Levenberg-Marquardt Algorithm in Robotic Controls

Isaac Burton Love

Advisor: Dr. James Morrow

Department of Mathematics, University of Washington

loveisa@uw.edu

isaacblove@gmail.com

May 2020

Abstract

Use of the Levenberg-Marquardt damped least squares method in inverse kinematics has a relatively short history. This method avoids some of the complications around singularities that can occur in robot control problems and results in a relatively stable solution to the instructions controlling how much each robot part moves. The paper presents some of the history of the algorithm, provides definitions and terminology for robotics and inverse kinematics, explores some of the math used to frame an inverse kinematics problem as an optimization, and, finally, provides and fully works a simple example problem. The example problem comes with accompanying C code which runs the algorithm to solve the simple inverse kinematic problem.

Introduction

The Levenberg-Marquardt technique has been widely used in a number of scientific fields. This solution method belongs to a class of numerical solution techniques for problems that cannot be solved analytically and so must be approached numerically. Typically, these problems involve an objective function that is to be minimized with respect to a set of variables. The analytic solution is often beyond reach due to the number of variables and the complicated curvature of the solution space.

The general pattern for iterative solution algorithms is simply and clearly laid out in Judge, et al. [9] and in Nocedal and Wright [15]. The problem begins with an objective function to be optimized (minimized or maximized), $M(x)$, a set of defined variables that will optimize the function, x, and an initial guess at a solution values for those variables, x_0. The value of the objective function is then calculated at the initial values $M_0 = M(x_0)$. The first iteration requires a second value for variables, x_1. To obtain this next set of variables, a step size is chosen, usually very small ϵ. The value of x_0 is incremented by ϵ and the value of the objective function is again calculated as $M_1 = M(x_1)$. If the new objective value is less than the old one, $M_1 < M_0$, in a minimization framework, and if the difference between M_0 and M_1 is sufficiently small, the process is over. If not, the variables are again incremented, a new objective value is calculated and the iterative process continues until the difference between objective function values is less than

the tolerance value.

The choice of exactly how to increment the variables has produced an array of suggested approaches to both direction to move from x_0 and the step size. One of the oldest is Newton-Raphson. This idea goes back to Isaac Newton, who proposed it in 1669 as a method to solve for roots of polynomials in an iterative fashion [17]. Joseph Raphson adapted and simplified Newton's approach in 1690 to focus on incrementing the variables x. In 1740, Thomas Simpson developed Newton's method as an iterative approach to solving general nonlinear equations and pointed out that the method can be used to solve optimization problems by setting the gradient equal to zero [17].

There is an excellent summary of the most used methods in Judge, et al. [9]. Gradient methods choose direction through selection of some form of steepest descent, where the increment to the variables moves them in the direction that most reduces the objective function within the allowable step size. In the simplest case that gets most quickly to the minimum, the variables are incremented with a large step size in the direction of steepest descent, but this method can go awry in complicated curvatures. Other methods tend to modify the steepest descent approach. Newton-Raphson uses the inverse of the Hessian matrix to determine the direction of each sequential increment [9] [15]. The disadvantages of Newton-Raphson are the need to come up with a formula involving analytic derivatives for the Hessian and that the Hessian may not be positive definite (required for a minimum) except in a very small neighborhood of the true optimum point.

Many other methods are focused on approximations of the Hessian or ways to approximate or linearize the problem. For example, Gauss-Newton is essentially a sequence of linear approximations of the nonlinear model that are estimated using least squares [9] [15]. The Levenberg-Marquardt method is cast between the pure steepest descent approach for large step sizes and the Gauss-Newton approach for small step sizes [9].

Levenberg

In 1943, Kenneth Levenberg of the Frankford Arsenal read a paper at the Annual Meeting of the American Mathematical Society in Chicago [10]. The paper was published as a Note in 1944. In this paper, Dr. Levenberg proposed the idea of damped least squares as an extension of Newton's method. He formulated the problem as one that had a set of nonlinear simultaneous equations $h(x)$ in variables x that can be approximated by $H(x)$. The next step is to calculate residuals, $f(x) = H(x) - h(x)$. Then, the least squares criterion requires minimization of

$$s(x) = \sum_{1}^{n} f_i^2(x), \tag{1}$$

in which the weighting on the squares is considered to be uniformly 1. Taking a Taylor series expansion around the initial point p_0 gives a set of linear approximations of the residuals $F_i(x)$, which can be summed

$$S(x) = \sum_{1}^{n} F_i^2(x). \tag{2}$$

This sum is minimized by setting the partial derivatives of $S(x)$ with respect to variables x equal to zero and solving.

Levenberg points out that this method may yield changes that are so large as to invalidate the approximations and result in huge step sizes. At this point, he suggests that it may be useful to limit or "damp" [10] the absolute values of the increments, which would improve the approximation and also minimize the sum of squared residuals. In particular, Levenberg suggests minimizing

$$\overline{S} = \omega S(x) + a(\Delta x_1)^2 + b(\Delta x_2)^2 + ..., \tag{3}$$

where $x_1, x_2, ...$ are components of x, where $x = [x_1, x_2, ...]^T$; $a, b, ...$ are weights expressing the relative weights of the different increments and indicate damping; and ω is a weighting factor on the sum of the residuals. Levenberg proceeds to show that this approach minimizes both the objective function and the sum of the squared residuals. He asserts that this weighted or damped least squares approach will allow least squares methods to be applied to problems beyond those to which the method was generally applied in 1944.

Marquardt

Donald Marquardt, an engineer from E.I. du Pont de Nemours & Co, published a paper in 1963 [12] that developed proofs underlying a method for estimating nonlinear parameters in a set of equations that was essentially as proposed by Levenberg. Hence, the technique is now known as the Levenberg-Marquardt method. In addition to providing proofs, Marquardt's approach addresses the issue of the step size carrying the estimates beyond the appropriate neighborhood of linear approximation of the nonlinear function. This is done by using a damping factor to shift the algorithm to be more like either gradient descent or Newton-Raphson. The algorithm proposed by Marquardt is like the Newton-Raphson algorithm with the inclusion of a damping factor. The scaling factor is adjusted at each iteration to ensure that the next estimate β results in a better value. The following is adapted from Marquardt [12] and tells how to update the damping factor. For a complete overview and walkthrough of the algorithm, see the Simple Example section below.

