Inverse Kinematics
A 3D game engine extension

Bart Verdonck

Mentor: Koen Samyn
Academic year 2015-2016
# Table of contents

1. Introduction .................................................................................................................................................. 3
2. Different algorithms ........................................................................................................................................ 3
   2.1 Jacobian algorithms ................................................................................................................................... 3
   2.2 Cyclic Coordinate Descent (CCD) ............................................................................................................. 4
   2.3 Sequential Monte Carlo Method (SMCM) ................................................................................................. 4
   2.4 Style-based Inverse Kinematics ................................................................................................................. 5
   2.5 The triangulation algorithm ....................................................................................................................... 5
   2.6 Follow-the-Leader (FTL) ........................................................................................................................ 6
   2.7 Forward And Backward Reaching Inverse Kinematics (FABRIK) ......................................................... 6
3. Performance comparisons ............................................................................................................................. 7
   3.1 Iterations required to cover a certain distance ............................................................................................ 7
   3.2 Time required for the algorithm to come to a solution within a given tolerance ........................................ 8
   3.3 Performance Reference Table .................................................................................................................. 8
4. FABRIK in-depth ............................................................................................................................................ 8
   4.1 Core of the algorithm ............................................................................................................................... 9
   4.2 Multiple end effectors ............................................................................................................................ 10
   4.3 Joint limits ............................................................................................................................................... 10
5. Designing a Forward and Inverse Kinematics Framework ......................................................................... 10
   5.1 Bone-chain class ..................................................................................................................................... 11
   5.2 Bone-object class ................................................................................................................................... 11
   5.3 Algorithm manager .................................................................................................................................. 11
   5.4 Abstract Algorithm ................................................................................................................................ 12
   5.5 The algorithm class .................................................................................................................................. 12
   5.6 Framework diagram ............................................................................................................................... 12
6. Overlord-Engine: Inverse Kinematics Breakdown ..................................................................................... 13
   6.1 RotateTowards(XMFLOAT3 targetPos) ...................................................................................................... 13
   6.2 RotateBoneLocal(XMFLOAT3 axis, float angle) ......................................................................................... 13
   6.3 Storing the Bone Hierarchy .................................................................................................................... 13
   6.4 Generate and draw Bézier curves ............................................................................................................ 14
   6.5 Example screenshots .............................................................................................................................. 15
7. Conclusion .................................................................................................................................................... 17
8. Acknowledgments ......................................................................................................................................... 17
9. Sources ......................................................................................................................................................... 17
1 Introduction
Inverse Kinematics is the idea of calculating a set of bone-joint configurations with the goal being that the end of the bone chain (the effector) reaches a certain point in space. This is often used in animations since it greatly reduces the effort one has to do to animate a long structure of bones. However for games, this has only been really used for baked animations since the cost is quite high to calculate all this complex math at runtime.

My research was focused on finding an algorithm that could offer a low computational cost while still producing realistic poses for the bone-chain. The other part of my research was designing a framework that could implement this inside an existing C++ game engine.

2 Different algorithms
In the field of Inverse Kinematics, many different approaches that have been tried. Each of them have their own advantages and disadvantages. I have researched the algorithms that are viable for real time applications and looked into their possible uses for games.

2.1 Jacobian algorithms
The Jacobian approach is to calculate a linear approximation using the matrices of each bone-joint. These include: Jacobian Transpose, Damped Least Squares (DLS), Damped Least Squares with Singular Value Decomposition (SVD-DLS) and Selectively Damped Least Squares (SDLS).

The algorithm basically divides the displacement towards the target in multiple smaller displacements. For each of the smaller displacements all the angles are calculated using matrix manipulations. This method is a very popular approach to the Inverse Kinematics problem, since it produces very smooth transitions and looks very natural.

The problem with this solution is that the computational cost is too high for complex bone-structures in games. The complex matrix calculations grow exponentially with every bone that is added. This is not a problem for animation software as it has a lot of resources to calculate the animations. But in games the resources available to calculate Inverse Kinematics are very limited since the rest of the game has to run as well at the same time.
Since real-time Kinematics are often not a part of the gameplay itself, they cannot have a big impact on the general performance of the game.

2.2 Cyclic Coordinate Descent (CCD)

This method is a very popular approach to Inverse Kinematics that has often been used in the game industry. CCD is a heuristic iterative algorithm with a low computational cost per iteration and solves the problem without having to deal with matrices. This results in a decent performance boost and makes this method useable for real time applications such as games.

