Intro to Robotics: Review for Exam 2

Neeloy Chakraborty

neeloyc2@illinois.edu

Agenda

- Recap on Screws
- Forward Kinematics
- Velocity Kinematics
- Inverse Kinematics

Recap on Screws

While rotation matrices capture rotation, homogeneous transformations capture both rotation and translation

We want to find an operator that governs the ODE for the pose of the robot body

$$\dot{R} = [\omega_s]R = R[\omega_b]$$

$$\dot{T} = [V_s]T = T[V_b]$$

Recap on Screws

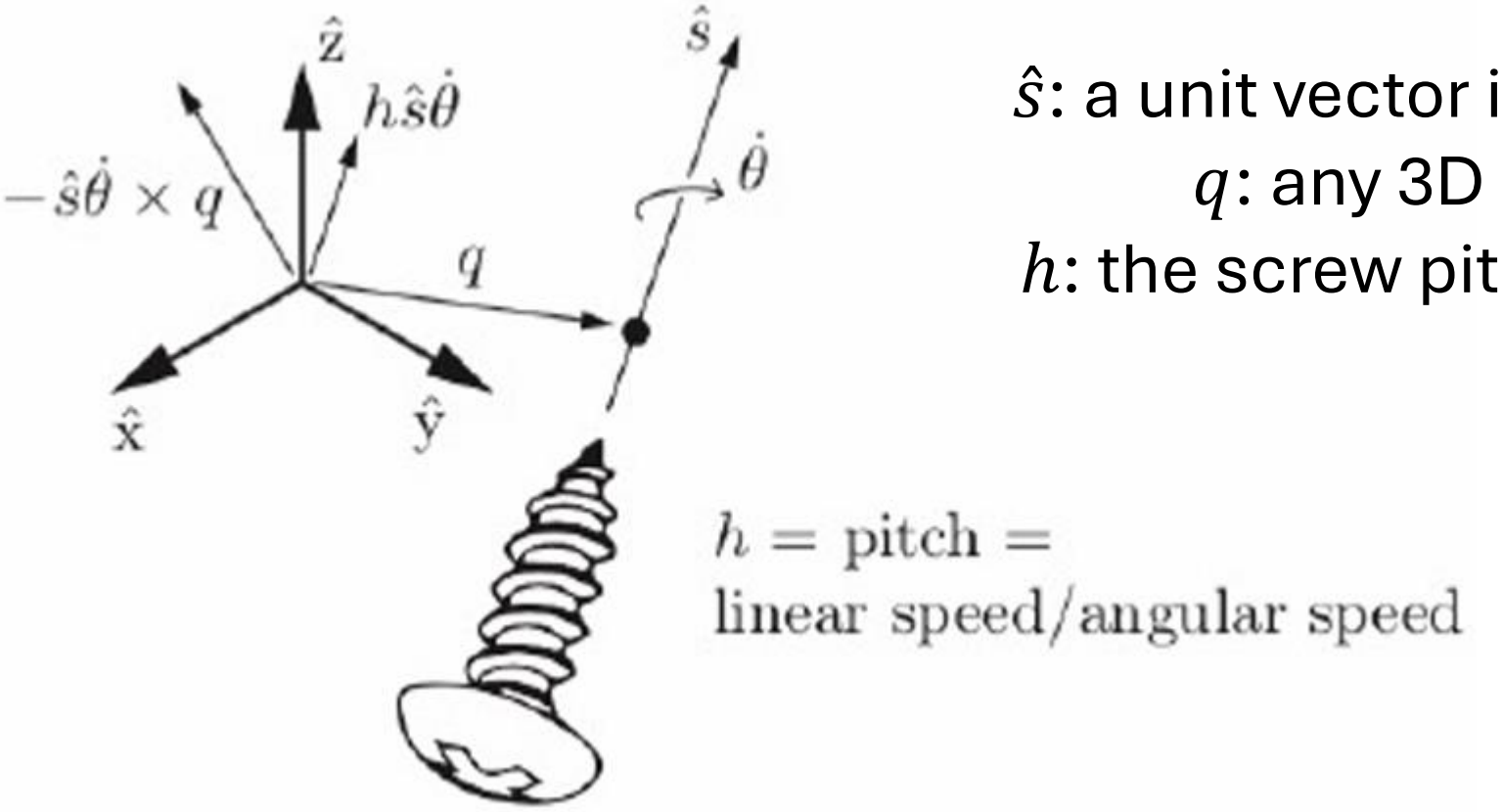
Twist V is a 6D real-valued vector containing the angular and linear velocity

$$\mathcal{V} = \begin{bmatrix} \omega \\ v \end{bmatrix} \quad [\mathcal{V}] = \begin{bmatrix} [\omega] & v \\ 0 & 0 \end{bmatrix}$$

$$\mathcal{V}_s = [Ad_T]\mathcal{V}_b = \begin{bmatrix} R & 0 \\ [p]R & R \end{bmatrix} \mathcal{V}_b$$

Recap on Screws

A twist V can be viewed in terms of a screw motion S



$\hat{s}$: a unit vector in the direction of the axis

q: any 3D point along the axis

h: the screw pitch (linear/angular speed)

Recap on Screws

A twist V for 1 second around the screw axis at angular velocity θ results in frame T

A twist V for θ seconds around the screw axis at angular velocity 1 results in frame T

Recap on Screws

$\hat{s}$: a unit vector in the direction of the axis
q: any 3D point along the axis
h: the screw pitch (linear/angular speed)

$$\mathcal{V} = \begin{bmatrix} \omega \\ v \end{bmatrix} = \begin{bmatrix} \hat{s}\dot{\theta} \\ h\hat{s}\dot{\theta} - \hat{s}\dot{\theta} \times q \end{bmatrix}$$

$$v = h\omega - \omega \times q$$

If $||\omega|| = 1$ and $h = 0$:
Pure rotation
Revolute joint

If $\omega = 0$ and $||v|| = 1$:
Pure translation
Prismatic joint

Solving the ODE $\dot{T} = [V_s]T$

scipy.linalg.expm()

Recall how we solved for the orientation R_{sb} given the angular velocity ω:

$$\dot{R} = [\omega_s]R = R[\omega_b]$$

$$R(\omega, \theta) = e^{[\omega_s]\theta}R(0) = R(0)e^{[\omega_b]\theta}$$

Similarly, solving for the pose T_{sb} after applying twist V:

$$\dot{T} = [V_s]T = T[V_b]$$

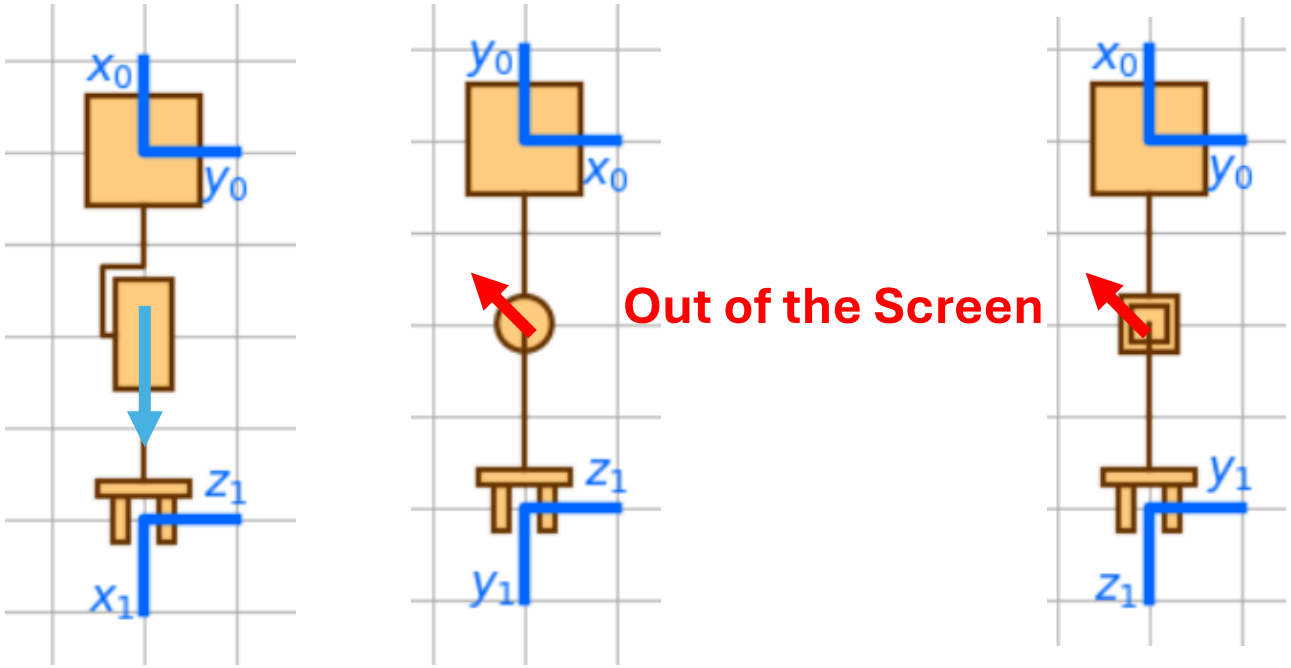
$$T(V, \theta) = e^{[V_s]\theta}T(0) = T(0)e^{[V_b]\theta}$$

Why do we care about screws???

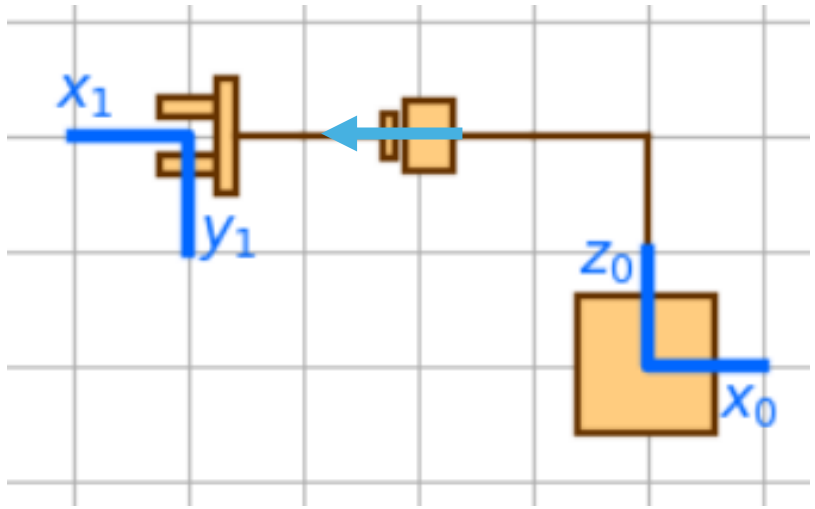
We can calculate the end effector pose given robot joint angles by treating each joint as a screw axis!

Forward Kinematics

Revolute Joint



Prismatic Joint



Product of Exponentials

$$T(\theta) = e^{[\mathcal{S}_1]\theta_1} e^{[\mathcal{S}_2]\theta_2} \dots e^{[\mathcal{S}_{n-1}]\theta_{n-1}} e^{[\mathcal{S}_n]\theta_n} M$$

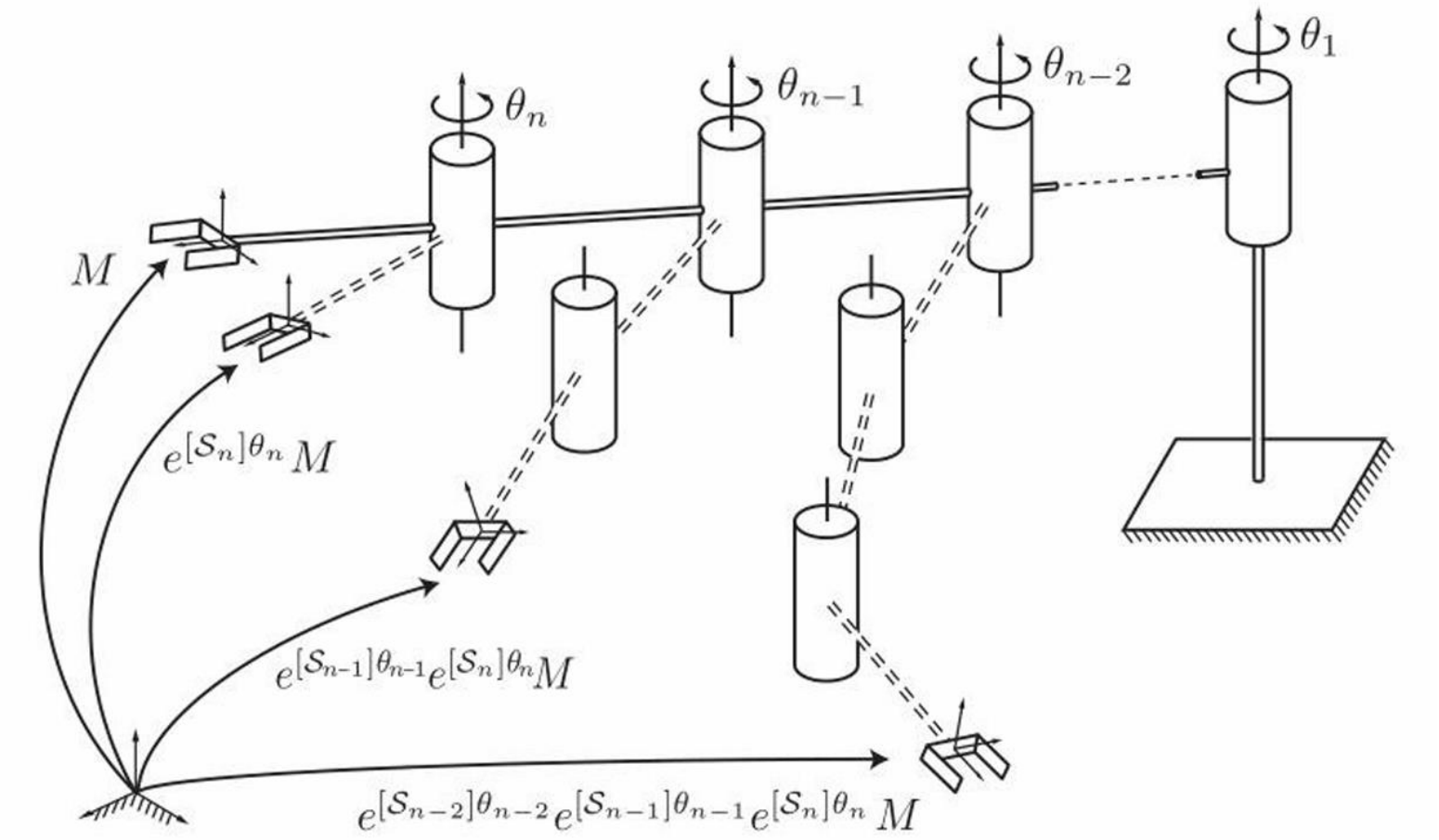


Figure 4.2: Illustration of the PoE formula for an n-link spatial open chain.

