Workshop Series: Inverse Kinematics

Benjamin Kenwright[1]*

Abstract
This practical focuses on the implementation of simple Inverse Kinematic (IK) chains. Inverse kinematics is a challenging and valuable multi-discipline technique (e.g., animation and robotics). The issue inverse kinematics attempts to resolve is to find a set of joint configurations of an articulated structure based upon the desirable end-effector location. The student should use this practical to implement a custom real-time interactive inverse kinematic system using either the cyclic coordinate descent algorithm (CCD) or the Jacobian pseudo-inverse method.

Keywords
Inverse Kinematics (IK), Cyclic Coordinate Descent (CCD), Jacobian, Iterative, Video Games, Real-Time, End-Effector, Linked-Chains, C++, Psuedo Inverse

[1] *Workshop Series (www.xbdev.net) - Benjamin Kenwright*

Contents

Introduction 1
1 Overview 1
2 Analytical Inverse Kinematics 2
3 Cyclic Coordinate Descent (CCD) 2
4 Iterative Jacobian Pseudo Inverse 3
5 Summary 4
6 Exercises 4
Acknowledgements 4
A Appendix 4
A.1 Cyclic Coordinate Decent (CCD) 4
A.2 Jacobian Matrix 5

Introduction

Inverse Kinematics (IK) The topic of this practical is the implementation of a real-time inverse kinematic simulation (i.e., a linked chain of rigid body objects). The user should be able to interact with the simulation while it is running (e.g., through the mouse or keyboard) to control the inverse kinematic target end-effector. The inverse kinematic chain can be implemented using either the cyclic coordinate descent (CCD) algorithm or the Jacobian pseudo-inverse matrix method.

Tasks

1. Visually display interconnected chain of 3D rigid bodies (i.e., base and end-effector)
2. Implement an uncomplicated single-joint IK system using an analytical method (i.e., axis-angle) which follows the mouse cursor around the screen (Section 2)
3. Implement a linked-chain of interconnected limbs using an iterative inverse kinematic technique (e.g., CCD IK with details given in Section 3)
4. User input (e.g., mouse or keyboard) to control and move the end-effect target
5. Areas to explore, include, altering the number of links (e.g., 10 to 1000). Adding physical constraints (i.e., limit the joint angles +/- specified amount). Modify the code to support multiple end-effectors (e.g., tree-like structure)

1. Overview

Principles and Concepts The application of kinematic algorithms for animation is used to control articulated postures based on a simple definition of joint angles and limb lengths. These structures may take practically any form from a humanoid bipedal character to just about anything that can be imagined using a hierarchy.

Hierarchy Structures described using a hierarchical form are defined using a parent-child system similar to that of tree structure. In the case of computer characters, each rigid limb of the structure is a child node in the tree whose parent node provides a reference point from which it is described. The parents are themselves child nodes of limbs above in the hierarchy and this recursive relationship continues up to a root node (i.e., the bottom of the tree). The parent of the root node is effectively taken as the *global frame of reference* and defined as such.

End-Effectors At the top end of the tree are leaf nodes which are children that have no descendants of their own. An end-effector in terms of kinematic chains is any node within the hierarchy that an animator wishes to directly position, for example, to interact with the environment. End-effectors are commonly the leaf nodes of an articulated structure, such

as, the feet and hands of an animated character since they generally interact with the world.

Graphics Inverse kinematics is a programmatic solution for controlling the animation of rigid models. When artists generate an animated model for a virtual environment, such as a game, the animations created by the artists won't necessarily work for every position in the world. For example, if an artist were to create an animation of a character pressing a button, the animation would only work so long as the button was always at the same position relative to the model. If the button were to be moved up or down the pre-made animation would have no way to account for that. With inverse kinematics, we can reconfigure the interconnected set of links (e.g., animate the model) so the end-effector (e.g., a hand) can be placed anywhere.

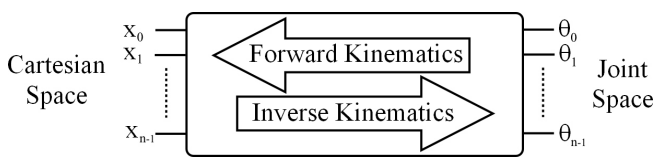

Figure 1. Forward & Inverse Kinematics - Illustrating the relationship between forward and inverse kinematics parameters

Inverse Kinematic (IK) Methods

1. Analytical (e.g., Geometric Analysis) - Section 2
2. Heuristic (e.g., CCD) - Section 3
3. Solver (e.g., Jacobian & Gradient-Based Search) - Section 4

2. Analytical Inverse Kinematics

The geometric/analytical algorithms tend to be very quick because they reduce the IK problem to a mathematical equation that need only be evaluated in a single step to produce a result (e.g., see Figure 2). The limitations of this class of solver becomes apparent in the case of large chains. In such cases, the task of reducing the problem to a single-step mathematical equation is impractical. Therefore geometric/analytical techniques tend to be less useful in the field of character animation.

