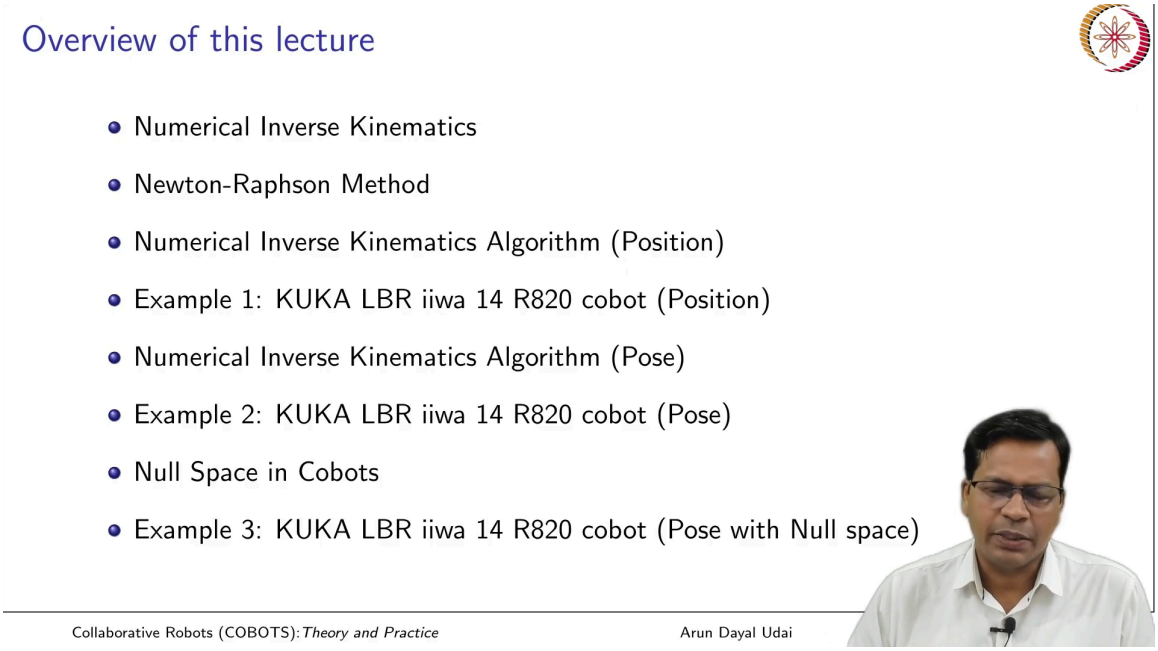
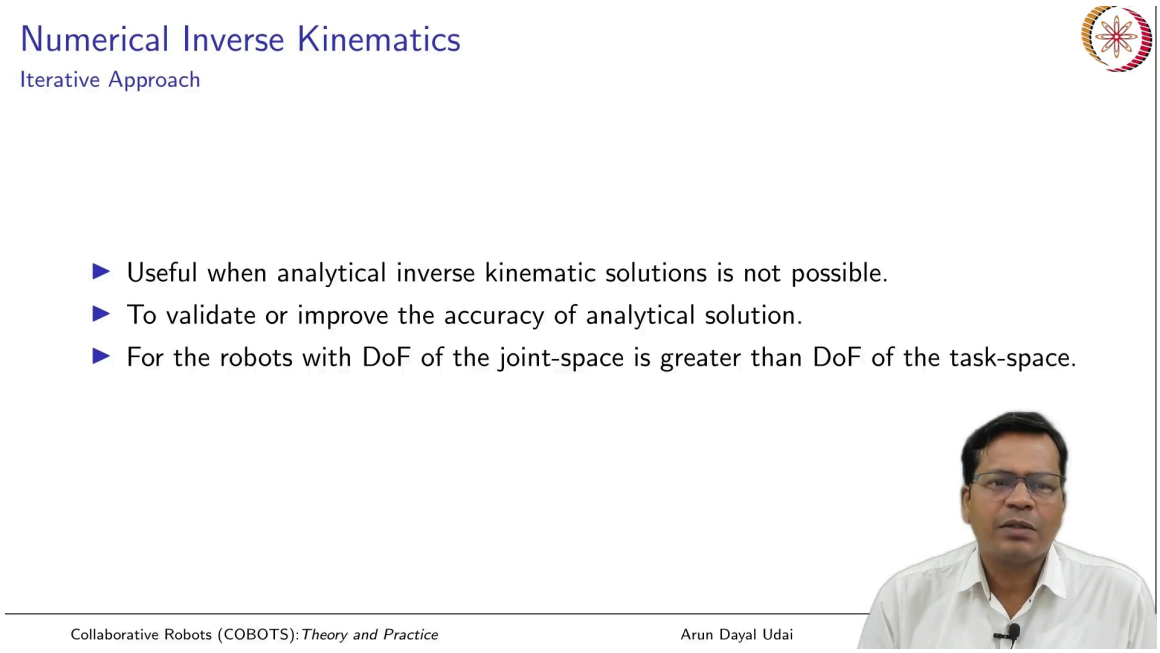
NPTEL Online Certification Courses
COLLABORATIVE ROBOTS (COBOTS): THEORY AND PRACTICE
Dr Arun Dayal Udai
Department of Mechanical Engineering
Indian Institute of Technology (ISM) Dhanbad
Week: 04
Lecture: 19

Inverse Kinematics of 7-DoF KUKA LBR iiwa Robot and Null Space



Welcome to the course Collaborative Robots: Theory and Practice, Lecture 5, which is Inverse Kinematics of 7 Degrees of Freedom, KUKA LBR iiwa Robot, and Null Space. So, in this lecture, we will discuss Numerical Methods to perform Inverse Kinematics. We will explore what Newton-Raphson's method is and how it can be used to perform inverse kinematics for a 7 degrees of freedom robot. We will examine a Numerical Inverse Kinematics Algorithm to perform position inverse kinematics using an example, that is, Example 1, the KUKA LBR iiwa R820 cobot for position inverse kinematics. We will explore the inverse kinematics algorithm for pose analysis, where 'pose' refers to position and orientation. I will demonstrate this using Example 2, which involves the KUKA LBR iiwa 14 R820 cobot for pose analysis. Additionally, we will discuss null

space in cobots and demonstrate it using Example 3, again involving the KUKA LBR iiwa robot. It will be for Pose analysis with Null Space. So, let us begin.



Numerical Inverse Kinematics. Why should we go for it? Why is this an iterative approach? Why should one choose an iterative approach to perform inverse kinematics? This is very useful when an analytical inverse kinematics solution is not possible. The closed-form analytical solution, which I demonstrated in my previous lectures, was done for the UR5e arm. However, the same is not possible if the robot has more than 6 degrees of freedom. In such cases, an analytical inverse kinematics solution is not possible. To validate or improve the accuracy of the analytical solution. Sometimes, the analytical solution does not lead to a very accurate result. In those cases, using an iterative approach for inverse kinematics can bring us much closer to the desired pose that we want. This is also applicable for robots where the degrees of freedom in the joint space are greater than the degrees of freedom in the task space, as in the case of the KUKA LBR iiwa robot, which is a 7-degree-of-freedom robot. So, we will use that to demonstrate all the algorithms that I am going to cover in this lecture.

Recap: Newton-Raphson Method

An iterative numerical algorithm for finding the roots of a real-valued function.

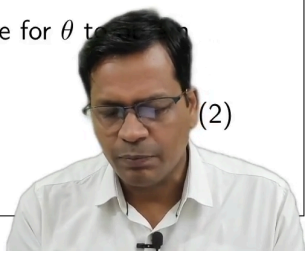
Equation to solve: $g(\theta) = 0$
Differentiable function $g : \mathbb{R} \to \mathbb{R}$
Initial guess or a nearby known solution: θ^0

Taylor series expansion of $g(\theta)$ at θ^0:

$$g(\theta) = g(\theta^0) + \frac{dg(\theta)}{d\theta}\Big|_{\theta=\theta^0}(\theta - \theta^0) + \text{higher-order terms} \cdots \quad (1)$$

Keeping only the terms up to first order, set the $g(\theta) = 0$ and solve for θ t

$$\theta = \theta^0 - \left(\frac{dg(\theta)}{d\theta}\Big|_{\theta=\theta^0}\right)^{-1} g(\theta^0) \quad (2)$$



So, let us have a quick recap on Newton's Raphson Method. This is a fundamental mathematical concept to understand this algorithm. This is an iterative numerical algorithm for finding the roots of a real-valued function. Let us take an equation that needs to be solved, like g(theta) is equal to 0, and I assume it is a differentiable function with real input and real output. Initial guess, let us say it is theta0, or a nearby known solution is theta0.

Equation to solve: $g(\theta) = 0$
Differentiable function $g : \mathbb{R} \to \mathbb{R}$
Initial guess or a nearby known solution: θ^0

So, these are the conditions to begin Newton's iteration approach. So, using Taylor's expansion, I can write g(theta) with theta0, theta0 that is the initial guess as this. This can be expanded like this.

$$g(\theta) = g(\theta^0) + \frac{dg(\theta)}{d\theta}\Big|_{\theta=\theta^0}(\theta - \theta^0) + \text{higher-order terms} \cdots \quad (1)$$

So, g(theta) is equal to g(theta0) plus the derivative of g(theta) at theta0, and this is the (theta minus theta0) plus higher-order terms there. So keeping only the first-order term and setting g(theta) equal to 0, and solving for theta to obtain this so that we can write the same equation here. We wrote g(theta) equal to 0, and this can be obtained.

$$\theta = \theta^0 - \left(\frac{dg(\theta)}{d\theta}\Big|_{\theta=\theta^0} \right)^{-1} g(\theta^0)$$

Recap: Newton-Raphson Method (cont.)

An iterative numerical algorithm for finding the roots of a real-valued function.

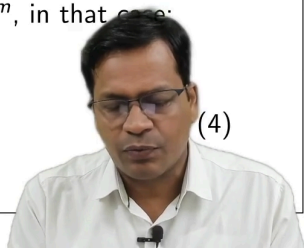
Using this value of θ as the new guess for the solution and repeating the Eq. (2), we get the following iteration:

$$\boxed{\theta^{k+1} = \theta^k - \left(\frac{dg(\theta)}{d\theta}\Big|_{\theta=\theta^k} \right)^{-1} g(\theta^k)} \quad (3)$$

Stopping criterion, e.g.,: $\frac{|g(\theta^k) - g(\theta^{k+1})|}{|g(\theta^k)|} \leq \epsilon$

This may be extended for a multidimensional function $g : \mathbb{R}^n \to \mathbb{R}^m$, in that case:

$$\frac{\partial g(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \begin{bmatrix} \frac{\partial g_1(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_1} & \cdots & \frac{\partial g_1(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_m(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_1} & \cdots & \frac{\partial g_m(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_n} \end{bmatrix} \in \mathbb{R}^{m \times n} \quad (4)$$



So now, using this value of theta as a new guess every time I obtain theta using this. I substitute that once again into the equation and obtain the next value of theta. So, this is the procedure of how the algorithm iterates. So, using this value of theta as a new guess for the solution and repeating equation 2 that we have seen here this equation, we will get the following. So, in this case, thetak plus 1, the new value of theta again based on the previous value of theta, will keep on moving towards the solution. The stopping criteria here will be the difference between the previous theta and the new theta if it goes below a certain value, that is, the stopping threshold decided before you begin the iteration. So, if that goes much below the required value, you stop the algorithm. Otherwise, it will keep iterating and keep improving.

