# CSC 473 - Implementing Inverse Kinematics using FABRIK

Thomas Gartside - V00944201

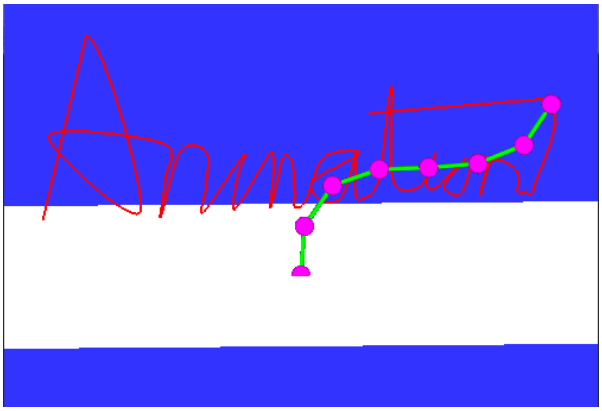

**Figure 1:** *Animating with FABRIK!*

**Abstract**

*Inverse Kinematics is important in the world of computer animation as it provides the means to calculate the configuration of a hierarchical chain when an end effector moves. An example of which being animating a character by positioning and orientating limbs with respect to the position of hands or feet. Many solvers for the Inverse Kinematics problem have been developed. This project looked to study FABRIK, (Forward and Backward Reaching Inverse Kinematics), understand and implement the base algorithm and demonstrate it's use in a variety of use cases. The model used for the project was a single chain with a fixed root, a single end effector and a variable number of joints and segments. The joints had three degrees of freedom and were configured to move in a hierarchical fashion. The algorithm implemented was based upon the paper Fabrik: A fast and iterative solver for the inverse kinematics problem by Andreas Aristidou and Joan Lasenby [AL11]. The implementation was tested using different numbers of joints and applying different styles of simulation, from spline following to dragging with a cursor. The examples show that FABRIK can successfully be implemented to move to and trace target positions in a realistic manner, performing the calculations at low computational cost.*

**CCS Concepts**

• ***Computing methodologies*** → *Inverse Kinematics;* • ***Algorithm*** → *FABRIK;*

## 1. Introduction

The goal of this project is to demonstrate an implementation of FABRIK (Forward And Backward Reaching Inverse Kinematics) in a variety of use cases. Inverse Kinematics is a way to animate articulated motion in a smooth and realistic-looking manner. It is used extensively in both the gaming and film industry as it allows an animated character to move naturally to a target point, by positioning and orientating joints and limbs with respect to the end effectors (e.g hands and feet) without the need for predefined animations. Inverse kinematics solvers can be accomplished in various ways, depending on the method used, but most methods are centered around taking a desired target location and calculating the joint angles or positions in order for the end effector of the object or character to reach the desired point. Generally, an articulated character will have multiple defined degrees of freedom and constraints for each individual joint, which can result in highly complex and expensive calculations.

This project focuses on the inverse kinematic solver known as FABRIK, which was developed by Andreas Aristidou and Joan Lasenby [AL11]. FABRIK uses a forwards and backwards iter-

ative approach which finds resulting joint positions by locating that position as a point on a line as opposed to using joint angles and complex matrix calculations. The goal of this project was to implement the FABRIK algorithm to enable a structure composed of multiple unconstrained sections and joints, each with three degrees of freedom, to respond to various input methods, such as being directed through mouse input, following a spline, or simply being issued a target end position through a console command. It is possible to apply rotational and orientational constraints to a structure using FABRIK, however for the purpose of this project, those functions were not implemented. The language chosen for this project was c++, using openGL and built upon the base code supplied to the CSC 473 course at the beginning of the spring semester of 2024.

## 2. Related Work

There are many solutions that have been developed for solving the inverse kinematics problem. Some of the most popular are: Jacobian transpose, Jacobian Pseudo inverse, Cyclic Coordinate Descent, and FABRIK. These, amongst other solutions, are discussed in a paper by Aristidou and Lasenby [AL09]

### 2.1. Jacobian Transpose

The Jacobian Transpose method for controlling Robot arms was initially proposed by [BDMS84]. This method uses the transpose of the Jacobian as opposed to the inverse as demonstrated by equation: 1

$$\Delta\theta = \alpha J^T \Delta x \tag{1}$$

Where $\alpha$ is the step size, J is the Jacobian, and $\Delta x$ is the change in location of the end effector.

