Computer Animation and Games I – CM50244

Coursework 1 Report

Unit Leader:
Dr. Yongliang Yang

Name:

Constantinos Theophilou

Degree Programme:

MSc Digital Entertainment
University of Bath

Inverse Kinematics solution

1 Overview

For the first coursework, we were assigned the task to explore the problem of inverse kinematics using a linkage in 2D space. Given the openness of this task, the exploration range was huge and this coursework turned out more complicated than I hoped for. Nonetheless, I implemented a solution while producing numerous functions to solve the problem at hand. For explanation purposes, the functions and MATLAB scripts are listed and summarised below for faster referencing.

Files:

1. Main.m: This script is responsible for supplying the initial parameters as well as the configuration of the inverse kinematics solution. The program is run from main.m by either holding down the CTRL and ENTER keys simultaneously or by pressing the Run button on MATLAB's interface.
2. InverseKinematics1.m: This script is the protagonist of my implementation, where all the major code blocks are run, while also providing visuals to show the 2D linkage's position. InverseKinematics1.m is my solution but without constraining the joints' angles to a range.
3. InverseKinematics2.m: This script is the protagonist of my implementation, where all the major code blocks are run, while also providing visuals to show the 2D linkage's position. InverseKinematics2.m is my solution given a range in which the angles can vary.
4. FK.m: Used to find the end-effector's position in the grid given the current joint angles and their corresponding rods' lengths.
5. Comp_jacob.m: This script is responsible for setting up the Jacobian matrix, and passing it back to the inverse kinematics code block.
6. Joints_pos: Works in a similar way as the FK.m script, but instead of finding only the end-effector's position it also computes the rest of the joints' positions given the joints' angles and the rods' lengths.
7. Sin_interp: This script is responsible for interpolating angle values represented on a sin curve, given only two keyframes, the start frame and the end frame.
8. Sin_interp2: This script is responsible for interpolating angle values represented on a sin curve, given the two major keyframes, the start frame and the end frame, as well as two additional intermediate keyframes.
9. Ease.m: Constructed with the aid of Rick Parent's book, [4]. It is responsible for producing the sin curve used for the trigonometric interpolation mentioned in the sin_interp scripts mentioned above.
10. Draw_circle_boundaries.m: Used to draw the circle on the axes based on the max stretch of the linkage.
11. Isclose.m: This script is my implementation of the isclose function included in Python's libraries, due to the fact that I could not find a similar function in MATLAB's libraries.
12. MouseMovement: Used to continuously capture mouse movement.

2 Introduction

The Inverse Kinematics problem, has been a hindrance to anyone wanting to accept the challenge of implementing the best possible solution, with many of its solutions being constructed by robotics experts to control robot movements, as well as for animation and games in most recent years. It is working in a way opposite of how forward kinematics is articulated, since it only considers the end-effector's current position and the linkage's current pose to move the linkage to the desired goal position. But due to its sparse nature, this problem has no single solution and the choices available at the moment have their advantages and disadvantages. On the next subchapters, one can find the description of my solution towards the inverse kinematics problem, in addition to information I have found while exploring the various solution mechanisms and the corresponding mathematics involved.

3 The Jacobian

Most of the currently available solutions for the Inverse Kinematics problem, involve the formulation of a matrix of first order partial derivatives called the Jacobian matrix and it is used to iteratively compute the change in joints angles until the end-effector finally reaches the goal position. Nonetheless, the Jacobian's

structure is not the same in all cases, as a linkage might have any number of joints and the size of the Jacobian is directly proportional to this number. For my implementation, I followed a paper by Kumar, and the book by Parent to formulate the Jacobian Matrix, [4]. To make things simpler, my implementation consists of a linkage of three joints and three rods, which is kept at a constant size all throughout, meaning that the size of the Jacobian is also kept constant.

3.1 Forming the Jacobian

As far as the formulation of the Jacobian matrix is concerned, the formula for forward kinematics was used to produce the partial derivatives. Before I go into explaining this, let's first look into the linkage, in order to describe how the thetas are captured, Figure 1.

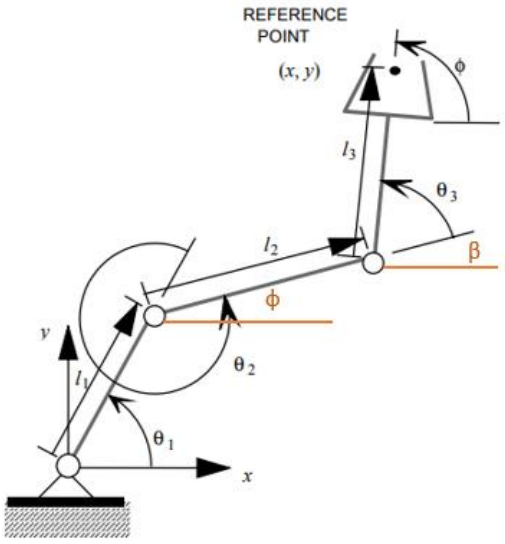

Figure 0:1: 2D linkage with three joints and three rods. Taken from [] with a few additions.