3. Cyclic Coordinate Descent (CCD)

IK solvers that are based on CCD use an iterative approach that takes multiple steps towards a solution (see Figure 4). CCD works by analysing each joint one-by-one in a progressive refinement philosophy. Starts with the last joint in the chain (e.g., a hand for a character) and tries to rotate it to orientate it toward the target. The steps that the method takes are formed heuristically, therefore each step can be performed relatively quickly. An example of a possible heuristic would be to minimise the angle between pairs of vectors created when

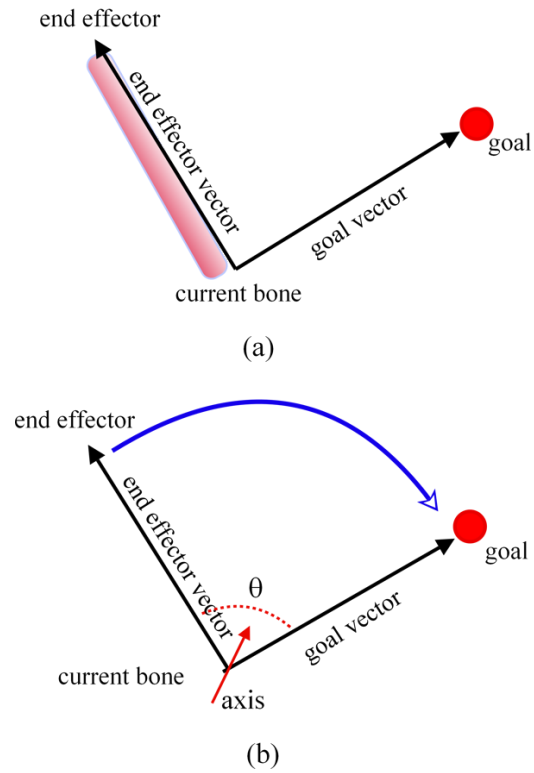

Figure 2. Uncomplicated Analytical Inverse Kinematic Example - Single limb and joint angle (e.g., can be solved using the dot and cross product).

projecting lines through the current node and end-effector and current node and desired location. However, because the iterative step is heuristically driven, accuracy is normally the price paid for speed. Another issue with this technique is that one joint angle is updated at a time as opposed to the complete hierarchical structure. This has the undesirable and unrealistic result of earlier joints moving much more than later limbs in the IK chain.

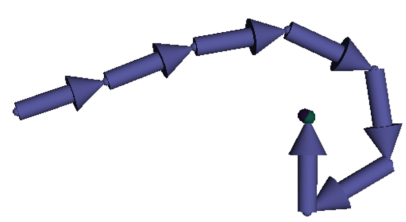

Figure 3. Animated Inverse Kinematic Simulation - Real-Time Interactive IK Simulation Screen Capture (i.e., following mouse around screen).

Implementation The Cyclic Coordinate Descent (CCD) algorithm is an iterative IK solution. The basic idea of the CCD algorithm is to loop over each bone in the IK chain and rotate

it such that the end effector (which is typically the last bone in the chain) will move as close as it can to the final position.

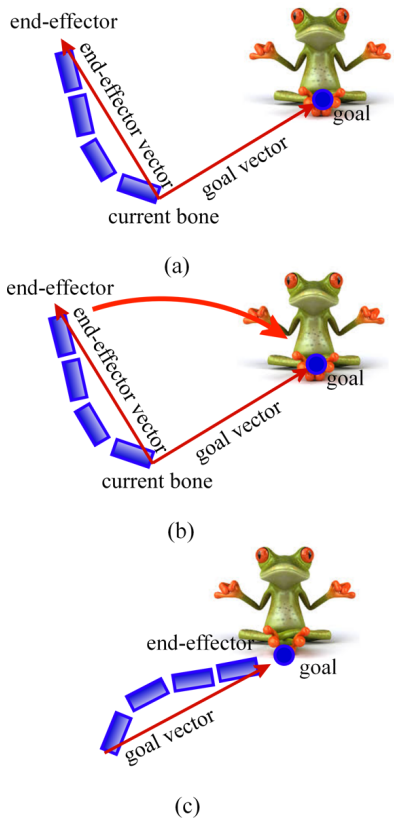

Figure 4. Cyclic Coordinate Descent (CCD) - Simple illustration showing the principle of the CCD technique. (a) Compute the vector direction of the current bone to the goal position and the current bone to the end effector; (b) Compute the rotation matrix that will rotate the end effector vector onto the goal vector. To do this, we first compute a rotation axis by taking the cross product of the end effector vector and the goal vector. Next, using the dot product, we calculate the angle between the two vectors. We then compute our rotation matrix using our new axis and angle; (c) Apply the rotation to the current bone.

