UBISOFT LA FORGE

A recipe to cook Jacobian

SHAHIN RABBANI

March 17, 2017

Contents

1 Theory . . . 4
  1.1 Definitions . . . 4
  1.2 Derivation using the Product of Exponentials formula . . . 5
  1.3 Jacobian derivative . . . 6
2 Implementation . . . 6
  2.1 The simplest Jacobian . . . 6
  2.2 Generic Jacobian . . . 12
  2.3 Adjusted method: Combining Spatial Axis and PofE . . . 17
3 Solving Inverse Kinematics . . . 18
  3.1 Formal definition . . . 19
  3.2 Solvers . . . 19
  3.3 Practical notes . . . 21

Jacobians are key component in many motion control and analysis techniques. Named after mathematician Carl Gustav Jacob Jacobi (1804-1851), the Jacobian matrix provides an important tool to best describe a linear approximation of a function f near a point x, given f is differentiable at that point. We use Jacobian extensively in the computer animation and robotics community, and as a result it is worth providing the reader with the technicalities that help understand its derivation and implementation.

While de Lasa (2010) explores different approaches to calculate the Jacobian matrix, we focus on two common methods, namely Product of Exponentials and Spatial Axis theorems, with more emphasis on the former, which is also discussed at length by Murray et al. (1994). Finite difference and unit-velocity methods also exist, but they are not efficient because they do not take advantage of the hierarchical tree structure of the multi-link character. Moreover, the finite difference method is an approximation of the Jacobian that is sensitive to the time step as well as the choice of joint angle perturbation, ε.

This document mostly covers the technical aspects of computing the Jacobian, and wherever possible, diagrams, algorithms and/or pseudo-codes are given to help the reader with practical hints that make the implementation process easier. If you are less interested in the theoretical foundations of computing the Jacobian, you can skip the derivations in Section 1.2, but you are encouraged to read the definitions in Section 1.1.

1 Theory

1.1 Definitions

For a multi-link setup Jacobian relates changes in joint angles, θ, to positional and orientational changes of some Cartesian point, p, on a link i

$$^{i}J(\theta) = \frac{\partial p_i}{\partial \theta}. \tag{1}$$

Jacobian can be described as the result of differentiating the forward kinematics map $g : Q \to SE(3)$. However, this representation is not so easily obtained because $\frac{\partial g}{\partial \theta}$ is not a natural quantity as g is a matrix-valued function. Instead, we can use twists in order to achieve a very natural and explicit description of the Jacobian.

Rearranging 1 and taking time derivative (using the chain rule) we get

$$\phi_i = {}^{i}J(\theta)\dot{\theta}, \tag{2}$$

where $\phi_i = \dot{p}_i$ is defined as the *twist* of the end effector and is written as

$$\phi_i = \begin{bmatrix} v \\ \omega \end{bmatrix} \in \mathbb{R}^6. \tag{3}$$

A twist has a linear velocity $v \in \mathbb{R}^3$ and an angular velocity $\omega \in \mathbb{R}^3$. Twists are the time derivative of a special class of rigid displacements called *screw* motions. We describe a screw motion of a rigid body from one frame to another through combining the translational and the rotational components of the motion along the same axis. In the derivations given in this document we write twists in the body frame b, i.e. at the origin of each link's frame of reference.

We use an *adjoint* matrix to transform a twist from coordinate frame i to j

$$^{j}\phi = {}^{j}_{i}Ad\,{}^{i}\phi. \tag{4}$$

An adjoint matrix transforming between two different coordinate frames is constructed from the transformation matrix corresponding to such change of frame. This is given by

$$^{j}_{i}Ad = \begin{bmatrix} ^{j}_{i}R & {}^{j}\widehat{p}_i\,{}^{j}_{i}R \\ \mathbf{0} & {}^{j}_{i}R \end{bmatrix} \in \mathbb{R}^{6\times 6}, \tag{5}$$

where $^{j}p_i$ is the origin of coordinate frame i in coordinate frame j and the rotation matrix is $^{j}_{i}R \in SO(3)$. The cross product (*wedge*) operator $\widehat{\ }$ is in the form of a skew symmetric matrix

$$\widehat{p} = \begin{bmatrix} 0 & -p_z & p_y \\ p_z & 0 & -p_x \\ -p_y & p_x & 0 \end{bmatrix}. \tag{6}$$

1.2 Derivation using the Product of Exponentials formula

We use the product of exponentials (PofE) formula (Murray et al. (1994)) to derive the Jacobian. An exponential mapping function $e^{\widehat{\zeta}\theta} : \mathbb{R}^6 \rightarrow SE(3)$ on a twist provides an elegant way of describing a homogenous rigid body motion that facilitates the computation of the Jacobian. Let

$$g(\theta) = e^{\widehat{\zeta}_1\theta_1} \cdots e^{\widehat{\zeta}_n\theta_n} g(0) \tag{7}$$

represent the mapping $g : Q \rightarrow SE(3)$. We define *unit twist*, $\widehat{\zeta} \in se(3)$ in the homogenous coordinates by a 4×4 matrix

$$\widehat{\zeta} = \begin{bmatrix} \widehat{\omega} & v \\ 0 & 0 \end{bmatrix}. \tag{8}$$

The columns of the body Jacobian is obtained by taking derivative of 7 with respect to all joint angles

$${}^{b}J(\theta) = \begin{bmatrix} {}^{b}\zeta_1 & {}^{b}\zeta'_2 & \cdots & {}^{b}\zeta'_n \end{bmatrix}. \tag{9}$$

Here, the jth column of J is the derivative of the mapping function $g(\theta)$ with respect to the corresponding joint angle θ_j, where ${}^{b}\zeta'_j \in \mathbb{R}^6$ and is computed using the adjoint matrix

$${}^{b}\zeta'_j = Ad^{-1}_{(e^{\widehat{\zeta}_j\theta_j} \ldots e^{\widehat{\zeta}_n\theta_n} g(0))} \zeta_j. \tag{10}$$

The Jacobian ${}^{b}J(\theta) : \mathbb{R}^n \rightarrow \mathbb{R}^6$ is a configuration-dependent matrix which maps joint velocities to end effector velocities. The jth column of the body Jacobian ${}^{b}J$ is the jth joint twist, transformed to the end effector link frame. Also, we observe that the special structure of the Jacobian results in relating the velocity of the end effector only to the joints that are present down the transformation chain to the root. For the rest of the joints that do not belong to such chain the corresponding columns simply become zero, signifying no contribution. Note that $g(0)$ in 10 appears explicitly. Choosing a spatial coordinate frame such that $g(0) = I$ simplifies the calculation of ${}^{b}J$.

Block-wise computation. In general, one can consider universal joints with 6 degrees of freedom to express the configuration of any multi-body. This is particularly useful to derive a Jacobian that deals with any rigid body system with arbitrary combination of prismatic and revolute joints. In practice, characters often only have revolute joints with only a translation part for the root. In such case, the redundant adjoint columns can be removed from the Jacobian to match the joint vector θ.

