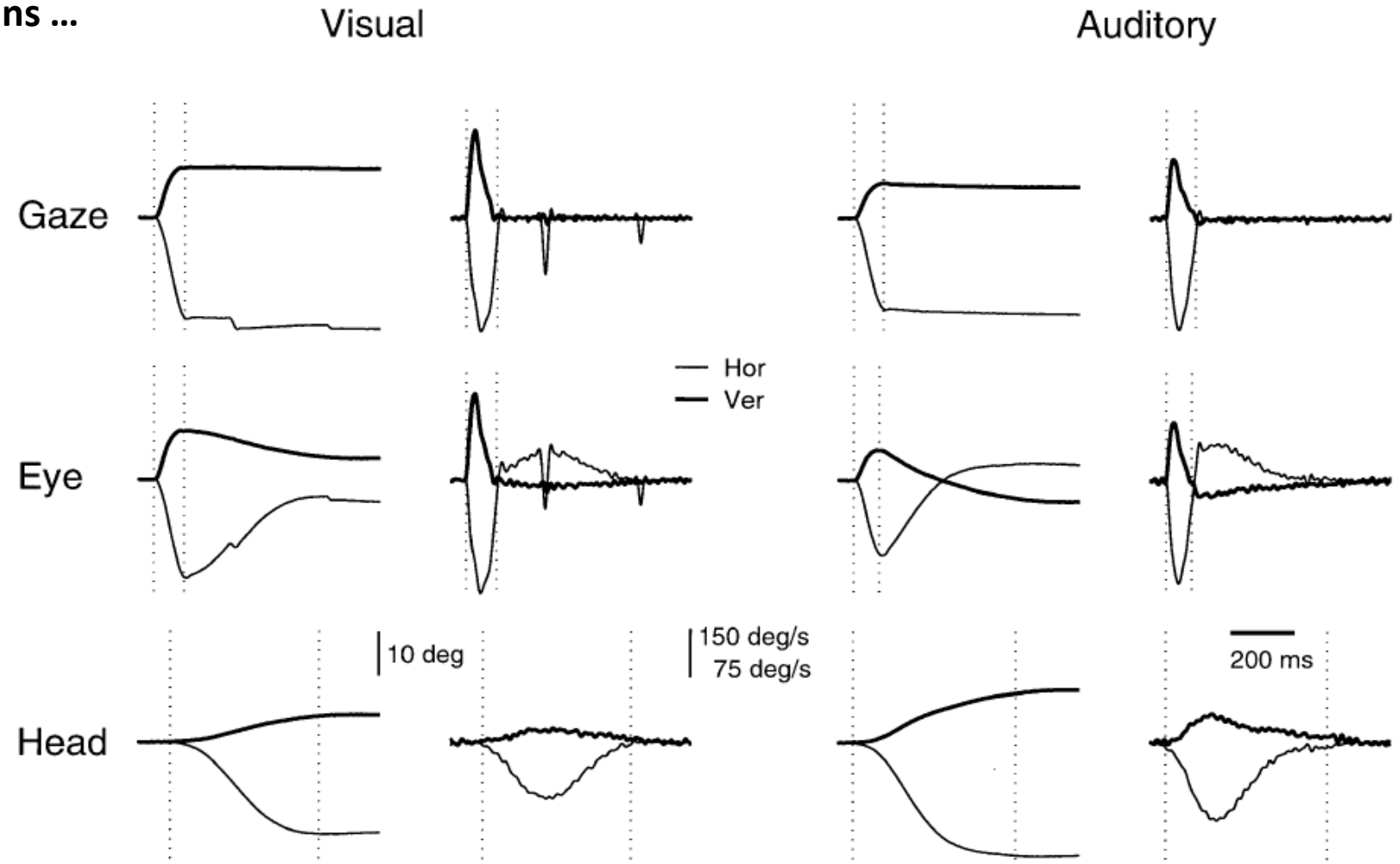
$$\frac{\dot{q}_{\text{neck}}}{q_{\text{neck}_d} - q_{\text{neck}}} = \frac{-a/T_{\text{neck}}}{s^2 - \left(c/T_{\text{neck}}^3\right)s - b/T_{\text{neck}}^2}$$

$$\frac{\dot{q}_{\text{eyes}}}{q_{\text{eyes}_d} - q_{\text{eyes}}} = \frac{-a/T_{\text{eyes}}}{s^2 - \left(c/T_{\text{eyes}}^3\right)s - b/T_{\text{eyes}}^2}$$

Feed Forward term delivered with low-level Position Control to implement **fast saccades**