Based on the linkage on the left, the angles we are concerned about are $\boldsymbol{\theta_{1,2,3}}$ and the lengths of the rods, $\boldsymbol{l_{1,2,3}}$. To formulate the Jacobian, the first step is to produce the forward kinematics formula which uses ϕ and β. ϕ can be computed by:

$$\boldsymbol{\phi = \theta_1 - (2\pi - \theta_2)}$$
$$\boldsymbol{= \theta_1 + \theta_2 - 2\pi}$$

Using the same method, β can be computed. Therefore, the formulas used to find the x and y coordinates of the end-effector based on the joints' angles and the rods lengths are:

$$\boldsymbol{E_x = l_1 * cos(\theta_1) + l_2 * cos(\phi) + l_3 * cos(\beta)}$$
$$\boldsymbol{E_y = l_1 * sin(\theta_1) + l_2 * sin(\phi) + l_3 * sin(\beta)}$$

where $\mathbf{l_{1,2,3}}$ are the rods' lengths and $\boldsymbol{\beta = \theta_1 + \theta_2 + \theta_3}$.

Following the above formulations, the Jacobian matrix can be brought to life by getting the first order partial derivatives of each **θ**, and structuring the Jacobian matrix, denoted as **J**, by:

$$\boldsymbol{J = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} & \frac{\partial x}{\partial \theta_3} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} & \frac{\partial y}{\partial \theta_3} \end{bmatrix}}$$

For the final step, of putting the partial derivatives in place, please refer to comp_jacob.m script, which holds the function that constructs the Jacobian matrix given the angles and lengths. Alternatively, the paper by Kumar is very useful for producing the Jacobian matrix, [2].

3.2 The Jacobian Pseudoinverse

For reference reasons, the inverse kinematics formula is introduced, but will be described in the next chapter:

$$\boldsymbol{V = J\odot}$$

Where ⊙, theta dot, is the vector holding the changes to the joint parameters and **V** is the vector holding the velocities or the distance of the current end-effector's position to the goal position. We ultimately, want to solve for ⊙, so rearranging this:

$$\boldsymbol{\odot = J^{-1}V}$$

This entails that to find the desired changes to angles, we need to find the inverse of the Jacobian matrix. Since I am only working in 2D space, the Jacobian matrix is not inversible, because it is not a square matrix. Additionally, inversing the Jacobian matrix introduces more problems, namely the problem of singularity as explained by Parent, page 178 [4]. With the above information, an alternative to using the inverse of the Jacobian had to be found to proceed with the formulation. To reach a solution, the pseudoinverse of the Jacobian matrix is used instead, denoted by $\mathbf{J^+}$:

$$\odot = \mathbf{J}^{+}\boldsymbol{V}$$

$$\text{where } \boldsymbol{J}^{+} = \boldsymbol{J}^{\boldsymbol{T}}\left(\boldsymbol{J}\boldsymbol{J}^{\boldsymbol{T}}\right)^{-\mathbf{1}}$$

This approach is possible, the $\mathbf{J}^{+}$ is now a square matrix and can be inversed as shown above, to be used for solving the inverse kinematics formula.

3.3 Damped least-squares

The pseudoinverse method proposed previously, is useful in cases where **J** is not a square matrix and to avoid singularities, but the pseudoinverse does not actually avoid all the singularities, as Parent suggests, page 180 [4]. A different approach is using the damped least-squares method as denoted below:

$$\odot = \mathbf{J}^{\mathbf{T}}\left(\boldsymbol{J}\boldsymbol{J}^{\boldsymbol{T}} + \boldsymbol{\lambda}^{\mathbf{2}}\boldsymbol{I}\right)^{-\mathbf{1}}\boldsymbol{V}$$

This is merely an extension to the pseudoinverse method, as the only difference is the addition of a user-supplied variable, $\boldsymbol{\lambda}$, called the damping constant. Squaring this new variable and multiplying it with the identity matrix, **I**, it is argued that it is more precise, with no chance of singularity, but the rate of convergence is decreased, producing a significant number of frames from start pose to end pose. Experiments have been conducted and will be shown in the next chapter.

4 Inverse Kinematics Solution

In this chapter, my implementation of solving the inverse kinematics problem will be thoroughly explained, by describing the steps taken and showing images of experiments and results. Initially, I will introduce the inverse kinematics problem. Based on this coursework, a linkage in 2D space exists with three angles, each describing the orientation of a joint, and three lengths, each describing the size of the rods, or the distance between the joints, which together produce an arm. The inverse kinematics problem, using the current lengths and angles as parameters, has to move the end-effector towards the goal position, which is the input obtained by the user. In order to move the end-effector efficiently to the goal position, the angles of the joints must change, allowing the end-effector to reach the desired position. Therefore, this change in angles is what will drive the end-effector and ultimately used to solve the inverse kinematics problem. The formula for the inverse kinematics, specifically this change in angle rotation for each joint, is given below:

$$\odot = \mathbf{J}^{-\mathbf{1}}\boldsymbol{V}$$

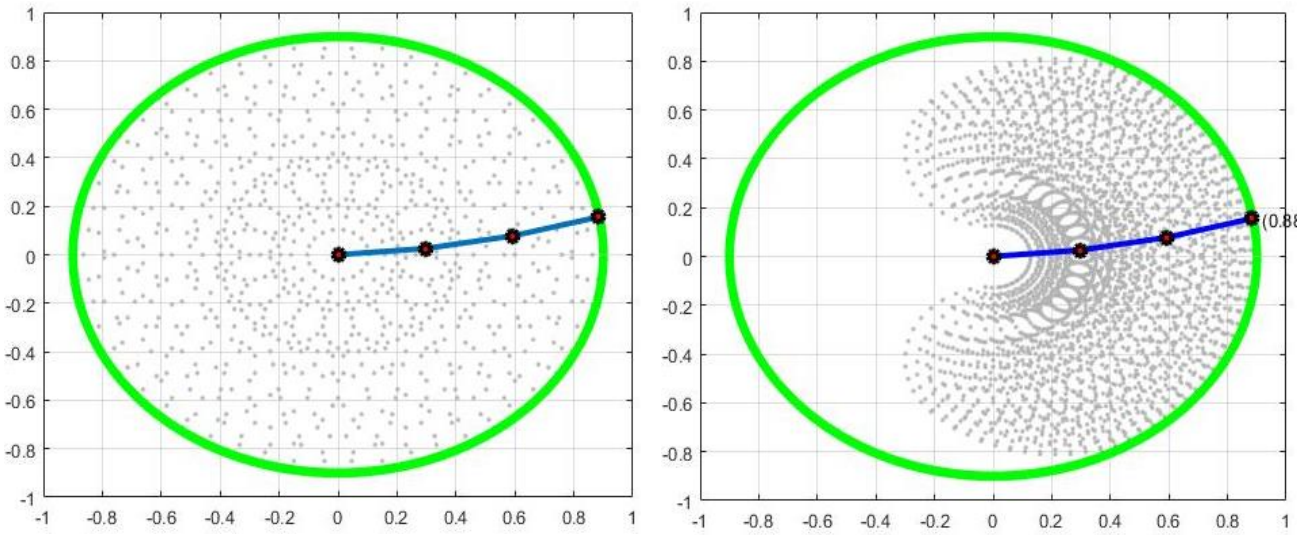

Figure 2: (Left) Show the grid and initial position of the linkage with angles [5,5,5]. The green circle shows the limits of the end-effector position given the max stretch, when all angles are 0, and the grey area shows where in the grid the end-effector can reach. It is dotted and sparse because of they were recorder using angle steps of 20 degrees to make it faster. Figure 3: (Right) Shows the same information as the left figure, but in this case, there are constraints on the angles, limiting their rotation, thus producing a

Where ⊙, theta dot, is the vector holding the changes to the joint parameters and V is the vector holding the velocities or the distance of the current end-effector's position to the goal position. The solution to the above formula does not entirely provide the total ⊙, but rather a portion of the change of the angles, which in turn provides intermediate key frames, between the initial and final pose, for the configuration of the linkage. Thus, this can be solved iteratively. Before I jump into the loop which will work out the change of angles, I

would like to show how the grid and linkage going to look, as well as the space where the end-effector of the 2D linkage can be moved, Figures 2 and 3 above.

The drawing of the initial position of the linkage, the green circle boundary and the grey "reachable" area, is done before entering the loop for computing the theda dot. Another thing done outside the loop, is to capture the goal position as input from the user using the location of the mouse with respect to the grid and calculate the total distance from the the point p0, where the linkage is constantly attached, in this case (0,0), to the goal position.

4.1 The Loop

The loop starts by first setting the stopping conditions. In this case the stopping conditions are when the distance from the current end-effector to the goal is close to zero. A tolerance is used instead of zero due to the fact that the program seems to get into an infinite loop, and this is presumably because of the floating points comparison for each of the positions. An alternative was to set how many decimal points to check for and then check whether the distance is zero. Below is the generalised version of the looping algorithm for solving inverse kinematics:

While the distance from end-effector to goal is greater than a tolerance:

1. Construct Jacobian matrix given current angles and lengths:

$$J = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} & \frac{\partial x}{\partial \theta_3} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} & \frac{\partial y}{\partial \theta_3} \end{bmatrix}$$

2. Find the pseudoinverse OR damped least-squares version of the Jacobian matrix:

$$J^{+} \text{ or } J^{T}\left(JJ^{T} + \lambda^{2} I\right)^{-1}$$

3. Find the end-effector's current position on the grid using the forward kinematics formula:

$$E_x = l_1 * cos(\theta_1) + l_2 * cos(\phi) + l_3 * cos(\beta)$$
$$E_y = l_1 * sin(\theta_1) + l_2 * sin(\phi) + l_3 * sin(\beta)$$

4. Find the end-effector velocity vector:

$$V = \begin{bmatrix} G_x - E_x \\ G_y - E_y \end{bmatrix}$$

5. Compute theta dot, the change of angles:

$$\odot = J^{+} \text{ or } \odot = J^{T}\left(JJ^{T} + \lambda^{2} I\right)^{-1} V$$

6. Add the corresponding change in angles to each angle contained in the linkage, and store as a frame:

$$\theta_1 = \theta_1 + a * \odot(1)$$
$$\theta_2 = \theta_2 + b * \odot(2)$$
$$\theta_3 = \theta_3 + c * \odot(3)$$

Where a, b, c are constants deciding the rates at which each joint angle can change

7. Calculate new end-effector's position based on the new angles obtained, using the forwards kinematics formula again.

8. Calculate new distance between end-effector and goal position and check whether the conditions of the loop still; if they do keep going otherwise stop and return the array of frames.

This is the generalised algorithm for the loop that produces the solution of the inverse kinematics problem using the Jacobian matrix. The word generalised is used because further extensions can be added to the loop, such as acknowledging angle constraints, which will be discussed shortly.

4.2 Inverse Kinematics without angle constraints

Producing an algorithm without the angle constraints for the inverse kinematics solution, is exactly like the algorithm shown above. The results were spot on, with the linkage changing configuration continuously based on the new angles from each iteration of the loop, and finally the end-effector managing to reach the goal position 100% of the time with no error. Below are some example runs showing the intermediate frames as well, as can be seen in Figures 4 and 5.

