Gradient optimization for robot inverse kinematics

Pramod Kotipalli[1]

***Abstract*— Inverse kinematics is one of the hardest problems in robotics. Given desired goal position and orientation for an end-effector, inverse kinematics seeks to find a suitable series of joint configurations to meet that goal. In this paper, we develop a forward kinematics model in Python and apply gradient-based methods using automatic differentiation to develop an inverse kinematics model of an arbitrary n-joint robot arm. We evaluate the performance of standard gradient descent against Nesterov Momentum gradient descent. We conclude with a discussion of the limitations of our 2D model and areas for extension to gain more fidelity with real-world robots.**

I. Introduction

Robotic arms are ubiquitous. These serial actuators help assemble cars on factory floors, help vacuum our floors, and even aid in film production.

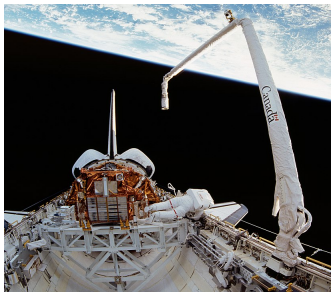

Fig. 1. The Canadarm on the NASA Space Shuttle [Wikipedia n.d.(a)]

For a robot arm to navigate to a goal position, the robot must first know the position of its links and the orientation of its joints. Together, this series of components determine the position of the robot arm's end-effector.

Given sufficient information about the position, orientation, and dynamics of these joints and links, the position and orientation of the end-effector can be computed directly with a series of matrix multiplications. This process is known as *forward kinematics*. Given a goal position for the end-effector, finding the position, orientation, and dynamics of intermediate joints and links is the process of *inverse kinematics*. This problem is one of the hardest problems in robotics; inverse kinematics forms a highly non-convex and discontinuous optimization problem.

In this paper we will investigate gradient descent algorithms applied to the problem of inverse kinematics. We will build a two dimensional forward kinematics model and simulation tool in the Python programming language. We then apply a Python-based automatic differentiation method to our forward kinematics model to provide gradient information for our gradient descent method. Doing so provides us with an inverse kinematics model.

The goal of inverse kinematics is to find a configuration of joint angles such that a goal position is reached by the end-effectors. As such, the top line metric to minimize is the distance between the end-effector and the goal position. Furthermore, mobile robots face practical constraints in terms of energy consumption and limited computation resources, both in terms of execution time and memory usage. We can approximate these resource costs through the number of iterations taken for the gradient descent method to converge.

II. Background

A. Forward kinematics

A robot arm is modeled as a series of links and joints. Links have a fixed length and are connected to each other by joints, either revolute joints (rotation around a common axis) or prismatic joints (providing a linear sliding movement). (See Figure 3.) We also assume that each joint only provides one degree-of-freedom. We discuss the geometry and kinematics modeling of serial robots based on [Touretzky 2017, Spong, Hutchinson, and Vidyasagar 2020].

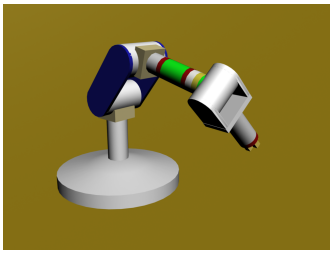

Fig. 2. A six degree-of-freedom serial robot arm [Wikipedia n.d.(b)]

A revolute joint is characterized by the angle between the two links it connects. The process of forward kinematics is to find the cumulative effect of all of these joint angles

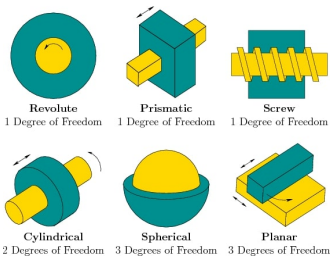

Fig. 3. Six different types of joints used in robotics [LaValle 2006]

[1]Pramod Kotipalli is with the Department of Computer Science, School of Engineering, Stanford University, Stanford, CA pramod.kotipalli@stanford.edu

on the position and orientation of the end-effector. A robot manipulator in n joints has $n+1$ links. Joint i connects link $i-1$ with link i. Joint 1 connects link 0 with link 1, namely the ground plane with the first link of the robot arm. We define the joint variable to be q_i which takes the angle of rotation, θ_i, for a revolute joint or the joint displacement, d_i, in the case of a prismatic joint.

