Using the A-Star Path-Finding Algorithm for Solving General and Constrained Inverse Kinematics Problems.

Samuel Grant Dawson Williams

February 10, 2008

Abstract

Inverse Kinematics is a topic with much application to real world problems. This report discusses a new approach to solving this problem, with an emphasis on typically constrained robotic systems. The A-Star algorithm is adapted for the first time to search through 3-space to solve the Inverse Kinematics problem, and can produce a decidable, sound and complete search, which can perform on par or better than other numeric methods when dealing with complicated constrained systems.

I dedicate this work to my grandfather Orm Walker. He is a great scientist and astronomer, and a much loved member of our family. I'll always fondly remember the times he took me to the observatory to look at the vastness that is the eternal universe.

Many Thanks to R. Mukunden, my supervisor for his time and input.

Many Thanks to T. Takaoka for his high expectations.

Contents

1 Introduction

Samuel R. Buss [1] defines:

> A rigid multibody system consists of a set of rigid objects, called links, joined together by joints. Simple kinds of joints include revolute (rotational) and prismatic (translational) joints. It is also possible to work with more general types of joints, and thereby simulate non-rigid objects. Well-known applications of rigid multibodies include robotic arms as well as virtual skeletons for animation in computer graphics.
>
> To control the movement of a rigid multibody it is common to use inverse kinematics (IK).

Inverse Kinematics is a difficult field that has its application in many areas of the real world, including medical research, robotics, and computer simulation/cinematography. The performance and quality of algorithms to solve this problem are therefore very important.

Because of the direct correspondence, the algorithms discussed in this paper will be contextualised within the problem domain of robotic movement. This is so that we are working within an easily visualised domain that should be familiar to most people.

Robotics plays an important role in our everyday life. We rely on robots for many different tasks, such as manufacturing, working in hostile environments, or working under extreme requirements (precision or strength). Without robotics, we could not enjoy many of the conveniences we have available.

Building robots requires that the software and hardware work together precisely. We must have strong and reliable joints, sensors must be accurate, and we must have controlling software adequate to describe the motions we require.

Such motions can be difficult or even impossible to find, depending on the configuration of the robot. The robot itself will usually have a finite reach, and a finite area in which it can move objects (we can refer to this space as the robot's workspace). Often, a robot will need to pick up an object, or perform a specific manipulation. This problem is commonly referred to as the inverse kinematics problem.

2 Background

2.1 Inverse Kinematics

A robotic arm consists of a kinematic chain - a sequence of jointed arms which can be rotated and translated. The end of this arm, where a hand or gripper may be located, is called the end effector. See figure 1 for an example.

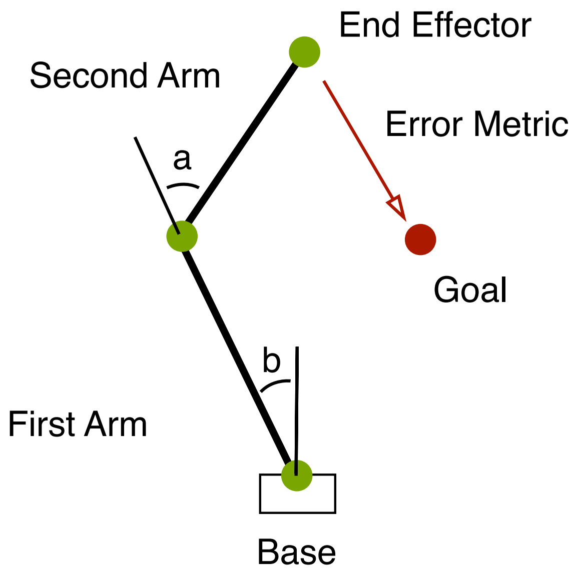


Figure 1: A simple 2-dimensional armature that consists of two arms. The rotations a and b can be adjusted. The error metric is the distance from the end effector to the goal.

Often, constraints are inherent in the construction of the robot arm. For example, most joints will rotate on a single plane, and may have a limited angle of rotation. Translational joints will have a minimum and maximum displacement.

An inverse kinematics solution provides a set of rotations in an attempt to make the end effector reach a target position or object, while taking into account any constraints applied to the arm.

We need some algorithms to solve this problem. An algorithm in this problem domain describes a set of steps we can take to find a set of angles that minimises the distance between the end effector and the goal. We can call this distance the error metric.