This may be extended for a multidimensional real-valued function for Rn as input and Rm as output. In that case, the same equation, equation 2, can be written as this.

$$\frac{\partial g(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \begin{bmatrix} \frac{\partial g_1(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_1} & \cdots & \frac{\partial g_1(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_m(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_1} & \cdots & \frac{\partial g_m(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

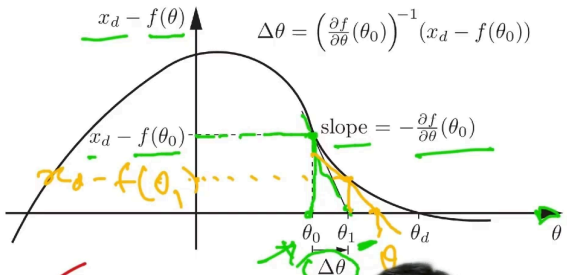
Numerical Inverse Kinematics Algorithm: For desired position vector only

Assume $\mathbf{x} = f(\boldsymbol{\theta})$ is the current position, where $\boldsymbol{\theta}$ is the vector of n joint coordinates.
$f : \mathbb{R}^n \to \mathbb{R}^m$ is differentiable (m is 2 or 3 for 2D or 3D, respectively.)

Let the desired end-effector position vector: $\mathbf{x}_d$

For Newton-Raphson method,
$g(\boldsymbol{\theta}) \equiv \mathbf{x}_d - f(\boldsymbol{\theta}_d) = 0$

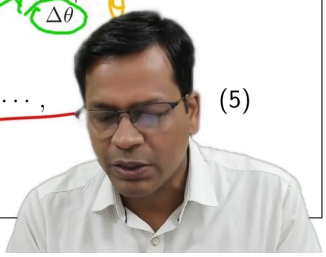
Taking the initial guess: $\boldsymbol{\theta}^0$
→ Usually the current joint position vector



This can be expressed as the Taylor expansion:

$$\mathbf{x}_d \equiv f(\boldsymbol{\theta}_d) = \underbrace{f(\boldsymbol{\theta}^0)}_{\mathbf{x}} + \underbrace{\frac{\partial f(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^0}}_{\mathbf{J}_p(\boldsymbol{\theta}^0)} \underbrace{(\boldsymbol{\theta}_d - \boldsymbol{\theta}^0)}_{\Delta\boldsymbol{\theta}} + \text{h.o.t.} \cdots, \quad (5)$$

where $\mathbf{J}_p(\boldsymbol{\theta}^0) \in \mathbb{R}^{m \times n}$ is the position Jacobian evaluated at $\boldsymbol{\theta}^0$.



So, let us see the numerical inverse kinematics algorithm for the desired position vector only. I assume x is equal to f theta, which is the current position, where theta is the vector of n joint coordinates for the given or current position. So, f is a real-valued function R^n, and Rn is differentiable. m is equal to 2 or 3, depending on whether we are solving for a 2-dimensional or 3-dimensional space. So, if it is a planar robot, it becomes 2, and if it is a 3-dimensional robot, it is 3. So, this is for the position vector only. So, it is 3. So, m is equal to 3, and n is the degrees of freedom of the robot. So, the number of joint angles will be m.

So, let the desired end effector position be x_d. This is the input to the inverse kinematics algorithm. This is where I want to go, and I want to find out the joint angle. So, for Newton-Raphson's method, I have written this equation with the desired one as g(theta) is equal to xd minus f(xd), and that should be equal to 0.

For Newton-Raphson method,
$g(\boldsymbol{\theta}) \equiv \mathbf{x}_d - f(\boldsymbol{\theta}_d) = 0$

So, this is my Newton-Raphson input equation.

Taking the initial guess as theta 0, I assume that is the initial guess. So, in most cases, the robot could be in a straight line, or it could have all angles set to 0. Let me just show you

the robot here. So, let us say this is my robot. With all the angles set as 0, it should look like this. So, you can start with this position or the current position. This is the KUKA iiwa robot. You see, it has seven degrees of freedom. This is the first degree of freedom. This is the first joint angle I am moving. This is the second one. This is the third one. This is the fourth one. This is the fifth one. This is the 6th, and finally, you see, this is the 7th. Let me bring it forward. So, this is your 7th joint. So, you see, this is your KUKA iiwa robot. So, usually, the current joint position is given as input so that going from the current position to the next position is a very short distance, and that becomes less computationally expensive. It also takes less time to calculate. So, instead of starting from the all-zero position every time, going to the next position would be too time-consuming. So, this can be expressed as a Taylor expansion like this,

$$\mathbf{x}_d \equiv f(\boldsymbol{\theta}_d) = \underbrace{f(\boldsymbol{\theta}^0)}_{\mathbf{x}} + \underbrace{\frac{\partial f(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^0}}_{\mathbf{J}_p(\boldsymbol{\theta}^0)} \underbrace{(\boldsymbol{\theta}_d - \boldsymbol{\theta}^0)}_{\Delta\boldsymbol{\theta}} + \text{h.o.t.} \cdots,$$

I just as I have followed the previous equation. So, xd is equal to f(theta d), and that is equal to f(theta 0) plus the derivative of this and the delta theta that is desired from the initial guess. The initial guess comes here, so that is the delta theta. Again, higher-order terms will come here, which I will neglect; where is J_p? What is J_p? It is the Jacobian evaluated at theta 0. If you remember what the Jacobian was, it is exactly this term. It is the del f theta by del theta, so this x actually is equal to the Jacobian of the robot, so that is evaluated at theta is equal to theta 0. So, in this case, it is the position Jacobian.

So, how does this Newton-Raphson approach work? You see, you have xd minus f theta that is plotted here, let's say, and this is your theta. So, for every value starting from theta 0, you start here, you go back, and project it to the curve that is here. So, the slope of that is something like this. So, it cuts the axis and gives you theta 1 over here. This is your delta theta. From there, you calculate the new value of this. So, xd minus f theta 0. So, you see, it has moved. So, again, it will be projected back onto the curve. You again, from here, you take the slope. You will reach somewhere over here, and again, you project back to calculate the xd minus f theta 1. And that is how you keep on moving theta 1. So, it is now theta 2. Next, theta 2 is calculated. So, that way, it keeps on moving, moving,

moving until it reaches theta d (Θ_d). That is the desired one, okay? And finally, you obtain and use theta d to get the actual position that was desired.

Numerical Inverse Kinematics Algorithm (cont.)

For desired position vector only

Truncating the Taylor series expansion at first order, we can approximate Eq. (5) as

$$\underbrace{\mathbf{x}_d - \mathbf{x}}_{\mathbf{e}_p} = \mathbf{J}_p(\boldsymbol{\theta}^0)\Delta\boldsymbol{\theta} \tag{6}$$

$$\Longrightarrow \Delta\boldsymbol{\theta} = \mathbf{J}_p^{-1}(\boldsymbol{\theta}^0)\mathbf{e}_p \tag{7}$$

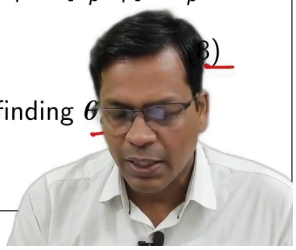
For $n > m$, and $\mathbf{J}_p$ becomes non-invertible. Eq. (6) can still be solved for $\Delta\boldsymbol{\theta}$ by replacing $\mathbf{J}_p^{-1}$ in Eq. (7) with the **Moore-Penrose pseudoinverse**, $\mathbf{J}_p^{\dagger} \equiv [\mathbf{J}_p^T\mathbf{J}_p]^{-1}\mathbf{J}_p^T$.

$$\Longrightarrow \Delta\boldsymbol{\theta} = \mathbf{J}_p^{\dagger}(\boldsymbol{\theta}^0)\mathbf{e}_p \tag{8}$$

Eq. (8) suggests using the Newton-Raphson iterative algorithm for finding $\boldsymbol{\theta}$

$$\boxed{\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i + \mathbf{J}_p^{\dagger}(\boldsymbol{\theta}^i)\mathbf{e}_p}$$

Collaborative Robots (COBOTS): *Theory and Practice* Arun Dayal Udai



So now, moving ahead with this equation, Truncating the Taylor series at the first-order term only, I have neglected the higher-order terms.

$$\underbrace{\mathbf{x}_d - \mathbf{x}}_{\mathbf{e}_p} = \mathbf{J}_p(\boldsymbol{\theta}^0)\Delta\boldsymbol{\theta}$$

We can approximate this equation to this: xd minus x is equal to Jp, that is, the Jacobian for position, and delta theta is theta d minus theta current. So, the same applies to the Cartesian position as well. So, this equation we all know very well, okay?

$$\Longrightarrow \Delta\boldsymbol{\theta} = \mathbf{J}_p^{-1}(\boldsymbol{\theta}^0)\mathbf{e}_p$$

Delta theta is equal to Jp inverse ep; that is the error in position, okay? This is what I call the error in position. We are already very familiar with this equation because this is the definition of the Jacobian. Also, you can directly get to this equation, okay?

So, using this for n greater than m, that is, the number of degrees of freedom of the robot in the input joint space is greater than the Cartesian space that it can attain. In the case of

a 2-dimensional robot, it is 2; in the case of a 3D robot for pose analysis, it is 6; for position only, it is 3, and Jp becomes non-invertible. So, if n is greater than m, it becomes non-invertible. So, equation 6 can still be solved for delta theta by replacing the Jp inverse with the Moore-Penrose pseudoinverse that we have already established earlier. Penrose pseudoinverse that we have already established earlier. So, that is like this.

$$\mathbf{J}_p^\dagger \equiv [\mathbf{J}_p^T \mathbf{J}_p]^{-1} \mathbf{J}_p^T$$

$$\implies \Delta\boldsymbol{\theta} = \mathbf{J}_p^\dagger(\boldsymbol{\theta}^0)\mathbf{e}_p$$

So, delta theta is equal to, instead of Jp inverse, I can write it as Jp dagger, which is equal to the Moore-Penrose pseudo-inverse, and the equation remains the same. Only the Jp inverse term is now calculated differently.

So, now equation 8 suggests the Newton-Raphson iterative algorithm for finding theta d as -

$$\boxed{\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i + \mathbf{J}_p^\dagger(\boldsymbol{\theta}^i)\mathbf{e}_p}$$

This time, you have theta i plus 1 is equal to theta i. So, this is the way it will keep on improving upon the new value of theta for the given input x. e_p is the error in position, and this is your pseudo inverse. Got it.