In order to design a routine that constructs the body Jacobian, we replace the columns of ${}^{b}J(\theta)$ in Equation 9 with 6×6 adjoint matrix blocks. Particularly, the jth adjoint transforms the jth joint velocity, and is obtained from the transformation function given in 10. The Jacobian becomes

$${}^{b}J = \begin{bmatrix} {}^{b}_{1}Ad & {}^{b}_{2}Ad & \cdots & {}^{b}_{m}Ad \end{bmatrix} \in \mathbb{R}^{6 \times m} \tag{11}$$

where m is the number of the joints.

Jacobian frames. What we have derived so far is the body Jacobian, i.e. the Jacobian for the point at the origin of the link's coordinate frame. To obtain the Jacobian for an arbitrary location point on a link an additional transformation is needed

$$^{p}J(\theta) = {}^{p}_{b}Ad\,^{b}J(\theta). \tag{12}$$

Because any point attached to a link will have the same angular velocity but not necessarily the same linear velocity, ${}^{p}_{b}Ad$ has its rotation set to identity and has only a translational part (for a proof see Appendix B of the PhD thesis by de Lasa (2010)). We can also transform the body Jacobian to a coordinate frame that is more convenient for the end effector manipulation, such as the world frame. Using an adjoint transforming twists from the body frame b to the world frame w gives such Jacobian

$$^{w}J(\theta) = {}^{w}_{b}Ad\,^{b}J(\theta). \tag{13}$$

1.3 Jacobian derivative

The time derivative of $^{b}J(\theta)$ is calculated by taking derivative of the adjoint matrices with respect to time. Given

$${}^{b}_{j}\dot{A}d\,{}^{b}_{j}Ad^{-1} = \begin{bmatrix} \widehat{\omega} & \widehat{v} \\ \mathbf{0} & \widehat{\omega} \end{bmatrix}, \tag{14}$$

the derivative of the adjoint is obtained

$${}^{b}_{j}\dot{A}d = \begin{bmatrix} \widehat{\omega} & \widehat{v} \\ \mathbf{0} & \widehat{\omega} \end{bmatrix} \begin{bmatrix} R & \widehat{p}R \\ \mathbf{0} & R \end{bmatrix} = \begin{bmatrix} \widehat{\omega}R & \widehat{\omega}\,\widehat{p}R + \widehat{v}R \\ \mathbf{0} & \widehat{\omega}R \end{bmatrix}. \tag{15}$$

The routine to construct $^{b}\dot{J}(\theta)$ is the same as $^{b}J(\theta)$ except we use the adjoint time derivative. Details on designing such routine are discussed in the rest of the document.

2 Implementation

In this section we mainly focus on the practical aspects of computing the Jacobian. The goal is to take a step-by-step approach to familiarize the reader with the process of applying the theories explained so far.

2.1 The simplest Jacobian

For the moment, let us forget about the mathematical derivations and try to use our intuition to build a simple Jacobian. After studying an example, we will then discuss how the computed Jacobian relates to our formulas.

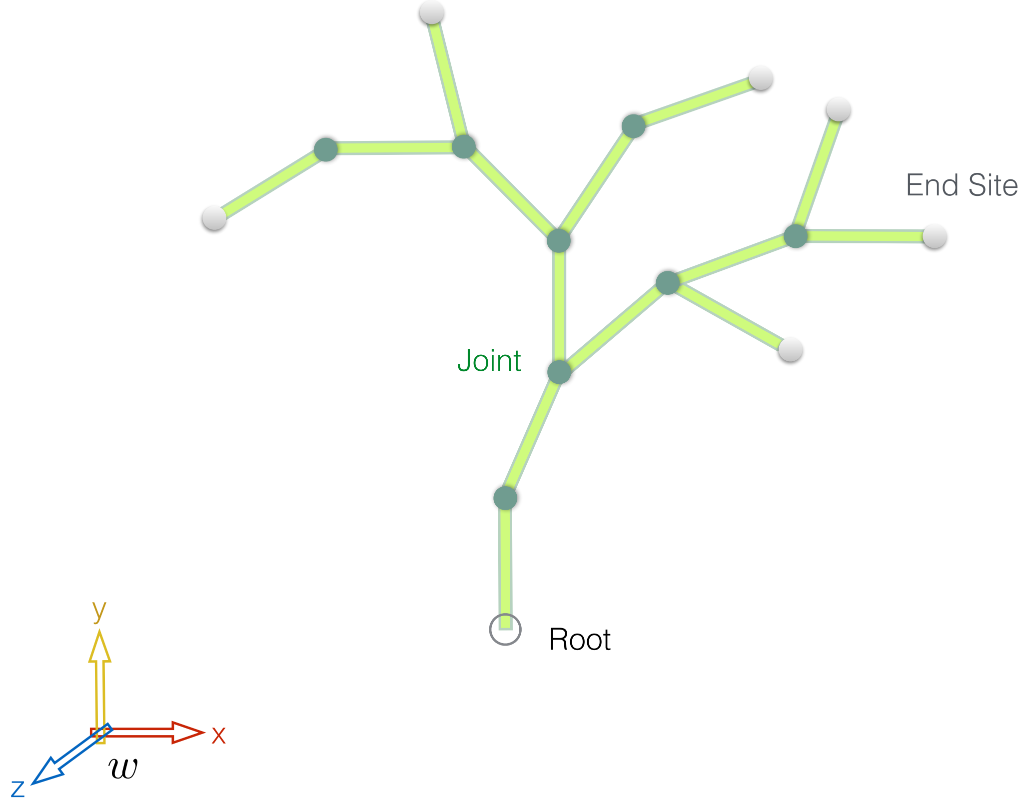

Figure 1: *Abstract tree-shape model with a root, joints (branches) and end sites (leaves),*

Model. Our example workstation model consists of a tree with a floating base (root), 6 end sites (leaves), and 8 spherical joints, that together with the root rotation and translation provide $8 \times 3 + 6 = 30$ DOF. We intentionally choose a different common frame than the root frame, calling it the world frame, to address how to deal with a floating base character where the root linear velocity can also be a part of the solution. This is shown in Figure 1 as the coordinate system with a *w* label. The tree based model is widely used to describe the topology of the human characters, but for the sake of generality, we have chosen a tree shape structure that does not resemble a human skeleton in order to study our methods on an abstract character.

Inverse kinematic problem setup. Figure 2 shows a situation where we would like to gain control over the position of one of the end sites. The selected node is known as the *End Effector*. Although in our example we picked an end site to be our end effector, it can basically be any of the joints, or some point on the link attached to a joint. Also, we assume that we are only interested in the positional adjustments to such node, while the rotational control will be discussed later on.