Every rotational or translational joint corresponds to at least one degree of freedom. A rotational joint could be considered more than one degree of freedom, depending on the set of allowed rotations. A translational joint that moves in more than one direction could also be considered more than one degree of freedom. In general, a single degree of freedom should only require one numeric input to control.

Given the number of degrees of freedom, we can consider that the function takes a single vector input, the location of the goal in Euclidian co-ordinates, and produces between zero and infinite solutions. Each solution consists of a set of rotations/translations for each degree of freedom. We can consider a solution to be acceptable if the error metric of the provided solution is acceptable. This determines how accurate a particular solution needs to be.

Many types of algorithms are well suited to this kind of problem - i.e. finding a minima. We can simply summarise the problem as a gradient search over N-space where N corresponds to the number of degrees of freedom. However, convergence is only one element of the problem. We must consider that there could be arbitrary important constraints applied to the arm. For example, a robot carrying a heavy load must remain balanced.

2.2 Constraints

We need to consider constraints on the armature. Below is a list of example constraints that can apply to a robotic arm. Although it is hard to divide them up exactly, they have been categorised to provide some insight into where they fit into the problem space.

Armature Constraints:

- Minimum and maximum angles of rotation (physical joints may not be able to rotate to arbitrary angles)
- Minimum and maximum displacements (physical joints will have limits on the distance they can extend)
- Avoidance of 'unusual' or 'strange' configurations (if visual appearance is important, we need to consider this)
- Minimise the motion of the end effector (it may be important to get to the goal as fast as possible)
- Minimise the rotation of joints (it may be important to minimise energy spent)
- Minimum or constant velocity/torque rotations (the arm may need high precision or accuracy)

Environmental Constraints:

- Areas of exclusion (i.e. areas the robot arms must not travel or intersect with)
- Areas of avoidance (i.e. a robot on a boat should avoid water unless directly ordered to move to an underwater goal)

Carrying Constraints:

- Maintain balance (i.e. a robot doesn't want to topple over when carrying something heavy)
- Maintain orientation (i.e. a robot arm should maintain a vertical grip if it is carrying a bucket of molten steel)

We would like to identify what constraints are impossible to avoid (such as limitations of rotation), which constraints limit the environment the robot can work in, and those which are simply limiting our problem domain. If there is a task that requires a particular constraint, and the robot can't be programmed to work within that constraint, then it may not be possible to use that robot for the task (such as if the arm can't guarantee it is always upright, then we can't use it for holding glasses of water). This may limit the application of robotics, even in places where they could be used successfully.

We can consider many more such constraints, this list is not exhaustive.

3 Current Solutions

A number of current algorithms can help us solve the inverse kinematics problem. They range from very simple to very complex. Most current solutions are numeric. Some methods for very simple robots will be pre-computed and completely deterministic, but we will not consider these solutions as they are not applicable to general armatures.

An inverse kinematics solution is acceptable if it meets two requirements: the error metric is less than a predefined tolerance, and the configuration of all joints and arms is within any constraints applied to the system.

An inverse kinematics algorithm is useful if it has the following properties: it will converge on a solution if one exists, it is always decidable, and its behaviour is predictable. An inverse kinematics algorithm that fails to converge or is not 'decidable' (i.e. never gives an answer of any kind) can't provide us with an acceptable result.

A numeric method is typically iterative. Because of this, we will see that many numeric algorithms fail to provide acceptable solutions. A numeric method typically will not backtrack - it will continue to make adjustments to approach a local minima, even if a more optimal solution exists.

It can be difficult to find an acceptable solution if constraints exist. A constraint restricts the set of solutions, and make it difficult for a numeric method to find an acceptable solution. This is because a local minima may be 'blocked' by a constraint, therefore the algorithm can never converge.

3.1 Cyclic Co-ordinate Descent

Cyclic Co-ordinate Descent [2, 3] is a iterative numeric method that evaluates one joint at a time. It is a simple method that may not converge, and may not find a result even if one exists.

We consider the joints in sequence, from the last joint to the first joint, and back again, in succession. At each joint we calculate the rotation required to minimise the error metric. We then apply that rotation to the joint, and move to the next joint.

This solution is not very good, because we are not guaranteed to converge on a solution, even if one exists, and if we don't find a solution, this is not a definite indication that no solution exists. We also might not find an acceptable solution - i.e. we may make rotations which are impossible given the constraints applied to the armature.

We may also spend a considerable amount of time converging on a solution, or we may never converge on a solution. This solution is not decidable, so we may or may not find a solution even after a very large number of iterations.

