Vol. 16, No. 4: 177–217

Inverse Kinematics – Cyclic Coordinate Descent (CCD)

Ben Kenwright
Newcastle University

Abstract. This article examines the popular *inverse kinematic* (IK) method known as *cyclic coordinate descent* (CCD) and its viability for creating and controlling highly articulated characters (e.g., humans and insects). The reason CCD is so popular is that it is a computationally fast, algorithmically simple, and straight-forward technique for generating IK solutions that can run at interactive frame rates. Whereas it can be relatively clear-cut to construct an IK system using CCD, we address a number of engineering solutions necessary to make the CCD technique a viable and practical method for character-based environments, such as games. We discuss implementation details, limitations (e.g., angle limits, performance tips, convergence problems, oscillation issues, and comfort factors), and their applicability to articulated configurations. Whereas a plain implementation may focus only on a single-linked chained IK problem and disregard multiple connected hierarchical goals (e.g., articulated characters), we examine both cases. We also examine why naïve constructions of the CCD algorithm can be incorrect even, though they converge on a solution. Furthermore, we discuss how the CCD algorithm can be fine-tuned to produce more natural lifelike character poses that can be used to generate realistic motions. Hence, after reading this article, the reader should have the knowledge to design and create an effective and flexible CCD implementation for real-time environments, such as games, while understanding and appreciating the limitations and hazards in a practical situation.

ISSN: 2165-347X print/2165-3488 online

1. Introduction

Inverse kinematics (IK) is an exciting and challenging subject that is used in a wide variety of areas that include the computer generated animated film industry, computer games, robotics, and biomedical. Although there are different methods for solving IK problems (e.g., Jacobian William [88]), the *cyclic coordinate descent* (CCD) method is one of the most computationally fast and least complex to put into practice. The CCD IK method is an iterative numerical algorithm that is straightforward and intuitive to implement while also boasting the added advantage of not requiring any complex matrix math decomposition. However, the reader needs to be fluent with elementary vector mathematics (i.e., cross, dot product, angles).

A basic rudimentary implementation of CCD often requires numerous additional engineering adjustments to make it a feasible and practical solution. These engineering solutions, should, ideally, not over complicate, impair, or affect the system's ability, speed, or robustness. To begin with, the problem starts with an arrangement of interconnected links that can be in an unpredictable arrangement. Maneuvering and arranging these interconnected links into a specific arrangement to achieve a particular goal is the task of the IK solver. However, accounting for the numerous problems (e.g., angular limits, oscillations, numerical errors, local minimums) and producing a reliable solution within an acceptable time (i.e., ideally, real time) can be highly challenging and difficult.

An exceptionally challenging and difficult area for IK is the generation of poses intended for highly articulated characters (e.g., humans). Because character IK problems are highly nonlinear and discontinuous due to the large number of interconnected joint types and angular limits, we discuss how the CCD algorithm can be applied to a character system and how we can modify the basic underlying implementation to cope with the added complexity (i.e., multiple end-effectors, priority control, comfort factors) while remaining computationally fast and robust. Furthermore, we briefly discuss the limitations and advantages of a character-based IK solution for modifying pre-recorded motion capture (MOCAP) animation data so it can be more interactive and engaging in order to produce characters that are more realistic and life like. In view of the fact that it can be costly to create a database of MOCAP animations, it can be useful and beneficial if we can retarget these MOCAP libraries using IK to fit dissimilar skeleton constructions (i.e., different number of bones and dimensions) easily.

In summary, the CCD method is a heuristic (i.e., *trial-and-error*) iterative approach that is ideal for real-time applications such as games because it has a very low computational overhead per joint iteration. Furthermore, CCD is able to solve IK problems without the need of complex mathematics or matrix manipulations. However, it requires various engineering modifications,

which we present here, to make the technique a viable solution for a complex IK system (e.g., a character with limits, comfort factors, and weighted links). Although the basic CCD is designed for serial chains, it can be difficult to modify the solution to work well with multiple conflicting end-effector goals. We also address the problems of erratic discontinuities and oscillation conditions whereby the solution never converges.

There are a few governing factors that are desirable in a practical IK solver that can be used for character-based problems. To summarize, a number of factors that we look at in this article and address with regard to the CCD approach are as follows:

- Complexity—avoid an over-engineered solution
- Real-time—fast as possible, numerical examples
- Diverse skeleton types (human, alien, spiders, snakes)
- Keeping balance (center-of-mass position control)
- Adaptation to constraints (feet on the ground, swap base constraints)
- Morphological adaptation (retargeting—useful for adapting motion capture)
- Fast enough multiple figures (crowds)
- Coherency—for small changes to the problem, the solver should find similar answers

2. Background

IK is a hot topic of research across numerous disciplines (e.g., graphics, robotics, and biomechanics), and there is a wide variety of interesting and novel papers and articles on the subject; however, we present a review only of past and recent literature that deals specifically with the CCD technique here, starting with its initial discovery and how it has been extended and modified over the past couple of decades to incorporate numerous enhancements to make it a more flexible, fast, and robust IK solution.

Because of the straightforwardness and simplicity of the CCD approach, it is surprising that the method was not published earlier. However, the discovery of the CCD method is credited to Wang and Chen [Wang and Chen 91] who published the technique in 1991 in a paper called "A Combined Optimization Method for Solving the Inverse Kinematics Problems of Mechanical Manipulators" for robotics. Then, a couple of years later, Welman [Welman 93] extended Wang's and Chen's work by including biomechanical constraints (i.e., angular limits).

Since the original proposal of CCD IK, there have been many interesting and novel articles published that have extended or used the approach while demonstrating its advantages and potential. In fact, Lander [Lander 98] presented an easy-going article on implementing the CCD IK for singly linked chains in real-time systems, in addition to discussing the numerous advantages of the method for interactive environments (i.e., games).

Although the initial research showed the potential of the CCD IK method for single-linked chain problems, Shin and colleagues [Shin et al. 01] modified the CCD method so that it could be used for human-like figures by subdividing the hierarchy into regions and iteratively adapting each subregion's solution until an answer was found. However, not long after, Kulpa and Multon [Kulpa and Multon] extended Shin and co-authors' [Shin et al. 01] work by producing more human-like and natural-looking character poses by again dividing the humanoid body into subgroups near the end-effectors (i.e., head, trunk, arms, and legs) but extending the model to encapsulate a *center-of-mass* (COM) control. The CCD algorithm was also used by Ren and colleagues [Ren et al. 10] for a humanoid character to generate running motions in real time; favoring the CCD method over the Jacobian approach in order to avoid singular values.

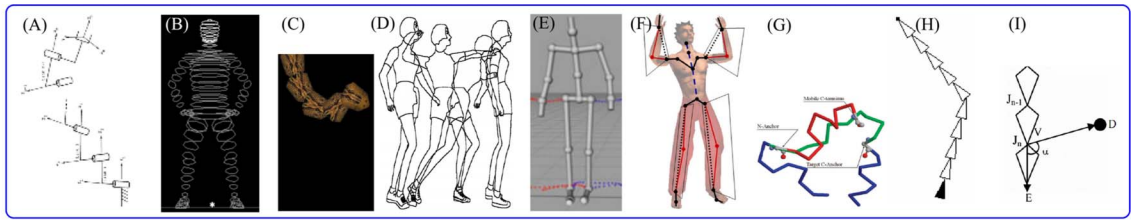
Furthermore, the speed and robustness of the CCD method makes it an ideal choice for solving a wide variety of IK problems. For example, Mahmudi and Kallmann [Mahmudi and Kallmann 11] used CCD for solving IK problems in highly constrained kinematic character chains because it offered the fastest results when compared with a Jacobian-based pseudo-inverse solver method. The IK method was primarily used to control feature-based locomotion animations for characters and incorporated tolerance boundaries to produce less-stiff and more-natural poses. Canutescu and Roland [Canutescu and Roland] applied the CCD algorithm in a closed loop configuration for protein structures.

There have been articles written that have described numerical comparisons between the CCD method and other approaches to show and discuss the computational advantages and straightforwardness of the method. For example, Mukundan [Mukundan 09] presents a single-pass IK solution (for single chains) in an analytical comparison with the CCD technique, and Fêdor [Fêdor 03] does a comparison of different real-time methods, including the CCD, and their applicability for manipulating complex skeletons.

A comparison grid is given in Table 1 to show a clear visual contrast of some of the important differences, modification, and extensions to the CCD technique by various researchers.

3. Overview

Essentially, IK is the method of finding one or more joint angles (and/or link lengths) that enable a collection of joints and links to accomplish an



	Wang & Chen 1991	Welman 1993	Lander 1998	Shins et al. 2001	Mahmudi 2011	Kulpa & Multon 2005	Canutescu & Dunbrack 2003	Mukundan 2009	Ren et al. 2010
Research Area	R	G	G	G	G	R	O	G	G
Real-Time	√	√	√	√	√	√		√	√
Joint Limits	√	√			√	√			
Skeleton Type	C	B	C	B	B	B	C	C	B
Multiple End-Effectors		√		√		√			
Multiple Figures (Crowds)						√			
Swap Base-Root					√	√			√
Handle Conflicts Gracefully		√				√	√		
Dimensions	2D/3D	3D	2D	3D	3D	3D	3D	2D	2D

R(Robotics), G(Graphics), Other(i.e.,Biomechanics)
C(Chain), B(Biped), Other(i.e., Spiders)
√ indicates the mention of the availability of the features in the article

Table 1. A comparison of the different properties that have been attained through the extension of the CCD IK technique. (a) Wang and Chen [91], (b) Welman [93], (c) Lander [98], (d) Shin et al. [01], (e) Mahmudi and Kallmann [11], (f) Kulpa and Multon [05], (g) Canutescu and Roland [03], (h) Mukundan [09], (i) Ren et al. [10].

objective. The system of links can be as simple as a single lever or as complex as a human body and can have a single, multiple, or even no solution to the problem. Hence, for each situation, we ideally want the IK solver to find the most satisfactory (e.g., least movement) or best-guess approximation (i.e., when no solution exists) to the problem.

Although there can be any number of IK joint constraint types, the two most common are the revolute and prismatic. However, for this article, we focus on character-based IK problems that can be represented solely using angular joint (i.e., revolute) types. Furthermore, we compound the IK explanation and demonstration into a task based end-effector problem that evolves around the calculation of joint axes and angles needed to place one or more end-effectors at specific goals. For example, a simple multi end-effector IK system composed of revolute joints and fixed length links is shown in Figure 1.

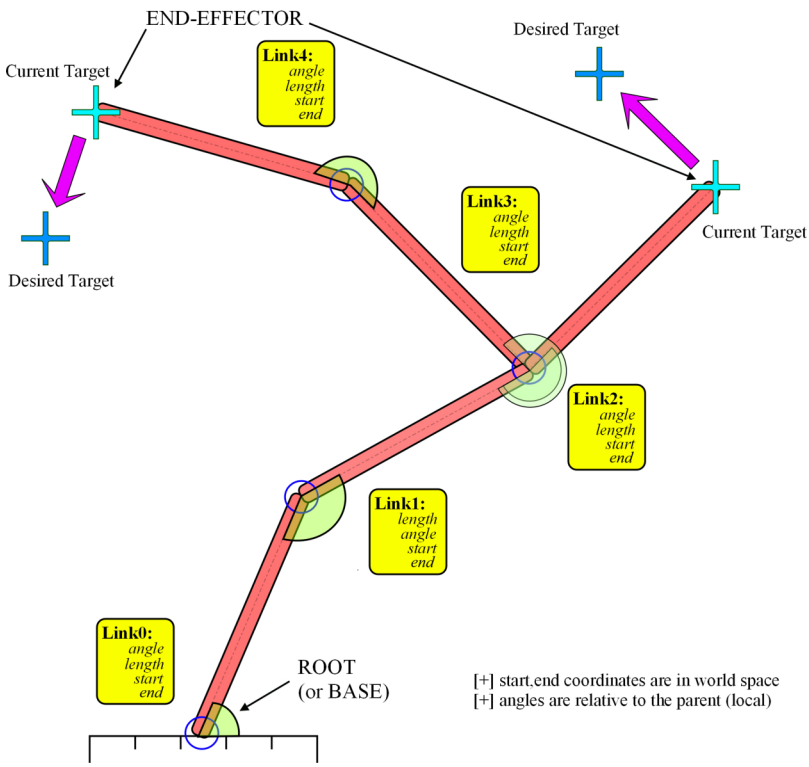


Figure 1. A simple IK problem with links, revolute joints, and two end-effectors.