1. Let $v > 1$. v is the scaling factor.

2. Let λ_{r-1} denote λ from the previous iteration. λ is damping factor and r is iteration number.

3. Compute the objective function Φ using the last damping factor and the scaled damping factor $\Phi(\lambda_{r-1})$ and $\Phi(\lambda_{r-1}/v)$. To do this, the

previous estimate should be used to compute a step. Φ should be evaluated at the estimate produced by adding the computed step to the previous estimate. Please see the Simple Example section below for a more detailed explanation.

4. Compare:

   (a) If $\Phi(\lambda_{r-1}/v) \leq \Phi_r$, let $\lambda_r = \lambda_{r-1}/v$.

   (b) If $\Phi(\lambda_{r-1}/v) > \Phi_r$ and $\Phi(\lambda_{r-1}) \leq \Phi_r$, let $\lambda_r = \lambda_{r-1}$.

   (c) If $\Phi(\lambda_{r-1}/v) > \Phi_r$ and $\Phi(\lambda_{r-1}) > \Phi_r$, increase λ by successive multiplication by v until for some smallest w, $\Phi(\lambda_{r-1}v^w) \leq \Phi_r$. Then let $\lambda_r = \lambda_{r-1}v^w$.

The damping factor is scaled at each step taken to ensure that the objective function continues to decline in value and the step taken remains in a valid linear approximation of the function [12].

The method is now widely known as Levenberg-Marquardt and has been applied in many disciplines. It has been adapted and augmented over the years. In days where computer power and size were small, there was work on how to better linearize and specify problems for greater efficiency. For example, More [13] presents a comparison of various iterative solution methods and reports the results in terms of computer time and the number of iterations before convergence. More recently, Huang and Ma evaluated the performance of algorithms involving different forms of the Levenberg-Marquardt parameter in terms of global convergence and computational efficiency [7].

Robotics

To understand the problem of controlling a robotic arm, one must first understand the terminology of robotic arms [2]. Any arm, robotic or otherwise, consists of a few fundamental building blocks: links, joints, and end-effectors. Links are the rigid structure of an arm. Joints are the connective elements which join links together. Generally, a joint joins exactly two links together, though a joint may join as many links as desired. A joint may also join a link with a fixed body or to the end-effector. The end-effector is typically, as the name implies, at the end of the arm. It is the manipulator or tool which the arm moves and utilizes. In a human body, the upper arm and forearm are links, the shoulder, elbow, and wrist are joints, and the hand is the end-effector. The torso could be considered the fixed body to which the arm is attached via the shoulder.

In the field of robotics, there are two kinds of common joints: revolute, and prismatic. Revolute joints are those that allow for rotation between the joined links. Prismatic joints allow for a sliding motion between the joined links. For simplicity, all joints are assumed to be revolute in this paper. Every joint will have a defined range of motion determined by the physical properties of the joint. The location of that joint at any given time within that range will be called its position [2].

The number of degrees of freedom (DOF) refers to the number of dimensions in which a joint or the arm as a whole may vary. For example, a human

elbow has one degree of freedom because it can only flex in one dimension, while a shoulder has three degrees of freedom because the shoulder can move the upper arm in three dimensions. The total degrees-of-freedom for the entire arm is given as the sum of the degrees-of-freedom from each joint.

The configuration of a link or end-effector consists of its three-dimensional cartesian coordinates and its three Euler angles (yaw, pitch, roll) relative to an external coordinate frame [22]. The Euler angles quantify a three-dimensional rotation relative to some frame. A configuration can therefore be represented as a pair of a vector, $p \in \mathbb{R}^3$ for cartesian position and a 3D rotation matrix $R \in SO(3)$. The configuration of a robot arm refers to the configurations of each link and end-effector in the arm. In a humanoid robotic arm, this will be two links, forearm and upper arm, and the hand.

The configuration of a joint refers to its position within its range of motion, typically measured in radians. The joint configuration vector is the vector of configurations of all joints in the arm.

Figure 1 is a picture of the PR2 robot from Willow Garage [23]. It stands roughly 5 feet tall and its arms are roughly human sized. The arms of the PR2 have two links each. The figure shows that the shoulder has three joints totaling to three degrees of freedom to move. The elbow and wrist only have two joints each: a flex joint and a roll joint. The end-effector or hand is a gripping claw attached to the forearm link by the wrist joints. Thus, the PR2 has seven total degrees of freedom in its arm: three in the shoulder, two in the elbow, and two in the wrist.

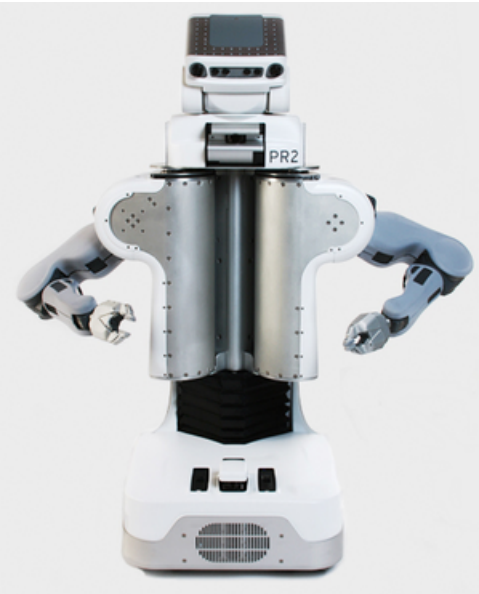

Figure 1: Willow Garage Robot PR2

For a robotic arm, a joint configuration vector implies the position of the links and end-effectors. Kinematics is the mathematics describing the relationship between the configuration of the joints and the configuration of the links and end-effectors. Forward kinematics involves calculating the configuration of a robot arm based on a joint configuration vector. Inverse kinematics involves calculating the joint configuration vector based on a desired robot arm configuration. A kinematic chain captures the concept that the links in an arm are connected and movement of one link affects the others. The base of the chain is connected to a fixed frame (e.g. the torso). If the upper arm is moved, the forearm and hand also move in a kinematic chain.

