Inverse Kinematics Techniques

Stephen Tu
University of California, Berkeley
stephen_tu@berkeley.edu

ABSTRACT

Inverse kinematics (IK) is an important part of animation in computer graphics. Being able to solve IK problems efficient is necessary for realistic computer animation rendering. In this paper we look at the common techniques used to implement inverse kinematics solvers.

1. INTRODUCTION

In the last decade or so, inverse kinematics has started to play a role in the computer graphics community, especially in animation. The problem statement is simple: *given a set of end point positions for a robot (effectors), find a configuration of angles between joints such that the end points of the robot links match the given set of positions.* The solution however, is not as straightforward, since in most cases there are either infinite many solutions, or no solutions; rarely is there ever exactly one solution. This makes the problem especially interesting in animation, since when there are many solutions, it is desirable to pick solutions which are consistent, in some sense. That is, the robot should not appear to oscillate from frame to frame, for example.

Fortunately, inverse kinematics is a well studied problem, and there are many common techniques for dealing with it. In this paper, I outline a few of the common ones.

2. PROBLEM STATEMENT

Before we proceed, we need to get formal about the problem statement. Let $\mathbf{x}$ represent the state of the robot, and $\mathbf{P}(\mathbf{x})$ be a vector of effector positions given the state $\mathbf{x}$. The inverse kinematics problem is to find $\mathbf{x}$ such that $\mathbf{r} = \mathbf{P}(\mathbf{x})$, where $\mathbf{r}$ is the desired positions of the effectors.

Now what exactly is this state vector $\mathbf{x}$? Well it is simply the parameterization of your robot. So if your robot has only rotational joints, then you can think of your state as just a bunch of angles describing how much each joint is rotated.

3. JACOBIAN MATRIX

Before we progress to methods for solutions, we must first become familiar with the Jacobian matrix. Now, there is nothing special about the Jacobian. It is simply, by definition, a matrix of partial derivatives. In inverse kinematics, we are often interested in the Jacobian matrix $\mathbf{J}$ of $\mathbf{P}(\mathbf{x})$. Supposing we have k effectors and n states, then $\mathbf{J}$ is simply given as:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial P_1}{\partial x_1} & \cdots & \frac{\partial P_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial P_k}{\partial x_1} & \cdots & \frac{\partial P_k}{\partial x_n} \end{bmatrix} \tag{1}$$

Now that is easy to write down mathematically, but how do we actually compute this matrix? It might be feasible if the robot you have is very small (one or two joints), or if you know the robot ahead of time, but we will need a method for doing this in general.

Well it turns out that we can compute the Jacobian just by knowing the types of joints we are dealing with. This should actually be fairly obvious if you really understand what the partial derivatives in the Jacobian mean. For some i, j, we can think of $\frac{\partial P_i}{\partial x_j}$ as the amount of change that a unit change of the j-th state causes to the i-th effector. Armed with this knowledge, we can now compute the Jacobian for our joints fairly easily.

3.1 Rotational Joint

Here we focus on a joint which rotates about an axis $\hat{v}$. Suppose we are computing $\frac{\partial P_i}{\partial x_j}$. Also suppose that the current position of this joint we are interested in is $\mathbf{z}$. Now there are two cases: either movement of joint x_j will cause a change in effector P_i, or it won't. If it does not, then our partial derivative is simply 0. If it does, then we'll need to do some computation here. For simplicity, I am going to assume that all our robots form a tree structure, that is each joint has only one parent (except the root). Under this assumption, the only way a joint can affect an effector is if the effector is a descendant of the joint in the tree. If so, then our partial derivative is given as:

$$\frac{\partial P_i}{\partial x_j} = \hat{v} \times (P_i(\mathbf{x}) - \mathbf{z}) \tag{2}$$

If this is not obvious, then draw it out. It should become clear why this is the case.

3.2 Translation Joint

A translation joint is even easier than a rotational joint. Once again, assume we are computing $\frac{\partial P_i}{\partial x_j}$. Now, if joint x_j affects P_i, then using the same notation from above, we get:

$$\frac{\partial P_i}{\partial x_j} = \frac{P_i(\mathbf{x}) - \mathbf{z}}{||P_i(\mathbf{x}) - \mathbf{z}||} \tag{3}$$

Note that $||\cdot||$ is the $L2$ norm of a vector. Quite simply, all this is saying is that a translation joint can only move an effector in the same direction it translates.