One case to consider is when the arm is currently pointing right through the goal, then in this case, no individual rotation can minimise the error metric. Thus, Cyclic Co-ordinate Descent is not a good solution, except for the most trivial and contrived examples.

3.2 Jacobian Matrices

A Jacobian matrix (the concept was conceived of by Carl Gustav Jacob Jacobi in the 1800s) can be used to iteratively solve the inverse kinematics problem [1, 4, 5]. For sufficiently small changes in the end effector position, the change in angle and position is linearly related by the Jacobian matrix.

We can build a single matrix that represents this relationship for all joints, and then we will need to solve for its inverse. The inverse matrix represents the function that gives us a change in angle/displacement for each joint, given a change in position for the end effector.

Calculating the inverse matrix may also be complicated. Typically, the Jacobian matrix will be irregular, and thus standard techniques cannot help. One option is to use a pseudo-inverse matrix. Because of this, it is possible for the solution to become unstable, because we are relating a non-linear and possibly non-contiguous function to a linear domain.

This method should eventually converge on a solution, which is why it is more useful than CCD. Unfortunately, being a numeric method, it still may be non-optimal (i.e. the solution it converges on may not be the best solution). It also will have a limited ability to take into account external constraints. Like CCD, if we don't find an acceptable solution, it doesn't mean that an acceptable solution does not exist.

It is also worth noting that while this method should converge when used appropriately, if the mathematics becomes unstable, then this algorithm will fail to provide us with any useful information.

3.3 Neural Networks

A number of techniques exist that utilise neural networks [6, 7]. There are a number of important factors to consider when dealing with neural networks in any type of algorithm: Training data is incredibly important, and the behaviour may be unknown in some cases. These two factors alone make any solution based on neural networks difficult. In safety critical applications, unpredictable behaviour is not good. Solutions based on neural networks may also be slow to evaluate.

Even though these problems exist, this kind of algorithm can provide us with some very useful results. Where training is good and we can be sure (i.e. through testing) of the behaviour of the algorithm in most cases, we can have deterministic output within a constant timeframe (running data through the neural network should be relatively fast).

Another advantage of this kind of solution, is that we can incorporate static constraints into our training data. This allows us to process areas of exclusion, and any other constraints which can be considered static.

Despite these advantages, ultimately neural networks still have some major problems. They are not a reliable general solution.

4 Fundamental Problems

Numeric solutions fail to take into account many real world problems. Some algorithms will become undecidable given a constrained system. For example, the Cyclic Co-ordinate Descent method, when constrained, may not converge on a solution. This makes it difficult to access whether or not there are any solutions to a particular problem with a numeric method.

An algorithm may not be able to find an acceptable solution, even though one may exist, because it may become stuck in a local minima. When a numeric algorithm manipulates joints individually, it may be impossible to rotate any particular joint that reduces the error metric. See figure 2 for an example of this. In other cases, such as with the Jacobian inverse solution, we may converge, but at a sub-optimal result.

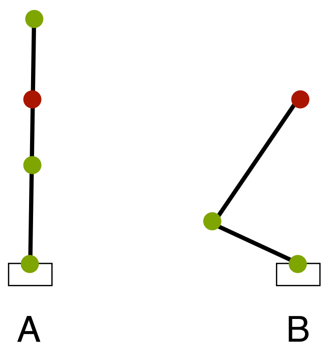


Figure 2: The top green dot represents the end effector, and the red dot indicates the goal. The distance between the two is the error metric. There is no single rotation which can be applied to the arm A which will minimise the error metric. Therefore, even though a solution exists, such as B, some numeric algorithms will have difficulty finding it.

If a numeric algorithm fails to find an acceptable solution, we can never be sure that a solution does not exist. With any algorithm that does not consider the whole environment, such as the ones discussed so far, we cannot perform a thorough search to check all possibilities - typically, our goal is to reduce the error metric in one way or another. Thus, we may need to actually 'backtrack' our search and try another avenue to find a solution.

We need some basic properties to be satisfied:

- We want to find all solutions (but if we can find at least one solution quickly on average that is good)
- We want the algorithm to be decidable. (i.e. it will halt eventually)
- We need the algorithm to be both sound and complete. (If a solution is found, it is a valid configuration, if no solutions are found, then no solutions exist.)
- We want to be able to apply arbitrary constraints, such as those listed earlier.
- We want to have explicit upper performance bounds.

