Numerical Methods for Inverse Kinematics

Niels Joubert, UC Berkeley, CS184

2008-11-25

Inverse Kinematics is used to pose models by specifying endpoints of segments rather than individual joint angles. We will go through the steps of deriving a simple inverse kinematics problem. We are given a model in a starting state and a goal point to which we want to move the end of the arm, and we will solve for the new joint angles. In doing this we will use two areas of knowledge - trigonometric relationships to describe the model we're posing, and linearization to find successive approximations of our answer.

1 Problem Description

We want to calculate the change in joint angles needed to bring the endpoint of our joint to the given position. **Forward kinematics** is concerned with the endpoint's position as you change the joint angles. **Inverse Kinematics** is concerned with the joint angles needed to produce a specific endpoint's position. A basic understanding of trigonometry should enable us to write down the forward trigonometric equations[1].

There is a collection of possible joint angles, all of which bring the endpoint's position to the wanted goal. An analytical solution becomes extremely complicated. Thus, we will render this as a minimization problem, so that we can apply root-finding techniques to it.

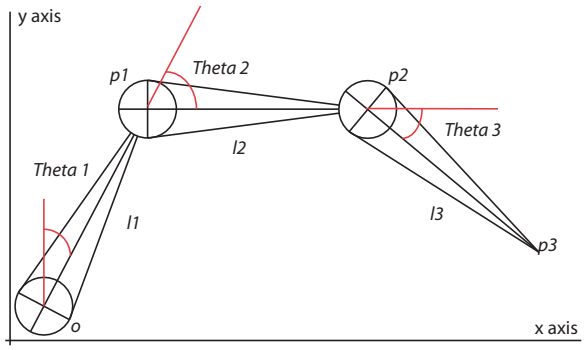
1.1 Terminology

- DOF - Degrees of Freedom - a parameter we can control. In 3-space: 3 translations, 3 rotations.
- Joint - A connection between bodies with some DOF that modifies and propagates its transform to its outbound body.
- Inbound body - A body determining the transform of a given joint.
- Outbound body - A body whose orientation is determined by the given joint.
- Root Body - A body with a fixed position set by the global transform.
- Root Joint - the joint on the root body (thus, fixed position) with some DOF (generally 1 or 2 rotations).
- Linearization - Finding a linear approximation to a nonlinear function
- Root-finding - Finding the solution to $f(x) = 0$.

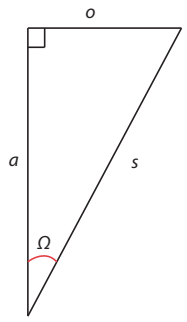
[1] Section 2

2 Forward Kinematics

Here we'll derive the forward kinematics equation for the arm of three joints in the following picture. We'll describe the endpoint in terms of the joint angles and lengths. I'll start by reminding you of the relationships



(a) 3-jointed arm



(b) Right angle triangle

between angles and sides found in the right angle triangle of Figure 1(b):

$$s * \sin(\Omega) = o \tag{1}$$

$$s * \cos(\Omega) = a \tag{2}$$

Write down the equations for p_{1x} and p_{1y} (the x and y coordinate of the endpoint of the first joint) in terms of θ_1 and l_1.

$p_{1x} =$

$p_{1y} =$

Write down the equation for p_{2x} and p_{2y} in terms of p_1, θ_2 and l_2. **Think about the angles very carefully!**

$p_{2x} =$

$p_{2y} =$

Write down the equation for p_{3x} and p_{3y} in terms of p_2, θ_3 and l_3. Then expand it out to find p_{3x} and p_{3y} in terms of the three joint angles (θ_1, θ_2, θ_3) and body lengths (l_1, l_2, l_3).

$p_{3x} =$

$p_{3y} =$

We can now describe the position of the endpoint of our chain in terms of the joint angles. **Notice that we defined all our rotations as clockwise from the y-axis.** This is arbitrary, but you need to be consistent.

3 Inverse Kinematics

We want to find the set of joint angles that produce a specific end position. Assume you are given a configuration for your skeleton, and you want to move it to a new position. Thus, we want to compute the **change in joint angles** needed to produce the **change in endpoint positon.** From lecture (and the trigonometry on the previous page) you know that the analytic solution for p_{3x} and p_{3y} in terms of the three joint angles (θ_1, θ_2, θ_3) is hard and messy. We will circumvent this by finding a linear approximation to the change in position bought about by the change in joint angle.

