Dynamic Optimization Assignment 1

Kazuya Otani

January 31, 2017

1 Using optimization to do inverse kinematics

The main goal of this assignment was to make an inverse kinematics solver for a robot arm with an arbitrary number of 3 DOF joints.

1.1 Forward Kinematics

The first step was to calculate the forward kinematics from the joint angles, assuming the base to a static reference frame. I used homogeneous coordinates/transformation matrices to calculate the position of each joint, starting from joint 0. The equations used are shown below, and implemented in fk.m

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} Ry = \begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} Rz = \begin{bmatrix} \cos\psi & -\sin\psi & 0 & 0 \\ \sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_i = R_x R_y R_z$$

$$x_{i+1} = \prod_{k=0}^{i} R_k \begin{bmatrix} l_i \\ 0 \\ 0 \\ 1 \end{bmatrix} + x_i$$

$\theta = roll$, $\phi = pitch$, $\psi = yaw$. x_i: position of joint i in base frame. R_i: Rotation at joint i.

1.2 Soft constraints / costs

The solver was guided by a cost function that determined how desirable its current state was, shown below. Each component of the cost function had a scaling constant, to account for differences in importance, units, and dimensionality. These calculations are implemented in criterion.m

$$cost = distance\_cost + orientation\_cost + jointlimit\_cost + obstacle\_cost$$

The position cost took the dot product of the end effector position error. c_pos is 10 in my code.

$$position\_cost = c_{pos}(x - x_d)^T (x - x_d)$$

The orientation cost penalizes solutions that put the orientation of the end effector in a different state than the target. I implemented this by converting the quaternion orientation target $q = q_0 + q_1 i + q_2 j + q_3 k$ to a rotation matrix, as follows.

$$R = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1 * q_2 - q_0 * q_3) & 2(q_0 * q_2 + q_1 * q_3) \\ 2(q_1 * q_2 + q_0 * q_3) & 1 - 2(q_1^2 + q_3^2) & 2(q_2 * q_3 - q_0 * q_1) \\ 2(q_1 * q_3 - q_0 * q_2) & 2(q_0 * q_1 + q_2 * q_3) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}$$

The difference between the desired orientation and actual orientation (from forward kinematics) is:

$$dR = R_{base,des}^T R_{base,current} = R_{des}^T R$$

dR is then converted into roll pitch yaw values with

$$d_{roll} = -\sin^{-1} R_{23}$$
$$d_{pitch} = \tan^{-1} R_{13}/R_{33}$$
$$d_{yaw} = \tan^{-1} R_{21}/R_{22}$$
$$d_{rot} = \begin{bmatrix} d_{roll} & d_{pitch} & d_{yaw} \end{bmatrix}^T$$

Finally, these values are used to compute the orientation cost. These conversions are implemented in quat2matrix.m and matrix2euler.m

$$orientation\_cost = c_{rot} d_{rot}^T d_{rot}$$

The joint limit code encodes the desire to stay away from joint limits, even if we are not hitting them. To achieve this, I formulated the joint limit cost as follows (p represents all roll/pitch/yaw angles).

$$mid = ub - lb$$
$$joint\_cost = (p - mid)^T (p - mid)$$

The obstacle constraint tries to keep the robot from getting too close to obstacles. I measured the closest distance between each link (represented as a line segment) and obstacle by projecting a joint-obstacle center vector on to the link to find the closest point. This is done in point-line-dist.m, and is outlined below. p_1 and p_2 are the end points of the link, p_3 is the center of an obstacle, and r is the obstacle's radius.

$$dp = p_2 - p_1, v = p_3 - p_1$$
$$c = \frac{dot(v, dp)}{norm(dp)}$$
$$p_{closest} = p_1 + \frac{dp}{norm(dp} * c, dist = norm(p_3 - p_{closest}), \text{ if } c \geq 0 \text{ and c<norm(dp)}$$
$$dist = min(norm(p_3 - p_1), norm(p_3 - p_2)), \text{ else}$$

1.3 Hard constraints

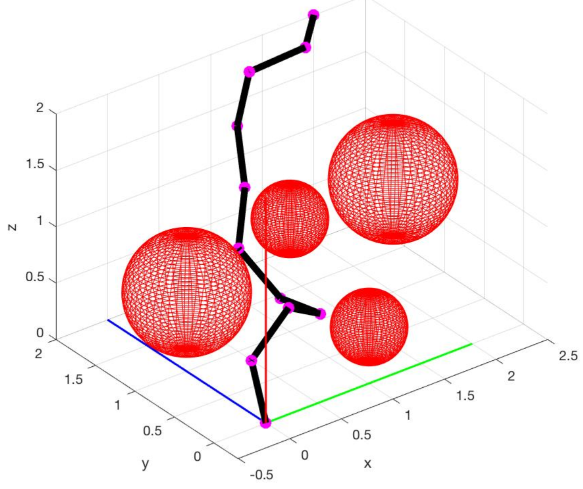
I also placed some hard constraints on the optimization algorithm to force it to find solutions that were not hitting obstacles or joint limits. I did this by formulating equality constraints so that interference (found by using the line segment to point calculations above) and joint limit overage were both constrained to zero. This is implemented in constraint.m

1.4 Results

After putting these parts together, I ran Matlab's fmincon on the cost function. The solver for this section was the active set method. For the initial guess $p0$, I either used all zeros or random joint angles between the lower and upper bounds.

```
options = optimset('Display','iter','MaxFunEvals',1000000,'Algorithm','active-set');
[answer,fval,exitflag]=fmincon(@criterion,p0, [], [], [], [], lb, ub, @constraint, options);
```

The figure below shows one solution from running the optimizer. The final cost function value was 0.0459, and the running time was 6.44 seconds.

