

## The Secret Life of the $\mathcal{N}$ -Matrix

Lutz "Gefahr" Freitag  
Freie Universität Berlin

RoHOW - Hamburg, 2015

Motivation

Let's derive

- ▶ Robots should move!
- ▶ But describing robot motions in terms of joints is cumbersome

There are quite a lot of different IK approaches. But most real-world robots use geometric IKs.

So did we...

# What do we want?

- ▶ Tell the robot where to move (not how)
  - ▶ Move left foot to position  $p_{left\_foot}$
  - ▶ Move right foot to position  $p_{right\_foot}$
  - ▶ While having the feet in a "natural" orientation
- ▶ Tell the robot how to move (constraints)
  - ▶ Keep the COM somewhere "safe"

We build tasks for all this

- ▶ **Joint:**  
Pretty self-explanatory (something elementary that moves)
- ▶ **Value:**  
The angle of a rotation-**joint** or the stroke of a piston-**joint**, etc.
- ▶ **Configuration/Pose/Posture:**  
A vector containing all **joint-values**
- ▶ **Task:**  
Something the robot shall accomplish, eg:
  - ▶ Move a body part to a certain **target**
  - ▶ Orient a body part in a **direction**
  - ▶ Have a certain **value** at a joint

Tasks generate Jacobians and errors
- ▶ **Target/Method:**  
The representation of the solution space for a task eg:
  - ▶ Point - move a bodypart to a position  $\rightarrow \dim(taskspace) = 0$
  - ▶ Line - move a bodypart onto a line  $\rightarrow \dim(taskspace) = 1$
  - ▶ Plane - move a bodypart onto a plane  $\rightarrow \dim(taskspace) = 2$
  - ▶ Space - for highdimensional robots you could solve for n-dim spaces!

## Demo

| python eins.py

Here comes the math!

Yey!



Define a loss function:

$$\mathcal{L}(\Delta q) = \|\phi(q + \Delta q) - y^*\|_C^2 + \|\Delta q\|_W^2 \quad (1)$$

$W$  and  $C$  are weighting metrics. We really care about the solution of our Task  $\phi(q + \Delta q) - y^* \rightarrow 0$  so we set  $C \rightarrow \infty$

$W$  scales the joint-changes. Since we treat every joint equally we choose  $W = I$

with  $\phi(q + \Delta q) = \phi(q) + J * \Delta q$  (local linearization) we get:

$$\mathcal{L}(\Delta q) = \underbrace{\|\phi(q) + J * \Delta q - y^*\|_C^2}_{\text{move to target}} + \underbrace{\|\Delta q\|_W^2}_{\text{be lazy}}$$

You know what time it is!

Its **derive and set zero**-time!

$$\begin{aligned}
 \frac{\delta}{\delta \Delta q} \mathcal{L}(\Delta q) &= 0^T = \|\phi(q) + J * \Delta q - y^*\|_C^2 + \|\Delta q\|_W^2 \\
 0^T &= 2(\phi(q) - y^* + J\Delta q)^T C J + 2\Delta q^T W \\
 0^T &= (J\Delta q - \tilde{e})^T C J + \Delta q^T W \\
 0 &= J^T C^T (J\Delta q - \tilde{e}) + W^T \Delta q \\
 0 &= -J^T C^T \tilde{e} + J^T C^T J \Delta q + W^T \Delta q \\
 J^T C^T \tilde{e} &= (J^T C^T J + W^T) \Delta q \\
 (J^T C^T J + W^T)^{-1} J^T C^T \tilde{e} &= \Delta q
 \end{aligned} \tag{2}$$

use the Woodbury identity:

$$\Delta q = W^{T^{-1}} J^T (J W^{T^{-1}} J^T + C^{T^{-1}})^{-1} \tilde{e} \tag{3}$$

with

$$\lim_{C \rightarrow \infty} \Delta q = W^{T^{-1}} J^T (J W^{T^{-1}} J^T + C^{T^{-1}})^{-1}$$

and  $W = I$  we get :

$$J^\# = J^T (J J^T + \epsilon I)^{-1} \quad (4)$$

that's the Moore-Penrose-inverse a.k.a. **pseudo-inverse!!!!**

Sweet!

Where are we?

- ▶ We can calculate a change in configuration for a task!

What do we need?

- ▶ Jacobians and error vectors?

- ▶ **Errors** are the difference of the current value in **target space** to the target in **target space**  
eg:  
Where do I want to put the finger minus where is my finger now.
- ▶ Jacobians are **local linearization** of the **task space** given infinitesimal changes in **configuration space**.  
eg:  
How does my finger move if I change the value of my elbow (and my wrist)

A Jacobian(-matrix) is the derivative of the **local linearization** of the forward kinematics of a **task**. eg:

How does my finger (**end-effector**) move if I change the value of my elbow (and my wrist).

Jacobians are dependant on the robots posture.