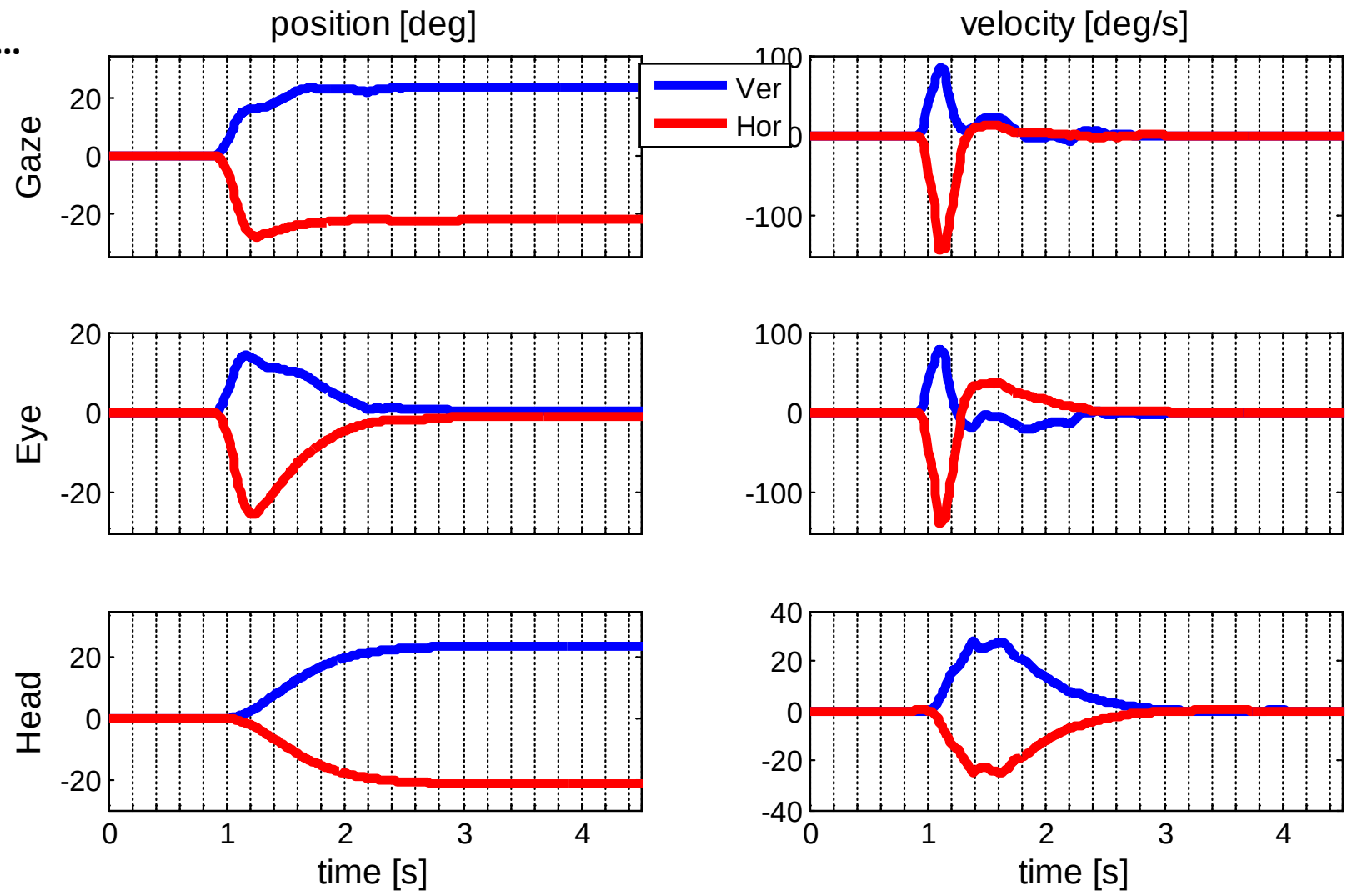
The Gaze Controller (7/9)