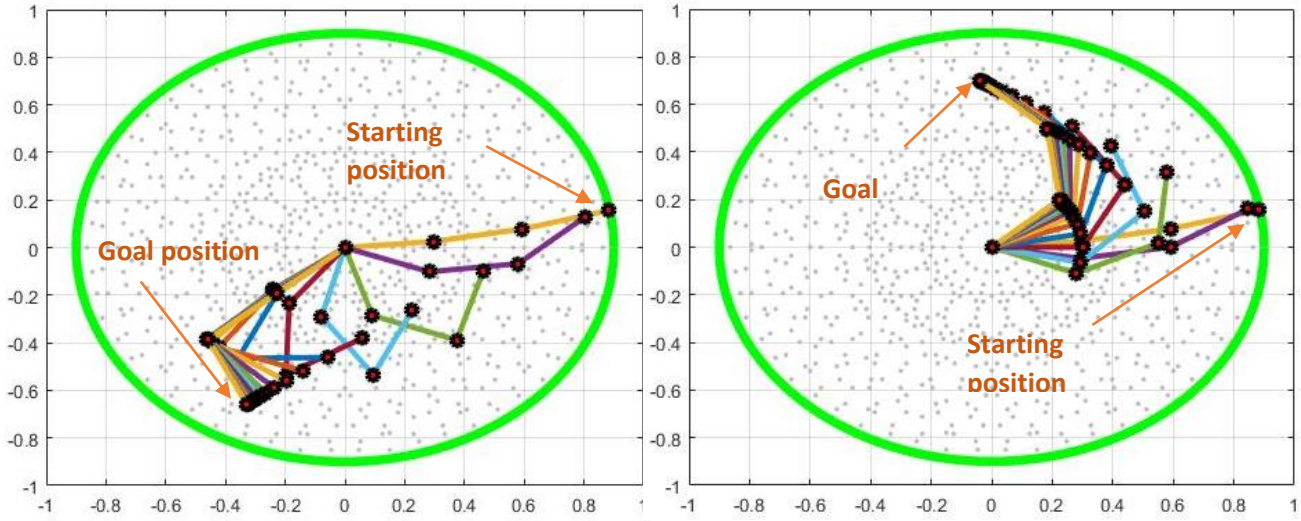

Figure 4 and 5: Example runs of the algorithm to achieve movement of whole linkage by changing joint angles, to reach the goal position. Intermediate frames are also captured, with a step variable recording frames every 20 iterations of the loop.

As the end-effector gets closer to the goal position, the intermediate change in angles is much smaller when compared to initial change in angles. For the above figures, the algorithm was set to record a frame every 20 iterations of the loop, but as you can see there are multiple frames towards the goal position. If every frame was captured, there would be around 1000 to 2000 frames to store. The pseudoinverse of the Jacobian method was used for creating the above figures and actually this method produces less frames than the damped least-squares method, and that is mainly due to the damping constant, λ. What is very interesting is that the path these two methods decide to take are also very different. The following figures, Figure 6 and 7, show the same example runs with almost identical goal positions, using the damped least-squares method.
The damped least-squares produces this arc-like movement due to the damping constant being squared when scaling the identiity matrix to be added to the pseudoinverse of the Jacobian. When comparing the two methods, one can argue that the pseudoinverse method is much faster and computationally cheaper, but the damped least-squares method produces a more natural arc-like movement with almost five times the number of frames, frames which are not useful at all since they are almost identical when the end-effector is very close to the target.

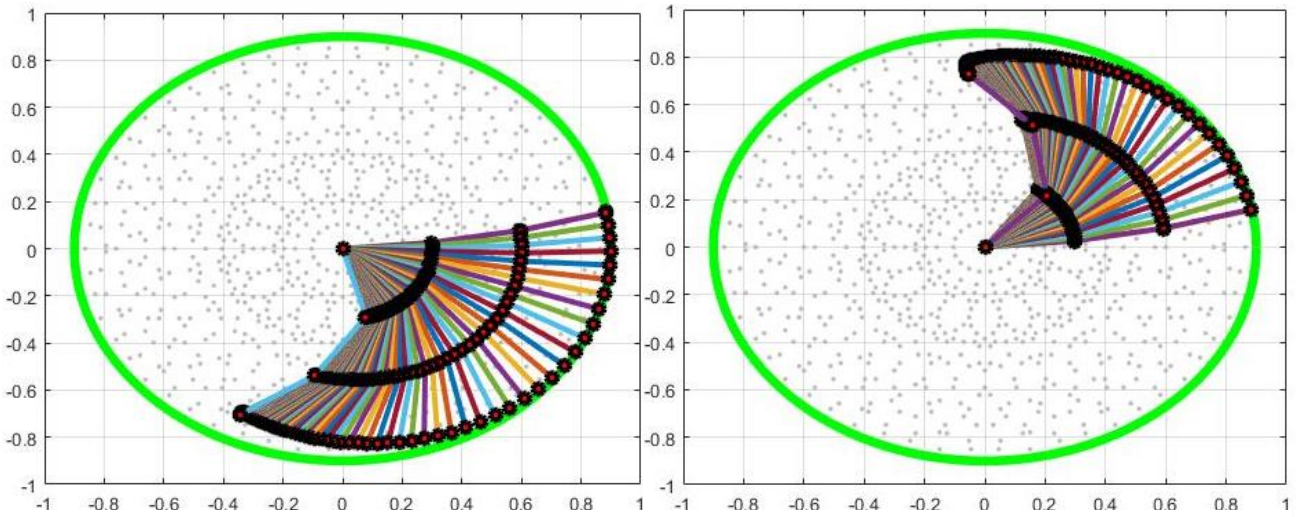

Figure 6 and 7: These two figures show the path of the joints' positions until the end-effector reaches the goal position, using the damped least-squares method of the Jacobian. The results produce an arc-like movement which is more natural but instead produces way more intermediate frames to reach the goal position. The algorithm still takes 1 frame every 20 iterations of the loop.