The algorithm will iterate over all the bones starting from the effector (the head of the end bone) and rotate each bone until the effector is on a line between the bone position and the target. When a cycle has been completed it starts again from the top. This algorithm is simple, performant and therefore good for games.

However the problem is that it can suffer from unrealistic movements and often produces weird looking end poses. It has also been designed for linear bone chains and it is nearly impossible to handle multiple ends and multiple target positions.

There has been much effort into this algorithm to make it better by adding constrains and adapting the algorithm to allow bone stretching but none of these solve the inherit problems of this method.

2.3 Sequential Monte Carlo Method (SMCM)

This method is based on the importance sampling principle and will thus solve the problem using a statistical approach and direct kinematics. This avoids the matrix inversions of the Jacobian methods. Because of this, it improves the performance of the algorithm.

SMCM requires that all the constrains and parameters of the articulated bone-structure are defined. Using a likelihood calculation it gives an evaluation of how the current pose can be altered to reach the destination. For every presented solution it checks if all the constraints are being respected.
If this is not the case, it will determinate which bones will solve the problem while minimizing the impact on the rest of the chain.

The main disadvantage of this method is that it uses approximations when calculating the likelihood of each pose which results in weird looking transitions between different poses. The constrains are required to be constant which is not good for game mechanics. And while its performance is better than the Jacobian, SMCM uses a statistical approach which can be performance heavy if the bones have only a few or no constrains. This is the case because the possible solutions decrease with each added constrain.

2.4 Style-based Inverse Kinematics

This method works in a completely different way than any of the previous methods. Using predefined poses this algorithm will check which predetermined pose is closest to the required pose, and then it will modify the predefined pose to get to the target. This method is excellent for real time inverse kinematics of predefined structures.

The main disadvantage of this method is of course that for every type of skeleton you will need an array of predefined structures. And when dealing with games, inverse kinematics will have to be applied to a large amount of different chains thus making this method too expensive on memory and development. Another disadvantage is that the generated poses are very dependent on the predefined poses. If these are suboptimal then all the solutions will be suboptimal.

2.5 The triangulation algorithm
This algorithm uses the cosine rule to calculate the angles for each bone-joint. Starting at the root and moving outward towards the end effector. This method can most of the time be completed in one iteration making it a highly performant Inverse Kinematics solver.

While it has a very low computational cost, the movement is most of the time visually unnatural. The last bones are often just a straight line, while in realistic movements most of the rotations are at the end. It also doesn’t support multiple ends and multiple targets, and it is also very hard to implement in type of game that is not in 2D.

2.6 Follow-the-Leader (FTL)

This method is a non-iterative technique and is specially designed for rope. While this method is excellent for these applications, it’s less than ideal for other applications.

The main disadvantage is that this algorithm doesn’t work with constraints, neither can it work with multiple ends and targets. This is a viable algorithm for displaying ropes/wires very realistically, but it doesn’t have many other uses sadly.

2.7 Forward And Backward Reaching Inverse Kinematics (FABRIK)

The FABRIK algorithm is a relatively new algorithm for Inverse Kinematics (2010), and is currently one of the most balanced algorithms for real time applications in terms of performance/quality. It combines the idea behind the CCD algorithm with the implementation and speed of FTL which avoids complex matrix calculations while keeping realistic movements.

This algorithm uses n-dimensional points and lines instead of angle rotations. This greatly improves the performance but it requires the rotation function to take care of the constrains.

However since this method iterates in both directions of the chain this can be achieved without much trouble.
The real strength of this algorithm comes from the fact that it changes each joint one at a time. Because of this double iteration the original solution gets evaluated and modified within the second iteration.

3 Performance comparisons

Since performance in games is equally important as quality, I looked into performance graphs and found some interesting facts about the performance that each algorithm claimed and delivered.

These following tests are performed on a linear bone-chain with 10 unconstrained joints. With the damping constant of the Jacobian methods set to 1.1

Only the Inverse Kinematic algorithms that have enough adaptability and performance for games have been tested.

All the following test data and graphs are provided by:

FABRIK: A fast, iterative solver for the Inverse Kinematics problem
Written by:
- Dr Joan Lasenby (University Senior Lecturer Engineering at University of Cambridge)
- Dr Andreas Aristidou (Post-Doc Researcher at University of Cyprus)

http://www.academia.edu/9165835/FABRIK_A_fast_iterative_solver_for_the_Inverse_Kinematics_problem

3.1 Iterations required to cover a certain distance

*note: different scaling for number of iterations between both graphs

In this test the target is located a certain distance away from the effector. These graphs show the remaining distance between the effector and the target at the end of each iteration. The first iterations will cover most of the distance and the closer the effector gets to its target, the smaller the distance each iteration will cover.

