Comparing Various Methods on Solving Inverse Kinematics

Xiaochun Tong

April 2020

1 Problem Formulation

First I will briefly introduce forward kinematics:
Each skeleton is made of m bones. Each bone has a unique parent, and has has three real number determine its position relative to its parent. Each bone at rest has its tail at (0,0,0) and its tip and (l,0, 0) where l is the length of the bone.
The transform of the bone from its rest positon to pose position is computed as the following: The three numbers x_1, z, x_2, describes the angle of rotation of the bone first along the x axis, then z, then x. After the bone is rotated, it is then applied with recursively its parent's transform and its tail is connected to its parents' tip.

So basically forward kinematics maps the angles of rotation of each bone (total 3m numbers) to its final pose position.

Now let's introduce inverse kinematics:

For n of these bones, their tips are marked as end effectors. We define function $x(A) : R^{3m} \rightarrow R^{3n}$, where A is a list of 3m numbers describing the joint angle of each bone. $A[3i+1, ..., 3i+3]$ is the joint angles for the ith bone. In other words, $x(A)$ maps the joint angles into end effector positions. Let the target end effector positions be $x_b \in \mathbb{R}^{3n}$,We want to find A such that $x(A) = x_b$.

This can be seen as an optimization problem where we minimize $E(x(A)) = \frac{1}{2}||x(A) - x_b||^2$ with a box constraint: each bone's angle of rotation has a min and max value.

2 Working with Constraints

The algorithms we use (gradients, BFGS) are used for unconstrained optimizations. To use them to solve IK, we have to find a way to deal with constraints.

Gradient Projection Method

Let the constraints be: for $1 \leq i \leq n$, $l_i < x_i < u_i$. The gradient projection method works as following:

For the kth iteration, we choose a descent direction p_k.
Then we want to find a step size α_k to minimize the function

$$\phi(\alpha) = P(x_k + \alpha_k p_k)$$

where $P(x)$ is defined as

$$P(x)_i = \min\{\max\{x_i, l_i\}, u_i\}$$

Although ϕ is only piecewise-differentiable, a_k can be efficiently found using backtracking on ϕ and applying wolfe conditions as such a_k satisfies the condition given in Paul H. Calamai & Jorge J. Moré's paper[1].

3 Methods

3.1 Stopping Criteria

The stopping criteria is defined as if any of these are true, the algorithm should stop:

1. $\alpha_k = 0$ (possibly p_k is not a descent direction)
2. $\|\nabla E(x(A))\| < \delta_g$
3. $E(x(A)) < \delta_f$

Thus if a method stops with a large $E(x(A))$, its probably get stuck or unable to proceed. Also any method would time out if it cannot find a solution in less than 1 second.

3.2 Projected Gradient Descent

The easiest way is to perform gradient descent on $E(x(A))$, the gradient is given as:

$$\begin{aligned}\nabla E(x(A)) &= D[x(A)]^T D_x(E(x)) \\ &= D[x(A)]^T (x(A) - x_b)\end{aligned}$$

where $D[x(A)]$ is the jacobian of $x(A)$.
We use finite difference to compute the jacobian. Though other methods such as automatic differentiation is applicable, we choose finite difference since its easy to implement. For each iteration, $A_{k+1} = A_k - \alpha_k \nabla E(x(A))$, where α_k is found using backtracking as mentioned above.

3.3 Newton's Method

A better way to optimize $E(x(A))$ is to utilize its second derivative, or the Hessian of $E(x(A))$. And each iteration can be computed as $A_{k+1} = A_k - \alpha_k H^{-1}(A)$. In this way, when k is sufficiently large, we can have at least quadratic convergence.

3.3.1 Problem

The Hessian is not analytically available when doing IK. We had to approximate it by computing the Jacobian of the gradient, which is a expensive process and suffers from numeric issues. In practice, I found that the Hessian is almost always singular, mostly because the floating point numbers are not sufficiently precise to be able to compute the hessian and its inverse.

As a result, the implemented Newton is not usable at all and we would not include it in the following comparison.

3.4 Damped BFGS With Gradient Projection

3.4.1 BFGS

We can avoid the problems with Newton methods but still take advantage of its quadratic convergence by using Quasi-Newton methods, which approximate Hessian by using first order derivative.
For each iteration, an approximation B_k is computed and the search direction can be found as $p_k = -B_k^{-1}\nabla f(x)$, B_k can be updated as:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T Bk}{s_k^T B_k s_k} + \frac{y_k y_k^T}{s_k^T y_k}$$

We employed BFGS updates to progressively compute B_k.
To minimize the cost of computing inverse as well as numerical issues, we keep tracks of ${B_k}^{-1}$ directly rather than B_k then compute inverse at each iteration.