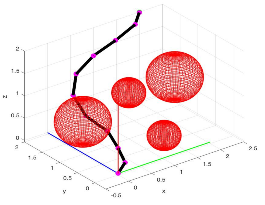
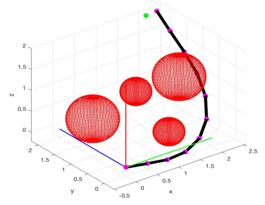


2 Incorporating derivatives

I first tried a few third-party automatic differentiation packages for Matlab: myAutomaticDifferentiation and AutoDiff. However, they did not work well with the matrices in my forward kinematics calculations. The approach I tried is in criterion_w_derivative.m

2.1 Finite differences

My next attempt at providing derivative information for the cost function was with finite differences. This is similar to what some of the algorithms do already, but my hypothesis was that providing a numerical derivative to the optimization algorithm, though computationally expensive and slow, would help the algorithms converge in less iterations. Another predicted effect was that fmincon would have to do less evaluations of the criterion function before reaching a solution (the F-count displayed by fmincon).
In practice, providing a derivative yielded mixed results. For some optimization algorithms, it led to better results, such as the one shown on the left below. This solution was found using the interior point algorithm, and the final cost function value was 0.033, and it was solved in 3.57 seconds. However, it also led to slow optimizations into local minima, such as the one on the right. This solution was found using the active set algorithm, the final cost function value was 6.29, and the solve time was 8.69 seconds. The sequential quadratic programming algorithm seemed to not use the derivative information even when the option was turned on; it produced identical results with and without.

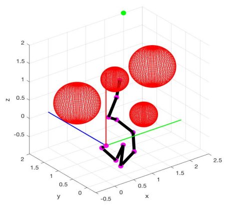
One thing the derivative information helped with was stopping earlier at "good enough" solutions that are unlikely to see any drastic improvements. The interior point algorithm, with all other settings being equal, converged to a cost function value of 0.034483 in 70 iterations and 2449 objective function evaluations when given derivative information. Without derivative information of the cost function, the same algorithm found a final cost function value of 0.024867 with 757 iterations and 25976.

3 Comparison of Algorithms

In this section, I tried three of the algorithms available in Matlab's fmincon (interior-point, sqp, active-set) and Covariance Matrix Adaptation Evolution Strategy, implemented by Nikolaus Hansen.

- Interior point

  This is the default optimization method in fmincon, and it had the worst performance. It was slow and tended to get stuck in local minima, as shown in the figure below.

- Sequential quadratic programming

  SQP worked fairly well, although it took a little longer than active set. The final cost function value was 0.0303, with elapsed time 9.47 seconds.

- Active set

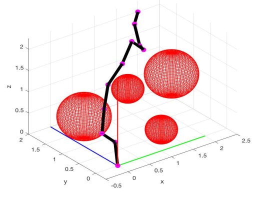
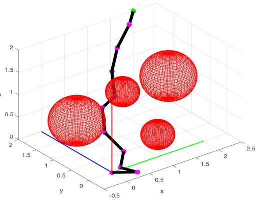
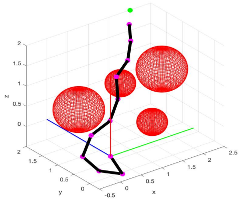
  This algorithm consistently found solutions with low cost (around 0.03) with randomized initial guesses, and ran in somewhere between 5 15 seconds.

- CMA-ES

  CMA worked very well for finding an optimal configuration. Although it took longer than the other algorithms, it consistently found a good solution. Because of loose termination conditions, the optimization algorithm tended to go on for a long time before it stopped. To alleviate this, I capped the iterations. I found that the solution converged to a good value after a few hundred iterations, but kept over iterations. The final cost function value for 500 iterations was 0.0328, and the solve time was 13.32 seconds. For 5000 iterations, it was 0.026704 and 118.702774 seconds.

4 Local minima

The optimization algorithm must search for optimal joint values to minimize the nonlinear cost function in a ($n_{joints} * 3$)-dimensional space, which is not straightforward. Because of this, it's highly unlikely that we will consistently find the global optimum. There are also many local minima that the algorithms get stuck on. Some of them are fairly close to the goal configuration, while others are far from desired. To mitigate the effects of local minima on the solution, we would like to be able to identify multiple local minima, and select the best one.
To achieve this, I maintained the initial guess as zero angle on all joints except the first one, where I rotated at uniform angles between the lower and upper bounds. I ran the same optimization algorithm (active set) on these different initial conditions, while recording their final cost function values and solutions. By choosing the solution with the best cost function value from these trials, I could make it more likely that I would find an acceptable solution. Below are three solutions that I got, with 0.0261, 0.0357, and 27.75 final cost function values, respectively. Note that the solutions take drastically different paths to the goal point.

I also tried running the same process using CMA-ES as the optimization algorithm. This time, I got solutions that were fairly similar in final cost function value, regardless of the initial conditions. This seems to show CMA-ES's superior abilities as a global optimizer. The algorithms in fmincon, while useful for global optimization in practice, rely on estimating gradient information and are more prone to falling into local minima (more sensitive to initial conditions). Below are three solutions that I obtained from CMA-ES (capped at 500 iterations), 0.0264, 0.0271, 0.0267, respectively.