Jacobian Solutions to the Inverse Kinematics Problem

Mike Tabaczynski
Tufts University

Math 128 Fall 2005 Final Project
Submitted December 16, 2005
Updated January 17, 2006

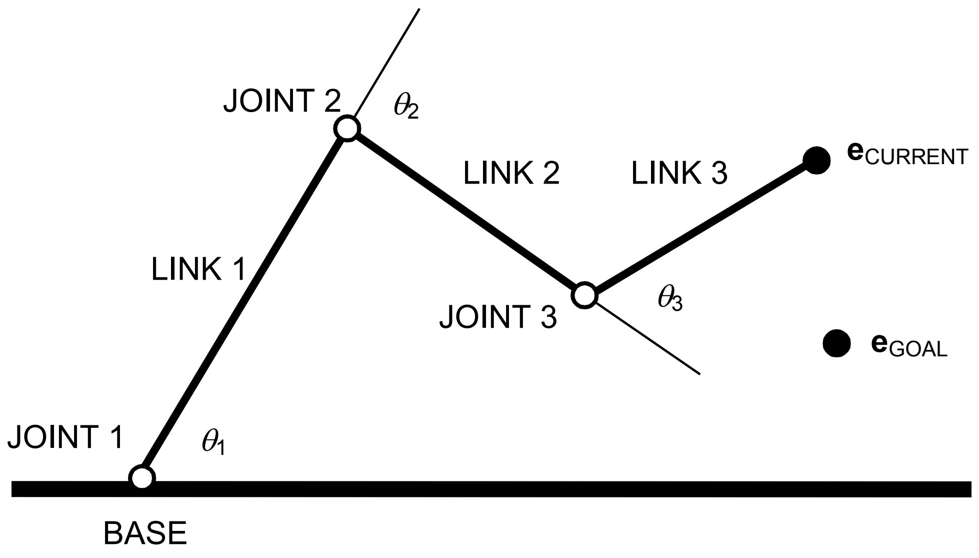
1 Introduction

1.1 Conventions and Notation

- Matrices are upper case italic.
- Vectors and points are lower case bold.
- Scalars are lower case italic or lower case Greek.
- $\| \ \|$ is the 2 (Euclidean) norm
- $\langle \mathbf{a},\mathbf{b} \rangle$ is the inner product of vectors **a** and **b**
- World position is position in absolute coordinates.
- Numbering convention root is the most fixed joint = zero, child joints numbered successively higher.

1.2 Problem Definition

A manipulator such as a robot arm or an animated graphics character is modeled as a *chain* composed of rigid *links* connected at their ends by rotating *joints*. The goal is to move the end of the chain to some point in space as smoothly, rapidly, and accurately as possible. [5]



The links are said to be hierarchical, so link 1 is the first link attached to a fixed base with "child" links numbered i+1, i+2, …, n successively attached to the chain. The "parent" of link i is link i-1. The mathematical model is each link i has an associated translation matrix T and variable rotation matrix R. Translation is from joint i to joint i+1 relative to the coordinate frame of parent link i-1 so is based on the effective length of link i (x = link length, y, z = 0 for a link whose rotation angle is defined relative to the line between its two pivots). Rotation is also relative to the coordinate frame of parent link i-1. Translation and rotation of joint 1 are relative to the fixed base, which is in the world coordinate frame, assumed at its origin. If in a real inverse kinetics problem it were not at the origin, a simple inexpensive translation can be applied to results to make it so.

Each link i has an associated transformation matrix $M_i = T(\mathbf{t}_i)R(\theta_i)$ where $\mathbf{t}_i$ is the translation vector from joint i to joint i+1 and θ_i is the angle of rotation of link i around the axis of rotation of joint i. The summary transformation between any 2 links i and j inclusive can be formed by concatenating the transforms M_i through M_j of the intervening links, so the position and orientation $\mathbf{e}$ of the end of a chain of n links for any state of translations and joints angles is given by the *forward kinematics* transformation matrix

$$M_e = M_1 M_2 \ldots M_n$$

The *inverse kinematics* (IK) problem is, given a goal position and a chain defined by $M_1 M_2 \ldots M_n$, calculate all the translations $\mathbf{t}_i$ and joints angles θ_i to move the end of the chain to that goal. [1]

1.3 Snags

Redundancy At most 6 degrees of freedom (DOF) are required for a chain to reach a goal (x, y, z position and 3 angles of end orientation). A typical chain with many links each having 6 DOF has more DOF than required to reach a goal so is an underdetermined system having multiple or an infinite number of solutions. A simple example is a 2 dimensional chain having 3 joints needs 2 DOF to reach a goal (x, y) but has 3 joint angles of rotation.

Unreachable targets The target can be farther than the chain can reach or can be at a point close to the base where no pivoting of links can bend the chain to reach.

Singularities occur when no change in joint angle can achieve a desired change in chain end position, like at a fully outstretched chain, resulting in solution algorithms demanding excessively large angle changes.

In summary, the IK problem requires solving an underdetermined nonlinear system that can have singularities. Exact solution methods are considered impractical, so the problem is approximated as a linear system and solved through iterative methods.

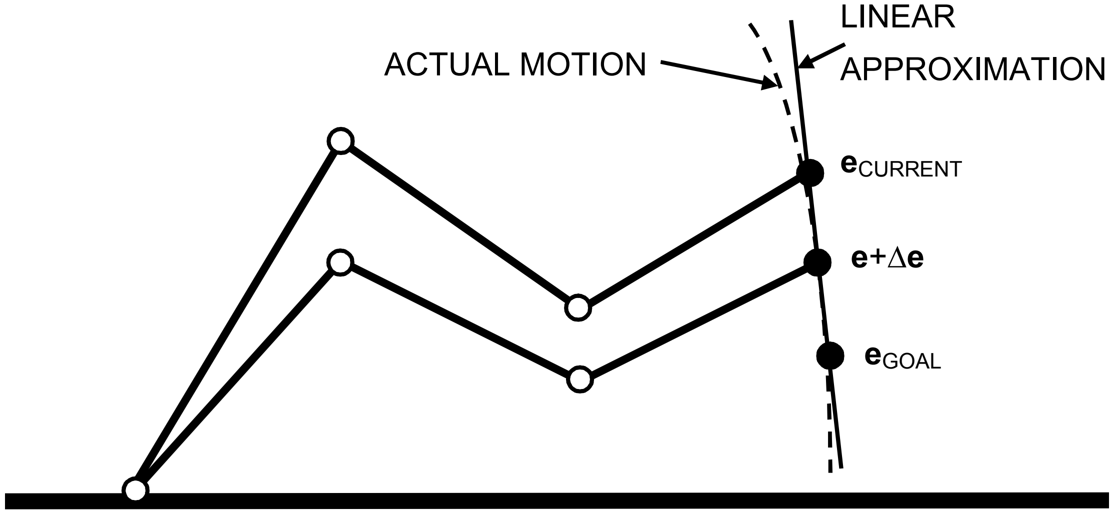
1.4 Objective

Implement, evaluate, and compare the Jacobian based solutions of the IK problem: pseudo inverse, truncated pseudo-inverse, transpose, and damped least squares (DLS).

2 Solutions

2.1 The Jacobian Matrix

Jacobian solutions are a linear approximation of the IK problem. [2][5]


LINEAR APPROXIMATION
ACTUAL MOTION
$\mathbf{e}_{\text{CURRENT}}$
$\mathbf{e}+\Delta\mathbf{e}$
$\mathbf{e}_{\text{GOAL}}$

In the interest of simplifying this solution, the translation matrices $T(\mathbf{t}_i)$ will be fixed and the orientation of chain end **e** will not be considered.

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \cdots \\ \theta_n \end{bmatrix}$$

is the set of rotation angles of the entire chain where θ_i is the rotation angle of joint i relative to joint i-1 or relative to the root if $i = 1$.

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

is the world position of the end of the chain. The forward kinematic solution is

$$\mathbf{e} = f(\boldsymbol{\theta}).$$

The Jacobian J is a matrix of partial derivatives of the entire chain system relative to **e**. It linearly models chain end movement (how **e** changes) relative to instantaneous system changes in link translation and joint angle. [4][5]

$$\dot{\mathbf{e}} = J\dot{\boldsymbol{\theta}}$$

$$J \equiv \frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}}$$

$$J = \begin{bmatrix} \frac{\partial e_x}{\partial \theta_1} & \frac{\partial e_x}{\partial \theta_2} & \cdots & \frac{\partial e_x}{\partial \theta_n} \\ \frac{\partial e_y}{\partial \theta_1} & \frac{\partial e_y}{\partial \theta_2} & \cdots & \frac{\partial e_y}{\partial \theta_n} \\ \frac{\partial e_z}{\partial \theta_1} & \frac{\partial e_z}{\partial \theta_2} & \cdots & \frac{\partial e_z}{\partial \theta_n} \end{bmatrix}$$

J_i, the ith column of the Jacobian, gives the velocity of **e** relative to a change in any joint angle θ_i. The inverse kinematic solution is

$$\boldsymbol{\theta} = f^{-1}(\mathbf{e})$$

so

$$\dot{\boldsymbol{\theta}} = J^{-1}\dot{\mathbf{e}}$$

This leads to an iterative solution to the IK problem. $\mathbf{e}_{\text{current}}$ and $\mathbf{e}_{\text{goal}}$ are known.

$$\Delta\mathbf{e} = \mathbf{e}_{\text{goal}} - \mathbf{e}_{\text{current}}$$
$$\Delta\boldsymbol{\theta} = J^{-1}\Delta\mathbf{e}$$
$$\boldsymbol{\theta}_{\text{current}} = \boldsymbol{\theta}_{\text{previous}} + \Delta\boldsymbol{\theta}$$

which, if $\Delta\mathbf{e}$ is limited to a small step, eventually converges $\mathbf{e}_{\text{current}}$ to $\mathbf{e}_{\text{goal}}$. [2]

Although f and consequently J are theoretically not guaranteed to always be invertible, in practice, a physical chain will never be exactly in a configuration that results in a singularity. The performance of IK problem solutions when a chain is near a singularity configuration varies widely.