3.4.2 Damped BFGS

To ensure p_k is a descent direction, we need to make sure that $p_k^T \nabla E(A) < 0$ or B_k is positive definite. However, this is not always the case. From the assignments we know that if $s_k^T y_k > 0$ is satisfied for all k or if both condition of the strong Wolfe condition are satisfied, B_k is guaranteed to be positive definite. However, non of them are guaranteed since a α_k that satisfies the condition can be greater than 1 and backtracking does not guarantee if such an α less than 1 could be found[2]. This is also observed when I was testing BFGS, where $s_k^T y_k > 0$ is not satisfied, resulting search direction to be invalid.

Nocedal and Wrights' book[2] mentioned that a damped BFGS could resolve this issue.
Introduce $r_k = \theta_k y_k + (1 - \theta_k) B_k s_k$,
where

$$\theta_k = \begin{cases} 1 & \text{if } s_k^T y_k \geq 0.2 s_k^T B_k s_k \\ (0.8 s_k^T B_k s_k)/(s_k^T B_k s_k - s_k^T y_k) & \text{if } s_k^T y_k < 0.2 s_k^T B_k s_k \end{cases}$$

And the damped BFGS update would be:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T Bk}{s_k^T B_k s_k} + \frac{r_k r_k^T}{s_k^T r_k}$$

This guarantees that $r_k^T s_k > 0$, as a result, B_k is always positive definite.

3.5 Gauss Newton

The problem of IK also exhibits the form of least-squares problems (Page 245 of Nocedal and Wrights' book[2]).
The general form of least-squares problems:

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x)$$

And the IK problem can be seen as:

$$E(x(A)) = \frac{1}{2}\left\|x(A) - x_b\right\|^2 = \frac{1}{2}\sum_{j=1}^{3m} r_j^2(A)$$

where $r_j(A) = x(A)_j - (x_b)_j$
Thus we could employ some methods on solving nonlinear least-squares problems.
We choose Gauss Newton methods since because it can take advantage of the Jacobian of $x(A)$, which is already computed as we compute $\nabla E(x(A))$.
Let $r(A) = (r_1(A), r_2(A), ..., r_{3m}(A)$

$$\begin{aligned}
\nabla E(A) &= \sum_{j=1}^{3m} r_j(A)\nabla r_j(A) = J(A)^T r(A) \\
\nabla^2 E(A) &= \sum_{j=1}^{3m} \nabla r_j(A)\nabla r_j(A)^T + \sum_{j=1}^{3m} r_j(A)\nabla^2 r_j(A) \\
&= J(A)^T J(A) + \sum_{j=1}^{3m} r_j(A)\nabla^2 r_j(A) \\
&\approx J(A)^T J(A)
\end{aligned}$$

As a result, in the Gauss Newton methods, we can take $J^T J$ as an approximation of the Hessian and the descent direction p_k^{GN} can be found as$-(J^T J)^{-1} J^T r_k$. Notice that $(J^T J)^{-1} J^T$ can be efficiently computed by using pseudo-inverse.

3.6 A Problem of the Data Set

Normally, when doing backtracking to find the step size, we would start with $\alpha = 1$. However, the data set we use describes angles in the unit of degrees rather than radians, which makes both $x(A)$ and $E(x(A))$ very flat. Since for a function $f(x)$ and scalar $k > 0$, $\nabla f(kx) = k\nabla f(x)$, $|\nabla f(kx)| = k|\nabla f(x)|$,this would make the gradient very flat as well. If we still use a step size of 1, the gradient descent method would be unacceptably and needlessly slow. Instead, if given a very large step size, like $\frac{180}{\pi}$, gradient descent would run must faster. After some testing, we eventually use $\frac{180}{\pi}$ for gradient descent to avoid running the algorithm for tens of minutes.

3.7 Implementation

All of the methods are implemented in C++ using Eigen, a high performance matrix library. Code related to model loading and rendering and GUI are using libigl and are from CSC418 A7[3]

4 Experiment

Two simple experiments are designed to determine the overall performance of each methods:

4.1 Incremental Target

In this experiment, we randomly generate a list of A's,
where each $A_{k+1} = \delta\xi + A_k, > 0, \xi \in [0, 1)^{3m}$
and each target position $x_{bk} = x(A_k)$.
This makes sure that each target position is not too far away from the previous one.

For the kth test, we start with initial angles of A_{k-1}. So that even if the method failed to find the solution at the $k-1$th iteration, it won't affect following tests.

This experiment mimics a scenario where the user is adjusting the target position in an interactive environment so that the target is gradually changed as the user dragging the end effectors around.

4.2 Random Target

In this experiment, we randomly generate a list of A's and each A_k is independent of each other. Each target position $x_{bk} = x(A_k)$.