4. METHOD OF PSEUDOINVERSE

Now that we are familiar with and know how to compute the Jacobian, we can discuss the first method of solving the inverse kinematics problem. To understand this method, we simply have to recall what a Taylor series expansion looks like. Recall that taking the Taylor expansion about $\mathbf{P}$ at point $\mathbf{x_0}$ gives:

$$\mathbf{P}(\mathbf{x}) \approx \mathbf{P}(\mathbf{x_0}) + (\mathbf{x} - \mathbf{x_0})^T \nabla \mathbf{P}(\mathbf{x}) + ... \tag{4}$$

Remember that the Taylor series is an infinite series; by disregarding the higher order terms, we *linearize* the equation. Now if you look hard enough, we can rewrite this equation as (dropping the higher order terms for good):

$$\mathbf{P}(\mathbf{x}) \approx \mathbf{P}(\mathbf{x_0}) + \mathbf{J}(\mathbf{x} - \mathbf{x_0}) \tag{5}$$

Now this is beginning to look useful. Let us rewrite this equation one more time in terms of differences (I am dropping the approximate equals sign now):

$$\Delta\mathbf{P} = \mathbf{J}\Delta\mathbf{x} \tag{6}$$

$$\Delta\mathbf{x} = \mathbf{J}^{-1}\Delta\mathbf{P} \tag{7}$$

Now we are almost there. This says that if we want to change the state of our robot by $\Delta\mathbf{P}$, then all we have to do is solve Equation 7 for $\Delta\mathbf{x}$, plug it back into our robot and call it a day. Unfortunately life is never that easy, and thus we cannot exactly do that. The big problem here is that $\mathbf{J}$ is in no way guaranteed to be invertible, and thus we cannot plug it into our favorite $Ax = b$ solver.

Here is where the name pseudoinverse comes from, because what we do now is instead of taking the inverse which we cannot, we take the pseudoinverse which we can! The pseudoinverse is guaranteed to exist for any matrix, which includes $\mathbf{J}$. I will not go into the details of solving for the pseudoinverse, but chances are the method is either already in your favorite linear algebra library (`pinv` for Matlab folks) or you can compute it from the singular value decomposition of $\mathbf{J}$. The point is, here is the method of pseudoinverse:

$$\Delta\mathbf{x} = \mathbf{J}^{+}\Delta\mathbf{P} \tag{8}$$

Where $\mathbf{J}^{+}$ is the pseudoinverse of $\mathbf{J}$.

Now there is yet another caveat. Usually you do not just compute Equation 8 once. Instead, you iteratively compute Equation 8, repeatedly applying the updates (possibly weighted by some factor) until $\Delta\mathbf{x}$ stops changing by some threshold (which means you are close to the solution). The reason you do this is due to the linearization, which means your solutions are only accurate in a local region. Therefore, you want to take small steps so that you do not overshoot the answer. There are other things you can try, some of which are suggested in [3], which by the way is a great reference.

5. METHOD OF DAMPED LEAST SQUARES

To understand the method of damped least squares we must first understand the method of least squares. The method of least squares starts by using the same linearization as described above, repeated below for clarity:

$$\mathbf{P}(\mathbf{x} + \delta_x) = \mathbf{P}(\mathbf{x}) + \mathbf{J}\delta_x \tag{9}$$

The goal of the least squares method, however, is to find the δ_x such that the sum of the residuals squared is a minimum, that is:

$$\underset{\delta_x}{\operatorname{argmin}} ||\mathbf{r} - \mathbf{P}(\mathbf{x} + \delta_x)||^2 \tag{10}$$

Recall that $\mathbf{r}$ is the desired effector positions, We can rewrite that as:

$$||\mathbf{r} - \mathbf{P}(\mathbf{x} + \delta_x)||^2 = ||\mathbf{r} - \mathbf{P}(\mathbf{x}) - \mathbf{J}\delta_x||^2 \tag{11}$$

Now to minimize this equation, we just take the gradient and set it equal to 0. This is a bit of algebra involved, but trust me that the result comes out to:

$$(\mathbf{J}^T\mathbf{J})\delta_x = \mathbf{J}^T(\mathbf{r} - \mathbf{P}(\mathbf{x})) \tag{12}$$

Another way to see this is to remember from linear algebra that δ_x is minimized exactly when $\mathbf{r} - \mathbf{P}(\mathbf{x}) - \mathbf{J}\delta_x$ is orthogonal to the column space of $\mathbf{J}$, which gives rise to:

$$\mathbf{J}^T(\mathbf{r} - \mathbf{P}(\mathbf{x}) - \mathbf{J}\delta_x) = 0 \tag{13}$$

This equation can easily be manipulated into Equation 12.