Product of Exponentials Algorithm

1. Identify the orientations of the fixed and tool frames with the right-hand rule

2. Create the homogeneous transformation M of the tool frame in the fixed frame

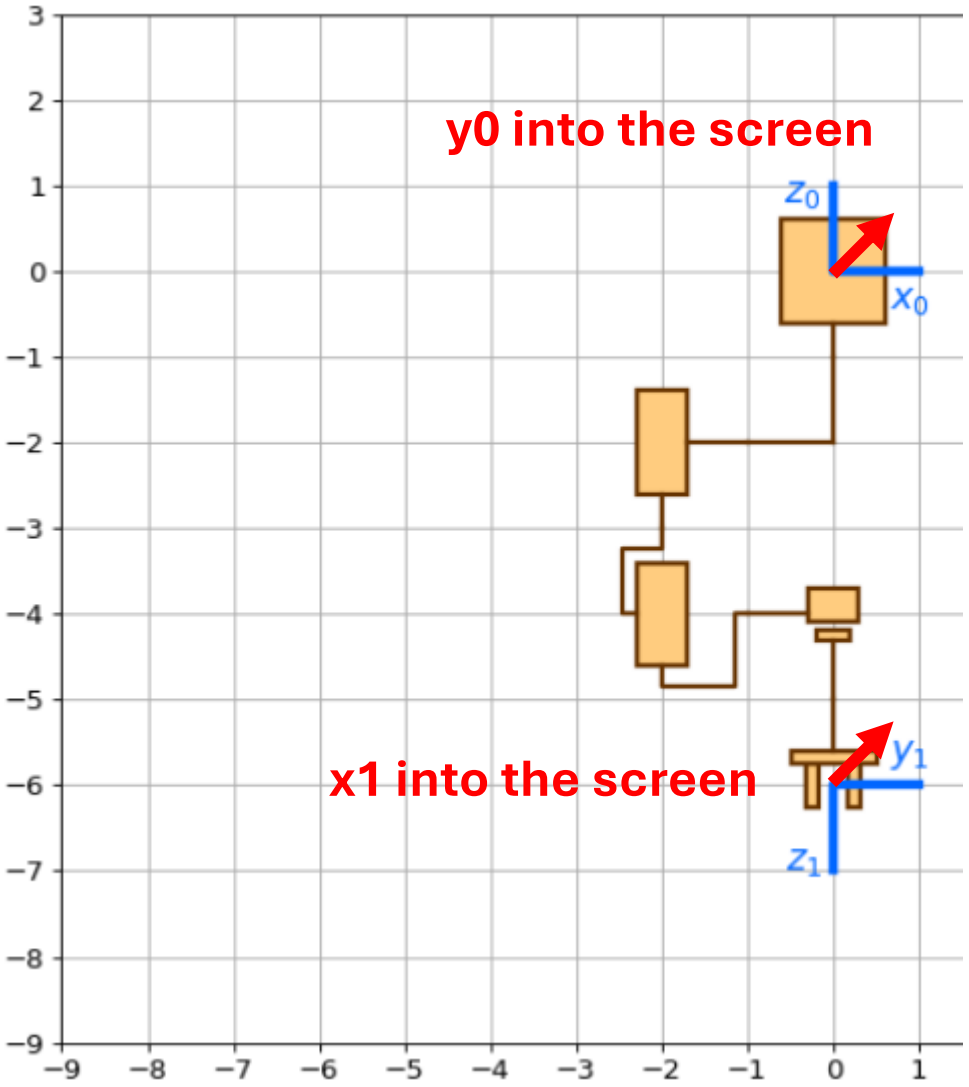
3. Build the screw axis for each joint
   - Ask yourself if we want the tool frame screw, the fixed frame screw, or either
   - If we specifically ask for S -> fixed frame screw
   - If we specifically ask for B -> body frame screw
   - Else -> your preference

4. Build the matrix form [V] of each screw axis

5. Use the PoE formula
   - If you used S in step 3 -> $T(\theta) = e^{[S_1]\theta_1} e^{[S_2]\theta_2} \dots e^{[S_n]\theta_n} M$
   - If you used B in step 3 -> $T(\theta) = M e^{[B_1]\theta_1} e^{[B_2]\theta_2} \dots e^{[B_n]\theta_n}$

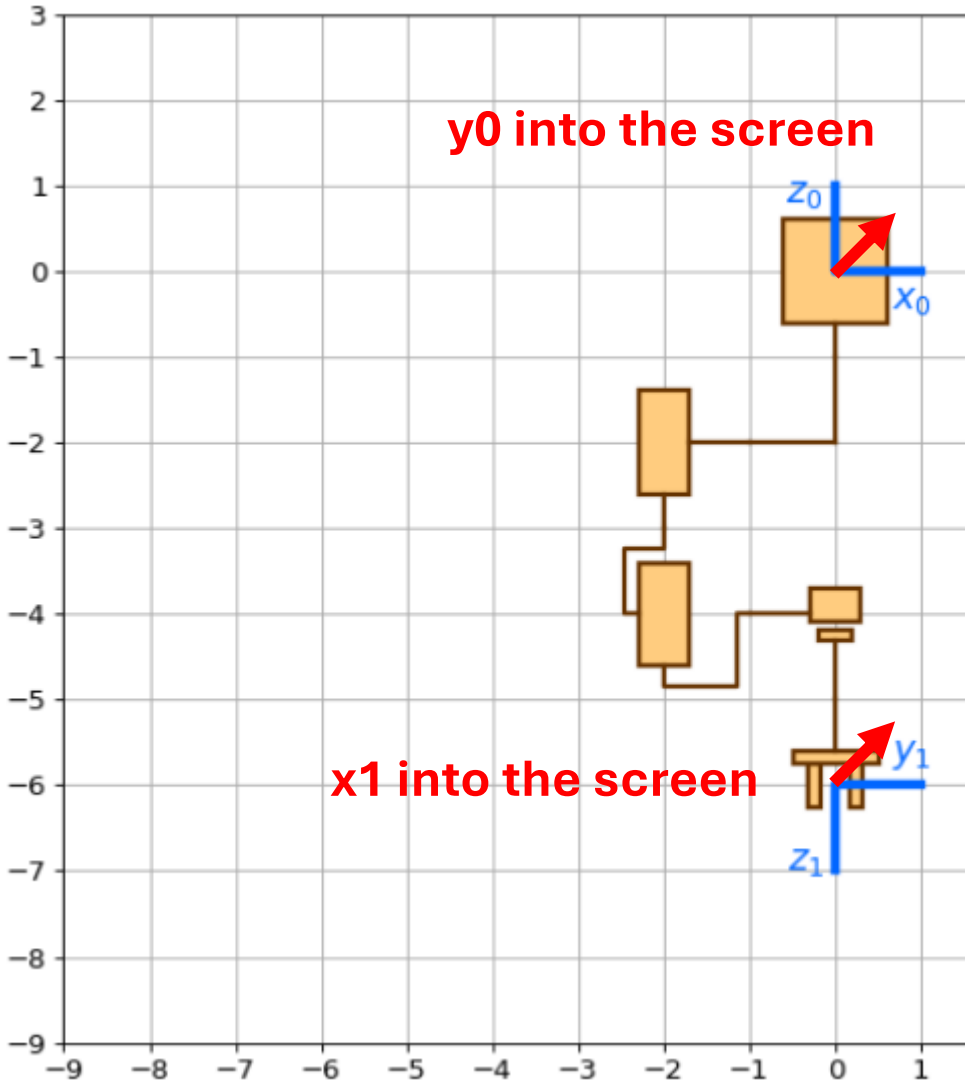
3 DoF RRP Robot Example

1. Identify the orientations of the fixed and tool frames with the right-hand rule



3 DoF RRP Robot Example

2. Create the homogeneous transformation M of the tool frame in the fixed frame



x1 is pointing in y0 direction
y1 is pointing in x0 direction
z1 is pointing in –z0 direction

Tool frame is translated by -6 in z0 direction

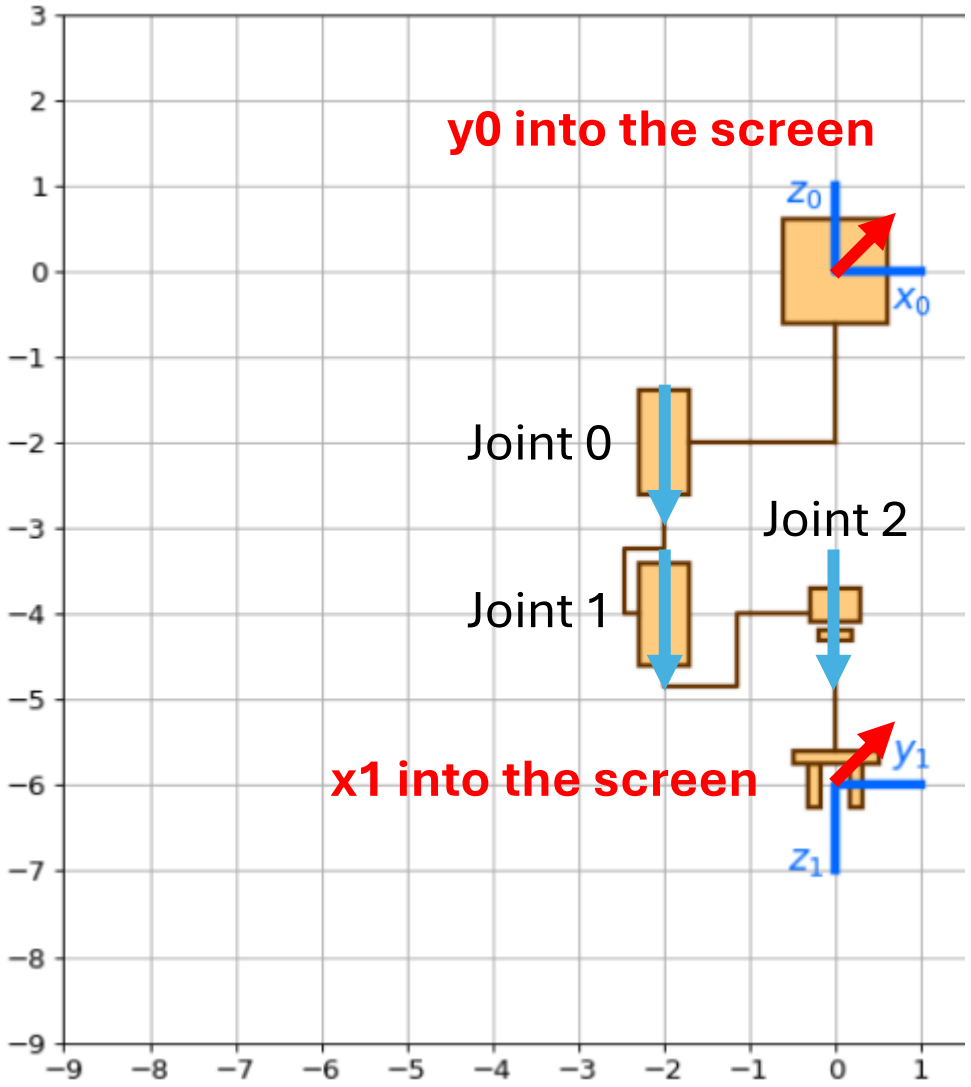
```
R = np.array([[0, 1, 0 ],
              [1, 0, 0 ],
              [0, 0, -1]])
```

```
t = np.array([[0],
              [0 ],
              [-6]])
```

```
M = np.block([[R, t],
              [0, 0, 0, 1]])
```

3 DoF RRP Robot Example

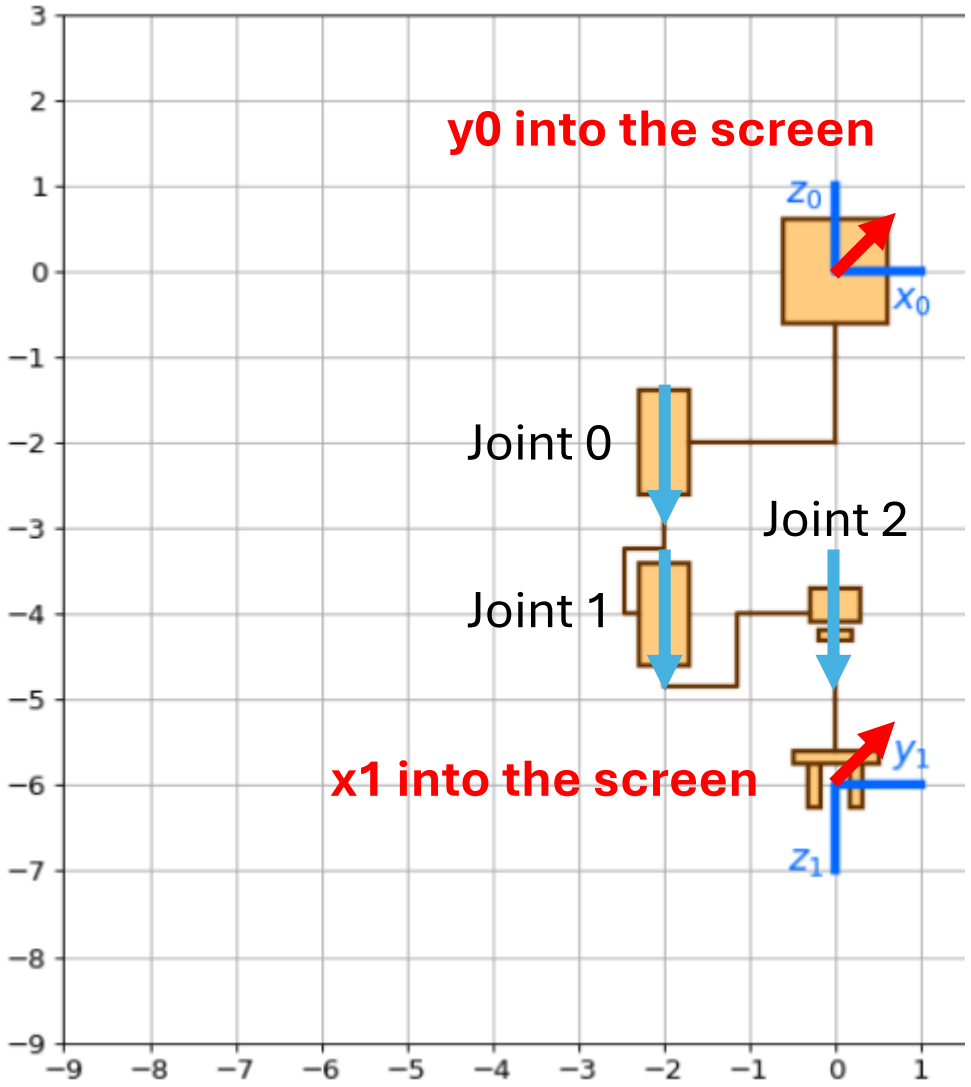
3. Build the screw axis for each joint



- Joint 0
  - revolute
  - S
    - w = [0,0,-1]
    - q = [-2,0,-2]
    - v = -w x q = [0,-2,0]
    - S = [w v] = [0,0,-1,0,-2,0]
  - B
    - w = [0,0,1]
    - q = [0,-2,-4]
    - v = -w x q = [-2,0,0]
    - B = [w v] = [0,0,1,-2,0,0]

