Inverse Kinematics Solver API

Version 1.3

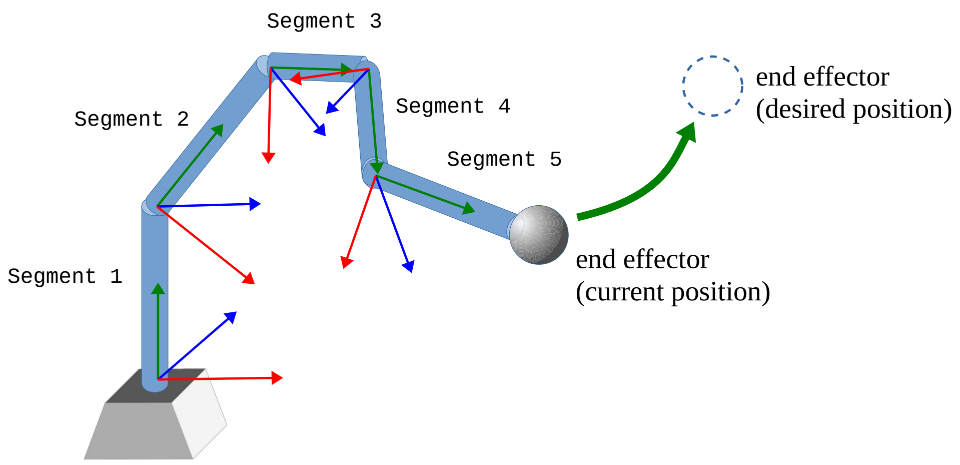
J. Radulovic

Dec 2019

Table of Contents

Introduction

The IK API solves or computes a set of joint angles so that the end of a given joint chain ends up at a desired position. The figure below illustrates the general idea.

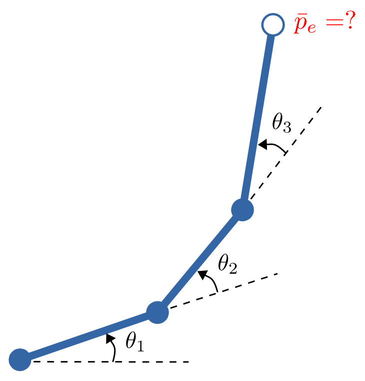


There are a few observations that should be made at this point. First, each segment of the chain is made up of two parts:

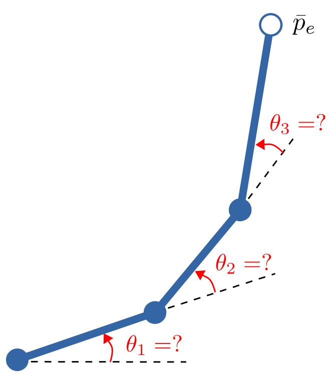
- joint angle(s) – depending on the type of joint and how its able to rotate, the joint may rotate about one or more axis.
- length of each segment (bone)

Each segment's orientation is defined with respect to the previous segment's orientation, resulting in a natural hierarchy such that if segment x is rotated, it will affect all other segments further down the hierarchy.

In order to simplify the explanation we will reduce our problem to a 2D plane such as the following:



Forward Kinematics



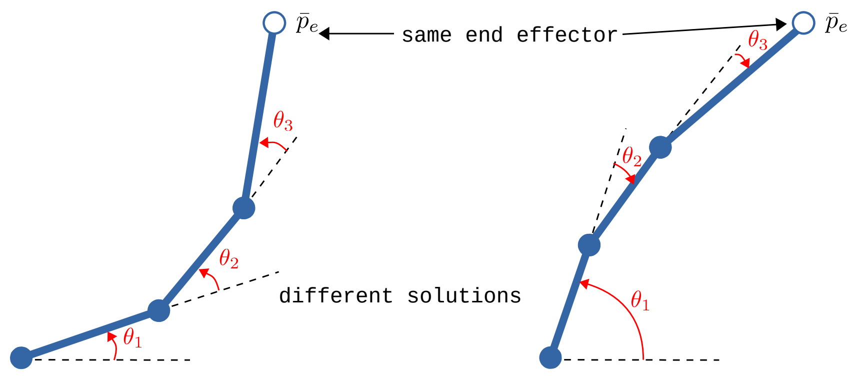
Inverse Kinematics

Notice that the position of the end of the last segment, $\bar{p}_e$, is a point of interest. It is usually called the end effector.