3.1 Linearization and Jacobians

Consider the single joint in the following figure:

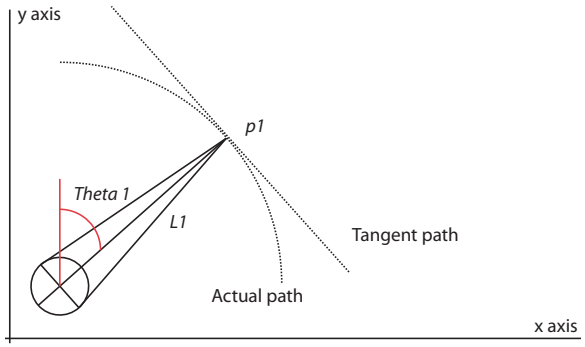


Figure 1: Single 1 DOF joint.

We plot the path of p_1 as it varies over θ_1, and we plot the tangent of the path at the current point of p_1. Notice how the tangent is a close approximation to the actual path in the region around p_1. **Linearization** is the process of approximating a nonlinear function by its tangent around a point.

In section 2 we described the actual path. We now need to find the tangent path - a straight line varying with parameter θ. Since a tangent is just a derivative of a function at some point, calculate the derivative of p_{1x} and p_{1y} (from section 2) with respect to θ_1 (remember your trigonometric derivatives...):

$$\frac{\partial p_{1x}}{\partial \theta_1} =$$

$$\frac{\partial p_{1y}}{\partial \theta_1} =$$

Since the tangent is a line, and an equation for a straight line from some point a is $f(x) = m*(x-a_x)+a_y$ we can now write the function $p_1(x)$ that determines the position of p_1. Assume the joint is currently at an angle of θ_1 at point (p_x, p_y) and we want to find $pl_1(\theta)$, the linear approximation of $p_1(\theta)$ around θ_1:

$$pl_{1x}(\theta) =$$

$$pl_{1y}(\theta) =$$

Consider the two functions you just found. They describe the position of the joint as a linear function of the joint angles. In other words, we've linearized the forward kinematics equation. What we did was write out the **Taylor expansion** for the function p_{1x} and p_{1y} around p_1's current position.

3.1.1 The Jacobian matrix

So far we've been working with the endpoint's position p_1 as two coordinates, p_{1x} and p_{1y}. We can collapse these two coordinates into p_1 and define $p_1(\theta)$ as a vector function - a function returning a vector. We just found a partial derivative for p_{1x} and p_{1y} - the change in each coordinate with respect to each joint angle. We will now introduce the Jacobian:

Definition 1. *The Jacobian matrix is a matrix of all the first order partial derivatives of a vector function.*

This means that the Jacobian matrix describe how each coordinate changes with respect to each joint angle in our system. You can think of it as a big function with two arguments - the joint angle and the coordinate - and it describes the first order linear approximation between these two arguments. Mathematically, we write this as:

$$\mathbf{J}_{ij} = \frac{\partial p_i}{\partial \theta_j} \qquad p_i = i\text{th coordinate of endpoint } p; \quad \theta_j = j\text{th joint angle.} \tag{3}$$

The Jacobian matrix is *extremely* useful, since it describes the first order linear behavior of a system. In spring-mass simulations it is used for integration purposes. In IK it is used to successively approximate the joint angles needed for a given endpoint. Let's apply this to our original three-joint arm:

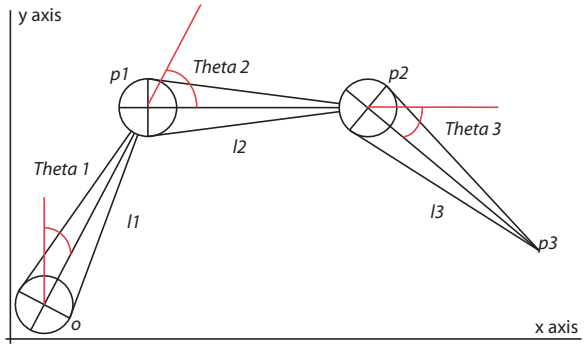


Figure 2: 3-jointed arm

Then the Jacobian will be the following 2 by 3 matrix (2 coordinates, 3 joint angles). You should notice that you computed some of these terms in section 2 and section 3.1 and you know how to compute them all.

$$\mathbf{J}(\theta) = \begin{bmatrix} \frac{\partial p_{3x}}{\partial \theta_1} & \frac{\partial p_{3x}}{\partial \theta_2} & \frac{\partial p_{3x}}{\partial \theta_3} \\ \frac{\partial p_{3y}}{\partial \theta_1} & \frac{\partial p_{3y}}{\partial \theta_2} & \frac{\partial p_{3y}}{\partial \theta_3} \end{bmatrix} = \begin{bmatrix} \begin{pmatrix} l_1\cos(\theta_1) + l_2\cos(\theta_1+\theta_2)+ \\ l_3\cos(\theta_1+\theta_2+\theta_3) \end{pmatrix} & \begin{pmatrix} l_2\cos(\theta_1+\theta_2)+ \\ l_3\cos(\theta_1+\theta_2+\theta_3) \end{pmatrix} & l_3\cos(\theta_1+\theta_2+\theta_3) \\ \begin{pmatrix} -l_1\sin(\theta_1) - l_2\sin(\theta_1+\theta_2) \\ -l_3\sin(\theta_1+\theta_2+\theta_3) \end{pmatrix} & \begin{pmatrix} -l_2\sin(\theta_1+\theta_2) \\ -l_3\sin(\theta_1+\theta_2+\theta_3) \end{pmatrix} & -l_3\sin(\theta_1+\theta_2+\theta_3) \end{bmatrix}$$