The normal equation and SVD of J can be used to solve for $\Delta\boldsymbol{\theta}$:

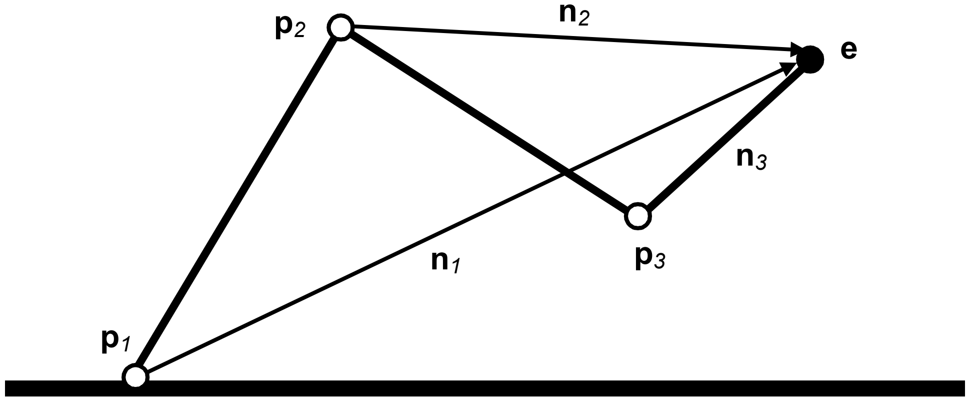
$$J\Delta\boldsymbol{\theta} = \Delta\mathbf{e}$$

$$J^T J\Delta\boldsymbol{\theta} = J^T\Delta\mathbf{e}$$

$$VS^2V^T\Delta\boldsymbol{\theta} = J^T\Delta\mathbf{e}$$

$\Delta\theta = VS^{-2}V^T J^T \Delta\mathbf{e}$

To compute J_i, the ith column of the Jacobian (there are alternate ways to do this):



$\mathbf{z}_i$ is a unit vector parallel to the axis of rotation in link coordinates of joint i, pointing into the page.
$\mathbf{p}_i$ is the world position of joint i.
$\mathbf{n}_i = \mathbf{e} - \mathbf{p}_i$ is the vector from joint i to the end of the chain.
$J_i = \mathbf{z}_i \times \mathbf{n}_i$. [5]

If links were allowed to have variable translation (all 6 DOF), $\mathbf{z}_i$ would be appended to the bottom of J_i. [1] If the chain were allowed to have multiple branches (chain ends), those ends would also add more rows to J. The dimensions of J are always rows m = DOF of the goal and columns n = total DOF in the chain.

Algorithm common to all methods

1. Compute J.
2. Compute $\Delta\theta$.
3. $\theta := \theta + \Delta\theta$.
4. Update the joint and end positions based on new θ.
5. Repeat until $\mathbf{e}_{\text{current}}$ is within tolerance of $\mathbf{e}_{\text{goal}}$ or iteration count exhausted.

2.2 Pseudo Inverse

Inverting J explicitly or implicitly in the above proposed solution is expensive and redundancy (underdetermination) in the system means J is almost always short and wide and therefore not invertible. A popular resolution of this issue is to use the pseudo inverse

$J^+ = (J^T J)^{-1} J^T$

in place of J^{-1}.

$\Delta\theta = J^+ \Delta\mathbf{e}$

J^+ exists even when J is not square and full rank. [6] Although not as mathematically correct as using the inverse (when it exists), this method tends to converge on the solution and error can be reduced by taking small steps per iteration.

The pseudo inverse can be calculated using the reduced singular value decomposition (SVD)

$$J = USV^{\mathrm{T}}$$

$$J^{+} = VS^{-1}U^{T}$$

S is diagonal and can be inverted rapidly. Further computation time may be saved by computing

$$J^{+} = \sum_{i=1}^{r} \frac{1}{\sigma_i} \mathbf{v}_i \mathbf{u}_i^T$$

where r is the rank of J, which is effectively one scalar/matrix and one matrix/matrix multiply instead of 2 matrix/matrix multiplies. [7] A few more flops can be saved by computing $\Delta\boldsymbol{\theta}$ directly in the above summation. [8]

$$\Delta\boldsymbol{\theta} = \sum_{i=1}^{r} \frac{1}{\sigma_i} \mathbf{v}_i \mathbf{u}_i^T \Delta\mathbf{e}$$

J^{+} can also be computed more rapidly without the SVD or any inversion by solving the normal equation

$$J^T J \Delta\boldsymbol{\theta} = J^T \Delta\mathbf{e}$$

for $\Delta\boldsymbol{\theta}$ using Cholesky factorization ($J^T J$ is symmetric positive definite). This method has disadvantages: It does not support the truncated SVD (see 2.3) and is sensitive to J being ill-conditioned.

Algorithm to compute $\Delta\boldsymbol{\theta}$ using pseudo inverse

1. Compute J^{+}.
2. Compute $\Delta\mathbf{e} = \mathbf{e}_{\text{goal}} - \mathbf{e}_{\text{current}}$.
3. Compute error = $\|(I - J^{+}J)\Delta\mathbf{e}\|$.
4. If error > tolerance, $\Delta\mathbf{e} := \Delta\mathbf{e}/2$ and repeat 3.
5. $\Delta\boldsymbol{\theta} = J^{+}\Delta\mathbf{e}$.

2.3 Truncated Pseudo Inverse

The concept of the truncated pseudo inverse solution is simple. Problems introduced by singularities in the system are manifest as singular values equal to or near zero, so simply omit the components of the solution corresponding to those very small or zero singular values. These components cause the excessive joint angle swings that inhibit convergence. The components corresponding to the larger singular values, which are retained in the solution, contribute the most toward smooth predictable convergence. This is easy to do in the J^{+} or $\Delta\mathbf{e}$ summations given above:

$$J^{+} = \sum_{i=1}^{k} \frac{1}{\sigma_i} \mathbf{v}_i \mathbf{u}_i^T \text{ or } \Delta\boldsymbol{\theta} = \sum_{i=1}^{k} \frac{1}{\sigma_i} \mathbf{v}_i \mathbf{u}_i^T \Delta\mathbf{e}$$

where $k \le r$ where r is the rank of J and σ_k is the smallest singular value that will contribute to the solution, determined by some threshold.

The behavior of truncated pseudo inverse is claimed to be similar to that of damped least squares. [8]

2.4 Transpose

The transpose method simply replaces the pseudo inverse of *J* with the transpose of *J,* the logic being that $\Delta\theta = \alpha J^T \Delta\mathbf{e}$ converges toward a solution very slowly but, unlike J^{-1}, J^{+}, and the SVD, has effectively zero computation cost making each iteration of the algorithm much shorter. The motion produced by the transpose method is claimed to be desirable because it closely matches the physics based model of the user pulling the end of the chain with an elastic band while other IK solutions produce successive chain configurations that can be dramatically different from each other resulting in jerky, unnatural motion.

The constant α is chosen to minimize the new value of $\Delta\mathbf{e}$ after the system is updated with the newly computed joint angles. [5][6][7]

$$\alpha = \frac{\left\langle \Delta\mathbf{e}, JJ^T \Delta\mathbf{e} \right\rangle}{\left\langle JJ^T \Delta\mathbf{e}, JJ^T \Delta\mathbf{e} \right\rangle}$$

2.5 Damped least squares (DLS)

The pseudo inverse and transpose methods are vulnerable to system singularities, reductions in rank of the Jacobian matrix to less than full. Those methods tend to oscillate at high amplitude near singularities as the algorithms try in vain to converge on unreachable targets. The DLS method compensates for these singularity problems by introducing a *damping constant λ*.

2.5.1 Algorithm

DLS finds the value of $\Delta\boldsymbol{\theta}$ that minimizes

$$\left\| J\Delta\boldsymbol{\theta} - \Delta\mathbf{e} \right\|^2 + \lambda^2 \left\| \Delta\boldsymbol{\theta} \right\|^2 \text{ (tracking error + joint velocities)}$$

$$= \left\| J\Delta\boldsymbol{\theta} - \Delta\mathbf{e} \right\|^2 + \left\| \lambda\Delta\boldsymbol{\theta} \right\|^2$$

$$= \left\| \begin{bmatrix} J\Delta\boldsymbol{\theta} - \Delta\mathbf{e} \\ \lambda\Delta\boldsymbol{\theta} \end{bmatrix} \right\|^2$$

$$= \left\| \begin{bmatrix} J\Delta\boldsymbol{\theta} - \Delta\mathbf{e} \\ \lambda\Delta\boldsymbol{\theta} - \mathbf{0} \end{bmatrix} \right\|^2$$

$$= \left\| \begin{bmatrix} J \\ \lambda I \end{bmatrix} \Delta\boldsymbol{\theta} - \begin{bmatrix} \Delta\mathbf{e} \\ \mathbf{0} \end{bmatrix} \right\|^2$$

Which is equivalent to minimizing

$$\left\| \begin{bmatrix} J \\ \lambda I \end{bmatrix} \Delta\boldsymbol{\theta} - \begin{bmatrix} \Delta\mathbf{e} \\ \mathbf{0} \end{bmatrix} \right\|$$