3 DoF RRP Robot Example

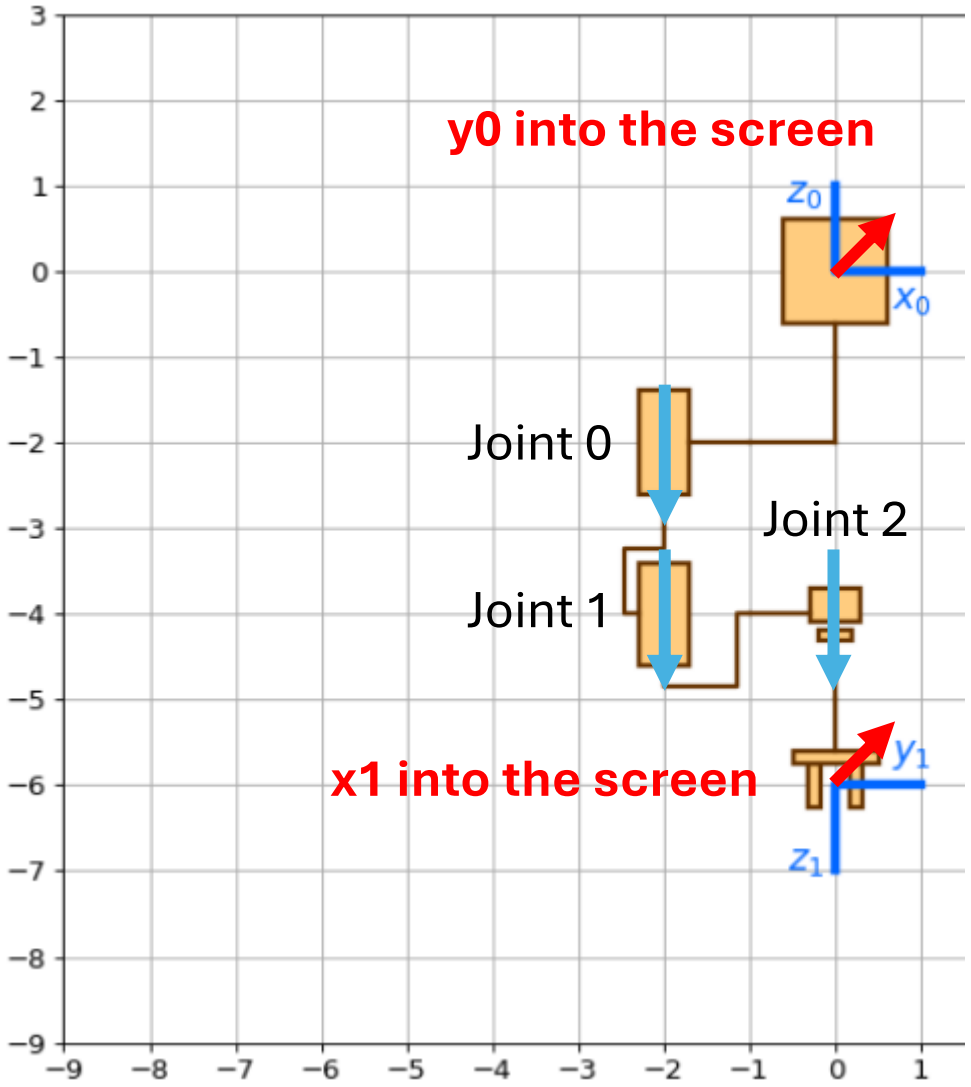
3. Build the screw axis for each joint



- Joint 1
  - revolute
  - S
    - w = [0,0,-1]
    - q = [-2,0,-4]
    - v = -w x q = [0,-2,0]
    - S = [w v] = [0,0,-1,0,-2,0]
  - B
    - w = [0,0,1]
    - q = [0,-2,-2]
    - v = -w x q = [-2,0,0]
    - B = [w v] = [0,0,1,-2,0,0]

3 DoF RRP Robot Example

3. Build the screw axis for each joint

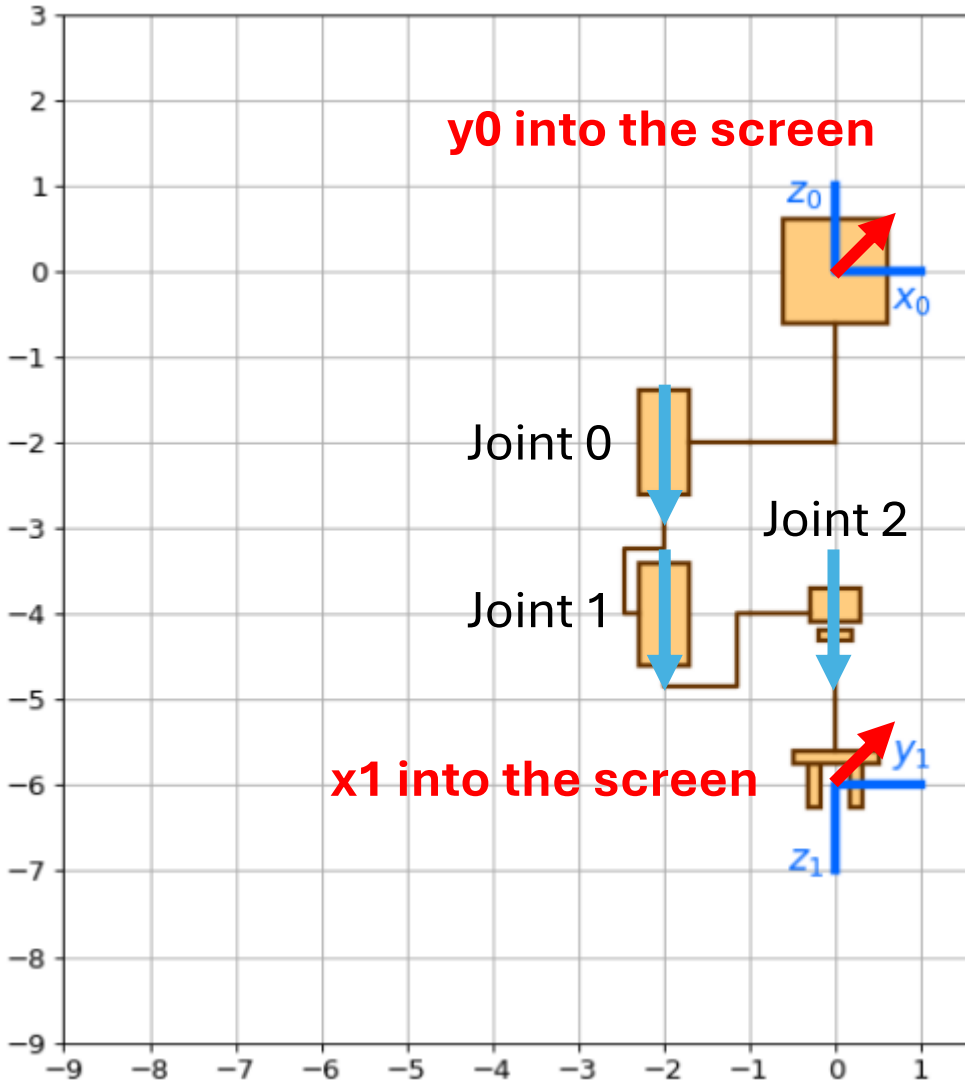


- Joint 2
  - prismatic
  - S
    - w = [0,0,0]
    - v = [0,0,-1]
    - S = [w v] = [0,0,0,0,0,-1]
  - B
    - w = [0,0,0]
    - v = [0,0,1]
    - B = [w v] = [0,0,0,0,0,1]

3 DoF RRP Robot Example

4. Build the matrix form of the screw axis per joint

$$[\mathcal{V}] = \begin{bmatrix} [\omega] & v \\ 0 & 0 \end{bmatrix}$$



- Joint 0
  - S = [w v] = [0,0,-1,0,-2,0]

```
[[ 0  1  0  0]
[-1  0  0 -2]
[ 0  0  0  0]
[ 0  0  0  0]]
```

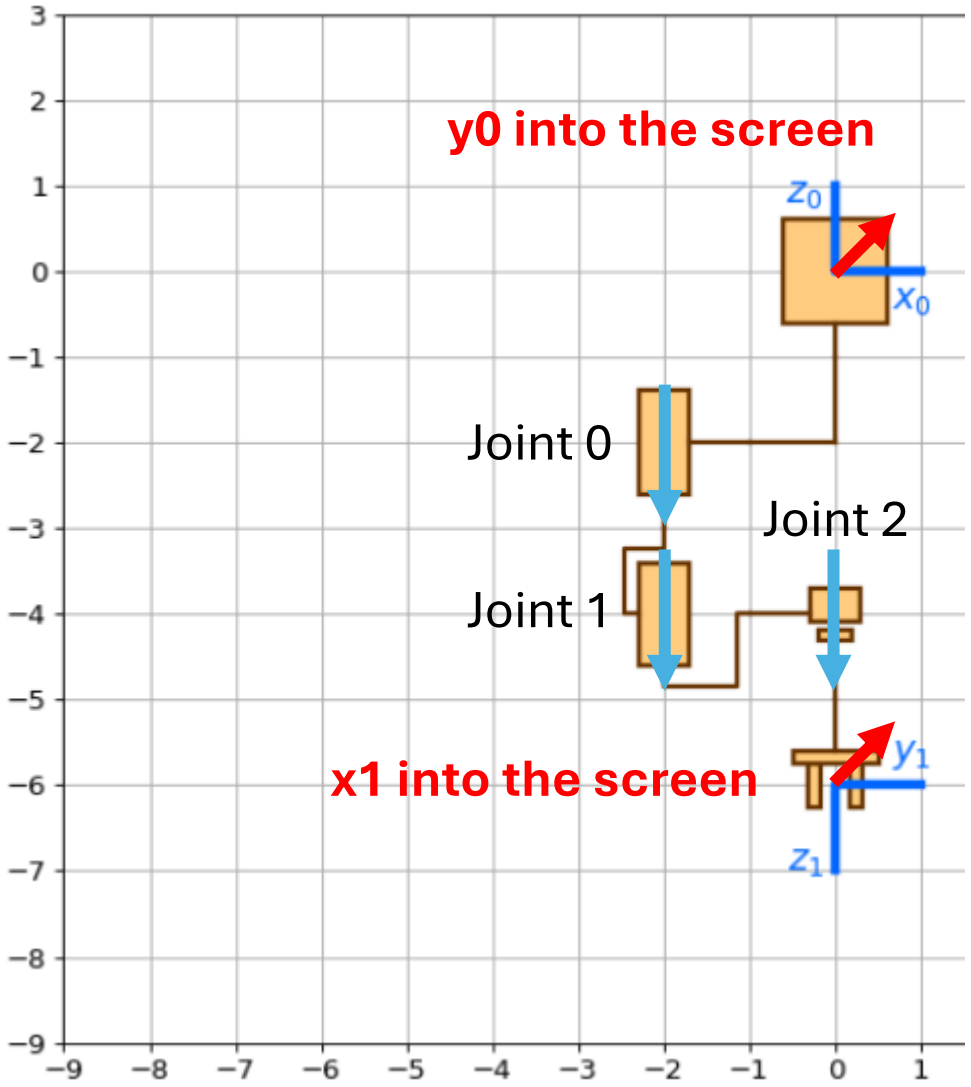
  - B = [w v] = [0,0,1,-2,0,0]

```
[[ 0 -1  0 -2]
[ 1  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]]
```

3 DoF RRP Robot Example

4. Build the matrix form of the screw axis per joint

$$[\mathcal{V}] = \begin{bmatrix} [\omega] & v \\ 0 & 0 \end{bmatrix}$$



- Joint 1
  - S = [w v] = [0,0,-1,0,-2,0]

```
[[ 0  1  0  0]
[-1  0  0 -2]
[ 0  0  0  0]
[ 0  0  0  0]]
```

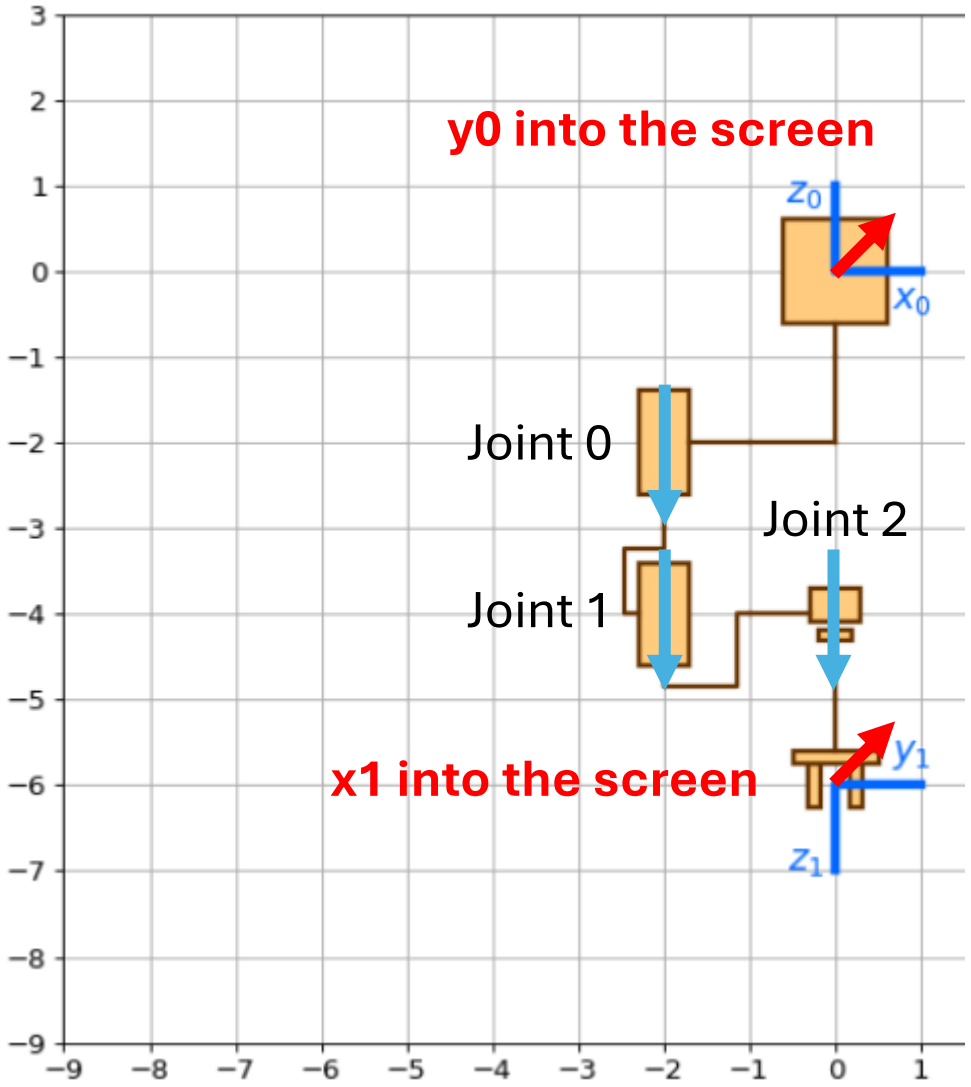
  - B = [w v] = [0,0,1,-2,0,0]

```
[[ 0 -1  0 -2]
[ 1  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]]
```

3 DoF RRP Robot Example

4. Build the matrix form of the screw axis per joint

$$[\mathcal{V}] = \begin{bmatrix} [\omega] & v \\ 0 & 0 \end{bmatrix}$$



- Joint 2
  - S = [w v] = [0,0,0,0,0,-1]