Studies on humans …

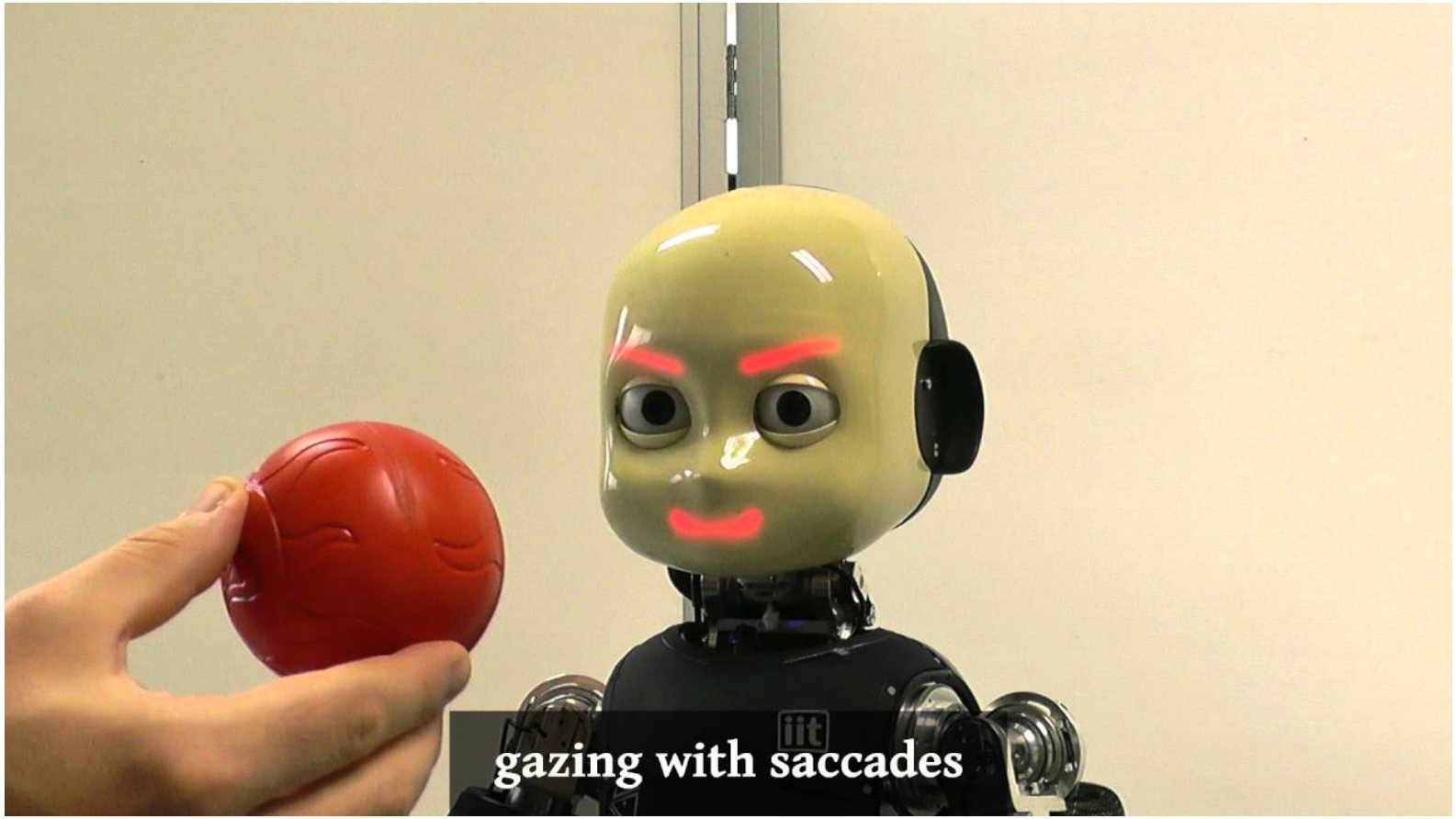


The Gaze Controller (8/9)

Results on iCub …



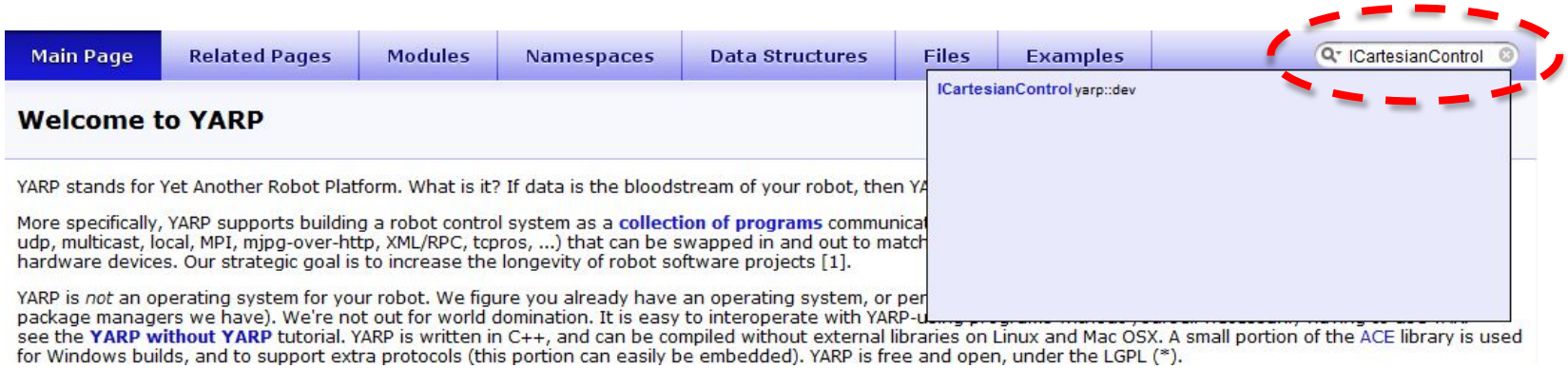
The Gaze Controller (9/9)