Numerical Inverse Kinematics Algorithm (cont.)

For desired position vector only

Algorithm 1 Numerical Inverse Kinematics - For position only

Input: Desired end-effector position vector ($\mathbf{x}_d$), Current joint position vector (θ^0), End-effector position error tolerance (ϵ_p)

Output: Desired joint position vector (θ_d)

1: $\mathbf{x} = \mathbf{T}_n^0|_{\theta=\theta^0}(1:3,4)$ % $\mathbf{T}_n^0$ - HTM of frame n as seen from base frame
2: $\mathbf{e}_p = \mathbf{x}_d - \mathbf{x}$
3: $i = 0$
4: **while** $\|\mathbf{e}_p\| > \epsilon_p$ **do**
5: $\quad \theta^{i+1} = \theta^i + \mathbf{J}_p^\dagger(\theta^i)\mathbf{e}_p$
6: $\quad \theta^i = \theta^{i+1}$
7: $\quad \mathbf{x} = \mathbf{T}_n^0|_{\theta=\theta^i}(1:3,4)$
8: $\quad \mathbf{e}_p = \mathbf{x}_d - \mathbf{x}$
9: $\quad i = i + 1$
10: **end while**
11: $\theta_d = \theta^{i+1}$

So, the numerical algorithm for this would be for position only. So, you see what we have done here. So, I have just extracted the value of the Cartesian position using the transformation matrix, homogeneous transformation matrix. I have just calculated so because you are inputting the initial guess position, so you have already put the current joint position vector. So, using that, do the forward kinematics obtain the HTM. From HTM, you can directly obtain the value of the end effector position, that is the current position. So, ep, that is the error, is calculated as xd minus x, starting from initial value i is equal to 0, and every time you keep on iterating till the value reaches very near to the actual value.

$$e_p = x_d - x$$

So, this is the ep. So, the magnitude of ep, error in position, becomes very near to the desired value. So, in that case, you have to stop this iteration. So, you keep on moving; use the equation as we have done here. I have used this equation again. So, that comes here, and you keep on increasing the next value of theta like this. And every time you calculate the new value of x. So, this was the starting value of x, the new value of x, and calculate the error, Cartesian error, increase i. And keep doing this whole loop till the error becomes very, very less. So, that is also position error tolerance that is set as an input before we begin this iteration. So, in the end, when this whole converges, and we

obtain the solution, theta d desired value is obtained, and that is stored here. This is what this function is going to return.

$$\Theta_d = \Theta^{i+1}$$

Example 1: KUKA LBR iiwa 14 R820 cobot

For desired position vector only

Assuming the Cobot is initially at the home configuration.

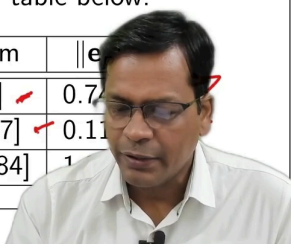
Input:

- Desired end-effector position $\mathbf{x}_d = [0.526, 0, 0.78]^T$
- Joint angles at initial position $\boldsymbol{\theta}^0 = [0, 0, 0, 0, 0, 0, 0]^T$
- Stopping criteria: $\epsilon_p = 10^{-3}$

Joint-space DoF, $n = 7$ and for 3D position vector, $m = 3$; $\implies \mathbf{J}_p(\boldsymbol{\theta}) \in \mathbb{R}^{3\times 7}$

Using **Algorithm 1**, the progress of the iteration is illustrated in the table below.

i	$\boldsymbol{\theta}^i = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \theta_7]$ deg	$\mathbf{x} = [x, y, z]$ m	$\|\mathbf{e}$
0	[0, 0, 0, 0, 0, 0, 0]	[0, 0, 1.3060]	0.7
1	[0, -24.009, 0, 13.35, 0, -3.198, 0]	[0.496, 0, 1.157]	0.1
2	[0, 126.398, 0, -75.203, 0, -101.176, 0]	[-0.315, 0, -0.284]	1



Collaborative Robots (COBOTS): *Theory and Practice* Arun Dayal Udai

So, here is example 1, when I have demonstrated the KUKA LBR iiwa robot, 14 kg payload, and reach is 820. It is a cobot for position vector only. So, I have done only position analysis. So, I will be using this algorithm, which is here. So, in this case, the desired end effector position is the input; this is what I want, and I want to find out the joint angles for that. I have started from the initial position when all the joint angles are zero. I showed you it is vertically up, exactly straight, okay?

So, this is the stopping criteria when the error I want is not more than 10 to the power of minus 3. So, in joint space, degrees of freedom n is equal to 7 for KUKA iiwa; it is a 7-degree-of-freedom robot. And because I am doing only position analysis, it is in 3D position vector m is equal to 3. So, the Jacobian, in this case, would be 3 cross 7. So, the algorithm progresses; it is illustrated here in the table. So, I started with this, okay?

$$\text{Joint angles at initial position } \boldsymbol{\theta}^0 = [0, 0, 0, 0, 0, 0, 0]^T$$

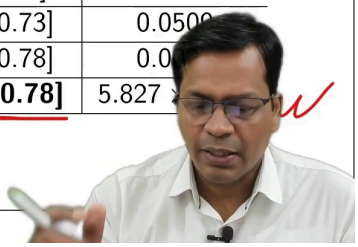
I obtained the value of x and calculated the error. Again, I have moved ahead, okay? The new value of theta k will be calculated. If you look carefully here, so every time you have reached a place, the next value of theta k will be calculated using this equation, and this is what is followed here. So, the new value of theta 1 is calculated using theta 0, and the new value of x is there. So, calculated the error, current and desired. So, you need to keep on calculating the error, error, error. So, it keeps on moving, moving, and ultimately, after 11 iterations, it finally converges to the desired value of x.

Example 1: KUKA LBR iiwa 14 R820 cobot (cont.)

For desired position vector only

i	$\boldsymbol{\theta}^i = [\theta_1,\ \theta_2,\ \theta_3,\ \theta_4,\ \theta_5,\ \theta_6,\ \theta_7]$ deg	$\mathbf{x} = [x,\ y,\ z]$ m	$\|\mathbf{e}_p\|$
3	[0, 28.292, 0, -8.129, 0, 71.225, 0]	[-0.557, 0, 1.014]	1.1075
4	[0, -119.778, 0, -172.21, 0, 20.455, 0]	[-0.073, 0, 0.432]	0.6925
5	[0, -26.33, 0, -70.491, 0, 26.01, 0]	[-0.211, 0, 1.066]	0.7905
6	[0, -127.274, 0, -153.444, 0, 65.956, 0]	[0.032, 0, 0.46]	0.5887
7	[0, -160.05, 0, -82.991, 0, 81.187, 0]	[0.524, 0, 0.181]	0.5996
8	[0, -118.852, 0, -116.516, 0, 81.748, 0]	[0.26, 0, 0.58]	0.3325
9	[0, -101.415, 0, -72.392, 0, 64.261, 0]	[0.533, 0, 0.73]	0.0500
10	[0, -95.625, 0, -69.167, 0, 61.581, 0]	[0.524, 0, 0.78]	0.0
11	**[0, -95.584, 0, -68.874, 0, 61.447, 0]**	**[0.526, 0, 0.78]**	5.827 ×

Hence, $\boldsymbol{\theta}_d = [0, -95.584, 0, -68.874, 0, 61.447, 0]^T$.



The error was 5.827 into 10 to the power of minus 6. So, this is the point when it should stop because it is much below the expected error, which I have set as a threshold for this algorithm to stop, and it has, you see, it has almost reached; it has reached the desired position. So, this is how it progresses.

$$\boldsymbol{\theta}_d = [0, -95.584, 0, -68.874, 0, 61.447, 0]^T$$

So, this is my solution for all the seven degrees of freedom. All seven joint angles are like this for the given position of the robot.

Example 1: KUKA LBR iiwa 14 R820 cobot (cont.)

For desired position vector only

Intuitively, the desired end-effector position (with a different orientation) can also be achieved by putting only $\theta_4 = 90^\circ$ while keeping all remaining joints at the home position.

Using forward kinematics:

Case - 1:
At $\boldsymbol{\theta}$ =[0, -95.584, 0, -68.874, 0, 61.447, 0]^T

$$\mathbf{T}_7^0 = \begin{bmatrix} 0.8218 & 0 & -0.5698 & 0.526 \\ 0 & 1 & 0 & 0 \\ 0.5698 & 0 & 0.8218 & 0.78 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

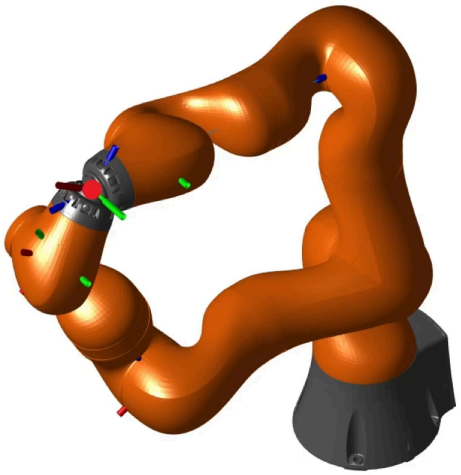
Case - 2:
At $\boldsymbol{\theta}$ =[0, 0, 0, 90, 0, 0, 0]^T

$$\mathbf{T}_7^0 = \begin{bmatrix} 0 & 0 & 1 & 0.5 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0.78 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Intuitively, the desired and affected position with different orientations can also be achieved. You know, this is a redundant robot, so it has multiple solutions. It can also be achieved by putting theta 4 equal to 90 degrees while keeping all the remaining joints at the home position. So, we just change one of the angles, and it reaches here. Because I had the datasheet, I know by I could just bend one of the joint angles, theta 4, by 90 degrees, and I can reach this position. So, it is x, this is y, this is z, and the orientation is something like this. So, I am not bothered about orientation here. So, the algorithm takes me to this, and this is the intuitive solution, and you see, in both cases, the orientations are different. So, this matrix is different.

Example 1: KUKA LBR iiwa 14 R820 cobot (cont.)

For desired position vector only



Although the end-effector achieved the desired position, the orientation is different in both the cases.

Hence, this algorithm is limited to inverse position kinematics only and do not have any provisi[illegible] orientation inputs.

So, this is how. So, this is my intuitive solution. Just one of the angles is set to 90 degrees, and this is the solution obtained by an iterative approach. Although the end effector achieved the desired position, the orientation is different in both cases. Hence, this algorithm is limited to inverse position kinematics only and does not have any provision for orientation inputs.

Numerical Inverse Kinematics Algorithm

For desired pose (position + orientation)

Forward Kinematics: $\mathbf{T} \equiv f(\boldsymbol{\theta}) = \begin{bmatrix} \mathbf{R} & \mathbf{x} \\ \mathbf{0} & 1 \end{bmatrix} \rightarrow$ current Homogeneous Transformation Matrix, where $\boldsymbol{\theta}$ is the vector of n joint coordinates.