Using this description of the system of joints, we can then ask two different questions:

1. **Forward Kinematics:** given the angles of each segment (which indirectly specifies the positions as well), what is the position of the end effector?
2. **Inverse Kinematics:** given the position of the end effector, what are the joint angles that will give us this position?

It should be obvious that if we know all the joint angles, then there will be a unique point for the end effector position $\bar{p}_e$. However, the opposite is not necessarily true. That is, if the position of the end effector is known, there may be more than one solution of joint angles that will give the desired position as in the following case:

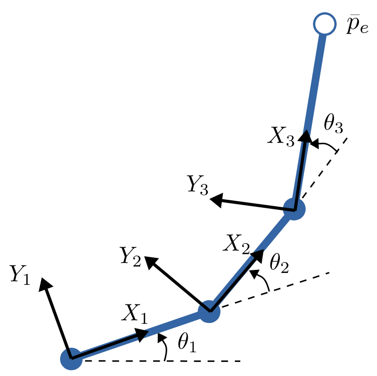


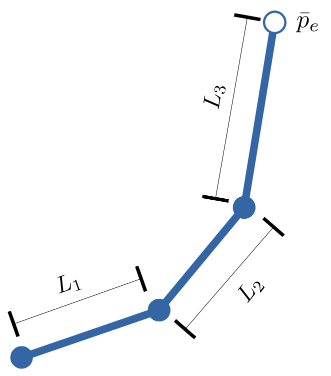
Example of an end effector with two different solutions.

This tells us that inverse kinematics is probably a more difficult problem to solve.

Defining the Chain of Segments

To make things manageable, we will define each segment in its own coordinate system. This will allow us to naturally implement the hierarchical nature of the entire system. As a consequence, each segment will be defined by a bone length and an orientation relative to it parent's position and orientation. The following diagram illustrates this idea:





The diagram on the left shows three 2D segments. Each bone has its own local coordinate system and the position of the bone that is connected to it is defined as a length in the direction of the X-axis of the parent bone. Similarly, the orientation of the child bone is specified as an angle relative to the parent's coordinate system. So, for example, the position of the 2nd bone is obtained by going along the X-axis of the parent coordinate system a distance of L_1 units.

The orientation of the second bone is similarly obtained by a rotation of θ_2 units with respect to the parent's coordinate system, in this case a rotation counter-clockwise from the X_1 axis.

Going from one coordinate system to the next is achieved by applying transformation matrices. Once again, each transformation matrix corresponding to a bone is defined relative to the previous bone. So if M_1 and M_2 are matrices corresponding to bones 1 and 2 respectively, then the position and orientation of bone 2 would be obtained by:

$$\bar{p}_2 = M_1\bar{p}_1$$

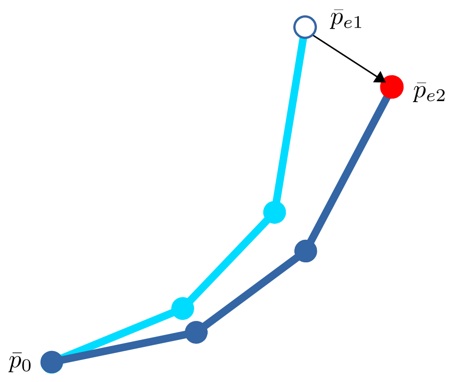
where $\bar{p}_1$ is a point defining the starting position of the chain. Note that $\bar{p}_2$ is the position of the start of the **second** bone defined in the coordinate system of the **first segment**. The end effector position is then given by:

$$\bar{p}_e = M_3M_2M_1\bar{p}_0$$

The transformation matrices themselves describe the translation and rotation(s) that are required to transform a point in one coordinate system to another. The rotation part can be described using Euler angles (one rotation about each of the X, Y and Z axis) or quaternions.