4.3 Inverse Kinematics with angle constraints (Part 3 of coursework/ flexibility)

The most challenging part when producing an animation using inverse kinematics, is limiting the angles of certain joints, to produce a more human-like pose and movement. For example, consider the arm of a human, it can bend and twist, but cannot bend past the 180-degree mark, past the elbow. Figure 3, shows the space of the 2D grid that the end-effector can reach, in grey, given the following constraints: $\boldsymbol{\theta_1}$**~ [-45 45],** $\boldsymbol{\theta_2}$**~ [-90 90] and** $\boldsymbol{\theta_3}$**~ [-170 170]**. When restricting an angle between a range, the outcome is not as expected, and instead depends on the initial pose of the linkage and the direction of the goal position, to derive the solution. The method I used is advised by Kang Teresa Ge, in her thesis [1]. It is called the Lagrange Approach and it is very simple but not always reliable. Kang suggests that clamping an angle that has passed a limit, to its lower or upper bound depending on which limit was passed, will cause the rest of the angles to account for this consistency, in the next iterations, of that angle, and eventually be forced to move towards a more suitable orientation so that the goal is reached nonetheless. When implementing this method myself, the result was as expected until I tried moving to a point behind the direction of the end-effector. To extend the algorithm stated in 4.1, to take into consideration angle range boundaries the following step must be added between steps 6 and 7. So after adding the ⊙ on each angle:

6.5 For i = 1 to size of angles vector
If the angle is greater than the maximum value of the range vector
angles(i) = max(range_theta(i))
else if the angle is less than the minimum value of the range vector
angles(i) = min(range_theta(i))
end

With this additional step, the inverse kinematics solution iteratively computes the rate of change of the angles that are within their boundaries by keeping the angles that are out of bounds to a constant value until and if that angle enters the allowed range again. This method is commonly referred to as adding hard constraints for limiting a variable. Nonetheless, the result is mostly successful, but there are times that it is incorrect, Figures 8 and 9.

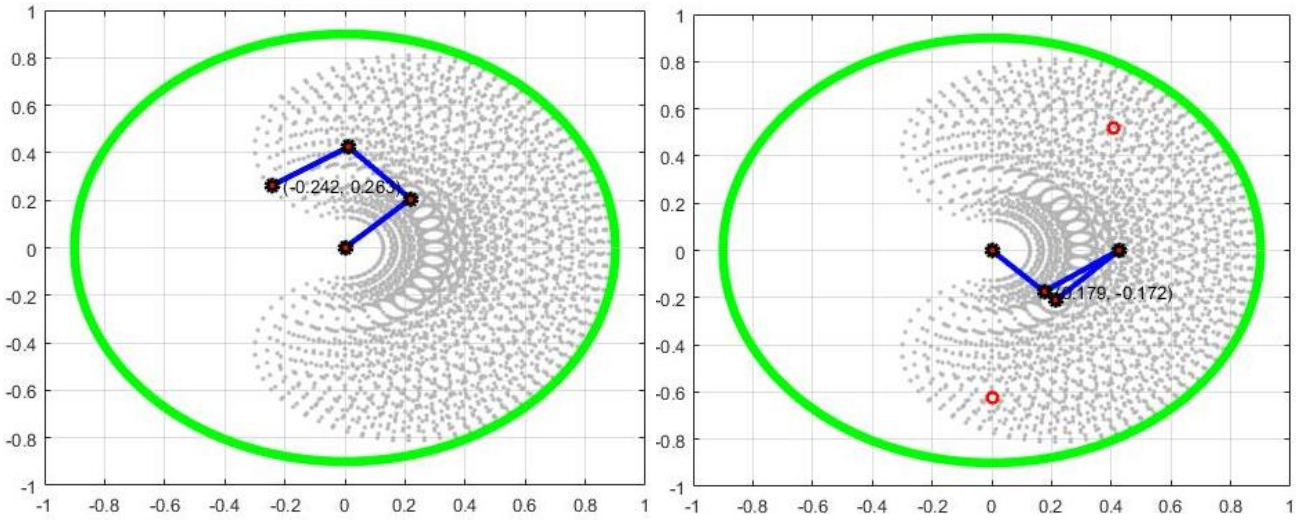

Figure 8: (Left) The linkage is in its final pose, with 2 of its angles at their limits, in this case the angles of the first and second joints, at 45 and 90 degrees respectively. Figure 9: (Right) The linkage is stuck while trying to move from the previous goal position, upper red circle, to the current goal position, lower red circle. More specifically the third joint is the one causing the issue, by not rotating the other way around to reach the goal position, forcing all the joints' angles to reach their limit. Figures were constructed using the pseudoinverse of the Jacobian matrix.