Desired/Given end-effector pose, $\mathbf{T}_d = \begin{bmatrix} \mathbf{R}_d & \mathbf{x}_d \\ \mathbf{0} & 1 \end{bmatrix}$

Initial guess: $\boldsymbol{\theta}^0 \longrightarrow$ In robotics, it is usually the current joint position vector

Extracted desired end-effector position vector from HTM: $\mathbf{x}_d = \mathbf{T}_d(1:3,4)$
Extracted desired end-effector orientation matrix from HTM: $\mathbf{R}_d = \mathbf{T}_d(1:3,1:3)$

The position error vector: $\mathbf{e}_p$ and orientation error vector: $\mathbf{e}_\omega$
$\Rightarrow$ The error vector, $\mathbf{e} = [\mathbf{e}_\omega^T, \mathbf{e}_p^T]^T$

So, now let us look at the algorithm that does it for the desired pose, that is, position and orientation. This time, in the forward kinematics, I will be using both the position and the orientation. This is R, a 3x3 submatrix inside the transformation matrix, the

homogeneous transformation matrix, which gives you the sense of rotation, where theta is the vector of n joint coordinates.

So, the given or the desired end-effector pose is this. This is where I want to go, T_d. So, this again will have x desired (x_d) and r desired (R_d), position and orientation. This is where I want to go. This is the current one.

So, the initial guess I start with is theta 0 (Θ^0). In robotics, it is usually the current position vector, not all straight. From straight, if you start all 0, it takes too much time to converge. So, I start with where I am to the next point, the next point. So, iteratively, you can even feed the trajectory current position to the next position, next position, next position, and the robot keeps on going using inverse kinematics. So, this is my initial guess.

So, extracted desired end effector position vector from the homogeneous transformation matrix. This is HTM. So, I can take it from the transformation matrix. So, xd is this. And end effector orientation matrix, again it is a sub-matrix from the homogeneous transformation matrix; this one is here. So, the position error vector e_p and the orientation error vector e omega (e_ω) would be combined together now. So, e is the error vector will have e omega and e_p both. So, it is written as this.

$$\mathbf{e} = [\mathbf{e}_\omega^T, \mathbf{e}_p^T]^T$$

Numerical Inverse Kinematics Algorithm (cont.)

For desired pose (position + orientation)

End-effector position error vector

$$\boxed{\mathbf{e}_p = \mathbf{x}_d - \mathbf{x}} \longleftarrow \text{(Similar as earlier)}$$

End-effector orientation error vector

Let $\mathbf{R}_e$ be the required rotation to make the end-effector achieve the desired orientation $\mathbf{R}_d$ from the current orientation $\mathbf{R}$. Mathematically,

$$\mathbf{R}_e\mathbf{R} = \mathbf{R}_d$$
$$\implies \mathbf{R}_e = \mathbf{R}_d\mathbf{R}^T \longleftarrow \text{(Property: } \mathbf{R}^{-1} = \mathbf{R}^T\text{)}$$

$\mathbf{R}_e$ can be expressed in terms of an axis $\mathbf{r} = [r_x,\ r_y,\ r_z]^T$ and angle of rotation

$$\mathbf{R_e} = \begin{bmatrix} r_x^2V\phi + C\phi & r_xr_yV\phi - r_zS\phi & r_xr_zV\phi + r_yS\phi \\ r_xr_yV\phi + r_zS\phi & r_y^2V\phi + C\phi & r_yr_zV\phi - r_xS\phi \\ r_xr_zV\phi - r_yS\phi & r_yr_zV\phi + r_xS\phi & r_z^2V\phi + C\phi \end{bmatrix}$$

where, $V\phi = 1 - \cos\phi$

Collaborative Robots (COBOTS): *Theory and Practice* Arun Dayal Udai

So, now, the end effector position error vector is quite trivial. It is very easy, similar to the earlier algorithm.

$$e_p = x_d - x$$

So, the error vector is simply xd minus x, which is the error vector for position. So, the end effector orientation error vector would be: let R_e be the required orientation to make the end effector achieve the desired orientation from the current orientation R. So, mathematically, R_e is a rotation matrix which, when applied to the current orientation R, gives you R_d, the desired orientation. So, this is basically the orientation Re. So, the whole sense of R remains with Re. So, Re is Rd and R transpose. I have used the property Re. R inverse is R transpose, okay?

$$\mathbf{R}_e\mathbf{R} = \mathbf{R}_d$$
$$\implies \mathbf{R}_e = \mathbf{R}_d\mathbf{R}^T \longleftarrow \text{(Property: } \mathbf{R}^{-1} = \mathbf{R}^T\text{)}$$

So, that is what I have put here. You see, you are basically rotating about an axis, something like this, so that your current orientation from here goes to a new location, okay? So, you have rotated about some angle, so that rotation matrix is your Re. So, you were this has the sense of current R, and finally, you reach till R desired. So, from R to R

desired, you rotate about an axis using Re, and you reach till there. So, this is the geometrical sense of this, but that is not so trivial here.

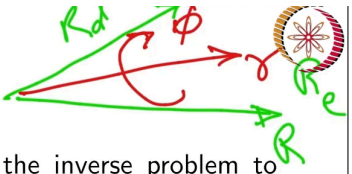
So, now R_e can be expressed as it is rotated about an axis R by an angle phi as this. So, this you remember, this is the general rotation matrix about an axis. The axis is given by r as this. This we have derived earlier also. So, V phi is versine phi, and it is 1 minus cos phi. So, this is it. So, re can be written like this.

$$\mathbf{R_e} = \begin{bmatrix} r_x^2 V\phi + C\phi & r_x r_y V\phi - r_z S\phi & r_x r_z V\phi + r_y S\phi \\ r_x r_y V\phi + r_z S\phi & r_y^2 V\phi + C\phi & r_y r_z V\phi - r_x S\phi \\ r_x r_z V\phi - r_y S\phi & r_y r_z V\phi + r_x S\phi & r_z^2 V\phi + C\phi \end{bmatrix}$$

where, $V\phi = 1 - \cos\phi$

Numerical Inverse Kinematics Algorithm (cont.)

For desired pose (position + orientation)



Given the rotation matrix $\mathbf{R_e} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \equiv \mathbf{R}_d\mathbf{R}^T$, the inverse problem to compute the axis $\mathbf{r}$ and angle ϕ may be obtained as:

$$\phi = cos^{-1}\left(\frac{r_{11} + r_{22} + r_{33} - 1}{2}\right) \quad (10)$$

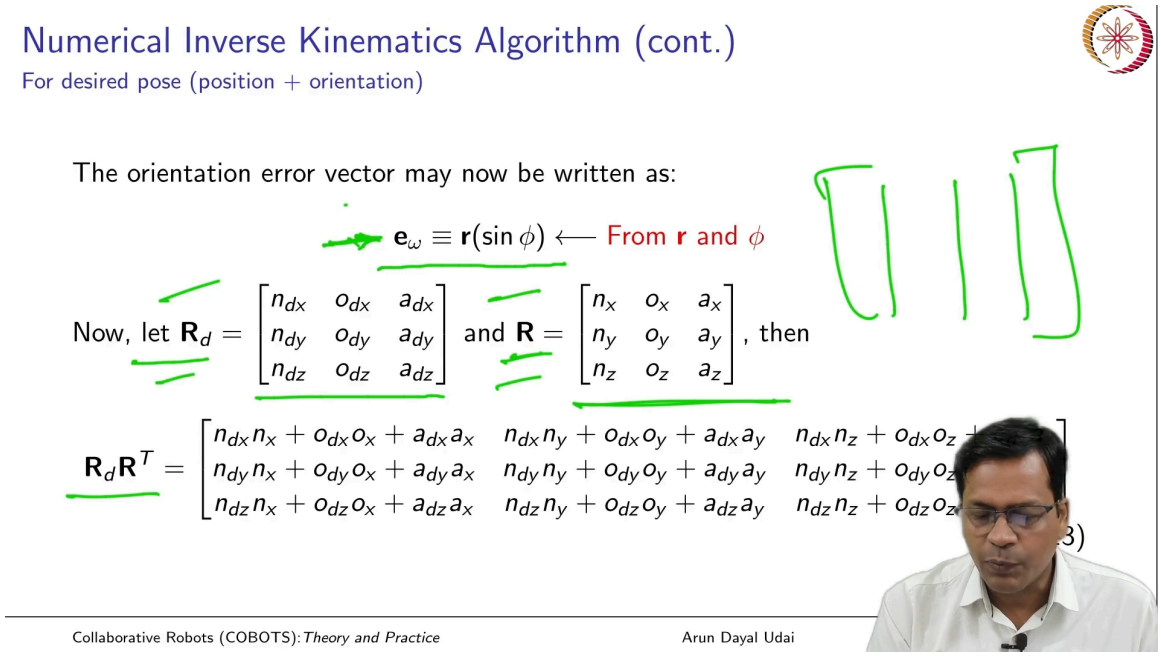
$$\mathbf{r} = \frac{1}{2\sin\phi}\begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \quad (11)$$

$$\implies \mathbf{r}\sin\phi \equiv \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} \sin\phi = \frac{1}{2}\begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \longrightarrow \mathbf{e}_\omega \quad)$$

Now, moving ahead. So, the given rotation, you know that the inverse solution for this also. So, given the rotation matrix Re, if this is already given and that is equivalent to this, we know where the inverse problem to compute the axis r and angle phi is this. So, phi is given by this, and r can be evaluated like this. This we have done earlier. So, it is exactly the same. So, r is equal to 1 by 2 sin phi is using the elements of this matrix; you can obtain this.

So, now I am rearranging this to make it very convenient to use further in the r algorithm. So, r sin phi is equal to This is what is here, so r sin phi, okay? r has components of rx, ry,

and rz, and that becomes equal to 1/2 of this. This has the sense of e omega (e_ω), the error orientation error, as I have shown here. So, you have an axis r about which you are rotating by an angle phi. This is r, so you started with here, and you reached till here. So, you started with this as your current, and you did something Re, and finally, you reached till R desired (R_d), okay? So, this is how it is happening. So, this is your e omega. So, e omega extracted out of this has a sense of error in omega, which actually takes you from R to Rd. This is what we have calculated here also.



So, the orientation error vector may now be written as e omega is equivalent to r sin phi because you know phi. Reducing phi using the Newton-Raphson scheme will keep on reducing phi, ultimately reducing e omega. So, that error is reduced. This is what the orientation error vector is.

$$e_\omega \equiv r(sin\phi)$$

So, now let us say R_d is equal to this, and R is equal to this. n is the normal orthogonal and approach vectors; you know this. So, any rotation matrix will have this form: n, o, and a club together. So, for Rd and R, I have kept it like this. Rd is the desired one, and R is the current one. So, if I use this R_dR^T transpose and do this calculation, I get to this directly, that is R_dR^T.