We can now rewrite out original taylor expansion of a coordinate around a point determined by the joint angles θ_{current} (The last two equations in setion 3.1) in terms of the vector function $p_1(\theta)$ and the Jacobian:

$$p(\theta) \approx p_{\text{linear}}(\theta) = p_(\theta_{\text{current}}) + \mathbf{J}(\theta_{\text{current}})(\theta - \theta_{\text{current}}) \tag{4}$$

This is a linear vector function that approximates our forward kinematics equation. And since it is linear, we can employ all our linear algebra techniques to find its inverse. Congratulations, you linearized the kinematics problem.

3.2 Solving for Joint Angles - Pseudoinverses and Root Finding

The original problem we set out to solve is described by figure 3.2:

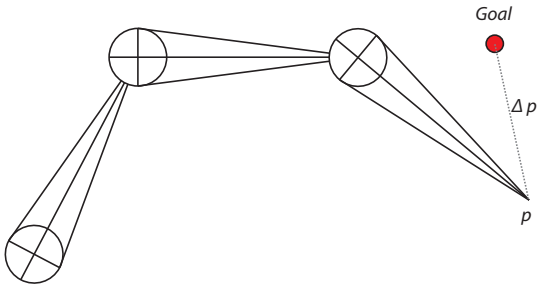


Figure 3: 3-jointed arm with goal position

What must the change in joint angles be to achieve the change in endpoint position Δp? Look again at equation 4. Let's define Δp and $\Delta\theta$ in terms of both the quantities in figure 3.2 and the quantities of equation 4:

$$\Delta\theta = \theta_{\text{goal}} - \theta_{\text{current}} \quad (5)$$

$$\Delta p = \text{Goal} - p_{\text{current}} \quad (6)$$

$$\approx p_{\text{linear}}(\theta_{\text{goal}}) - p_{\text{current}} \quad (7)$$

Notice that I collapsed all the joint angles into one θ variable. We can do this by letting $\theta = (\theta_1, \theta_2)$. Thus, θ is a vector. Notice that in lecture professor O'Brien followed a slightly different approach by defining θ^* as a linear combination of θ_1 and θ_2.

Now rewrite equation 4 in terms of equations 5 and 7, and manipulate it so that you find an equation for $\Delta\theta$:

We just found an expression for the change in joint angles needed to move the endpoint from its current position to the goal position, in terms of quantities we can compute. **But this still doesn't completely work!** $\mathbf{J}(\theta_{\text{current}})$ in our example is not even a square matrix, thus there is no way to invert it. This is the same problem we originally had - there is a whole set of solutions that satisfy our goal position, thus the matrix is not invertible. Luckily we have ways of dealing with that for matrices.

3.3 How to solve this set of equations

We're now entering the realm of higher level linear algebra, thus I will unfortunately not go through the derivations of how things work, but rather only propose a way to find the solution to $\Delta\theta$.

You should have found the following equation on the previous page:

$$\Delta\theta = \mathbf{J}(\theta_{\text{current}})^{-1} * \Delta p$$

But, as we noted, $\mathbf{J}(\theta_{\text{current}})^{-1}$ is almost never invertible, thus we have to compute a *pseudoinverse*. The pseudoinverse of a matrix has most but not necessarily all of the properties of an inverse, and it not necessarily unique. Since we are already taking a linear approximation of our $p(\theta)$ function, we can justify taking another approximation, since we have to account for these assumptions in our final solution in either case. (We will do this by successive iterations of this algorithm).

3.3.1 Calculating a Pseudoinverse using Singular Value Decomposition

The Singular Value Decompositon is a technique that allows us to decompose *any* matrix into three matrices, two square orthogonal matrices and a possibly non-square diagonal matrix:

$$\mathbf{T} = \mathbf{USV}^{\mathrm{T}} \tag{8}$$

There exists algorithms to compute the SVD decomposition of a matrix. Once you've found the SVD decomposition, the pseudoinverse of T is defined as:

$$\mathbf{T}^{+} = \mathbf{VS}^{+}\mathbf{U}^{\mathrm{T}} \tag{9}$$

where S^{+} is S with all its nonzero diagonal entries replaced with its corresponding reciprocals. Why does this work? Replacing the diagonals with its reciprocals is the same as inverting the matrix, but in the case that there are zeros on the diagonals, we can't take the reciprocals, so we calculate an approximate inverse by taking the reciprocals only of the nonzero terms. Since $\mathbf{U}$ and $\mathbf{V}$ are orthogonal, their transpose is their inverse. Thus the pseudo-inverse of $\mathbf{T}$ consists of the inverse of $\mathbf{U}$, $\mathbf{V}$ and an approximate inverse of $\mathbf{S}$.