http://y2u.be/I4ZKfAvs1y0

Interface Documentation

In the search field: type **ICartesianControl/IGazeControl**



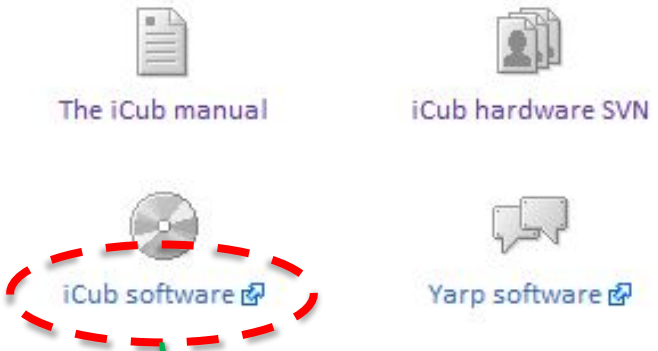
• • •

Public Member Functions

virtual	**~ICartesianControl** () Destructor.
virtual bool	**setTrackingMode** (const bool f)=0 Set the controller in tracking or non-tracking mode.
virtual bool	**getTrackingMode** (bool *f)=0 Get the current controller mode.
virtual bool	**getPose** (**yarp::sig::Vector** &x, **yarp::sig::Vector** &o)=0 Get the current pose of the end-effector.
virtual bool	**getPose** (const int axis, **yarp::sig::Vector** &x, **yarp::sig::Vector** &o)=0 Get the current pose of the specified link belonging to the kinematic chain.
virtual bool	**goToPose** (const **yarp::sig::Vector** &xd, const **yarp::sig::Vector** &od, const double t=0.0)=0 Move the end-effector to a specified pose (position and orientation) in cartesian space.
virtual bool	**goToPosition** (const **yarp::sig::Vector** &xd, const double t=0.0)=0 Move the end-effector to a specified position in cartesian space, ignore the orientation.
virtual bool	**goToPoseSync** (const **yarp::sig::Vector** &xd, const **yarp::sig::Vector** &od, const double t=0.0)=0 Move the end-effector to a specified pose (position and orientation) in cartesian space.
virtual bool	**goToPositionSync** (const **yarp::sig::Vector** &xd, const double t=0.0)=0 Move the end-effector to a specified position in cartesian space, ignore the orientation.
virtual bool	**getDesired** (**yarp::sig::Vector** &xdhat, **yarp::sig::Vector** &odhat, **yarp::sig::Vector** &qdhat)=0 Get the actual desired pose and joints configuration as result of kinematic inversion.
virtual bool	**askForPose** (const **yarp::sig::Vector** &xd, const **yarp::sig::Vector** &od, **yarp::sig::Vector** &xdhat, **yarp::sig::Vector** &odhat, **yarp::sig::Vector** &qdhat)=0 Ask for inverting a given pose without actually moving there.
virtual bool	**askForPose** (const **yarp::sig::Vector** &q0, const **yarp::sig::Vector** &xd, const **yarp::sig::Vector** &od, **yarp::sig::Vector** &xdhat, **yarp::sig::Vector** &odhat, **yarp::sig::Vector** &qdhat)=0 Ask for inverting a given pose without actually moving there.
virtual bool	**askForPosition** (const **yarp::sig::Vector** &xd, **yarp::sig::Vector** &xdhat, **yarp::sig::Vector** &odhat, **yarp::sig::Vector** &qdhat)=0 Ask for inverting a given position without actually moving there.

Doxygen Documentation

Interface Tutorials



- **Software** - most of the software (including **iCub modules**)
- **Applications** - a list of documented applications (collections of modules)
- **Tutorials** - a set of tutorials to learn how to use the software
- The documentation for contributed software is here: Contrib documentation
- Programmer's checklist:
  - **Compile status** - check if your code is compiling on a test server
  - Licensing - have you declared your authorship, and rights granted?
  - Coding guidelines - some tips on how to write your code
  - Modules and CMake - some tips on how to make your code compilable
  - Committing to the repository - things to check before committing files to the repository
- Reference material:
  - The The iCub manual
  - The RobotCub Website.
  - Getting the software.
  - Our software architecture, YARP.

- **The classic hello world** - how to write the very first program
- **Getting accustomed with motor interfaces** - a tutorial on how to use the motor interfaces
- **Getting accustomed with torque/impedance interfaces** - a tutorial on how to use the joint level torque/impedance interface
- **Basic Image Processing** - a tutorial on a basic image processing
- **The ResourceFinder Class (basic)** - a tutorial on how to organize the command line parameters of your modules
- **The ResourceFinder Class (advanced)** - organizing parameters: advanced tutorial
- **The RFModule Class** - a tutorial on how to use the module helper class to write a program
- **The RateThread Class** - a tutorial on how to write a control loop using threads
- **The Cartesian Interface** - a tutorial on how to control a robot's limb in the operational space
- **The Gaze Interface** - a tutorial on how to control the robot gaze through a Yarp interface
- **A short introduction to iDyn** - a short introduction to the iDyn library
  - **Computation of torques in a single chain, using iDyn** - how to compute torques in a single chain, using iDyn library

Interface Communalities (1/3)

OPENING THE CARTESIAN INTERFACE

```
#include <yarp/dev/all.h>
Property option;

option.put("device","cartesiancontrollerclient");
option.put("remote","/icub/cartesianController/right_arm");
option.put("local","/client/right_arm");

PolyDriver clientCartCtrl(option);

ICartesianControl *icart=NULL;
if (clientCartCtrl.isValid()) {
   clientCartCtrl.view(icart);
}
```