The key modeling feature of forward kinematics is how we define and attach coordinate frames to each link in the series. Specifically, we rigidly attach a coordinate frame $o_i x_i y_i z_i$ to link i such that the coordinates of any point on link i is constant when expressed in i^{th} coordinate frame. When joint i is actuated, link i and its corresponding coordinate frame $o_i x_i y_i z_i$ move together.

We now have coordinate frames attached to each joint in the robot. Each joint variable q_i defines a homogeneous coordinate transform (as matrix A_i) from coordinate frame $i-1$ (i.e. $o_{i-1}x_{i-1}y_{i-1}z_{i-1}$) to coordinate frame i (i.e. $o_i x_i y_i z_i$). The transformation matrix A_i is a function of the joint variable q_i. As such, $A_i = A_i(q_i)$, a notation simplification. The coordinate transform from link i to link j is given by $T_j^i = A_{i+1} \cdots A_j$ which can equivalently be expressed in $\mathbb{R}^{4\times4}$ composed of a rotation matrix $R_j^i \in \mathbb{R}^{3\times3}$ and position base frame $O_j^i \in \mathbb{R}^3$:

$$T_j^i = \begin{bmatrix} R_j^i & O_j^i \\ \vec{0} & 1 \end{bmatrix} \tag{1}$$

The coordinate transform from link 0 (attached to the robot's fixed origin reference frame) to link n (the end-effector) is notated by T_n^0. This is the formulation of forward kinematics. We define the function `fk` to take in the joint variables vector $\vec{q}$ and link lengths vector $\vec{l}$ and to return the position vector of the end-effector.

B. Inverse kinematics

Forward kinematics produces the position $\texttt{goal} = \vec{g} = O_n^0$ of the end-effector given the joint variables $\vec{q} \in \mathbb{R}^{n\times d}$ for $d \in \{2, 3\}$ (i.e. in two or three dimensions). Inverse kinematics produces the joint variables $\vec{q}$ given a target end-effector position O_n^0. We are tasked with finding the series of matrices $A_i(q_i) = A_i \in \mathbb{R}^{4\times4} \forall i \in \{1, \ldots, n\}$ that when multiplied with the design variables q_i will produce the composite transformation T_j^i that (1) respects the kinematic constraints of the robot (e.g. link lengths and rotation constraints) and (2) reaches the end-effector to the goal position $\vec{g}$.

C. Inverse kinematics as an optimization problem

Solving for the joint positions and orientations requires trigonometric modeling. The inclusion of sine and cosine terms in this modeling make inverse kinematics a highly non-convex problem. As such, neither constrained optimization nor linear programming are suitable options to solve this inverse problem. Inverse kinematics in its general, non-simplified form construes a non-convex optimization method where it is generally impossible to find a globally-optimum solution.

To use a gradient-based method, we must have first-order information, namely the Jacobian of the forward function $\vec{J}_{\texttt{fk}}$. (With the Hessian $\vec{H}_{\texttt{fk}}$ of the forward function, we can use second-order methods.) Computing the gradient function of `fk` provides us with sufficient information to find a local optima joint variable vector $\vec{q}$ given the link lengths vector $\vec{L}$ and a goal position.

D. Automatic differentiation for gradient information

Automatic differentiation techniques use the chain rule of calculus to numerically evaluate the derivatives of functions defined by a computer program. Because a computer program is ultimately composed of elementary operations, such as addition and division, we can apply the chain rule several times to find the derivative of a computer program [Kochenderfer and Wheeler 2019].

We represent the forward kinematics process in a computational graph, namely through a Python function definition `fk`. We use the third-party Python library `autograd` which was designed to differentiate programs written in Python and `numpy` code. (`numpy` is a popular third-party numerical computation library for Python.)

III. Implementation

[Miranda 2017, Morais 2019] aided in our implementation of the forward and inverse kinematic modeling.

A. Kinematic modeling

We use Python 3 to model and compute forward kinematics function `fk`. We model this problem in two dimensions. The transformation matrix $T_j^i \in \mathbb{R}^{3\times3}$ combines a rotation matrix $R_j^i \in \mathbb{R}^{3\times3}$ and base position vector $O_j^i \in \mathbb{R}^2$ and decomposes into a series of matrices $\vec{A}$. For $i = 0$ and $j = n$, we can state the forward kinematics problem as:

$$T_n^0 = \begin{bmatrix} R_n^0 & O_n^0 \\ \vec{0} & 1 \end{bmatrix} = A_1^0 \cdots A_n^{n-1} \tag{2}$$

Each update matrix takes the form:

$$A_i^{i-1} = \begin{bmatrix} \cos\theta & -\sin\theta & O_i^{i-1}(0) \\ \sin\theta & \cos\theta & O_i^{i-1}(1) \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

We must then compute the Jacobian of the forward function, as $\vec{J}_{\texttt{fk}} = \texttt{jacobian(fk)}$. We use the Jacobian to inform our gradient descent search.