4. Algorithm

First, we need to be clear about what we have and what problem we are trying to solve. Essentially, we are searching for a complete set of joint angles that will position the end-effectors at their desired locations. At any point, we can use the complete set of joint angles to calculate the end-effectors' distance error to the target. To do this, we propagate each link's angle and length forward from the base of the connected hierarchy to the end-effectors (i.e., forward kinematics). At this point, we can identify the error between the end-effectors and the targets, for example, as shown in Figure 2.

The CCD algorithm goes from *joint-to-joint* and rotates the end-effector as close as possible to the target. After each iterative update, the algorithm measures the distance between the end-effector and the target to decide if it is close enough and should exit. Furthermore, to avoid infinite recursive loops due to unreachable and conflicting goals, the algorithm must set a maximum iteration count.

In essence, the base-root of the IK chain is *immovable* while each link is iteratively rotated around a specified axis and angle degree to position the end-effector as close to the target as possible. This axis and angle is calculated using Equation (1) and is illustrated in Figure 2; it is also applicable to both 2D and 3D coordinates.

Forthrightly, a reader keen on implementing the CCD algorithm at this point might naïvely make the mistake of inadvertently iterating from the leaf

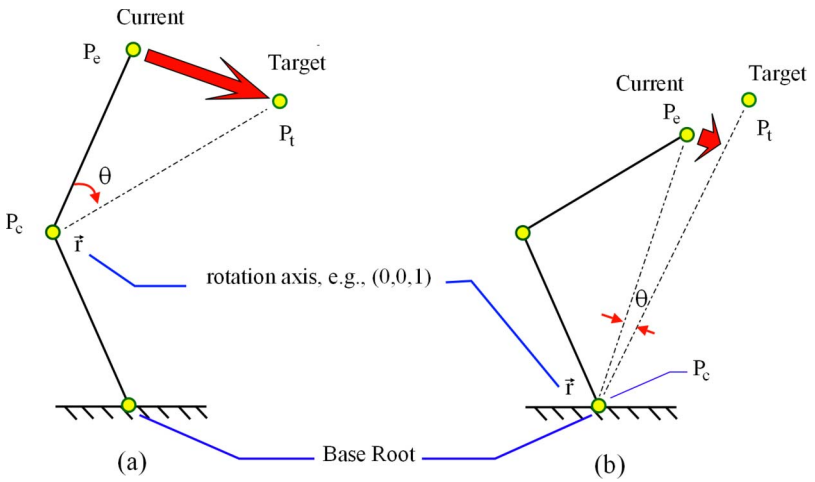


Figure 2. Shows a simple two-link chain with a single end-effector, where P_e, P_c, and P_t correspond to three positions in world space, and $\vec{r}$ is the axis of rotation for the link.

(i.e., end-effector) *all the way* to the base-root to find a solution. By "all the way," we mean that every link in the connected chain of links is iteratively checked and updated to move the end-effector closer to the target. However, as we point out later, we can correctively update the links in any order to achieve different results; iteratively correcting links toward the one end of the linked chain will project the majority of the movement to those joints at that end. For example, this can be desirable for a character's hand movement when it is favored that we move only his lower and upper arm to reach the target (see Figure 9) (i.e., not his pelvis and feet). Furthermore, this can be beneficial when multiple end-effectors (e.g., left and right arms) are both reaching for different goals—they can avoid conflicting and fighting with each other if we move only the links necessary to reach the final targets.

We give a straightforward, clear-cut, step-by-step example of a three-link chain with a single end-effector to help the reader get a definitive understanding of how the CCD algorithm works, in Figure 3. The example starts at the joint closest to the end-effector of the linked-chain and rotates the link so that the end-effector is as close as possible to the target. It then moves onto the next link the chain and repeats. Furthermore after each iteration, it checks the end-effector's distance to the target, and if it is within a specified tolerance, it has reached its goal and exits.

$$\begin{aligned} cos\,(\theta) &= \frac{P_e - P_c}{\|P_e - P_c\|} \cdot \frac{P_t - P_c}{\|P_t - P_c\|} \\ \vec{r} &= \frac{P_e - P_c}{\|P_e - P_c\|} \times \frac{P_t - P_c}{\|P_t - P_c\|}, \end{aligned} \tag{1}$$

where the values P_e, P_c, and P_t refer to the positions in Figure 2.

The reader should be able to follow the systematic example in conjunction with the explanation of the three-link chain shown in Figure 3 to ascertain a fundamental understanding of how the CCD IK algorithm works. Furthermore, Figure 3 helps the reader to clearly see how the two end links quickly converge on the target, while at the same time oscillating back and forth to complement each other on reaching a solution.

A few observations worth consideration about the CCD IK algorithm:

- Which joints and in which order they are updated (e.g., top-down/bottom-up, or joints with the maximum error first)
- Corrective angular amount (exact or scaled up/down; e.g., more reliable and smoother transition)
- Reach tolerances (e.g., regions for the control of the final posture)

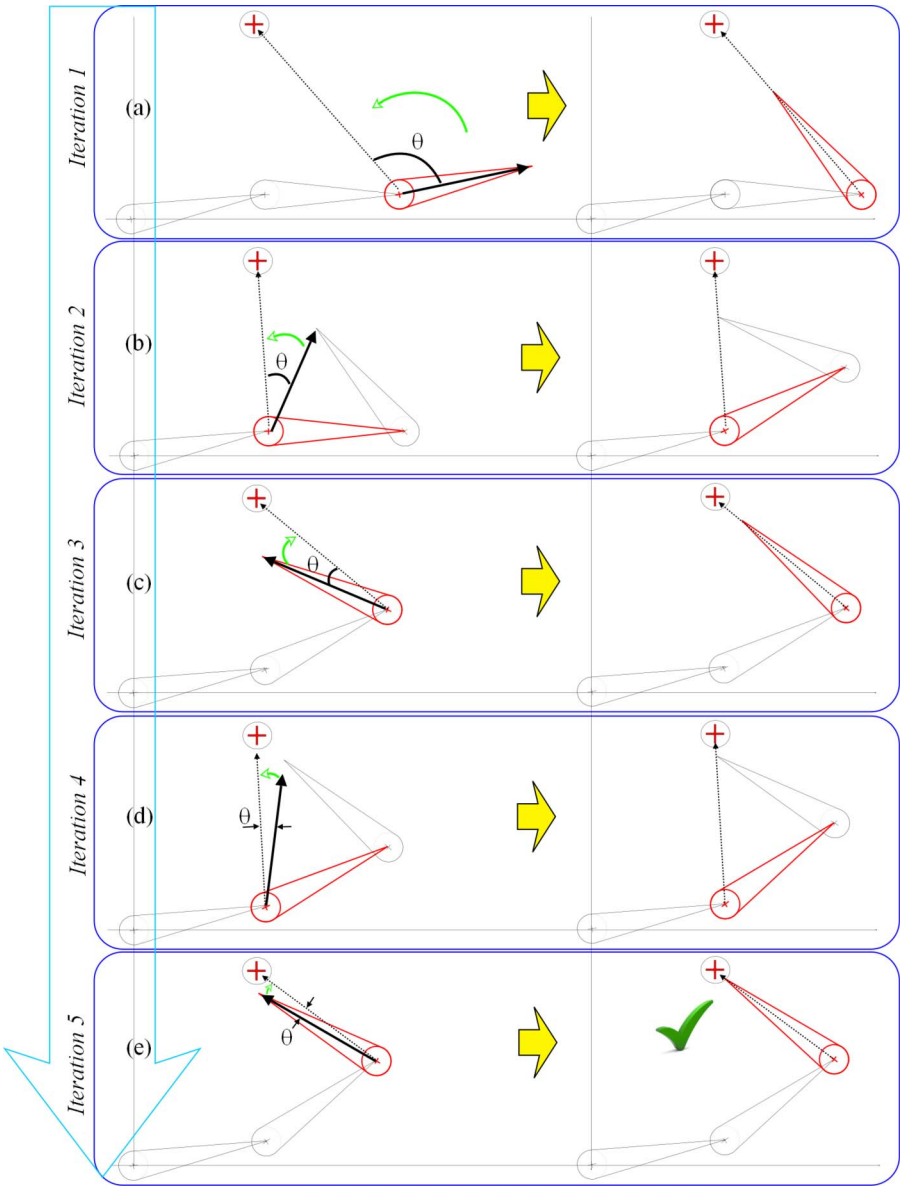


Figure 3. Step-by-step example of a three-link chain with a single end-effector that recursively iterates backward from the leaf to the base-root to move the end-effector toward the target.

5. Top or Bottom (Forward or Backward)

For a single-linked chain that is iterated along the full length, we can start at either the leaf or root link and iteratively update the joint angles to move the end-effector closer to its desired target.

For example, an extremely simple implementation of an interconnected chain of links in 2D requires that we store each link's length and angle (as shown in Figure 1). This has the advantage of having a minimalistic memory overhead. Furthermore, the algorithm does not require any memory overhead (e.g., allocation of large blocks of memory as used by matrix-based approaches [William 88, Kenwright 126]).

However, the reader may ask, why would we want to start from the bottom and why would we want to start from the top of the linked chain? Well, it depends on the circumstances and the type of effect you are aiming for. For example, Mahmudi and Kallmann's [Mahmudi and Kallmann 11] CCD IK implementation was solved from the base toward the end-effector so that the final locomotive motions were more natural, because a majority of a character's movement during walking comes from its feet (i.e., a top-down implementation from the end-effector would produce a robot-like walk). A bottom-up approach that works from the root toward the end-effectors will incur movement with links furthest away from the end-effectors target because the corrective rotations start at the root and propagate outward to the end-effectors.

Alternatively, for arms and hands, it is more desirable to go from the end-effectors toward the base, because this offers the least disruption of the lower links. Again, this makes sense, for an articulated human character; for example, if we reach for an object, the majority of the motion would be in our upper body (i.e., arms and pelvis) with little or no movement of our legs or feet. Finally, for a bottom-up implementation, extra effort needs to be incorporated into the logic to determine the path from the root to the end-effector for articulated configurations with multiple (i.e., tree-like) ends.

Figure 4, Figure 5, and Figure 6 show different examples of the contrasting results between the initial starting pose and the results from using a bottom-up and top-down solution. It can be seen that starting at the root (i.e., the fixed base) or the leaf node (i.e., end-effector), they both converge on a solution with the same number of iterations. The major point is that, for the bottom-up (i.e., forward) approach, all the links will move. One other point which will be important later on when we discuss character motions, is that the *end-effector does not necessarily move as the crow flies* (i.e., directly from the initial starting position toward the final goal) but instead can oscillate and circle the solution multiple times before reaching a stop distance within the predefined tolerance.

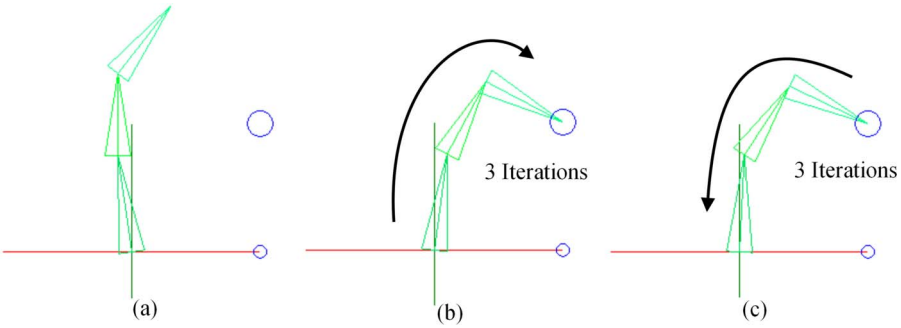


Figure 4. (a) Initial pose, (b) forward, and (c) backward solution.