Thus we can compute a pseudoinverse of the jacobian, and we can solve for the change in joint angles needed to achieve the change in position we want. Except this change is still only a linear approximation. So, we'll turn it into a root-finding problem, which we'll explain in the last section of this worksheet.

3.3.2 Root-finding

Root-finding is the numerical process by which we solve for x in the equation $f(x) = 0$. With some simple algebra, we can rearrange our terms so that our equations are in this form. Rewrite the equation you derived on page 5 in the form $f(x) = 0$:

You can now apply any of the many root-finding techniques to it, but you will realize that Inverse Kinematics is a *really nice* problem in this case, and a simple iterative approach can work. I will justify this statement as follows: If we compute an approximate change in joint angles needed to achieve this change in position, we can simply *move* our system of joints and bodies to this new location and *repeat the procedure we just went through*. This is known as **Newton's Method**, and is a well-studied algorithm for finding roots to something of the form $f(x) = 0$. We implement it with successive iterations of finding the change in joint angles using the pseudoinverse of the jacobian. There is one more caveat - Newton's method does not always converge.

3.3.3 Convergence

Newton's method does not always converge. Thus, calculating the approximate change in angle we need and simply applying that interactively does not necessarily bring us closer to the correct solution. We can amend this situation by implementing a binary search on the change in angle:

1. Calculate the new position the current change in θ would cause.
2. Calculate the distance from the goal.
3. If the distance decreased, take the step.
4. If the distance did not decrease, set the change in θ to be half the current change, and try again.

3.3.4 Minimization

We can also cast everything we've done as a minimization problem. We're attempting to minimize the error between the current position and the goal position ($\Delta\theta = \theta_{\text{goal}} - \theta_{\text{current}}$), and we're achieving that using a linear approximation of the function and applying root-finding techniques to it. This is a common method to solve these types of problems, and appears in numerical simulations in physics, math and engineering. We implicitly did exactly this, thus I mention it for completeness.

4 Summary and Overview

We've gone through a long derivation of a general numerical approach to solving the inverse kinematics problem. To summarize our approach, the following diagram demonstrates a simple IK solver:

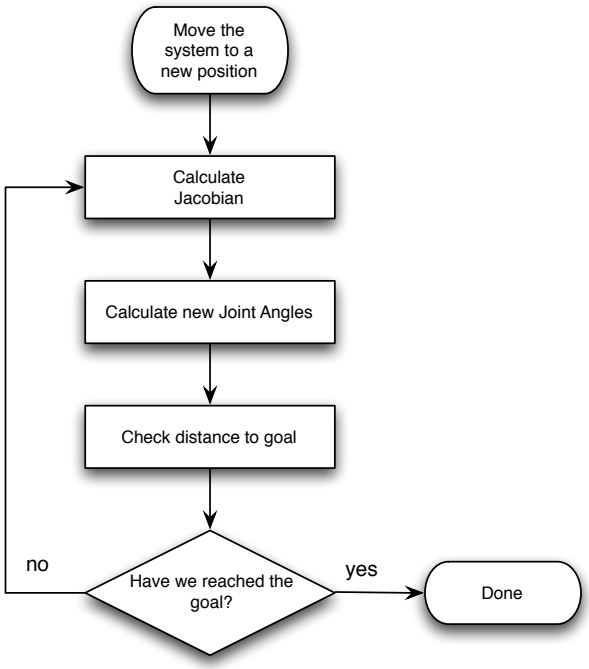


Figure 4: A simple IK solver.

Finally, the Jacobian is a very powerful tool. To extend this system to allow for multiple goals with multiple arms as one IK system, you simply expand your Jacobian to include all the joints. If you want to

constrain some joints not to move at all, you can simply reduce your jacobian not to include them. More advance techniques to implement constraints make use of Lagrangian Multipliers. Also, it is worth noting that our solution is a *local solver*. We make an approximation of the function that holds in the area around the current joint angles.

4.1 Real World IK Solvers

Real World IK Solvers are not necessarily built in the same manner as this tutorial explains, but this is one of the accepted approaches. Others are particle-based IK and Cyclic Coordinate Descent. We did not cover one important facet though - real world IK solvers does includes a strong framework for setting constraints. You often want to use an IK solver to pose your model, but you want specific joints to have specific orientations - the modeler wants control over which list of joint angles is picked by the solver.

For more information: http://billbaxter.com/courses/290/html/