5 A-Star Algorithm

The A-Star algorithm [8] is a flexible cost based algorithm, which can help satisfy all our requirements.

The A-Star algorithm is a best-first search algorithm that finds the least costly path from an initial configuration to a final configuration. It uses an exact + estimate cost heuristic function. The exact cost is known for any previous steps, while the estimated cost to reach the final configuration from the current can be estimated. The cost function must be admissible, i.e. the estimated cost must be less than the actual cost, this produces computationally optimal results.

The most essential part of the A-Star algorithm is a good heuristic estimate function. This can improve the efficiency and performance of the algorithm, and depends on the specific state space being explored.

Generally, the A-Star algorithm maintains two lists, an open list and a closed list. The open list is a priority queue of states, where we can pick out the next least costly state to evaluate. Initially, the open list contains the starting state. When we iterate once, we take the top of the priority queue, and then initially, check whether it is the goal state. If so, we are done. Otherwise, we calculate all adjacent states and their associated costs, and add them into the open queue. See figure 3 on page 12 for an example.

A-Star is complete. It will find a solution if a solution exists. If it doesn't find a solution, then we can guarantee that no such solution exists.

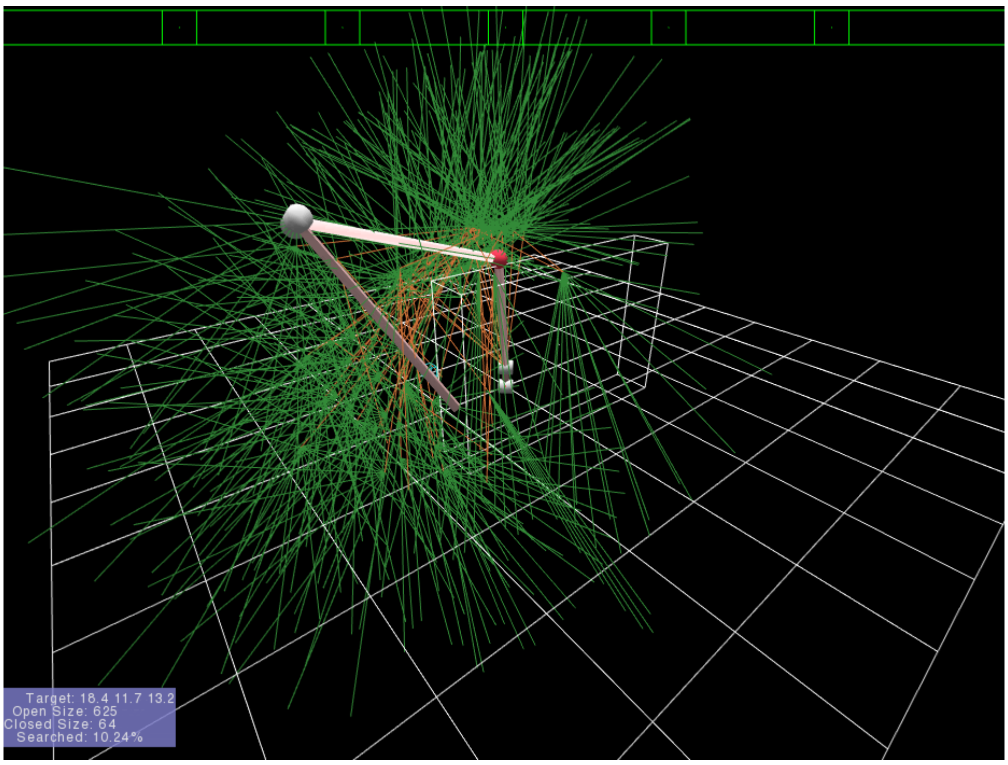
A-Star will find a path with the lowest possible cost. This will depend heavily upon the quality of the cost function, and estimates provided.

5.1 How do we apply it?

From an implementation point of view, we can use quaternions to represent rotations. Every joint consists of a quaternion rotation and a displacement.

Each state in the search space is an arm configuration. This configuration may not specify the position for all arms, but a subset of arms sequentially from the base of the armature. We must subdivide the problem into a set of states at each joint. A state encapsulates any possible changes to a joint.

To produce adjacent states, we consider the kind of joint constraints. If we have a planar constraint, we limit our rotations to those on the plane, and within any angle constraint. If we have no constraints, or angle based constraints, we build a set of rotations to represent a finite set of all rotations possible. We specify a constant value for the subdivision size. See figure 4 on page 14 to see various different joints and their adjacent states.