Equation 12 is the least squares method. It is not obvious, but this equation is guaranteed to produce a set of linear equations which can always be solved for [1]. In other words, $\mathbf{J}^T\mathbf{J}$ is invertible, which gives rise to the following update:

$$\delta_x = (\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T(\mathbf{r} - \mathbf{P}(\mathbf{x})) \tag{14}$$

Equation 14 is used in the exact same manner as Equation 8 in terms of getting a solution to the inverse kinematics problem.

The damped least squares method is very similar. It simply rewrites Equation 12 to be:

$$(\mathbf{J}^T\mathbf{J} - \gamma\mathbf{I})\delta_x = \mathbf{J}^T(\mathbf{r} - \mathbf{P}(\mathbf{x})) \tag{15}$$

Where γ is a constant which can either be fixed or fine tuned per iteration[1]. This gives rise to the damped least squares method update:

$$\delta_x = (\mathbf{J}^T\mathbf{J} - \gamma\mathbf{I})^{-1}\mathbf{J}^T(\mathbf{r} - \mathbf{P}(\mathbf{x})) \tag{16}$$

I did not show this, but $\mathbf{J}^T\mathbf{J} - \gamma\mathbf{I}$ is indeed invertible.

6. METHOD OF CONSTRAINT FORCE AND VIRTUAL WORK

The previous two methods have described the unconstrained inverse kinematics problem. In these next few sections I want to focus on methods for dealing with constraints, specifically constraints which are functions of the state variables.

[1]In the literature, γ is often written as λ. I deliberately avoided this, however, because I did not want to confuse these λ constants with Lagrange multipliers (which will be covered later on), which are also conventionally written as λ.

We will first look at constraints of the form $\mathbf{C}(\mathbf{x}) = 0$, and then generalize to constraints of the form $\mathbf{C}(\mathbf{x}) \geq 0$.

In this section I outline the method proposed by [4] for dealing with equality constraints. Note that while the previous methods I discussed are very popular in the literature, this is a bit of a unique method which I have only seen Welman talk about in the context of inverse kinematics.

We start with our constraints $\mathbf{C}(\mathbf{x}) = 0$. This approach assumes that the system starts with the constraints already satisfied. It focuses on making sure that the constraint do not become unsatisfied by applying *constraint forces* to counteract any *applied forces* which make the constraints unsatisfied. When I say *force*, I am really talking about δ_x, because Welman uses a heuristic that $f \propto v$ instead of the usual Newtonian law that $f \propto a$ in order to simplify the equations.

We start by enforcing the constraints remain satisfied. This means that:

$$\mathbf{J}_c \delta_x = 0 \tag{17}$$

Here, $\mathbf{J}_c$ is the Jacobian of the constraint vector $\mathbf{C}(\mathbf{x})$. Note that we are now talking about δ_x as:

$$\delta_x = \delta_{x,a} + \delta_{x,c} \tag{18}$$

Where $\delta_{x,a}$ is the applied forces (what we have been calling just δ_x all along) and $\delta_{x,c}$ is the constrained forces. Plugging δ_x into Equation 17 yields:

$$\mathbf{J}_c \delta_{x,a} = -\mathbf{J}_c \delta_{x,c} \tag{19}$$

Now we assume that $\delta_{x,c} = \beta \mathbf{J}_c$, where $\beta = [\beta_1, ..., \beta_q]$ and q is the number of constraints in the system. This assumption comes from the *principle of virtual work*, which says that the constraint force must be in the same direction as the direction which the system itself cannot move. We now can write:

$$\mathbf{J}_c \delta_{x,a} = -\mathbf{J}_c \beta \mathbf{J}_c = -\mathbf{J}_c \mathbf{J}_c^T \beta \tag{20}$$

We can now solve the linear system of equations for β:

$$\beta = -(\mathbf{J}_c \mathbf{J}_c^T)^{-1} \mathbf{J}_c \delta_{x,a} \tag{21}$$

Note that we assume $\delta_{x,a}$ is a known, since we can use any of the previous methods discussed to recover it. Once we solve for β, we use it to recover $\delta_{x,c}$. Finally, we apply the update to the system given in Equation 18.