The corresponding normal equation is

$$\begin{bmatrix} J \\ \lambda I \end{bmatrix}^T \begin{bmatrix} J \\ \lambda I \end{bmatrix} \Delta\boldsymbol{\theta} = \begin{bmatrix} J \\ \lambda I \end{bmatrix}^T \begin{bmatrix} \Delta\mathbf{e} \\ \mathbf{0} \end{bmatrix}$$

$$\begin{bmatrix} J^T & \lambda I \end{bmatrix} \begin{bmatrix} J \\ \lambda I \end{bmatrix} \Delta\boldsymbol{\theta} = \begin{bmatrix} J^T & \lambda I \end{bmatrix} \begin{bmatrix} \Delta\mathbf{e} \\ \mathbf{0} \end{bmatrix}$$

$$(J^T J + \lambda^2 I)\Delta\boldsymbol{\theta} = J^T \Delta\mathbf{e}$$

J is m by n and λI is n by n, so $\begin{bmatrix} J \\ \lambda I \end{bmatrix}$ is $m+n$ by n (n is its minimum dimension). λI is of rank n because it is diagonal and so must have n linearly independent columns. λI makes the rank of $\begin{bmatrix} J \\ \lambda I \end{bmatrix} = n$, which means it has full rank because n = its minimum dimension. If $\begin{bmatrix} J \\ \lambda I \end{bmatrix}$ is full rank then so is $\begin{bmatrix} J \\ \lambda I \end{bmatrix}^T$ and $\begin{bmatrix} J \\ \lambda I \end{bmatrix}^T \begin{bmatrix} J \\ \lambda I \end{bmatrix}$, which = $(J^T J + \lambda^2 I)$, which in turn must be invertible. The normal equation and solution for $\Delta\boldsymbol{\theta}$ is then

$$\Delta\boldsymbol{\theta} = (J^T J + \lambda^2 I)^{-1} J^T \Delta\mathbf{e}$$

$(J^T J + \lambda^2 I)^{-1} J^T = J^T (JJ^T + \lambda^2 I)^{-1}$ so the above solution can be rewritten as

$$\Delta\boldsymbol{\theta} = J^T (JJ^T + \lambda^2 I)^{-1} \Delta\mathbf{e}$$

which, in typically IK problems, requires inversion of a matrix of much lower dimension. In terms of the SVD of J

$$J^T (JJ^T + \lambda^2 I)^{-1} = \sum_{i=1}^{r} \frac{\sigma_i}{\sigma_i^2 + \lambda^2} \mathbf{v}_i \mathbf{u}_i^T$$

where σ_i, $\mathbf{u}_i$, $\mathbf{v}_i$ are singular values and column vectors from the SVD of J [4][6]. After substitution, the final DLS solution is

$$\Delta\boldsymbol{\theta} = \left(\sum_{i=1}^{r} \frac{\sigma_i}{\sigma_i^2 + \lambda^2} \mathbf{v}_i \mathbf{u}_i^T \right) \Delta\mathbf{e}$$

DLS works because when the system is near a singularity and a σ_i approaches zero, the λ dominates the term

$$\frac{\sigma_i}{\sigma_i^2 + \lambda^2}$$

in the equation above preventing the sum from growing excessively as it would in the analogous sum in the pseudo inverse method where the same term is

$$\frac{1}{\sigma_i}$$

To save computation time over the SVD based equation above, $\Delta\boldsymbol{\theta} = J^T(JJ^T + \lambda^2 I)^{-1}\Delta\mathbf{e}$ can be solved by using Cholesky factorization ($JJ^T + \lambda^2 I$ is symmetric) to solve $(JJ^T + \lambda^2 I)\mathbf{z} = \Delta\mathbf{e}$ for **z** and substituting it into $\Delta\boldsymbol{\theta} = J^T\mathbf{z}$. [6][8]

2.5.2 Determining λ

λ must be large enough to restrain angular velocity near singularities but small enough to allow rapid convergence. [6] There are several ways to find the optimal value. [8]

Maximums Given the singular values σ_i of *J*, there are relatively simple formulas to compute λ based on conditions imposed on the maximum allowable values of $\Delta\boldsymbol{\theta}$, $\Delta\mathbf{e}$, or condition number.

Minimum singular value λ can be computed via a somewhat more complicated way based on the minimum singular value of *J*.

Numerical filtering This method uses different damping constants for each singular value rather than a single damping constant for the entire system. Higher damping constants are applied to small singular value that are likely to cause oscillation but not contribute significantly to convergence. Lower damping constants are applied to larger singular values that contribute significantly to convergence and won't cause oscillation. The effect is to lower the overall damping constant applied to the system so near singularity effects are reduced without appreciably slowing convergence. The result is claimed to be similar to the truncated pseudo inverse method.

2.5.3 Moving the Target Closer

When converging to unreachable targets, the Jacobian solution system gets close to a singularity as the chain attempts to stretch to reach the target. Even methods like the pseudo inverse that are well behaved at a singularity are unstable near that singularity [6] and will oscillate the chain trying to converge [7]. To reduce this problem, the target can effectively be "moved" closer to the chain end by modifying $\Delta\mathbf{e}$ as follows:

$$\Delta\mathbf{e} := \begin{cases} \Delta\mathbf{e} & \text{if } \|\Delta\mathbf{e}\| \le d_{\max} \\ d_{\max}\dfrac{\Delta\mathbf{e}}{\|\Delta\mathbf{e}\|} & \text{otherwise} \end{cases}$$

[7] recommends setting $d_{\max}$ several times larger than the chain end typically moves in a single step, or half the length of a typical link. [7] used 0.7 in experiments

Algorithm to compute $\Delta\boldsymbol{\theta}$ using DLS

1. Compute SVD.
2. Compute $\Delta\mathbf{e} = \mathbf{e}_{\text{goal}} - \mathbf{e}_{\text{current}}$.

3. Modify $\Delta\mathbf{e}$ as described above
4. Compute λ if not fixed
5. $\Delta\boldsymbol{\theta} = \left(\sum_{i=1}^{r} \frac{\sigma_i}{\sigma_i^2 + \lambda^2} \mathbf{v}_i \mathbf{u}_i^T \right) \Delta\mathbf{e}$.

3 Implementation

3.1 Simplifications and Limitations

The model is only 2D with simple hinged joints (1 degree of freedom per joint). Only a single chain end is supported. Goals are position only, desired chain end orientation is not considered. Constraints such as joint rotation limits or self collisions are not handled. Scaling transforms are not supported and the solution does not attempt to minimize the "energy" used to move the links.

3.2 All Algorithms

All implementations were written and tested in MATLAB 7.0.1.15.

Excessively large angle swings are limited by scaling $\Delta\boldsymbol{\theta}$ so its norm is within a constant value. Experimentally, there was not a direct relationship between this value and convergence. Very small values would never converge as would very large values, like π. Values around $\pi/4$ worked the best. In cases where the chain end started about a quadrant away from the target, convergence was significantly faster than without a limit on $\Delta\boldsymbol{\theta}$. Where the chain end started opposite the target, the solution converged about as fast as without a limit, but with smoother motion. As expected, limiting $\Delta\boldsymbol{\theta}$ effectively controlled oscillation amplitude of some methods when converging on unreachable targets.

3.3 Truncated Pseudo Inverse

The pseudo inverse algorithm implementation is based on the SVD, not the Cholesky factorization. This algorithm needs to pay attention to system rank because the summation of $J^{+}\Delta\boldsymbol{\theta}$ fails at singularities due to divide by zero, so a non-truncated pseudo inverse implementation is not really practical. A value of 0.0001 was arbitrarily chosen as the threshold for zero singular values.

Step 3 of the algorithm given in 2.2 that limits the size of $\Delta\mathbf{e}$ was not implemented.

```
% No error checking
% deLimit not used in this version
function [dTheta,rank_used] = Calc_dTheta_PseudoInverse_SVD(end_goal, e, J,
singularTol, deLimit)

[U,S,V] = svd(J,0);

% Simple version
% [U,S,V] = svd(J,'econ');
% Sinv = diag(1./diag(S));
% Jplus = V*Sinv*U';

[m,n] = size(J);
```

```
dTheta = zeros(n,1);
de = end_goal - e;

max_rank = min([m n]);

% Sum the product components of the pseudo inverse
for i=1:max_rank
    % Stop summing columns whose corresponding singular value is below a
    % threshhold. Zero eliminates singularities, > zero eliminates near
    % singularities.
    if (S(i,i) <= singularTol)
        % If breaking, this singular value's component did not contribute
        i = i - 1;
        break;
    end
    dTheta = dTheta + (V(:,i)*U(:,i)')*de/S(i,i);
end

dTheta = transpose(dTheta);
rank_used = i;
```

3.4 Transpose

The transpose method is simple and is implemented as described in 2.4.

```
% No error checking
% deLimit not used in this version
function [dTheta] = Calc_dTheta_Transpose(end_goal, e, J, deLimit)

de = end_goal - e;

% Compute alpha constant as quotient of 2 inner products as suggested by
% Buss & Kim, to minimze de after caller updates theta
JJTde = J*J'*de;
alpha = (de'*JJTde)/(JJTde'*JJTde);

dTheta = transpose(alpha * J' * de);
```

3.5 Damped least squares (DLS)

Like the pseudo inverse algorithm, DLS was implemented based on the SVD, not the Cholesky factorization. A fixed value was used for λ rather than a computed one.

```
% No error checking
function [dTheta] = Calc_dTheta_PseudoInverse_DLS(end_goal, e, J, lambda,
deLimit)

[U,S,V] = svd(J,0);

[m,n] = size(J);
dTheta = zeros(n,1);

de = end_goal - e;
% To reduce oscillation when seeking unreachable targets, "move" the target
% closer
norm_de = norm(de);
if (norm_de > deLimit)
   de = deLimit*de/norm_de;
end

max_rank = min([m n]);
```

```
% Sum the product components of the pseudo inverse with a damping constant
for i=1:max_rank
    dTheta = dTheta + S(i,i)*(V(:,i)*U(:,i)')*de/(S(i,i)*S(i,i)+lambda);
end

dTheta = transpose(dTheta);
```

3.6 Test Application

The normal entry point to the test application is the function

`[final_error, iterations] = IK_Jacob_file(filename, max_iterations)`

which loads a MATLIB file of the user's choice. The file must have been previous saved from MATLIB with the variables described in the comments at the top of the `IK_Jacob_file` function file. This provides an easy way to maintain a collection of named test scenarios. `IK_Jacob_file` calls `IK_Jacob`, which is the meat of the IK solver. `IK_Jacob` in turn calls one of the functions corresponding to the algorithms described above, selected by commenting and uncommenting lines 94 through 96.

For details on input, see the comments at the top of the `IK_Jacob_file` and `IK_Jacob` files.

Output from the application consists of per iteration text output of the current iteration number, the actual rank used in the algorithm (relevant to truncated pseudo inverse only), the magnitude of the error between the current chain end and the target position, and the contents of the current $\Delta\theta$ (dTheta) array. Error magnitude is the best gauge of how the algorithm is progressing. The final 3D error and iteration count values returned by `IK_Jacob_file` and `IK_Jacob` tell how well the goal was actually achieved.