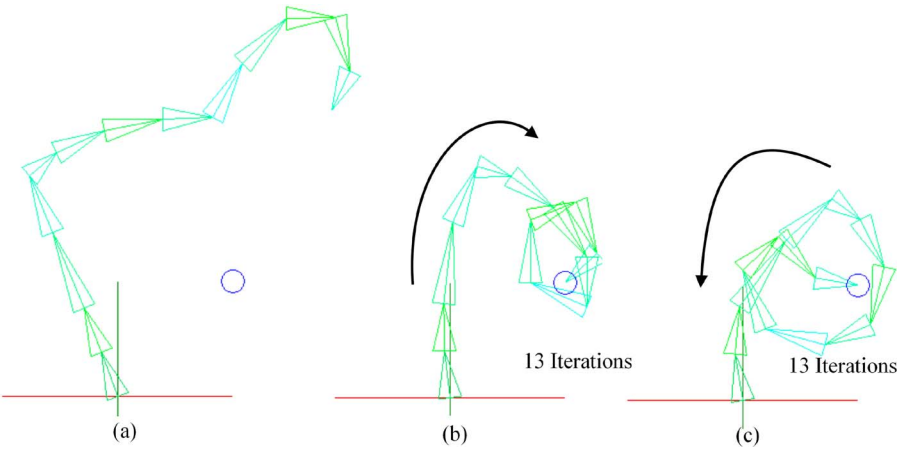


Figure 5. (a) Initial pose, (b) forward and (c) backward solution.

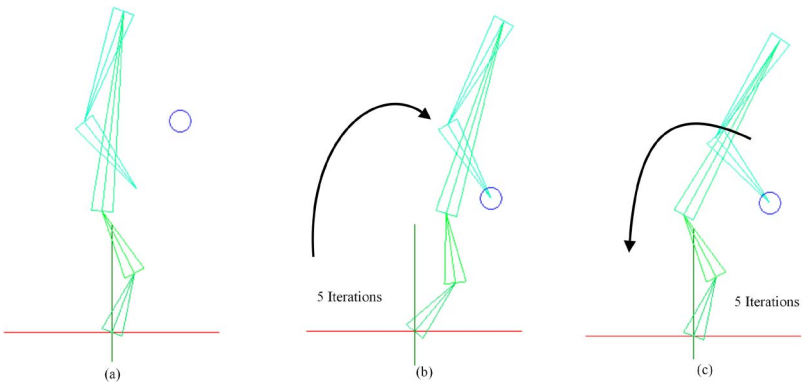


Figure 6. (a) Initial simple pose reach starting; (b) forward and (c) backward.

The CCD algorithms for top-down and bottom-up approaches are shown in Listing 1 and Listing 2. Furthermore, it should be noted that Aristidou and Lasenby combine both forward and backward CCD methods successively to improve performance (i.e., reduce the number of iterations) and perhaps also the homogeneity of the solution in an interesting and detailed paper called "FABRIK: a Fast, Iterative Solver for the Inverse Kinematics Problem" [Aristidou and Lasenby 11].

5.1. Bouncing

We can reduce movement of limbs further away from the start end by recursively redoing previous links to bring them closer when we rotate children links. Hence, as each link is rotated we recursively jump (i.e., bounce) back to the starting end and incorporate the changes back into earlier links. The bounce approach does not include any intelligent logic, it just recursively bounces back to the start after each new link is updated to project the majority of the link's movement onto the starting end. For example, the implementation of the bounce code can be done with a nested loop as shown in Listing 3.

```
1: procedure BackwardCCDIK // i.e., Top–Downward
2:   Input e          // threshhold
3:   Input kmax        // max iterations
4:   Input n           // link number (0 to numLinks–1 chain)
5:
6:   k = 0          // iteration count
7:   while ( k < kmax )
8:   {
9:             for (int i=n−1; i¿0; −i) // link index (starting at leaf)
10:            {
11:                Compute u, v            // vector Pe–Pc, Pt–Pc (see Fig 2)
12:                Compute ang            // using Equation 1
13:                Compute axis           // using Equation 1
14:                Perform axis–angle rotation (ang,axis) of link i
15:                Compute new link positions
16:
17:                if (|Pe–Pt| < e)      // reached target
18:                {
19:                    return;           // done
20:                }
21:            }
22:            k++;
23:  }
24: end procedure
```

Listing 1. Algorithm to demonstrate what we mean, in code, for the bottom-up update of chain of links.

```
1: procedure ForwardCCDIK // i.e., Bottom–Upward
2:     Input e                    // threshhold
3:     Input kmax                 // max iterations
4:     Input n                    // link number (0 to numLinks–1 chain)
5:
6:     k = 0              // iteration count
7:     while ( k < kmax )
8:     {
9:         for (int i=0; i¡n; ++i) // link index (starting at base)
10:        {
11:             Compute u, v        // vector Pe–Pc, Pt–Pc (see Fig 2)
12:            Compute ang          // using Equation 1
13:            Compute axis         // using Equation 1
14:            Perform axis–angle rotation (ang,axis) of link i
15:            Compute new link positions
16:
17:            if (|Pe–Pt| < e)          // reached target
18:            {
19:                return;               // done
20:            }
21:        }
22:        k++;
23:    }
24: end procedure
```

Listing 2. Algorithm to demonstrate what we mean, in code, for a top-down of the chain of links.

The bounce can start at either the root or the end-effector link; for example, see Figure 7 and Figure 8. This projects the majority of the link movement to either the start or the end. Because there is no smart logic in the detection of movement, the movement is mostly on the end that we start from but can still induce movement in the whole chain.

5.2. "Smart" Bounce

The bouncing method is a dumb and simple method of ensuring that the majority of movement occurs at the start end of the iteration. It does not include any checking or analyses if a link moved, and if it should return to the start to incorporate the changes. Hence, we can incorporate additional tests to decide if it is necessary to move the links further along the chain, or if it can be accomplished by moving only the links at the start end (i.e., only move links if necessary).

We extend the bounce strategy of repeatedly returning to the start and working back along the links (as shown in Listing 4). However, we include a "smart" check that identifies if any movement has occurred while we are moving down the link. If a movement has occurred, then we return to

```
1: // Top-Downward 'Bounce'–
2: // recursively returning to the
3: // top-reduce movement of the lower links
4: procedure BackwardBounceCCDIK
5:     Input e          // threshhold
6:     Input kmax       // max iterations
7:     Input n          // link number (0 to numLinks-1 chain)
8:
9:     k = 0            // iteration count
10:    while ( k < kmax )
11:    {
12:        for (int b=0; b¡n-1; ++b)
13:        {
14:            for (int i=n-1; i¿b-1;−i)  // link index (starting at leaf)
15:            {
16:                Compute u, v     // vector Pe-Pc, Pt-Pc (see Fig 2)
17:                Compute ang      // using Equation 1
18:                Compute axis     // using Equation 1
19:                Perform axis-angle rotation (ang,axis) of link i
20:                Compute new link positions
21:
22:                if (|Pe-Pt|<e)           // reached target
23:                {
24:                    return;              // done
25:                }
26:
27:            // e.g. (link 5 - starting at the top):
28:            // 5
29:            // 5, 4
30:            // 5, 4, 3
31:            // 5, 4, 3, 2
32:            // ...
33:            }
34:        }
35:        k++;
36:    }
37: end procedure
```

Listing 3. Algorithm to demonstrate what we mean, in code, for the simple bounce approach for updating each link.

the start and iterate along the linked hierarchy again. This has the added benefit of preventing the unnecessary movement of links (i.e., limbs) that are furthest away from the start end. This is highly beneficial later on when we have multiple end-effector constraints connected to a shared hierarchy. For example, if we have a character with two arms and both arms are being moved about, if they can reach their targets without affecting the body or pelvis this means that the two movements will not conflict or cause a problem in solving a solution for the IK problem (e.g., see Figure 9).

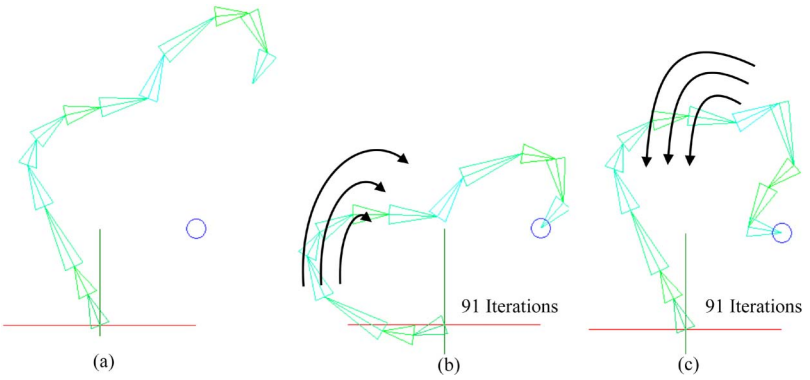


Figure 7. (a) Initial pose, (b) bouncing forward and (c) bouncing backward.

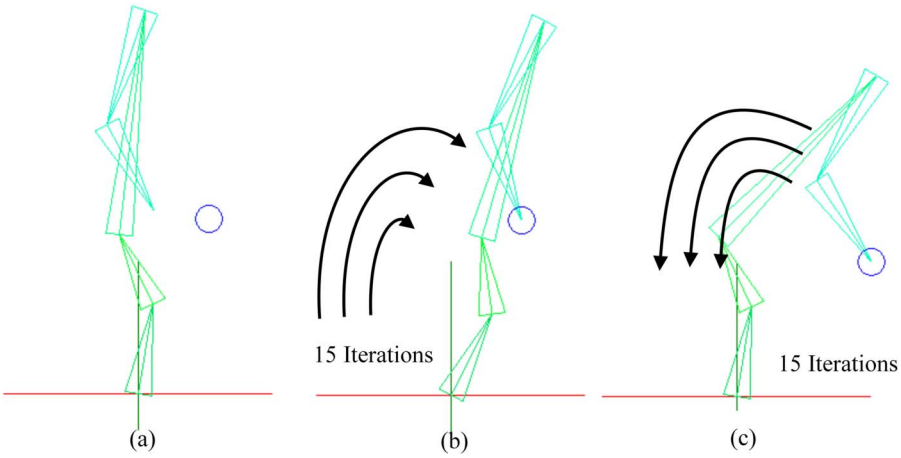


Figure 8. (a) Initial character pose, (b) bouncing forward, and (c) bouncing backward.

The downside, however, of a "smart" bounce implementation is that we need to incorporate an additional check into each iteration. Furthermore, it can require more iterations to converge on a solution because it has to return to the start more times and move smaller corrective rotations of the start links. For long chains, it has the added advantage of moving only the necessary number of chains needed.