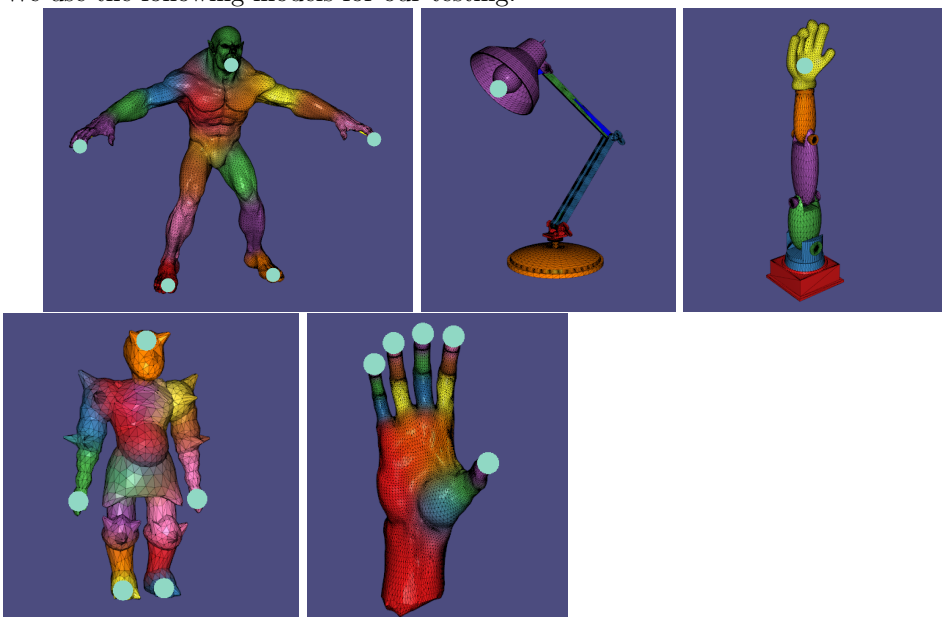
This experiment targets at the method's ability to find a solution accurately and quickly.

Notice in both experiements, we generate angle vector A instead of target end effector position because we want to make sure every target is reachable. For each experiement, we record the average time, iteration a method takes to find a solution A^*. We also record the average $E(x(A^*))_{avg}$, interquartile average $E(x(A^*))_{IQM}$ and maximum $E(x(A^*))$ to determine whether the method has found a solution or get stuck in a local minimum. $E(x(A^*))_{IQM}$ tells us the average performance when the method is able to find a solution.

5 Test and Results

5.1 Test Data

We use the following models for our testing:



From left to right and up to down: beast, ikea-lamp, robot-arm, knight, chimpanzee-hand

We run 1000 test samples for each experiment. However, when testing gradient descent methods, we only run 100 test samples since the method is very slow. For the 'beast' model, gradient tolerance is 0.01 and $E(x(A))$ tolerance is 0.002. For all other models, thoese tolerances are 0.0001 and 0.0002. The random data used for the test is identical across different methods.

Test Result

ikea-lamp(5 bones)

Incremental Target

Method	t_{avg}	$iter_{avg}$	$E(x(A^*))_{avg}$	$E(x(A^*))_{IQM}$	max $E(x(A^*))$
GD	0.2067	61.24	0.000190497	0.00019732	0.000199985
BFGS	0.0495268	7.1	0.000113054	0.000116104	0.000199923
GN	0.0427263	2.292	0.000105443	0.000100182	0.000199953

Random Target

Method	t_{avg}	$iter_{avg}$	$E(x(A^*))_{avg}$	$E(x(A^*))_{IQM}$	max $E(x(A^*))$
GD	0.484205	3146.33	0.00211125	0.000232641	0.103756
BFGS	0.059513	28.652	0.014172	0.000286993	0.744005
GN	0.045211	7.352	0.0124934	0.00014515	0.591626

robot-arm (7 bones)

Incremental Target

Method	t_{avg}	$iter_{avg}$	$E(x(A^*))_{avg}$	$E(x(A^*))_{IQM}$	max $E(x(A^*))$
GD	0.21928	55.47	0.000187641	0.000197819	0.000199957
BFGS	0.0507253	6.137	8.41677e-05	7.63603e-05	0.000199966
GN	0.0430753	2.351	0.0001019	9.57523e-05	0.000199331

Random Target

Method	t_{avg}	$iter_{avg}$	$E(x(A^*))_{avg}$	$E(x(A^*))_{IQM}$	max $E(x(A^*))$
GD	0.651476	3071.31	0.000552185	0.000331803	0.00368995
BFGS	0.0648178	26.039	0.00364678	0.000335149	0.0838877
GN	0.0486621	8.035	0.0283915	0.0203519	0.168183

knight (19 bones)