Naturally, in the case of Figure 9, the loop should have gone into an infinite state, since the end-effector never reached the goal position, thus causing the algorithm to loop indifinetely. To overcome this problem, I added a penalty schedule to avoid not having a final pose to show. This penalty increments every time two consecutive frames are equal or very close, using the isclose.m script, and when the penalty reaches a certain number, in this case 100, then the loop halts and returns the current frames. Eventually it felt normal to have

such a behaviour since if this was a human arm then it would, indeed, be unable to reach that goal position as the elbow would be an obstacle in its path. When running the same examples as above, and alternatively use the damped least-squares method, the results were better but due to the penalty check and the vast amount of frames produced by this method, which were almost identical, the end-effector is not always on target but a little bit off. To avoid this, a new check was added to see whether the Jacobian method used, is the damped least-squares method and increase the number of decimals the isclose function compares, if it is.

Successfully constraining the inverse kinematics problem is a field where many researchers tried to tackle and not everyone was successful in doing so. There exist ways to achieve this, though, and these ways are complex, in the sense that the algorithm must keep communicating back and forth to keep the angles constrained. When researching about inverse kinematics incorporating constraints, one can find a few reasonable solutions which are not very straightforward and hard to follow. One mechanism found is to produce a simple projection of the unconstrained solution to achieve a more natural pose, which is proposed in, [6] by Wellman, but this does not always produce a result that will be optimal. Another mechanism is to use soft constraints as mentioned by Rick Parent in the section called, "Adding more control", page 181 [4]. This section talks about an alternative to pseudoinverse or damped least-squares method, but biasing the solution to stay in the centre of the soft-range constraints, thus finding an optimal solution. Additionally this method does not guarantee that the constraints will always be met, hence the term "soft-constraints". When trying to implement this, the results were completely unexpected causing the linkage to immediately move to the maximum of its constraints for each joint. A successful solution available is to use weighting of moves for each joint individually, suggested in [3] by Meredith et al. This would cause the $\dot{\theta}$ vector to be weighted thus keeping the joint angles within the constraints range.

5 Interpolation and slow-in/slow out movement

When animation was at its initial steps, the early Disney group compiled a list of principles that each animation must follow to look lifelike, [5]. One of these significant principles is the slow in and slow out of movement of a character. If an animator wants to mimic the human movement, then the animator must take into account the speed at which humans move their limbs as well as the acceleration and deceleration associated with each movement. While researching on how to incorporate the slow in/slow out movement into the 2D linkage, I stumbled upon a chapter in Parent's book, specifically chapter 3 [4]. Chapter 3's main topic is interpolation and a very important aspect in animation and games. An interpolation method that stood out from the rest, when considering the slow in and slow out of movement, is sine interpolation. For this part of the coursework, I proceeded with implementing two sine interpolation mechanisms that differ on the number of frames they deploy to produce the animation frames. These will be discussed below.

5.1 Sine Interpolation

The base of sine interpolation is the well-known sine curve, explicitly a segment of the sine curve. From $-\frac{\pi}{2}$ **to** $+\frac{\pi}{2}$ the sine curve is producing a segment that is optimal for slow in/slow out movement. The trick here, though, is to displace this segment as to start from the origin and have the y-axis being the distance, **s**, and the x-axis being the time**, t**, and both axes taking values from 0 to 1, thus scaling the sine segment to include only useful values. The following figure shows the sine curve and the optimal segment taken from the book, [4].

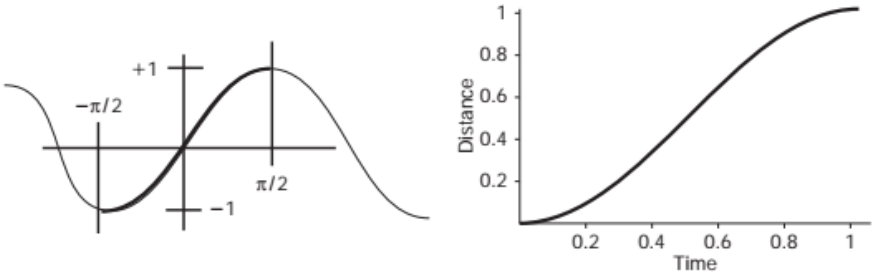

Figure 9: On the left is the sine curve and labelled is the segment that is desirable, $-\frac{\pi}{2}$ to $+\frac{\pi}{2}$ and -1 to 1. On the right, is the outcome we wish to produce, with the y-axis being the distance from initial pose to end pose, normalised from 0 to 1, and on the x-axis the time from 0 to 1.

A step further is to break the segment into three further segments, called sinusoidal pieces, one for accelerating from a complete stop, one for keeping a constant velocity and the final one for decelerating to a complete stop where there are no sudden spikes in velocity. To do so, one must take into account two further variables, namely **k1** and **k2**. These variables govern where each sinusoidal piece starts and ends, with **k1** being the time in which we want to stop the acceleration and **k2** the time in which we want to start deceleration. This gives us three equations each representing one sinusoidal piece:

$$s = k1 * \left(\frac{2}{\pi}\right) * (sin\left(\frac{t}{k1} * \frac{\pi}{2} - \frac{\pi}{2}\right) + 1)) \qquad if\ t \leq k1$$

$$s = (\frac{\frac{k1}{\pi}}{2} + t - k1) \qquad if\ k1 \leq t \leq k2$$

$$s = (\frac{\frac{k1}{\pi}}{2} + k2 - k1 + \left((1 - k2) * \frac{2}{\pi}\right) sin((\left(\frac{t - k2}{1 - k2}\right) * \frac{\pi}{2} \qquad if\ k2 \leq t$$