Note, for small IK problems (e.g., one or two links) an analytical solution is a fast and easy approach that we can apply to solving leaf node IK problems (i.e., the first and second start ends). Whereby, mixing in an analytical solution would quickly allow us to adapt end links if necessary with a lower computation cost, compared with completely redoing the iterative process, as

```
1:// Top-Downward 'Smart Bounce'-
2:// Analyze and returning to the
3:// top-prevent movement of the lower
4:// links if not needed
5: procedure BackwardSmartCCDIK
6:     Input e              // threshhold
7:     Input kmax          //max iterations
8:     Input n              // link number (0 to numLinks-1 chain)
9:
10:     k=0              // iteration count
11:     while (true)
12:     {
13:          for (int i=n−1;i>0; −−i) // link index (starting at leaf)
14:          {
15:               Compute u, v       // vector Pe-Pc, Pt-Pc (see Fig 2)
16:               Compute ang        // using Equation 1
17:               Compute axis       // using Equation 1
18:               Perform axis-angle rotation (ang, axis) of link i
19:               Compute new link positions
20:
21:               if (|Pe-Pt|<e)                  // reached target
22:               {
23:                    return;                  // done
24:               }
25:
26:               if ((k/n) ¡ kmax)    // divide by n so iterations
27:               {                              // count is per chain length
28:                    return;
29:               }
30:
31:               // If the link's corrective rotations exceeds
32:               // the tolerance-redo other links
33:               if (angle ¿ 0.001)
34:               {
35:                    //Restart at the end-link
36:                    i = n+1; // +1 'for' loop subtracts 1
37:               }
38:               k++
39:          }
40:     }
41:  end procedure
```

Listing 4. Algorithm to demonstrate what we mean, in code, by a smart bounce method of updating links (end-effector backward towards the base).

done by Shin and colleagues [Shin et al. 01] and Kulpa and Multon [Kulpa and Multon 05].

For articulated character figures, it is desirable that a small change in end-effector position corresponds to a small coherent change in joint angles (i.e., we want to avoid sporadic random jumps for small end-effector movements).

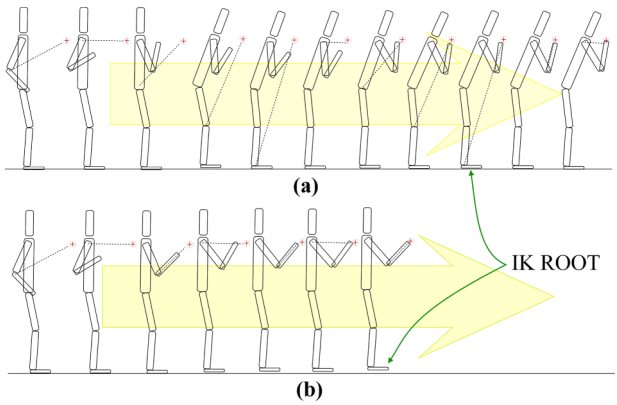


Figure 9. (a) Iterating all the way to the root joint to accomplish the end-effector reach. (b) Alternative, smarter method of jumping back to the start joint after each iteration if the previous links can still be moved closer to the target.

Hence, the smart bounce CCD implementation accomplishes this by moving only those links necessary to accomplish the task. For example, if we take a human character and we have his hand move forward and pick up an object, we can accomplish this by moving only two links (i.e., lower and upper arm).

The advantage of rotating links that only need to move to reach the solution means we have a lower computational cost. Furthermore, the solution is more stable because there is less movement and fewer oscillations of the links around the answer.

The example in Figure 10 shows a simple skeleton pose with a single end-effector being moved around. However, the crucial information is that we

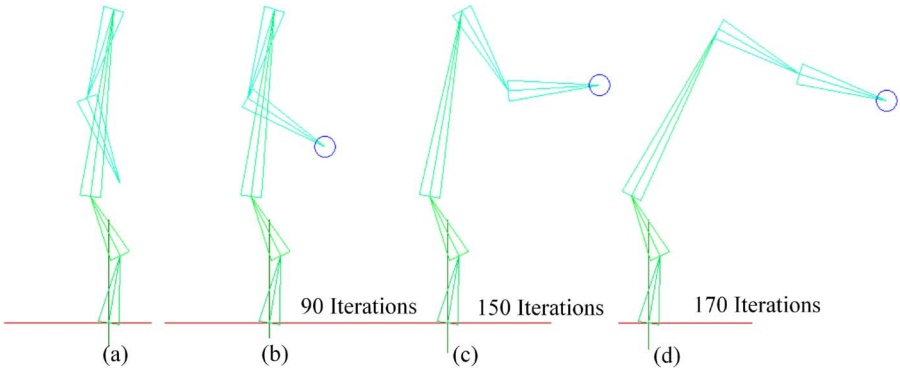


Figure 10. (a) Initial pose, (b)–(d) moving the links that need to be moved to reach the target—smart approach.

are now moving only the upper links to achieve the target. Furthermore, we are using more iterations than previous versions that do not check if we can continue reaching the target with the end-effector links. In addition, we do not need to update the rest of the hierarchy, which can be highly beneficial if we have multiple end-effectors, because they will not interact or interfere with one another (e.g., the left and right arms).

6. Link Distance Limits

We can incorporate a maximum link count limit for each end-effector, so links over a predefined number of steps away from the end-effector are not updated. This can provide us with additional posture and performance control and prevent certain end-effectors disrupting and affecting links that we do not want them to move. For example, we can define a maximum number of link steps that can be updated by an end-effector of a character's hand so that only the lower and upper arms are allowed to move to reach the target. Any further links beyond the link count (i.e., past the upper elbow) will bes ignored.

7. Angular Limits

Animated characters (e.g., human figures) have limits on how they can move and reach. For example, when we simulate an arm reaching we do not want it to bend in unnatural and impossible ways. We want to impose joint constraints, such as angular limits and degrees of freedom (DOF) that can represent the elbow or knee. Each joint can twist and stretch only so far. This also raises the question of comfort because it can appear more natural to move certain limbs to achieve a task (e.g., reach with our arms or bend our legs to pick up an object). A popular method that is used in IK schemes [Kenwright 126] and what we employ in this article is to iteratively clamp the angles after each update to ensure that they always remain within acceptable limits.

Angular limits can be the most challenging and difficult aspect of the CCD IK solution. After each iteration, the limits are checked and clamped to ensure that they are always enforced. However, it can affect CCD's ability to converge on a solution. Essentially, CCD is blind and will repeatedly keep trying a particular route, even though after each increment the joint gets pushed back because of angle limits. We can detect this scenario and attempt to solve this problem by restarting the joints at different random locations (known as simulation annealing—see Dowsland [Dowsland 95] for an easy-to-follow introductory explanation).

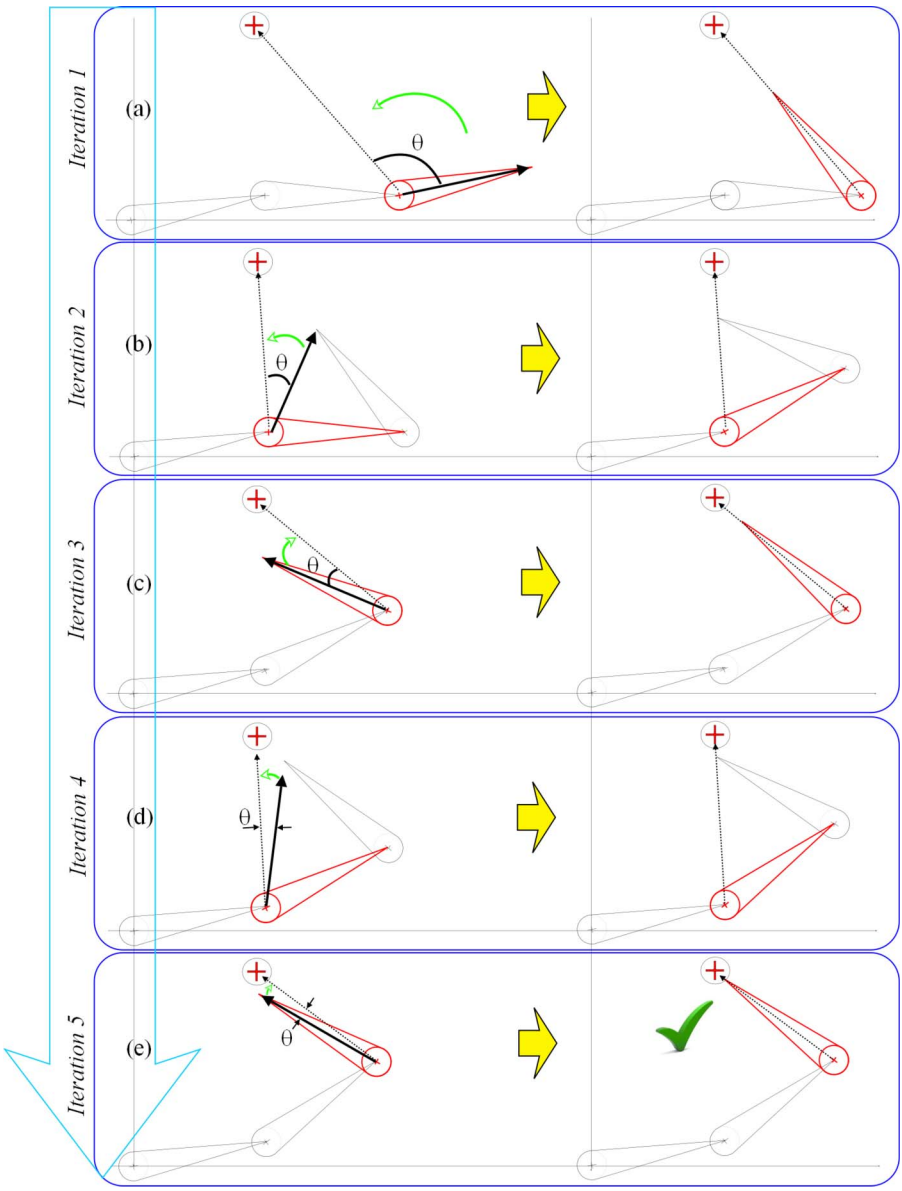


Figure 11. Step-by-step example of a three-link chain from (a) to (e) recursively moving the end-effector position closer to its desired target.

Simulation annealing approaches:

- Random joint pose
- Start with a prestored joint pose (large number of best poses from which we try to guess the best, e.g., approximate distance error)—difficult to find the closest cluster match

Considerations:

- Tunneling (jumping across angular limits)
- Longest or shortest path
- Randomly orientating joints to find solutions

For systems that desire smooth interpolation, the CCD IK system iteratively churns alway and finds the solution of angles for the overall system that we pass to the interpolation algorithm for the smooth interpolation (e.g., quaternion spherical linear interpolation—i.e., SLERP—for continuous smooth movement).

Because we are using discrete values after updating the links with their new angles, we may have jumped into an invalid angular region and should determine the shortest distance to rotate the link to get it out of penetration.

7.1. Visualizing Angular Joint Limits

It can be difficult to visualize the joint limits on complex character models. However, a cone offers a good visual representation of the upper and lower boundaries of rotation available to the link. Note, the joint's angular limits are with respect to the parent's orientation; hence, a cone boundary can be drawn using the parent's transform, because the upper and lower limits are relative to the parent. For example, Figure 12 shows the constraint cones for a simple 2D chain.

There are numerous ways of representing and calculating the angular joint limits. For example, one approach is to use Euler angles, whereby, you specify upper and lower limits with respect to the parent. In 2D this may be fine, however, for 3D it is necessary to use a less ambiguous method (e.g., quaternions). For joints with multiple degrees of freedom (e.g., a ball joint), we can decompose the quaternion into individual components using a "twist-and-swing" approach so that we can reliably and robustly clamp 3D angular limits.

In this study, we follow the convention that the bone's local transform is orientated along the z-axis (i.e., if you are looking along the bone you will be looking down the z-axis). Furthermore, we use quaternions to represent

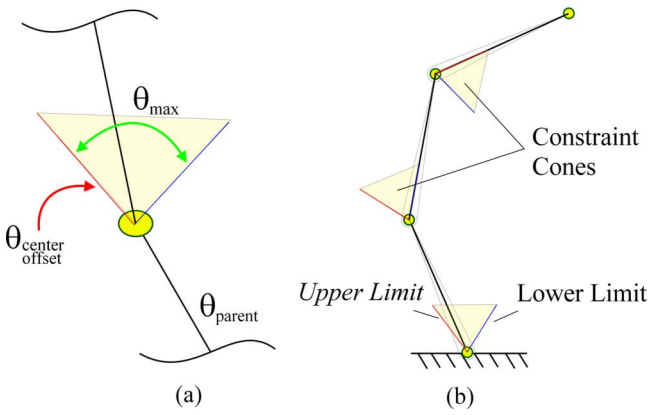


Figure 12. Visualizing constraint limits as cones.

the joint angle transforms, so it is relatively easy and computationally fast to calculate angular differences in order to identify limits. If a link rotates out of its allowed region, it is snapped back to the limit edge (e.g., by means of axis-angle and twist-and-swing decomposition).