The cost function is defined as the distance between the goal position $\texttt{goal} = \vec{g}$ and result of the forward kinematics model for a given joint configuration $\vec{q}$ [Morais 2019]:

$$\texttt{cost}_{\texttt{goal}}(\vec{q}) = \|\texttt{goal} - \texttt{fk}(\vec{q})\| \tag{4}$$

B. Automatic differentiation with Python `autograd`

We then use the `autograd` [Maclaurin, Duvenaud, and Adams 2015] Python library to find the Autograd is a third-party Python library that computes the derivative of a vector-input, scalar-output function specified with Python and `numpy`. We use the `autograd.grad` gradient function as part of the gradient descent algorithm:

$$\vec{J}_{\texttt{fk}} = \texttt{autograd.grad(fk)} \tag{5}$$

We formulate the optimization problem as one of the cost function, yielding our inverse kinematics solution function:

$$\texttt{ik(goal)} = \arg\min_{\vec{q}} \texttt{cost}_{\texttt{goal}}(\vec{q}) \tag{6}$$

IV. Evaluation

We would like to understand the accuracy and performance of our inverse kinematics model `ik`. We specify various parameters of the algorithm and report the cost value and gradient descent (g.d.) iteration counts and Nesterov Momentum g.d. (n.g.d.) iteration counts. We used a fixed seed value in `numpy` to ensure reproducible results. We used an epsilon of $\epsilon = 0.001$ as the threshold to terminate our gradient descent methods.

Table IV: Method results

n	$\texttt{goal} = \vec{g}$	g.d. iters.	n.g.d iters.
2	$[\sqrt{2}, \sqrt{2}]^T$	3047	255
3	$[2, 1]^T$	693	40
4	$[3, 2]^T$	1111	48
5	$[3, 3]^T$	742	30
6	$[2, 3]^T$	346	53

For every robot size and goal position evaluated, n.g.d. performed about 10 times faster than g.d. to converge to a solution. The momentum factor in n.g.d. allowed the optimization method to move quickly through an initially-"flat" configuration space to more quickly find a valid locally-optimum solution.

The initial joint configuration angles are randomly initialized from $[0, 2\pi)$. At each iteration of the gradient descent method, a new frame is rendered showing the optimal joint configuration for that time in the optimization process. In the simulation, we view the robot arm's joints slowly converging to a local optima configuration until the epsilon value is exceeded during gradient updates.

We visualize the joint configurations resulting from Table IV in Figures 4, 5, 6, 7, and 8.

V. Limitations and Future Work

A. Operating in 3D

This implementation of inverse kinematics is simulated in two-dimensions. However, most robotic arms operate in three-dimensions utilizing a different update matrix $A_i^{i-1} \in \mathbb{R}^{4\times4}$ with a rotation matrix $R_i^{i-1} \in \mathbb{R}^{3\times3}$ and base position