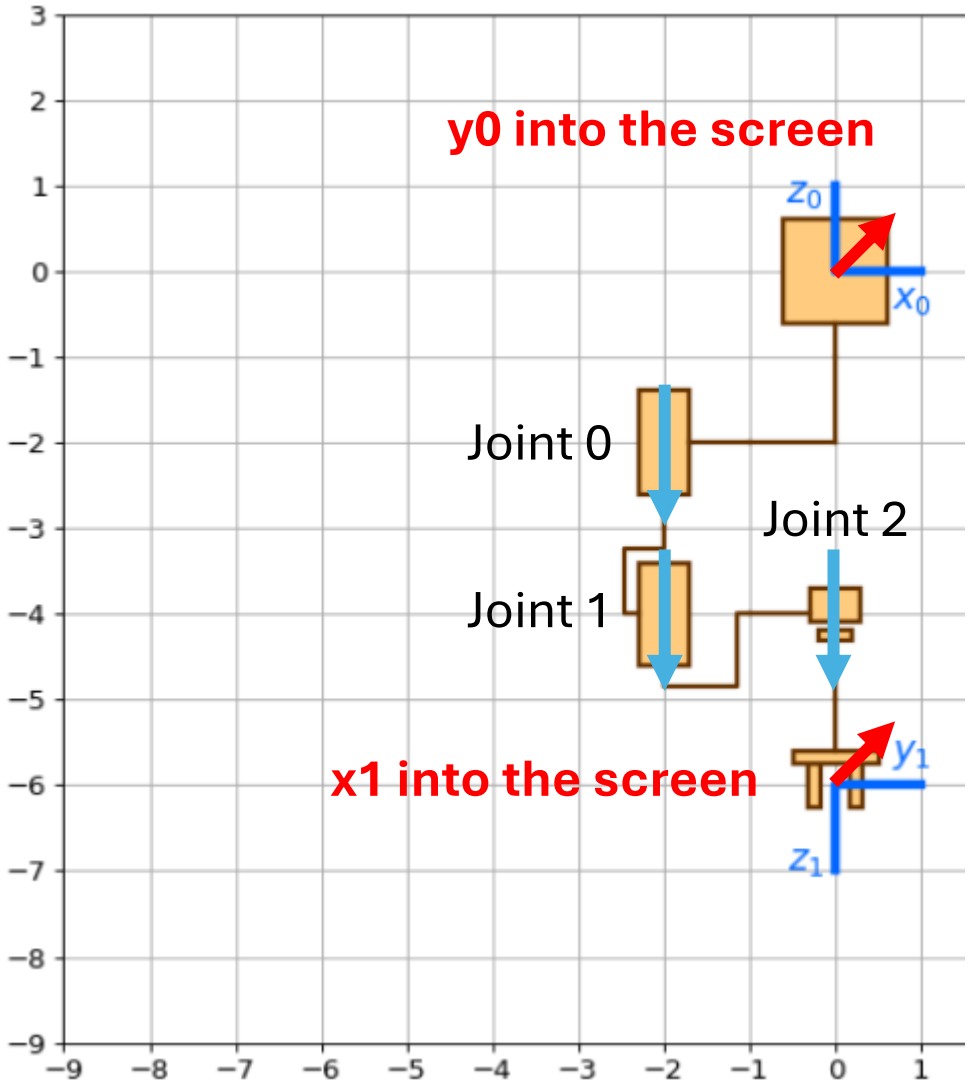
```
[[ 0  0  0  0]
 [ 0  0  0  0]
 [ 0  0  0 -1]
 [ 0  0  0  0]]
```

  - B = [w v] = [0,0,0,0,0,1]

```
[[ 0  0  0  0]
 [ 0  0  0  0]
 [ 0  0  0 -1]
 [ 0  0  0  0]]
```

3 DoF RRP Robot Example

5. Apply product of exponentials formula



- Fixed frame conventions

$$T(\theta) = e^{[S_1]\theta_1} e^{[S_2]\theta_2} \dots e^{[S_n]\theta_n} M$$

```
[[ 0.68163876  0.73168887  0.          -0.53662226]
 [ 0.73168887 -0.68163876  0.          -1.36327752]
 [ 0.          0.         -1.          -6.75      ]
 [ 0.          0.          0.           1.        ]]
```

- Body frame conventions

$$T(\theta) = M e^{[B_1]\theta_1} e^{[B_2]\theta_2} \dots e^{[B_n]\theta_n}$$

```
[[ 0.68163876  0.73168887  0.          -0.53662226]
 [ 0.73168887 -0.68163876  0.          -1.36327752]
 [ 0.          0.         -1.          -6.75      ]
 [ 0.          0.          0.           1.        ]]
```

Velocity Kinematics

Forward Kinematics

Calculate the position of the end-effector of an open chain given joint angles

$x(t) = f(\theta(t))$

$x(t)$: pose of tool

$\theta(t)$: joint angles

f: FK map / $T(\theta)$ / PoE

Velocity Kinematics

Calculate the velocity (twist!) of the end-effector frame

$$\dot{x} = \frac{d}{dt}x(t) = \frac{\partial f}{\partial \theta}(\theta)\cdot\dot{\theta}$$

$\frac{\partial f}{\partial \theta}(\theta) = J(\theta)$ → Jacobian

Definition 5.1. Let the forward kinematics of an n-link open chain be expressed in the following product of exponentials form:

$$T = e^{[\mathcal{S}_1]\theta_1} \dots e^{[\mathcal{S}_n]\theta_n} M. \tag{5.9}$$

The **space Jacobian** $J_s(\theta) \in \mathbb{R}^{6\times n}$ relates the joint rate vector $\dot{\theta} \in \mathbb{R}^n$ to the spatial twist $\mathcal{V}_s$ via

$$\mathcal{V}_s = J_s(\theta)\dot{\theta}. \tag{5.10}$$

The ith column of $J_s(\theta)$ is

$$J_{si}(\theta) = \mathrm{Ad}_{e^{[\mathcal{S}_1]\theta_1 \dots e^{[\mathcal{S}_{i-1}]\theta_{i-1}}}}(\mathcal{S}_i), \tag{5.11}$$

for $i = 2, \dots, n$, with the first column $J_{s1} = \mathcal{S}_1$.

On the formula sheet …

Velocity Kinematics

Space Jacobian

$$\mathcal{V}_s = J_s(\theta)\dot{\theta} = [J_{s,1}\ J_{s,2}\ \cdots\ J_{s,n}] \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \vdots \\ \dot{\theta}_n \end{bmatrix}$$

$$J_{s,1} = S_1 \quad J_{s,2} = Ad_{e^{[S_1]\theta_1}}(S_2) \quad \cdots \quad J_{s,n} = Ad_{e^{[S_1]\theta_1 \ldots e^{[S_{n-1}]\theta_{n-1}}}}(S_n)$$

Algorithm for Building the Jacobian

1. Identify the orientations of the fixed frame with the right-hand rule

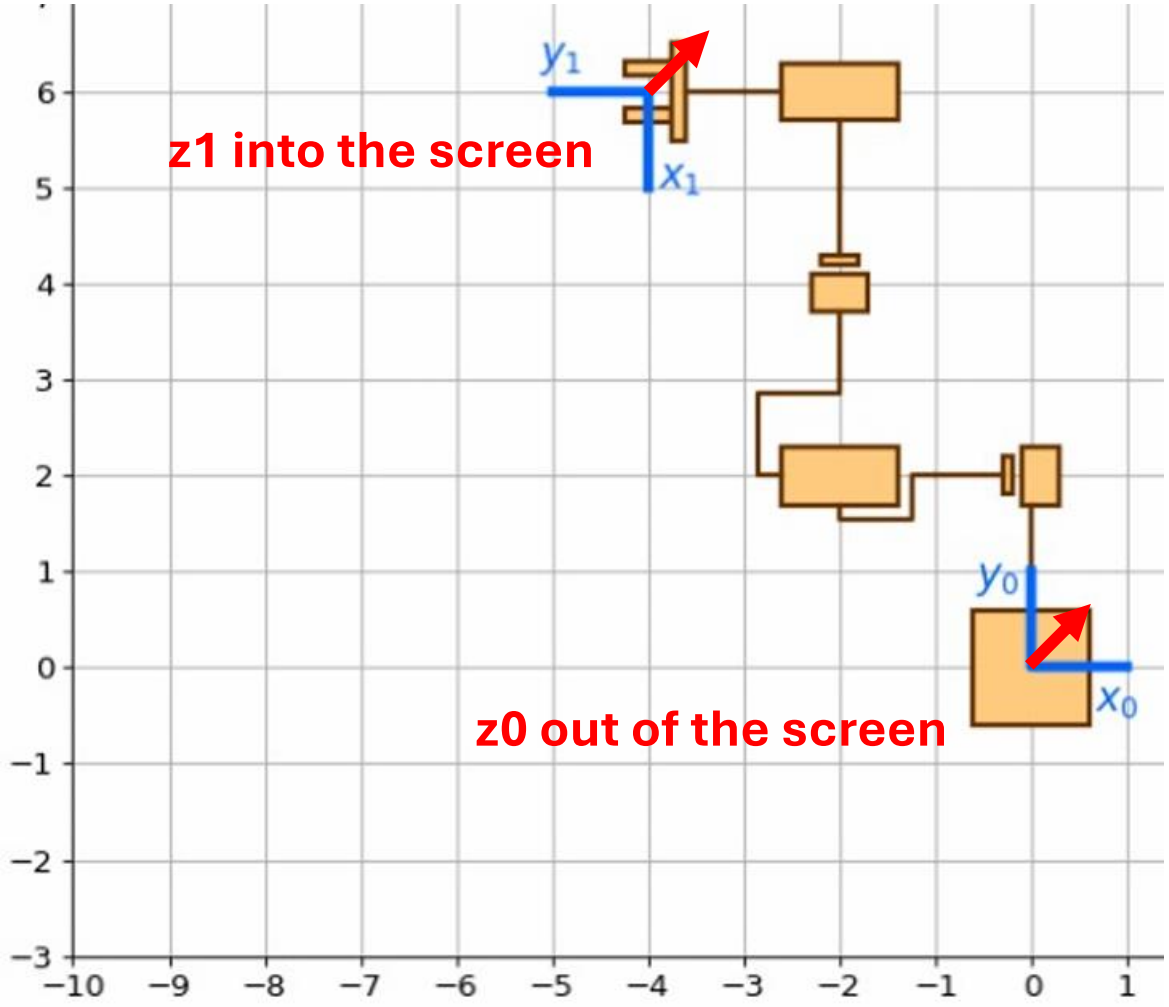
2. Build the screw axis for each joint
   - Ask yourself if we want space Jacobian or the body Jacobian
   - If we specifically ask for Js -> fixed frame screw
   - If we specifically ask for Jb -> body frame screw
     - I prefer to solve for Js and then use the Adj map to convert to Jb

3. For each column i of Js
   - Solve for the PoE up to joint i – 1
   - Compute its adjoint
   - Multiply the adjoint by the screw axis of joint i