These three equations are responsible for moving a point along the curve based on **k1** and **k2**, and normally all of these values will add up to 1. But that is not the case here since there are two more aspects one must take into consideration. Firstly, the resulting values must be scaled down by a factor which is equal to the total distance travelled and given by, **f:**

$$\mathbf{f} = \mathbf{k1}(\frac{\mathbf{2}}{\boldsymbol{\pi}} + \mathbf{k2} - \mathbf{k1} + (\mathbf{1} - \mathbf{k2})(\frac{\mathbf{2}}{\boldsymbol{\pi}})$$

$$s = \frac{s}{f}$$

Secondly, the values outputted from the above equations do not add up to 1, but rather accumulate the difference and add up to a value which lies around 50.5. To overcome this issue, I performed the following:

$$s(i+1) = s(i+1) - s(i) \quad for\ i = 1\ to\ the\ (size\ of\ the\ time\ array) - 1$$

The time array must be from 0 to 1 with constant step size and in my case **t = 0.01:0.01:1**, which means the time array is set to have 100 intermediate values from 0 to 1, thus producing 100 frames between start position and goal position of the end-effector, using the ease.m script to calculate the value of **s**. To visualise this, the following figure, Figure 10, shows exactly how **k1** and **k2** affect the acceleration and deceleration segments respectively.

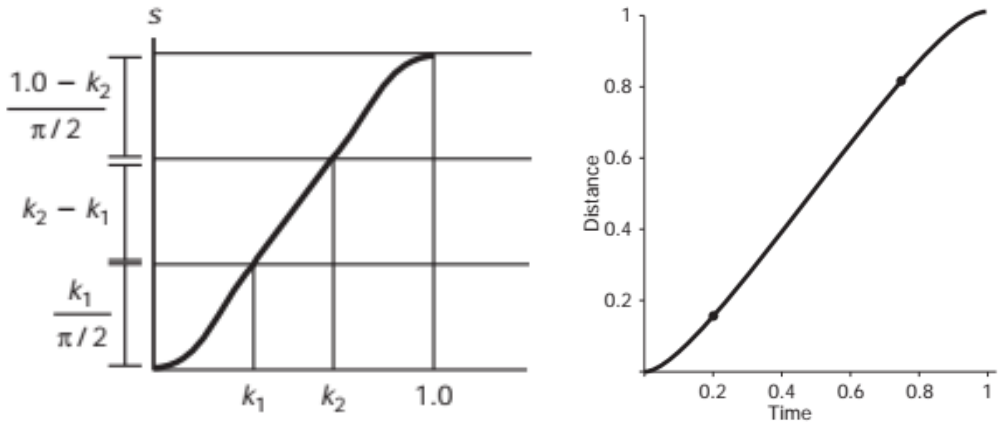

Figure 10: On the left, the figure shows exactly how the sinusoidal pieces are optimised based on the two variable k1 and k2 which govern the time at which acceleration stops and deceleration starts respectively, and how the y-axis is scaled. On the right-hand side, the figure shows the optimised curve segment with k1 = 0.2 and k2 = 0.75. This figure was taken from the book [4].

5.2 Sine Interpolation using two keyframes Vs. using four keyframes

For the interpolation mechanism, I performed two solutions, one which takes two keyframes, start frame and end frame, to perform the slow-in/slow-out of movement and one which takes four key frames, start frame, end frame and two intermediate frames depending on the values of **k1** and **k2**. As for the solution taking two keyframes, the total distance between the start and end frame is calculated and the value computed for **s**, in the equations above, is multiplied to the portion of distance representing the 0.01 time step to produce the 100 frames for the animation. This solution is very straightforward and produced reasonable results, that can be seen in Figure 11 and 12, and show the results of using the sin_interp.m function.

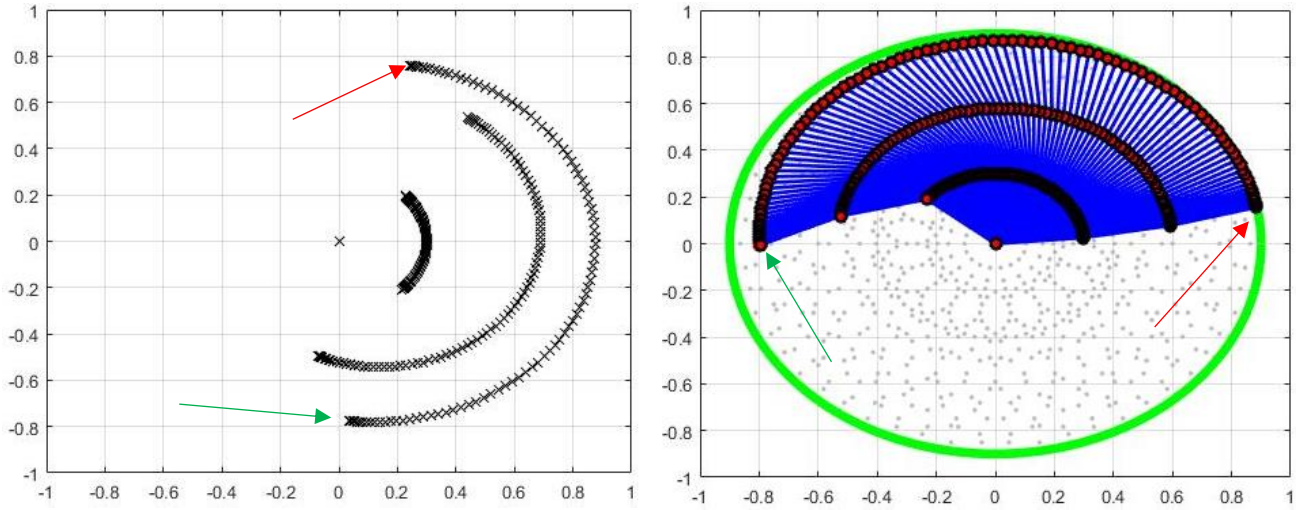

Figure 11: (Left) Shows the position of each joint as an 'x' for all the frames produce with sine interpolation. Figure 12: (Right) Shows all the frames produced including the whole linkage pose for each frames. Both figures were produced using k1 = 0.3 and k2 = 0.7. Notice how at the beginning and end of the movement, in both cases, the frames are denser producing the desired slow in\slow out of movement, with the red arrows indicating the start position of the end-effector and the green arrows indicating the end or goal position of the end-effector.