4. Iterative Jacobian Pseudo Inverse

Due to their scalability, numerical techniques often form part of an inverse kinematics solver. However, because of their iterative nature, such methods can be slow. So far research into the field of kinematics has failed to find a general non-numerical solution to the problem. Many researchers have proposed hybrid techniques yet these still rely on a numerical aspect. It is therefore important to find ways of using numerical techniques as efficiently as possible. In this paper we take a look at the Jacobian-based IK solver and techniques that allow this method to be used as an efficient real-time IK solver.

Algorithm 1 Algorithm for the CCD system - where we start at the final bone in the system and work backwards.

while While distance from effector to target > threshold and numloops<max **do**
  Take current bone
  Build vector V1 from bone pivot to effector
  Build vector V2 from bone pivot to target
  Get the angle between V1 and V2
  Get the rotation direction
  Apply a differential rotation to the current bone
  If at the base node then the new current bone is the last bone in the chain
  Else the new current bone is the previous one in the chain
  EndIf
end while

The end effector velocities are related to the joint velocities of the robot through the Jacobian matrix as follows:

$$\dot{e} = J\dot{\theta} \tag{1}$$

where J is the Jacobian representing the partial derivatives for the change in end-effectors locations with change in joint angles (e.g., see Appendix for details).

Typically, the system of joints consists of a non-square Jacobian, hence, we need to perform the 'pseudo inverse' operation to yield the joint velocities and solve for the angular displacement:

$$\begin{aligned} J^{+} &= J^{T}(JJ^{T})^{-1} \\ \dot{\theta} &= J^{+}\dot{e} \end{aligned} \tag{2}$$

where we are able to connect $\dot{\theta}$ for change in joint angles with change in end-effector location $\dot{e}$. This method sets the angle values to the pseudo inverse of the Jacobian. It tries to find a matrix which effectively inverts a 'non-square' matrix. It has singularity issues which tend to the fact that certain directions are not reachable. The problem is that the method first loops through all angles and then needs to compute and store the Jacobian, perform a pseudo inversion, calculate the changes in the angle, and last apply the changes (see Appendix for details on formulating the Jacobian).

Algorithm 2 Algorithm Jacobian System

while e is too far from t **do**
  Compute $J(e,\theta)$ for the current pose
  Compute J^{-1} i.e., invert the Jacobian matrix
  $\Delta e = \beta(t-e)$ - pick approximate step to take
  $\Delta\theta = J^{-1}\,\Delta e$ - compute change in joint DOFs
  $\theta_{current} = \theta_{previous} + \Delta\theta$ - apply change to DOFs
  Compute new e vector - apply forward kinematics to see where we ended up
end while

5. Summary

We have introduced the basic inverse kinematic concepts, and had a quick refresher course of the kinematics required to develop the algorithms. Your module work should now be structured in such a way that the inverse kinematic simulation is separate from the renderer update, and so that the implementation which you will develop can easily interface with the rest of your system (i.e., modular component programming).

6. Exercises

This practical only gives a brief taste of inverse kinematics. As an exercise for the student to help enhance their understanding:

Intermediate

- Implement a hierarchy with multiple interconnected limbs
- Add physical constraints (i.e., impose joint limits) - e.g., an arm is able to make unrealistic rotations and even pass through itself. Adding physical constraints will improve the realism of the animation. Examples of ways to do this include clamping the rotation angles or even implementing self collision.
- Optimize for speed. The practical inverse kinematic algorithms are written with readability in mind and does little to optimize for speed
- Bias the priority of the joints - so different joints converge on the target at different rates

Advanced

- Try and load in an animated skeleton (e.g., human body or arm) and connect the limbs together using a kinematic hierarchy - then try and control the character's motion using inverse kinematics (e.g., reaching for an object or walking)

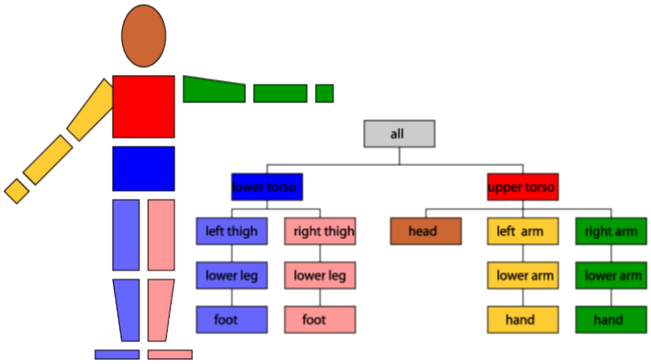

Figure 5. Hierarchy - Creating an appropriate hierarchy is particularly important when it comes to animating an object or character.