4. If we want Jb, $J_b = [Ad_{T_{bs}}] J_s$ $\qquad [Ad_T] = \begin{bmatrix} R & 0 \\ [p]R & R \end{bmatrix}$

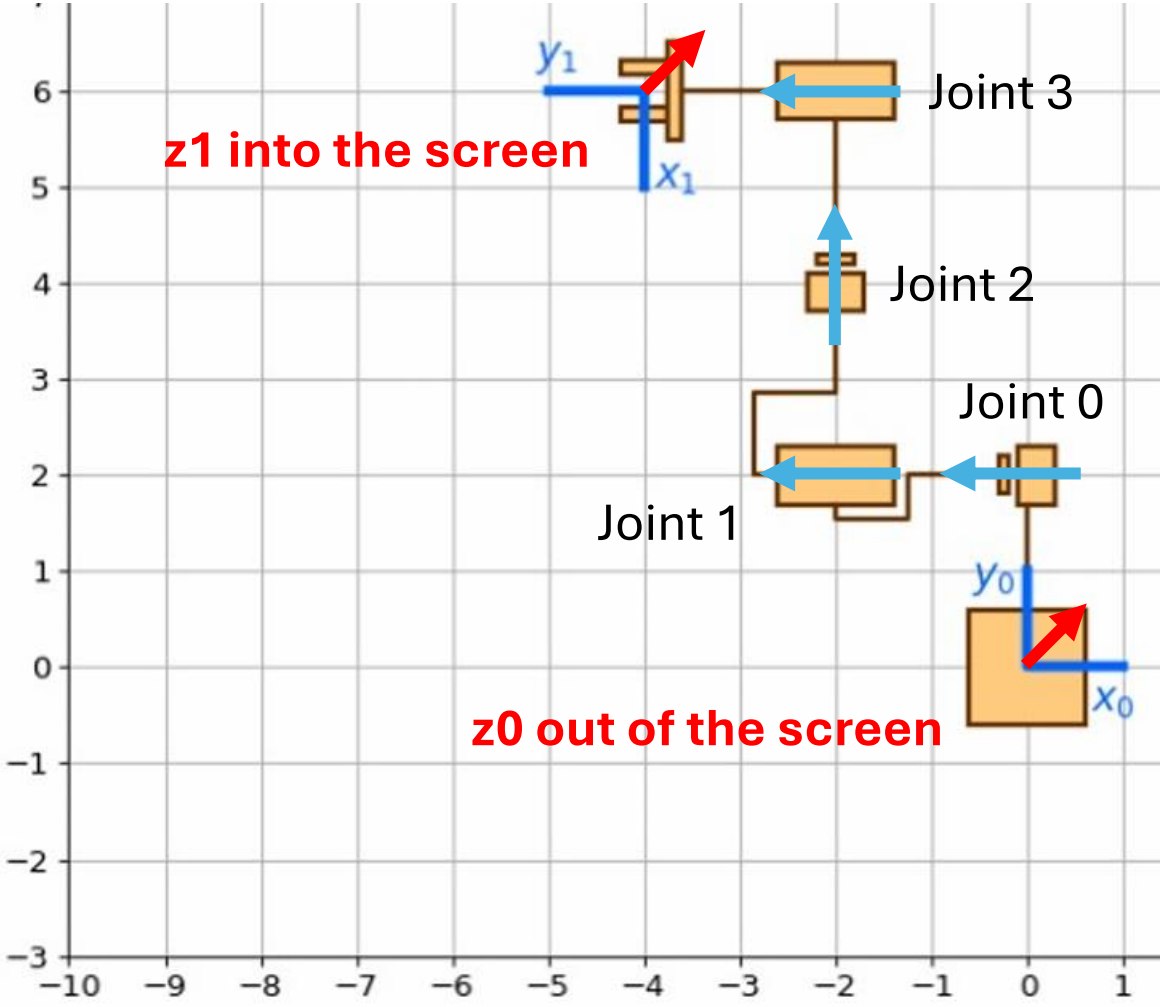
4 DoF PRPR Robot Example

1. Identify the orientations of the fixed and tool frames with the right-hand rule



4 DoF PRPR Robot Example

2. Build the screw axis (and matrix form) for each joint

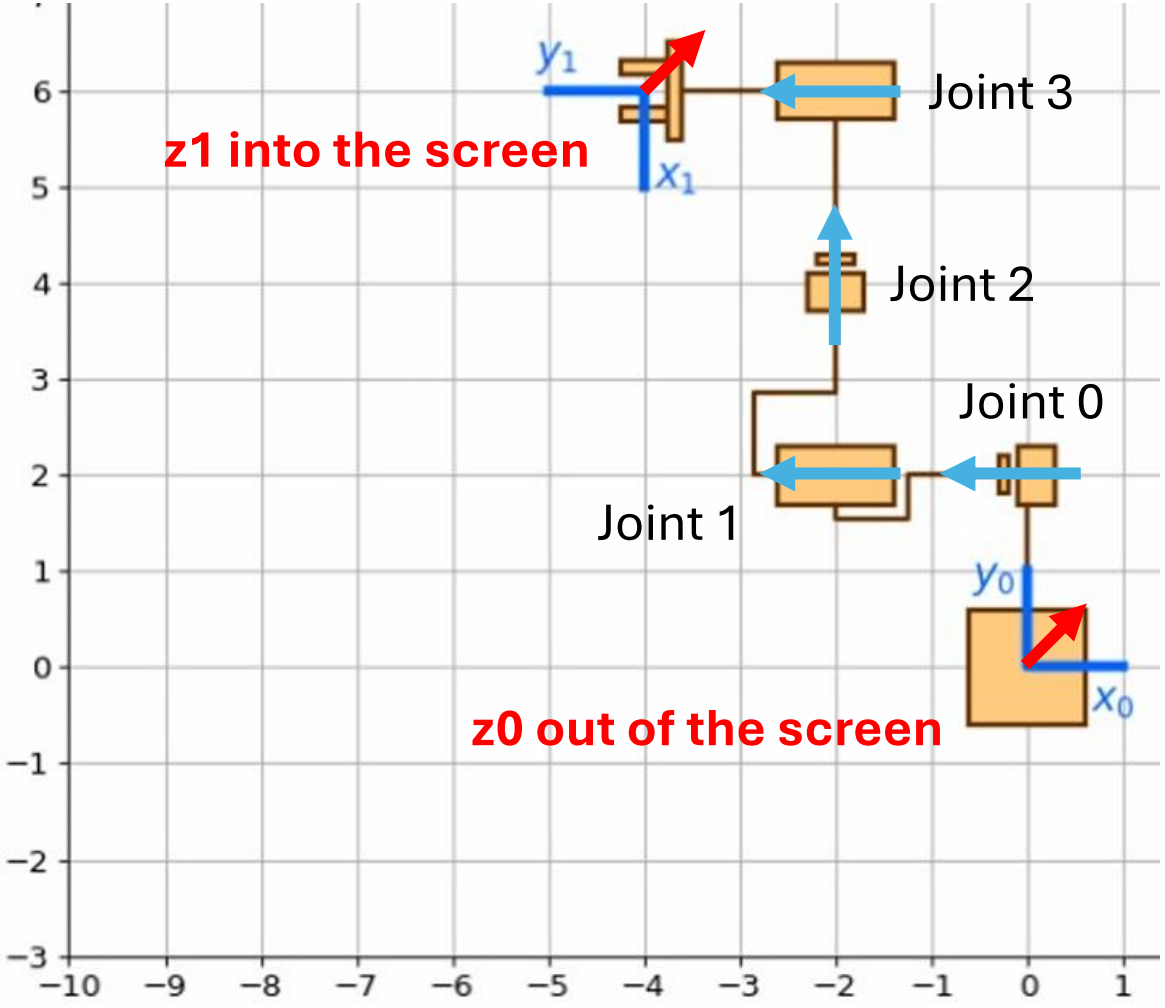


- Joint 0
  - prismatic
  - S0
    - w = [0,0,0]
    - v = [-1,0,0]
    - S = [w v] = [0,0,0,-1,0,0]
  - [S0]

```
[[ 0  0  0 -1]
 [ 0  0  0  0]
 [ 0  0  0  0]
 [ 0  0  0  0]]
```

4 DoF PRPR Robot Example

2. Build the screw axis (and matrix form) for each joint

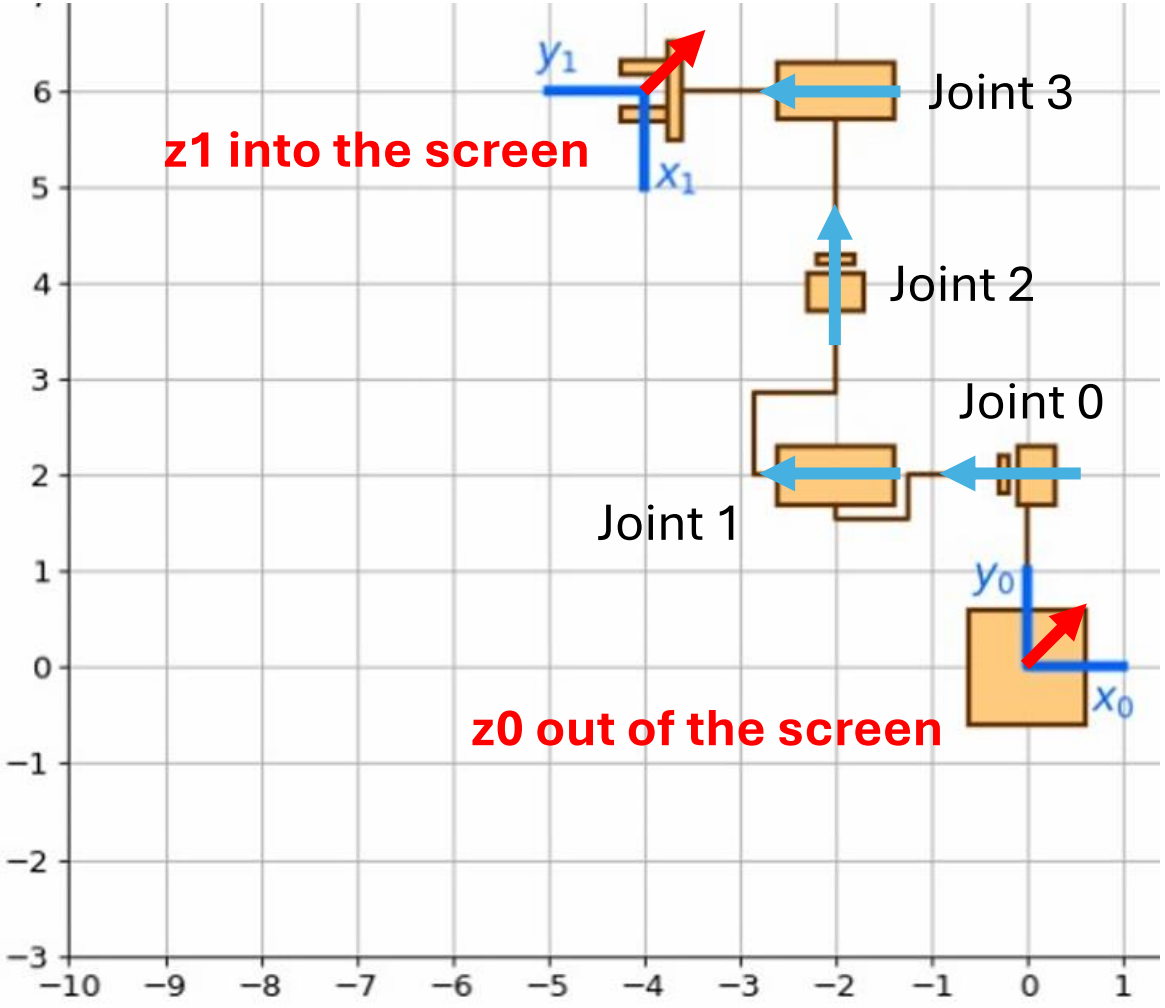


- Joint 1
  - revolute
  - S1
    - w = [-1,0,0]
    - q = [-2,2,0]
    - v = -w x q = [0,0,2]
    - S = [w v] = [-1,0,0,0,0,2]
  - [S1]

```
[[ 0  0  0  0]
 [ 0  0  1  0]
 [ 0 -1  0  2]
 [ 0  0  0  0]]
```

4 DoF PRPR Robot Example

2. Build the screw axis (and matrix form) for each joint

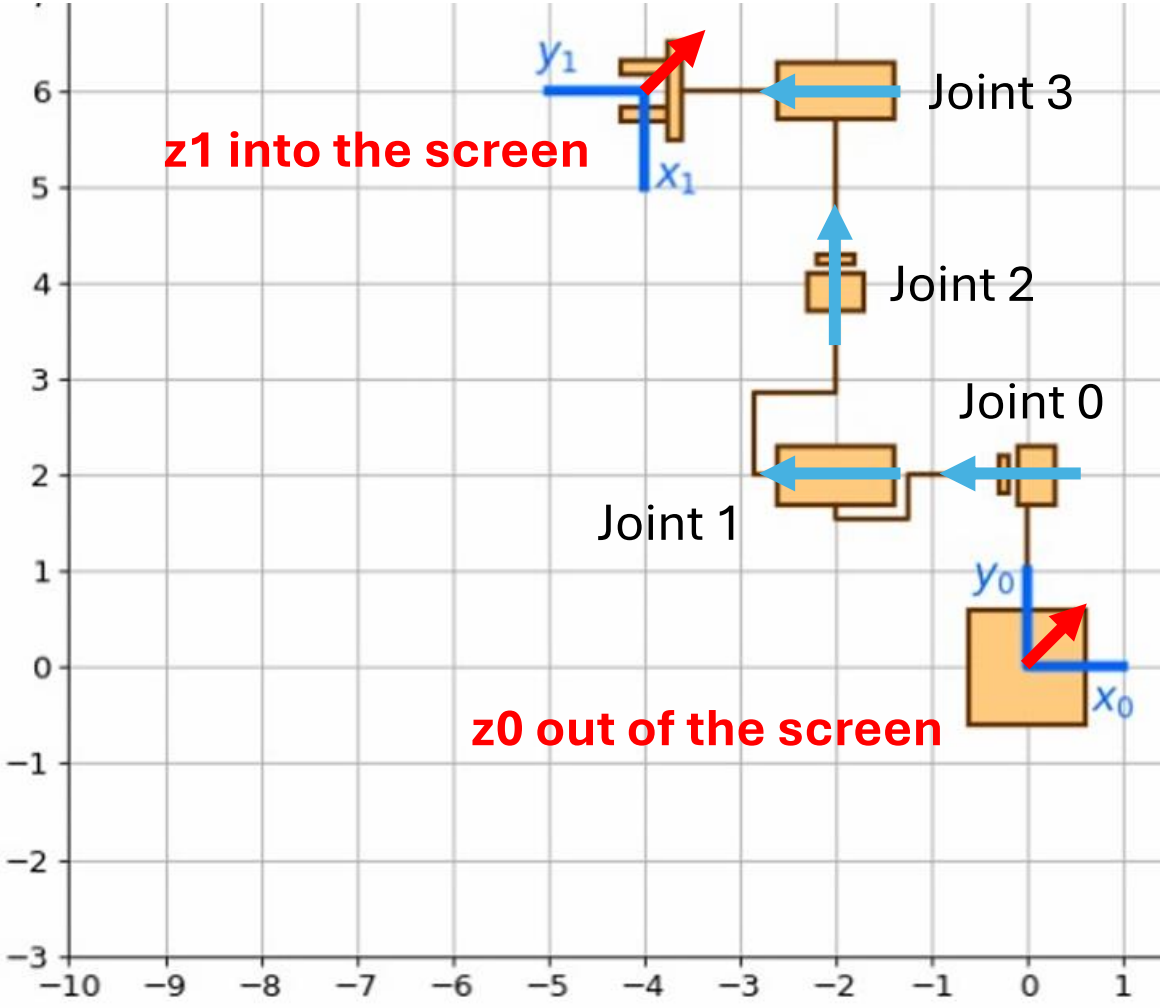


- Joint 2
  - prismatic
  - S2
    - w = [0,0,0]
    - v = [0,1,0]
    - S = [w v] = [0,0,0,0,1,0]
  - [S2]

```
[[0 0 0 0]
[0 0 0 1]
[0 0 0 0]
[0 0 0 0]]
```

4 DoF PRPR Robot Example

2. Build the screw axis (and matrix form) for each joint

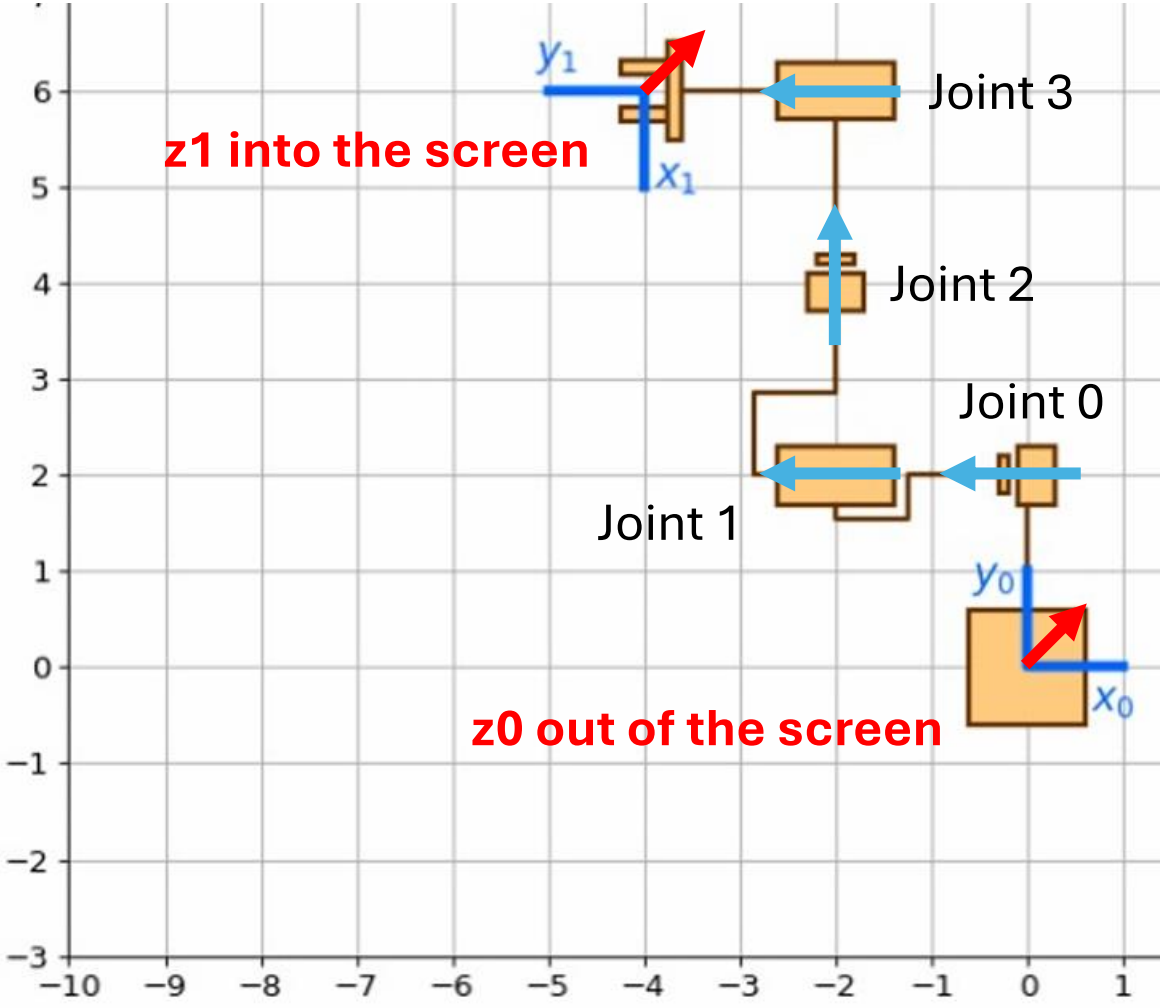


- Joint 3
  - revolute
  - S3
    - w = [-1,0,0]
    - q = [-2,6,0]
    - v = -w x q = [0,0,6]
    - S = [w v] = [-1,0,0,0,0,6]
  - [S3]

```
[[ 0  0  0  0]
 [ 0  0  1  0]
 [ 0 -1  0  6]
 [ 0  0  0  0]]
```