Forward Kinematics

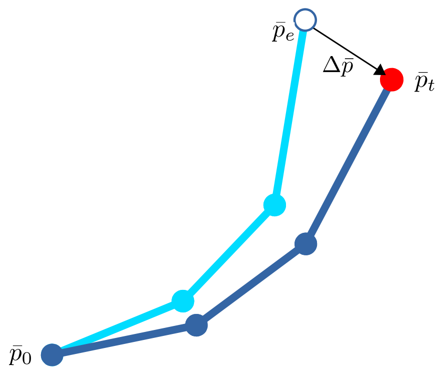
Once we have defined the system of bones we can then clearly define our goal. In the case of forward kinematics, we would like to know the position of the end effector after the angles have changed:



As discussed above, the new point's position/orientation can be determined by multiplying the appropriate transformation matrices, which are ultimately determined by the new angles:

$$\begin{aligned}\bar{p}_{e2} &= M_n \cdots M_3M_2M_1\bar{p}_0 \\ &= M\bar{p}_0 \qquad\qquad \text{where } M = M_n \cdots M_3M_2M_1\end{aligned}$$

Inverse Kinematics



Now we'll consider going in the opposite direction and try to determine the joint angles of the system that will produce a desired or target end effector position, $\bar{p}_t$, if we start at $\bar{p}_e$. We'll call the difference between these Δp, so that Δp = $\bar{p}_t$ - $\bar{p}_e$.

The solution to these types of systems is generally too difficult to solve analytically so we will resort to a computational algorithm that will incrementally change the joint angles so as to direct the current position and orientation of the end effector towards its desired final state. This computation is performed by taking the partial derivatives of each joint function with respect to its angles. For example, if we have 3 functions each describing the x, y, and z positions of the end effector $\bar{p}_e$ in terms of all the joint angles, we may have something like this for a 4-joint system:

$$x = f_1(\theta_1, \theta_2, \theta_3, \theta_4)$$
$$y = f_2(\theta_1, \theta_2, \theta_3, \theta_4)$$
$$z = f_3(\theta_1, \theta_2, \theta_3, \theta_4)$$

We can then define a function $F(x, y, z)$ in terms of $x, y, \text{ and } z$.

Jacobian

Assuming that F is relatively smooth, we can calculate the Jacobian of F, denoted by J, as a matrix of partial derivatives:

$$J = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} & \frac{\partial x}{\partial \theta_3} & \frac{\partial x}{\partial \theta_4} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} & \frac{\partial y}{\partial \theta_3} & \frac{\partial y}{\partial \theta_4} \\ \frac{\partial z}{\partial \theta_1} & \frac{\partial z}{\partial \theta_2} & \frac{\partial z}{\partial \theta_3} & \frac{\partial z}{\partial \theta_4} \end{bmatrix}$$

Now if we want to know how the x-value is affected by a small change in any of the angles, we could take the slope with respect to any of the angles $\dfrac{\partial x}{\partial \theta_i}$ and, if $\Delta\theta_i$ is small enough, we can approximate the new x position by:

$$x_{new} = x_{old} + \frac{\partial f_1}{\partial \theta_i} \cdot \Delta\theta_i$$
$$x_{new} - x_{old} = \Delta x_i = \frac{\partial f_1}{\partial \theta_i} \cdot \Delta\theta_i$$

If this is done for all $\Delta\theta_i$, then we could obtain the total Δx as:

$$\Delta x = \frac{\partial f_1}{\partial \theta_1} \cdot \Delta\theta_1 + \frac{\partial f_1}{\partial \theta_2} \cdot \Delta\theta_2 + \frac{\partial f_1}{\partial \theta_3} \cdot \Delta\theta_3 + \frac{\partial f_1}{\partial \theta_4} \cdot \Delta\theta_4$$

If we represent the set of joint angles as a vector, then the above equation can be seen as a dot product:

$$\Delta x = \begin{bmatrix} \frac{\partial f_1}{\partial \theta_1} & \frac{\partial f_1}{\partial \theta_2} & \frac{\partial f_1}{\partial \theta_3} & \frac{\partial f_1}{\partial \theta_4} \end{bmatrix} \cdot \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \\ \Delta\theta_3 \\ \Delta\theta_4 \end{bmatrix}$$