The workspace of a robotic arm refers to the set of all possible configurations of the end-effector. This space can be highly complex and is almost never convex for robot arms with several joints and links.

An important problem arises when computing inverse kinematics solutions. Often, the Jacobian matrix is used to estimate how changes in the joint configuration effect the configuration of the end-effector. In some situations, the Jacobian matrix may be singular, or non-invertible. Regions in the workspace where this occurs are referred to as singularities. The two main causes of singularities occur when the configuration of the arm causes rotation axes for joints to line up and when a particular joint reaches a bound of its motion. The former occurs in a human arm. If a person holds his arm out straight, the hand can be rotated either by rotating the shoulder or by rotating the wrist. The latter can occur in many places in the workspace and can also cause a phenomenon often called stuttering. In practice, many joints have a 2π range of motion and many have a smaller range with bounds causing discontinuities. Stuttering refers to situations where the inverse kinematics solutions for a particular desired end-effector configuration bounce back and forth on either side of the discontinuity. For example, with an out-streched arm, a person can face his palm upward either by rotating the entire arm all the way clockwise or all the way counter clockwise. A motion path bouncing between the two solutions would cause the robot arm to jerk or stutter.

Singularities present tough challenges for any proposed inverse kinemat-

ics method. Many solution methods see success away from singularities, but have undesirable behavior near them. The LM method previously discussed has important properties that make it robust around singularities. Importantly, the LM method does not require the Jacobian matrix to be inverted or pseudo-inverted. Further, the weighting factors of the LM method can bias the algorithm to avoid problems like stuttering.

Levenberg-Marquardt in Robotics

The Levenberg-Marquardt method has been applied in robotics, particularly to deal with singularities and boundary problems in robot control. A nice survey of methods and related applications is in Buss [2]. In the 1980s, science was still very concerned about computational efficiency and the ability of an algorithm to converge, or at least not crash. For example, Nakamura and Hanafusa [14] introduce a damping factor into a use a singularity-robust inverse matrix to resolve the configuration problems around a singularity. Wampler [8] proposes a damped least squares algorithm to address rank-deficiency and singularity issues as well as minimize the number of iterations in the context of computing end-effector velocity in robotics.

One of the best expositions of the application to robotics is in Sugihara [21]. The following general formulation is adapted from Sugihara [21]. Define $\mathbf{q} \in \Omega$ as a vector of n joint configurations for the robot:

$$\mathbf{q} = [q_1, q_2, q_3, ...q_n]^T \in \mathbb{R}^n, \tag{4}$$

where n is the number of joints. Ω is the set of all possible joint configurations. Typically, Ω bounds each joint to a range of positions, i.e. $\mathbf{q} \in \Omega \iff q_i \in [a_i, b_i]$ for $a_i \leq b_i$.

Let the vector-valued function $\mathbf{p_i}(\mathbf{q})$ describe the cartesian position for the configuration of link i. Let $\mathbf{R_i}(\mathbf{q})$ be a matrix-valued function describing the rotation of link i in $\mathbf{SO(3)}$, or special orthogonal group of order 3, a way of defining parameters in rotational space. Both the position and orientation are given in terms of a fixed coordinate frame. These two functions define the forward kinematics of link i relative to an external frame. Thus, the forward kinematic problem has a geometric interpretation and formula.

The inverse problem consists of taking a set of position and orientation goals for the end-effectors and finding a $\mathbf{q} \in \Omega$ that most closely achieves those goals.

To solve the inverse kinematic problem, some way to measure the distance from the goals and joint configuration $\mathbf{q} \in \Omega$ is required. This is done using residuals $\mathbf{e_i} \in \mathbb{R}^\mathbf{3}$:

$$\mathbf{e_i}(\mathbf{q}) = \begin{cases} {}^d\mathbf{p_i} - \mathbf{p_i}(\mathbf{q}) & \text{if } i \text{ is a position goal} \\ \alpha({}^\mathbf{d}\mathbf{R_i}\mathbf{R_i}^\mathbf{T}(\mathbf{q})) & \text{if } i \text{ is an orientation goal,} \end{cases} \tag{5}$$

where ${}^\mathbf{d}\mathbf{p_i} \in \mathbb{R}^\mathbf{3}$ is a vector of desired joint positions, $\mathbf{p_i}(\mathbf{q}) \in \mathbb{R}^\mathbf{3}$ is a vector of positions, and the difference gives a position goal residual $\mathbf{e_i}$. Alternatively, if i corresponds to an orientation goal, the desired set of ori-

entations ${}^{\mathbf{d}}\mathbf{R_i} \in \mathbf{SO(3)}$ for the joints is compared to current orientations $\mathbf{R_i(q)} \in \mathbf{SO(3)}$. The vector valued function $\alpha(\mathbf{R}) \in \mathbb{R}^3$, for any arbitrary $\mathbf{R} \in \mathbf{SO(3)}$, gives the Euler axis-angle vector. The Euler axis-angle vector is a vector $\mathbf{r} \in \mathbb{R}^3$ such that $\mathbf{r} = \theta\mathbf{k}$, where $\mathbf{k}$ is a unit vector representing the rotation axis and θ is the rotation angle clockwise about the axis. By noting that $\mathbf{R_i}^{\mathbf{T}}\mathbf{(q)} = \mathbf{R_i}^{-\mathbf{1}}\mathbf{(q)}$, it can be seen that $\alpha({}^{\mathbf{d}}\mathbf{R_i}\mathbf{R_i}^{\mathbf{T}}\mathbf{(q)})$ computes the axis-angle vector for the rotation between ${}^{\mathbf{d}}\mathbf{R_i}$ and $\mathbf{R_i}$. While computing residuals for position goals is straightforward, computing residuals for orientation goals is more complex. The following is adapted from Appendix A in Sugihara [21].