The Jacobian Transpose is a simple calculation and involves no complex matrix inversions, but often needs many iterations and produces unreliable joint angles.

### 2.2. Jacobian Pseudo Inverse

The Jacobian Pseudo Inverse allows us to find the changes in $\theta$ for each joint using the equation: 2

$$\dot{\theta} = J^T (JJ^T)^{-1} \dot{x} \tag{2}$$

Because the jacobian is not always square, finding the inverse of $JJ^T$ is resolved by setting $\beta$ to equal $(JJ^T)^{-1}\dot{x}$ as shown in the following equation: 3

$$\beta = (JJ^T)^{-1} \dot{x} \tag{3}$$

Using LU decomposition allows us to find $\beta$ and consequently solve equation 4

$$\dot{\theta} = J^T \beta \tag{4}$$

The pseudo Inverse solution becomes unstable around singularities. Both Jacobian solutions have the disadvantage of high computational costs and complicated matrix manipulations.

### 2.3. Cyclic coordinate descent

The Cyclic Coordinate Descent (CCD) method of solving the inverse kinematics problem was originally proposed by Wang and Chen in 1991 [WC91]. The CCD approach starts with the last joint in the chain and iteratively works through the joints in a hierarchical chain to calculate the rotation required to reduce the distance between the end effector and target. The advantages of CCD over other IK solvers are that it is simple to implement and computationally cheap. Unfortunately, the motion can sometimes appear unnatural as the outer joints are rotated first, and it is not always smooth.

### 2.4. FABRIK

FABRIK has become a popular model for solving inverse kinematics problems because it is simple to implement and computationally fast. Many researchers have looked to extend its capabilities by intoducing methods for obstacle avoidance [TTY21], model constraints such as fixed inter-joint distance and closed loop chains [ACL16], and mobile manipulators (a robot arm mounted on a mobile platform) [SFC*20]

## 3. Overview

### 3.1. Articulated Hierarchical models

In animation, articulated hierarchical models usually consist of a root node and one or more end effectors connected by a series of nodes that act as joints and rigid segments of defined, constant lengths that connect each node to its neighbors. The root of a structure is the uppermost element of the hierarchy from which all other elements stem, and therefore if the root is moved, all other elements in the structure must move accordingly. An end effector is the opposite of a root node, as they are the final node of a branch of the hierarchy and therefore have no child nodes. In a humanoid model, the torso is most often used as the root, whereas the hands and feet would be examples of end effectors. In order to make the model move realistically, moving any given node will also move all other nodes lower in the hierarchy through space without altering their angles or translations in relation to each other, while any nodes that are higher up in the hierarchy remain unaffected.

### 3.2. The FABRIK model

While many inverse kinematics solvers use matrices and the angles of the joints to calculate the new angles of each joint in a structure, which can be computationally expensive, FABRIK uses a back-and-forth method of solving the problem using only the positions of the joints.

The FABRIK algorithm begins by getting the target location for the end effector and then sets the position of the end effector of the structure to that location. Afterwards, it draws a line to the next node, or joint, in the chain and moves that node up the line to be the same distance from the previous node that it was before the previous node was moved, thus maintaining a consistent length for the segment between the two nodes. This process repeats until every single node has been moved. Once all nodes have been repositioned, the root node will no longer be in its starting position. It is