We can clearly see from the graph that FABRIK has the fewest iteration of all the Inverse Kinematic algorithms.

An interesting note is that because the Jacobian algorithms uses approximations of small steps, their performance is several times more expensive than both CCD and FABRIK which both avoid matrix manipulations.
3.2 Time required for the algorithm to come to a solution within a given tolerance

![Graph](image)

*note: time scale is different for both graphs

In this test the target is placed in a certain reachable destination, the tolerance is adjusted and we can see the time it takes to get within a certain range of the given target. Here we can see similar results as with the previous test.

Since CCD and FABRIK rotate the joints one by one, they can get in a more precise position much faster, since they can just alter the last few joints and stop.

Jacobian algorithms calculate all the angles each time which results in a massive performance disadvantage when precision is required, this disadvantage grows even larger when used on longer chains.

3.3 Performance Reference Table

<table>
<thead>
<tr>
<th>Reachable Target</th>
<th>Number of iterations</th>
<th>Matlab exec. time (sec)</th>
<th>Time per iteration (in mm)</th>
<th>Iterations per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABRIK</td>
<td>15,463</td>
<td>0.01328</td>
<td>0.86</td>
<td>1194</td>
</tr>
<tr>
<td>CCD</td>
<td>26,306</td>
<td>0.12536</td>
<td>4.64</td>
<td>213</td>
</tr>
<tr>
<td>Jacobian Transpose</td>
<td>1511.190</td>
<td>12.98947</td>
<td>9.90</td>
<td>101</td>
</tr>
<tr>
<td>Jacobian DLS</td>
<td>998.584</td>
<td>19.48951</td>
<td>10.49</td>
<td>95</td>
</tr>
<tr>
<td>Jacobian SVD-DLS</td>
<td>898.727</td>
<td>9.23612</td>
<td>11.50</td>
<td>87</td>
</tr>
<tr>
<td>FTL</td>
<td>21.125</td>
<td>0.02065</td>
<td>0.97</td>
<td>1033</td>
</tr>
<tr>
<td>Triangulation</td>
<td>1.000</td>
<td>0.05747</td>
<td>57.47</td>
<td>21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unreachable Target</th>
<th>Number of iterations</th>
<th>Matlab exec. time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABRIK</td>
<td>67564</td>
<td>0.06207</td>
</tr>
<tr>
<td>CCD</td>
<td>390,125</td>
<td>3.92809</td>
</tr>
<tr>
<td>Jacobian Transpose</td>
<td>6549,000</td>
<td>33.96475</td>
</tr>
<tr>
<td>Jacobian DLS</td>
<td>2881,667</td>
<td>14.67918</td>
</tr>
<tr>
<td>Jacobian SVD-DLS</td>
<td>2988,452</td>
<td>13.97591</td>
</tr>
<tr>
<td>FTL</td>
<td>22,325</td>
<td>0.02526</td>
</tr>
<tr>
<td>Triangulation</td>
<td>1.000</td>
<td>0.06893</td>
</tr>
</tbody>
</table>

4 FABRIK in-depth

The performance of the Forward And Backward Reaching Inverse Kinematics seemed very promising to me, so I researched how this algorithm works. I found out how FABRIK is able to achieve this performance while other similar algorithms apparently perform much worse.

The main advantage of FABRIK is that the algorithm itself doesn’t work with joint angles at all. It works with points and vectors defined within n-dimensions. The strength of this approach is that the performance in 2D and 3D is nearly identical. If you look at how the Jacobian algorithms works, you can see just why the difference in speed is so large. The Jacobian algorithms calculate all the angles with matrices, which in 3D applications with multiple bones can become very costly.
4.1 Core of the algorithm

The basic concept of FABRIK is quite simple actually as this image shows. A chain of bones and joints is presented with a target position. The first step of this algorithm is to rotate the last bone to the target. Then it will snap the head of this bone to the target position (steps 1 – 3). All the other bones will be rotated towards their child bone instead of the target. Repeat these steps for all the bones in the chain.