Given an orientation matrix $\mathbf{R} = \{\mathbf{r_{i,j}}\} \in \mathbf{SO(3)}$ for $i = 1, 2, 3$ and $j = 1, 2, 3$, it can be represented as a rotation of an angle θ about an axis $\mathbf{k} \in \mathbb{R}^3$. This relationship is described by Rodrigues' rotation formula. Let the vector $\mathbf{k}$ be of unit length and given as $\mathbf{k} = [k_1, k_2, k_3]^T$, and let the matrix $\mathbf{K}$ be defined as follows:

$$\mathbf{K} = \begin{bmatrix} 0 & -k_3 & k_2 \\ k_3 & 0 & -k_1 \\ -k_2 & k_1 & 0 \end{bmatrix}. \tag{6}$$

This denotes the cross-product matrix for the unit vector $\mathbf{k}$. Rodrigues' rotation formula, given in equation 7, relates the rotation matrix $\mathbf{R}$ to the vector $\mathbf{k}$ and angle θ using the cross-product matrix for $\mathbf{k}$:

$$\mathbf{R} = \mathbf{I} + \sin\theta\mathbf{K} + (1 - \cos\theta)\mathbf{K}^2. \tag{7}$$

The goal, given an $\mathbf{R} \in \mathbf{SO(3)}$, is to find $\mathbf{k}$ and θ that satisfy 7 and then compute the function given in 8, which gives the axis-angle vector for the rotation:

$$\alpha(\mathbf{R}) = \theta\mathbf{k}. \tag{8}$$

This function has a few notable special properties. Notice in particular that $||\alpha(\mathbf{R})|| = \theta$. Therefore, $||\alpha(\mathbf{R})|| = 0 \iff \mathbf{R}$ is the identity rotation (i.e. $\mathbf{R} = \mathbf{I}$). To compute $\alpha(\mathbf{R})$, first check that the matrix $\mathbf{R}$ is not the identity matrix. If it is, then $\theta = 0$, and so $\alpha(\mathbf{R}) = \mathbf{0}$, where $\mathbf{0} \in \mathbb{R}^3$ is the 0-vector. Otherwise, the axis of rotation $\mathbf{k}$ and rotation angle θ must be computed. These can be computed using Rodrigues' rotation formula. In particular, note that multiplying on both sides by the vector $\mathbf{k}$ gives $\mathbf{Rk} = \mathbf{k}$, so solving the system $(\mathbf{R} - \mathbf{I})\mathbf{k} = \mathbf{0}$ for $\mathbf{k}$ yields the desired vector. By noting additionally that $\mathbf{K}^2 = -(\mathbf{I} - \mathbf{kk^T})$, θ can be computed by taking the trace of both sides of Rodrigues' rotation formula [18]:

$$\begin{aligned} \text{trace}(\mathbf{R}) &= \text{trace}(\mathbf{I}) + \sin\theta\text{trace}(\mathbf{K}) + (1 - \cos\theta)\text{trace}(\mathbf{K}^2) \\ &= \text{trace}(\mathbf{I}) + (\cos\theta - 1)\text{trace}(\mathbf{I} - \mathbf{kk^T}) \\ &= 1 + 2\cos\theta. \end{aligned} \tag{9}$$

The final equality comes from the fact that for a vector of unit length $\mathbf{u}$, trace$(\mathbf{u}\mathbf{u}^{\mathbf{T}}) = 1$.

However, in a common special case, there is another way to compute $\alpha(\mathbf{R})$. Suppose that $\mathbf{R} \in \mathbf{SO(3)}$, and that $\mathbf{R}$ is not symmetric. Then, define the vector $\mathbf{t}$ as given by 10:

$$\mathbf{t} = \begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix}. \tag{10}$$

Since $\mathbf{R}$ is not symmetric, $\mathbf{t}$ is not zero. It will be shown that α is given by 11.

$$\alpha(\mathbf{R}) = \frac{atan2(\|\mathbf{t}\|, r_{11} + r_{22} + r_{33} - 1)}{\|\mathbf{t}\|}\mathbf{t}, \tag{11}$$

where function $atan2(y, x)$ is defined as the angle in the Euclidean plane, given in radians, between the positive x axis and the ray to the point $(x, y) \neq (0, 0)$ [16].

Rodrigues' rotation formula is used to prove the equality in 11. First, subtract $\mathbf{R}^{\mathbf{T}}$ from both sides:

$$\begin{aligned}\mathbf{R} - \mathbf{R}^{\mathbf{T}} &= \mathbf{I} + \sin\theta\mathbf{K} + (1-\cos\theta)\mathbf{K}^2 - (\mathbf{I} + \sin\theta\mathbf{K} + (1-\cos\theta)\mathbf{K}^2)^{\mathbf{T}} \\ &= \sin\theta\mathbf{K} - \sin\theta\mathbf{K}^{\mathbf{T}} \\ &= 2\sin\theta\mathbf{K}.\end{aligned} \tag{12}$$

Notice that equation 12 shows that $\mathbf{t} = 2\sin\theta\mathbf{k}$. Thus, $||\mathbf{t}|| = 2\sin\theta$, which is the y-argument of the $atan2$ function in 11. Recall, also that from equation 9, that the x-argument of the $atan2$ function in 11 is equivalent to $2\cos\theta$. Combining gets the desired result:

$$\begin{aligned}\frac{atan2(\|\mathbf{t}\|, r_{11}+r_{22}+r_{33}-1)}{\|\mathbf{t}\|}\mathbf{t} &= \\ \frac{atan2(2\sin\theta, 2\cos\theta)}{\|\mathbf{t}\|}\mathbf{t} &= \\ \frac{\theta}{\|\mathbf{t}\|}\mathbf{t} &= \\ \theta\mathbf{k} &= \alpha(\mathbf{R}).\end{aligned} \tag{13}$$

There are also special cases if $\mathbf{R}$ is diagonal. There are only four such cases, including the identity:

$$(r_{11}, r_{22}, r_{33}) \in \{(1,1,1), (1,-1,-1), (-1,1,-1), (-1,-1,1)\}. \tag{14}$$