Numerical Inverse Kinematics Algorithm (cont.)

For desired pose (position + orientation)

Subtracting the element (2, 3) from the element (3, 2) of Eq. (13) gives

$$2r_x S\phi = (n_{dz}n_y - n_{dy}n_z) + (o_{dz}o_y - o_{dy}o_z) + (a_{dz}a_y - a_{dy}a_z)$$

Subtracting the element (3, 1) from the element (1, 3) of Eq. (13) gives

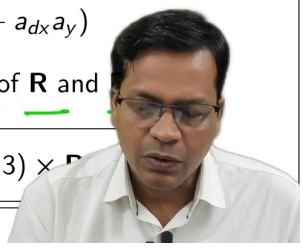
$$2r_y S\phi = (n_{dx}n_z - n_{dz}n_x) + (o_{dx}o_z - o_{dz}o_x) + (a_{dx}a_z - a_{dz}a_x)$$

Subtracting the element (1, 2) from the element (2, 1) of Eq. (13) gives

$$2r_z S\phi = (n_{dy}n_x - n_{dx}n_y) + (o_{dy}o_x - o_{dx}o_y) + (a_{dy}a_x - a_{dx}a_y)$$

Rearranging the above three expressions using vector-matrix forms of **R** and

$$\mathbf{e}_\omega \equiv \mathbf{r}(S\phi) = \frac{1}{2}\Big(\mathbf{R}(:,1) \times \mathbf{R}_d(:,1) + \mathbf{R}(:,2) \times \mathbf{R}_d(:,2) + \mathbf{R}(:,3) \times \mathbf{R}$$



Now, subtract the elements 2 and 3 from the elements 3 and 2. Why did I do it? You just check this equation. 3, 2 and 2, 3. Similarly, 1, 3 and 3, 1. 2, 1 and 1, 2. So that I can calculate exactly rx sin phi, ry sin phi, rz sin phi, and 1 by 2 will come to this side. So, if I do that, I can directly get 2 times rx sin phi, 2 times ry sin phi, and 2 times rz sin phi, and I can quickly obtain this.

So, again, rearranging the above three expressions using vector-matrix forms of R and Rd gives me this.

$$\mathbf{e}_\omega \equiv \mathbf{r}(S\phi) = \frac{1}{2}\Big(\mathbf{R}(:,1) \times \mathbf{R}_d(:,1) + \mathbf{R}(:,2) \times \mathbf{R}_d(:,2) + \mathbf{R}(:,3) \times \mathbf{R}_d(:,3)\Big)$$

You can do this exercise on your own. So, it is exactly using the matrices that you can obtain this. You can do this, and you can directly obtain this. So, this is what will directly give you r sin phi. This is all in vector.

Numerical Inverse Kinematics Algorithm (cont.)

For desired pose (position + orientation)

Similar to Eq. (8),

$$\Delta\boldsymbol{\theta} = \mathbf{J}^{\dagger}(\boldsymbol{\theta}^0)\mathbf{e} \tag{14}$$

where
$\mathbf{J}(\boldsymbol{\theta}^0) \in \mathbb{R}^{6\times n}$ is the body Jacobian evaluated at $\boldsymbol{\theta}^0$ and
$\mathbf{J}^{\dagger} \equiv [\mathbf{J}^T\mathbf{J}]^{-1}\mathbf{J}^T$ is the Moore-Penrose pseudoinverse of $\mathbf{J}$.

Using Eq. (14) the Newton-Raphson iterative algorithm for finding $\boldsymbol{\theta}_d$:

$$\boxed{\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i + \mathbf{J}^{\dagger}(\boldsymbol{\theta}^i)\mathbf{e}}$$

So, similar to equation 8, you see you have delta theta is equal to J inverse at theta 0, evaluated at theta 0, and this is an error (e).

$$\Delta\boldsymbol{\theta} = \mathbf{J}^{\dagger}(\boldsymbol{\theta}^0)\mathbf{e} \tag{14}$$

The error here is a composition of e omega and e position. So, both of them are clubbed together. So, J theta 0 is the body Jacobian evaluated at theta 0, and J inverse is a Moore-Penrose pseudo-inverse of J that is evaluated like this because it is m cross n.

So, using equation 14, the Newton-Raphson iterative algorithm for finding theta d would be this.

$$\boxed{\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i + \mathbf{J}^{\dagger}(\boldsymbol{\theta}^i)\mathbf{e}}$$

So, you keep on calculating the next value of theta using the previous one and the Jacobian inverse of error. So, this becomes my new equation. I will use this and do these calculations.

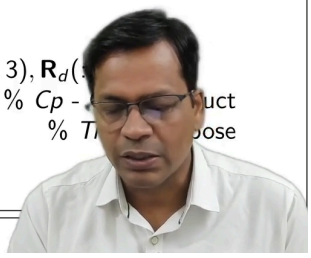
Numerical Inverse Kinematics Algorithm (cont.)

For desired pose (position + orientation)

Algorithm 2 Numerical Inverse Kinematics Algorithm - For pose

Input: Desired end-effector pose ($\mathbf{T}_d$), Current joint position vector (θ^0), End-effector position error tolerance (ϵ_p), End-effector orientation error tolerance (ϵ_ω)
Output: Desired joint position vector (θ_d)

1: **function** ERROR($\mathbf{T}$, $\mathbf{T}_d$)
2: $\mathbf{x} = \mathbf{T}(1:3, 4)$
3: $\mathbf{x}_d = \mathbf{T}_d(1:3, 4)$
4: $\mathbf{e}_p = \mathbf{x}_d - \mathbf{x}$
5: $\mathbf{R} = \mathbf{T}(1:3, 1:3)$
6: $\mathbf{R}_d = \mathbf{T}_d(1:3, 1:3)$
7: $\mathbf{e}_\omega = 0.5 * \left(Cp(\mathbf{R}(:,1), \mathbf{R}_d(:,1)) + Cp(\mathbf{R}(:,2), \mathbf{R}_d(:,2)) + Cp(\mathbf{R}(:,3), \mathbf{R}_d($ % Cp - uct
8: $\mathbf{e} = Tr([Tr(\mathbf{e}_\omega), Tr(\mathbf{e}_p)])$ % T ose
9: **return e**
10: **end function**



It is exactly similar to the previous one, but in this case, you also have e omega because you are doing both position and orientation analysis. So, the desired end effector pose is given that is input to this inverse kinematics algorithm, current joint position vector or 000 positions and a factor position tolerance where you want to stop your algorithm. This is tolerance in terms of orientation. So, both the inputs should be there. Yes, so you have again extracted x and xd and calculated ep directly and calculated e omega. I have used this equation this one,

$$\mathbf{e}_\omega \equiv \mathbf{r}(S\phi) = \frac{1}{2}\Big(\mathbf{R}(:,1) \times \mathbf{R}_d(:,1) + \mathbf{R}(:,2) \times \mathbf{R}_d(:,2) + \mathbf{R}(:,3) \times \mathbf{R}_d(:,3)\Big)$$

And that equation comes directly here. e omega is calculated like this, and total error, an error will be composed of e omega and ep transpose of that. cp is the cross-product that is used here. So, I'll return the error finally, st what is my error? I'll come to know also that becomes the stopping criteria also for this. So, and function, so this is the evaluating the error function, and that will be used here.

Numerical Inverse Kinematics Algorithm (cont.)

For desired pose (position + orientation)

Algorithm 2 Numerical Inverse Kinematics Algorithm - For pose (cont.)

11: $\mathbf{T} = \mathbf{T}_n^0(\boldsymbol{\theta}^0)$ % $\mathbf{T}_n^0$ - HTM of frame n as seen from base frame
12: $\mathbf{e} = \text{ERROR}(\mathbf{T}, \mathbf{T}_d)$
13: $i = 0$
14: **while** $\|\mathbf{e}(1:3)\| > \epsilon_\omega$ **or** $\|\mathbf{e}(4:6)\| > \epsilon_p$ **do**
15: $\quad \boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i + \mathbf{J}^\dagger(\boldsymbol{\theta}^i)\mathbf{e}$
16: $\quad \boldsymbol{\theta}^i = \boldsymbol{\theta}^{i+1}$
17: $\quad \mathbf{T} = \mathbf{T}_n^0(\boldsymbol{\theta}^i)$
18: $\quad \mathbf{e} = \text{ERROR}(\mathbf{T}, \mathbf{T}_d)$
19: $\quad i = i + 1$
20: **end while**
21: $\boldsymbol{\theta}_d = \boldsymbol{\theta}^{i+1}$

This error function is used here. This is the numerical inverse kinematics algorithm for position and orientation. In this case, the error is calculated starting from i is equal to zero. Now, the actual iteration begins with the threshold. So as long as the error comes down below e omega and e_p. So, epsilon omega and epsilon p, I will keep on moving it. e is the error.

So, the new value of thetas are calculated. You set the new value of theta then to the previous value, update with the new value of theta, and calculate the HTM Homogeneous Transformation Matrix, calculate the error again, and increase the value of i by 1 and keep on doing this iteration till the thresholds are achieved, and finally, you return theta d, theta d that is the desired theta d, you reach till there.

$$\Theta_d = \Theta^{i+1}$$

Example 2: KUKA LBR iiwa 14 R820 Cobot

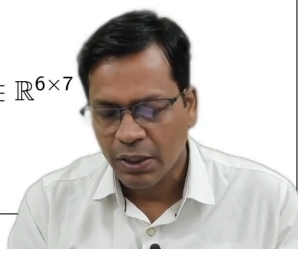
For desired pose (position + orientation)

Assuming the Cobot is initially at the home configuration.

Input:

- $\mathbf{T}_d = \begin{bmatrix} 0.997 & 0.046 & -0.063 & 0.358 \\ -0.078 & 0.527 & -0.847 & -0.484 \\ -0.006 & 0.849 & 0.528 & 0.825 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- $\boldsymbol{\theta}^0 = [0, 0, 0, 0, 0, 0, 0]^T$ (current joint position)
- $\epsilon_p = 10^{-3}$ and $\epsilon_\omega = 10^{-3}$

Joint-space DoF $(n) = 7$ and Task-space DoF $(m) = 6 \implies \mathbf{J}(\boldsymbol{\theta}) \in \mathbb{R}^{6\times 7}$



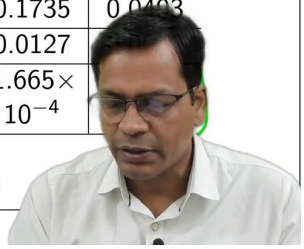
So, I have again used this algorithm to perform inverse kinematics analysis of the KUKA iiwa robot for position and orientation. I have assumed it is initially at the home position or 0, 0, 0. So, that is the reason the current joint position is taken as 0. T_d is equal to this, which is what I desire for the end effector position and orientation 0. So, these are the thresholds where I want the algorithm to stop iterating. 10 to the power of minus 3, 10 to the power of minus 3. In both position and orientation, corresponding units are taken. The joint space degrees of freedom is 7. Task space is 6. So, that is the reason J theta would be 6 cross 7.