The application plots its incremental results as lines representing the chain in its current configuration. The fixed base (world (0,0,0)) is in the center of the grid. A small blue square marks the goal. Each segment of the line corresponds to a link and the intersecting end points of the segments are the joints. The color of the lines is ramped over time from red to green, so the plot looks like a colored stroboscopic photo of a very skinny robot arm. The color fade helps differentiate the configurations created as the algorithm progresses, and will be more useful if the iteration limit is set just high enough to converge.

Joint position versus time can be plotted by uncommenting line 115. The path of each joint in the chain is plotted as a line in a color chosen by MATLIB. To unclutter the view, the 4 plot statements just above line 115 may need to be commented out.

IK_Jacob_file

```
% Loads a canned test scenario from filename and uses the Jacobian matrix
% to solve the Inverse Kinematics problem defined by the variables in that
% file.
%
% Args:
%   filename    input file containing the variables listed below
%   max_iterations how many times to iterate the IK solver before
%               terminating if a tolerance is not met
%
% Variables expected to be in the input file for a chain of n links:
%   For scenarios defined by translation and angle:
```

```
%   translate   3 by n matrix of x, y, and z values representing the
%               starting translation vector of each link relative to the
%               link's local coordinate system
%   theta       1 by n vector of the starting rotation angle in radians of
%               eachjoint relative to the parent link's local coordinate
%               system
%   XOR For scenarios defined by link end positions:
%   position    3 by n vector of the starting x, y, and z positions of
%               the end of each link in the chain in world coordinates
%   For all scenarios:
%   end_goal    3 by 1 vector specifying the target x, y, and z position
%               for the end of the chain in world coordinates
%   tolerance   scalar representing the norm of the vector of the
%               difference between the position of the chain end and
%               end_goal below which IK iteration terminates
%
% Returns:
%   final_error vector of the best difference between the position of the
%               chain end and end_goal
%   iterations  total iterations executed by the IK solver
%
% Comments: Jacobian IK solutions naturally operate on chains whose
% characteristics and state are defined by translation vectors and
% rotation angles. The option to operate on chains defined by their
% link end positions is provided because creation of test chains that way
% is more natural to humans than trying to figure link lengths and angles.
% All chains are converted to translation vector/rotation angle data
% before the IK solver is applied to them.
function [final_error, iterations] = IK_Jacob_file(filename, max_iterations)

load(filename);

if (exist('position','var'))
    % Scenario in file is defined by link end positions
    disp ('Scenarios defined by link end positions not implemented')
    [final_error] = [-1];
    return;
end

[final_error, iterations] = IK_Jacob(translate, theta, end_goal, tolerance,
max_iterations);
```

IK_Jacob

```
% Uses the Jacobian matrix to solve an Inverse Kinematics problem.
%
% Args for a chain of n links:
%   translate   3 by n matrix of x, y, and z values representing the
%               starting translation vector of each link relative to the
%               link's local coordinate system
%   theta       1 by n vector of the starting rotation angle in radians of
%               eachjoint relative to the parent link's local coordinate
%               system
%   end_goal    3 by 1 vector specifying the target x, y, and z position
%               for the end of the chain in world coordinates
%   tolerance   scalar representing the norm of the vector of the
%               difference between the position of the chain end and
%               end_goal below which IK iteration terminates
%   max_iterations how many times to iterate the IK solver before
%               terminating if tolerance is not met
%
% Returns:
```

```
%   error       vector of the best difference between the position of the
%               chain end and end_goal
%   iterations  total iterations executed by the IK solver
function [error, iterations] = IK_Jacob(translate, theta, end_goal, tolerance,
max_iterations)

% Constants - used to a varying degree to control algorithm behavior
% Components corresponding to singular values smaller than this are ignored
% in the solution
singularTol = 0.0001;
% Limit applied to delta before delta theta is computed
% Buss & Kim suggested 0.7
deLimit = 0.7;
% Algorithms routinely produce angle changes like 8pi. This limits them.
dThetaLimit = pi/4;
% Buss & Kim suggested 1.1 for lambda
lambda = 1.1;

[m, n, result] = validate(translate, theta, end_goal, tolerance,
max_iterations);
if (result ~= 0)
    final_error = [-1];
    return;
end

% Pre-allocate link end position matrix, rows are x, y, z
position = zeros(3,n);
% Create a constant matrix of the joint rotation axis vectors. In a 3D
% version, this would be a characteristic of the chain and would be passed
% in to this function
r_axis = zeros(3,n);
% In 2D, all joint axes are the z axis, pointing away from the viewer
r_axis(3,:) = 1;

% DEBUG
plot (end_goal(1), end_goal(2), 's','Color', [0 0 1]);
axis square
axis on
axis([-3, 3, -3, 3])
hold all;
disp('       k    rank used  normError  dTheta ----------->');
joint_motion = zeros(2*(n+1), max_iterations);

for k = 1:max_iterations
    % Compute the position of every link end, save for computation of
    % the Jacobian. Start at the joint between link 1 and the base (world
    % origin)
    originAffine = [0; 0; 0; 1];
    eAffine = originAffine;
    M = eye(4);

    for i = 1:n
        position(:,i) = eAffine(1:3);

        %DEBUG
        joint_motion(2*i-1:2*i,k) = position(1:2,i);

        % 3D Affine transform matrix corresponding to link
        M = M * LinkTransform(translate(:,i), theta(1,i));
        eAffine = M * originAffine;
    end
    % eAffine contains the position of the chain end
    e = eAffine(1:3);
    %DEBUG
    joint_motion(2*(n+1)-1:2*(n+1),k) = e(1:2);

    % Terminate if chain end is within tolerance of the target position
    error = e - end_goal;
    normError = norm(error);
    if (normError <= tolerance)
        break;
    end

    J = Jacobian(position, r_axis, e);

    rank_used = -1;
%    [dTheta,rank_used] = Calc_dTheta_PseudoInverse_SVD(end_goal, e, J,
singularTol, deLimit);
    [dTheta] = Calc_dTheta_PseudoInverse_DLS(end_goal, e, J, lambda, deLimit);
%    [dTheta] = Calc_dTheta_Transpose(end_goal, e, J, deLimit);

    % Scale delta theta to within reasonable limit
    scale = min(1, dThetaLimit/max(abs(dTheta)));
    dTheta = scale * dTheta;

    % DEBUG
    disp([k rank_used normError dTheta])
    plot (position(1,:), position(2,:), 'Color', [1-k/max_iterations
k/max_iterations 0]);
    plot ([position(1,n),e(1)], [position(2,n),e(2)], 'Color', [1-
k/max_iterations k/max_iterations 0]);

    theta = theta + dTheta;
end

% DEBUG
disp([k rank_used normError dTheta])
plot (position(1,:), position(2,:), 'Color', [1-k/max_iterations
k/max_iterations 0]);
plot ([position(1,n),e(1)], [position(2,n),e(2)], 'Color', [1-k/max_iterations
k/max_iterations 0]);
for i=2:n+1
%    plot (joint_motion(2*i-1,1:k), joint_motion(2*i,1:k));
end
hold off;

iterations = k;
```

validate

```
function [m, n, result] = validate(translate, theta, end_goal, tolerance,
max_iterations)

result = 0;

[m, n] = size(translate);
[m_theta, n_theta] = size(theta);
[m_end_goal, n_end_goal] = size(end_goal);

if (m ~= 3)
    disp ('translate matrix must have 3 rows')
    result = -1;
end
```

```
if (m_theta ~= 1)
    disp ('theta matrix must have 1 row')
    result = -1;
end

if (n ~= n_theta)
    disp ('Number of columns in translate and theta matrices are not equal')
    result = -1;
end

if (m_end_goal ~= m)
    disp (['end_goal matrix must have ', int2str(m), ' rows'])
    result = -1;
end

if (m_end_goal ~= m || n_end_goal ~= 1)
    disp ('end_goal matrix must have 1 column')
    result = -1;
end

if (max_iterations < 1)
    disp ('Invalid iteration limit')
    result = -1;
end

if (tolerance < 0)
    disp ('Tolerance must be positive')
    result = -1;
end
```

<u>LinkTransform</u>

```
% Compute 3D affine transform matrix corresponding to link having the
% given translation vector and rotation angle
% theta is in radians
% No error checking
function M = LinkTransform(translate, theta)

% Compute M = T(translate)R(theta)
% Only 2D rotation around z axis is implemented in this version

% Rotation
R = eye(4);
R(1,1) = cos(theta);
R(2,2) = R(1,1);
R(2,1) = sin(theta);
R(1,2) = -R(2,1);

% Translation. The translation must be rotated to the coordinate frame
% of the parent link before being applied to the link's summary transform
translate_affine = [translate; 1];
translate_affine = R * translate_affine;
T = eye(4);
T(1:3,4) = translate_affine(1:3);

M = T * R;
```

4 Results

The results for the methods tested are summarized and compared in a table in 4.4.

4.1 Truncated Pseudo Inverse

SVD based pseudo inverse failed at singularities due to divide by zero, so only the truncated pseudo inverse was tested. Without truncating the SVD, the pseudo inverse method behavior generally involved the chain appearing to flail for a while until it happened upon a "comfortable" configuration from which it could acquire the target in a stable way and finally settle on a smoother, slower trajectory to target. Even with the truncated SVD, angle swing limits were still needed reduce excessive joint motion and maintain relatively smooth trajectories. In practice the algorithm always truncated the rank from 3 to use only the 2 largest singular values, as shown in the output below. This non-coincidently = DOF of the IK solver implementation.

```
EDU>> [final_error, iterations] = IK_Jacob_file('5_link', 8)
       k    rank used  normError  dTheta ----------->
    1.0000    2.0000    3.1772   -0.2548   -0.3650   -0.4744   -0.2963
0.1077
    2.0000    2.0000    2.2797    0.5740    0.5341   -0.0189   -0.3098
0.2384
    3.0000    2.0000    1.3181    0.0453    0.2095   -0.3655   -0.3570
0.3858
    4.0000    2.0000    0.2902    0.1706    0.0340   -0.1028   -0.0130
0.2009
    5.0000    2.0000    0.0388   -0.0047   -0.0036   -0.0015   -0.0017   -
0.0028
    6.0000    2.0000    0.0002    0.0001   -0.0000   -0.0001   -0.0000
0.0002
    7.0000    2.0000    0.0000    0.0001   -0.0000   -0.0001   -0.0000
0.0002
final_error =
  1.0e-007 *
   -0.1211
    0.2040
         0
iterations =
     7
```

The truncated pseudo inverse method was incapable of converging on a target that was unreachable because it was too close (Figure 1), and was characterized by the jerky motion depicted in Figure 2 and Figure 3.