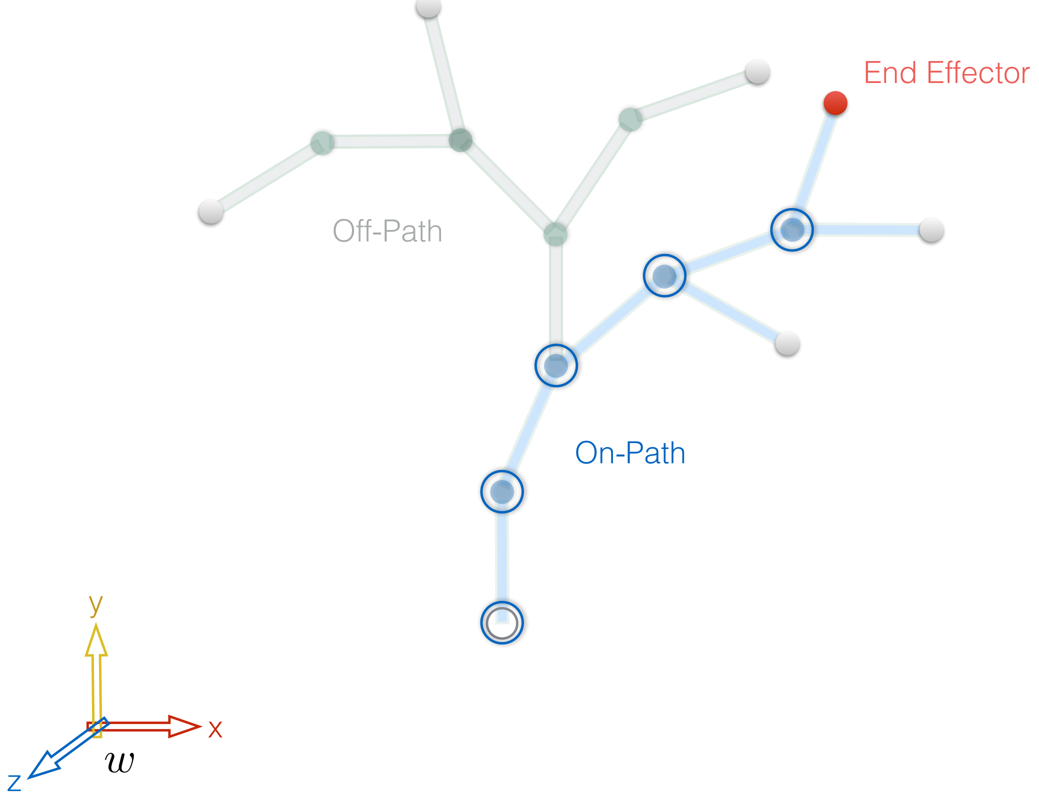

Figure 2: *Separating the tree into the sub-tree containing the end effector (On-Path) and the one that has no relationship with the end effector (Off-Path).*

Identifying the main path. We begin by identifying those joints whose angular changes would directly affect the end effector position. The tree morphology offers the advantage of systematically labeling the joints from the root up to the end effector, leaving out the joints down the other branches. This divides the tree into *On-Path* and *Off-Path* subtrees, as shown in Figure 2, where the contributor joints are highlighted by blue rings. Note that a necessary condition for the On-Path joints is that they should belong to the set of the end effector ancestors, leaving out the children of the end effector, if there are any.

Relating the joint velocities to the end effector velocity. At this stage we compute the relationship between each joint angular velocity and the end effector linear velocity. To that end, we recursively traverse the On-Path joints, starting with the end effector parent and all the way to the root, and compute 3×3 contribution blocks that will find their place in the Jacobian matrix based on the index of each joint. For a spherical joint with x, y, and z rotational axis, such block effectively consists of 3 columns, each corresponding to one of these axes. Since we are solving the problem in 3D each column is a 3×1 vector, obviously. Note that in case of having other types of joint constraints, this block will likely acquire a different size. For example, replacing spherical

joints with hinge joints of 1 DOF will result in a block with a single column, reducing its size to 3×1.

Figure 3 shows an example of computing the contribution of the z axis of one of the On-Path joints to the end effector positional change, or putting it in a more technically correct term, *instantaneous* linear velocity. Assume that all the other joints in the tree, whether On-Path or Off-Path, are frozen except for the selected ith joint. Also, since in the given example we are interested in the effect of the joint angular velocity around its z axis, the rotations around the other two axes, x and y, are also frozen. This reduces the problem to a single hinge with an arm connecting its pivot to the end effector. Choosing the world frame as the common frame, we can denote the end effector position in the world coordinates by ${}^{w}p_e$, and its instantaneous linear velocity by ${}^{w}\dot{p}_e$. Since everything is done in the world frame, we also need to transform the local z axis of the ith joint to the world frame, denoted by ${}^{w}z_i$. As such transformation would require knowledge about the order of the joint axes Euler angles ($\overleftarrow{xyz}$ in our example), we skip this part for the moment and will explain shortly after how to compute ${}^{w}z_i$. Assuming we have found ${}^{w}z_i$, and knowing the contributor joint and the end effector positions in the world frame, ${}^{w}p_i$ and ${}^{w}p_e$, we use a cross-product to compute ${}^{w}\dot{p}_e$ by

$$ {}^{w}\dot{p}_e = {}^{w}z_i \times ({}^{w}p_e - {}^{w}p_i) \tag{16} $$

to be the contribution of the z axis of joint i to the end effector positional change. The resulting 3×1 vector will go to the ith 3×3 contribution block in the Jacobian matrix, locating in either of the 3 vertical locations based on some arbitrary convention set by the user. The other two columns corresponding to the x and y axes will be computed in the similar fashion, except we use ${}^{w}x_i$ and ${}^{w}y_i$ instead of ${}^{w}z_i$ in Equation 16.

In general, the contribution matrix block of the ith contributor joint is

$$ {}^{w}\Gamma_i = \begin{bmatrix} {}^{w}\zeta_x & {}^{w}\zeta_y & {}^{w}\zeta_z \end{bmatrix} \tag{17} $$

where ${}^{w}\zeta_{x,y,z}$ are the computed ${}^{w}\dot{p}_e$ due to x, y and z rotational axes. This is similar to Equation 9 with grouping three columns as one block ${}^{w}\Gamma_i$ pertaining to the ith joint. However, observe that we use a different notation for the columns in this equation where the superscript $\prime$ is dropped to highlight the fact that these are not exactly the same twist vectors as in Equation 9.

Transforming a single joint axis. What we just described assumed that we already know how to properly take care of transforming a single rotational joint axis from a local frame to the world frame. In practice, the transformation matrix should be adjusted such that its rotational components respect the Euler angles order. To understand this point, consider how we do such transformation for the z axis in the above example.

$$ {}^{w}z_i = {}^{w}_{i}E\, {}^{i}z_i \tag{18} $$

where ${}^{i}z_i = (0,0,1)^T$ and

$$ {}^{w}_{i}E = \begin{bmatrix} {}^{w}_{i}p & {}^{w}_{i}R \\ 0 & 1 \end{bmatrix}, \tag{19} $$

is the transformation matrix mapping from the ith joint frame to the world frame, and ${}^{w}_{i}p$ and ${}^{w}_{i}R$ are the translational and rotational components, respectively. However, the problem with Equation 18