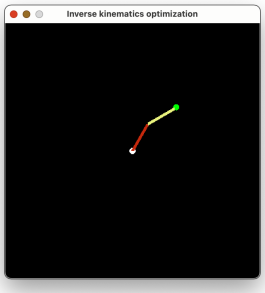

Fig. 4. $n = 2$, $\vec{g} = [\sqrt{2}, \sqrt{2}]^T$

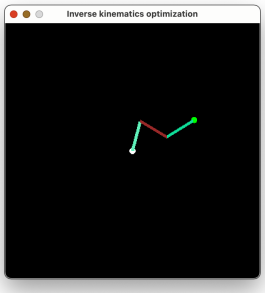

Fig. 5. $n = 3$, $\vec{g} = [2, 1]^T$

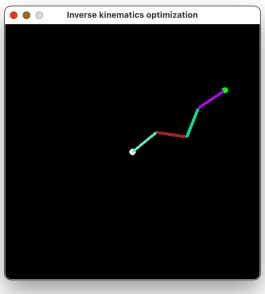

Fig. 6. $n = 4$, $\vec{g} = [3, 2]^T$

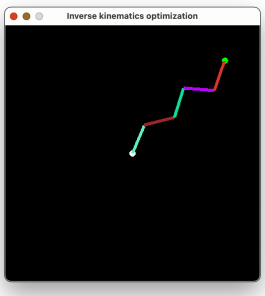

Fig. 7. $n = 5$, $\vec{g} = [3, 3]^T$

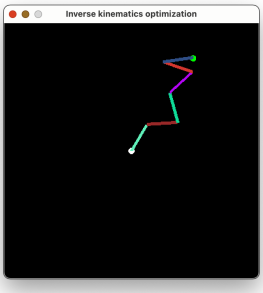

Fig. 8. $n = 6$, $\vec{g} = [2, 3]^T$

vector $O_i^{i-1} \in \mathbb{R}^3$. This 2D model can easily be extended to 3D by updating the `fk` method appropriately. The gradient descent method does not need to be changed to account for this new problem formulation.

Typically, robots' joints are constrained in their movement. For example, one rotation joint may only be able to operate in $[0, \pi)$ due to its construction. The Stanford Arm (Figure 9) is a classic example of a serial manipulator robot arm.

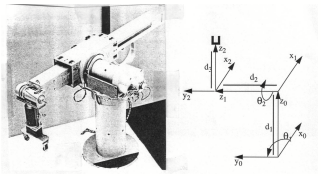

Fig. 9. The Stanford Arm [Allen 2015]

B. Denavit-Hartenberg convention for 3D kinematic chains

Every joint of this robot is constrained in its motion. Using the Denavit-Hartenberg (DH) convention, we are able to model the joints' coordinate frames and define the update matrix A_i^{i-1} as a product of two rotation matrices and two transformation matrices. The DH convention makes a few simplifying assumptions about the robot's kinematic constraints allowing for most serial robots to be modeling in 3D with DH rules. More details can be found in [Spong, Hutchinson, and Vidyasagar 2020: Ch. 3-4]. 3D modeling with DH conventions is one area of extension for this paper's work.

Through its design, a serial robotic arm provides *a priori* information about link lengths, joint angle constraints, and coordinate frame displacements which are all encoded into the DH matrix. An optimization method then operates on the joint angle configuration variables in this matrix to find a locally-optimal joint configuration.

C. Path constraints and optimization

Moving an end-effector from an initial position $\vec{o}$ to a goal position $\vec{g}$ is usually not sufficient for a robot to operate in the real world. Robots are constrained by energy requirements, weight requirements, and torque output limits among other factors. Further, the path that the robot's joints can move in space must also be constrained to avoid self-intersection and to avoid obstacles. Incorporating these constraints into this inverse kinematic model would make this system more useful in real-world contexts.

VI. Conclusion

In this paper, we develop a rigorous model for the forward kinematics process for serial robot arms. We employ automatic differentiation to provide gradient information about the forward process to inform the gradient descent method used to solve the inverse kinematics process. We compare standard gradient descent with Nesterov gradient descent applied to five instances of the inverse problem. We discuss methods for augmenting our 2D simulation into 3D while incorporating real-world constraints imposed on physical robots.

References

[All15] Peter K. Allen. *CS 4733, Class Notes: Forward Kinematics I*. `http://www1.cs.columbia.edu/~allen/F15/NOTES/forwardkin2.pdf`. 2015.

[KW19] Mykel J Kochenderfer and Tim A Wheeler. *Algorithms for optimization*. Mit Press, 2019.

[LaV06] S.M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006. ISBN: 9781139455176. URL: `https://books.google.com/books?id=-PwLBAAAQBAJ`.

[MDA15] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. "Autograd: Effortless gradients in numpy". In: *ICML 2015 AutoML Workshop*. Vol. 238. 2015, p. 5.

[Mir17] LJ Miranda. *Solving the Inverse Kinematics problem using Particle Swarm Optimization*. `https://ljvmiranda921.github.io/notebook/2017/02/04/inverse-kinematics-pso/`. 2017.

[Mor19] Pedro Morais. *py-k*. `https://github.com/p-morais/py-k`. 2019.

[SHV20] M.W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. Wiley, 2020. ISBN: 9781119524076. URL: `https://books.google.com/books?id=WC7PDwAAQBAJ`.

[Tou17] Dave Touretzky. *Cognitive robotics: kinematics*. `https://www.cs.cmu.edu/afs/cs/academic/class/15494-s17/lectures/kinematics/kinematics.pdf`. 2017.

[Wika] Wikipedia. *Canadarm*. `https://en.wikipedia.org/wiki/Canadarm`.

[Wikb] Wikipedia. *Serial manipulator*. `https://en.wikipedia.org/wiki/Serial_manipulator`.