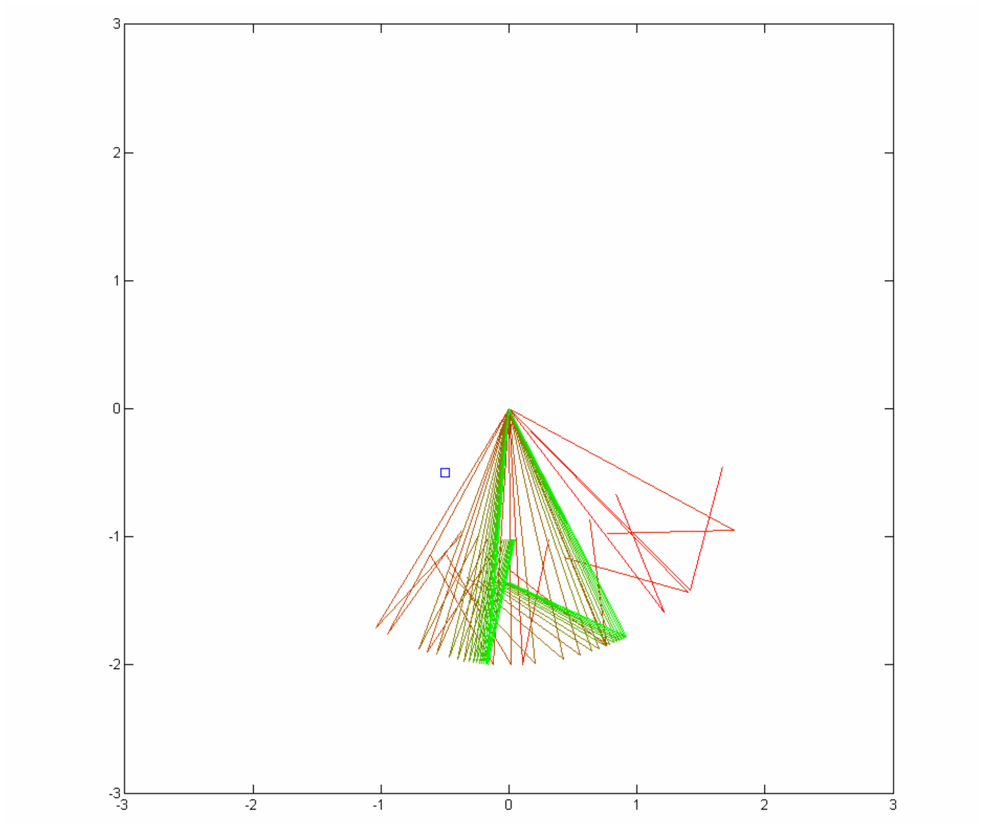


Figure 1: Unreachable target (too close) with truncated pseudo inverse

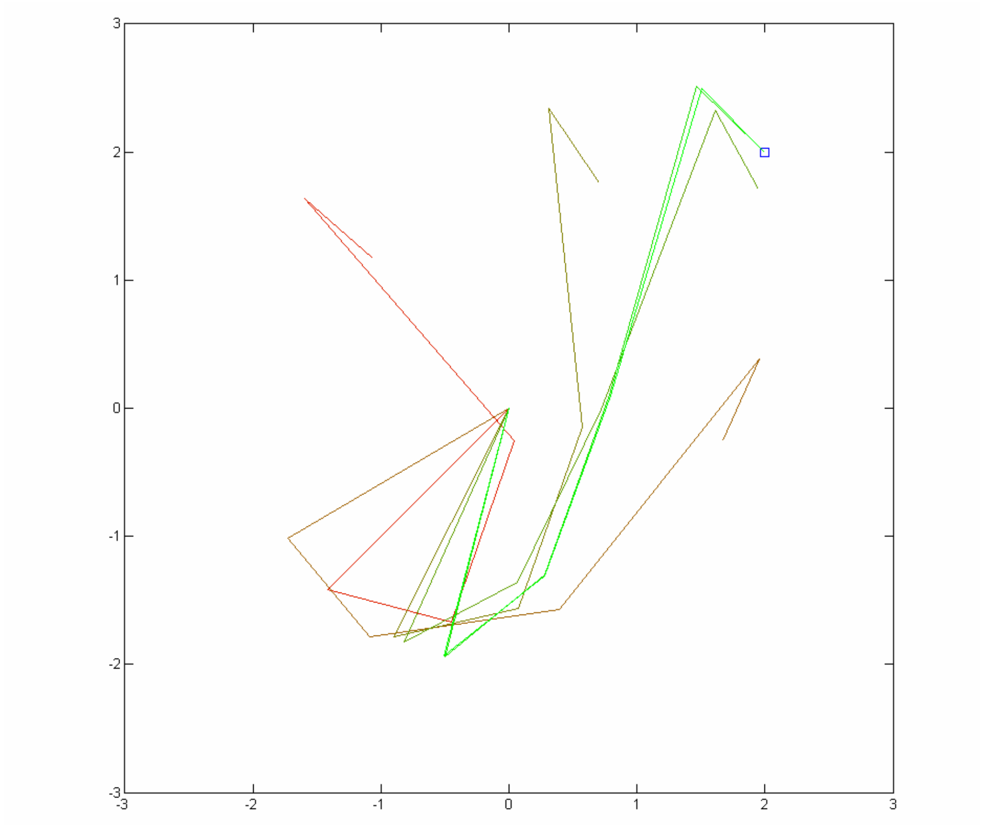


Figure 2: 5 link chain with truncated pseudo inverse

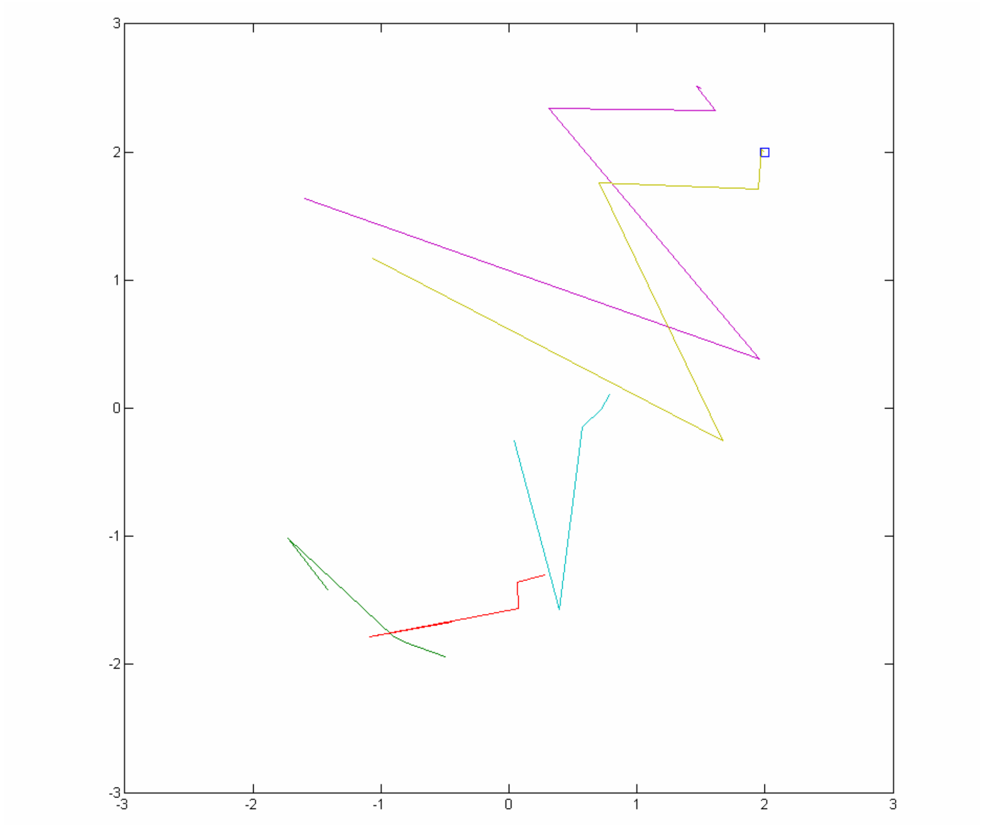


Figure 3: Joint motion of 5 link chain with truncated pseudo inverse

4.2 Transpose

The motion produced by the transpose method appeared aesthetically smooth (Figure 4), but the joint motions exhibited large oscillations (Figure 5). The transpose method converged much more slowly than others, typically a magnitude worse. The speed difference compared to other methods diminished as the number of links in the chain was increased, taking only 4 iterations to converge within 0.01 with an 8 link chain.