Acknowledgements

We would like to thank all the students for taking time out of their busy schedules to provide valuable and constructive feedback to make this practical more concise, informative, and correct. However, we would be pleased to hear your views on the following:

- Is the practical clear to follow?
- Are the examples and tasks achievable?
- Do you understand the objects?
- Did we missed anything?
- Any surprises?

The practicals provide a basic introduction for getting started with physics-based animation effects. So if you can provide any advice, tips, or hints during from your own exploration of simulation development, that you think would be indispensable for a student's learning and understanding, please don't hesitate to contact us so that we can make amendments and incorporate them into future practicals.

Recommended Reading

Computer Animation: Algorithms & Techniques, Rick Parent, Publisher: Morgan Kaufmann, ISBN: 978-0124158429

A Mathematical Introduction to Robotic Manipulation, Richard M. Murray, Zexiang Li, S. Shankar Sastry, Publisher: CRC Press, ISBN: 978-0849379819

Code Complete: A Practical Handbook of Software Construction, Steve McConnell, ISBN: 978-0735619678

Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, ISBN: 978-0132350884

Making kine more flexible, Lander, Jeff and CONTENT, Game Developer Magazine, 1, 15-22, 1998

Game Inverse Kinematics: A Practical Introduction (2nd Edition) Kenwright. ISBN: 979-8670628204
Kinematics and Dynamics Paperback. Kenwright. ISBN: 978-1539595496
Game Collision Detection: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1511964104
Game C++ Programming: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1516838165
Computational Game Dynamics: Principles and Practice (Paperback). Kenwright. ISBN: 978-1501018398
Game Physics: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1471033971
Game Animation Techniques: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1523210688

1. Appendix

A.1 Cyclic Coordinate Decent (CCD)

Listing 1. A minimilistic C++ implementation of the Cyclic Coordinate Descent Inverse Kinematic algorithm for a singly linked chain.

```
class Link
{
public:
  Link(Vector3& axis, float angle)
  {
    m_axis  = axis;
    m_angle = angle;
  }
  // Note− we could store this as a quaternion
  Vector3   m_axis;   // local axis
  float   m_angle;  // local angle

  ..
  // other helper variables to represent the
  // structure (e.g., length, quaternion, matrix)
}; // End Link



// Build or set of links (i.e., axis and angle)
void Setup()
{
  for(int i=0; i<numLinks; ++i)
  {
    links.push_back( new Link( Vector3(0,0,1), 0 ) );
  }
}




// This is the heart of the program − this is what performs
// all the IK work − if we have a problem − it will
// typically be located within this function
void UpdateIK()
{
  // 1. Get and draw the current 'end' effector
  // position we desire based on the mouse position..

  // Current 'end' effector position
  Vector3 target = GetMousePosition2Dto3D();
  DrawSphere( target, 0.16f, 0.1f,0.5f,0.4f );


  // Either work from the end towards the base or from
  // the base towards the leaf
  //for (int i=links.size()−1; i>=0; −−i)
  for (int i=0; i<(int)links.size(); i++)
  {
    // 2. UpdateHierarchy();

    // Update the hiearchy − however, we can
    // optimize this to only update the section of the
    // hierarchy or update the target based on the
    // modified link transformation
    UpdateHierarchy();

    3. Peform iterative IK link−by−link..

    // We iteratively update the 'target' based on
    // the new position of the limb − so we don't have
    // to keep updating the hierarchy − performance
    // improvement
    Reach( i, target );
  }

  // 4. Draw Hiearchy
  for (int i=0; i<(int)links.size(); ++i)
  {
    DrawSphere( GetTranslation(links[i]−>m_base), 0.1f, ↩
        0.5f,0.5f,0.9f );

    Vector3 base = GetTranslation(links[i]−>m_base);
    Vector3 end  = Transform( links[i]−>m_base, Vector3(↩
        linkLength,0,0) );

    DrawArrow(base, end, 0.2f);
  }

}// End UpdateIK(..)
```

A.2 Jacobian Matrix

The Jacobian J is a matrix that represents the change in joint angles $\Delta\theta$ to the displacement of end-effectors Δe. Each frame we calculate the Jacobian matrix from the current angles and end-effectors. We assume a right-handed coordinate system. To illustrate how we calculate the Jacobian for an articulated system, we consider the simple example shown in Figure 6.