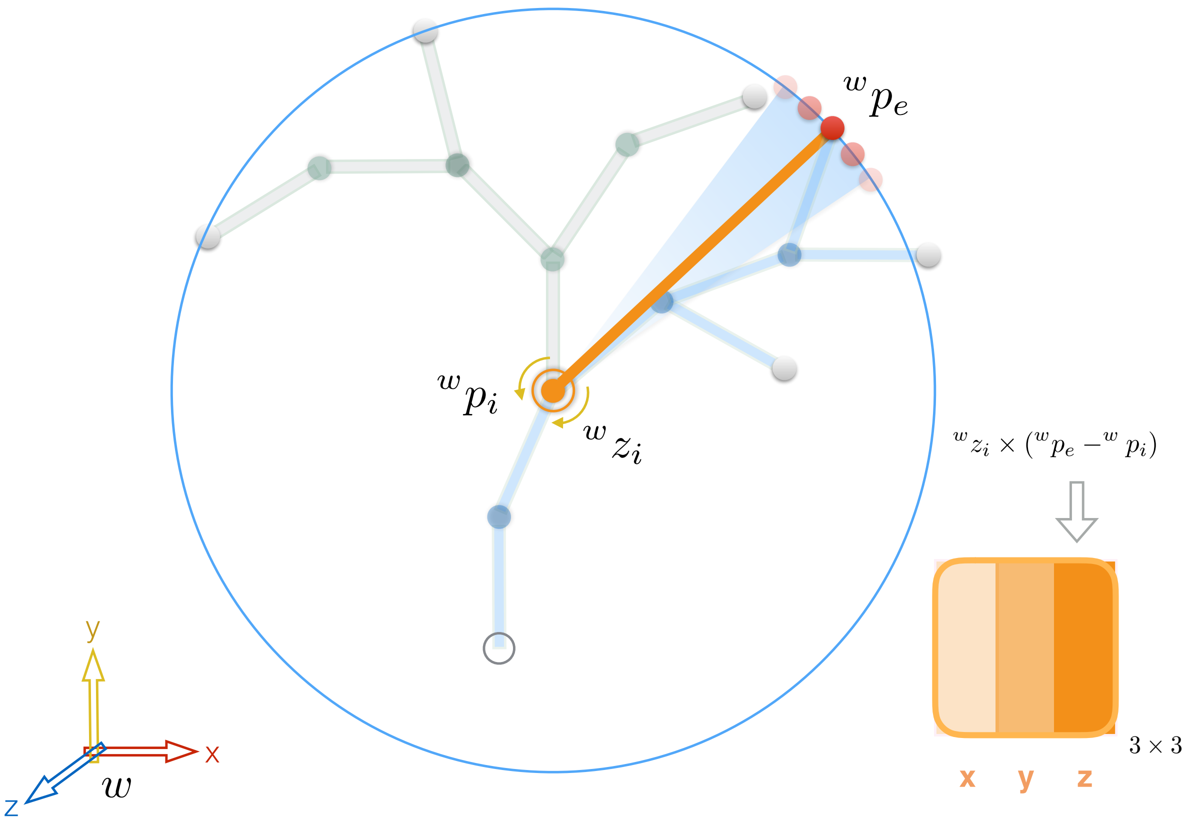

Figure 3: *Using cross product to compute the effects of small angular perturbations to the z axis of a contributor joint j_i on the end effector position p_e in the world frame. This problem can be viewed as freezing all other DOFs, reducing the computation to a single hinge that rotates an arm going from the joint position $^w p_i$ to the end effector position $^w p_e$.*

is the mismatch between the transformation matrix dimension (4×4) and the joint axis dimension (3×1). The reason is using homogeneous coosrdinates system for the transformation matrix. To obtain a consistent from, we need to augment the z axis with 0 (as opposed to 1 for points): ${}^i z_i{}' = (0,0,1,0)^T$. It is straightforward to show that with everything adjusted to be in homogeneous system we can ignore the translational part of ${}^w_i E$ and re-write Equation 18 as

$$ {}^w z_i = {}^w_i R \; {}^i z_i. \tag{20} $$

Note that we do not have to use the augmented vector ${}^i z_i{}'$ in this case as ${}^w_i R$ is 3×3. Yet, ${}^i z_i{}'$ helps to reduce Equation 18 to Equation 20. Also, as a careful manipulation of the rotation matrix based on the channel order only involves adjusting the local rotation matrix from the current joint i to its parent p, we can write

$$ {}^w z_i = {}^w_p R \; {}^p_i R \; {}^i z_i = {}^w_p R \; {}^p z_i. \tag{21} $$

and then only focus on how ${}^{p}z_i$ is computed. To address the issue of Euler angles order let us go further and break down the local rotation matrix ${}^{p}_{i}R$ in terms of individual rotations around each axis. For a channel order $\overleftarrow{xyz}$ we get

$$ {}^{p}z_i = ({}^{p}_{i}R_x \ {}^{p}_{i}R_y \ {}^{p}_{i}R_z)^{i}z_i. \tag{22} $$

This means the rotation order $\overleftarrow{xyz}$ requires us to first rotate around z, then y, then x axis. Since the z axis is the right-most Euler angle, any rotation around x or y would affect the z axis, and hence transforming the z axis will need to include the other two rotations as well. On the other hand, for transforming the y axis, rotations around z will not influence the transformation, and hence becoming identity matrix

$$ {}^{p}y_i = ({}^{p}_{i}R_x \ {}^{p}_{i}R_y \ \mathbf{I})^{i}y_i = ({}^{p}_{i}R_x \ {}^{p}_{i}R_y)^{i}y_i. \tag{23} $$

Likewise, for the x axis we have

$$ {}^{p}x_i = ({}^{p}_{i}R_x \ \mathbf{I} \ \mathbf{I})^{i}x_i = ({}^{p}_{i}R_x)^{i}x_i \tag{24} $$

For ${}^{p}z_i$ with a different channel order, say $\overleftarrow{zxy}$, the rotation matrix break-down would become

$$ {}^{p}z_i = ({}^{p}_{i}R_z \ {}^{p}_{i}R_x \ {}^{p}_{i}R_y)^{i}z_i, \tag{25} $$

and following the same approach as above, we would have needed to exclude the x and y rotations to get

$$ {}^{p}z_i = ({}^{p}_{i}R_z \ \mathbf{I} \ \mathbf{I})^{i}z_i = ({}^{p}_{i}R_z)^{i}z_i. \tag{26} $$

In general, one can say computing the local axis rotation is the part that needs most attention. Any mismatch between the Euler angles enforced by the character model and computing the transformation of the joint individual axis is potentially the main cause of failing to compute the Jacobian properly.