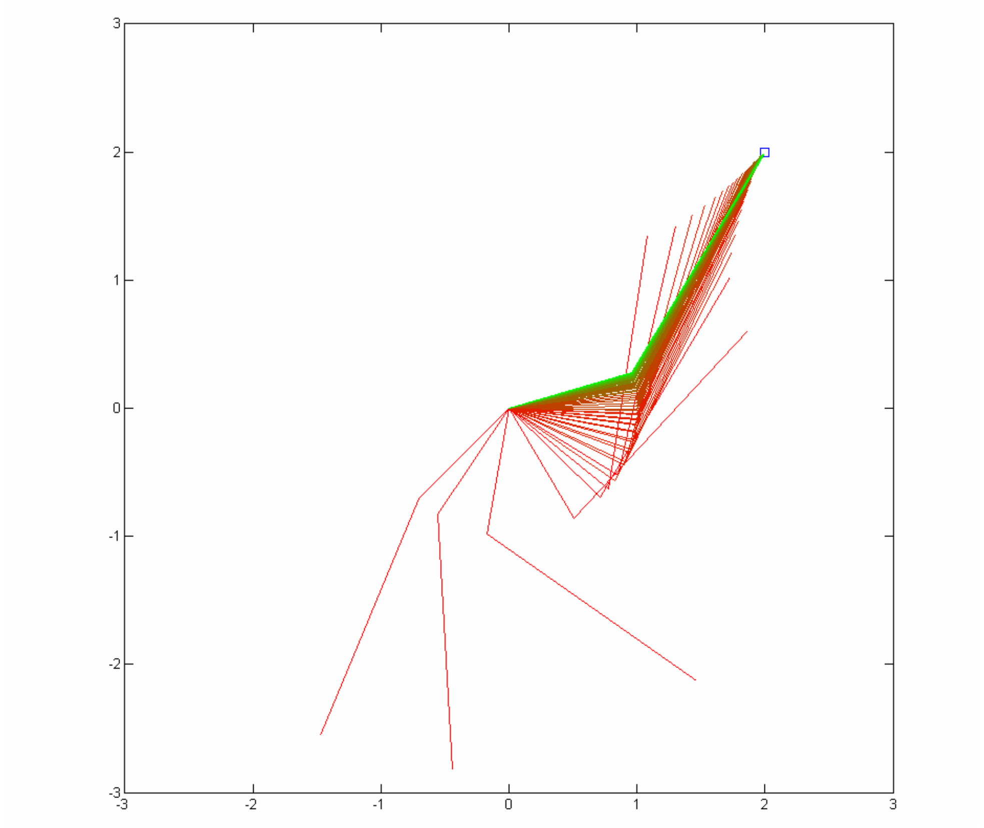


Figure 4: 2 link chain with transpose

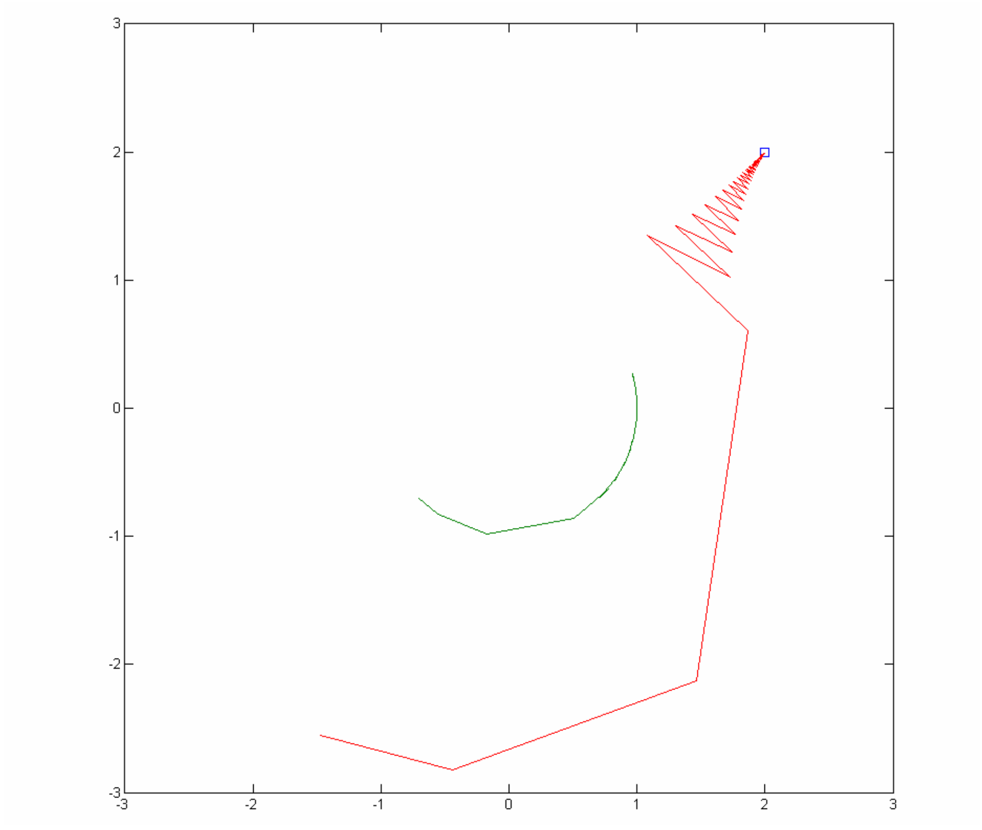


Figure 5: Joint motion of Figure 4

4.3 Damped Least Squares

The value of λ had a noticeable effect on convergence speed, with lower converging more rapidly and higher converging more slowly. The implementation was tested using 1.1, as recommended by [7]. Convergence was always smooth and predictable, but slow, particularly when the chain end was near the target. With unreachable targets performance far exceeded the other methods. In this case, DLS converged smoothly but slowly to the target line with no oscillation while the others in some cases didn't come anywhere near the target. The oscillation problem described in the references might be more present when tracking moving targets. Chains having many links seemed to converge faster, typically to 0.01 in about 10 iterations.

The tests used 0.7 as recommended by [7] for the $d_{\max}$ constant ('deLimit' in the code) that controls how the algorithm "moves" the target closer to the end of the chain. This processing slowed overall convergence somewhat and had no visible effect on convergence to unreachables, which was already good, but noticeably smoothed the trajectory to some reachable targets. Output from a typical solution is below. Figure 6 through Figure 13 through show the chain configurations and joint motions of all 4 cases listed in the comparison table (4.4).

```
EDU>> [final_error, iterations] = IK_Jacob_file('5_link_coarse', 17)
       k    rank used  normError  dTheta ----------->
    1.0000   -1.0000    3.1772   -0.0515   -0.0782   -0.0991   -0.0602
0.0216
    2.0000   -1.0000    2.5221   -0.0815   -0.0020   -0.0966   -0.1190
0.0555
    3.0000   -1.0000    1.9160   -0.0714    0.0885   -0.1012   -0.1738
0.0979
    4.0000   -1.0000    1.4197   -0.0331    0.1578   -0.1122   -0.1903
0.1403
    5.0000   -1.0000    1.0371   -0.0063    0.1844   -0.1228   -0.1628
0.1800
    6.0000   -1.0000    0.7239   -0.0061    0.1774   -0.1248   -0.1120
0.2196
    7.0000   -1.0000    0.4460   -0.0166    0.0969   -0.0734   -0.0375
0.1621
    8.0000   -1.0000    0.2767   -0.0173    0.0536   -0.0412   -0.0133
0.1064
    9.0000   -1.0000    0.1722   -0.0138    0.0315   -0.0238   -0.0054
0.0675
   10.0000   -1.0000    0.1072   -0.0098    0.0190   -0.0141   -0.0025
0.0423
   11.0000   -1.0000    0.0667   -0.0067    0.0117   -0.0085   -0.0013
0.0264
   12.0000   -1.0000    0.0415   -0.0044    0.0072   -0.0052   -0.0007
0.0164
   13.0000   -1.0000    0.0258   -0.0028    0.0045   -0.0032   -0.0004
0.0102
   14.0000   -1.0000    0.0161   -0.0018    0.0028   -0.0020   -0.0002
0.0064
   15.0000   -1.0000    0.0100   -0.0011    0.0017   -0.0012   -0.0001
0.0040
   16.0000   -1.0000    0.0062   -0.0011    0.0017   -0.0012   -0.0001
0.0040
final_error =
   -0.0047
   -0.0041
         0
iterations =
    16
```