Example 2: KUKA LBR iiwa 14 R820 Cobot (cont.)

For desired pose (position + orientation)

i	$\boldsymbol{\theta}^i = [\theta_1,\ \theta_2,\ \theta_3,\ \theta_4,\ \theta_5,\ \theta_6,\ \theta_7]$ deg	$\|\mathbf{e}_\omega\|$	$\|\mathbf{e}_p\|$
9	[-90.15, 173.72, -81.31, 88.83, 13.31, -88.82, 63.33]	0.9977	1.2300
10	[141.49, 173.41, 119.56, -49.52, -23.72, 70.29, -46.71]	0.8679	1.2435
11	[83.6, 3.74, 140.14, 0.12, -129.47, 46.03, -95.39]	0.1817	0.6550
12	[161.43, -45.16, -4.98, -96.09, 28.91, 133.89, 73.02]	0.4503	0.6152
13	[-146.89, 14.96, -51.73, -80.80, -59.39, 110.9, 8.85]	0.8667	0.3451
14	[-168.24, 11.33, -63.02, -84.34, -111.39, 56.02, -18.04]	0.6589	0.0939
15	[170.37, 27.41, -59.82, -74.54, -108.87, 34.38, 31.08]	0.1735	0.0493
16	[174.35, 26.27, -61.61, -73.56, -116.15, 40.76, 39.14]	0.0127	
17	[173.98, 26.64, -61.56, -73.01, -114.87, 40.73, 37.9]	1.665×10^{-4}	

Hence, $\boldsymbol{\theta}_d = [173.98,\ 26.64,\ -61.56,\ -73.01,\ -114.87,\ 40.73,\ 37.9]^T$.



So, you see, I have again started with 0, 0, 0 and I have kept on updating the new values of theta to this. So, it took a total of 17 iterations to converge to the solution, and finally, the threshold in error was achieved, and I gave I got the final theta d desired value. So, this is where I wanted to go, and this gives me directly.

Example 2: KUKA LBR iiwa 14 R820 Cobot (cont.)

For desired pose (position + orientation)

Validating using forward kinematics:

At $\boldsymbol{\theta}$ =[173.98, 26.64, -61.56, -73.01, -114.87, 40.73, 37.9]^T

$$\mathbf{T}_7^0 = \begin{bmatrix} 0.997 & 0.046 & -0.063 & 0.358 \\ -0.078 & 0.527 & -0.847 & -0.484 \\ -0.006 & 0.849 & 0.528 & 0.825 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$\mathbf{T}_7^0 = \mathbf{T}_d$ shows the desired pose has been achieved.

So, this can be verified using forward kinematics. If you put that, do forward kinematics,

you should get the input homogeneous transformation matrix. So, both are the same in my case. So, I got the desired pose, which was achieved. So, it looks like this.

$\mathbf{T}_7^0 = \mathbf{T}_d$ shows the desired pose has been achieved

Any random test you can do, not just this one.

Redundancy

Null space: Quick Recap

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ $(n > m)$ be a non-square matrix defining a system of linear equations as $\mathbf{Ax} = \mathbf{b}$.

The null space $\mathcal{N}(\mathcal{J})$ of $\mathbf{A}$ is the set of all $\mathbf{x}$ that, when multiplied by $\mathbf{A}$, results in a zero vector.

Mathematically,

$$\text{Null}(\mathbf{A}) = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{Ax} = \mathbf{0}\}.$$

- In robotics: $\mathbf{J}\dot{\boldsymbol{\theta}} = \mathbf{t}_e$.
- For redundant manipulators, $\mathbf{J}$ is a non-square matrix. In case of KUKA iiwa $\mathbf{J} \in \mathbb{R}^{6 \times 7}$
- There exists $\dot{\boldsymbol{\theta}}_0 \in \mathcal{N}(\mathcal{J})$ for which $\mathbf{J}\dot{\boldsymbol{\theta}}_0 = \mathbf{0}$.
- This means that the joint-space motion is not creating any motion

Collaborative Robots (COBOTS): *Theory and Practice* Arun Dayal Ud

Let us discuss what Null Space is. This is quite common in the case of redundant robots, as in the case of the KUKA iiwa. You have seen the number of angles; joint angles are 7. Whereas the output, you can only obtain 6 parameters, which are x, y, z, and roll, pitch, and yaw of the end effector. So, 6 in the output, 7 in the input. So, your Jacobian is no longer a square matrix. So, in those cases, mathematically, let us have a quick recap of what null space is.

So, let A be a real-valued matrix of m cross n, where n is greater than m. In our case, it was n for degrees of freedom, which is 7, and m was 6. It is a non-square matrix defining the system of linear equations as Ax equals b.

$$Ax = b$$

So, the null space of A is the set of all x, which, when multiplied by A, results in a zero vector.

$$\text{Null}(\mathbf{A}) = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{Ax} = \mathbf{0}\}$$

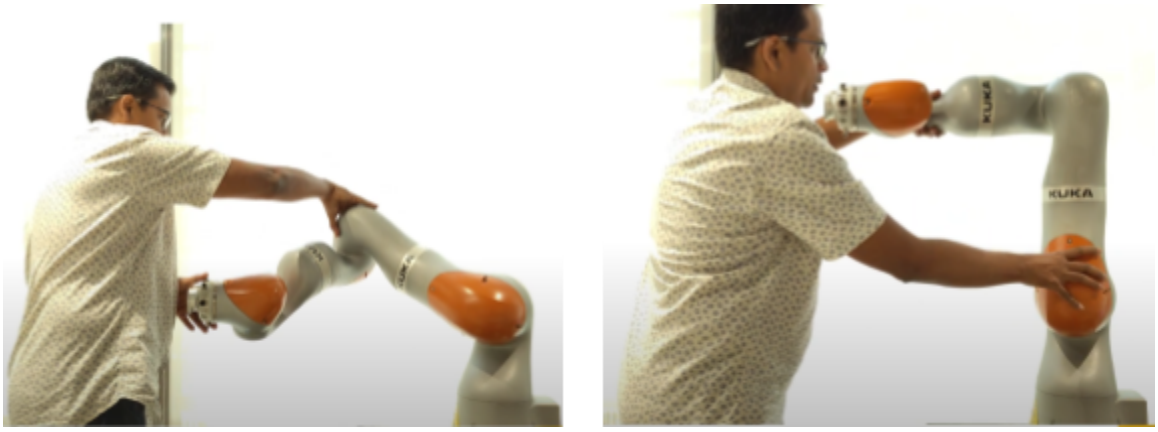
So, mathematically, the null space of A is the set of all x, which, when multiplied with A, results in a zero vector. So, it can be given as So, the null space of A is x, a real-valued vector, and Ax is equal to 0, such that A x is equal to 0.

So, in robotics, you have j theta dot is equal to twist. J is the Jacobian, theta dot is the joint rate, and t_e is the end effector velocities, angular as well as linear velocity, that is known as the twist.

$$J\dot{\Theta} = t_e$$

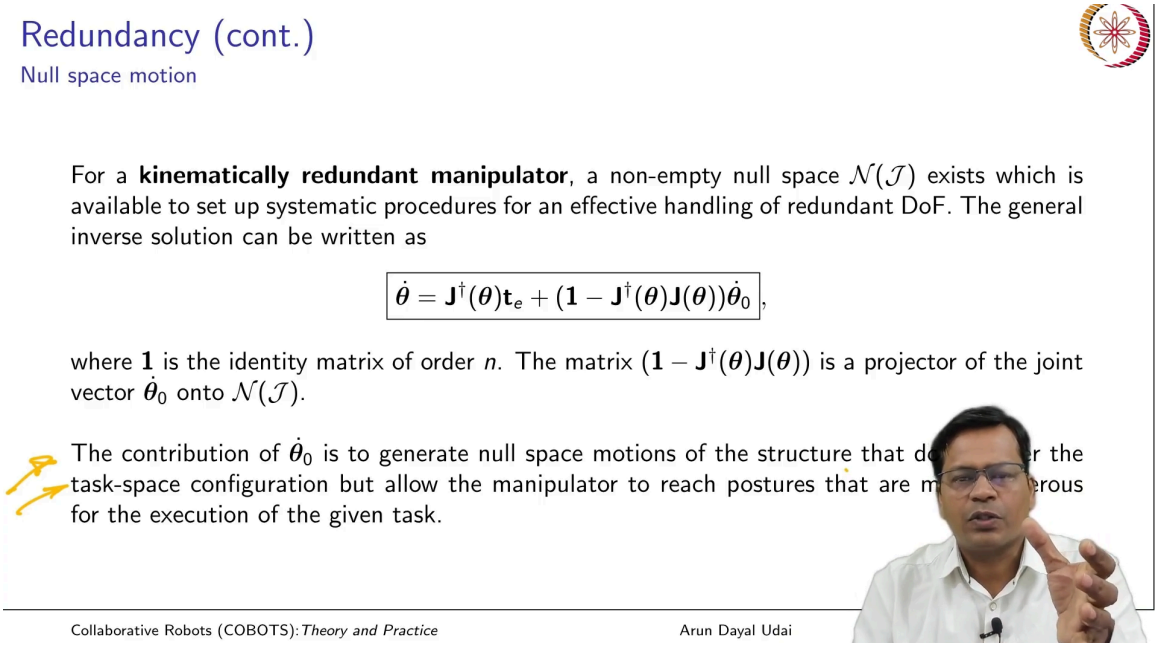
So, this has a similar form as this one, you see.

So, now, for a redundant manipulator, J is a non-square matrix. In the case of the KUKA iiwa robot, J is 6 by 7. So, there exists the matrix theta 0 dot so you will have a set of theta 0 dot for which the twist becomes equal to 0. So, you see, there is no motion at the end effector where the theta 0 dot still exists, for which J theta 0 dot is equal to 0. So this means that joint space motion is not creating any motion in the task space. So, in the task space, you don't see any motion, but the joints are moving. So, that is the case.



So, let me show you a small video here, which will clear it up very well, okay? That experiment that I have done, so this is the video. You see the whole of the robot is moving, but the end effector is stationary. There is no velocity at the end effector, okay?

So, this is a redundancy demonstration when you see the robot is moving in the null space.