As for the second solution using four keyframes, the values of **k1** and **k2** were used to obtain an index of the intermediate frames to be used, from the frame array being inputted into the sin_interp2.m function. This method though does not account for the distance between these frames to be proportional to the values of **k1** and **k2**, therefore producing spastic results rather than a smooth movement as seen in the previous method of interpolation. This spastic behaviour is due to the negligence of the distance for each portion of the curve. This can be seen in the next figures, namely Figures 13 and 14.

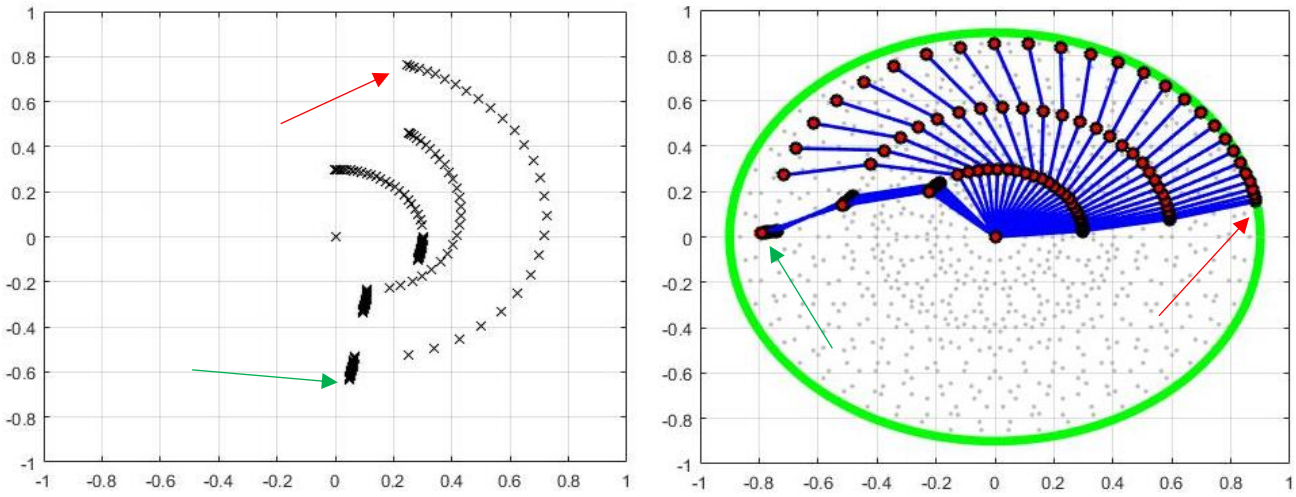

Figure 13: (Left) Shows the position of each joint as an 'x' for all the frames produce with sine interpolation using four keyframes. Figure 14: (Right) Shows all the frames produced including the whole linkage pose for each frames. Both figures were produced using k1 = 0.3 and k2 = 0.7. Notice how at the beginning and end of the movement, in both cases, the frames are denser producing slow in\slow out of movement but this time there is a spastic change due to the inconsistency in the distance compared to the total distance, with the red arrows indicating the start position of the end-effector and the green arrows indicating the end or goal position of the end- effector.

One thing worth mentioning, is that the interpolation of frames was performed on the angles of the joints rather than the position of the joints on the grid. This produced a more natural arc like movement and did not suffer from linear movement, as it would when using the joints' positions. Needless to say, interpolation using two keyframes is much smoother, more eye-friendly and more natural when compared to four keyframe interpolation.

6 Conclusion

Inverse kinematics, as proposed throughout this report, is a very challenging problem to tackle. Nonetheless, a successful implementation was produced and the aspects of this implementation are examined in the previous chapters. The mathematics evolved are provided and this coursework was both exciting and stressful to work on, but very satisfying when producing the desired results.

7 References

1. Kang, T. (2000). Solving Inverse Kinematics Constraint Problems for Highly Articulated Models. [online] pp.19-21. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.7145&rep=rep1&type=pdf.
2. Kumar, V. (2017). Introduction to Robot Geometry and Kinematics. [online] Chapter 5, pp.1-25. Available at: http://www.seas.upenn.edu/~meam520/notes02/IntroRobotKinematics5.pdf.
3. Meredith, M. and Maddock, S. (2005). Adapting motion capture data using weighted real-time inverse kinematics. *Computers in Entertainment*, 3(1), p.5.
4. Parent, R. (2012). *Computer animation*. Amsterdam: Morgan Kaufmann.
5. Thomas, F., Johnston, O. and Thomas, F. (1995). *The illusion of life*. New York: Hyperion.
6. Welman, C. (1994). *Inverse kinematics and geometric constraints for articulated figure manipulation*. Ottawa: National Library of Canada = Bibliothèque nationale du Canada.