In the case of $(1,1,1)$, $\alpha = \mathbf{0}$, as discussed above. In the other three cases,

$$\alpha(\mathbf{R}) = \frac{\pi}{2} \begin{bmatrix} r_{11}+1 \\ r_{22}+1 \\ r_{33}+1 \end{bmatrix}. \tag{15}$$

Returning to equation 5, the total number of equations to be simultaneously solved is $3m$ for m end-effector position and/or orientation goals. Recall that each constraint as described by 5 is a vector in $\mathbb{R}^3$. The residual vector is therefore $\mathbf{e}(\mathbf{q}) \in \mathbb{R}^{\mathbf{3m}}$. If the inverse kinematic problem has a solution, then the goal is to solve 16 for $\mathbf{q}$:

$$\mathbf{e}(\mathbf{q}) = [\mathbf{e_1^T}(\mathbf{q}),\ \mathbf{e_2^T}(\mathbf{q}),\ ...,\ \mathbf{e_m^T}(\mathbf{q})]^{\mathbf{T}} = \mathbf{0}. \tag{16}$$

.

However, a solution to equation 16 is not always tenable. For example, a position goal may lie beyond the workspace of the arm, or may be mutually exclusive with the orientation goal. In such cases, the problem can be rephrased as a least squares minimization problem, as in 17:

$$E(\mathbf{q}) = \frac{\mathbf{1}}{\mathbf{2}}\mathbf{e^T W_E e}, \tag{17}$$

where $\mathbf{W_E} = diag\{w_{E,i}\}, w_{E,i} > 0\ \forall i = 1,2,3$, is the weighting matrix reflecting the priority of each goal. Typically, the solution to least squares problems, such as those of the form of 17, has been sought numerically using NR (Newton-Raphson) techniques that begin at an initial value $\mathbf{q_0}$ and

update the solution point,

$$\mathbf{q_{k+1}} = \mathbf{q_k} - \nabla \mathbf{e}(\mathbf{q_k})^{-1}\mathbf{e_k}, \tag{18}$$

where $\mathbf{e_k} = \mathbf{e}(\mathbf{q_k})$ and k is the iteration step.

An aside on iterative solution methods using the Newton-Raphson approach comes from the seminal book *Numerical Recipes* [19]. In Newton-Raphson, both the function and its first derivative are required. The idea derives from the Taylor series expansion around a point x:

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + ..., \tag{19}$$

where δ defines the neighborhood around point x. For small δ, the second and higher order terms approach zero and so are unimportant. The expression reduces to

$$f(x+\delta) \approx f(x) + f'(x)\delta. \tag{20}$$

The next step is to solve for δ in $f(x+\delta) = 0$,

$$\delta = -\frac{f(x)}{f'(x)}. \tag{21}$$

This leaves only the function and the first derivative to be valued. In a multidimensional world, this is the function and the Jacobian, or matrix of first derivatives. However, in applications to robotics, the NR method

can have slow or poor convergence and flounder when the Jacobian matrix becomes singular. By adding a damping factor to the Jacobian, the poor behavior near and at singularities can be avoided. Therefore, methods like the LM method are often preferred, due to their faster convergence and robustness. Please see the Simple Example section for a more clear intuition as to why this is.

Robotics Literature

Ananthanarayanan and Ordonez [1] reduce their problem of robot arm manipulation to solving a system of three non-linear equations in two unknowns. They apply the Levenberg-Marquardt technique to find the roots of the equations that satisfy constraints, praising the algorithm's stability. In their example, the LM method uses steepest descent to get through areas of complexity in curvature until the method reaches a point where a quadratic solution is possible. At that point, the LM methods functions like a Gauss-Newton method. The inclusion of constraints is the reason they choose this over competing methods for their application.

A good review of the application of damped least squares to robotic control is Chiaverini, et al. [5]. They stress that, because the target for an end-effector may be at or near an edge of the workspace, the method to compute control commands must include the ability to operate at or near singularities. They stress that the damping factor should be chosen to facilitate this.

More recently, Buss and Kim [3] propose what they call a clamping mech-

anism to deal with the issue that when damping factors are put on added to the Jacobian too close to a singularity, there is a tendency for the solutions to oscillate across the singularity into another region of the workspace.

Di Vito, et al. [6] focus on evaluating several damped least squares algorithms for inverse kinematics in robots. They find that there is a trade-off between imposing a damping factor and the speed of the approach and resulting errors. Invoking the damping factor too close to the singularity can result in failure, but invoking the damping factor too far away can result in significant slowness in the robotic approach to the target and large errors. In fact, they find that none of the algorithms that they evaluate have satisfactory results.

Sugihara [21] proposed an improved damping factor in a Levenberg-Marquardt solution method, leading to a robust algorithm that improves numerical stability and convergence even when applied to previously unsolvable cases. His work demonstrates that target position-orientations could be done more easily to facilitate robot motion.

Chan and Lawrence [4] propose use of the LM method together with pseudoinverses to reduce computational needs, a real concern in 1988.

Limon, et al. [11] point out that the LM method is particularly useful at behavior near singularities and results in a stable way to compute the angles θ, which they verify using a series of simulations in the Advanced Robotic Motion Simulator.

Sudheer, et al. [20] find that the iterative Levenberg Marquardt method is

useful in computing and evaluating stable walking methods for bipeds. The limits on the degrees of freedom in the problem make the LM method most appropriate.

Simple Example

This section will consider a simple inverse kinematics problem and apply the LM method to compute a solution. The section Example C Code below provides C code for the given solution.

Consider a robot arm with exactly one link connected to a fixed frame by a single degree of freedom joint. Let the end-effector be the end of the link, and let the link have length l. Let the joint have an unbounded motion. Thus, the end-effector moves in a plane. Let this plane be described by an x and y axis with all the usual conventions centered on the axis of rotation for the joint.

The joint configuration is a scalar value describing the position of the single degree of freedom joint. Call this configuration β. Let $\beta = 0$ correspond to the arm lining up with the positive x axis. The configuration of the end-effector therefore is given by the vector valued function given in 22:

$$\mathbf{p}(\beta) = \begin{bmatrix} p_x(\beta) \\ p_y(\beta) \end{bmatrix} = \begin{bmatrix} l\cos\beta \\ l\sin\beta \end{bmatrix}. \tag{22}$$

.