Quaternions allow us to manage angular differences, for example:

- Direction of the link (or bone)
- Orientation—up, right, forward for each link (or bone)
- Handle twisting

8. Over and Under Damping (Angle Delta)

The CCD iterative algorithm can take a number of oscillatory iterations to converage on an acceptable solution. We can reduce the number of iterations by introducing a biasing factor into each iteration's corrective rotation. Whereas, by default, we use the angle to rotate the limb, so the end-effector is as close as possible to the target. However, to accelerate convergence, we multiply the angle by a bias factor (e.g., 1.1 to 1.2). This means the end-effector will overshoot and pass the target; nevertheless, for a serial chain of links, it can reduce the number of iterations needed for the end-effector to converge on a solution. For example, Figure 13 shows a step-by-step example with a biasing factor of 1.2 compared with the example in Figure 11 that had no biasing.

On the other hand, it is difficult to choose an under-damped (i.e., magnification) value that provides an overall stable and fast solution. Furthermore, if

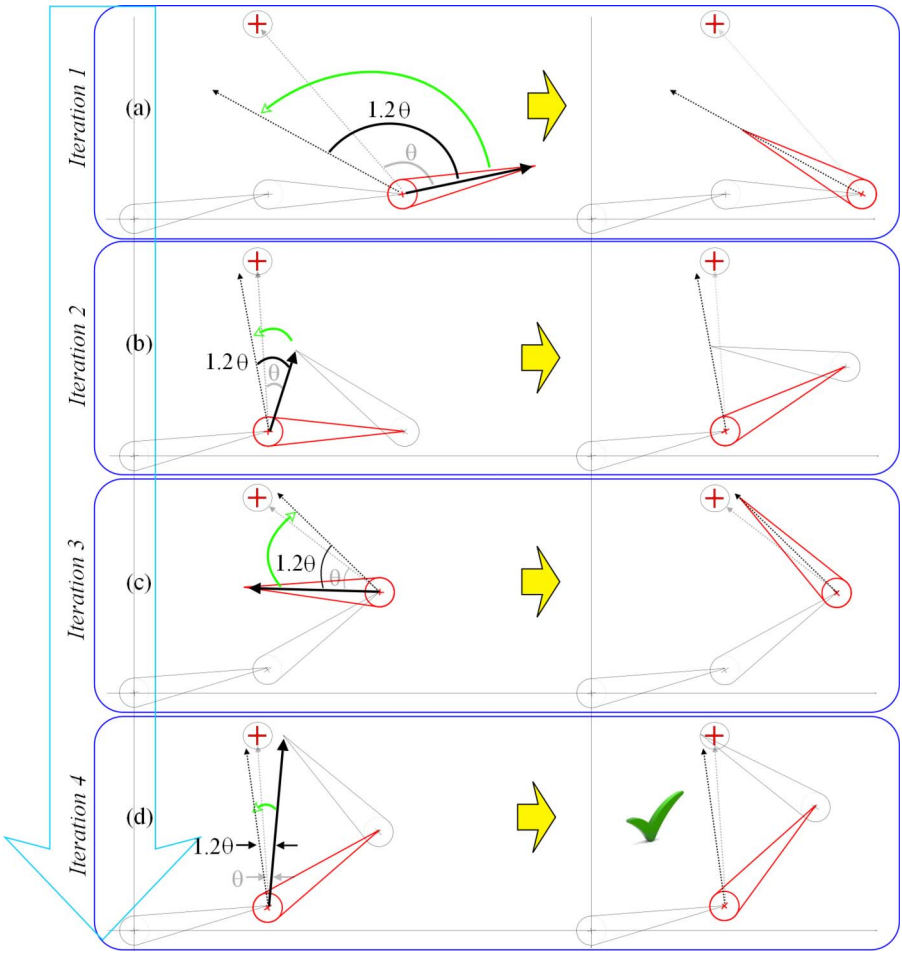


Figure 13. Step-by-step iterative example of the corrective angle scaled by 1.2 to accelerate the convergence on a solution.

the value is to large, it can ultimately result in large oscillations and prevent convergence. However, we can improve convergence by adding an additional feedback constant. This feedback constant is the error magnitude (i.e., distance between end-effector and target). We multiply the error feedback with the under-damped factor to produce smaller values closer to the target (i.e., to reduce the overshoot error). One final note: in practice, it is crucial that the under-damped values are clamped to acceptable limits (i.e., min and max range) so the simulation always remains stable and controlled.

Alternatively, instead of magnifying the result to achieve a faster convergence, we can use a dampening magnitude that is less than 1.0 (e.g., 0.9 to 0.8). This can produce a smoother result with less overshoot of the target. However, it can again require user intervention and manual tweaking to determine a suitable set of values for the desired system. This also moves in the direction of "per-joint" weighting, whereby, different joints can have different bias factors so that different joints converge on the solution at different speeds to try to achieve a desired movement.

9. Multiple Links

Although chains are a good test bed, the majority of the time a character-based rig will have more than one end-effector (unless you are simulating a snake or a worm) and so you will have multiple end-effectors and a single base-root (see Figure 14).

However, you may ask what happens when we have multiple end-effectors? Well, it depends on the circumstances of the configuration (e.g., if they are out of reach and the order they are in is updated). For example, if each end-effector can reach its goal without needing to disturb and move shared links, then each end-effector will reach its target position (as shown in Figure 14). Alternatively, if a shared link needs to be moved, the end-effector that is updated last will get priority and the other links will be pulled away. For

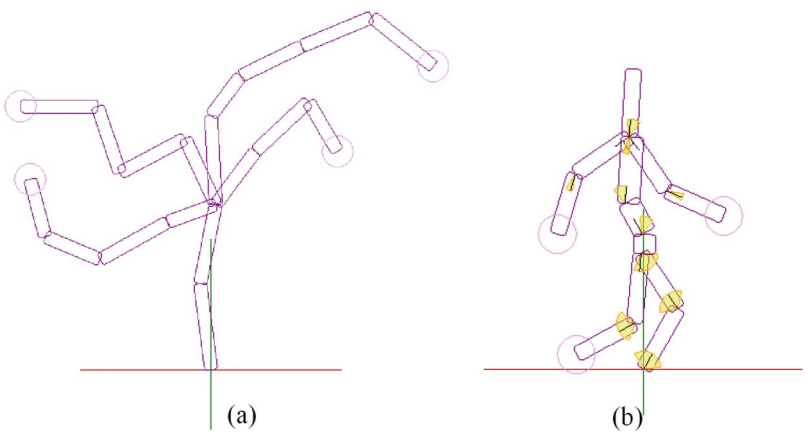


Figure 14. (a) A tree configuration of 19 links and 4 end-effectors. (b) Biped configuration (12 links) with the left foot as the root and 3 end-effectors for the hands and right foot—also showing the angular joint limits.

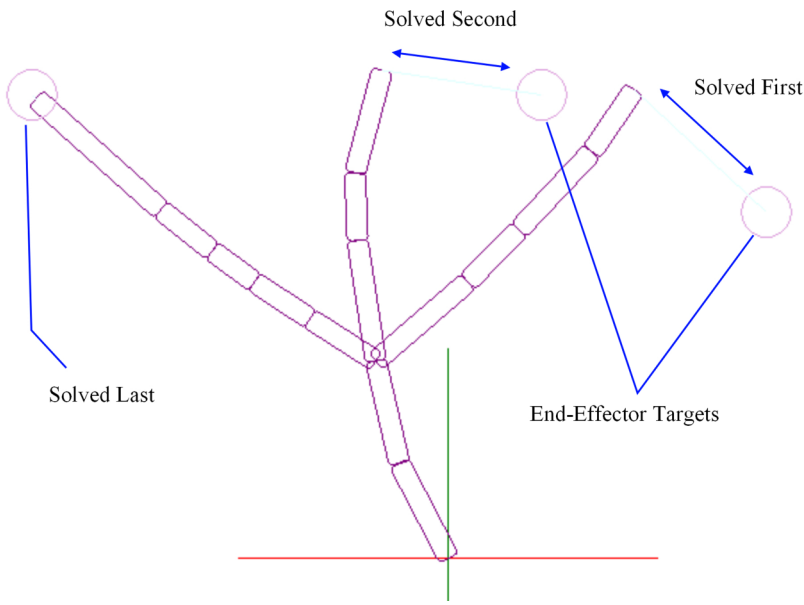


Figure 15. Multiple end-effectors (last end-effector updated using CCD is primary).

example, in Figure 15, we have three links and each is updated one after the other, and we can see how the last end-effector is the one that is primary.

9.1. Ordered Priority

Having the last updated link take priority can be a desirable effect, because the order in which we solve each end-effector target problem is the order of priority. Therefore, we have an ordered priority tree. For example, the primary problem will be solved; however, if the secondary problem can be solved it will be, or a best-guess closest approximation to solving the secondary goal will be made, while the primary problem remains solved.

9.2. Equal Priority

We can make multiple links have equal priority when their goals conflict by weighting "shared" links between the end-effectors. For example, we can identify which links are shared between multiple end-effectors, then include a

weighting factor so that their angular change is shared among the numerous end-effectors. As a side note, it should be mentioned that weighted joint constraints can also be used to control the importance of joint movement when solving the IK problem. However, in practice, constraints are often used with priority (e.g., to ensure that stance poses are enforced at all times).

10. Retargeting Character Motions with IK

IK is an excellent method for retargeting character motions to achieve specific goals. For example, preventing feet from slipping and have hands and arms links move so that they reach and touch their desired targets. Furthermore, IK can modify MOCAP data to accommodate different character skeleton types; however, not too different (see Figure 16), otherwise the positions of the end-effectors can be greatly mismatched and result in strange, unnatural, and unfeasible poses.

We can make motions more casual and less stretched and uncomfortable by incorporating reach limits for each joint by defining *reach windows*. The reach window is a predefined region or area for each joint that they cannot go beyond, and thus controls how stretched a chain of links may appear, because as they reach their reach window, they pass their movement along to the next attached joint.

Interpolating changes in joint orientation using quaternion spherical linear interpolation (SLERP) enables us to generate consistently smooth character motions. However, angles should not exceed π radians, and in practice should be as small as possible, because angles greater than π are indistinct. Hence,

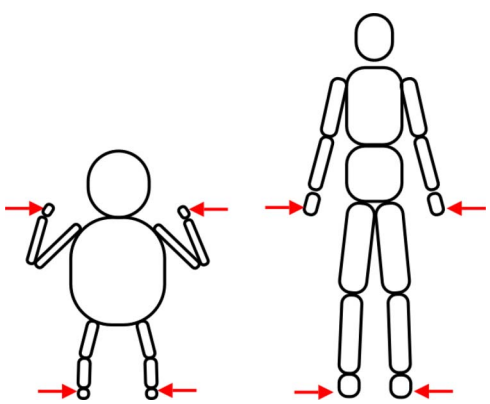


Figure 16. End-effectors position constraints applied to different skeleton types.

during each iteration, you should ensure that large angular displacements are clamped to permissible values in order to guarantee that the IK solver remains stable. Hence, to reiterate, to ensure motions are always smooth and constant, it is preferable to clamp and interpolate the angular displacements between updates. Note, for small angular changes it is easier and more efficient to use a normalized linear interpolation (NLERP), which is applicable to 2D and individual Euler angles.