Target: 18.4 11.7 13.2
Open Size: 625
Closed Size: 64
Searched: 10.24%

Figure 3: This is solved search. Green lines represent open search space, while orange lines represent closed search space. A rectangle is occluding the goal, and thus the arm must reach over it. This is done by culling arms which intersect with the bounding volume.

The starting state is the base of the armature. From there, we can consider all configurations of the first joint. We put all these states into the open list.

We then incrementally search the open list for a solution. We take out a state, evaluate it, and find out what next states are possible, and put them back in.

We introduce a tolerance value for the error metric. This tolerance controls how small the error metric needs to be before we consider that we have an acceptable solution.

5.2 Considerations

Finite search space may mean that a solution is a false positive, but we will never miss a solution as long as our tolerance is acceptable.

The cost function is incredibly important for the general solution to be efficient. A joint that is displaced or rotated in such a way that changes its distance from the goal is one useful metric. However, occasionally we have a joint that rotates around its own axis, and this causes no change in distance. So, we need to introduce another metric, which we will call approximate final position cost.

5.3 Approximate Final Position

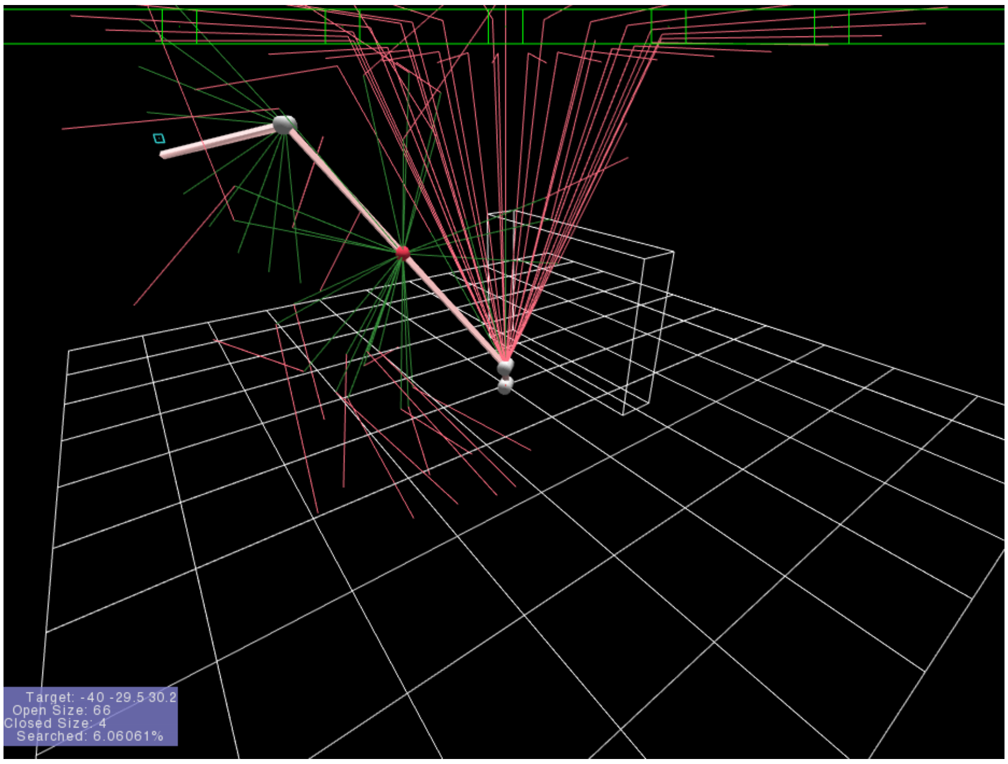
When a joint rotates around its own axis, we need to consider how constrained the armature is. In some cases, a rotation such as this may have little effect on the final position (for example, if the next joint is a ball joint, or a rotation on the same axis). In other cases, this rotation may be very important to finding a solution (such as if all further rotations along the armature are perpendicular to this one). See figure 4 on page 14 for an example of this.

So, we construct the approximate final cost, by looking at the average position of constrained arms. For example, if an arm is constrained to rotate between $0°$ and $90°$ and limited to rotations around the plane $< 1, 0, 0 >$ then we consider the average final position to be a $45°$ rotation on this plane.