A Jacobian(-matrix) is the derivative of the **local linearization** of the forward kinematics of a **task**.

$$J = \frac{\delta}{\delta q} \phi(q) = \begin{pmatrix} \underbrace{\frac{\delta}{\delta q_1} \phi(q)_1}_{*1} & \underbrace{\frac{\delta}{\delta q_2} \phi(q)_1}_{*2} & \dots & \underbrace{\frac{\delta}{\delta q_n} \phi(q)_1}_{*3} \\ \frac{\delta}{\delta q_1} \phi(q)_2 & \frac{\delta}{\delta q_2} \phi(q)_2 & \dots & \frac{\delta}{\delta q_n} \phi(q)_2 \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta}{\delta q_1} \phi(q)_d & \frac{\delta}{\delta q_2} \phi(q)_d & \dots & \frac{\delta}{\delta q_n} \phi(q)_d \end{pmatrix} \quad (5)$$

- \*1 : Where goes the endeffector when we change the first joint
- \*2 : Where goes the endeffector when we change the second joint
- \*3 : Where goes the endeffector when we change the n-th joint



- ▶ Jacobians are  $n \times d$ -matrixes
  - ▶ one **column** for each **joint**
  - ▶ one **row** for each **target-space** dimension

```
class RotationJoint:
    def getLocationDerivative(self, location):
        return np.matrix([[−location[1, 0]], [location[0, 0]])

    def getOrientationDerivative(self, orientation):
        return np.matrix([[−orientation[1, 0]], [orientation[0, 0]])

class PistonJoint:
    def getLocationDerivative(self, location):
        return np.matrix([[1],[0]])

    def getOrientationDerivative(self, orientation):
        return np.matrix([[0],[0]])
```

```
class LocationTask(PathedTask):
    def __init__(self, robot, path):
        PathedTask.__init__(self, robot, path)

    def getJacobian_(self):
        transform = np.matrix(np.eye(3, 3))
        jacobian = np.matrix(np.zeros((2, self.dof)))
        for node, direction in self.path.path:
            subtransform = np.matrix(np.eye(2, 2))
            if node.numDOF != 0:
                idx = self.robot.getIndexOfActiveNode(node.name)
                if direction == Direction.FROM_CHILD:
                    jacobian[:,idx] = node.getLocationDerivative(transform[0:2,2])
                elif direction == Direction.FROM_PARENT:
                    jacobian[:,idx] = -node.getLocationDerivative(transform[0:2,2])
            if direction == Direction.FROM_CHILD:
                subtransform = node.getTransform()[0:2,0:2]
                transform = node.getTransform() * transform
            elif direction == Direction.FROM_PARENT or direction == Direction.LINK:
                subtransform = node.getBackTransform()[0:2,0:2]
                transform = transform * node.getBackTransform()
            jacobian = subtransform * jacobian
        return jacobian

    def getError(self):
        return self.method.getTarget() - self.method.transform(self.getCurrentValue())
```

## Demo

| python linearIK1.py

```
def solve_simple(robot, task, epsilon=0):  
    helper = J * J.transpose()  
    J_pinv = J.transpose() * inv(helper * epsilon * np.matrix(np.eye(helper.shape[0])))  
    dq = J_pinv * task.getError()  
    return dq
```

But wait there is more!

- ▶ what if we want to do several things simultaneously?

Simultaneous tasks:

- ▶ "stack" the jacobians and the error vectors into a **big**-jacobian and a **big**-error

## Demo

| python linearIK2.py



```
def solve_simple(robot, tasks, epsilon=0):  
    numCols = robot.getDOF()  
    bigJacobian = np.matrix(np.zeros((0, numCols)))  
    bigError = np.matrix(np.zeros((0, 1)))  
    for task in tasks:  
        bigJacobian = np.concatenate((bigJacobian, task.getJacobian()))  
        bigError = np.concatenate((bigError, task.getError()))  
    helper = bigJacobian * bigJacobian.transpose()  
    J_pinv = bigJacobian.transpose() * inv(helper * epsilon * np.matrix(np.eye(helper.shape[0])))  
    dq = J_pinv * bigError  
    return dq
```

But wait there is even more!!!

- ▶ Tasks might not fully define the robot
  - ▶ For the remaining degrees of freedom we want to solve other tasks
  - ▶ Those other tasks must not interfere with the primary tasks!

- ▶ when  $\text{rank}(J) \neq \text{numDOF}$ :  
 $J$  does not fully utilize the robot

$$J^\# J \tag{6}$$

is the (orthogonal) range-projector of  $J$

$$I - J^\# J = \mathcal{N} \tag{7}$$

is the (orthogonal) **nullspace-projector** of  $J$

- ▶ any vector we multiply with  $\mathcal{N}$  will be within the **nullspace** of  $J$

This is what we are looking for:  $\mathbf{0} = J\mathcal{N}\vec{v}$

Utilizing the  $\mathcal{N}$ -matrix:

$$\begin{aligned}\Delta q &= \Delta q_1 + \Delta q_2 \\ \Delta q_1 &= J_1^\# \tilde{e}_1 \\ \Delta q_2 &= \mathcal{N}_1 J_2^\# \tilde{e}_2\end{aligned}\tag{8}$$