11. Comfort Factors

An interesting and novel technique for producing character poses that are more natural looking and casual by means of the CCD IK technique was presented by Mahmudi and Kallmann [Mahmudi and Kallmann 11]. Although the originally proposed technique was used as a branch deformation technique, it also helps to produce poses that are less rigid and more comfortable looking. For example, when a character reaches to pick up an object, the whole arm will move, even if the object can be reached without needing to move the upper arm. The approach is very simple and elegant and works by adding tolerance regions to each link's goal. As each link is iteratively rotated toward its goal, if it comes within its defined tolerance of the target, it stops and moves onto the next link (see Figure 17).

An alternative technique that can be used to produce less abrupt joint changes and more smooth and casual poses for character movement is to use under-damped angle scaling, as discussed earlier. Whereby, each joint

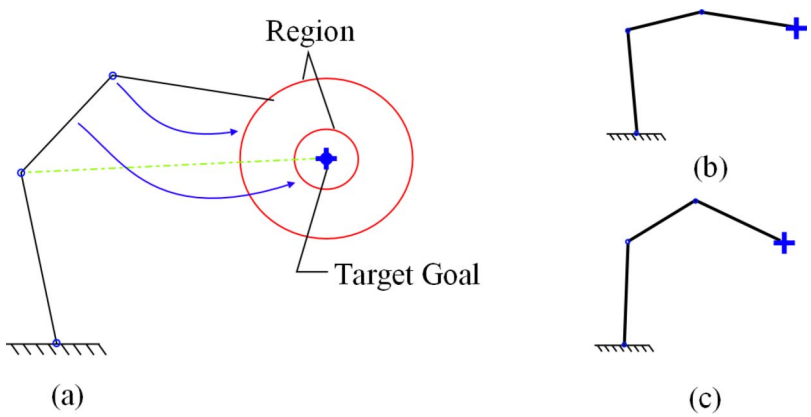


Figure 17. Reach regions so poses look less casual and comfortable (a) shows the starting pose and joint regions, (b) result without regions, and (c) result with regions.

moves only a small amount toward the goal and distributes the movement across multiple links. However, this can result in more iterations and increased computational cost.

12. 2D to 3D

The 2D implementation is much simpler than the 3D one, because the 2D version has a single axis and a single angle whereas the 3D version can have three primary axes (x, y, and z) around which we can rotate (i.e., three angles). This can make the 3D version vastly more complex to solve and can introduce singularities and result in unnatural and uncomfortable poses.

Whereas in 2D it is possible to store each link's local angle and length and up-date the hierarchy each frame, for 3D this is inadequate. Naïvely, you could use Euler angles to represent the three axis orientations (x, y and z); however, this method can suffer from gimbals lock and ambiguity. Alternatively, the popular approach for representing each link's 3D orientation in the hierarchy is with matrices (i.e., Denavit–Hartenberg [Denavit and Hartenberg 55] convention), quaternions [Shoemake 85] or dual-quaternions [Kenwright 12a]. These methods are straightforward to work with and are combined through concatenation (i.e., multiplication). However, quaternions and dual-quaternions are more efficient and are less ambiguous, can be interpolated, easy to invert, and correct for numerical drift, Furthermore, it is simple to take the axis-angle calculation from Equation (1) and convert it to a quaternion, matrix, or dual-quaternion representation, then apply it to the current link.

The same 2D technique (i.e., axis and angle) can be used in 3D, as shown in Figure 18 and Figure 19.

12.1. Local and World Coordinates

The axis-angle calculation from Equation (1) can be converted to a quaternion. However, the axis-angle quaternion is in world space and must be converted to local space. This is crucial for 3D (but unnecessary for 2D), otherwise links will oscillate, sporadically rotate, and do strange irregular movements that ultimately prevent the system from converging.

$$\begin{aligned} q_w &= \cos\left(\frac{\theta}{2}\right) \\ q_x = n_x \sin\left(\frac{\theta}{2}\right) \quad q_y &= n_y \sin\left(\frac{\theta}{2}\right) \quad q_z = n_z \sin\left(\frac{\theta}{2}\right), \end{aligned} \tag{2}$$

where q is a quaternion, θ is the angle, and $\mathbf{n}$ is the axis of rotation.

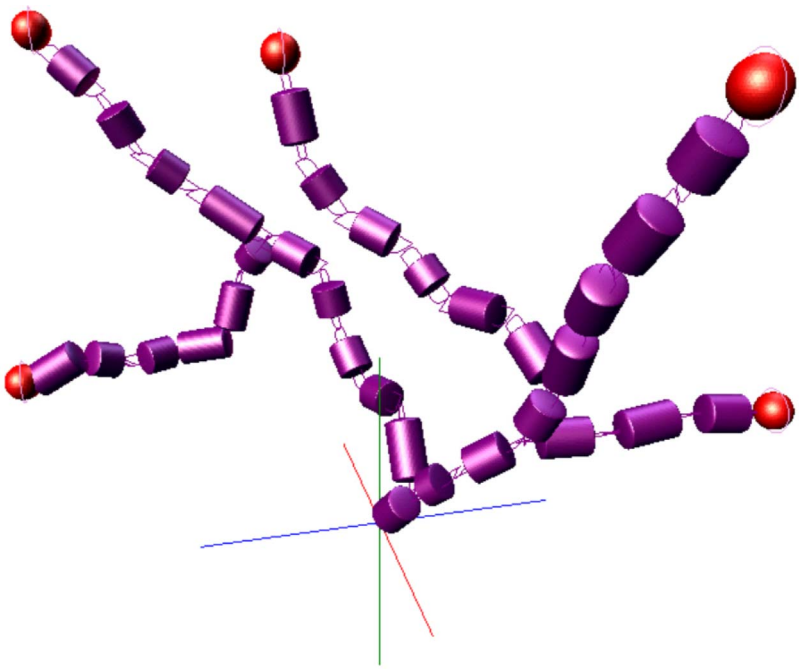


Figure 18. Tree configuration with 32 links (peak of 60 iterations at the start and 10 iterations on average for interactive movement).

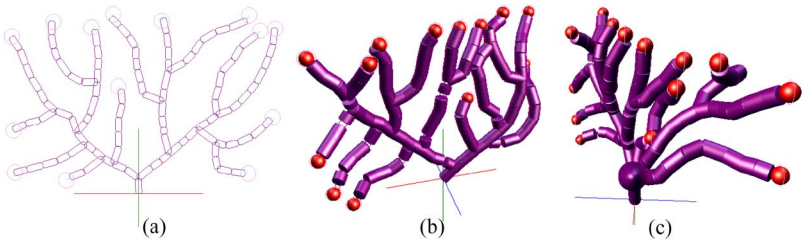


Figure 19. Many links (i.e., 100 links) connected in a tree-like configuration with 14 end-effectors moving in 3D (with a peak of 100 iterations initially, as a result of the end-effectors jumping at the start of the simulation and an average of 10 iterations for small rotations from then on). (a) Initial pose (b)–(c) end-effectors gradually rotated around the trunk.

It is very straightforward to convert the axis-angle representation of Equation (1) to its quaternion form as shown in Equation (2). Then we can store each links local and world orientation as a quaternion, which makes it extremely easy to convert from world to local coordinates (and vice versa),

because the inverse of a unit-quaternion is the conjugate, which is an algorithmically simple and computationally fast operation, and is accomplished by negating the vector component part (i.e., q_x, q_y, q_z) of the quaternion. Hence, to calculate the new local orientation for the link we use Equation (3). Additionally, because there is a limited amount of accuracy in floating-point operations that can result in numerical drift and approximation errors, we can reduce this by renormalizing the quaternion at key points (i.e., easier to correct drift and renormalize a quaternion compared with a pure matrix method).

$$q_{\text{local}} = q^*_{\text{parent}} \left(q_{\text{increment}} q_{\text{world}} \right), \tag{3}$$

where q_{local} and q_{world} are the local and world orientation of the link, $q^*{}_{\text{parent}}$ is the conjugate of the parent link orientation, and $q_{\text{increment}}$ is the corrective rotation using Equation (1) and Equation (2).

13. Articulated Characters

The body of an articulated character is divided into numerous interconnected links with multiple end-effectors (e.g., hands, head, and feet). The links (i.e., bones) are connected by numerous angular joint types (e.g., hinges and ball joints) that we can represent using multiple concatenated single degree of freedom (DOF) angular joints with zero length (e.g., see Figure 20). We can create a single DOF angular joint by modifying the basic revolute (i.e., ball joint) equation for CCD by projecting the axis of rotation onto the plane so that we obtain a single axis-of-rotation error.

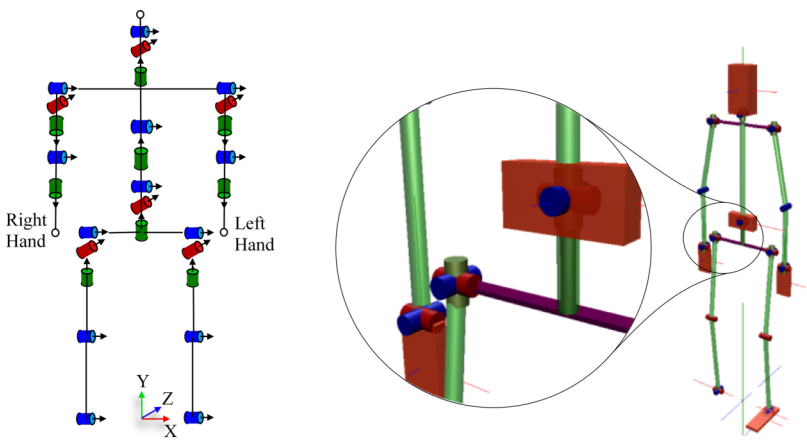


Figure 20. Representing an articulated biped character by cascading multiple single degrees of freedom (DOF) joints.

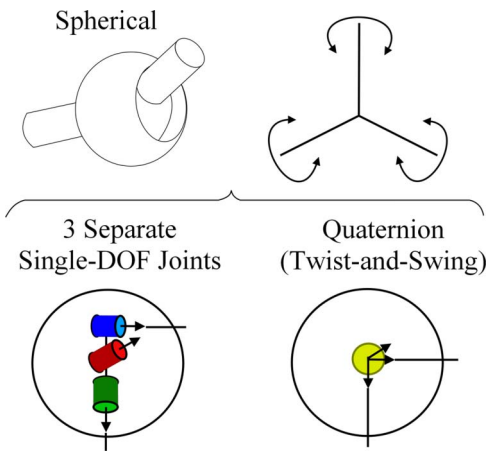


Figure 21. The ball joint, with multiple DOE joints. We can either decompose the joint into separate multiple single DOF parts (i.e., three separately connected single DOF joints with their own limits), or as a single quaternion, which we decompose into its swing-and-twist components to enforce joint limits.

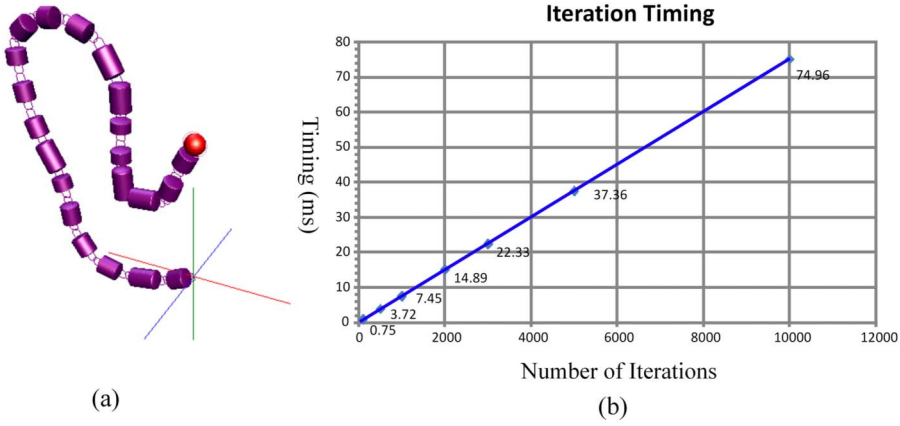


Figure 22. Timing information for a 20-link chain in 3D with a single end-effector. With each iteration, we update the full hierarchy and do a top-down analysis of the error of each link angle.