Putting everything together. At his point we have addressed everything regarding computing a Jacobian block matrix for a contributor joint. Algorithm 1 summarizes the routine to build the Jacobian. Figure 4 uses a color code to demonstrate how the On-Path joints are traversed and how each ${}^{w}\Gamma$ block is put in the appropriate place in J, matching the joint angles order in $\dot{\theta}$. For a specific color code, there are 3 different shades for each axis. The grey blocks indicate zero matrices for the Off-Path joints. As we are dealing with a floating base root, we also need to include the contribution of each of the root translational components in both J and $\dot{\theta}$. We can choose any part of the Jacobian to place a 3×3 root translation contribution (shown by light blue) as long as $\dot{\theta}$ follows the same order. Conventionally this block is either placed in the left-most or the right-most position of J. Each column of this block matrix is basically a unit vector because of the one-to-one correspondence between a unit change along each root translational component and the unit positional change for the end effector (for instance we insert $(1,0,0)^T$ for the x axis column in the allocated Jacobian root block). Note that while the order of rotational channels in J and $\dot{\theta}$ should match, it remains an arbitrary decision that only depends on the user's preference of how to set them up. This means the column order of the Jacobian matrix is not necessarily the

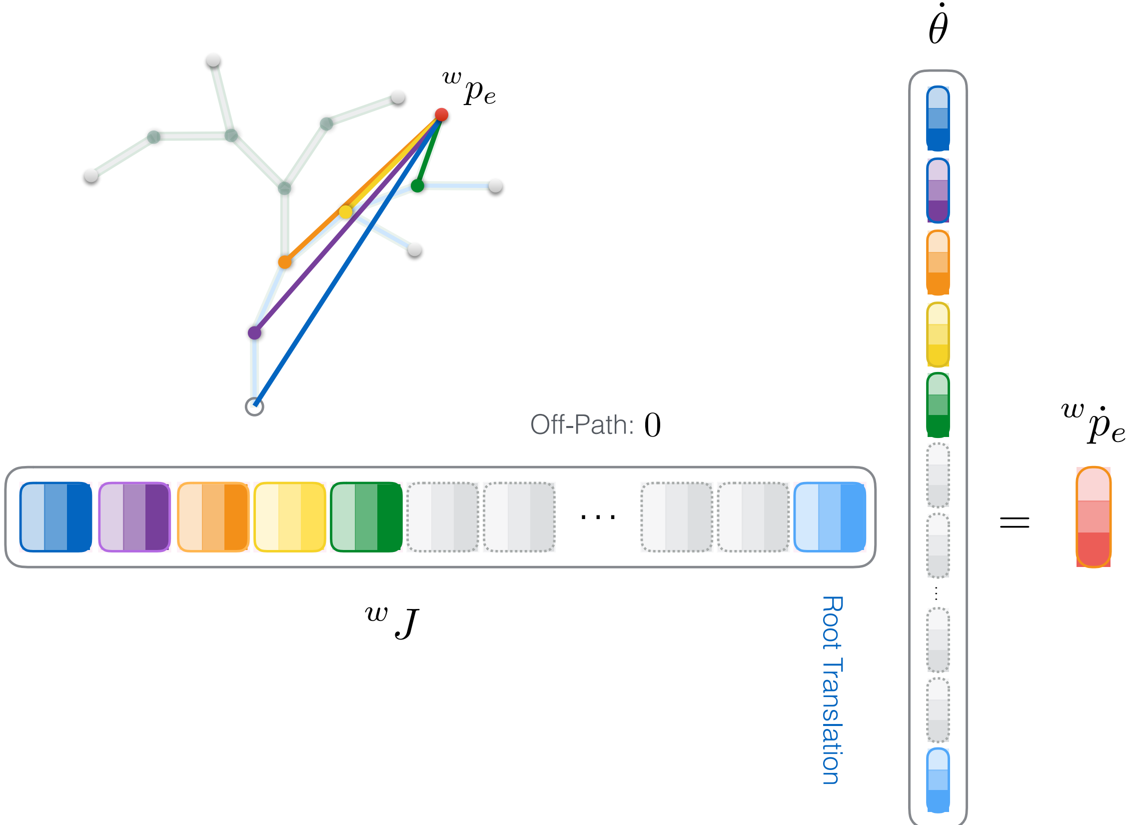

Figure 4: *Inverse kinematics problem setup with only revolute joints and only positional control over the end effector. Jacobian J and the end effector linear velocity ${}^{w}\dot{p}_e$ are computed in the world frame w. Blocks are ${}^{w}\Gamma_i$ from Equation 17. The color code visualizes the correspondence between the joint blocks and the joint angles.*

same as the Euler angles order. The former depends on the way we define the problem, and the latter depends on the rotational channel order inherited from the model design.

2.2 Generic Jacobian

By far you have learnt how to assemble a basic Jacobian with only revolute joints and only positional control over the end effector. In general, we might be dealing with a system with a mixture of prismatic and revolute joints, as well as requiring to gain control over both orientation and the position of the end effector. The cross product method explained above, although simple to implement and accurate, can not be generalized in a simple manner to address more general problems. Also, this method does not help with other rotational representations, such as quaternions and rotation matrices. To achieve a more generic Jacobian we need to go back to Equation 11. In what follows

Procedure 1 Simple Jacobian

1: ${}^{w}J \leftarrow 0$
2: ${}^{w}p_e \leftarrow \textsc{TransformToWorld}(b)$ // end effector position in world
3: $j \leftarrow b$ // iterator j starts with end effector b
4: **while** j has parent **do**
5: $\quad j \leftarrow \textsc{Parent}(j)$
6: $\quad {}^{w}p_j \leftarrow \textsc{TransformToWorld}(j)$ // joint position in world
7: $\quad {}^{w}d = {}^{w}p_e - {}^{w}p_j$ // rotation arm in world
8: $\quad$ **for** $axis : x, y, z$ **do**
9: $\qquad {}^{w}axis \leftarrow \textsc{TransformAxisToWorld}(axis, j)$ // Equations 20 to 26
10: $\qquad {}^{w}\zeta_j = {}^{w}axis \times {}^{w}d$ // one column of Equation 17
11: $\qquad {}^{w}\Gamma_j \leftarrow \textsc{setBlock}(axis, {}^{w}\zeta_j, {}^{w}\Gamma_j)$ // Equation 17
12: $\quad$ **end for**
13: $\quad index \leftarrow \textsc{getIndex}(j)$ // the joint block index
14: $\quad {}^{w}J \leftarrow \textsc{setMatrixBlock}(index, {}^{w}\Gamma_j, {}^{w}J)$ // setting the joint contribution block
15: **end while**
16: $\mathbf{I} \leftarrow \textsc{MakeIdentity}(3, 3)$ // root translation: identity matrix
17: $index \leftarrow RootTranslationIndex$
18: ${}^{w}J \leftarrow \textsc{setMatrixBlock}(index, \mathbf{I}, {}^{w}J)$

we will examine how to compute such generic Jacobian, and we will see how the cross-product method is just a special case of this generic technique.

The generic Jacobian we are going to build consists of 6×6 adjoint blocks that are either inserted in the Jacobian as a whole or re-adjusted before insertion based on the joint types and the sort of end effector velocity we want to control. This is shown in Figure 5. As introduced in Equation 5, an adjoint block is in charge of mapping spatial velocity quantities, known as *twists*, without engaging us in the cumbersome details, like the cross-product method explained before.

Assembling the adjoints. An adjoint contains four 3×3 sub-blocks: two rotations, one rotated cross product, and one zero block. To compute an adjoint mapping twists from frame a to frame b (${}^{b}_{a}Ad$), we first compute the transformation matrix with the same mapping (${}^{b}_{a}E$), then extract the rotational and translational components from ${}^{b}_{a}E$ to build ${}^{b}_{a}Ad$. While it is straightforward to copy the rotational components from ${}^{b}_{a}E$ to ${}^{b}_{a}Ad$, the rotated cross-product block should be computed by first forming $\widehat{p}$ from Equation 6 and then multiplying it by the rotation matrix to get $\widehat{p}R$.