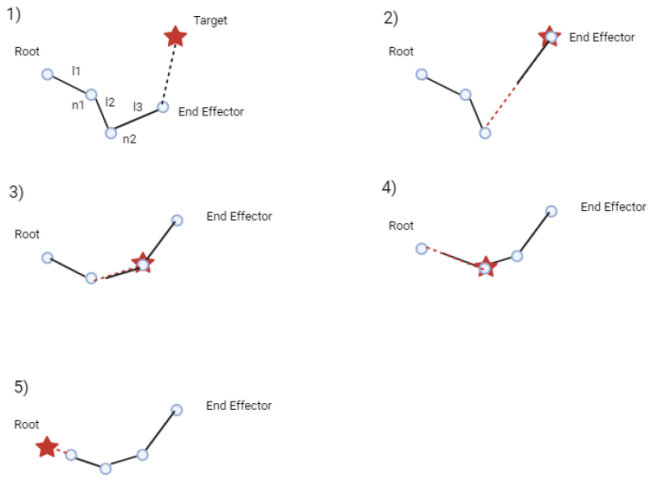


**Figure 2:** *The Forward portion of the FABRIK algorithm*

then moved back to the starting location and the whole process is repeated, but this time in the opposite direction, moving forwards along the chain towards the end effector. Once the end of the chain has been reached, the algorithm checks whether the end effector is situated at the target position, or within a defined margin of error of that position. If it is not, the algorithm is run again and the process is repeated until the end effector is within the margin of error of the starting position.

Fig.2 demonstrates the first portion of the FABRIK algorithm. Step 1) shows the initial starting position of every node and segment, as well as the desired target location of the end effector. In step 2), the end effector has been moved to the target position and a line has been drawn to the next node in the chain. In step 3), that node is then moved to be the same distance from the end effector as it was in the first step. The same process is then repeated for the next node in the chain in step 4), and finally in step 5), the root is moved up and the target position is set to be the initial starting location of the root. Once all of the steps have been completed, the process is run again in reverse, starting by moving the root to its original location, and the algorithm works its way back up the chain to the end effector.

### 3.2.1. Implementation of FABRIK written for this project

The algorithm used in this project is a recursive approach to the FABRIK algorithm created by [AL11]. It works with a single, non-branching chain consisting of any given number of joints as specified by the user, each with three degrees of freedom and connected by segments of individual, constant length. The algorithm can receive a target location through various different forms of input, ranging from the user clicking and dragging the mouse, to reading and following a spline without intervention from the user.

The algorithm used in this project starts off by calculating the distance between the root node and the given target's ($t$) location, and then checks to see whether that distance is longer than the total length of every segment of the chain combined. If the target is out of range, a for loop is run $n$ times, where $n$ is the number of nodes in the chain. In each iteration of this loop, the distance between the current point, $p_i$, and the target is calculated and stored in the variable $r_i$: 5

$$r_i = |t - p_i| \tag{5}$$

The length of the segment at index $i$ ($d_i$) is then divided by $r_i$ and stored in a new variable $l_i$: 6

$$l_i = d_i / r_i \tag{6}$$

Finally, the new location for point $p_{i+1}$ is calculated using equation 7:

$$p_{i+1} = (1 - l_i)p_i + l_i t \tag{7}$$

By the end of the loop, the chain will be positioned in a straight line pointing directly at the target location.

If the target location is within reach, then the initial position of the root node is saved and the function *doFabrik*() is called with the index of the end effector and the direction "$f$" as parameters. When *doFabrik*() is called, it first checks if the direction passed in is "$f$" or "$b$", representing the Forward and Backward portions of the algorithm respectively. If the given direction is "$f$", the algorithm checks if the current index ($i$) is that of the end effector. If it is, then the end effector's position is updated to be at the location of the target and *doFabrik*() is called with $i$-1 and the current direction as parameters. If the index is not that of the end effector, the distance between $p_{i+1}$ and $p_i$ is calculated and stored in the variable $r_i$: 8

$$r_i = |p_{i+1} - p_i| \tag{8}$$

The length of the segment at index $i$ ($d_i$) is then divided by the value in $r_i$ and stored in the variable $l_i$: 6

With the value of $l_i$ calculated the new position of node $p_i$ is calculated using equation 9:

$$p_i = (1 - l_i)p_{i+1} + l_i p_i \tag{9}$$

Lastly, a check is done to see if the index is 0 and therefore that of the root node. If it is not, *doFabrik*() is called with $i$-1 and the current direction as parameters. If it is the index of the root node, *doFabrik*() is called with the current index and the direction "$b$" as its parameters and the direction of the FABRIK algorithm is reversed.