We can reduce the problem of working in a highly discontinuous and nonlinear workspace (i.e., angular limits with multiple solutions) for the IK solver by using a library of prestored lookup character poses. Therefore, the IK solver can search the repertoire of actions to find the best matching pose as an initial starting approximation for the IK solver. This approach also allows the

final motion to possess natural and lifelike properties, which can be influenced by the animation data and hence an artist (i.e., controllable attributes). For example, how a character would get up after a fall, or place his/her arms when climbing a wall. Furthermore, for a more advanced implementation, the IK solver in conjunction with the animation library can generate torques to control a physically accurate rigid body representation to produce more dynamic, responsive, and interactive characters [Van Welbergen and Ruttkay 09].

The biped model shown in Figure 20 has the base-root for the IK solver at the foot. We swap the base-root from foot to foot during walking (i.e., support foot is the base-root) and the pelvis is made the root for any other cases (for example, jumping or falling) when the body is not in contact with the surroundings. However, to accomplish this, we need to be able to reassemble the hierarchy from a different point while maintaining the current pose. We accomplish this by starting at the new base-root joint, reiterating along the hierarchy, and recalculating each joint's orientation so that it corresponds with the old base-root world coordinates. A note to the wise, when re-rooting the base, the local transform for each joint (i.e., local root of each bone) needs recalculating, and can affect branching and the hierarchy structure.

14. Quaternion Twist-and-Swing Angular Limits

For simple joints (e.g., 1-DOF hinge) we can easily enforce angular limits by means of clamping angles. However, for complex joints that contain multiple degrees of freedom (i.e., 3-DOF) we can decompose the quaternion into a twist-and-swing component to ensure that angular limits are kept (e.g., see Figure 21). For example, in Figure 20, we connected complex joints, such as the shoulder, by means of multiple single-DOF joints; however, we can form similar joints, such as a ball-and-socket joint, and ensure that the angular limits are enforced by means of a quaternion twist-and-swing decomposition. The twist-and-swing allows us to define and enforce joint limits intuitively. For example, the twist is around the z-axis while the swing is around the xy-plane. We can decompose a quaternion orientation into its twist-and-swing components shown in Equation (4). Note, the twist-and-swing limit is in world space but can easily be converted to local space (i.e., joint space).

$$\begin{aligned} q_{\text{twist } z} &= \left(\frac{q_w}{\sqrt{q_w^2 + q_z^2}}, 0, 0, \frac{q_z}{\sqrt{q_w^2 + q_z^2}} \right) \\ q_{\text{swing } xy} &= \left(\sqrt{q_w^2 + q_z^2}, 0, \frac{q_w q_x - q_y q_z}{\sqrt{q_w^2 + q_z^2}}, \frac{q_w q_y - q_x q_z}{\sqrt{q_w^2 + q_z^2}} \right), \\ q &= q_{\text{swingxy}} q_{\text{twistz}} \end{aligned} \tag{4}$$

where q is the original quaternion representing the rotation; q_w, q_x, q_y, and q_z are the components. We can multiply the separate twist-and-swing quaternion components together to construct the final quaternion. If we do not change the separate twist-and-swing components (e.g., clamping), then multiplying them together gives us the original quaternion. (Note, the algebraic proof of twist-and-swing is given in the appendix.)

15. Coding Reliability Remarks (Fast vs. Robust)

The reader should be aware of *not-a-number* (NAN) problems during implementation caused by impossible numerical calculations. For example, dividing a number by zero or normalizing a zero-length vector. In the majority of cases, if you perform an invalid calculation that results in a NAN, then the simulation will proceed as normal and the error will propagate until the whole system falls down, because any operation (i.e., addition, multiplication, and so on) performed using a NAN results in a NAN answer. It can happen at the most unlikely places, and if you are not vigilant and careful it can be almost impossible to identify the cause of the problem.

The best practices are to incorporate as many sanity checks and asserts within the code to trigger halts or log problems as they occur so they can be identified and fixed easily and quickly.

Typical problems to keep an eye out for:

- A cross product normalized without checking if the two vectors were perpendicular
- Cross product with almost perpendicular vectors (very noisy jittery result)
- Miniscule oscillations around the target resulting from numerical inaccuracies caused by rounding approximations
- Trigonometry (e.g., boundary checks for trig functions)
- Numerical rounding (e.g., 0.2 becomes 0 when converting a float to integer)

For example, a hidden trigonometry bug that can cause serious ramifications if left unmanaged is the calculation of the popular acos. The problem arises from numerical errors as floats and rounding approximations that can result in slight differences (i.e., trying to calculate acos (1.000001)). Because the acos of a number greater than 1 (or less than -1) will result in a NAN, everything will explode if the result is allowed to be used in other calculations and propagate throughout the system.

Some programmers will try to avoid the additional sanity checks will assert logic (i.e., if statements) within time-critical loops to try to achieve maximum

speed; however, it is no use having a fantastically fast implementation that crashes 5% of the time. It is worth the extra effort of implementing a test bed to try out your CCD IK implementation and try to catch and identify any anomalies or bugs before you integrate the system into a larger project. For example, interpolate the end-effectors through numerous trajectories and visually watch that the solution consistently converges without jittering and exploding. A bit of extra time testing and adding sanity checks can save hours or days of painstaking extra debugging trying to track down erroneous and difficult-to-repeat bugs.

16. Performance

The CCD IK method is an iterative method that gradually converges on a solution, and hence, its performance cost revolves around the cost of each iteration and how many iterations are necessary for the IK system to converge. However, it is good to know that the computational cost per iteration is minimalistic, requiring only a dot and cross product. Furthermore, we are able to exploit coherency, because small changes between frames require fewer iterations to find a solution.

For real-time systems, speed is always a crucial consideration and so you should avoid computationally expensive and complex calculations (e.g., such as sin, cos, sqrt) within the main iteration loop because these operations will be performed many times. In retrospect, for a large implementation (such as crowds of characters, each possessing IK), we can expect the internal iteration loop to run hundreds or thousands of times per frame. Furthermore, with any simulation or code, the secret to identifying bottlenecks or problem areas is with profiling. This will show you why and where the slowest part of the algorithm is and how you can begin to make it run faster. In addition, for interactive character systems, it may not be necessary to run the IK solution every frame because the results from each IK solution are interpolation using SLERP or NLERP for angles with small changes (i.e., in 2D or individual Euler angles in 3D).

The CCD method is capable of solving numerous diverse skeleton types with multiple end-effectors in real time reliably and is even computationally fast enough to simulate large crowds of characters in real time (e.g., see Figure 23).

16.1. Number of Iterations

The number of iterations needed for the CCD IK method to converge on an answer depends upon how large a change has occurred since the previous

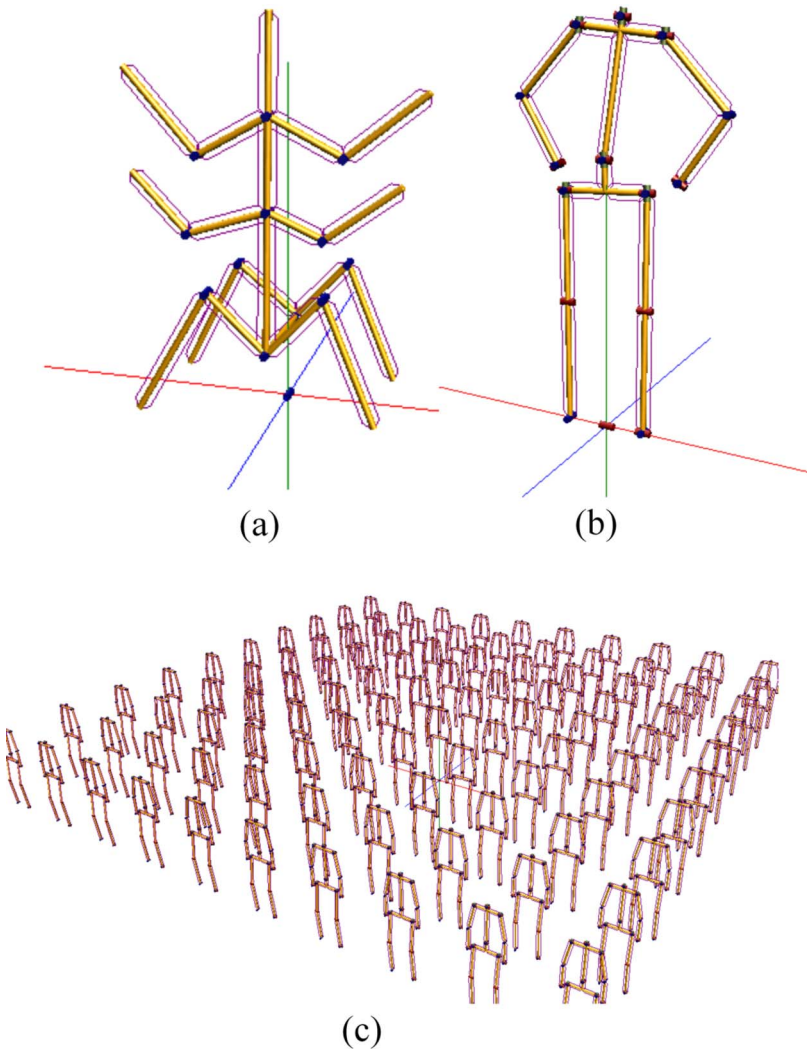


Figure 23. (a) Multileg multiarm skeleton decomposition with 20 links, (c) biped skeleton with 11 links and 30 DOF, and (c) crowd of 100 skeleton bipeds.

solution. Hence, for small changes between frames there are fewer iterations and no update overhead on sublinks that have had no change in end-effector position.

Infinite loops can occur if tolerances are made too large and links are allowed to oscillate around the solution without converging on an answer. There is also the problem of numerical inaccuracies and rounding approximations that

must be accounted for (e.g., almost perpendicular vectors, and renormalizing quaternions between frames for large hierarchies).

To give a numerical association of the computational cost of iteratively updating an articulated CCD IK chain, we did a plot of time against the number of iterations for a 20-link chain shown in Figure 22. We performed the simulation in C# with Windows 7 $\times$ 64 OS using an Intel i7-3.4 Ghz and 16 GB RAM machine in Visual Studio 2010. Furthermore, we used quaternions to represent each hierarchy link's orientations and no optimization enhancements were incorporated, so the results present a worst-case timing scenario. The results show us that we can perform thousands of iterations each frame and stay well within the boundary of achieving real-time results, whereas we found in practice that it is typical to expect a worst-case of between 100 and 300 iterations for large movements of the end-effectors.

17. Discussion

The CCD algorithm is an elegant, straightforward, and robust technique for generating IK solutions in real time. The CCD method can be used for both serial chain and multiple (i.e., tree-like) chain problems. The basic algorithm can be modified to produce character poses that are less rigid and robotic-like and more casual and human-like. Furthermore, the algorithm is computationally fast enough to be used to generate continuous motion from trajectory paths at interactive frame rates.

We have shown the reader how the CCD algorithm can be modified to incorporate additional features to make it a suitable method for different situations (e.g., character poses, articulated chains, trees and snakes). In addition, whereas CCD is a computationally fast algorithm, it is flexible enough to trade speed for accuracy in certain circumstances (e.g., updating different parts of the hierarchy and tolerance limits).