Adjoints are suitable for computing J. Since adjoints transform twists, we can consider transforming joint velocities (ζ), to end effector velocities (ϕ), each with one or both of the linear and angular components. As shown in Figure 5, we can arbitrarily add or remove joint linear or angular velocities, denoted by $\dot{d}$ and $\dot{\theta}$, as well as including or excluding control over the end effector linear or angular velocity, shown by v and ω respectively, to achieve a wide verity of arrangements in our problem setup. Note how we trim the adjoints in the diagram to match the actuation and control dimensions in different example scenarios. The trimmed sub-blocks, or the complete adjoint

Figure 5: *Several scenarios for adjoint block extraction. First row: full adjoint for a universal joint and full control over the end effector linear and angular velocities. Second row: only revolute joint but still full velocity control over the end effector. Third row: only prismatic joint and again full end effector velocity control. Pay attention to* ω *that despite being included in control is forced to be 0 by the setup. This makes sense as a prismatic joint can not affect the angular velocity of the end effector. Last row: a very common setup with only a revolute joint and positional control over the end effector.*

matrix in the full control scenario shown on top, will be inserted in the Jacobian matrix without any further adjustments. This provides a nice and generic routine with minimal ad-hoc design decisions. The procedure is given in Algorithm 2. The given procedure iterates through all joints in the hierarchy that affects a link of interest computing the relevant body adjoints. Those joints with no contribution to the desired link velocity are considered frozen. This means the corresponding adjoint blocks in the Jacobian are zero.

You can compare this algorithm to Algorithm 1 where you probably find the new method easier to implement. Note the left superscripts in both algorithms. Algorithm 1 uses w, world frame, and Algorithm 2 uses b, body frame. Also, pay attention to the case of computing the root block, where we need to make sure *trim* variable always includes the root translation.

Procedure 2 Generic Jacobian

1: ${}^{b}J \leftarrow 0$
2: $trim \leftarrow$ SETDIM$(ctrl, dof)$ // set the trim dimensions
3: $j \leftarrow b$ // iterator j starts with end effector b
4: **while** j has parent **do**
5: $j \leftarrow$ PARENT(j)
6: ${}^{j}_{b}E \leftarrow$ TRANSFORMFROMTO(b, j) // local transformation from b to j
7: ${}^{b}_{j}Ad \leftarrow$ ADJOINT$({}^{j}_{b}E^{-1})$ // adjoint from j to b
8: $index \leftarrow$ GETINDEX(j)
9: ${}^{b}J \leftarrow$ SETMATRIXBLOCK$(index, {}^{b}_{j}Ad, {}^{b}J, trim)$ // setting the matrix block
10: **end while**

Body frame instead of world frame. In Section 1.2 we used PofE to derive a Jacobian based on mapping twists. However, we did not mention why we use the body frame instead of the world frame to be our common coordinates system. While in general the choice of the common frame does not affect the computed Jacobian accuracy (except for very small numerical errors that are safe to be ignored), it might determine if we compute J in an efficient manner.

Adjoints by nature are computed from local transformations between two joints, without the need to first map all the quantities to a more general frame, like the world frame. As Figure 6 shows, when using Algorithm 1 we need multiple passes (at least two) to transform our matrices to the world frame, while we only need 1, often shorter, pass in Algorithm 2 to map our quantities to the body frame. While in both cases we can take advantage of the recursive structure of the tree to reuse the results of the previous steps, less and shorter passes give intrinsic performance improvement to our code.

Relationship to the cross-product method. You might have noticed the last row of Figure 5 resembles what we did for creating the simple Jacobian. It involves a cross product and only deals with revolute joints and the end effector linear velocity. This is not a coincidence. In fact, we can show the method we used was a special case of the PofE method.

Recall that a column of the body Jacobian formula in Equation 9 was given as

$$ {}^{b}\zeta'_{j} = Ad^{-1}_{(e^{\hat{\zeta}_j \theta_j} \cdots e^{\hat{\zeta}_n \theta_n} g(0))} \zeta_j \tag{27} $$

which means an adjoint transforming the jth twist ζ_j to the body frame b. Take note that one should not confuse the jth axis in this equation with the jth joint in the algorithm discussed earlier. To clarify this, we can rewrite Equation 27 for, say, the z axis of the jth contributor joint, while using a more familiar adjoint notation that we just introduced

$$ {}^{b}\zeta'_{j_z} = {}^{j_z}_{b}Ad^{-1} \zeta_{j_z}. \tag{28} $$

Here, ζ_{j_z} means the twist due to the z axis of the joint j, and ${}^{j_z}_{b}Ad^{-1}$ is the inverse of an adjoint transforming the body frame b to the z axis of the joint j frame, shown by j_z. Applying additional

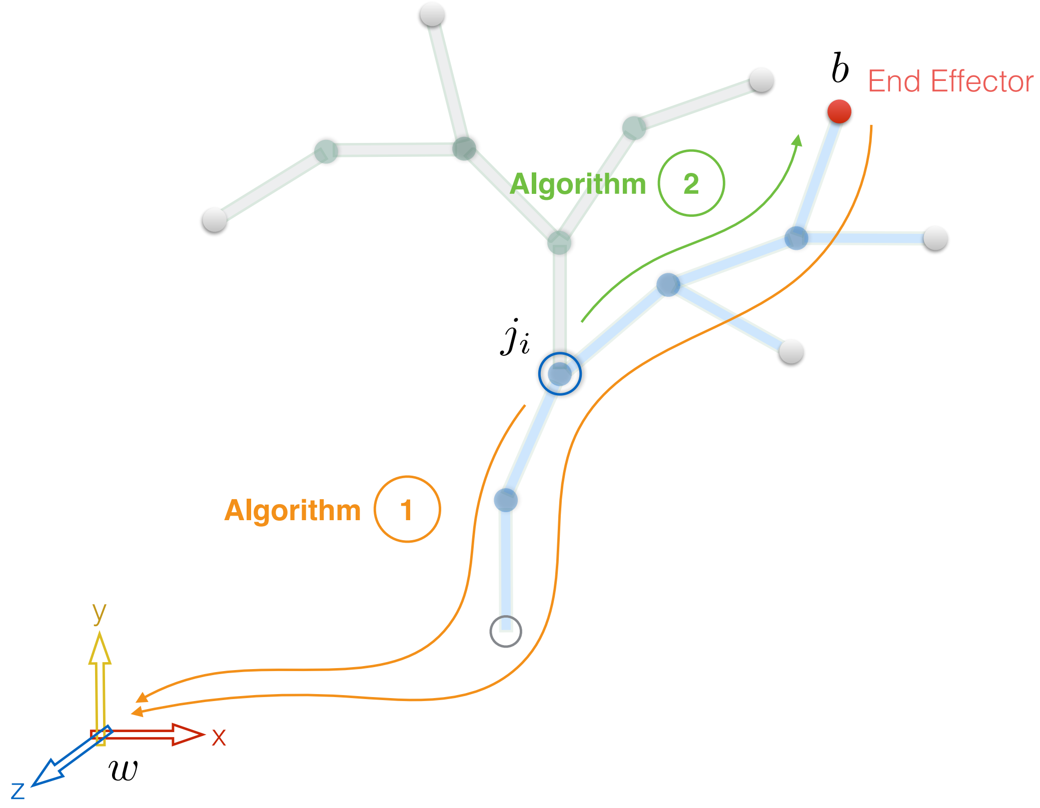

Figure 6: *Comparing the passes for the two algorithms.*

transformation to Equation 28 we get the contribution twist in the world frame

$$^{w}\zeta_{j_z} = {}^{w}_{b}Ad({}^{j_z}_{b}Ad^{-1}\zeta_{j_z}) = {}^{w}_{j_z}Ad\ \zeta_{j_z}. \tag{29}$$