Incremental Target

Method	t_{avg}	$iter_{avg}$	$E(x(A^*))_{avg}$	$E(x(A^*))_{IQM}$	max $E(x(A^*))$
GD	0.38882	151.71	0.000197025	0.000199088	0.000199978
BFGS	0.11039	17	0.00015558	0.000164004	0.000359556
GN	0.0584754	3.816	0.000842643	0.000121646	0.346694

Random Target

Method	t_{avg}	$iter_{avg}$	$E(x(A^*))_{avg}$	$E(x(A^*))_{IQM}$	max $E(x(A^*))$
GD	1.00051	821.19	0.031724	0.0206912	0.177997
BFGS	0.463753	127.195	0.0264604	0.011391	0.334629
GN	0.101948	14.533	0.0626536	0.0455104	0.31967

chimpanzee-hand (21 bones)

Incremental Target

Method	t_{avg}	$iter_{avg}$	$E(x(A^*))_{avg}$	$E(x(A^*))_{IQM}$	max $E(x(A^*))$
GD	0.445665	175.18	0.000197199	0.000199246	0.000199964
BFGS	0.105163	13.653	0.000160643	0.000166628	0.000309528
GN	0.0498712	2.862	0.000107013	0.000101983	0.000199933

Random Target

Method	t_{avg}	$iter_{avg}$	$E(x(A^*))_{avg}$	$E(x(A^*))_{IQM}$	max $E(x(A^*))$
GD	1.00067	727.4	0.0126034	0.0116024	0.0307831
BFGS	0.400857	95.202	0.00407872	0.00147883	0.101507
GN	0.103962	17.514	0.000856469	0.00019759	0.0130153

beast (22 bones)

Incremental Target

Method	t_{avg}	$iter_{avg}$	$E(x(A^*))_{avg}$	$E(x(A^*))_{IQM}$	max $E(x(A^*))$
GD	0.919767	503.02	0.0118904	0.00605991	0.0822987
BFGS	0.14636	21.357	0.00110218	0.00104838	0.040058
GN	0.0977179	14.477	0.00134316	0.00133473	0.0283549

Random Target

Method	t_{avg}	$iter_{avg}$	$E(x(A^*))_{avg}$	$E(x(A^*))_{IQM}$	max $E(x(A^*))$
GD	0.987723	562.75	15.8306	1.15661	379.588
BFGS	0.323342	60.327	64.9053	0.606521	5838.137
GN	0.359758	96.757	1906.21	12.3538	35050.5

Analysis

First we can notice that gradient descent is the slowest among all the methods but given enough time it converges to a pretty good solution. Both BFGS and Gauss-Newton are very fast. There are a few things worth noticing:

1. For all data sets, Gauss Newton is the fastest to achieve a solution (having the least t_{avg} and least $iter_{avg}$) in incremental target test.

2. For random target tests on complex models (knight and beast), Gauss Newton has the highest $E(x(A^*))_{IQM}$ and very high max $E(x(A^*))$, suggesting it has difficulty in finding a solution. Althogh chimpanzee-hand has lots of bones, each bone has only very limited angle of movement.
   (1) and (2) suggests that Gauss Newton might perform better when the target position is close to the initial position.

3. GD is still very slow despite the fact it is given a large initial step size when doing backtrack. Notice in some data set t_{avg} is about 1, which means the algorithm has timed out in almost every test.

4. On complex models, BFGS has the lowest $E(x(A^*))_{IQM}$, suggests it is more robust against difficult configuration. However a large max $E(x(A^*))$ suggests it occasionally gets stuck at local minimum.

Interesting Fact Found in Interactive Demo

When the user moves a single end effector, gradient descent would most likely keep unaffected end effector still and only moves the affected one while BFGS would introduce change to all end effectors. Though both of them will reach the same solution, BFGS creates a little annoying wobbling to the model as it searches for the solution.

6 Conclusion

Gradient descent is the simplest method among them and is the lowest. Roughly 10x or times or even more slower than the others but at the end it is able to find a nice solution.
Gauss-Newton method is also much faster than gradient descent, but sometimes it fails to find the solution. Gauss-Newton perform better when initial target is near the target position.
BFGS is fast and robust for nearly all tests. Despite the fact that BFGS is has higher per-iteration cost than gradient descent, the benefit brought by quadratic rate of convergence far outweighs the cost.

7 References

[1] P. Calamai and J. Moré, "Projected gradient methods for linearly constrained problems," Mathematical Programming, vol. 39, no. 1, pp. 93–116, Sep. 1987.
[2] Numerical Optimization, Second Edition, Jorge Nocedal and Stephen J. Wright, Springer, 2006.
[3] CSC418 Assignment 7 https://github.com/alecjacobson/computer-graphics-kinematics