4 DoF PRPR Robot Example

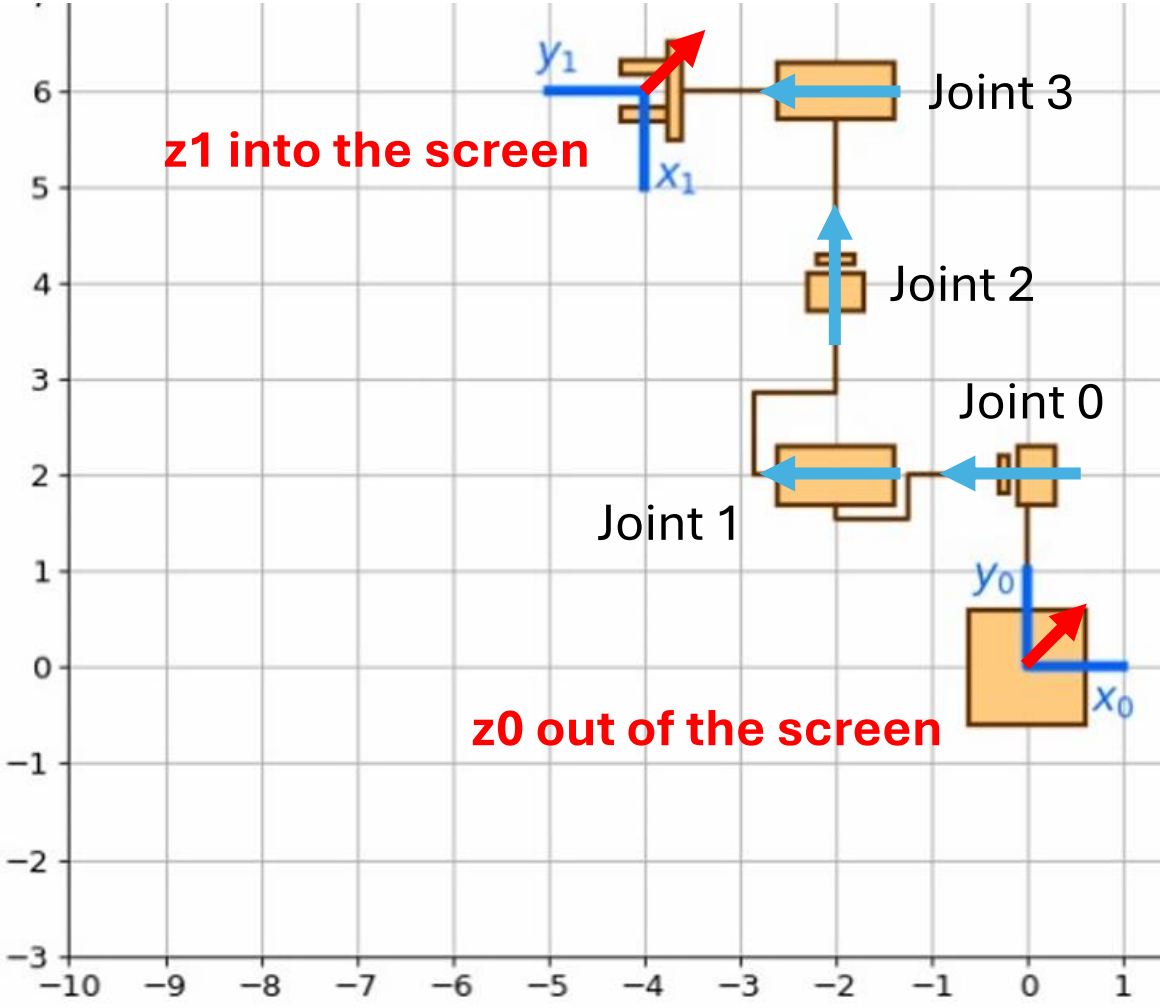
3. Fill in the columns of the Jacobian



- Column 0
  - S0 = [0,0,0,-1,0,0]
  - J0 = [0,0,0,-1,0,0]

4 DoF PRPR Robot Example

3. Fill in the columns of the Jacobian



- Column 1
  - S1 = [-1,0,0,0,0,2]
  - J1 = [Ad_T1] S1

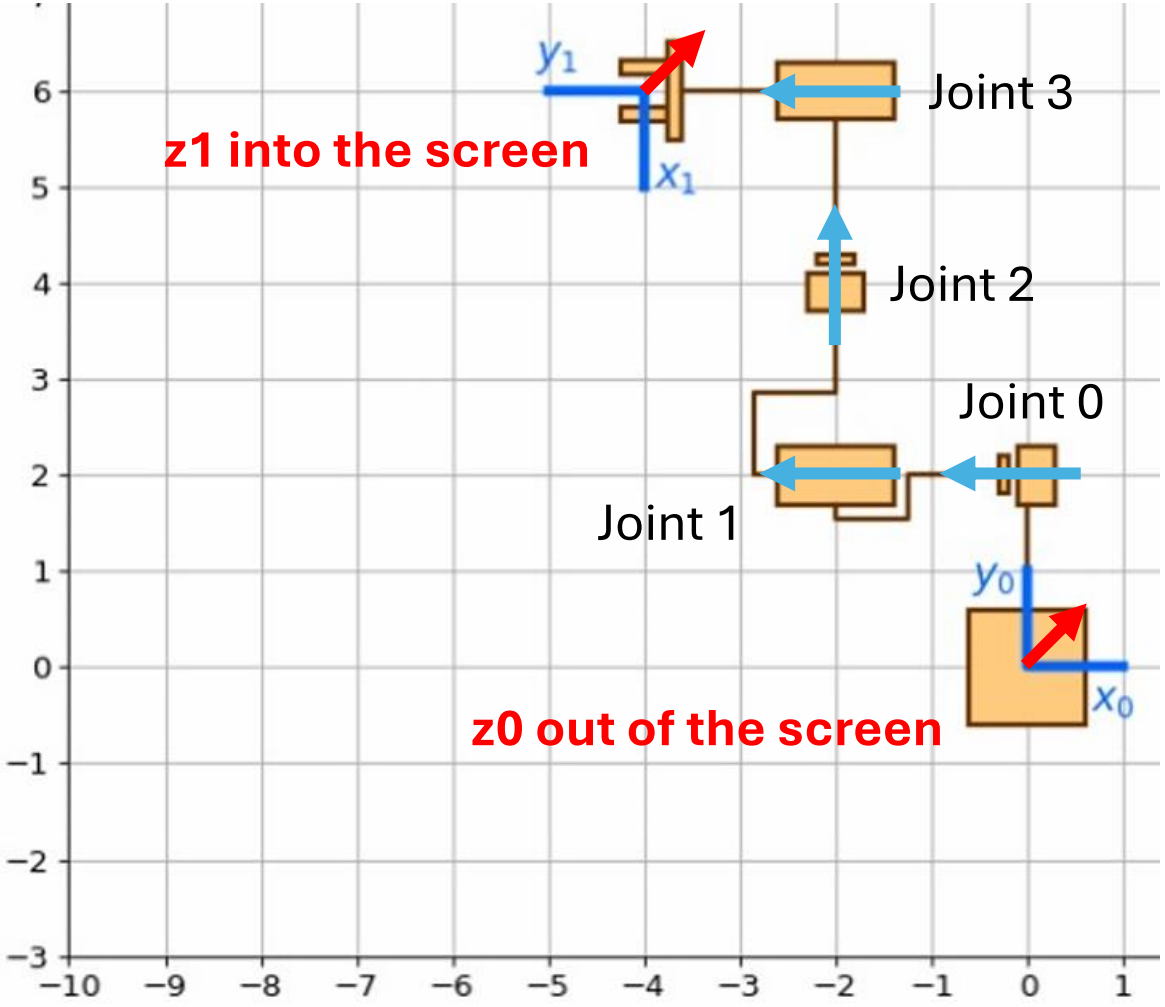
```
Ad_exp_s_0_theta_0 = adjoint(expm(mat_s_0 * theta_0))
J_s_1 = Ad_exp_s_0_theta_0 @ s_1
```

$$[Ad_T] = \begin{bmatrix} R & 0 \\ [p]R & R \end{bmatrix}$$

```
[[-1.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 2.]]
```

4 DoF PRPR Robot Example

3. Fill in the columns of the Jacobian



- Column 2
  - S2 = [0,0,0,0,1,0]
  - J2 = [Ad_T2] S2

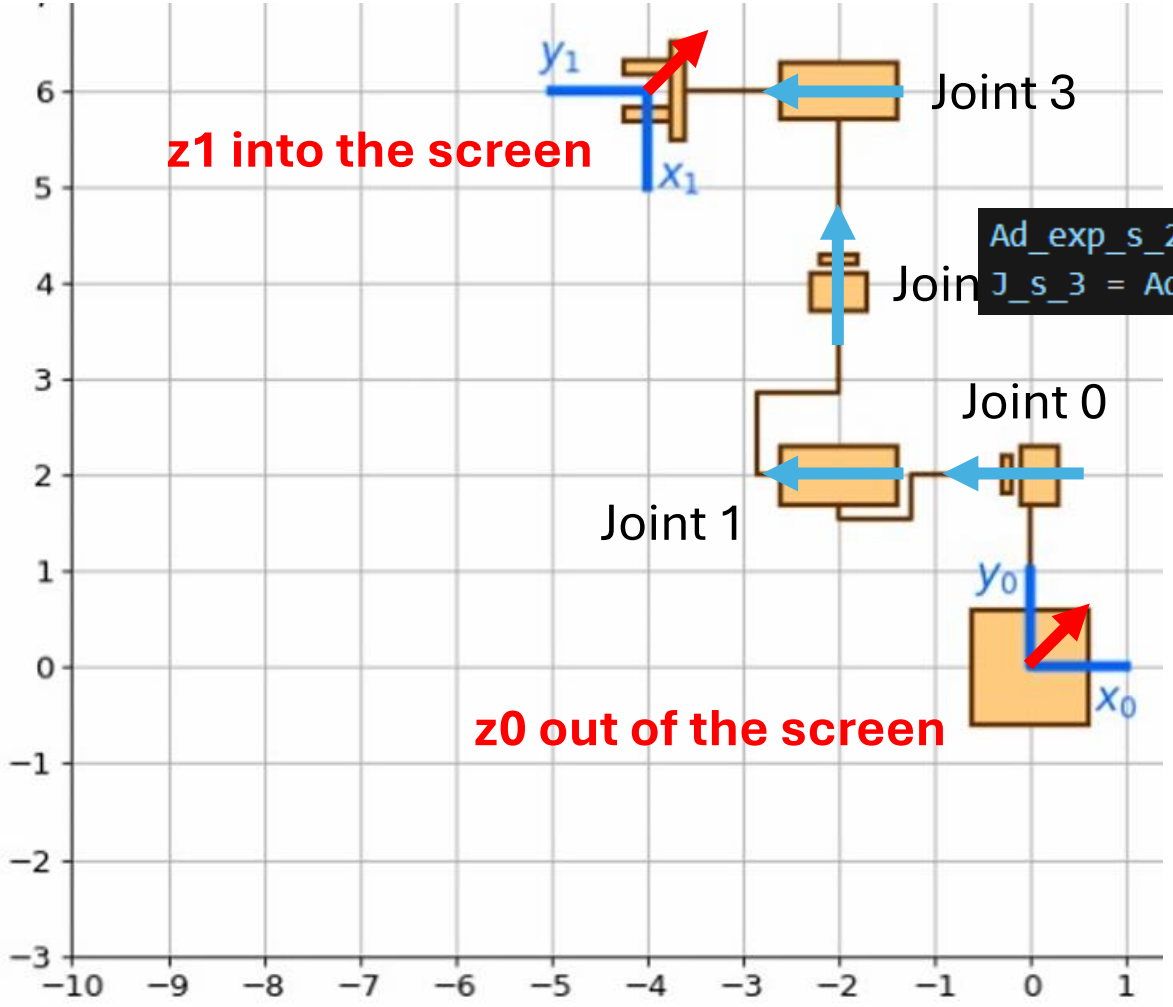
```
Ad_exp_s_1_theta_1 = adjoint(expm(mat_s_1 * theta_1))
J_s_2 = Ad_exp_s_0_theta_0 @ Ad_exp_s_1_theta_1 @ s_2
```

$$[Ad_T] = \begin{bmatrix} R & 0 \\ [p]R & R \end{bmatrix}$$

```
[[ 0.        ]
 [ 0.        ]
 [ 0.        ]
 [ 0.        ]
 [ 0.87758256]
 [-0.47942554]]
```

4 DoF PRPR Robot Example

3. Fill in the columns of the Jacobian



- Column 3
  - S3 = [-1,0,0,0,0,6]
  - J3 = [Ad_T3] S3

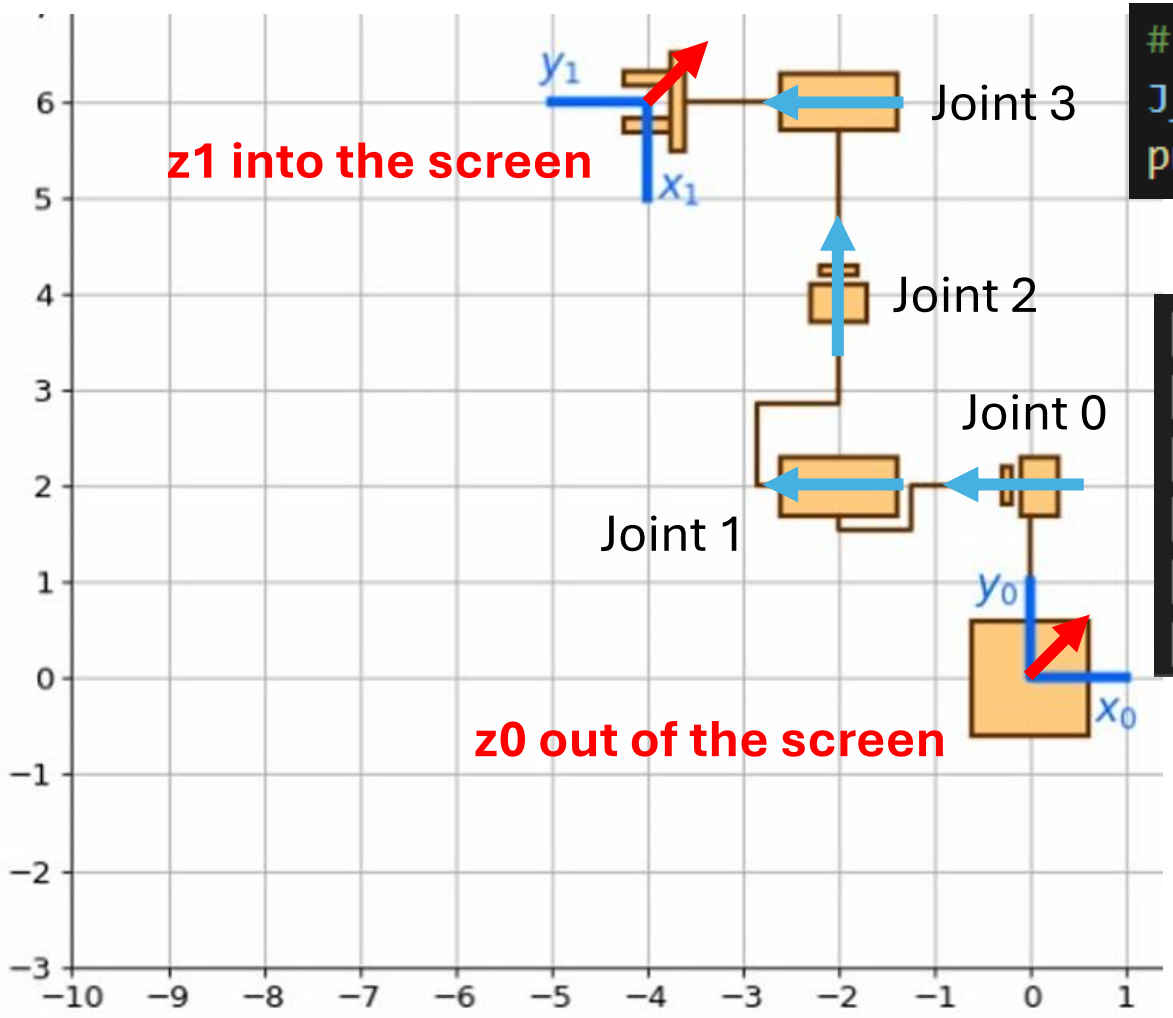
```
Ad_exp_s_2_theta_2 = adjoint(expm(mat_s_2 * theta_2))
J_s_3 = Ad_exp_s_0_theta_0 @ Ad_exp_s_1_theta_1 @ Ad_exp_s_2_theta_2 @ s_3
```

$$[Ad_T] = \begin{bmatrix} R & 0 \\ [p]R & R \end{bmatrix}$$

```
[[-1.        ]
 [ 0.        ]
 [ 0.        ]
 [ 0.        ]
 [ 2.27727131]
 [ 6.16851717]]
```