Define the residual $\mathbf{e}(\beta)$ as follows:

$$\mathbf{e}(\beta) = {}^{d}\mathbf{p} - \mathbf{p}(\beta) = \begin{bmatrix} {}^{d}x - l\cos\beta \\ {}^{d}y - l\sin\beta \end{bmatrix}, \tag{23}$$

.

where ${}^{d}\mathbf{p} = [{}^{d}x, {}^{d}y]^T$ gives the desired end-effector position. The goal is to find the β that minimizes the objective function given in 24:

$$\phi(\beta) = \frac{1}{2}\mathbf{e^T W_E e} = \frac{({}^{d}x - l\cos\beta)^2 + ({}^{d}y - l\sin\beta)^2}{2}. \tag{24}$$

.

For simplicity, it is assumed that $\mathbf{W_E}$ is the identity matrix, which gives the second equality in 24. Also for simplicity, the orientation constraint has been removed.

In order to use the LM method, at the r^{th} iteration, values for the step $\delta_r \in \mathbb{R}^1$, damping factor $\lambda_r \in \mathbb{R}^1$, and next estimate $\beta_{r+1} \in \mathbb{R}^1$ must be computed. The value for δ_r is given by solving the following for δ_r:

$$(\mathbf{J^T J} + \lambda_r)\delta_r = \mathbf{J^T e}, \tag{25}$$

where $\mathbf{J}$ refers to the Jacobian for β_r. In this problem, the Jacobian is a vector in $\mathbb{R}^2$. The formula for the Jacobian is given in 26:

$$\mathbf{J}(\beta) = \begin{bmatrix} \frac{d}{d\beta}p_x(\beta) \\ \frac{d}{d\beta}p_y(\beta) \end{bmatrix} = \begin{bmatrix} -l\sin\beta \\ l\cos\beta \end{bmatrix}. \tag{26}$$

Putting 25 and 26 together gives a formula for the step δ_r:

$$\begin{aligned}
(\mathbf{J^T J} + \lambda)\delta_r &= \mathbf{J^T e} \\
(l^2(\sin^2\beta_r + \cos^2\beta_r) + \lambda_r)\delta_r &= -l\sin\beta_r({}^dx - l\cos\beta_r) + l\cos\beta_r({}^dy - l\sin\beta_r) \\
(l^2 + \lambda_r)\delta_r &= {}^dyl\cos\beta_r - {}^dxl\sin\beta_r \\
\delta_r &= \frac{{}^dyl\cos\beta_r - {}^dxl\sin\beta_r}{l^2 + \lambda_r}.
\end{aligned} \tag{27}$$

With a δ_r in hand, the new estimate β_r is computed as follows:

$$\beta_{r+1} = \beta_r + \delta_r. \tag{28}$$

.

All that is then left to determine is how to choose the damping factor λ_r. The algorithm given by Marquardt is discussed in the section titled Marquardt above. A summary is provided below in 29:

$$\lambda_r = \begin{cases} \frac{\lambda_{r-1}}{v} & \text{if } \phi(\frac{\lambda_{r-1}}{v}) \leq \phi_r \\ \lambda_{r-1} & \text{if } \phi(\frac{\lambda_{r-1}}{v}) > \phi_r \text{ and } \phi(\lambda_{r-1}) \leq \phi_r \\ \lambda_{r-1}v^w & \text{if } \phi(\frac{\lambda_{r-1}}{v}) > \phi_r \text{ and } \phi(\lambda_{r-1}) > \phi_r. \end{cases} \tag{29}$$

In equation 29, $\phi_r = \phi(\beta_r)$ gives the error for the last iteration, and $\phi(\frac{\lambda_{r-1}}{v})$ and $\phi(\lambda_{r-1})$ refer to the error if a δ and a β were computed with the

values inside the argument for ϕ are used for λ. In other words, in order to guarantee that $\phi_{r+1} < \phi_r$, different λs are tried. The scaling factor v is a constant such that $v > 1$.

In order to actually solve the problem, values must be provided for l, ${}^{d}x$, and ${}^{d}y$. For a sample run using the program given in the Example C Code section, let $l = 1$, ${}^{d}x = l\cos\theta$, and ${}^{d}y = l\sin\theta$ for some angle θ.

A good value for the scaling factor v was determined experimentally to be $v = 10$. Let the initial values be given as $\beta_0 = \psi$ for some angle ψ, and $\lambda_0 = 0.01$. The example program was run with these values and the target $\theta = \frac{\pi}{2} = 1.57079632679$ and the starting value $\psi = 0$. Additionally, the implemented code terminates when the error $\phi_r < \frac{1}{10000}$. For this program, such a termination condition works, as the desired position is guaranteed to be reachable. Generally speaking, the algorithm should terminate when the size of δ_r gets small to allow the algorithm to optimize when the desired end-effector configuration is unreachable. The results of the program are given below.

```
$ ./lm_example 1.57079632679 0

Initial Beta:   0.000000
Initial Phi:    1.000000
Initial Lambda: 0.010000

-------------------------
Iteration: 1
Beta:   0.999001
Phi:    0.159069
```

```
Lambda: 0.001000
Delta:  0.999001

-------------------------
Iteration: 2
Beta:   1.540090
Phi:    0.000471
Lambda: 0.000100
Delta:  0.541089

-------------------------
Iteration: 3
Beta:   1.570791
Phi:    0.000000
Lambda: 0.000010
Delta:  0.030702

*************************
Algorithm terminated! Final results:

Iterations: 3
Final Beta: 1.570791, Target: 1.570796
Final Phi:  0.000000

Final Lambda: 0.000010
Final Delta:  0.030702
```

Conclusions

The Levenberg-Marquardt method has a long history. In robotics, it has provided a solid platform for improvement in calculations involved in motion planning for robots. When computation time was an important consideration, the LM approach provided a robust and fast method to find a solution. Though computers have improved, the sheer number and complexity

of calculations for robot motion control means that robots still move slowly relative to human movement. In particular, methods such as the Levenberg-Marquardt damped-least squares approach show the direction research may move to significantly speed up calculations while preserving the algorithmic robustness needed around singularities and bounds to be practically operational for robots.