After this is completed the bone-chain has reached the target but the root has been moved from its original position. (step 4) So the algorithm will now do the entire process again but in reverse order starting with the root, going up to the effector. (step 5 – 6)

This algorithm can be easily used for any amount of dimensions. Ideally the algorithm calls the RotateTowards() function of a bone which handles the rotations and constraints. Because of this, the only thing that has to be modified is the RotateTowards() function, while still keeping the algorithm the same.

Because of the double iteration, each movement gets redone in the second iteration which makes the movement seem a lot more natural (Each movement is an average of the ideal position and the current position). It also enables to alter the middle of a bone-chain and get a realistic response from the middle to the root and the middle to the end. As is the case with the FTL algorithm.
4.2 Multiple end effectors

The algorithm is also able to calculate multiple end effectors. It notices when a bone has multiple child bones attached. If this is the case it will make sure that all the child bones have been iterated before moving further.

This will give an array of desired position of the sub-base, which will then be positioned in the middle of all these suggested positions. Afterwards all the children will be reiterated and snapped to this new base position.

This ensures that all the child bone structures will calculate their new positions. These new positions are based on the new position of the sub-base or root. This sub-base or root position is the average of all the ideal positions that each child wants.

Image source: http://www.academia.edu/9165835/FABRIK_A_fast_iterative_solver_for_the_Inverse_Kinematics_problem

4.3 Joint limits

Since most bone structures have joint limits it is necessary that this algorithm can deal with limits in an efficient manner. Because FABRIK is iterative, the joint limits can be forced on each joint without creating extra logic. Every time a joint is being altered, the joint itself is responsible to make sure that it does not go out of bounds. Because the iteration goes over bones one by one in two directions, this will work for any number of constraints with any number of bones.

The limitation itself can be done in two different ways.

One way is to alter the RotateTowards() function to make sure that this will never rotate a bone outside of the bounds despite its target position.

This new position of the constrained bone is the new target of the next bone in the iteration. Therefore it will make sure that all the constraints are being respected at all times.

The second approach for limitations is to check if the target position is out of bounds before rotating the bone. When this is the case, it will calculate a new target position that is as close as possible to the original position while still being in bounds. This method requires to map the constrains onto positions. Using these positions you can determine if the target is reachable. When the target is not reachable, the function will translate the target to a point that is reachable.

Both approaches have their advantages and disadvantages but the performance is about the same since comparing angles or positions depends largely on the amount of dimensions. However, when implementing joint limit functionality in a framework, I assume it can be easier to limit the RotateTowards() function since this is more user-friendly. The user will most likely be adding rotation constrains using angles instead of positions and this will simplify the code.

5 Designing a Forward and Inverse Kinematics Framework

After researching the algorithms, I started thinking about how to efficiently add Inverse Kinematics to an already existing game engine. My design heavily relies on the DirectXMath library, however any library that offers Vector and Matrix manipulations can be used.
The main flow of the framework can be simplified to:

1. create a bone chain
2. create bone object
3. add bone object to a bone chain
4. when using forward kinematics the user can rotate bones himself, and when using Inverse Kinematics all rotations are black boxed

5.1 Bone-chain class
The basic idea is to have a bone-chain class which contains the entire bone structure. When a developer is using the framework nearly everything will be done through this class.

This class must be able to handle at least all the following functionality:

- Always force the chain to have a root-bone, ideally added/generated in the constructor
- Adding existing bones to the structure, ideally let the user decide which existing bone will be the parent of this added bone. The user should be required to give the bone a unique name.
- Get bones quickly by name (by using a C++ map or similar container)
- Call the Inverse Kinematics algorithm at every Tick() when the chain uses Inverse Kinematics.

5.2 Bone-object class
A bone-object is an individual bone which can be seen as a vector with added functionality. The object itself knows the direction it is facing, its length and is responsible for its own rotations. Meshes will also be added to the bone-object, so they will be rotated and moved along with the bone-object.

This class must have at least this functionality:

- Rotate towards a given point in the same object space. Ideally this will also take into account any constraints and will make sure that the final rotation never goes out of these constrains.
- Return the head position of the bone in world space (this makes the fabric algorithm easier to implement)
- Set rotational constraints, given in roll, pitch, yaw format since all rotations in 3D will be done with quaternions.
- A multitude of rotate functions for when the bone is part of a forward kinematics chain. (Rotate around local axis, rotate around global axis, rotate around a given point in space)
- Add a mesh to the bone.
- Optionally adding more constrain functionality such as extruding bones for pistons.
- Optionally add a debug drawer to make it easier to see the rotations, constrains, etc...