4 DoF PRPR Robot Example

3. Fill in the columns of the Jacobian




```
# Jacobian matrix (fixed frame)
J_s = np.block([[J_s_0, J_s_1, J_s_2, J_s_3]])
print("Jacobian matrix in the fixed frame J_s:\n", J_s)
```

```
[[ 0.          -1.           0.          -1.        ]
 [ 0.           0.           0.           0.        ]
 [ 0.           0.           0.           0.        ]
 [-1.           0.           0.           0.        ]
 [ 0.           0.           0.87758256   2.27727131]
 [ 0.           2.          -0.47942554   6.16851717]]
```

4 DoF PRPR Robot Example

Computing the spatial twist given joint angle velocities

```python
# Example joint velocities for joints 0, 1, 2, and 3
theta_dot_0 = 0.1
theta_dot_1 = 0.2
theta_dot_2 = 0.3
theta_dot_3 = 0.4

theta_dot = np.array([[theta_dot_0],
                      [theta_dot_1],
                      [theta_dot_2],
                      [theta_dot_3]])

V_s = J_s @ theta_dot

print("Twist V_s given the Jacobian and joint velocities:\n", V_s)
```

```
[[-0.6       ]
 [ 0.        ]
 [ 0.        ]
 [-0.1       ]
 [ 1.17418329]
 [ 2.72357921]]
```

$$\mathcal{V}_s = J_s(\theta)\dot{\theta} = [J_{s,1}\ J_{s,2}\ \cdots\ J_{s,n}] \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \vdots \\ \dot{\theta}_n \end{bmatrix}$$

4 DoF PRPR Robot Example

Computing the body Jacobian

You can either directly compute the body Jacobian columns as follows:

Body Jacobian

$$\mathcal{V}_b = J_b(\theta)\dot{\theta} = [J_{b,1}\ J_{b,2}\ \cdots\ J_{b,n}] \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \vdots \\ \dot{\theta}_n \end{bmatrix}$$

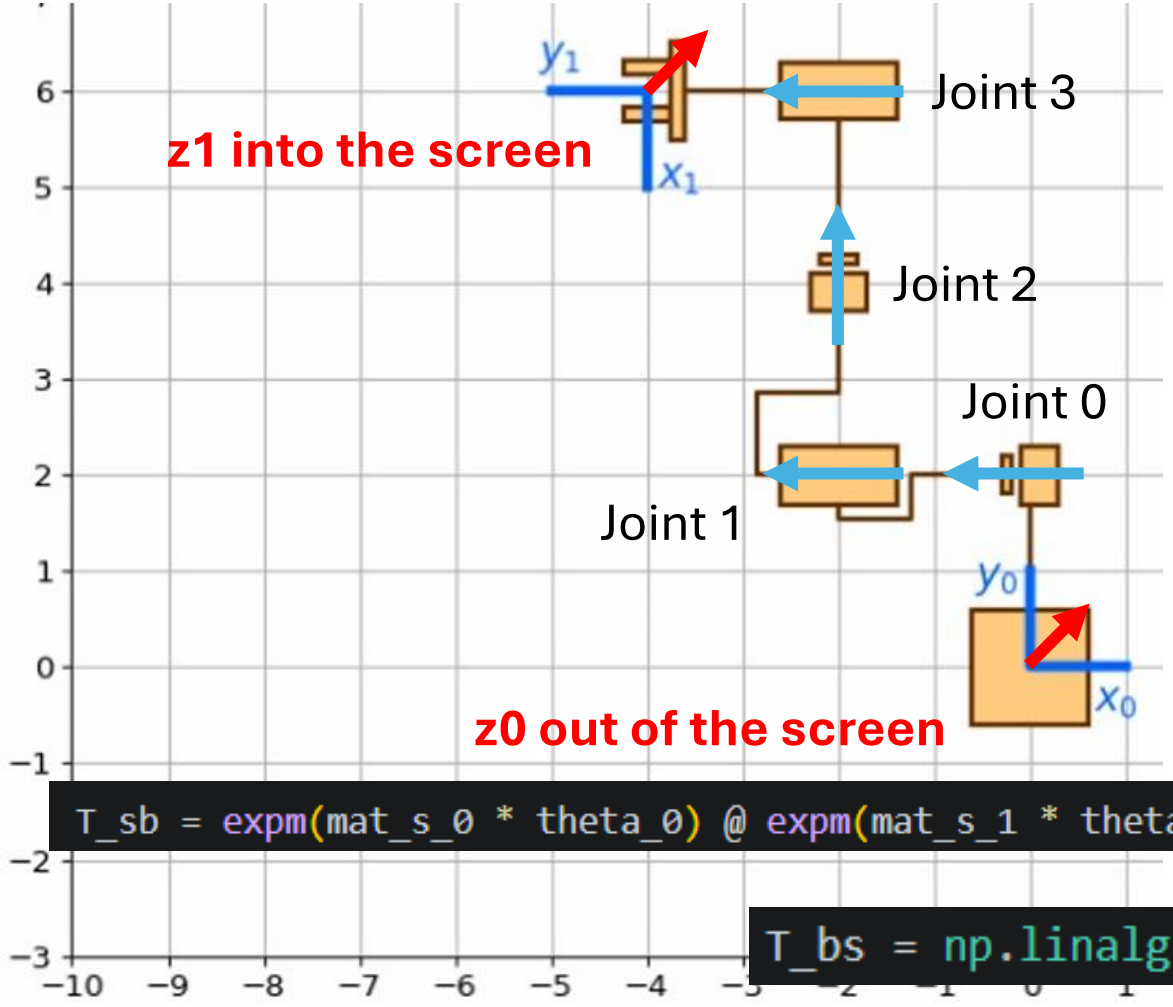
$$J_{b,n} = \mathcal{B}_n \quad J_{b,n-1} = Ad_{e^{-[\mathcal{B}_n]\theta_n}}(\mathcal{B}_{n-1}) \quad \cdots \quad J_{b,1} = Ad_{e^{-[\mathcal{B}_n]\theta_n \ldots e^{-[\mathcal{B}_2]\theta_2}}}(\mathcal{B}_1)$$

Or use the relationship between the fixed and body frames:

$$J_b(\theta) = Ad_{T_{bs}}(J_s(\theta)) = [Ad_{T_{bs}}]J_s(\theta)$$

4 DoF PRPR Robot Example

Finding T_{bs} and using it to solve for J_b



x1 is pointing in -y0 direction
y1 is pointing in -x0 direction
z1 is pointing in –z0 direction

Tool frame is translated by 6 in y0 and -4 in x0 directions

```
# Solving for the body twist
R = np.array([[0,  -1, 0 ],
              [-1, 0,  0 ],
              [0,  0,  -1]])
t = np.array([[-4],
              [6 ],
              [0 ]])
M = np.block([[R, t],
              [0, 0, 0, 1]])
```

```
T_sb = expm(mat_s_0 * theta_0) @ expm(mat_s_1 * theta_1) @ expm(mat_s_2 * theta_2) @ expm(mat_s_3 * theta_3) @ M
```

```
T_bs = np.linalg.inv(T_sb)
```

```
J_b = adjoint(T_bs) @ J_s
print("Body Jacobian J_b:\n", J_b)
```

4 DoF PRPR Robot Example

Computing the body twist given joint angle velocities

```
J_b = adjoint(T_bs) @ J_s
print("Body Jacobian J_b:\n", J_b)
```

```
Body Jacobian J_b:
 [[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  1.00000000e+00  0.00000000e+00  1.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00 -3.99698718e+00 -5.40302306e-01  1.58228342e-16]
 [ 1.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  2.56643595e+00 -8.41470985e-01  8.98254244e-16]]
```

```
V_b = J_b @ theta_dot
```

```
Body twist V_b
 [[ 0.        ]
 [ 0.6       ]
 [ 0.        ]
 [-0.96148813]
 [ 0.1       ]
 [ 0.2608459 ]]
```

$$\mathcal{V}_b = J_b(\theta)\dot{\theta} = [J_{b,1}\ J_{b,2}\ \cdots\ J_{b,n}] \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \vdots \\ \dot{\theta}_n \end{bmatrix}$$

Inverse Kinematics

- **Forward Kinematics:** computes the end-effector position from joint angles:

joint angles $\longrightarrow$ $SE(3)$

$\theta \longmapsto T(\theta)$

- **Inverse Kinematics:** computes the possible joint angles from the pose of the end-effector

$SE(3) \longrightarrow$ joint space

$X \longmapsto \theta$

- Given $T(\theta)$, find solutions θ that satisfy $T(\theta) = X$

soln methods $\longrightarrow$ analytic

$\longrightarrow$ numerical

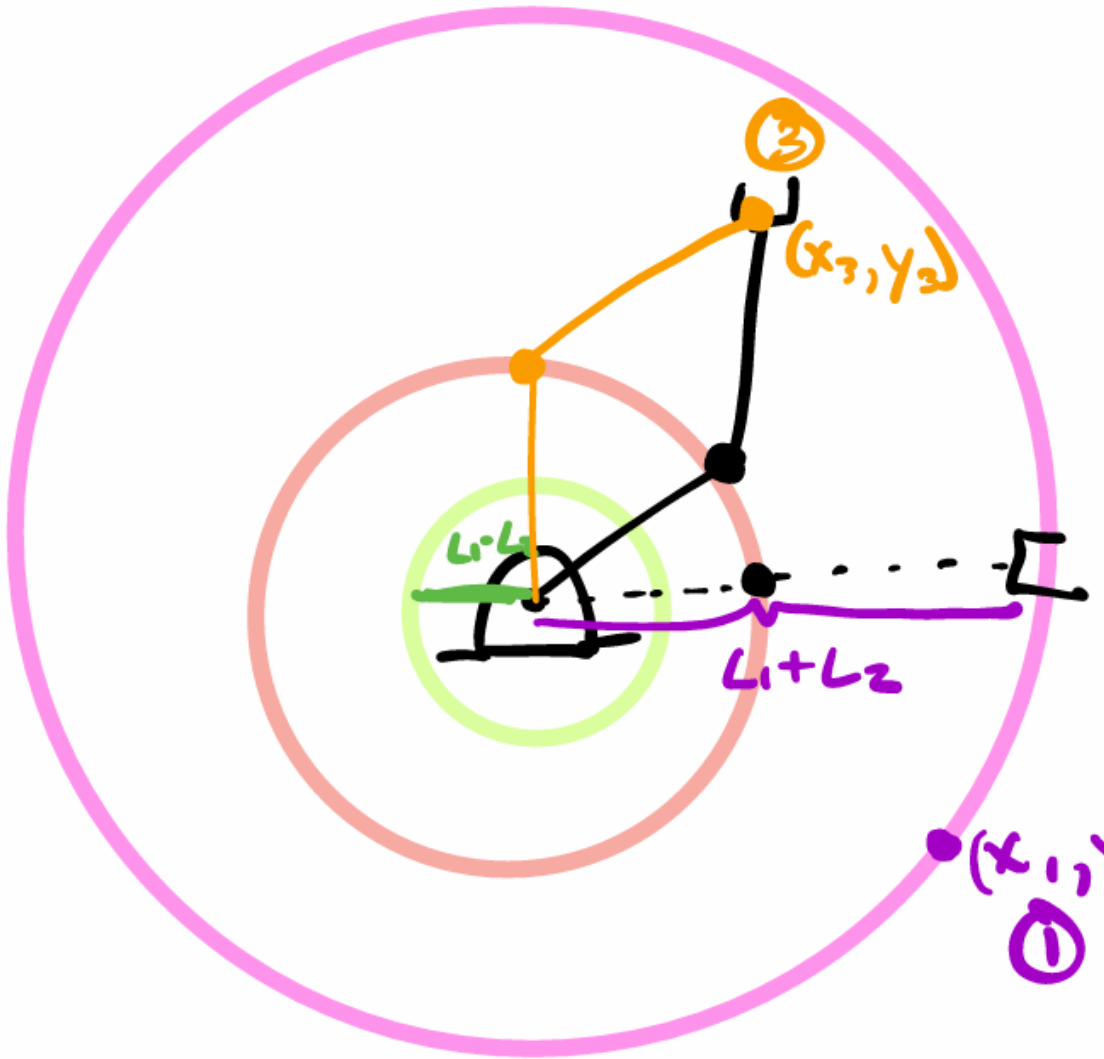
Analytical vs. Numerical IK

Analytical approaches use geometric relationships to find solutions for joint angles that result in a desired end effector pose

Numerical methods iteratively update the guess of the joint angles that result in the desired end effector pose

Numerical methods may be used when analytic approaches are too hard to solve (or may even be impossible)

Inverse Kinematics (IK) Solutions



① one soln

② no soln

③ two solns

→ lefty, righty

→ elbow up/down

→ IK is often multi valued

If the number of joint angles is greater than DOFs of the end effector, there may be an infinite number of solutions because of free variables

Newton-Raphson Method

The Newton-Raphson algorithm is one numerical method we can use to solve IK

We are trying to find the roots of the following equation:

$$f(\theta) - X = g(\theta) = 0$$

We assume that $g(\theta)$ is differentiable and start with an initial guess θ^0

Then we write out the first order Taylor expansion of g at θ^0 (linearization)

$$g(\theta) \approx g(\theta^0) + \frac{dg}{d\theta}(\theta^0) * (\theta - \theta^0)$$

Newton-Raphson Method

Then we write out the first order Taylor expansion of g at θ^0 (linearization)

$$g(\theta) \approx g(\theta^0) + \frac{dg}{d\theta}(\theta^0) * (\theta - \theta^0)$$