Example C Code

```
////////////////////////////////////////////////////////////////////////////////
// Main.cpp
//
// By Isaac Love
// Copyright 2020 (C) Isaac Love
// isaacblove@gmail.com
//
// Purpose: Uses the Levenburg-Marquardt algorithm to compute a simple
//  inverse kinematics problem.
//
// Usage: ./lm_example <target_angle> <start_angle>
//  <target_angle>: angle (in radians) to set the target coordinates to.
//  <start_angle>:  angle (in radians) to set the starting arm configuration.
////////////////////////////////////////////////////////////////////////////////

//////////////////////
// Library Includes //
//////////////////////

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/////////////////
// Definitions //
/////////////////

// Acceptable error to stop iterating.
#define PHI_CUTOFF 0.0001

//////////////////////////
// Function Prototypes //
//////////////////////////

// Name: ComputeFKX
//
```

```
// Purpose: Computes the forward kinematic function for the X-coordinate given
//  a beta value.
double
ComputeFKX(double l,
           double beta);

// Name: ComputeFKY
//
// Purpose: Computes the forward kinematic function for the Y-coordinate given
//  a beta value.
double
ComputeFKY(double l,
           double beta);

// Name: ComputeResidualX
//
// Purpose: Computes the X-coordinate residual.
double
ComputeResidualX(double l,
                 double x_goal,
                 double beta);

// Name: ComputeResidualY
//
// Purpose: Computes the Y-coordinate residual.
double
ComputeResidualY(double l,
                 double y_goal,
                 double beta);

// Name: ComputePhi
//
// Purpose: Computes the new phi value given a beta value.
double
ComputePhi(double l,
           double x_goal,
           double y_goal,
           double x_weight,
           double y_weight,
           double beta);
```

```
// Name: ComputeDelta
//
// Purpose: Computes the new delta value given the lambda and current beta.
double
ComputeDelta(double l,
             double x_goal,
             double y_goal,
             double lambda,
             double beta);

// Name: ComputeBeta
//
// Purpose: Computes the new beta value given the current beta and a delta.
double
ComputeBeta(double beta,
            double delta);

// Name: ComputeLambda
//
// Purpose: Computes the new lambda value given the prior phi value and current
//  beta.
double
ComputeLambda(double v,
              double phi_last,
              double lambda_last,
              double l,
              double x_goal,
              double y_goal,
              double x_weight,
              double y_weight,
              double beta);

//////////////////////////
// Function Definitions //
//////////////////////////

int
main(int argc, char** argv)
{
```

```
// Check Number of arguments.
if (argc != 3)
{
    printf("Usage: ./lm_example <target_angle> <start_angle>\n"
           "  <target_angle>: angle (in radians) to set the target "
            "coordinates to.\n"
           "  <start_angle>:  angle (in radians) to set the starting "
            "arm configuration.\n");
    return EXIT_SUCCESS;
}

double target_angle = atof(argv[1]);
double start_angle  = atof(argv[2]);

// Arm properties.
double l = 1.0;

// X and Y goals for the IK.
double x_goal = l * cos(target_angle);
double y_goal = l * sin(target_angle);

// Scaling factor.
double v = 10;

// Weighting
double x_weight = 1.0;
double y_weight = 1.0;

// Current Lambda value.
double lambda_cur = 0.01;

// Step size (and direction).
double delta_cur;

// Current Beta value (i.e. current position).
double beta_cur = start_angle;

// Current Phi value.
double phi_cur = ComputePhi(l,
                            x_goal,
```

```
                            y_goal,
                            x_weight,
                            y_weight,
                            beta_cur);

printf("Initial Beta:   %f\n"
       "Initial Phi:    %f\n"
       "Initial Lambda: %f\n",
       beta_cur,
       phi_cur,
       lambda_cur);

// Iterate.
unsigned int iteration = 0;

while (phi_cur > PHI_CUTOFF)
{
    iteration++;

    lambda_cur = ComputeLambda(v,
                               phi_cur,
                               lambda_cur,
                               l,
                               x_goal,
                               y_goal,
                               x_weight,
                               y_weight,
                               beta_cur);

    delta_cur = ComputeDelta(l,
                             x_goal,
                             y_goal,
                             lambda_cur,
                             beta_cur);

    beta_cur = ComputeBeta(beta_cur,
                           delta_cur);

    phi_cur = ComputePhi(l,
                         x_goal,
```

```
                            y_goal,
                            x_weight,
                            y_weight,
                            beta_cur);

        printf("\n\n-------------------------\n"
               "Iteration: %u\n"
               "Beta:   %f\n"
               "Phi:    %f\n"
               "Lambda: %f\n"
               "Delta:  %f\n",
               iteration,
               beta_cur,
               phi_cur,
               lambda_cur,
               delta_cur);

        if (lambda_cur == -1.0 || delta_cur == 0.0)
        {
            break;
        }
    }

    printf("\n\n*************************\n"
           "Algorithm terminated! Final results:\n\n"
           "Iterations: %u\n"
           "Final Beta: %f, Target: %f\n"
           "Final Phi:  %f\n\n"
           "Final Lambda: %f\n"
           "Final Delta:  %f\n",
           iteration,
           beta_cur,
           target_angle,
           phi_cur,
           lambda_cur,
           delta_cur);

    return EXIT_SUCCESS;
}
```

```
double
ComputeFKX(double l,
           double beta)
{
    return l * cos(beta);
}

double
ComputeFKY(double l,
           double beta)
{
    return l * sin(beta);
}

double
ComputeResidualX(double l,
                 double x_goal,
                 double beta)
{
    return x_goal - ComputeFKX(l, beta);
}

double
ComputeResidualY(double l,
                 double y_goal,
                 double beta)
{
    return y_goal - ComputeFKY(l, beta);
}

double
ComputePhi(double l,
           double x_goal,
           double y_goal,
           double x_weight,
           double y_weight,
           double beta)
{
    double x_res = ComputeResidualX(l, x_goal, beta);
    double y_res = ComputeResidualY(l, y_goal, beta);
```

```
    return 0.5 * (x_res*x_res * x_weight + y_res*y_res * y_weight);
}

double
ComputeDelta(double l,
             double x_goal,
             double y_goal,
             double lambda,
             double beta)
{
    double top = l * (y_goal*cos(beta) - x_goal*sin(beta));
    double bottom = l * l + lambda;

    return top / bottom;
}

double
ComputeBeta(double beta,
            double delta)
{
    return beta + delta;
}

double
ComputeLambda(double v,
              double phi_last,
              double lambda_last,
              double l,
              double x_goal,
              double y_goal,
              double x_weight,
              double y_weight,
              double beta)
{
    double beta_temp;
    double delta_temp;
    double lambda_temp;
    double phi_temp;

    // Try lambda = lambda_r-1 / v.
```

```
lambda_temp = lambda_last / v;

delta_temp = ComputeDelta(l, x_goal, y_goal, lambda_temp, beta);
beta_temp  = ComputeBeta(beta, delta_temp);
phi_temp   = ComputePhi(l, x_goal, y_goal, x_weight, y_weight, beta_temp);

if (phi_temp < phi_last)
{
    return lambda_temp;
}

// Try lambda = lambda_r-1.
lambda_temp = lambda_last;

delta_temp = ComputeDelta(l, x_goal, y_goal, lambda_temp, beta);
beta_temp  = ComputeBeta(beta, delta_temp);
phi_temp   = ComputePhi(l, x_goal, y_goal, x_weight, y_weight, beta_temp);

if (phi_temp < phi_last)
{
    return lambda_temp;
}

// Try lambda = lambda_r-1 * v^w.
while (1)
{
    lambda_temp = lambda_temp * v;

    delta_temp = ComputeDelta(l, x_goal, y_goal, lambda_temp, beta);
    beta_temp  = ComputeBeta(beta, delta_temp);
    phi_temp   =
            ComputePhi(l, x_goal, y_goal, x_weight, y_weight, beta_temp);

    if (phi_temp <= phi_last)
    {
        return lambda_temp;
    }
}

// Failed.
```

```
    return -1;
}

////////////////////////////////////////////////////////////////////////////////
// END OF FILE
////////////////////////////////////////////////////////////////////////////////
```