This is pretty much exactly what you'll find in textbooks

## Slightly Better Simple Solver

```
def solve_simple(robot, tasks, epsilon=0):
    numCols = robot.getDOF()
    dq = np.matrix(np.zeros((numCols, 1)))
    Ny = np.matrix(np.eye(numCols, numCols))
    reallyBigJacobian = np.matrix(np.zeros((0, numCols)))
    for taskGroup in taskGroups:
        bigJacobian = np.matrix(np.zeros((0, numCols)))
        bigError = np.matrix(np.zeros((0, 1)))
        for task in taskGroup:
            bigJacobian = np.concatenate((bigJacobian, task.getJacobian()))
            bigError = np.concatenate((bigError, task.getError()))
        jacobian_pinv = pinv(bigJacobian, epsilon)
        dq += Ny * jacobian_pinv * bigError
        reallyBigJacobian = np.concatenate((reallyBigJacobian, bigJacobian))
        Ny = np.matrix(np.eye(numCols, numCols)) - pinv(reallyBigJacobian, nullspaceEpsilon) *
            reallyBigJacobian
    return dq
```

## Demo

```
| python linearIK3.py  
| python linearIK3_bad.py
```

# What Happened?

What happened?

- The calculation for  $\Delta q_2$  does not yield optimal solutions!

$$J_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$J_1^\# = J_1 \tag{9}$$

$$\mathcal{N}_1 = I - J_1^\# J_1 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



# What Happened?

What happened?

- ▶ The calculation for  $\Delta q_2$  does not yield optimal solutions!
- ▶ Looks like we're utilizing the  $\mathcal{N}$ -matrix incorrectly!

$$\begin{aligned} \Delta q_2 &= \mathcal{N}_2^\# \vec{e}_2 \\ &= \underbrace{\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{this eats } \Delta q_{2_1} \text{ and } \Delta q_{2_2}} \begin{pmatrix} .5 & 0 \\ 0 & .5 \\ .5 & 0 \\ 0 & .5 \end{pmatrix} \vec{e}_2 \end{aligned} \quad (10)$$

Lets look again at:

$$\mathcal{L}(\Delta q) = \|\phi(q + \Delta q) - y^*\|_C^2 + \|\Delta q\|_W^2 \quad (11)$$

$$\Delta q = W^{T^{-1}} J^T (J W^{T^{-1}} J^T + C^{T^{-1}})^{-1} \tilde{e} \quad (12)$$

- ▶  $C$  weights the error in task-space (which is important)
- ▶  $W$  weights the error in joint-space
  - ▶ we need a matrix that punishes usage of joints that are already utilized
  - ▶  $W = \mathcal{N}^{-1}$  does exactly what we need!

$$\Delta \mathbf{q}_i = \mathcal{N}_i \mathbf{J}_i^T (\mathbf{J}_i \mathcal{N}_i \mathbf{J}_i^T + \epsilon \mathbf{I})^{-1} \vec{\mathbf{e}}_i \quad (13)$$

with:

$$\hat{\mathbf{J}}_{0,i-1} = \begin{pmatrix} \left. \begin{matrix} J_{0,0} \\ \vdots \\ J_{0,0n} \end{matrix} \right\} & \text{jacobians from task-group 0} \\ \left. \begin{matrix} J_{1,0} \\ \vdots \\ J_{1,0n} \end{matrix} \right\} & \text{jacobians from task-group 1} \\ \vdots & \\ \left. \begin{matrix} J_{i-1,0} \\ \vdots \\ J_{i-1,i-1n} \end{matrix} \right\} & \text{jacobians from task-group i-1} \end{pmatrix} \quad (14)$$

$$\mathcal{N}_i = \mathbf{I} - \hat{\mathbf{J}}_{0,i-1}^{\#} \hat{\mathbf{J}}_{0,i-1}$$

With  $\mathcal{N}$  as  $W^{-1}$  we get:

$$\begin{aligned}\Delta q_2 &= \mathcal{N} j_{2, \mathcal{N}_1}^{\#} \vec{e}_2 \\ &= \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \vec{e}_2\end{aligned}\tag{15}$$

## Demo

```
| python linearIK4.py  
| python linearIK4_better.py
```

But wait there is even more again!!!!

- ▶ The robot still converges weirdly
  - ▶ We calculate  $\Delta q$  as

$$\Delta q = \sum_i^N \Delta q_i \quad (16)$$

- ▶ But we could already know  $q$  when we calculate  $\Delta q_i$ !

$$q_i = q_{i-1} + \Delta q_i \quad (17)$$

read this as:

- ▶  $q_i$  is the robots posture after applying  $\Delta q_i$




## Demo

```
| python linearIK5.py  
| python nice_robot.py
```

This presentation and the code can be found here:





-  **Marc Toussaint**  
*Robotics Course*  
online
-  **Michael Gienger and Marc Toussaint and Christian Goerick**  
*Whole-body Motion Planning – Building Blocks for Intelligent Motion Planning for Humanoid Robots*  
online
-  **Gunüter Schreiber, Christian Ott, Gerd Hirzinger**  
Interactive Redundant Robotics: Control of the Inverted Pendulum with Nullspace Motion