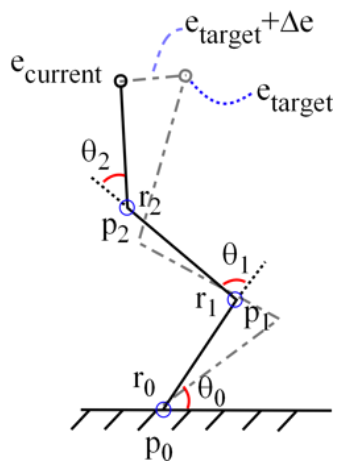

Figure 6. Jacobian IK Example - Relationship between multiple joint angles and end-effectors.

The example demonstrates how we decompose the problem and represent it as a matrix for a sole linked chain with a single three degree of freedom (DOF) end-effector. We then extend this method to multiple linked-chains with multiple end-effectors (each with six DOF) to represent a structured hierarchy.

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \dots \\ \theta_n \end{bmatrix} \tag{3}$$

$$e = \begin{bmatrix} e_x \\ e_y \\ e_z \end{bmatrix} \tag{4}$$

where θ is the rotation of joint i relative to joint $i-1$, and e for the end-effectors global position. The angles for each joint and the error for each end-effector are represented by matrices.

From these matrices, we can determine that the end-effectors, and the joint angles are related. This leads to the forward kinematics definition, defined as:

$$e = f(\theta) \tag{5}$$

We can differentiate the kinematic equation for the relationship between end-effectors and angles. This relationship between change in angles and change in end-effectors location is represented by the Jacobian matrix.

$$\dot{e} = J(\dot{\theta}) \tag{6}$$

The Jacobian J is the partial derivatives for the change in end-effectors locations by change in joint angles.

$$J = \frac{\delta e}{\delta \theta} \tag{7}$$

If we can re-arrange the kinematic problem:

$$\theta = f^{-1}(e) \tag{8}$$

We can conclude a similar relationship for the Jacobian:

$$\dot{\theta} = f^{-1}(\dot{e}) \tag{9}$$

For small changes, we can approximate the differentials by their equivalent deltas:

$$\Delta e = e_{target} - e_{current} \tag{10}$$

For these small changes, we can then use the Jacobian to represent an approximate relationship between the changes of the end-effectors with the changes of the joint angles.

$$\Delta \theta = J^{-1} \Delta e \tag{11}$$

We can substitute the result back in:

$$\theta_{current} = \theta_{previous} + \Delta \theta \tag{12}$$

The practical method of calculating J in code is used:

$$\frac{\delta e}{\delta \theta_i} = r_j \times (e_{target} - p_j) \tag{13}$$

where r_j is the axis of rotation for link j, e_{target} is the end-effectors target position, p_j is end position of link j.

For example, calculating the Jacobian for Figure 7 gives:

$$J = \begin{bmatrix} \frac{\delta e}{\delta \theta_0} \\ \frac{\delta e}{\delta \theta_1} \\ \frac{\delta e}{\delta \theta_2} \end{bmatrix} = \begin{bmatrix} r_0 \times (e_{current} - p_0) \\ r_1 \times (e_{current} - p_1) \\ r_2 \times (e_{current} - p_2) \end{bmatrix} \tag{14}$$

and

$$e = e_{current} - e_{target} \tag{15}$$

The Jacobian matrix is calculated for the system so that we can calculate the inverse and hence the solution. One the Jacobian system has been defined, we iteratively solve for the change in angle using an approximate technique, such as, Pseudo-Inverse Transpose.

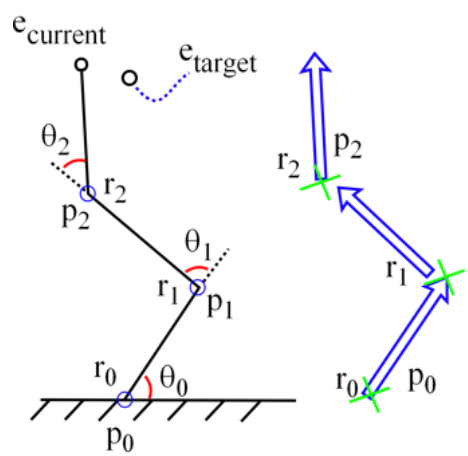

Figure 7. Jacobian - Iteratively calculating the Jacobian on a frame by frame basis.