Incidentally the expression $\begin{bmatrix} \frac{\partial f_1}{\partial \theta_1} & \frac{\partial f_1}{\partial \theta_2} & \frac{\partial f_1}{\partial \theta_3} & \frac{\partial f_1}{\partial \theta_4} \end{bmatrix}$ is also known as the gradient of f_1 or ∇f_1. Using this we can write the above equation in vector form as:

$$\Delta x = \nabla f_1 \cdot \Delta\theta$$

Similar operations are done on the y, z components so that by using the Jacobian we can write:

$$\Delta p = J\Delta\theta$$

So finally we have an expression for the change in end effector position due to changes in the joint angles. But what we are really after is the inverse operation, ie. the changes in the joint angles due to a change in the end effector position:

$$\Delta\theta = J^{-1}\Delta p$$

Pseudoinverse of Jacobian

Taking the inverse of a non-square matrix is generally not possible but finding an approximation to it in the form of the pseudoinverse is. Here is a short derivation of the pseudoinverse:

$$\begin{aligned} \Delta p &= J\Delta\theta \\ J^T\Delta p &= J^T J\Delta\theta \\ (J^T J)^{-1} J^T \Delta p &= \Delta\theta \\ J^+ \Delta p &= \Delta\theta \end{aligned}$$

So the pseudoinverse is defined as:

$$J^+ = (J^T J)^{-1} J^T$$

The pseudoinverse[1] approach performs poorly when the joint chain stretches out and the axes of rotation align. A more stable and robust solution is to use the Damped Least Squares method. Instead of the pseudoinverse, we have a new update matrix J^*, of the form:

$$J^* = (J^T J + \lambda^2 I)^{-1} J^T$$

Both variants of the pseudoinverse are implemented in the code.

1 Driscoll, M. Numerical Stability of Iterative Solvers for Inverse Kinematics, Dec 15, 2014

Code Implementation

The Java code to implement inverse kinematics is described here for a programmer who wishes to use the code in their own programs.

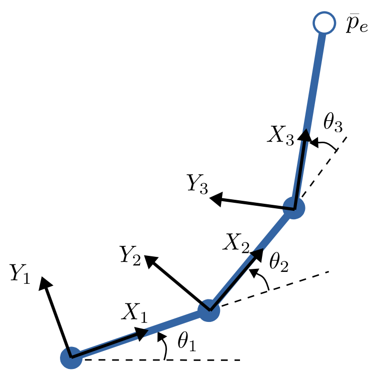
The IK Solver implementation has 2 main classes:

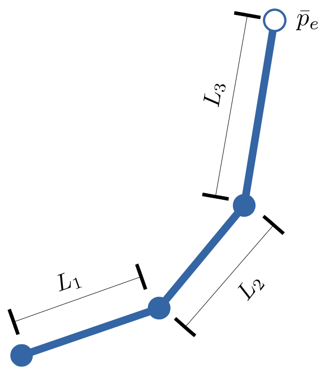
1. JointSystem – describes the entire system of joints and bones
2. Bone – describes a single bone (which mainly consists of a length and an orientation with respect to the previous bone)

There are also more general classes that are also used:

- Point3Df
- Vector3f
- Matrix – used for Jacobian and related functions
- Transform – used to define the hierarchical structure of the JointSystem
- Matrix4f – used for 3D graphics
- Mesh - used to store a 3D mesh of the bones for graphical visualization
- Camera – used to position the viewer in the 3D scene for graphical visualization

In order to create a new Joint system, an example is provided. Suppose that we wish to implement the following system:





Each of the bones needs to be defined first. In this example the code required would be:

```
Bone rootBone = new Bone(new Point3Df(0,0,0), L_1, theta_1, new Vector3f(0,1,0));
Bone b1 = new Bone(rootBone, L_2, theta_2, new Vector3f(0,1,0));
Bone b2 = new Bone(b1, L_3, theta_3, new Vector3f(0,1,0));
```

Notice that the root bone has a slightly different constructor specifying it's world position. The other bones are attached to the root bone either directly or indirectly through other bones. Their constructors need a reference to the parent bone as the first argument. The next argument is the length of the bone. And finally, the angles are specified in two parts: the angle with respect to the previous bone's coordinate system and its axis of rotation. These are the last two arguments of the constructor. In the example above all of the joint angles are defined about the y-axis.