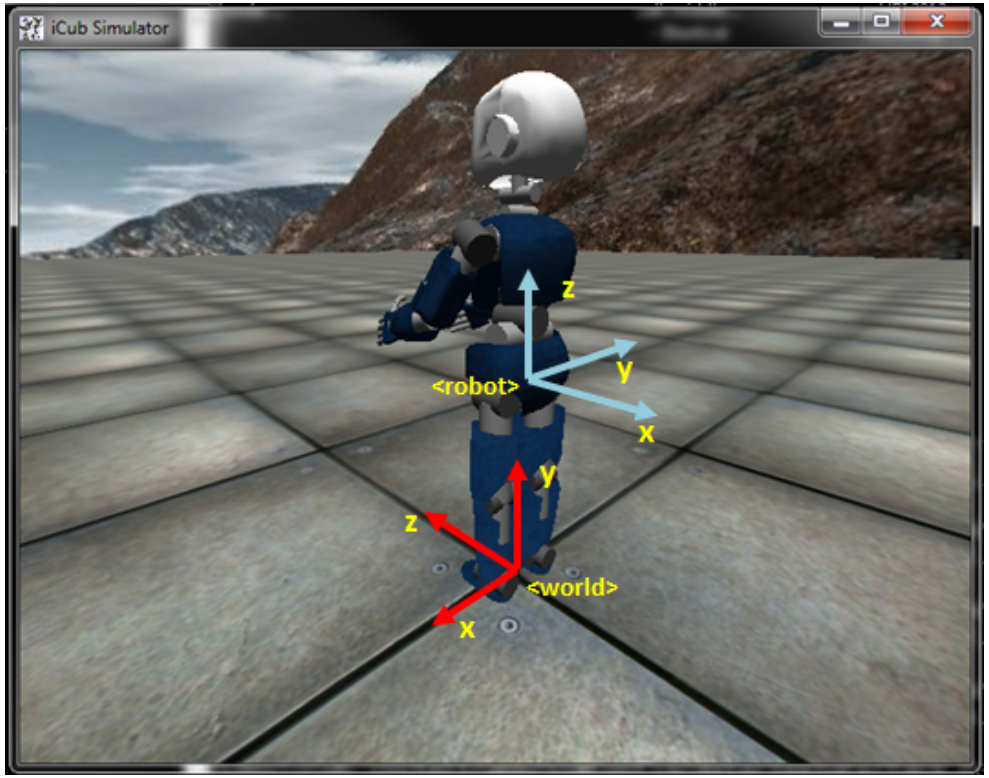
Interface Communalities (2/3)

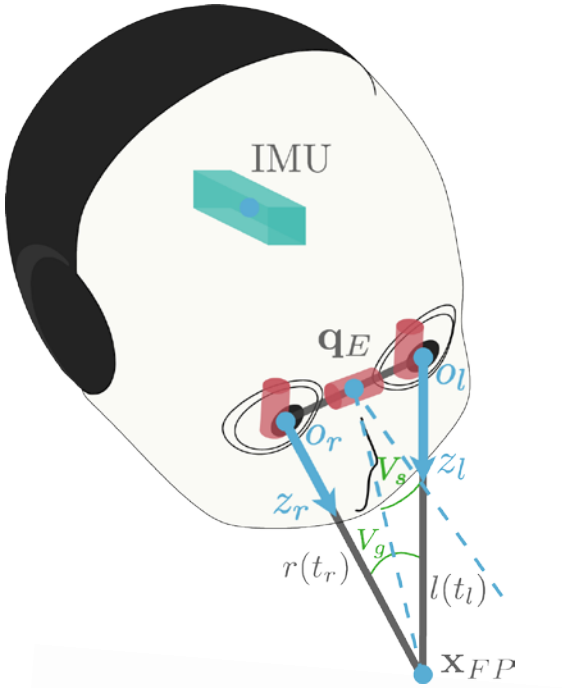
OPENING THE GAZE INTERFACE

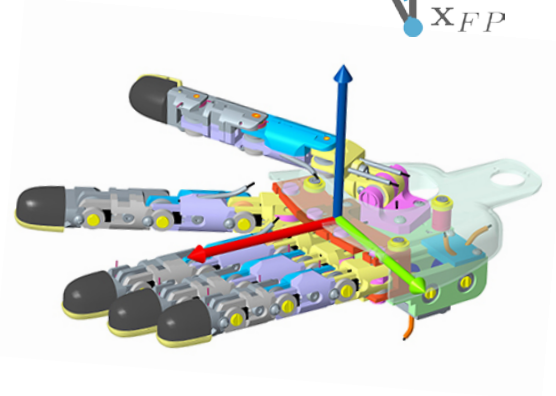
```
#include <yarp/dev/all.h>
Property option;

option.put("device",“gazecontrollerclient");
option.put("remote","/iKinGazeCtrl");
option.put("local","/client/gaze”);

PolyDriver clientGazeCtrl(option);

IGazeControl *igaze=NULL;
if (clientGazeCtrl.isValid()) {
   clientGazeCtrl.view(igaze);
}
```

Interface Communalities (3/3)

Coordinate Systems



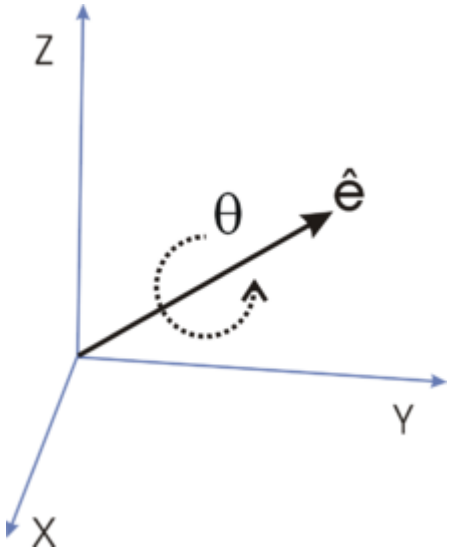




Cartesian Interface (1/7)