We took liberty to drop $'$ of the left hand side twist because it is now a different vector. End effector contribution twist $^{w}\zeta_{j_z}$ is the same as the third column of the contribution block we computed for the simple Jacobian method in Equation 17. The other two columns for x and y are obtained the same way. This shows how the cross product method can be derived using the same technique as PofE. However, the key observation here is computing ${}^{w}_{j_z}Ad$, which is slightly different from ${}^{w}_{j}Ad$. While the latter does not take any Euler angles order into account and is directly achieved from ${}^{w}_{j}E$, the former assumes different transformation matrices when mapping from x, y or z rotational axis, due to the existence of some Euler angles order.

Although the generic Jacobian provides many advantages, and being particularly useful for dealing with joint constraints in quaternion or matrix forms, it is somewhat not very straightforward how to use it with Euler angles. In the above discussion if we had ${}^{w}_{j_x}Ad$, ${}^{w}_{j_y}Ad$ and ${}^{w}_{j_z}Ad$ we could have easily solved this problem. But what does it mean to compute a Jacobian from adjoints for a joint axis respecting a certain Euler order? What follows in the next section proposes a comprehensive

solution to generalize our method even more so that the Jacobian can be built for any rotational representation.

Summary. Advantages over the cross-product method:

- directly computing adjoints from transformations.
- block-wise opertations
- more convenient to pick the body frame as the common frame (most conventional Jacobians are computed this way), providing more local methods to compute Jacobian.
- provides an analytical way for computing Jacobian derivative.
- compute Jacobian from other representations than Euler angles, such as quaternions and rotation matrices.

2.3 Adjusted method: Combining Spatial Axis and PofE

The aim of this section is to add a few lines to Algorithm 2 such that the PofE method can also address the Euler angle representation. Particularly, we add additional steps between lines 7 and 8 of Algorithm 2 to implement the modifications. The presented approach is inspired by, but differs from, the Spatial Axis method discussed by Featherstone (2008) as well as de Lasa (2010).

In the computation of the generic Jacobian, we used to focus on the local path between joint j_i and end effector b, constructing the adjoint operator from the corresponding transformation. To account for the contribution of a single rotational axis of j_i, we take an extra step to study the effects of axial changes of j_i in its parent frame j_{i_p}. Similar to what we explained for transforming a single joint axis in Section 2.1 (Equations 20 to 26) we compute single axis transformations for each of the joint x, y and z axes in the parent frame, denoted by ${}^p x_i$, ${}^p y_i$ and ${}^p z_i$. We then define *local contribution* rotation matrix by combining these vectors to be its columns

$$ {}^{p}_{i}R' = \begin{bmatrix} {}^p x_i & {}^p y_i & {}^p z_i \end{bmatrix} \tag{30} $$

We use $'$ on the constructed rotation to emphasize its difference from the actual rotation of j_i in j_{i_p} frame, shown as ${}^{p}_{i}R$. Using these two rotations we formulate an *adjustment* rotation as

$$ {}^{i}_{i}R^* = {}^{p}_{i}R^T \; {}^{p}_{i}R'. \tag{31} $$

Having computed adjoint ${}^{b}_{j}Ad$ from line 7 of Algorithm 2, we right-multiply its rotational component by ${}^{i}_{i}R^*$ to get an extended rotation

$$ {}^{b}_{i}R' = {}^{b}_{i}R \; {}^{i}_{i}R^* \tag{32} $$

and use it to update the adjoint.

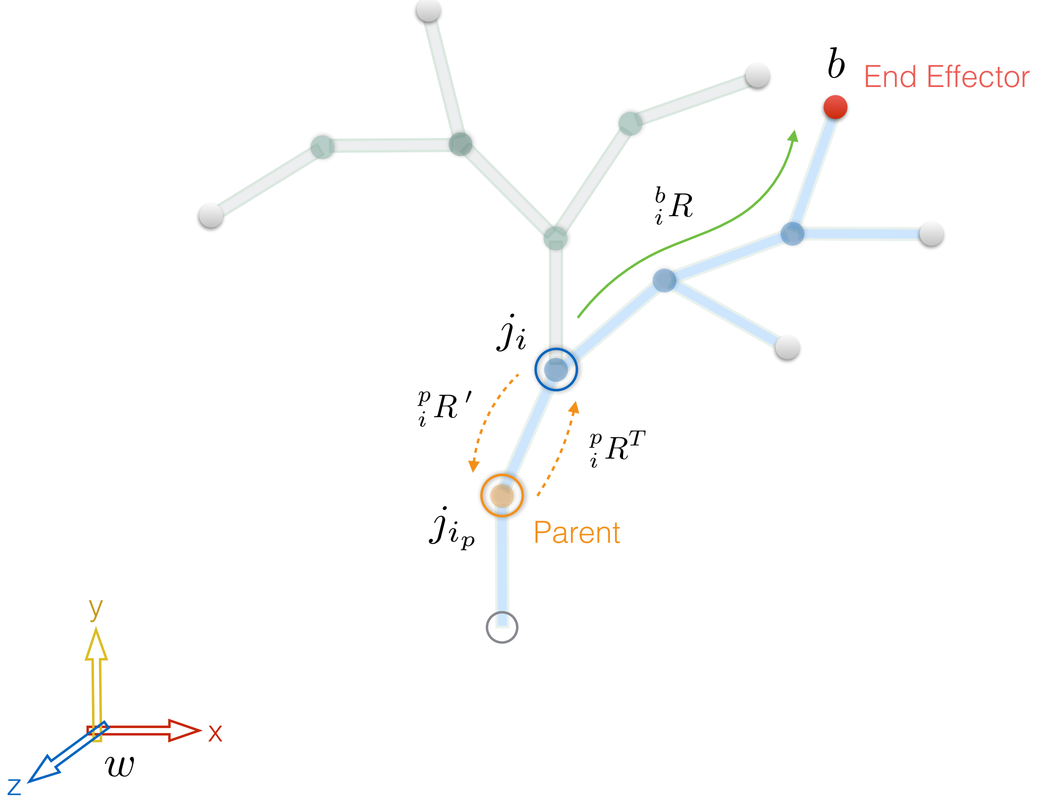

Figure 7: *Computing the adjustment rotation by taking an extra step to include the parent of the current contributor joint into account.*

The idea is shown in Figure 7. It goes without saying that although our modification does not affect the translational part of the adjoint, we need to recompute $\widehat{p}R$ block of the adjoint with the new rotation. Algorithm 3 lists the extra steps between lines 8 and 15. The reader should specifically pay attention to line 9 of this procedure. It is similar to line 9 of Algorithm 1, yet we transform individual axes to parent instead of world. Also, this is the only part that depends on the Euler angles order in our algorithm.