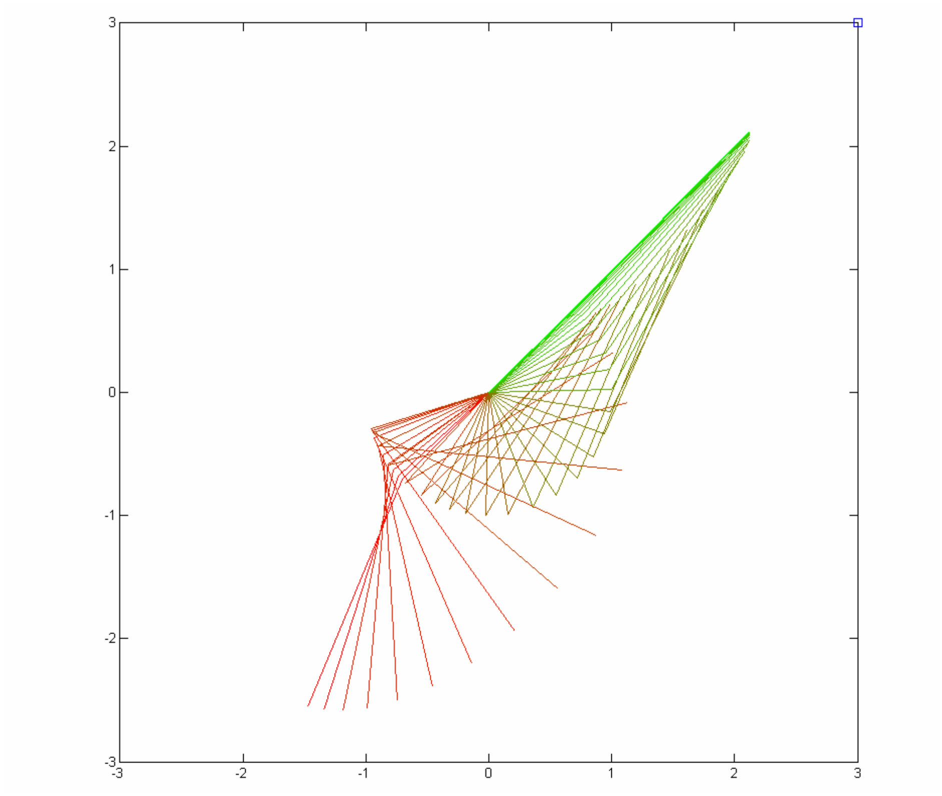


Figure 6: Unreachable target (too far) with damped least squares

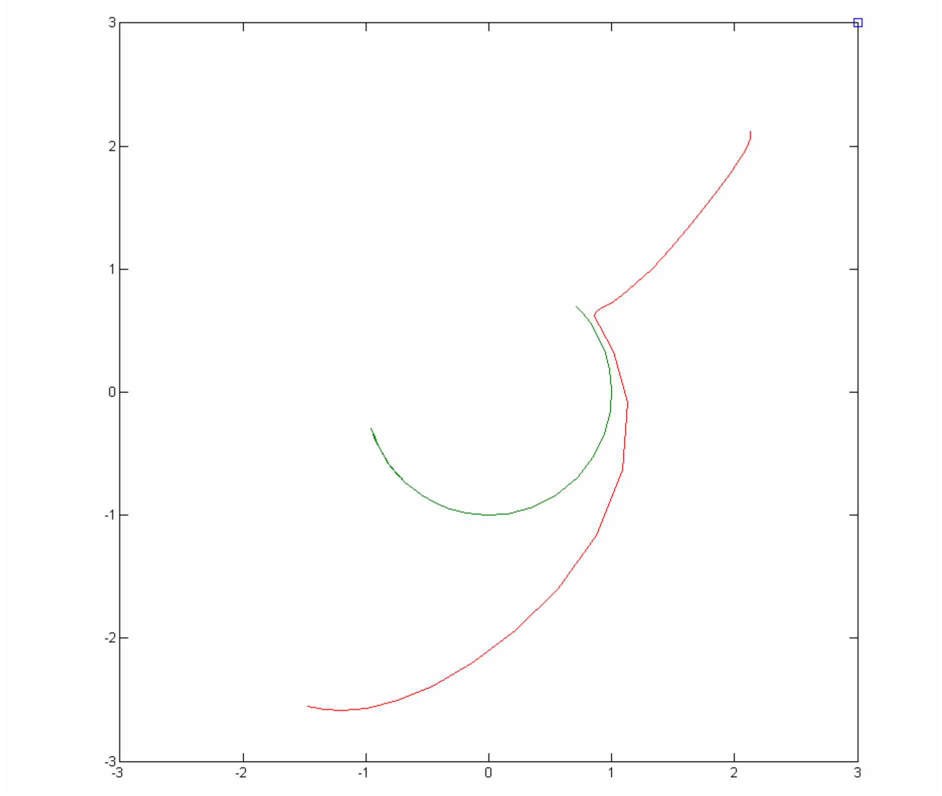


Figure 7: Joint motion of Figure 6

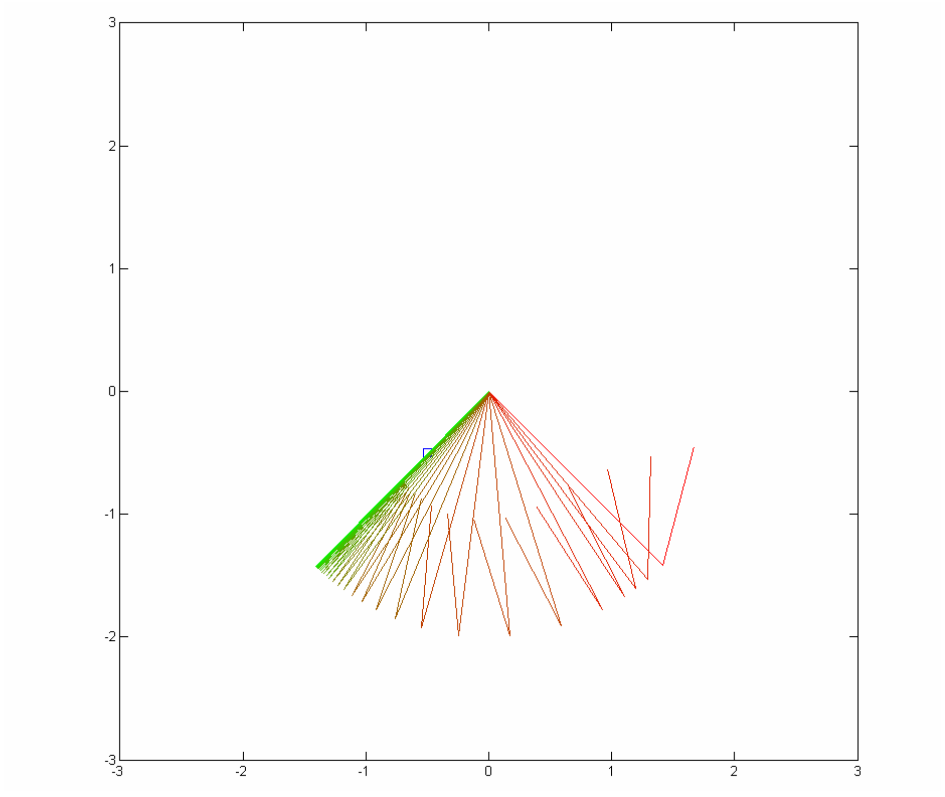


Figure 8: Unreachable target (too close) with damped least squares

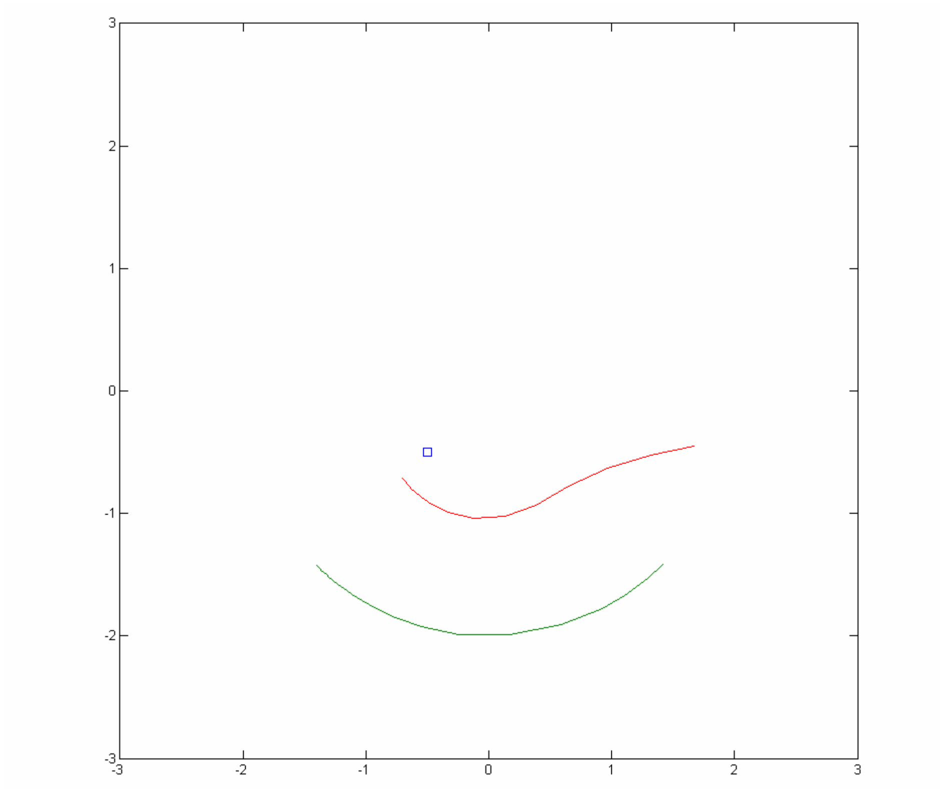


Figure 9: Joint motion of Figure 8

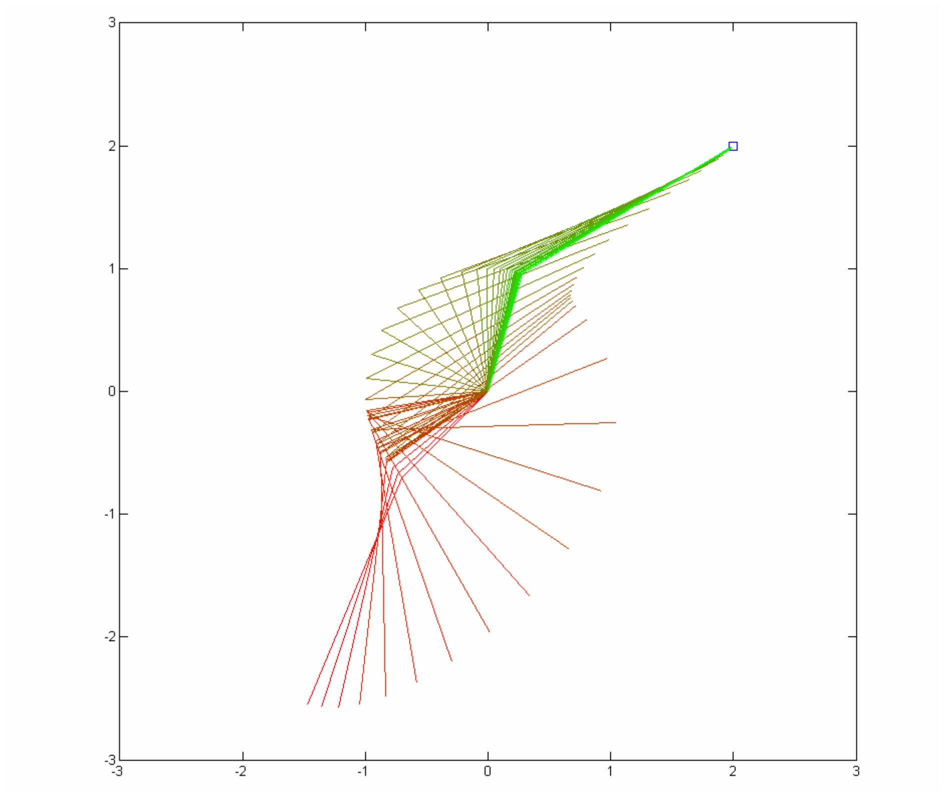


Figure 10: 2 link chain with damped least squares

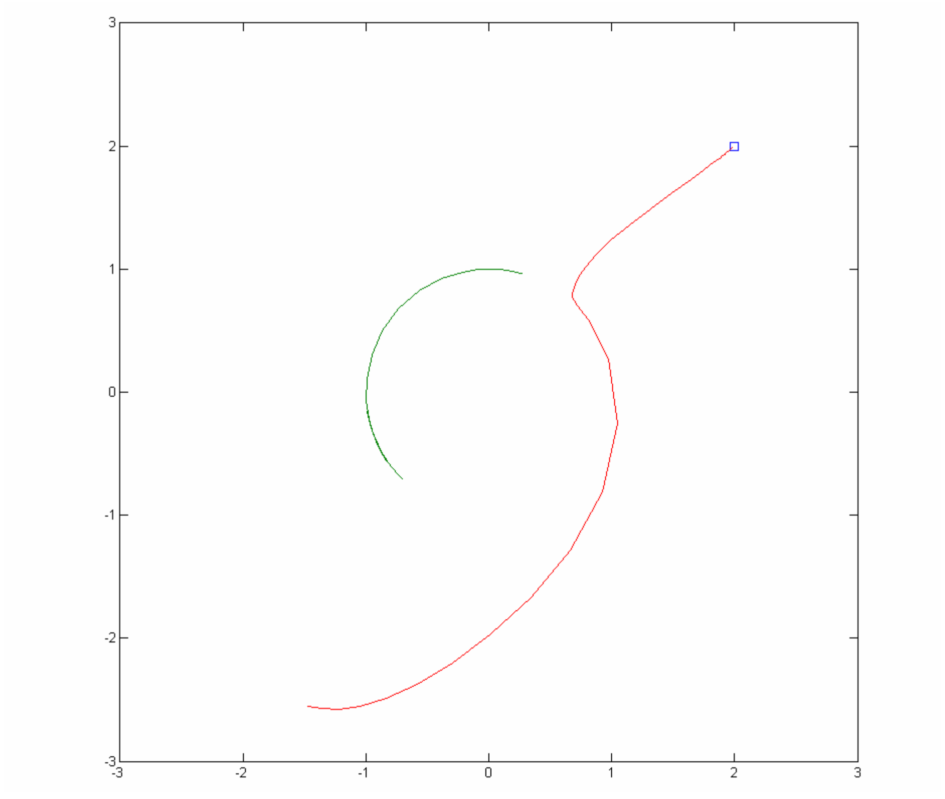


Figure 11: Joint motion of Figure 10

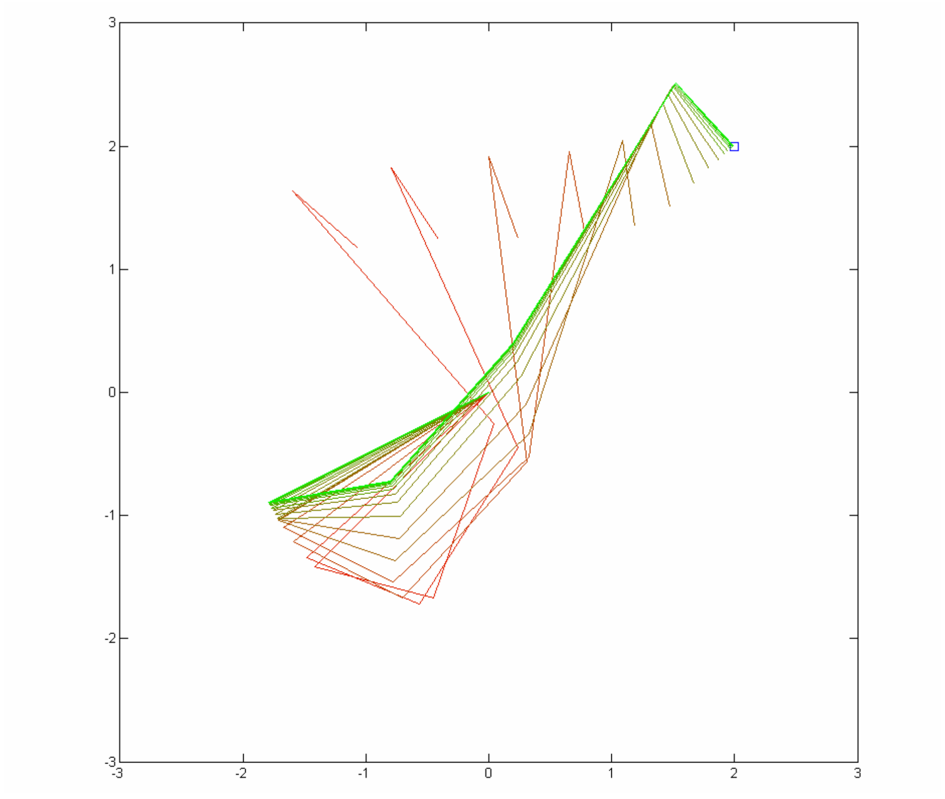


Figure 12: 5 link chain with damped least squares

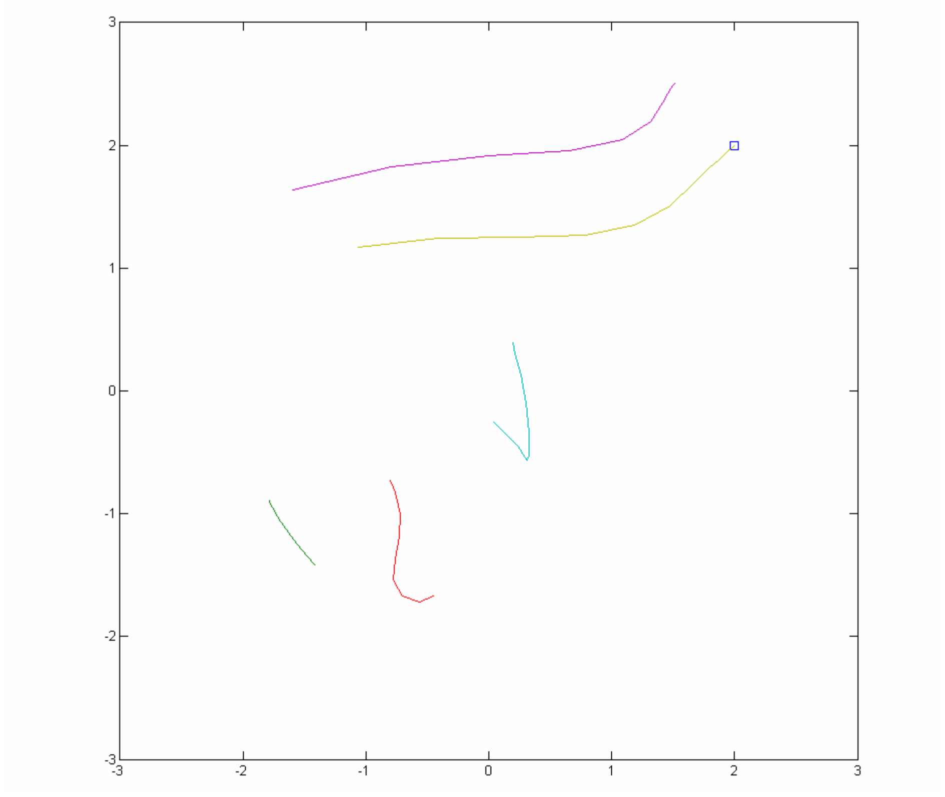


Figure 13: Joint motion of Figure 12

4.4 Comparison

Test Configuration	Truncated Pseudo Inverse	Transpose	Damped Least Squares
2 links, too far			
· Converge to target line	Never	100	45
· Trajectory	Wide oscillation	Wide oscillation	Very smooth
2 links, too close			
· Converge to target line	Never	68	30
· Trajectory	Wide oscillation	Wide oscillation	Very smooth
2 links			
· Converge 0.01	14	120	50
· Converge 1e-7	16	570	141
· Trajectory	Jerky	Wide oscillation	Very smooth
5 links			
· Converge 0.01	6	140	16
· Converge 1e-7	7	459	40
· Trajectory	Jerky	Wide oscillation	Very smooth

5 Conclusions

For all the methods considered, the key to getting good results is using the optimal control constants, which may be difficult to find for every possible chain configuration, but the results show that picking some fixed value by trial and error can work acceptably. Schemes that compute control constants on the fly show promise, but are harder to implement and, in some cases, add significant computation time.

