2.3 Making General IK Well-Posed

The trick of turning a potentially ill-posed system of equations into a well-posed regularized least-squares minimization problem is one of the most powerful, and commonly used, methods in many fields. We can use it directly for Inverse Kinematics.

Turning to our earlier two limb example, we can resolve the problem of a solution not existing by asking for the end-effector to be as close as possible to the target instead of actually at it:

$$\min_{\theta,\phi} \left\| \big(x(\theta,\phi) - x,\, y(\theta,\phi) - y\big) \right\|^2, \tag{17}$$

where the functions $x(\theta,\phi)$ and $y(\theta,\phi)$ are the coordinates of the end-effector as a function of the joint angles—see equation (2).

Now we know there will always be a solution—and a fairly natural one at that (put the end-effector as close to the target as possible)—but we still have the uniqueness issue. If (θ,ϕ) is a solution, then $(\frac{1}{2}\pi - \theta, -\phi)$ is an equally good solution. To get a well-posed problem we have to bias it one way or the other.

One possibility mentioned before is to put in an additional constraint, like $\phi \geq 0$. Depending on the method we use to solve the final optimization problem, an inequality constraint like this may or may not be easy to incorporate. If it does pose trouble, we can again take the regularization approach, or more specifically include a *penalty term* to steer the solver away from the "bad" solution. Define the negative part of ϕ as

$$\phi^- = \begin{cases} 0 & : \phi \geq 0 \\ |\phi| & : \phi < 0 \end{cases}. \tag{18}$$

We can penalize solutions with a negative ϕ by again including a small ϵ constant:

$$\min_{\theta,\phi} \left\| \big(x(\theta,\phi) - x,\, y(\theta,\phi) - y\big) \right\|^2 + \frac{1}{\epsilon}|\phi^-|^2, \tag{19}$$

In fact, there's still a problem for well-posedness if $a \neq b$ and the target is $(0, 0)$, but other than that one problem spot which we can probably safely ignore, we're good.

To the general case: we are faced with a set of nonlinear equations, placing an end-effector in some desired position (for example) by solving for appropriate joint parameters. We can express that as a vector-valued function of the joint parameters being zero:

$$\vec{r}(\theta_1, \ldots, \theta_n) = \vec{0}. \tag{20}$$

Here I've used θ_i for the i'th joint parameter to emphasize they are often angles, but this is not a restriction in general: they could be translations along certain axes, for example. The exact form of $\vec{r}$ is determined by Forward Kinematics.

Our first trick is to replace this with a (nonlinear) least-squares problem that is bound to have a solution even if it is not unique:

$$\min_{\theta_1,\ldots,\theta_n} \|\vec{r}(\theta_1, \ldots, \theta_n)\|^2. \tag{21}$$

We may then regularize this in some way to avoid multiple solutions. For example, penalizing large angles with some tiny parameter ϵ:

$$\min_{\theta_1,\ldots,\theta_n} \|\vec{r}(\theta_1, \ldots, \theta_n)\|^2 + \epsilon\|\vec{\theta}\|^2. \tag{22}$$

Here I'm using $\vec{\theta}$ as an abbreviation for the vector of all joint angles: $\vec{\theta} = (\theta_1, \theta_2, \ldots, \theta_n)$. Since we phrased the regularization as a sum of squares (the Euclidean norm of $\vec{\theta}$ squared), we still have a least-squares problem!

Another common regularization that comes up in IK for animation, or interactive use, is to penalize joint parameters that deviate too much from the last solution (at the previous frame, or the previous mouse position, etc.). If $\vec{\theta}_{\text{old}}$ is the previous solution, we can instead minimize:

$$\min_{\theta_1,\ldots,\theta_n} \|\vec{r}(\theta_1, \ldots, \theta_n)\|^2 + \epsilon\|\vec{\theta} - \vec{\theta}_{\text{old}}\|^2. \tag{23}$$

More advanced ideas along this line are possible, for example Grochow et al.'s idea for biasing the IK solution to stay close to known "good" poses extracted from motion capture of real movement [GMHP04].

Further constraints on IK solutions, such as joint angle limits (e.g. the elbow isn't allowed to bend backwards) or non-penetration of the limb with the body, can be placed as actual *hard constraints* on the least-squares problem, which formally might look like

$$\min_{\vec{\theta}:\, \vec{C}(\vec{\theta}) \geq \vec{0}} \|\vec{r}(\theta_1, \ldots, \theta_n)\|^2 + \epsilon \|\vec{\theta} - \vec{\theta}_{\text{old}}\|^2, \tag{24}$$

using a possibly vector-valued constraint function $\vec{C}(\vec{\theta})$. The statement $\vec{C}(\vec{\theta}) \geq \vec{0}$ here means that each element of the vector-valued function $\vec{C}$ has to be non-negative separately. We call these *hard* constraints because the problem, as stated, requires them to be exactly true at a solution.