Orientation: Axis-Angle

$$r = [\underbrace{e_x\ e_y\ e_z}_{\|e\| = 1}\ \underbrace{\theta}_{rad}]$$



TARGET ORIENTATION through DIRECTION COSINE MATRIX

```
Matrix R(3,3);
// pose x-axis   y-axis         z-axis
   R(0,0)= 0.0;  R(0,1)= 1.0;  R(0,2)= 0.0;  // x-coordinate
   R(1,0)= 0.0;  R(1,1)= 0.0;  R(1,2)=-1.0;  // y-coordinate
   R(2,0)=-1.0;  R(2,1)= 0.0;  R(2,2)= 0.0;  // z-coordinate

Vector o=ctrl::dcm2axis(R);
```

Cartesian Interface (2/7)

RETRIEVE CURRENT POSE

```
Vector x,o;
icart->getPose(x,o);
```

REACH FOR A TARGET POSE (SEND-AND-FORGET)

```
icart->goToPose(xd,od);
icart->goToPosition(xd);
```

REACH FOR A TARGET POSE (WAIT-FOR-REPLY)

```
icart->goToPoseSync(xd,od);
icart->goToPositionSync(xd);
```

REACH AND WAIT

```
icart->goToPoseSync(xd,od);
icart->waitMotionDone();
```

Cartesian Interface (3/7)

ASK FOR A POSE (without moving)

```
Vector xdhat,odhat,qdhat;
icart->askForPose(xd,xdhat,odhat,qdhat);
```

MOVE FASTER/SLOWER

```
icart->setTrajTime(1.5);  // point-to-point trajectory time
```

REACH WITH GIVEN PRECISION

```
icart->setInTargetTol(0.001);
```

KEEP THE POSE ONCE DONE

```
icart->setTrackingMode(true);
```

Cartesian Interface (4/7)

ENABLE/DISABLE DOF

```
Vector curDof;
icart->getDOF(curDof); // [0 0 0 1 1 1 1 1 1 1]

Vector newDof(3);
newDof[0]=1; // torso pitch: 1 => enable
newDof[1]=2; // torso roll: 2 => skip
newDof[2]=1; // torso yaw: 1 => enable
icart->setDOF(newDof,curDof);
```

GIVE PRIORITY TO REACHING IN POSITION/ORIENTATION

```
icart->setPosePriority("position");  // default
icart->setPosePriority("orientation");
```

Cartesian Interface (5/7)

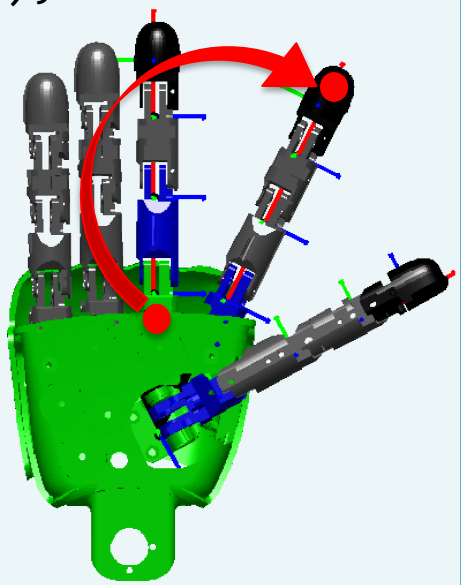
CONTEXT SWITCH

```
icart->setDOF(newDof1,curDof1);    // prepare the context
icart->setTrackingMode(true);

int context_0;
icart->storeContext(&context_0);   // latch the context

icart->setDOF(newDof2,curDof2);    // perform some actions
icart->goToPose(x,o);

icart->restoreContext(context_0); // retrieve context_0
icart->goToPose(x,o);             // perform with context_0
```

Cartesian Interface (6/7)

DEFINING A DIFFERENT EFFECTOR

```
iCubFinger finger("right_index");
Vector encs; iencs->getEncoders(encs.data());
Vector joints; finger.getChainJoints(encs,joints);
Matrix tipFrame=finger.getH((M_PI/180.0)*joints);

Vector tip_x=tipFrame.getCol(3);
Vector tip_o=ctrl::dcm2axis(tipFrame);

icart->attachTipFrame(tip_x,tip_o);

icart->getPose(x,o);
icart->goToPose(xd,od);

icart->removeTipFrame();
```



Cartesian Interface (7/7)

Find out more (e.g. **Events Callbacks** …):
http://wiki.icub.org/iCub/main/dox/html/icub_cartesian_interface.html

USING THE INTERFACE ALONG WITH THE SIMULATOR

```
1> iCub_SIM
2> yarprobotinterface --context simCartesianControl
3> iKinCartesianSolver --context simCartesianControl --part left_arm


option.put("device","cartesiancontrollerclient");
option.put("remote","/icubSim/cartesianController/left_arm");
option.put("local","/client/right_arm");
```

Gaze Interface (1/7)

GET CURRENT FIXATION POINT IN CARTESIAN DOMAIN

```
Vector x;
igaze->getFixationPoint(x);
```