The experimental results basically match the expectations set by the references. There was strong indication that DLS with its smooth motion and immunity to singularities and unreachable targets is the best all around solution, but could be too slow if high convergence accuracy and interactive speed is required. For example, converging in about 30 iterations is pretty good, but that is only to within 0.01 unit, which represents about 1 pixel in screen graphics, barely good enough. The methods that exhibit slow convergence always do so in the last few percent of the trajectory, indicating that dual solutions that switch from DLS to pseudo inverse methods when the end is near the target have merit.

6 References

[1] C. WELMAN, *Inverse Kinematics and Geometric Constraints for Articulated Figure Manipulation*, Masters Thesis, Simon Frasier University, 1993.

[2] <Author unknown>, kinematics.pdf, Powerpoint presentation

[3] DAVID BROGAM, *Kinematic Lecture 09*, Powerpoint presentation, University of Virginia, Fall 2003.

[4] K. J. CHOI, *Implementation of Inverse Kinematics and Application*, Powerpoint presentation.

[5] JASON CLARK, *Inverse Kinematics*, Powerpoint presentation, Essential Math for Games

[6] S. R. BUSS, *Introduction to inverse kinematics with Jacobian transpose, pseudoinverse, and damped least squares methods*. Typeset manuscript, available from http://math.ucsd.edu/~sbuss/ResearchWeb, April 2004.

[7] S. R. BUSS AND J. KIM, *Selectively Damped Least Squares for Inverse Kinematics*, October 25, 2004.

[8] A. MACIEJEWSKI AND C. KLEIN, *Numerical Filtering for the Operation of Robotic Manipulators through Kinematically Singular Configurations*, Journal of Robotic Systems, 5 (1988), pp. 527-552.