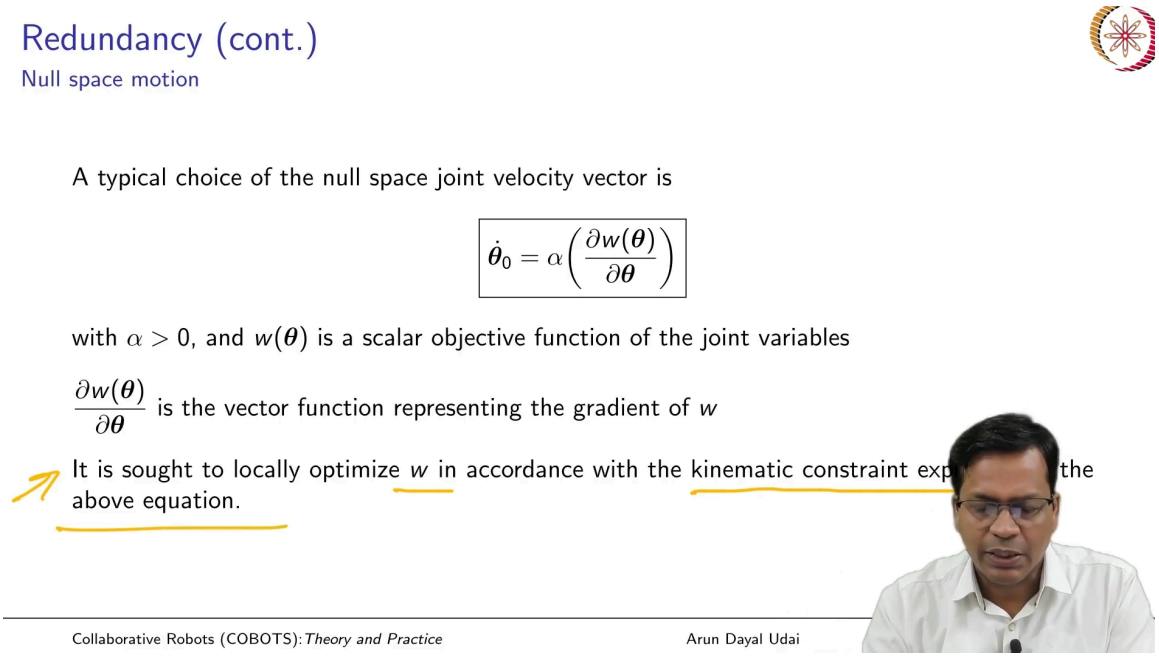
For null space motion for a kinematically redundant manipulator, a non-empty, non-null space exists, which is available to set up a systematic procedure for an effective handling of redundant degrees of freedom. The general inverse kinematic solution can now be written using the null space like this.

$$\dot{\theta} = \mathbf{J}^{\dagger}(\theta)\mathbf{t}_e + (\mathbf{1} - \mathbf{J}^{\dagger}(\theta)\mathbf{J}(\theta))\dot{\theta}_0$$

So, this was our earlier equation. So, apart from that, if we consider the null space also, this is my inverse kinematic solution. Mathematically, it is there where one vector is the identity matrix of order n. The matrix 1 minus J inverse J is a projector of the joint vector theta 0 dot onto the null space.

The contribution of theta0 dot is to generate null space motions of the structure that do not alter the task space configuration but allow the manipulator to reach postures that are more dexterous for the execution of a given task. So this is very useful. You still have the same position and orientation, but the robot can move, so that it may be very helpful to do some kind of dexterous manipulation. As in the case where we want to pick up

something from within a jar and the jar mouth is actually obstructing it. So, you can have multiple orientation combinations to actually get into the jar without directly obtaining the object that is inside the jar without directly obtaining it. You go from the top and get in. So, that is basically giving you a sense of a collision-free path or maybe those kinds of things.



So, typically, the choice of the null space velocity vector is given by this.

$$\dot{\boldsymbol{\theta}}_0 = \alpha\left(\frac{\partial w(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)$$

So, this is your theta 0 dot that is what was used here. This theta 0 dot is given as alpha times del w by del theta, where alpha is a number greater than 0 and w theta is a scalar objective function, an objective function of the joint variables, and del omega by del theta is a vector function representing the gradient of w. So, it is sought to locally optimise w in accordance with kinematic constraints, constraints by the above equation. So, based on different types of kinematic constraints, you can define theta 0 dots using this equation.

Redundancy (cont.)

Null space motion

Usual objective functions are:

- The manipulability measure defined as

$$w(\boldsymbol{\theta}) = \sqrt{\left|\left(\mathbf{J}(\boldsymbol{\theta})\mathbf{J}^T(\boldsymbol{\theta})\right)\right|},$$

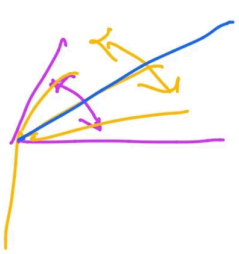
which vanishes at a singular configuration, and thus redundancy may be exploited to escape singularities.

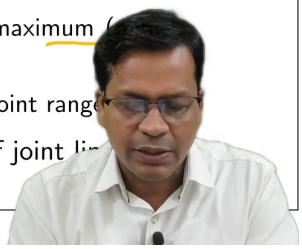
- The distance from mechanical joint limits defined as:

$$w(\boldsymbol{\theta}) = -\frac{1}{2n}\sum_{i=1}^{n}\left(\frac{\theta_i - \bar{\theta}_i}{\theta_{iM} - \theta_{im}}\right)^2,$$

  - θ_{iM} (θ_{im}) denotes the maximum (...mum) limit for θ_i
  - $\bar{\theta}_i$ is the middle of the joint rang...

thus, redundancy may be exploited to keep the manipulator off joint li...





So, let us see what could be the multiple objective functions. Usually, the objective functions are the manipulability measure as defined by JJ transpose square root. So, this is the determinant of JJ transpose, which is a number. Taking the square root, it is defined by the manipulability measure that we have seen earlier, which vanishes at a singular configuration. Thus, redundancy may be exploited to escape singularities. So, you see, the function w can now be used to escape the singularities if at all it is there in the path. If it is coming, you can use this function to define your theta 0 dot so as to define your robot's inverse kinematic solution and finally escape the singularity. So, this is one way to escape from singularity.

The next function could be the distance from mechanical joint limits. So, a robot joint always has some limits. It cannot go below this and above this, and it should operate within this range. So, I want to be somewhere in between all the time. My solution, I want, even though it has infinite solutions in between, but I want to be somewhere in between this so as to avoid any joint limiting.

The distance from the mechanical joint limit is defined as this:

$$w(\theta) = -\frac{1}{2n}\sum_{i=1}^{n}\left(\frac{\theta_i - \bar{\theta}_i}{\theta_{iM} - \theta_{im}}\right)^2$$

w theta is equal to minus 1 by 2 n, n being the degrees of freedom, summation of this, okay, where theta im denotes the maximum, inside one is the minimum limit of theta i. Theta i bar is the middle of the joint range. So, it is somewhere here, the middle of the joint range. So, that is used here. So, using that, it is defined as this. So, this is the function making it go somewhere within the joint limits, and thus redundancy may be exploited now to keep the manipulator within the joint limits.

Redundancy (cont.)

Null space motion

- The distance from an obstacle defined as

$$w(\theta) = \min_{\mathbf{p},\mathbf{o}} \|\mathbf{p}(\theta) - \mathbf{o}\|,$$

where $\mathbf{o}$ is the position vector of an opportune point on the obstacle and $\mathbf{p}(\theta)$ is the position vector of the closest manipulator point to the obstacle, and thus redundancy may be exploited to avoid collisions with obstacles.

So, again, there is another type of function that may be used. This is the distance from the obstacle, where the o vector here is the position vector of an opportune point on the obstacle, and p theta is the position vector of the closest manipulator point to the obstacle and thus, redundancy may now be exploited to avoid collision with any obstacle on the way. This is an external obstacle I am talking about, okay? So, these are a few functions that can be used.

Redundancy (cont.)

Null space motion

Usage of null space motion to avoid mechanical joint limits

Since $w(\boldsymbol{\theta}) = -\frac{1}{2n}\sum_{i=1}^{n}\left(\frac{\theta_i - \bar{\theta}_i}{\theta_{iM} - \theta_{im}}\right)^2$

$$\implies w(\boldsymbol{\theta}) = -\frac{1}{2n}\left(\left(\frac{\theta_1 - \bar{\theta}_1}{\theta_{1M} - \theta_{1m}}\right)^2 + \left(\frac{\theta_2 - \bar{\theta}_2}{\theta_{2M} - \theta_{2m}}\right)^2 + \cdots + \left(\frac{\theta_n - \bar{\theta}_n}{\theta_{nM} - \theta_{nm}}\right)^2\right)$$

Taking partial derivative with respect to each joint coordinate

$$\frac{\partial w(\boldsymbol{\theta})}{\partial \theta_1} = -\frac{1}{n}\left(\frac{\theta_1 - \bar{\theta}_1}{(\theta_{1M} - \theta_{1m})^2}\right), \frac{\partial w(\boldsymbol{\theta})}{\partial \theta_2} = -\frac{1}{n}\left(\frac{\theta_2 - \bar{\theta}_2}{(\theta_{2M} - \theta_{2m})^2}\right), \text{ and}$$

So, let us use one of them. So, as you have seen, the null space motion to avoid the mechanical joint limit is defined as this. So, it is defined as this. So, if I expand this summation, it brings you to this, and taking the partial derivative of this, I can now create the vector of del w by del theta 1, theta 2, theta 3, like that. So, this is your calculation for that. So, individual partial derivatives are written here.

Redundancy (cont.)

Null space motion

Arranging in vector form will result in

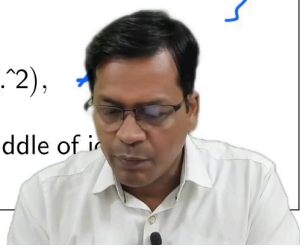
$$\begin{bmatrix} \frac{\partial w(\boldsymbol{\theta})}{\partial \theta_1} \\ \frac{\partial w(\boldsymbol{\theta})}{\partial \theta_2} \\ \vdots \\ \frac{\partial w(\boldsymbol{\theta})}{\partial \theta_n} \end{bmatrix} = -\frac{1}{n}\left(\begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} - \begin{bmatrix} \bar{\theta}_1 \\ \bar{\theta}_2 \\ \vdots \\ \bar{\theta}_n \end{bmatrix}\right) ./ \begin{bmatrix} (\theta_{1M} - \theta_{1m})^2 \\ (\theta_{2M} - \theta_{2m})^2 \\ \vdots \\ (\theta_{nM} - \theta_{nm})^2 \end{bmatrix} \longleftarrow (./ : \text{means element-wise division})$$

$$\implies \frac{\partial w(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = -\frac{1}{n}(\boldsymbol{\theta} - \bar{\boldsymbol{\theta}})./((\boldsymbol{\theta}_M - \boldsymbol{\theta}_m).\hat{}2) \longleftarrow (.\hat{}2 : \text{means element-wise square})$$

Then,

$$\implies \dot{\boldsymbol{\theta}}_0 = \alpha\left(\frac{\partial w(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right) = -\frac{\alpha}{n}(\boldsymbol{\theta} - \bar{\boldsymbol{\theta}})./((\boldsymbol{\theta}_M - \boldsymbol{\theta}_m).\hat{}2),$$

where $\boldsymbol{\theta}$, $\bar{\boldsymbol{\theta}}$, $\boldsymbol{\theta}_M$, and $\boldsymbol{\theta}_m$ are the vector of current joints position, middle of i joints maximum limit and joints minimum limit, respectively.