And we update our guess for the solution

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

And we keep iterating until we get close enough to the desired result, or the solution doesn't change very much

Newton-Raphson Method for Numerical IK

The same approach but in terms of inverse kinematics:

$$X_d = f(\theta) \approx f(\theta^0) + \frac{df}{d\theta}(\theta^0) * (\theta - \theta^0) + h.o.t$$

$J(\theta^0)$ $\Delta\theta$ 0

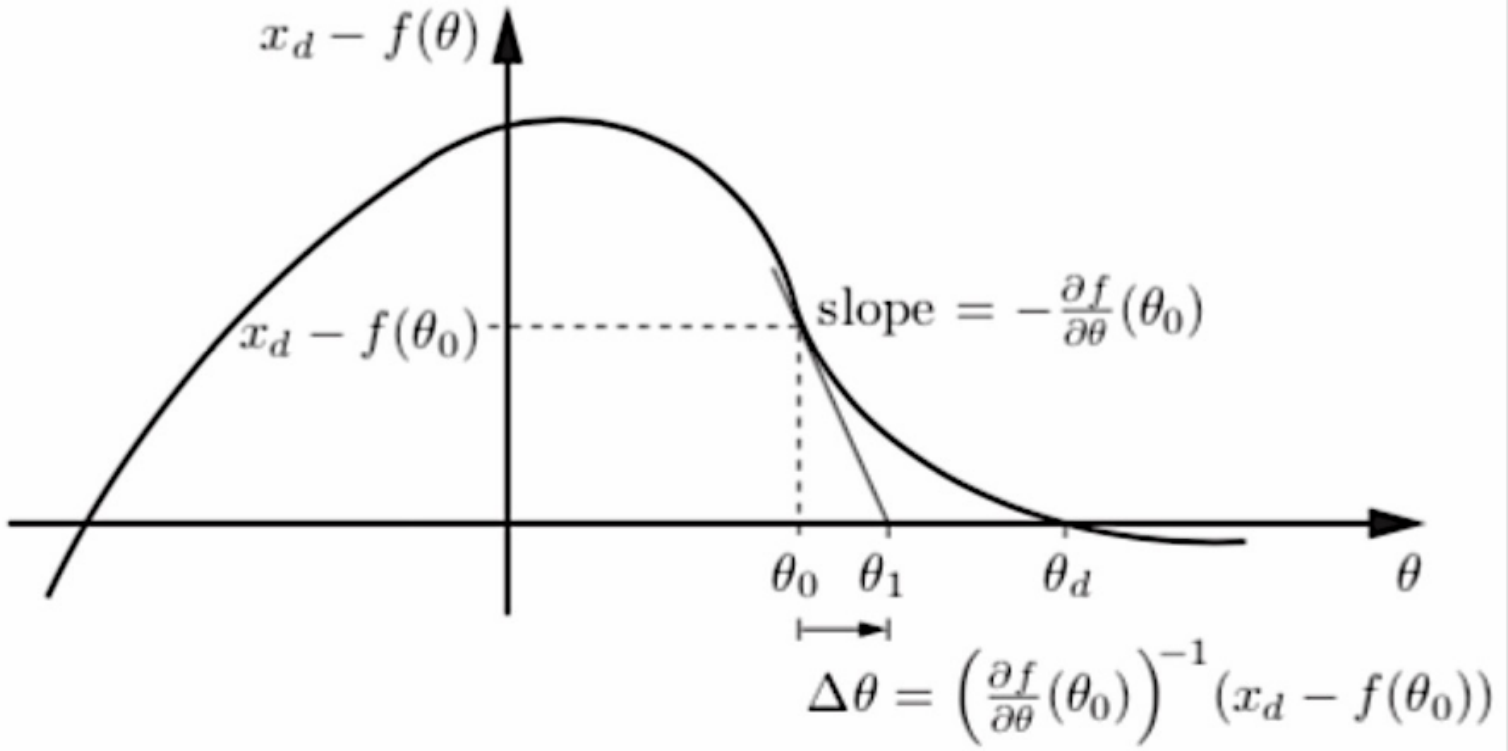
$$X_d - f(\theta) = J(\theta^0)\Delta\theta$$

If the Jacobian J is invertible:

$$J(\theta^0)^{-1}\big(X_d - f(\theta)\big) = \Delta\theta$$

Numerical Inverse Kinematics

- If the forward kinematics is linear in θ (i.e., h.o.t. are zero), then θ^1 exactly satisfies $x_d = f(\theta^1)$
- If the forward kinematics is not linear in θ (as is usually the case), then θ^1 should be closer to the root than θ^0, but will require more iterations



Newton-Raphson Method for Numerical IK

The Newton-Raphson does need to be modified slightly since the desired end effector pose X_d is in SE(3), not just any random 4x4 matrix

We need to find the twist V_b, which if applied for 1 second, goes from our current guess $T_{sb}(\theta^i)$ to the desired pose $X = T_{sd}$

$$T_{sd} = T_{sb} e^{[V_b]}$$

$$\mathcal{V}_b = J_b(\theta)\dot{\theta}$$

$$T_{sb}^{-1}(\theta^i) T_{sd} = e^{[V_b]}$$

$\dot{\theta}$ is our update

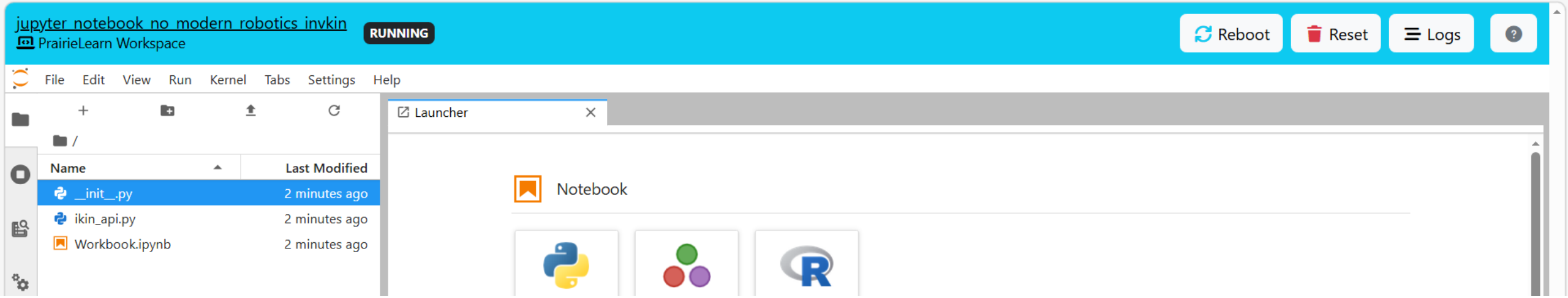
$$\log\left(T_{sb}^{-1}(\theta^i) T_{sd}\right) = [V_b]$$

Numerical Inverse Kinematics: Algorithm

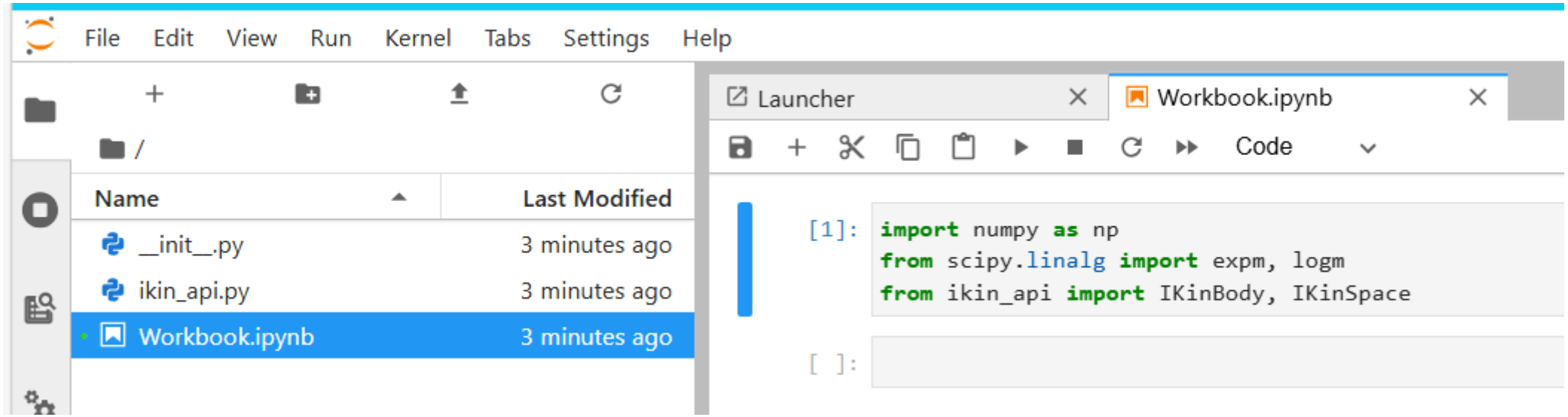
0. Given $X = T_{sd}$ and the forward kinematics map $T_{sb}(\theta)$
   Given tolerances ϵ_ω and ϵ_v
1. Initialize θ^0 and set $i = 0$
2. While $\|\omega_b\| > \epsilon_\omega$ or $\|\omega_v\| > \epsilon_v$
   1. Set $[\mathcal{V}_b] = \log T_{sb}^{-1}(\theta^i) T_{sd}$
   2. Set $\theta^{i+1} = \theta^i + J_b^\dagger(\theta^i)\mathcal{V}_b$
   3. Increment i

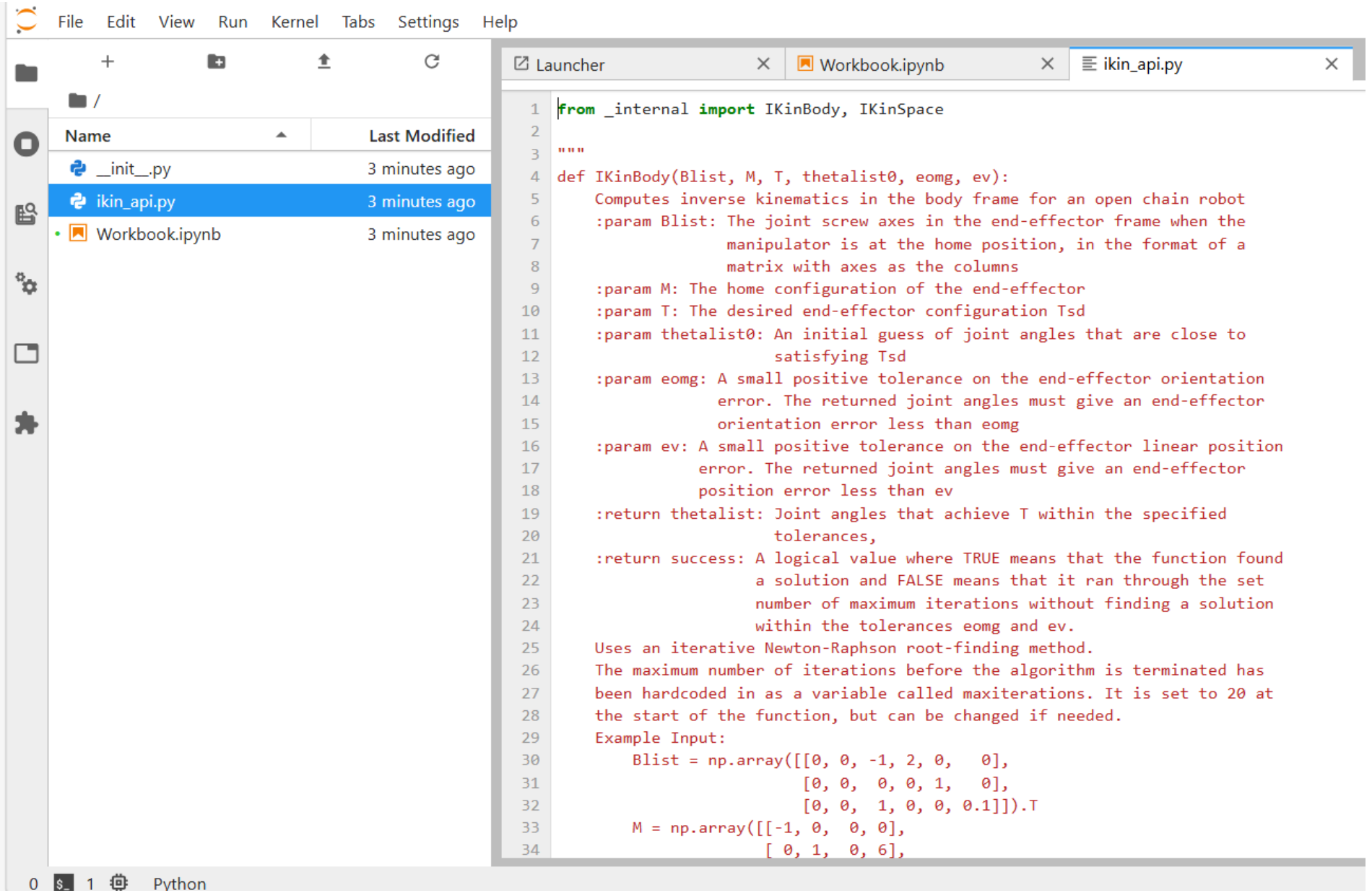
$$V_b = J_b(\theta)\dot{\theta} \quad \longleftarrow \quad J_b(\theta)^{-1} V_b = \dot{\theta}$$

The Jupyter workspace now provides access to functions for inverse kinematics!
This is what you'll see when you startup the server:



In addition to numpy, expm, and logm, we now import IKinBody and IKinSpace by default:





ikin_api.py explains how to call both functions and has examples

You can ignore __init__.py

[1]:
```
import numpy as np
from scipy.linalg import expm, logm
from ikin_api import IKinBody, IKinSpace
```

[2]:
```
# Desired end effector pose - 4x4 homogeneous transformation matrix
T_1in0 = np.array([[0.64469519, 0.00045421, -0.76443961, -1.72006131], [-0.68757886, 0.43735862, -0.57961430, -3.20838971], [0.33407099, 0

# Initial pose of end effector - 4x4 homogeneous transformation matrix
M = np.array([[1.00000000, 0.00000000, 0.00000000, 2.00000000], [0.00000000, 1.00000000, 0.00000000, -4.00000000], [0.00000000, 0.00000000

# Screw axes - 6xNUM_JOINTS where each column is the screw of each joint (in this example, they are in the fixed frame)
S = np.array([[0.00000000, 0.00000000, 1.00000000, 0.00000000, 1.00000000, 0.00000000], [1.00000000, 0.00000000, 0.00000000, 0.00000000, 0

# Initial guess for what each theta should be - list of length NUM_JOINTS
thetalist0 = np.array([0., 0., 0., 0., 0., 0.])

# Tolerance on error of end effector orientation
eomg = 0.01

# Tolerance on error of end effector position
ev = 0.01
```

[5]:
```
thetalist, success = IKinSpace(S, M, T_1in0, thetalist0, eomg, ev)

# success is True if the error of the solution is less than eomg and ev
# thetalist is of shape (NUM_JOINTS,)
# PrairieLearn expects thetalist is a column vector
thetalist = thetalist.reshape((1,-1)).T
print(thetalist.tolist())
print(thetalist.shape)
```

```
[[-0.35111565922256915], [-0.6162067021942831], [0.34969128094474744], [0.9405509369400207], [0.6555028624350827], [-0.17241756757718962]]
(6, 1)
```