Note that many times, $\mathbf{J}_c \mathbf{J}_c^T$ contains very small numbers numerically, so it might be difficult to calculate the inverse. In these cases, you can take the singular value decomposition of $\mathbf{J}_c \mathbf{J}_c^T$ instead and use it to compute the inverse.

7. METHOD OF NON-LINEAR OPTIMIZATION

Another way to solve inverse kinematics problems is to simply treat it as a non-linear optimization problem. This is the most flexible alternative which also allows us to handle inequality constraints. However, this is also the most difficult method to implement and getting good results requires a pretty thorough understanding of what is actually going on.

7.1 Formulation as an Optimization Problem

We can reformulate the problem in Section 2 as follows. The inverse kinematics problem is to find the $\mathbf{x}$ such that the energy of the system is minimized. Formally, we want:

$$\underset{\mathbf{x}}{\operatorname{argmin}} \, ||\mathbf{r} - \mathbf{P}(\mathbf{x})||^2 \tag{22}$$

Why is this useful? Well because we can actually formulate the problem very straightforward with constraints now. Here is the inverse kinematics problem with an equality constraint:

$$\underset{\mathbf{x}}{\operatorname{argmin}} \, ||\mathbf{r} - \mathbf{P}(\mathbf{x})||^2 \text{ subject to } \mathbf{C}(\mathbf{x}) = 0 \tag{23}$$

Let us focus on Equation 22 first. This is a pretty standard optimization problem. Luckily, optimization is a well studied subject with many known algorithms. I won't go into them all, but I will highlight some of the well known ones. It should not be surprising that these algorithms are all very similar in structure. You start with a guess at $\mathbf{x}$ and iteratively improve that guess until either your guess stops changing beyond a certain threshold or you simply give up. The latter option is important, because these algorithms generally have no guarantees of convergence. Of course there are specific conditions which you can meet to guarantee you convergence for specific algorithms, but in general there is no one size fits all algorithm, at least not that I know of. Luckily the well known algorithms generally do a pretty good job in practice, so that is not too big of a concern.

7.2 Gradient Descent

The most straightforward way to approach non-linear optimization is by a method called gradient descent. This is a really intuitive algorithm: you descend your function along the negative of the gradient, because that is the direction which the function is the steepest. If you keep doing this, you should arrive at a minimum point, in theory. The gradient descent update looks like this:

$$\mathbf{x}_{n+1} = \mathbf{x} - \alpha \nabla f(\mathbf{x}) \tag{24}$$

where α is a fixed or variable step size parameter which you adjust, and $f(\mathbf{x}) = ||\mathbf{r} - \mathbf{P}(\mathbf{x})||^2$ in the case of inverse kinematics.

The problem with gradient descent is that it is known to have many convergence issues. In practice, people often do not use this method because of this problem.

7.3 Gauss-Newton

This method assumes that $f(\mathbf{x})$ is a sum of squared function values, but that is exactly what we have in inverse kinematics, so we can use it. In fact, Gauss-Newton looks very similar to the least squares update we covered in Section 5. The Gauss-Newton update is given as:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - (\mathbf{J}_f^T \mathbf{J}_f)^{-1} \mathbf{J}_f^T (\mathbf{r} - \mathbf{P}(\mathbf{x})) \tag{25}$$

Where $\mathbf{J}_f$ is the Jacobian matrix of $f(\mathbf{x})$, the residual error function.

7.4 Levenberg-Marquardt

At this point you may be wondering if we can play the same trick we did for damped least squares in the Gauss-Newton

method. The answer is yes, and in fact this is exactly what the Levenberg update looks like:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - (\mathbf{J}_f^T\mathbf{J}_f + \gamma\mathbf{I})^{-1}\mathbf{J}_f^T(\mathbf{r} - \mathbf{P}(\mathbf{x})) \quad (26)$$

One trick was provided by Marquardt, who suggested that we use the diagonal entries of $\mathbf{J}_f^T\mathbf{J}_f$ instead of $\mathbf{I}$ in the update as follows:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - (\mathbf{J}_f^T\mathbf{J}_f + \gamma\text{diag}(\mathbf{J}_f^T\mathbf{J}_f))^{-1}\mathbf{J}_f^T(\mathbf{r} - \mathbf{P}(\mathbf{x})) \quad (27)$$

Equation 27 is known as the Levenberg-Marquardt update. As mentioned in Section 5, γ is an adjustable parameter per each iteration. Because of this, Levenberg-Marquardt is often thought of as combining both gradient descent and Gauss-Newton into one algorithm. When γ is small, then Levenberg-Marquardt essentially reduces to Gauss-Newton. When γ is large, then it is similar to gradient descent.