Sometimes this greatly complicates methods for solving the problem, or puts us back at the problem of no solution existing because the constraints are inconsistent—for example, $\theta_1 \geq 0$ and $-\theta_1 - \pi \geq 0$ can't both be true. In this case, it may be better to make them *soft constraints*, again adding a penalty term to the *objective function* (what it is we are trying to minimize) like we did with regularization, so as to make sure the solution we get is unlikely to violate the constraints too much. For example, using another small parameter $\mu > 0$, and again the notation $c^- = \max(0, -c)$, we could solve:

$$\min_{\vec{\theta}} \|\vec{r}(\theta_1, \ldots, \theta_n)\|^2 + \epsilon \|\vec{\theta} - \vec{\theta}_{\text{old}}\|^2 + \frac{1}{\mu}[\vec{C}(\vec{\theta})^-]^2. \tag{25}$$

As long as μ is small enough (so $1/\mu$ is big enough) this extra penalty term makes sure the solution can't violate $\vec{C}(\vec{\theta}) \geq 0$ too much.

3 Numerical Optimization

Once we have set up a reasonable optimization problem for IK, we have to write code to solve it. Occasionally it can be solved analytically—perhaps using a computer algebra package to help with differentiating and solving the resulting equations—but most likely that will be too difficult especially if the form of the problem isn't known in advance (it's specified by the artist during runtime).

However, we don't need an exact solution for animation. As long as we can efficiently produce a set of joint parameters which come close enough to solving the problem, so that visually there is no obvious error, we're fine. In fact, since we almost certainly will be using floating-point arithmetic which can't exactly represent the solution due to limited precision, there's usually no point at all in worrying about exactness—only efficiency.

The general idea of *numerical optimization*, algorithms for finding a "good enough" solution to a minimization problem, is to construct a sequence of guesses that eventually will converge to a minimum, stopping at a guess that is found to be good enough. The big questions are how to get from one guess to the next (hopefully better guess), what the initial guess should be, whether the sequence of guesses is guaranteed to eventually converge to a minimum (i.e. if you leave the algorithm running long enough, you can achieve any level of accuracy you desire), and when to stop and call it good enough.

Optimization is a gigantic topic in its own right, and of immense use to a gigantic number of fields, not just IK. Rigourously proving convergence, or even analyzing the problem well enough to understand some of the more sophisticated algorithms, is far out of scope for this course. However, for the small and reasonably well-behaved problems we face, there are several simple and practical algorithms that are good enough. For more on the subject, Nocedal and Wright's book is an excellent resource [NW99].

3.1 Cyclic Coordinate Descent

Before worrying about when to stop the iteration, let's look at one of the simplest and most popular algorithms for IK: *Cyclic Coordinate Descent*, sometimes abbreviated to CCD, and sometimes also called Gauss-Seidel.

Let's simplify the notation a bit first: call the objective function (again, what we are trying to minimize) just $f(\theta_1, \ldots, \theta_n)$. It may have several nonlinear sum-of-squares terms as before, or be constructed some other even more complicated way, but for the purposes of this method we don't need to know anything about f: it can be just a black box, a bit of code we can call to evaluate the objective and that's all.

Cyclic Coordinate Descent works by starting with an initial guess—perhaps the solution from the previous frame in an animation or the previous mouse position in an interactive program, or perhaps just all zeros if there's nothing better—and adjusting one parameter at a time to steadily decrease the value of f. That is, the first substep approximately solves:

$$\theta_1^{\text{new}} = \arg\min_{\theta_1} f(\theta_1, \theta_2, \ldots, \theta_n), \tag{26}$$

where $\theta_2, \ldots, \theta_n$ are held fixed. Once we've found a new value for θ_1 that decreases f reasonably, or discovered θ_1 is already as good as it gets, we then move on to optimizing θ_2 with the other parameters held fixed:

$$\theta_2^{\text{new}} = \arg\min_{\theta_2} f(\theta_1^{\text{new}}, \theta_2, \ldots, \theta_n). \tag{27}$$

Be careful here—I'm using the new value θ_1^{new} found in the first substep, but it's fixed as only θ_2 is allowed to vary here. We proceed on like this, optimizing θ_3, ..., θ_n separately in sequence. That is one sweep of Cyclic Coordinate Descent.

After the first sweep, we are probably not yet at the solution: it may well be that after modifying θ_2, ..., θ_n, there is now a better value for θ_1 that can decrease f even further. So we proceed with another sweep, and then another, and so on until we think we're close enough to the solution. The reason for the name of this algorithm should be clear now: we are cycling through the coordinates of the parameter vector (θ_1, ..., θ_n) trying to "descend" in the value of f.

The general term "descent", which comes up a lot in optimization algorithms, is best understood if you plot the function f over the domain of $\vec{\theta}$, thinking of $f(\vec{\theta})$ as the "height" of the point $\vec{\theta}$ in the domain. The optimization algorithm starts at some point on this plot (the initial guess), and attempts to find a sequence of points that descend into a minimum of f. Cyclic Coordinate Descent finds these points by moving parallel to each coordinate axis (of $\vec{\theta}$) in turn, cycling through the axes again and again, but of course many other strategies are possible.

Every substep in a sweep of Cyclic Coordinate Descent requires approximately solving a minimization problem in a single variable. It can be very approximate, since we'll have to take multiple sweeps anyhow, but we have to make a sufficient improvement for the method to be able to work. This is still a tricky problem, but a lot easier to tackle when we only have one variable to deal with. Especially for IK, it is also often the case that the objective function can be evaluated more efficiently if you know in advance that only one parameter will be modified inside a substep—for example, you can precalculate the transformation up to the joint angle being changed, and the transformation after that joint in the hierarchy, and then as the joint changes only have $O(1)$ work to do in determining the end-effector instead of a full FK tree traversal.