GET CURRENT FIXATION POINT IN ANGULAR DOMAIN

```
Vector ang;
igaze->getAngles(ang);
// ang[0] => azimuth [deg]
// ang[1] => elevation [deg]
// ang[2] => vergence [deg]
```

LOOK AT 3D POINT

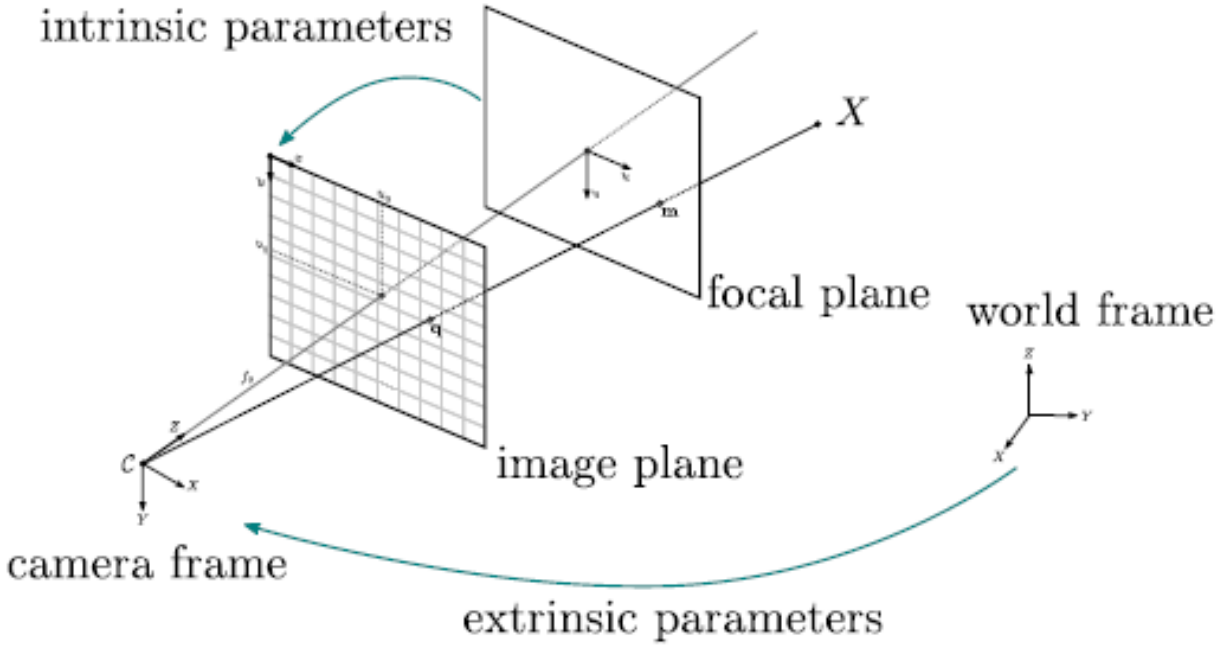
```
igaze->lookAtFixationPoint(xd);
```

… IN ANGULAR DOMAIN

```
igaze->lookAtAbsAngles(ang);
igaze->lookAtRelAngles(ang);
```

Gaze Interface (2/7)

LOOK AT POINT IN IMAGE DOMAIN



Gaze Interface (3/7)

LOOK AT POINT IN IMAGE DOMAIN

```
int camSel=0;  // 0 => left, 1 => right
Vector px(2);
px[0]=100;
px[1]=50;
double z=1.0;

igaze->lookAtMonoPixel(camSel,px,z);
```

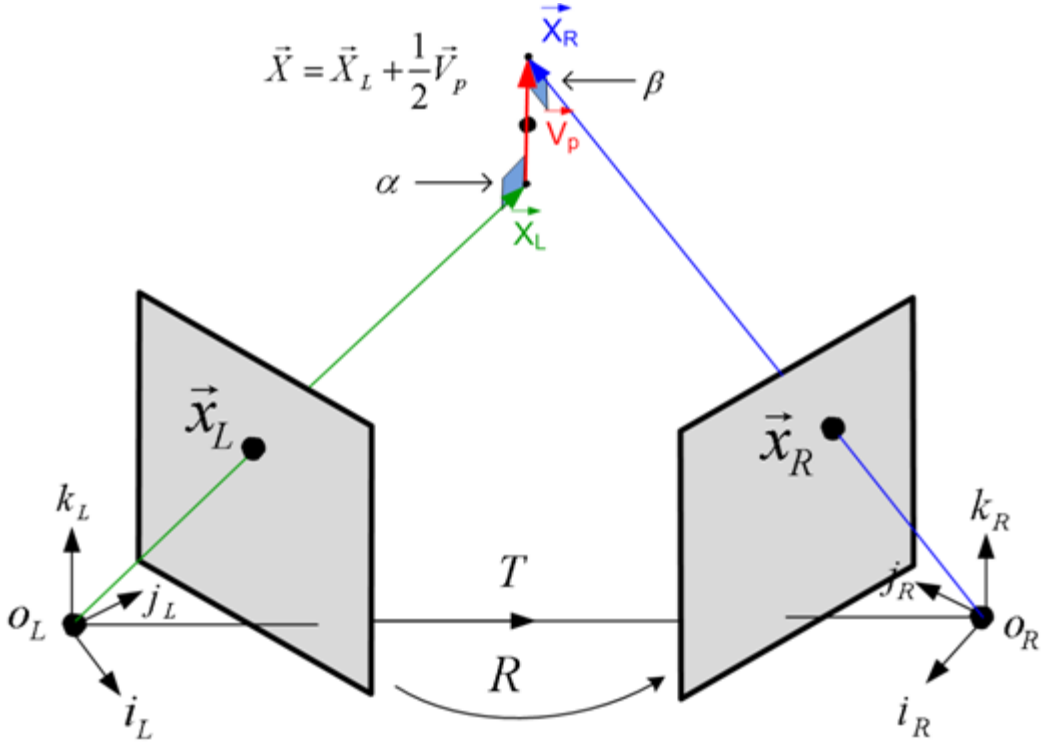
… EQUIVALENT TO

```
Vector x;
igaze->get3DPoint(camSel,px,z,x);
igaze->lookAtFixationPoint(x);
```