The nice property of the proposed adjustment is that $^i_i R^*$ automatically becomes identity if the Euler angles order would not matter, reducing our algorithm back to its generic form.

3 Solving Inverse Kinematics

So far we have investigated different methods to compute the Jacobian. In the following parts we briefly give guidance on setting up in inverse kinematic (IK) problem with a few practical hints.

Procedure 3 Adjusted Jacobian

```
1:  ^bJ ← 0
2:  trim ← setDim(ctrl, dof)                               // set the trim dimensions
3:  j ← b                                                  // iterator j starts with end effector b
4:  while j has parent do
5:      j ← Parent(j)
6:      ^j_bE ← TransformFromTo(b, j)                      // local transformation from b to j
7:      ^b_jAd ← Adjoint(^j_bE^{-1})                       // adjoint from j to b
8:      for axis : x, y, z do
9:          ^paxis ← TransformAxisToParent(axis, j)        // similar to Equations 20 to 26
10:         ^p_jR' ← setColumn(^paxis, ^p_jR')             // collecting contribution of each axis
11:     end for
12:     ^p_jR ← RotationToParent(j)                        // local rotation from j to parent p
13:     ^b_jR ← getRotation(^b_jAd)                        // original rotation from j to b
14:     ^b_jR' ← ^b_jR (^p_jR^T ^p_jR')                    // adjusted rotation
15:     ^b_jAd ← recompute(^b_jR', ^b_jAd)                 // update adjoint with new rotation
16:     index ← getIndex(j)
17:     ^bJ ← setMatrixBlock(index, ^b_jAd, ^bJ, trim)     // setting the matrix block
18: end while
```

3.1 Formal definition

In order to derive equations for the inverse kinematics solver we need differential quantities of the desired control handles with respect to the joints. That is, given a forward kinematics map $f : Q \to \mathbb{R}^n$ and a desired configuration $f_d \in \mathbb{R}^m$, we would like to solve the equation $f(\theta) = f_d$ for some $\theta \in Q$, where n and m vary based on the type of joints and the controlled features. This is a root finding problem, and it may have multiple solutions, a unique solution, or no solution at all, as discussed by Murray et al. (1994). Solutions to this mapping are obtained by iterative solvers that require differentiating the forward kinematic map with respect to the control quantities. This differentiation results in what we have studied as Jacobian.

3.2 Solvers

Since the Jacobian matrix is not square, we use a pseudo-inverse method to compute the solution

$$\dot{\theta} = J^{\dagger}\, \dot{p} \tag{33}$$

where $J^{\dagger}$ is the pseudo-inverse of J and is computed by

$$J^{\dagger} = J^T (J J^T)^{-1}. \tag{34}$$

Remember all terms in Equation 33 *must* be computed in a common frame (conventionally body or world frame). Equation 33 provides a least square solution with minimal norm that remains robust as long as singular configurations are avoided [Buss (2004)]. We use an iterative method to

Method	Performance (Robustness, Speed)	How	Complexity (flops)
Singularity Robust		$\mathbf{J}^+ = \mathbf{J}^T(\mathbf{J}\mathbf{J}^T + \lambda\mathbf{I})^{-1}$ $\dot{\theta} = \mathbf{J}^+ \dot{p}$	$\approx 2n^2$
SVD		$\mathbf{J} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^*$ $\mathbf{J}^+ = \mathbf{V}\boldsymbol{\Sigma}^+\mathbf{U}^*$ $\dot{\theta} = \mathbf{J}^+ \dot{p}$	$2mn^2 + 2n^3 \approx 2n^3$
QR Factorization		$\mathbf{J} = \mathbf{Q}\mathbf{R}$ $y = \mathbf{Q}^T \dot{p}$ $\mathbf{R}\dot{\theta} = y$	$2n^3 + 3n^2 \approx 2n^3$
Jacobian Transpose		$\dot{\theta} = \mathbf{J}^T \dot{p}$	n^2
Pseudo-Inverse (Matlab)		$\mathbf{J}^+ = \text{pinv}(\mathbf{J})$ $\dot{\theta} = \mathbf{J}^+ \dot{p}$	$16mn^2 + 12n^3 \approx 12n^3$

Figure 8: *Solvers*

solve the IK problem because our characters typically have many degrees of freedom (25 >) and the target configuration is defined by a few control quantities which makes it impossible to apply a closed-form IK solver.

To help deal with singular configurations, a singularity robust method (SR), also known as damped pseudoinverse, uses a regularization parameter to relax the solution by letting the norm of the solution have some error

$$J^* = J^T(JJ^T + \lambda\mathbf{I})^{-1} \tag{35}$$

where J^* is the SR inverse of J, matrix $\mathbf{I}$ is the identity, and λ is the damping parameter weighting between the error and the norm of the solution. For small values of λ we get smaller errors, but we might as well encounter larger solutions around singular points. Using the SR inverse along with proper tuning of damping λ helps us ease the singularity problem by allowing errors near singular points. Figure 8 summarizes the common methods to solve our linear system. As it is clear from the figure, an SR method makes a very suitable candidate solver both in terms of robustness and speed. Note that the inverse of the term $JJ^T + \lambda\mathbf{I}$ can be computed by any method of choice, but since it is a square matrix, we can use LU decomposition for fast computations.

3.3 Practical notes

- To compute rotational error for the angular part in $\dot{p}$ use $R_d R_e^T$, where R_d is the desired rotation and R_e^T is the end effector rotation.
- Joints can be relaxed from the solution and DOFs can be unconstrained and returned to the system by simply setting the corresponding columns in the Jacobian to zero.
- Joints can attain different *strength* through scaling the adjoint blocks in the Jacobian from a user-friendly gain slider.
- Regularization scalar λ is typically set to values < 1. Larger values might make the solution too *indifferent* to the control quantities.
- Multi-end effector IK setup is achieved by computing separate Jacobians for each individual end effector, stacking up the Jacobian matrices as well as the control vectors (in the same order), and solving all linear systems simultaneously since all Jacobians with a shared subset of joints have duplicate columns.
- For a specific Euler axis we can implement a joint limit by adding a rotational constraint to the control components, projecting out-of-range angles to clamping limits and using the constraint error in the IK solve. Likewise, setting the error to zero when within the allowed joint limit range relaxes the solver not to try to do any clamping. Some people might prefer a simpler clamping method through directly clamping the solution $\dot{\theta}$. This in general might lead to unstable behaviours because manipulating the solution anywhere outside the solver may potentially violate the linear mapping used in differentiating the system. In simple words, the best place for the constraints to fight is within the solver, not relying on a manual approach.

Bibliography

Buss, S. R. Introduction to inverse kinematics with Jacobian transpose, pseudoinverse and damped least squares methods, 2004. unpublished survey.

de Lasa, M. *Feature-Based Control for Physics-Based Character Animation*. PhD thesis, University of Toronto, October 2010.

Featherstone, R. *Rigid Body Dynamics Algorithms*. Springer-Verlag, 2008.

Murray, R. M., Li, Z., and Sastry, S. S. *A Mathematical Introduction to Robotic Manipulation*. CRC, March 1994. ISBN 0849379814.