When *doFabrik*() is called with the direction "$b$" as its direction, the function first checks if the given index is 0. If so, the position of the root node is set to be that of its original starting location. Whether the check passes or not, the function then calculates the distance between points $p_{i+1}$ and $p_i$ and stores the value in the variable $r_i$: 8

The length of the segment at the given index $i$ ($d_i$) is then divided by the value of $r_i$ and once again stored in the variable $l_i$: 6

With that completed, the new position of point $p_{i+1}$ is then calculated using equation 10:

$$p_{i+1} = (1 - l_i)p_i + l_i p_{i+1} \tag{10}$$

Finally, the function checks whether the current index is that of the node before the end effector. If it is not, *doFabrik*() is called again with *i*+1 and the current direction as parameters. If it is, that means that the end effector has been updated to its new position and a check needs to be done to see if it is within an acceptable margin of error from the target location. The distance between the target and the end effector is then calculated, and is checked to see if it is less than the specified margin of error. If it is not, *doFabrik*() is called with the index of the end effector and the direction "*f*" as parameters, meaning the direction is reversed and the entire process is run again. If the position of the end effector is within the margin of error, however, then the function issues a return statement as the structure has reached the target position, and the algorithm has reached its end.

**Algorithm 1** FABRIK algorithm

**Input:** Joint positions $p_i$ for i = 1,...,n, target position t, and distances between joints $d_i = |p_{i+1} - p_i|$ for i = 1,...,n-1.
**Output:**New joint positions $p_i$ for i = 1,...,n.

```
//distance between root and target
dist ← |t-p_i|

// Check if target is within reach
if dist >totlength then
    // Target out of reach
    for i=0,...numpoints -1 do
        // Find distance r_i between target and joint p_i
        r_i ← |t − p_i|
        l_i ← d_i/r_i
        // Get new joint positions
        p_{i+1} ← (1 − l_i)p_i + l_i t
    end for
else
    // Target is within range
    // Save the starting position of the root
    initRoot ← p_0
    doFabrik(numPints − 1,"f")
end if
```

### 3.2.2. Restricting FABRIK for rotational and Orientational constraints

In the paper introducing FABRIK [AL11], Aristidou and Lasenby describe approaches to constraining joints by applying both orientational constraints and rotational constraints. These were not implemented for this project.

Orientational constraints are achieved by defining a range of allowed bounds for the joint in question, if after running the FABRIK algorithm the target point for the joint is outside the pre-described range of values then the joint is moved to the edge of the allowed values closest to the target. Once the joint orientation has been calculated the rotational limits are applied. For example a ball and socket joint would have its orientational limits defined by an irregular cone where the elipsoid end of the cone represents the range of angles that the joint can move with respect to its degrees of freedom and the conical part of the cone represents the range of motion. If the target does not map to the conic section defined by the restrictions, then the point is moved to the nearest point on the conic section from the target. This approach is applied iteratively to every joint in the model as in the original algorithm. As with the original algorithm the segment lengths never change.

**Algorithm 2** doFabrik algorithm

```
Function{doFabrik}{i, direction}
if direction = "f" then
    // Forward Reaching
    // if i is end effector
    if i = n − 1 then
        // set end effector to target location
        p_i ← t
        doFabrik(i − 1, direction)
    else
        // Find distance r_i between new and old joint positions
        r_i ← |p_{i+1} − p_i|
        l_i ← d_i/r_i
        // Get new joint positions
        p_i ← (1 − l_i)p_{i+1} + l_i p_i
        if i=0 then
            // if i is root reverse direction
            doFabrik(i,"b")
        else
            // Go to next joint in the chain
            doFabrik(i − 1, direction)
        end if
    end if
else if direction = "b" then
    // Backward Reaching
    // if i is root
    if i = 0 then
        // Set root to be initial position
        p_i ← initRoot
    end if
    // Find distance r_i between new and old joint positions
    r_i ← |p_{i+1} − p_i|
    l_i ← d_i/r_i
    // Get new joint positions
    p_{i+1} ← (1 − l_i)p_i + l_i p_{i+1}
    if i = n − 2 then
        // If i is node before end effector
        // Get difference between p_n and t
        diff gets |p_{n−1} − t|
        if diff >MoE then
            // Not within the margin of error, so reverse and run again

            doFabrik(i + 1,"f")
        end if
    else
        // Go to next joint in the chain
        doFabrik(i + 1, direction)
    end if
end if
```

## 4. Evaluation

### 4.1. Qualitative Evaluation

The FABRIK algorithm was relatively straight forward to implement. For the purpose of the project, the hierarchical model implemented was comprised of a chain of rigid segments of fixed length, connected by joints with three degrees of freedom. The number of joints was variable and could be specified by the user. Figure: 3 shows the before and after position of a ten link hierarchical chain as it moves from it's initial position to the target position. It can be seen that the chain moves in a smooth shape to the destination point.