We compute the average final position by looking at all joints along the chain that are constrained in this way, and then we build a new armature out of these positions. We then compute the cost of this approximate final position, and the goal, and use this as a metric.

6 Results

Because the new method described is so different from any other existing method, it is difficult to compare them closely. Therefore, we will consider a very simple metric.



```
   Target: -40 -29.5 30.2
 Open Size: 66
Closed Size: 4
 Searched: 6.06061%
```

Figure 4: Here we can see a solution. The pink lines show the approximate final position. The green lines show open search space. The blue square in the upper left corner indicates the goal. Also of interest is the different type of joints shown. The top joint is a limited planar rotation, and we can see that there are minimal green lines extending from it (i.e. adjacent states). The red ball joint in the middle can rotate to any angle, and we can see that many adjacent states extend from it.

Given an arm with n joints and m planar constraints, we will compare each method and record the time it takes to either find a solution, or fail. If a numeric algorithm fails to converge, we will also note this. The results of this are in table 1 on page 15.

Table 1: The results from testing several different methods on varying complex arms. Failures indicates the percentage of tests which did not converge or produce an acceptable result. All times are in seconds.

		CCD		A-Star	
n	m	Failures	Time	Failures	Time
3	0	0%	0.0000704	0%	0.119
3	1	0%	0.00277	0%	0.0706
3	2	12%	0.0262	4%	0.0244
3	3	70%	1.19	18%	0.0148
4	0	0%	0.000386	0%	4.71
4	1	0%	0.00109	0%	2.93
4	2	0%	0.00668	0%	0.554
4	3	22%	0.814	0%	0.0461
4	4	40%	0.462	0%	0.0258

These results provide an insight into how the algorithm can be applied. As expected, the performance of A-Star algorithm is very poor when the arm is not constrained. However, in this case, a numeric method is very efficient.

In the case where the A-Star algorithm cannot find a solution, such as in cases $(3, 2)$ and $(3, 3)$, the average performance is still very good (compares equally or better than CCD). However, in these cases, the results of CCD are not very useful, the worst case being a 70% failure rate. It should be noted that when the A-Star algorithm fails to find a solution, it means that no such solutions exist - which means that the CCD algorithm will obviously fail in these cases also. However, the failure rate for CCD indicates that there were failures even when a solution existed.

The performance of the A-Star algorithm was very good, and surpassed the CCD algorithm, in cases where every arm was constrained. These cases generally reflect real world conditions. We also had no failures in any case, where as CCD performed very poorly in this regard, which generally had a non-convergence (i.e. the algorithm didn't halt) of about 5% in cases where failures to find a solution were recorded.

7 Discussion

When considering the results, we can see an important trend: the numeric method becomes more and more expensive and begins to fail, but the A-Star algorithm gets faster and faster and will always find a solution if one exists. This is most important part of this algorithm - as we add constraints - things get faster, but with numeric algorithms, it is the complete opposite.

One other useful element to consider, is that we can introduce new metrics into the cost function, so that arbitrary constraints can be included. This makes the algorithm much more powerful.

We can also cull the search space as required. For example, any arm which traverses into the floor can be immediately culled, thus reducing the size of the search space. In the benchmark, this was not done, however it would immediately improve performance by approximately 2 times.

A single unconstrained joint can cause the search space to become very big. This is shown in the speed that it takes to resolve unconstrained joints in the results. However, in the real world, armatures are normally heavily constrained (i.e. cases $(3, 3)$ and $(4, 4)$), therefore search space will often be quite small. Generally, a robot with n arms will have $m = n$ constraints, in the form of limited rotation around a single axis.

7.1 Performance Analysis

We can consider that the performance of the algorithm in general is bounded by the performance of the A-Star algorithm. A-Star algorithm can reduce the search space logarithmically, depending on the quality of the heuristic estimate. So, we will discuss the size of the search space in relation to the number of constraints applied.

In the case that no constraints are applied to a joint, we must build a large set of possible rotations. This generally is S^2, where S is the subdivision size. We actually only need to consider sets of rotations constructed from rotations in the x and y planes. We could also add rotations from the y plane, but this has been found to increase the search space with no real gain in performance. We can see this in the results - for a fully unconstrained arm, we typically have a 0% failure rate.

In the case where a joint is constrained, we just need to consider S possibilities. Because of this, the total size of the state space can be reduced dramatically by adding constraints.

Generally, we can consider the state space to be of size $O(N^{S^2})$ for cases where there are N unconstrained joints. When we have a fully constrained system, this becomes $O(N^S)$.