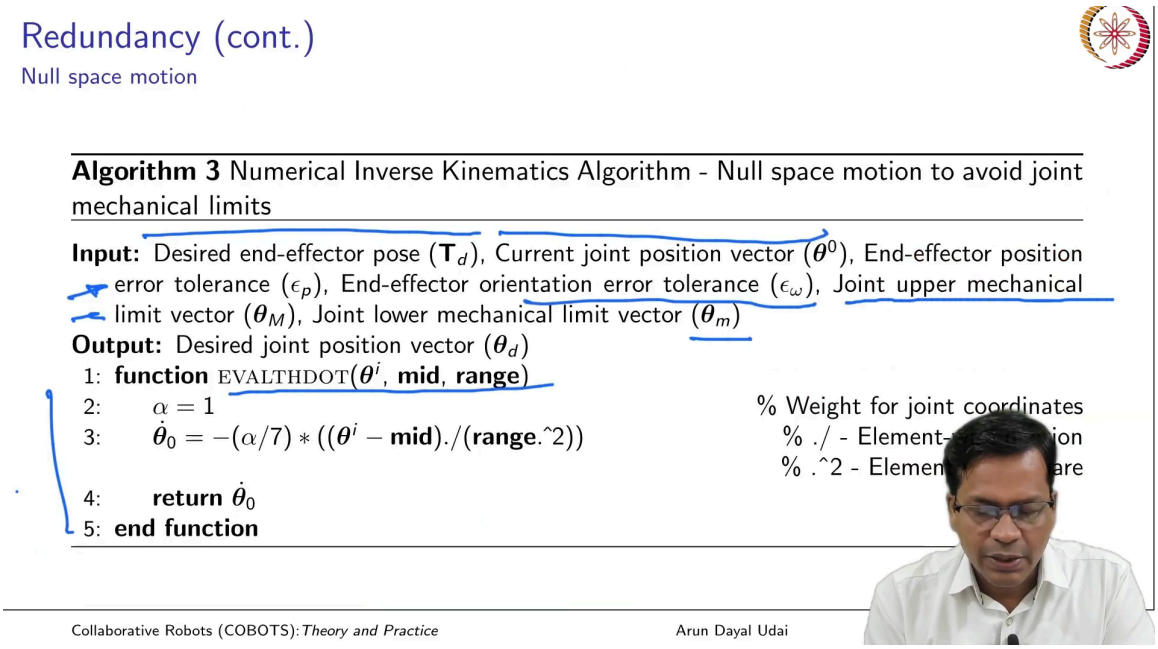


So, if I pack them together, I create the vector form. It will result in this, so this I have

expressed as one can use directly in MATLAB. So, dot slash means element-wise division. So, the vector can be expressed like this, and similarly, del w by del theta can now be written. So now, this is in vector form. Dot square 2 means element-wise square.

$$\implies \dot{\boldsymbol{\theta}}_0 = \alpha\left(\frac{\partial w(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right) = -\frac{\alpha}{n}(\boldsymbol{\theta} - \bar{\boldsymbol{\theta}})./((\boldsymbol{\theta}_M - \boldsymbol{\theta}_m).\hat{}2)$$

So, then theta 0 dot is equal to alpha times of del w by del theta that one you have calculated here, that comes here. Alpha is the scalar value, so now this can be written as this. So, this is compatible with MATLAB programming if somebody wants to do it. Where theta is the current joint position. Theta bar is the middle of the joint range. Theta M is the maximum limit, and theta small m is the joint minimum limit. So, using this, this is expressed.



Now, let us put this in algorithm form. So, the input is the end effector pose. Okay, Td. This is the desired pose; that is where you want to go. The current joint position vector this is the initial guess for the Newton-Raphson scheme now. End effector position error tolerance, so these are the tolerance for position and orientation which you specify for the Newton-Raphson scheme to stop somewhere. Joint upper mechanical limit and lower mechanical limit to make use of this, so that is what is used here, and evaluation of theta

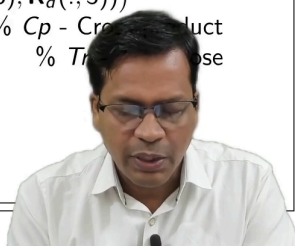
dot is done over here using this part of the algorithm. Theta dot that is calculated using this, so that is calculated here. This function will return theta 0 dot.

Redundancy (cont.)

Null space motion

Algorithm 3 Numerical Inverse Kinematics Algorithm - Null space motion to avoid joint mechanical limits (cont.)

```
function ERROR(T, T_d)
    x = T(1:3, 4)
    x_d = T_d(1:3, 4)
    e_p = x_d − x
    R = T(1:3, 1:3)
    R_d = T_d(1:3, 1:3)
    e_ω = 0.5 * (Cp(R(:,1), R_d(:,1)) + Cp(R(:,2), R_d(:,2)) + Cp(R(:,3), R_d(:,3)))
                                                  % Cp - Cr...uct
    e = Tr([Tr(e_ω), Tr(e_p)])                    % Tr...se
    return e
end function
mid = 0.5 * (θ_M + θ_m)
range = θ_M − θ_m
```



Now, the error. Error, as we have calculated earlier also, this is the way you calculate the error. Error in position, error in omega, and pack them together, return it. Mid-range is defined as theta m capital m small m by 2 mid, and this is the range. This is the range.

Redundancy (cont.)

Null space motion

Algorithm 3 Numerical Inverse Kinematics Algorithm - Null space motion to avoid joint mechanical limits (cont.)

```
T = T_n^0(θ^0)                          % T_n^0 - HTM of frame n as seen from base frame
e = ERROR(T, T_d)
i = 0
while ||e(1:3)|| > ε_ω or ||e(4:6)|| > ε_p do
    θ̇_0 = EVALTHDOT(θ^i, mid, range)
    θ^{i+1} = θ^i + J†(θ^i)e + (1 − J†(θ)J(θ))θ̇_0      % θ̇_0 - Joint velocity vector in null space
    θ^i = θ^{i+1}
    T = T_n^0(θ^i)
    e = ERROR(T, T_d)
    i = i + 1
end while
θ_d = θ^{i+1}
```

These are also some of the calculations which are used for this calculation used in this

algorithm. Now, again, starting the Newton-Raphson iteration. You keep on doing it till you achieve the ERROR below some values. So, this is for error calculation. The error you are calculating here increases the i, right? This is iterating with theta and calculating the new theta, new theta, new theta, and finally, you reached here. Evaluate theta dot; this is from the previous function, and keep on doing iteration till you reach the threshold, and ultimately, you return the desired theta. If it reaches near to the threshold of stopping criteria, you directly return the theta.

Example 3: KUKA LBR iiwa 14 R820 cobot

For desired pose with null space motion for joint limits

Considering the problem mentioned in Example 2.

Incorporating the joint limits provided by the manufacturer (KUKA) to generate the inverse kinematics solution within joint limits.

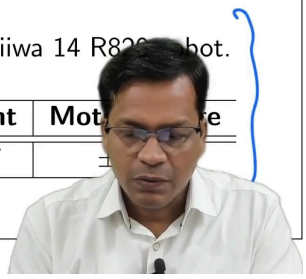
This can be achieved using the null space motion by utilizing the objective function for mechanical joint limits.

Below are the joint limits provided in the datasheet of KUKA LBR iiwa 14 R82[illegible]bot.

Joint	Motion range
A1	±170°
A2	±120°
A3	±170°

Joint	Motion range
A4	±120°
A5	±170°
A6	±120°

Joint	Mot[illegible]e
A7	±

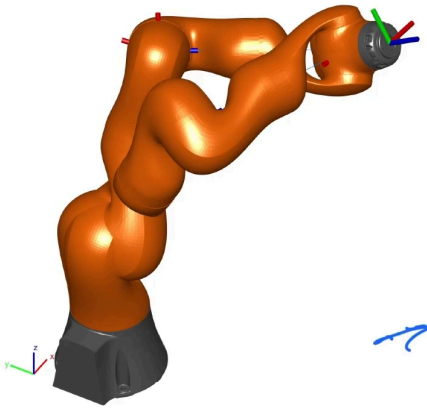


Collaborative Robots (COBOTS): *Theory and Practice* Arun Dayal Udai

So, considering this problem mentioned in example 2, example 2, we did it once. Using pose analysis, the same one we will incorporate now, and we will also implement null space motion, and we will exploit the joint limits now. So, incorporating the joint limits provided by the manufacturer, that is, KUKA in this case, to generate the inverse kinematic solution within the joint limits. So, I don't want my solution to go off the joint limits. This can be achieved using null space motion by utilising the objective function of the mechanical joint limits. So these are the values which are given by the manufacturer here. So, axis 1 cannot go more than plus or minus 170 degrees, axis 2, axis 3, 4, 5, 6, and 7. So, all the motion is already given by KUKA here. So, this is what the joint angle limits are, which are given by the KUKA LBR iiwa manufacturer, KUKA.

Example 3: KUKA LBR iiwa 14 R820 cobot (cont.)

For desired pose with null space motion for joint limits



Using the **Algorithm 3**,
We get
$\boldsymbol{\theta}_d = [-87.28, -28.15, 65.39, 73.01, -148.2, -68.95, -171.69]^T$
$\|\mathbf{e}_\omega\| = 7.932 \times 10^{-5}$ and
$\|\mathbf{e}_p\| = 4.064 \times 10^{-5}$.

That earlier in Example 2 was,
$\boldsymbol{\theta}_d = [173.98, 26.64, -61.56, -73.01, -114.87, 40.73, 37.9]^T$

It may be observed that all the joints are within ... echanical limits, unlike in Example 2, and still, the ... or has achieved the desired pose.

So, using algorithm 3, you see now I got theta d is this, theta 1, theta 2, theta 3, theta 4, 5, 6, and 7. All the 7 joint angles are obtained with the tolerance like this, So, this is where this was the stopping value of e omega and e_p, that is, the offset from the desired position and orientation. This is what I got, and the algorithm stopped. This is what we got.

Earlier in algorithm 2, you see, you got 173, 26, 61, 73, like this, where you see theta 1; if you see the manufacturer's joint limits, it should not have gone beyond 170 degrees plus or minus, right? But in this case, it is much above 170 degrees. So, it has gone up to 173 degrees. So, example 2 which did not use the joint angle limits in its algorithm. So, it went, the solution went beyond that. But in this case, it is well within the limits.

So, it may be observed that all the joints are within their mechanical limits, unlike example 2, and still, the end effector has achieved the desired pose.

That's all for this lecture. So, in the next lecture, I will start with a new module, that is Robot Statics. Thanks a lot.