This same model can also be run by dragging the end point with the cursor. In testing, the model moved smoothly and realistically as would a real body of similar design.

When run with a spline as input, and setting the number of joints to simulate a robot arm, with an upper-arm, forearm, hand and end effector, the arm followed the spline and the angles of the joints were realistic to an actual robot arm.

Figure 4 demonstrates a similar chain with more joints and a fixed root successfully following a spline.

At this point in time, the program has a chance of crashing when taking mouse input that is out of range. the chance increases as the number of joints gets larger. In testing it is believed that the issue is tied to the mouse input handling associated with zooming the camera in and out, not the FABRIK algorithm itself.

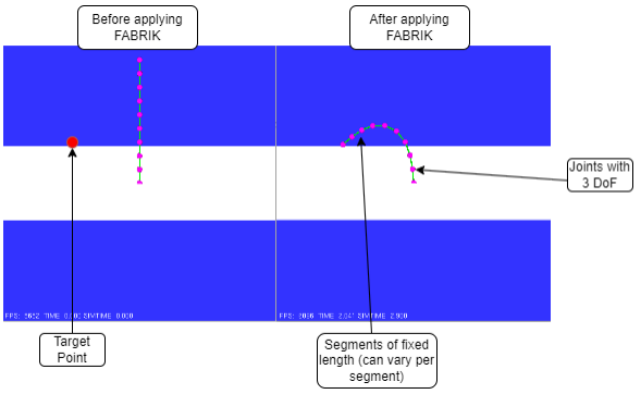


**Figure 3:** *A model with fixed equal length segments moving to a designated point using FABRIK. In the image on the left the user has specified a target poin, after running the simulation the end effector moves to the supplied target point and the previous joints move appropriately as if following*

### 4.2. Quantitative Evaluation

#### 4.2.1. FPS

The model was run with varying numbers of joints and following the "Animation" spline shown in figure: 1, The average FPS was recorded by taking intermittent values across the spline and calculating an average. The FPS was seen to decrease with increased number of joints. See Table: 1 and Figure: 5. There was no noticeable visual degradation in performance.

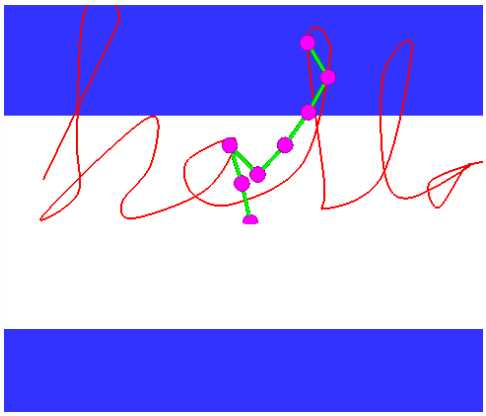

**Figure 4:** *Model following a spline with the FABRIK algorithm*

| Num Joints | 5 | 10 | 15 | 20 |
|---|---|---|---|---|
| Avg. FPS | 8750 | 4600 | 3300 | 2500 |
| Sim Time | 11.05s | 11.05s | 11.05s | 11.05s |
| Real Time | 14.70s | 14.70s | 14.70s | 14.70s |

**Table 1:** *FPS performance at different numbers of joints when following the 'Animation' spline shown in Figure:1*

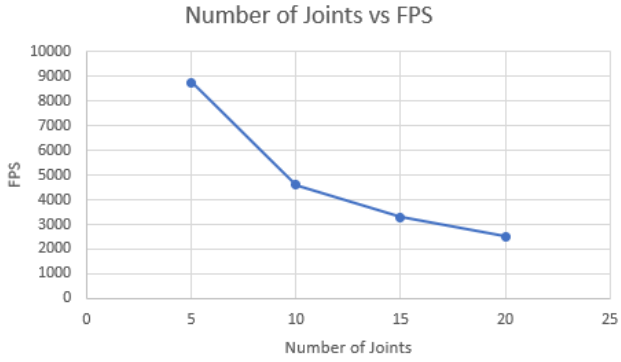


**Figure 5:** *Graph showing approximate FPS recorded when following the 'Animation' spline with a model with differing numbers of joints.*