Bibliography

[1] Hariharan Ananthanarayanan and Raul Ordonez. Real-time inverse kinematics of (2n + 1) DOF hyper-redundant manipular arm via a combined numerical and analytical approach. *Mechanism and Machine Theory*, 91:209–226, 2015.

[2] Samuel R. Buss. Introduction to inverse kinematics with Jacobian transposes, pseudoinverse and damped least squares methods. 2009.

[3] Samuel R. Buss and Jin-Su Kim. Selectively damped least squares for inverse kinematics. *Journal of Graphics Tools*, 102(3):37–49, 2005.

[4] Stephen K. Chan and Peter D. Lawrence. General inverse kinematics with the error damped pseudoinverse. In *IEEE International Conference on Robotics and Automation, CH2555*, pages 834–839, 1988.

[5] Stefano Chiaverini, Cruno Siciliano, and Olav Egeland. Review of the damped least-squares inverse kinematics with experiments on an industrial robot manipulator. *IEEE Transactions on Control Systems Technology*, 2(2):123–134, 1994.

[6] Danielle DiVito, Ciro Natale, and Fianluca Antonelli. A comparison of damped least squares algorithms for inverse kinematics of robot manipulators. In *IFAC International Federation of Automatic Control Papers Online, 50-1*, pages 6869–6874, 2017.

[7] Baohua Huang and Changfeng Ma. A Shamanski-like self-adaptive Levenberg-Marquardt method for nonlinear equations. *Computers and Mathematics with Applications*, 77:357–373, 2019.

[8] Charles W. Wampler II. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares methods. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC–162(1):93–101, 1986.

[9] George G. Judge, W.E. Griffiths, R. Carter Hill, Helmut Lutkepohl, and Tsoung-Chao Lee. *The Theory and Practice of Econometrics, Second Edition.* John Wiley and Sons, 1985.

[10] Kenneth Levebnerg. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2:164–168, 1944.

[11] Rafael Disneros Limon, Jan Manuel Ivarra Zannatha, and Manuel Angel Armada Rodriguez. Inverse kinematics of a humanoid robot with non-spherical hip: A hybrid algorithm approach. *International Journal of Advanced Robotic Systems*, 10:213–227, 2013.

[12] Donald W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial And Applied Mathematics*, 11(2):431–441, 1963.

[13] Jorge J. More. The Levenberg-Marquardt algorithm: Implementation and theory. In *Conference on Numerical Analysis, University of Dundee, Scotland*, 1977.

[14] Yoshihiko Nakamura and Hideo Hanafusa. Inverse kinematic solutions with singularity robustness for robot manipulator control. *Journal of Dynamic Systems, Measurement, and Control*, 108:163–171, 1986.

[15] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization, Second Edition*. Springer, 2006.

[16] A. Nonymous. atan2. *Wikipedia*, 2020.

[17] A. Nonymous. Newton's method. *Wikipedia*, 2020.

[18] A. Nonymous. Rodrigues' rotation formula. *Wikipedia*, 2020.

[19] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vettering. *Numerical Recipes, The Art of Scientific Computing*. Cambridge University Press, 1986.

[20] A.P. Sudheer, R. Vijayakumar, and K.P Mohanda. Stable gait synthesis and analysis of a 12-degree of freedom biped robot in sagittal and frontal planes. *Journal of Automation, Mobile Robotics and Intelligent Systems*, 6(4):36–44, 2012.

[21] Tomomichi Sugihara. Solvability-unconcerned inverse kinematics by the Levenberg-Marquardt method. *IEEE Transactions on Robotics*, 27(5):984–991, 2011.

[22] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2006.

[23] WillowGarage. *PR2 Hardware Specs*. http://www.willowgarage.com/pages/pr2/specs, 2020.