Advantages:

- Simple to implement
- Computationally fast
- Stable around singular configurations
- Can be used with other methods to produce a fast, robust hybrid solution
- Low memory overhead

Disadvantages:

- Difficult to create smooth motions (need to solve and interpolate, or use smaller steps but slower)

- Need to clamp deltas to prevent erratic jumps
- Difficult to support nongeometric constraints (e.g., position of the total center of mass)

18. Further Reading

It is recommended that the reader take a hands-on approach to help him or her solidify understanding and enable appreciation of the elegance and simplicity that the CCD algorithm uses to solve a complicated, ambiguous problem. Although it is hoped that this article has opened the reader's eyes to a number of appealing and novel ideas regarding the CCD algorithm and is practicality for real-time character systems, there are still a number of interesting and challenging areas for the reader to pursue if he/she so desires. For example, although the CCD algorithm is computationally fast, the reader can investigate further software and hardware tricks to try to push the algorithm to its limits and gain the maximum possible amount of speed. This can be through profiling or modification of the algorithm (e.g., fast sin/cos, stack push/pop instead of recursive calls).

Furthermore, we have not touched on the area of parallel processing of the CCD algorithm for multichain IK problems. A prepass phase could identify individual (i.e., not connected) chains that can be solved separately on different threads. Alternatively, the same configuration can be solved multiple times on different threads with diverse starting approximations simultaneously to ensure a greater chance of finding a solution; IK angular limits can make the problem highly nonlinear because of dead regions and can require the IK solver to try to solve the problem by using different starting approximations.

Acknowledgments. We are indebted to the reviewers for taking the time and for providing invaluable comments and suggestions to help to improve the quality of this paper.

Appendix A

A.1. Proof of Twist-and-Swing Decomposition Formula

We show through quaternion algebraic mathematics the proof for Equation (4) and how a 3D unit-quaternion can be decomposed into two parts: the twist-and-swing components, which are valuable for ensuring that angular limits are enforced.

We start with a unit-quaternion rotation shown in Equation (5).

$$\mathbf{q} = (q_w, q_x, q_y, q_z), \tag{5}$$

where $\mathbf{q}$ is the vector component, q_w, q_x, q_y and q_z the scalar component. We can calculate a quaternion from an axis-angle using Equation 6.

$$\begin{aligned} q_w &= c_{xyz} = \cos\left(\frac{\theta}{2}\right) \\ q_x &= s_x = \hat{v}_x \sin\left(\frac{\theta}{2}\right) \\ q_y &= s_y = \hat{v}_y \sin\left(\frac{\theta}{2}\right) \\ q_z &= s_z = \hat{v}_z \sin\left(\frac{\theta}{2}\right), \end{aligned} \tag{6}$$

where $\vec{v}$ is a unit-vector representing the axis of rotation, and θ is the angle of rotation.

Hence, we can say that because the twist is only around the z-axis, we can deduce that the xy-axis components will be zero and will give us Equation (7):

$$\mathrm{q}_{\mathrm{twist}} = \mathrm{q}_x = (c_z, 0, 0, s_z) \tag{7}$$

Furthermore, we can also deduce that the swing x-axis component will be zero in the resulting quaternion, as shown in Equation (8):

$$\mathrm{q}_{\mathrm{swing}} = \mathrm{q}_{yz} = (c_{zy}, s_x, s_y, 0), \tag{8}$$

where c and s represent the scalar cos and sin component of the half angles (i.e., see Equation (6)).

A unit-quaternion must obey Equation (9):

$$q_w^2 + q_x^2 + q_y^2 + q_z^2 = 1. \tag{9}$$

Hence, from Equation (7) and Equation (8), we can derive Equation (10):

$$\begin{aligned} c_z^2 + s_z^2 &= 1 \\ c_{xy}^2 + s_x^2 + s_y^2 &= 1. \end{aligned} \tag{10}$$

Subsequently, if we multiply the individual twist-and-swing quaternions together, we can reconstruct the original quaternion as shown in Equation (11).

$$\begin{aligned} \mathrm{q}_{xyz} &= \mathrm{q}_{xy}\mathrm{q}_z \\ &= (c_{xy}, s_x, s_y, 0)(c_z, 0, 0, s_z) \\ &= ((c_z c_{xy}), (c_z s_x + s_z s_y), (c_z s_y - s_z s_x), (s_z c_{xy})). \end{aligned} \tag{11}$$

Hence, from Equation (7) and knowing the vector sum of the two nonzero components from Equation (11) sums up to one, we can derive c_{xy}, as shown in Equation (12):

$$\begin{aligned} \mathrm{q}_{\mathrm{twistz}} &= \mathrm{q}_w^2 + \mathrm{q}_z^2 \\ &= (c_z c_{xy})^2 + (s_z c_{xy})^2 \\ &= c_{xy}^2 (c_z^2 + s_z^2) \qquad (\text{knowing, } \cos^2 + \sin^2 = 1) \\ &= {c_{xy}}_s{}^2. \end{aligned} \tag{12}$$

Therefore, we have Equation (13):

$$c_{xy} = \sqrt{q_w^2 + q_z^2}. \tag{13}$$

We can multiply Equation (11) by the inverse of Equation (13) to remove the quaternion swing component and leave the quaternion twist part, shown in Equation (14):

$$\begin{aligned} q_{\mathrm{twistz}} &= q_z \\ &= (c_z, 0, 0, s_z) \\ &= ((c_z c_{xy}), 0, 0, (s_z c_{xy})) \frac{1}{c_{xy}} \\ &= (q_w, 0, 0, q_z) \frac{1}{\sqrt{q_w^2 + q_z^2}}. \end{aligned} \tag{14}$$

We extract the swing component by multiplying the quaternion by the inverted (conjugated) twist quaternion, shown in Equation (15):

$$\begin{aligned} q_{\text{swing}xy} &= q_{xyz} q^*_{\text{twist}} \\ &= (q_w, q_x, q_y, q_z)\,(q_w, 0, 0, -q_z)\,\frac{1}{\sqrt{q_w^2 + q_z^2}} \\ &= (c_{xy}, s_x, s_y, 0) \\ &= \left(\left(q_w^2 + q_z^2\right), (q_w q_y - q_x q_z), (q_w q_z + q_x q_y), 0\right) \frac{1}{\sqrt{q_w^2 + q_z^2}}. \end{aligned} \tag{15}$$

Hence, Equation (14) and Equation (15) sum up the algebraic proof.

A.2. Joint Types

The equations for a CCD revolute and prismatic joint are shown in Figure 24.

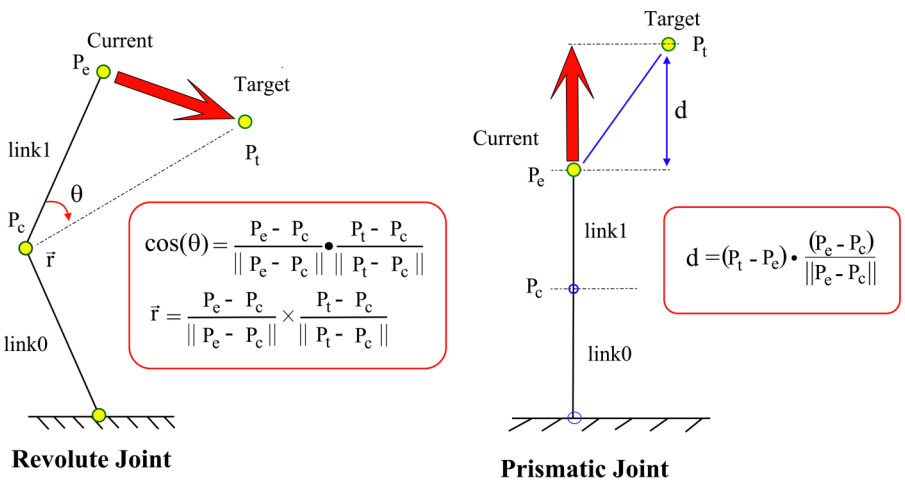


Figure 24. The basic equations for both a prismatic and revolute joint for CCD IK system.

References

[Aristidou and Lasenby 11] Andreas Aristidou and Joan Lasenby. "FABRIK: A Fast, Iterative Solver for the Inverse Kinematics Problem." *Graphical Models* 73:5 (2011), 243–260.

[Canutescu and Roland 03] Adrian A Canutescu and Jr L Dunbrack Roland. "Cyclic Coordinate Descent: A Robotics Algorithm for Protein Loop Closure." *Protein Science* 12:5 (2003), 963–972.

[Denavit and Hartenberg 55] J Denavit and R. S. Hartenberg. "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices." *Journal of Applied Mechanics* 22:June (1955), 215–221.

[Dowsland 95] K.A. Dowsland. *Simulated Annealing. In Modern Heuristic Techniques for Combinatorial Problems*. New York: McGraw-Hill, 1995.

[Fêdor 03] M Fêdor. "Application of Inverse Kinematics for Skeleton Manipulation in Real-Time." *Proc. 19th Spring Conference on Computer Graphics* (2003), 203–212.

[Kenwright 12a] Ben Kenwright. "A Beginners Guide to Dual-Quaternions: What They Are, How They Work, and How to Use Them for 3D Character Hierarchies." *International Conference on Computer Graphics, Visualization and Computer Vision* 20:June (2012), 1–13.

[Kenwright 12b] Ben Kenwright. "Real-Time Character Inverse Kinematics Using the Gauss-Seidel Iterative Approximation Method." *International Conference on Creative Content Technologies* 4:July (2012), 63–68.

[Kulpa and Multon 05] Richard Kulpa and Franck Multon. "Fast Inverse Kinematics and Kinetics Solver for Human-like Figures." *IEEE Humanoid Robots* December (2005), 38–43.

[Lander 98] J Lander. "Making Kine More Flexible." *Game Developer Magazine* November (1998), 15–22.

[Mahmudi and Kallmann 11] Mentar Mahmudi and Marcelo Kallmann. "Feature-Based Locomotion with Inverse Branch Kinematics." *Motion in Games* (2011), 39–50.

[Mukundan 09] R. Mukundan. "A Robust Inverse Kinematics Algorithm for Animating a Joint Chain." *International Journal of Computer Applications in Technology* 34:4 (2009), 303.

[Ren et al. 10] JingLi Ren, ZhubBin Zheng, and ZhongMing Jiao. "Simulation of Virtual Human Running Based on Inverse Kinematics." *International Conference on Educational Technology and Computer (ICETC)* 2 (2010), 360–363.

[Shin et al. 01] H.J. Shin, Jehee Lee, S.Y. Shin, and Michael Gleicher. "Computer Puppetry: An Importance-Based Approach." *ACM Transactions on Graphics (TOG)*, 20:2 (2001), 67–94.

[Shoemake 85] K Shoemake. "Animating Rotation with Quaternion Curves." *Proc. ACM SIGGRAPH'85 Computer Graphics*, 19:3 (1985), 245–254.

[Wang and Chen 91] Tommy Li-Chun Wang and Chih Cheng Chen. A Combined Optimization Method for Solving the Inverse Kinematics Problems of Mechanical Manipulators. *IEEE Robotics and Automation* 7:4 (1991), 489–499.

[Van Welbergen and Ruttkay 09] Herwin Van Welbergen and Zsofia M. Ruttkay. Real-Time Animation Using a Mix of Physical Simulation and Kinematics. *Journal of Graphics, GPU, and Game Tools* 14:4 (2009), 1–20.

[Welman 93] Chris Welman. "Inverse Kinematics and Geometric Constraints for Articulated Figure Manipulation." PhD thesis, Master's Dissertation, Simon Fraser University, Department of Computer Science, 1993.

[William 88] W Hagger William. *Applied Numerical Linear Algebra.* Englewood Cliffs, NJ: Prentice Hall, 1988.

Web Information:

http://www.ncl.ac.uk/computing/people/student/b.kenwright

Ben Kenwright, School of Computing Science, Newcastle University, Claremont Tower, Claremont Road, Newcastle-Upon-Tyne, NE17RU, UK (b.kenwright@ncl.ac.uk)

Received August 21, 2010; accepted in revised form June 28, 2013.