The way we update γ on every iteration is by checking the change in residual. If the change is small, then we can increase γ which causes us to move along the gradient more. If the change is large, then we can decrease γ which lets us focus on reducing the sum of squares in the error. Because of this adaptive behavior, Levenberg-Marquardt often performs very well in practice.

7.5 Lagrange Multipliers

So far we have not exploited the real advantage of the non-linear formulation. Let us consider how to solve Equation 23, or the non-linear problem with equality constraints. Again let us assume that we have q constraints in $\mathbf{C}(\mathbf{x})$. Let $\lambda = [\lambda_1, ..., \lambda_q]$. Define the *Lagrangian* $L(\mathbf{x}, \lambda)$ of $f(\mathbf{x})$ as:

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) - \lambda\mathbf{C}(\mathbf{x}) \quad (28)$$

Now here is the important part. It turns out that solving the non-linear problem given by:

$$\underset{\mathbf{x},\lambda}{\text{argmin}}\, L(\mathbf{x}, \lambda) \quad (29)$$

is completely equivalent to solving the non-linear problem presented in Equation 23. Now this is actually quite easy to see why the constraints are satisfied. Suppose that $\mathbf{x}', \lambda'$ is the solution to Equation 29. Then we know that $\nabla L(\mathbf{x}', \lambda') = 0$, because we are at a minimum. Therefore:

$$\nabla L(\mathbf{x}', \lambda') = \left[\frac{\partial L}{\partial \mathbf{x}}(\mathbf{x}', \lambda), \frac{\partial L}{\partial \lambda}(\mathbf{x}', \lambda')\right] = 0$$

$$\frac{\partial L}{\partial \lambda} = -\mathbf{C}(\mathbf{x}) = 0 \rightarrow \frac{\partial L}{\partial \lambda}(\mathbf{x}', \lambda') = \mathbf{C}(\mathbf{x}') = 0$$

So therefore we see that our solution is guaranteed to have the constraints satisfied. What about optimality? That one is a little tricker, but is formalized in the *Karush-Kuhn-Tucker*, or KKT conditions.

What about inequality constraints? Well now we have all this prior knowledge, inequality constraints are actually quite simple. The key insight is to see that the constraint $\mathbf{C}(\mathbf{x}) \geq 0$ is equivalent to the constraint $\mathbf{C}(\mathbf{x}) - \mathbf{s}^2 = 0$, where $\mathbf{s} = [s_1, ..., s_q]^T, s_i \in \mathbb{R}$ is now a free variable to optimize over. Our new Lagrangian function is therefore:

$$L(\mathbf{x}, \lambda, \mathbf{s}) = f(\mathbf{x}) - \lambda(\mathbf{C}(\mathbf{x}) - \mathbf{s}^2) \quad (30)$$

which gives rise to our non-linear problem formulation:

$$\underset{\mathbf{x},\lambda,\mathbf{s}}{\text{argmin}}\, L(\mathbf{x}, \lambda, \mathbf{s}) \quad (31)$$

If it is not clear why the $\mathbf{s}$ variable guarantees us that $\mathbf{C}(\mathbf{x}) \geq 0$, then suppose towards a contradiction that for some i, $C_i(\mathbf{x}) < 0$. But since $C_i(\mathbf{x}) - s_i^2 = 0$, this would mean that $s_i \in \mathbb{C}$. However, we only allow $s_i \in \mathbb{R}$. This is a contradiction, because the functions we are minimizing are not defined over the complex domain, meaning the minimum cannot be a complex solution.

8. CONCLUSION

In this paper I have surveyed very briefly the commonly used techniques for solving inverse kinematics problems. I hope to have given you a sense for how to approach inverse kinematics programatically.

Additional sources of reading, besides the sources cited in this paper, can be found at [2, 5].

9. REFERENCES

[1] S. R. Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods.

[2] M. P. Engell-Norregard, S. M. Niebe, and M. B. Bonding. Inverse kinematics with constraints. 2007.

[3] N. Joubert. Numerical methods for inverse kinematics.

[4] C. Welman. Inverse kinematics and geometric constraints for articulated figure manipulation. 1989.

[5] J. Zhao and N. Badler. Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Transactions on Graphics*, 13:313–336, 1994.