In most real cases, joint angles can only rotate within a certain range. In order to constrain joint angles, limits can be placed on their minimum and maximum angles by using the following method:

```
rootBone.setAngleConstraint(min, max);//default min = 0, max = 180
```

Once all of the bones are defined, the JointSystem object can be created by passing it the rootBone:

```
JointSystem RobotArm = new JointSystem(rootBone);
```

That's it for the setup of the system of joints!

To get the position of the end effector, we use forward kinematics by calling the method:

```
Point3Df p_e = rootBone.getEndEffector();
```

To set the desired position of the end effector to a certain point $\bar{p}_d$, we can use the following method:

```
Point3Df p_d = new Point3Df(3,4,5);

RobotArm.setDesiredEndEffectorPos(p_d);
```

And, finally to move the current end effector towards the desired position:

```
RobotArm.moveToTarget();
```

The last command will compute the angles necessary to move the end effector closer to the desired location. You can get the position or angle of any joint at any time by the following methods:

```
Point3Df p = RobotArm.getJointPos(int i);    // get the position of the ith
                                             // joint in world coords
float a = RobotArm.getJointAngle(int i);     // get the angle (in degrees) of
                                             // the ith joint w.r.t. the
                                             // previous bone
```

Since the change in the joint positions Δp is small, the moveTarget() method will likely be called many times until the target is reached. In the code provided this method is placed inside the main loop().

The Graphical Interface

In order to visual the movement of the JointSystem, a simple graphics UI is provided. Each bone can have a 3D mesh assigned to it. In order to use this feature the mesh needs to be created by reading in a .OBJ file:

```
Mesh boneMesh = new Mesh("Bone.obj");
```

Then the mesh can be assigned to a bone:

```
rootBone.setMesh(boneMesh);
```

Currently the code is optimized to load the provided Bone.obj model, but future versions will have the ability to load any 3D model and use it as a joint without having to alter to the code.

The JointSystem can now be drawn using the draw() method:

```
RobotArm.draw(0.01, camera);
```

The first argument is the change in time since the last call, but is currently ignored in this version. The second argument defines the camera, which determines the vantage point from which to render the scene.

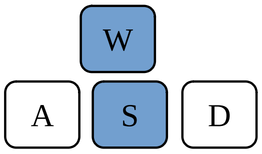
For the above argument to work a Camera object needs to be created. There are 3 different constructors:

```
public Camera(float left, float right, float bottom, float top, float near, float far)
public Camera(float fov, float aspect, float zNear, float zFar)
public Camera(Matrix4f projection)
```

The first two constructors will calculate the projection matrix from various properties of the view volume. The last constructor allows you to calculate your own projection matrix and pass it in as an argument.

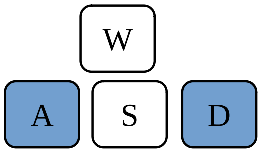
Interacting With the Scene

The user can move around the scene by using a combination of keyboard and/or mouse input. The input(s) are defined as follows:



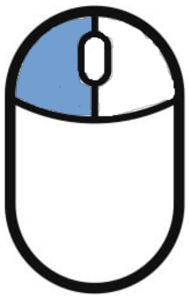
The W-key moves the camera forward.

The S-key moves the camera backwards



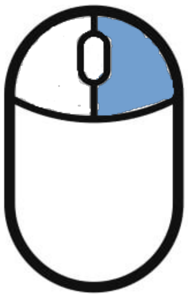
The A-key pans the camera to the left.

The D-key pans the camera to the right.



Moving the mouse horizontally while pressing the left-mouse button will rotate the camera about the y-axis.

Moving the mouse vertically while pressing the left-mouse button will rotate the camera up or down.



Right-clicking on the mouse will set the end effector target position to be highlighted in red. Currently, the end-effector position can only be set to a point somewhere on the x-z plane.

The spacebar toggles the simulation on/off.