5.3 Algorithm manager
This class only needs to be implemented when the framework needs to support multiple Inverse Kinematics algorithms. The reason for this, as stated before, is that for some cases an algorithm is better fitted than another. This is class contains all the algorithms and executes the right one.

In the GameTick of the bone-chain class, the algorithm is executed. But when implementing multiple algorithms the bone-chain class calls the execute of the algorithm manager. Which then calls the right algorithm and passes all the parameters to this algorithm.
It requires only this functionality:

- Add algorithms to the collection.
- Execute the chosen algorithm. (chosen by the developer)

### 5.4 Abstract Algorithm

This class also only needs to be implemented when planning to use multiple Inverse Kinematics algorithms. This class basically has one pure virtual function which executes the algorithm. When algorithms derive from this class, it makes sure that each algorithm has been implemented.

### 5.5 The algorithm class

This class derives from Abstract Algorithm when using multiple Inverse Kinematics algorithms, and is responsible for executing the algorithm on the bone-chain. The bone-structure can be passed on as reference to the algorithm manager which is passed on by reference to this class.

This makes sure that the algorithm is able to manipulate the entire structure while making sure that it cannot alter anything else. The implementation of the Execute function will of course depend completely on the specific algorithm it is using.

### 5.6 Framework diagram
The above UML diagram is my design for the Inverse and Forward Kinematics Framework. This is still subject to change as my implementation is far from finished yet but this concept has all the required elements to achieve Inverse Kinematics.

The framework can be made in multiple ways and this diagram is only one of the many possibilities.

6 Overlord-Engine: Inverse Kinematics Breakdown

In the previous chapter, I discussed Inverse Kinematics on an abstract level, which can be applied to any programming language. But for my case I had to create the framework in C++ using the DirectX Math Library. This however, will probably be similar in any game engine since C++ is very performant and the DirectX Math Library is widely used in games and very similar to other math libraries.

Most of the framework did not raise any difficulties since it mostly uses basic features such as parent-child relationships, translating objects and rotating objects.
I will discuss some of the features that were more challenging to implement, or that required a major design decision that impacted the rest of the framework.

6.1 RotateTowards( XMVECTOR targetPos )

The hardest part however was of course in the RotateTowards() function of the bone-object. The Overlord-Engine uses quaternions to perform rotations in 3D. And since the engine did not contain a RotateTowards() function which rotates to a given point in space, I had to create this function myself.

I used documentation of Unreal Engine 4 to create the correct roll, pitch and yaw angles, but the problem was that this is done in world space. To create believable Inverse Kinematics each bone should prefer to have a rotation as close as possible to its parent’s rotation. I created this effect by converting the target position from world space to parent-bone space. This way it would use the parent’s bone Up-Axis as a world axis, and make the rotations look more natural.

After calculating the ideal rotating I still need to clamp each angle to the limits if they are set. Since each iteration will call the rotate of each bone once before its parent and once after its parent. All limits will be respected in both ways at the end of each iteration of the FABRIK algorithm.

6.2 RotateBoneLocal( XMVECTOR axis, float angle )

This function is used when Forward Kinematics are active on the BoneObject. This function seems relatively simple, but when constrains are added rotating the bone suddenly becomes a lot more complicated.

The working of the function can be simplified to:

- Rotate the bone with a constructed quaternion, generated from the parameters
- Check if the new rotation violates the rotation limits
- Calculate and apply a new quaternion that contains the negative rotation which will snap the bone back to its limits

However

6.3 Storing the Bone Hierarchy

The most important tasks of the BoneChain class is to keep track of the entire hierarchy of all the bones it contains, and making sure that the user can access any bone at any time as fast as possible while still being user-friendly.
I decided to create a small struct “BoneHierarchyObject” which contains a pointer to a BoneObject and a std::vector of pointers to all its child BoneObjects. A pointer to its parent-bone is not needed since the BoneObject already contains this pointer.

In the BoneChain class I used a std::map with as value type this struct and as key type std::string. This map uses the name which the user gives to the bone as key value, so it is very user-friendly and the inner workings of a map result is very fast searching on the key value.

By storing the BoneObjects this way I can easily give a reference to this std::map as a parameter to the Inverse Kinematics algorithm. Using this, the algorithm has direct access to all the bones and all the links that connect the bones to each other.

6.4 Generate and draw Bézier curves

Bézier curves are not really a part of Inverse Kinematics but I added this feature to the engine because it very useful to create a path which a target of a BoneChain can follow.