Because of this performance, this algorithm may become too slow when many arms are considered. However, the A-Star algorithm can help reduce the number of states that need to be evaluated in this huge space.

Generally, it was found that $S = 6$ produced good results.

8 Future Work

8.1 Improvements

There are a number of ideas which could be made to this algorithm to improve performance.

One major problem is that the algorithm will not be able to produce a result if the tolerance is not high enough to account for the finite subdivision of space. In order for the algorithm to find all solutions, the tolerance must be acceptable.

Also, the solution found may not be minimal. Therefore, we may need to evaluate it further with another algorithm to find an exact solution.

To improve this behaviour so that we can produce an optimal result, we could look at implementing a multi-resolution search. In this case, instead of taking a constant subdivision size to produce new states, we could look at the cost of the current orientation, and subdivide based on its distance from the goal. So, generally, the closer to the goal we get, the more we subdivide the search space. This may provide us the means to reach a very minimal solution, but still within a finite search space.

Another issue concerns the case when no solution exists. That is, when there is no solution, the entire state space will be explored, which could be very costly. There are several ways to reduce this. One such approach would be to calculate the radius of reach for each joint - i.e. what is the maximum distance we can get to from this particular joint. With this information, we can immediately cull states from which we can never reach the goal through any permutation.

This is not an exhaustive list of all possible improvements - these are intuitively the next steps to improve the current implementation.

9 Conclusion

This new algorithm can provide more information than any other algorithm. We can analyse the open and closed list to learn about our environment in ways that a numeric algorithm can never be used.

We can deal with arbitrary dynamic constraints, such as moving objects, static constraints such as walls, and internal constrains such as joint rotation limitations.

We can manipulate any size armature with exact performance upper bounds. We can evaluate the search space size beforehand to get an idea of how long a search could take, if there are no solutions. However, if a solution does exist, it will typically be found very quickly, depending on the complexity of the constraints applied and the suitability of the cost function used.

We will find all solutions that are possible, and we can be sure when told that no solutions exist. We can rely on results to be accurate and useful in the case that such information is important for further processing (e.g. a robot on a moving platform may first need to check whether it can reach an object, if not it will move its platform closer.)

No other inverse kinematics algorithm has all these properties, and this is a completely new approach to solving this kind of problem.

Because this research has yielded interesting results, an article may be published.

References

[1] Samuel R. Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods, 2004. [Online; accessed Feb-2008]. Available from: `http://math.ucsd.edu/~sbuss/ResearchWeb/ikmethods/iksurvey.pdf`.

[2] L. C. T. Wang and C. C. Chen. A combined optimization method for solving the inverse kinematics problems of mechanical manipulators. *IEEE Transactions on Robotics and Automation*, 7:489–499, August 1991.

[3] Chris Welman. Inverse kinematics and geometric constraints for articulated figure manipulation, 1993. [Online; accessed Feb-2008]. Available from: `http://fas.sfu.ca/pub/cs/theses/1993/ChrisWelmanMSc.ps.gz`.

[4] M. Girard and A. A. Maciejewski. Computational modeling for the computer animation of legged figures. *ACM Computer Graphics 19, 3*, pages 263–270, 1985.

[5] Jianmin Zhao and Norman I. Badler. Inverse kinematics positioning using nonlinear programming for highly articulated figures, 1994. [Online; accessed Feb-2008]. Available from: `http://ai.stanford.edu/~latombe/cs99k/2000/badler.pdf`.

[6] Benjamin B. Choi and Charles Lawrence. Inverse kinematics problem in robotics using neural networks, 1992. [Online; accessed Feb-2008]. Available from: `http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19930009687_1993009687.pdf`.

[7] Eimei Oyama. Inverse kinematics learning for robotic arms with fewer degrees of freedom by modular neural network systems, 2005. [Online; accessed Feb-2008]. Available from: `http://www.macdorman.com/kfm/writings/pubs/Oyama2005InverseKinematicsIROS.pdf`.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC4 (2)*, pages 100–107, 1968.