Gaze Interface (4/7)

GEOMETRY OF PIXELS

```
Vector x;
igaze->get3DPointOnPlane(camSel,px,plane,x);
igaze->get3DPointFromAngles(mode,ang,x);
igaze->triangulate3DPoint(pxl,pxr,x);
```



Gaze Interface (5/7)

GEOMETRY OF PIXELS

```
Vector x;
igaze->get3DPointOnPlane(camSel,px,plane,x);
igaze->get3DPointFromAngles(mode,ang,x);
igaze->triangulate3DPoint(pxl,pxr,x);
```

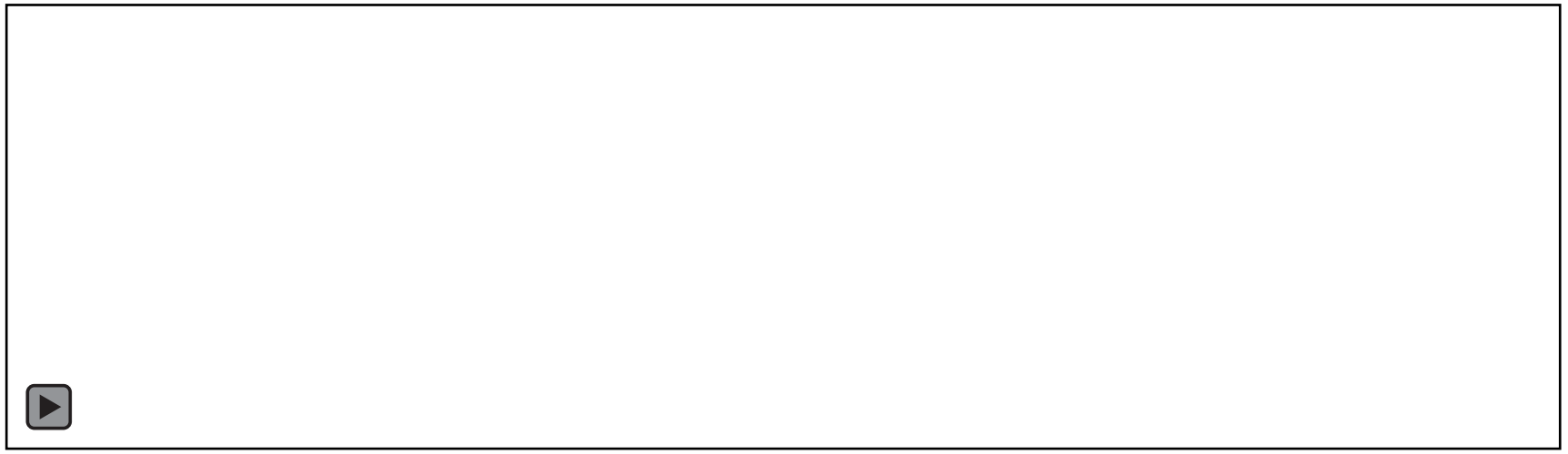
LOOK AT POINT WITH STEREO APPROACH => LOOPING!

```
Vector c(2); c[0]=160.0; c[1]=120.0;
bool converged=false;

while (!converged) {
    Vector pxl(2),pxr(2);
    pxl[0]=...; pxl[1]=...;  // retrieve data from vision
    pxr[0]=...; pxr[1]=...;

    igaze->lookAtStereoPixels(pxl,pxr);
    converged=(0.5*(norm(c-pxl)+norm(c-pxr))<5);
}
```

Gaze Interface (6/7)

GEOMETRY OF PIXELS

```
Vector x;
igaze->get3DPointOnPlane(camSel,px,plane,x);
igaze->get3DPointFromAngles(mode,ang,x);
igaze->triangulate3DPoint(pxl,pxr,x);
```



Gaze Interface (7/7)

Find out more (e.g. **Events Callbacks, Fast Saccadic Mode** …):
http://wiki.icub.org/iCub/main/dox/html/icub_gaze_interface.html

USING THE INTERFACE ALONG WITH THE SIMULATOR

```
1> iCub_SIM
2> iKinGazeCtrl --from configSim.ini

option.put("device",“gazecontrollerclient");
option.put("remote","/iKinGazeCtrl”);
option.put("local","/client/right_arm");
```