#### 4.2.2. Iterations to Target

The system was tested with varying numbers of joints (5, 10, 15, and 20) of equal spacing, and a margin of error of 0.01. For all numbers of joints, the algorithm reached the target within the margin of error in less than or equal to three iterations with the most common amount being one iteration. The likelihood of more iterations occurring increased with the number of joints present, as well as when the target point was close to the maximum range of the

structure. The system was then tested with twenty joints and a margin of error of 0.001, and all tests movements reached the target in less than or equal to four iterations, with the most common amount being two iterations. When tested with differing distances between joints, the number of iterations had a much higher likelihood of being larger. In testing with a five joint system with segment lengths 1, 2, 3, and 1, the maximum number of iterations observed was 18, while the minimum was one.

#### 4.2.3. System and Software Specs.

The program was run on a Windows 10 system with the hardware configuration shown in Table: 2 The code was written in Visual Studio Code using c++ with openGL, and based upon the base code supplied at the beginning of the semester.

| | |
|---|---|
| Processor | 5 AMD Ryzen 7 5800X 8-core |
| RAM | 32.0 GB |
| Disk | 500GB Samsung 970 EVO Plus NVMe M.2 SSD |
| GPU | NVIDIA GeForce RTX 3000 10GB GPU |

**Table 2:** *Configuration of hardware used for the project*

## 5. Conclusion

This project successfully implemented the FABRIK algorithm. FABRIK is a relatively simple algorithm to implement. it appears upon testing to produce smooth animation, reaching the target quickly and with low computational cost and within few iterations. Even without joint rotational restrictions, the structure moved in a natural and realistic fashion.

### 5.1. Challenges

The main challenge encountered during the process of implementing FABRIK was learning inverse kinematics and FABRIK with no prior knowledge and before it was covered in the course material.

Other challenges were associated with inexperience with openGL, in particular taking mouse inputs and transitioning between window coordinates and object coordinates.

Lastly, implementing multiple forms of input proved to be challenging as the algorithm was being given target point from multiple methods, so a toggle to only have one method active at a time was implemented.

### 5.2. Future Work

If work was to continue on this project, implementing compatibility with structures with multiple roots or end effectors would be the primary goal. Implementing a method to restrict joint rotations and degrees of freedom would be another main focus. If both these systems were achieved, modeling a humanoid character that would be realistically articulated should be possible.

## References

[ACL16] ARISTIDOU A., CHRYSANTHOU Y., LASENBY J.: Extending fabrik with model constraints. *Computer Animation and Virtual Worlds 27*, 1 (2016), 35–57.

[AL09] ARISTIDOU A., LASENBY J.: Inverse kinematics: a review of existing techniques and introduction of a new fast iterative solver.

[AL11] ARISTIDOU A., LASENBY J.: Fabrik: A fast, iterative solver for the inverse kinematics problem. *Graphical Models 73*, 5 (2011), 243–260.

[BDMS84] BALESTRINO A., DE MARIA G., SCIAVICCO L.: Robust control of robotic manipulators. *IFAC Proceedings Volumes 17*, 2 (1984), 2435–2440.

[SFC*20] SANTOS P. C., FREIRE R. C. S., CARVALHO E. A. N., MOLINA L., FREIRE E. O.: M-fabrik: A new inverse kinematics approach to mobile manipulator robots based on fabrik. *IEEE Access 8* (2020), 208836–208849.

[TTY21] TAO S., TAO H., YANG Y.: Extending fabrik with obstacle avoidance for solving the inverse kinematics problem. *Journal of Robotics 2021* (2021), 1–10.

[WC91] WANG L.-C., CHEN C.-C.: A combined optimization method for solving the inverse kinematics problems of mechanical manipulators. *IEEE Transactions on Robotics and Automation 7*, 4 (1991), 489–499.