Appendices

A Research Log

In mid-November Mukunden and myself discussed a paper that he had previously supervised regarding inverse kinematics. This paper discussed an interesting technique incorporating triangulation as a method to find inverse kinematics solutions. (1-2 hours, mid November)

I researched inverse kinematics via the internet, and found a huge amount of information. There were several example programs for which I examined the source code to learn about the approach they were taking. I ported one example to my computer (It's a Mac so it wasn't directly compatible). This particular example was demonstrating the use of Jacobian inverses. It was pretty unstable. (10-15 hours, end November)

At this point, I was interested to implement my own method of solving this problem. I decided to write something incredibly simple, and built a random permutation solver. This method worked by making random rotations, and discarding rotations that moved the end effector away from the goal. This method was fun to watch and produced some interesting results. It obviously wasn't very optimal, but had an interesting quality of being approximately constant-time based on the distance to the goal. (20-25 hours early December)

It was at this point I thought about how to improve this algorithm. I knew how CCD functioned, but didn't see that as being useful. I wanted to consider how to control a robot that could play some game, or move around like the 'Sentinel' robots from the Matrix (movie). These robots have arms that can reach through holes, and plan incredibly complex motion (obviously these are not real robots, but the fictional example was inspirational). My question was: how can we really find these solutions? (5-10 hours early December)

This is when I began to investigate neural networks. I found some interesting papers regarding these kinds of algorithms. The concept was very interesting, but also very complex. (2-5 hours mid December).

It was at this point, that I considered the problems with numeric solutions. The random solver that I made was really as good as it gets for a numeric solution - we can improve performance, but we can't really improve the result of the algorithm. I was studying some pages that described inverse kinematics as simply a gradient solving problem, and this I found an interesting conclusion. We can easily implement some basic searches across this kind of space. (2-5 hours mid December)

So, I started working on a new solution. I wanted to implement a solver using the A-Star algorithm. Working on this took a huge amount of time, because conceptually, it was a new approach to the problem and I've only ever used A-Star algorithm in simple problems. It really took me a long time to understand how to get everything working correctly.

I was able to use previously written code for the A-Star algorithm, but it had to be adapted to incorporate support for new requirements. Some other previously written code such as window management and user interface was also used to aid my testing and experimentation. This actually saved a considerable amount of time, probably in excess of several hundred hours. It may also be worth noting, that without previously implementing the A-Star algorithm, I might not have had the ability to see its application to this problem.

In terms of the implementation, the first issue was the abstraction of the Armature structure. This wasn't too complicated. I implemented arms in terms of State and Position. State represented any adjustments to rotation or displacement, while Position was information evaluated to give 3-space co-ordinates for the start, end and rotation of an arm. An Arm itself could have constraints applied to it, and this was also a slightly difficult concept to specify.

All rotations are stored as quaternions. This allowed for flexible structure and helped to add mathematical rigidity to position calculations. This caused minor problems at first because I didn't use quaternions throughout the entire system, but once they were used in all places necessary, everything worked accurately and robustly.

The next conceptual step to make was the idea of sets of rotations. It is necessary to build a set of all possible rotations for a particular joint. Quaternions are utilised and help us produce sets of all rotations, though the rotations produced are not initially intuitive... they do make sense if you consider how quaternions function, however. (70-90 hours end December to mid January for initial working implementation)

The biggest area of concern was the cost function. This was initially just a distance based function, but in the case that an arm rotated around its own axis, the distance function did not change. This caused the evaluation of unnecessary states. As discussed, an approximate final position function was devised which tries to estimate the average position that an arm will reach when rotated a particular way. (10-25 hours, mid January)

Once this was achieved, the function performed very well. Solutions could be found efficiently, and I was very happy with the algorithm.

I then decided to test the capabilities of the new algorithm, and added collision detection. This produced very interesting results and worked as expected. (10 - 15 hours end January)

I then decided to implement CCD, to use this as a benchmark for performance analysis. This was trivial, but had some minor problems getting it to work correctly. (10 - 15 hours end January)

Once this was done, I produced a benchmark between the various algorithms. This benchmark was fully automatic. (10-15 hours early February)

The presentation was produced in Keynote. I decided to take movies of my example programs to minimise the possibility of problems and time usage. I spent time doing trial runs with my flatmates and tweaking the content to take into account the short time we had to give the talk. I had to strip out a lot of content, however my Girlfriend Ayako who was one of the last people to listen to it (and not proficient in this field at all), told me she could understand the general idea of what I was trying to do, and so I was pleased. (20 hours mid February)

This report is produced using LaTeX. It has been proof read by a number of people, and feedback has been incorporated. Many people at the University have been impressed by the report, and more importantly, understood its content. Mukunden has recommended that I publish an article regarding this research, and so this report has been written with that in mind. (25-35 hours mid February)