When implementing this feature I tried to be as user-friendly as possible, I designed it so it can be used outside Inverse Kinematics as well.

The generate function requires a std::vector of XMFLOAT3 points, these are point through which the generated bézier will pass. Other parameters are tessellation, colour, Boolean value if the ends should be connected.

First the function will generate the control points that are needed for a quadratic bézier. The first point is the hardest to calculate. First you calculate a vector that is perpendicular to the vector from point 0 to point 2 in the plane constructed by the vector from point 0 to point 1 and the vector from point 0 to point 2. Then you project point 1 perpendicular on to this new vector and that position is the first control point. From then on, the new control points can be calculated by inverting the vector that goes from the next point to the previous control point, then translating this vector to the point itself.

Afterwards I calculate the points of the actual bézier by using a quadratic bézier formula which uses the tessellation parameter provided by the user.

To connect the begin point and end point I used a cubic bézier formula since you have two control points for this piece.

At the end, this function return all the points that the bézier curve contains.

I also provided this class with a function to obtain a point on this path by providing a percentage. If the user increased this with deltaTime you receive a smooth translation along the bézier, this is extremely useful for performance testing since both the path and the speed are consistent.
6.5 Example screenshots

Fig1: An example of a simple BoneChain with a few bones and no constrains.

Fig2: An example of a simple BoneChain without constraints following an object that moves along a spline.
Fig3: An example of a BoneChain that has multiple constrains. Two bones are constrained with rotational constrains. And the middle bone has an extruding constraint.

Fig4: An example of a BoneChain using a model, the bones and debug lines are disabled so the model can be seen.
7 Conclusion
While the FABRIK algorithm doesn’t give the absolute best quality of all the algorithms, it is the best fit for performance-heavy real-time applications such as games. It can handle nearly any bone structure, it’s fast and simple. No pre-recorded data has to be loaded for it to work, and it can be expanded with multiple types of constrains.

It is the best overall algorithm for real-time calculation which is the reason I think it is best selected by default. But in specific cases some algorithms may prove to be better (FTL for rope animations, Jacobian for when quality is more important than performance) which is why a good Inverse Kinematics framework should allow multiple algorithms.

8 Acknowledgments
My mentor Samyn Koen helped me to design my version of an Inverse Kinematics Framework. He also had an influence on the direction of my research.

All the performance data and most of the information about the FABRIK algorithm came from:
FABRIK: A fast, iterative solver for the Inverse Kinematics problem.
By Andreas Aristidou and Joan Lasenby.
Departement of Engineering, University of Cambridge, Cambridge CB2 1PZ, UK
Online available from:
http://www.academia.edu/9165835/FABRIK_A_fast_iterative_solver_for_the_Inverse_Kinematics_problem

9 Sources
All of the following sources are officially published papers by universities around the world.

Jacobian methods:
Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squared methods
Written by Samuel R. Buss;
University of California

Cyclic Coordinate Descent:
Inverse kinematics and geometric constraints for articulated figure manipulation
Written by Chris Welman;
Simon Fraser University
http://graphics.ucsd.edu/courses/cse169_w04/welman.pdf

Sequential Monte Carlo Method:
Inverse Kinematics using Sequential Monte Carlo Methods
Written by Nicolas Courty, Elise Arnoud;
Université Joseph Fourir

Style-Based Inverse Kinematics:
Style-Based Inverse Kinematics
Written by Keith Grochow, Steven L. Martin, Aaron Hertzmann, Zoran Popovic;  
University of Washington, University of Toronto  


Triangulation algorithm:
Triangulation: A new algorithm for Inverse Kinematics  
Written by R Müller-Cajar and R. Mukundan;  
University of Canterbury  
http://digital.liby.waikato.ac.nz/conferences/ivcnz07/papers/ivcnz07-paper34.pdf

Follow-the-Leader:
A Follow-the-Leader Approach to Serpentine Robot Motion Planning  
Written by Howie Choset, Wade Henning;  
Carnegie Mellon University  
http://repository.cmu.edu/cgi/viewcontent.cgi?article=1759&context=robotics

Forward And Backward Reaching Inverse Kinematics (FABRIK):
FABRIK: A fast, iterative solver for the Inverse Kinematics problem  
Written by Andreas Aristidou, Joan Lasenby;  
University of Cambridge  
http://www.academia.edu/9165835/FABRIK_A_fast_iterative_solver_for_the_Inverse_Kinematics_problem