ECSE 6440: Review of A Dynamic Programming Approach to Trajectory Planning of Robotic Manipulators

Due on Thursday, May 12, 2016

Prof. Wencen Wu 4:00 - 5:20 PM

Andrew Cunningham

October 7, 2016

Contents

1 Foreword

To satisfy the requirements for a course project for ECSE 6440 Optimal Control in the Spring of 2016, a review of "A Dynamic Programming Approach to Trajectory Planning of Robotic Manipulators" [3] was undertaken. Further, effort was put into implementing some of the results presented in the paper. While the results were promising in creating a parameterized path, the end result was unable to be verified due to computational limitations in the simulation framework used. This report will summarize my understanding of the paper's results and contain a review of implementation efforts.

2 Abstract

A Dynamic Programming Approach to Trajectory Planning of Robotic Manipulators focuses on developing a dynamic programming approach to cost-optimal trajectory planning of manipulators under joint torque constraints. The authors restrict the scope of the problem by assuming that a desired path for the manipulator to take has already been specified. The goal of the algorithm is to find a time-indexed cost-optimal series of joint positions, velocities and torques that direct the robot to follow the path. The authors go on to analyze the effectiveness of their algorithm and discuss convergence properties. One result of particular interest is that the parameterization of the path reduces the state space down to only two state variables regardless of the number of joints. This makes dynamic programming much more feasible because the curse of dimensionality is avoided.

3 Introduction

Robot manipulators are widely used in industry and optimal operation of these devices can reduce monetary costs for manufactured goods. However, finding optimal control strategies for robot manipulators is not easy because robot manipulators have coupled non-linear dynamics. To approach this tough problem, robot control is typically broken down into two steps: trajectory planning and trajectory following. Trajectory planning takes a spatial path descriptor as input and outputs a time indexed plan of robot's joint positions, velocities and torques that will direct the robot to follow the path [1]. Path tracking is the control loop that is used to follow the output that is generated by the path planner.

Previous attempts at solving this problem have utilized linear and non-linear programming to plan a trajectory [2]. In these approaches, the path was described as a set of endpoints and intermediary points. The main drawback of this approach is that there is no method to obtain the max accelerations and velocities. Thus, it is difficult to verify that the robot's constraints have not been violated by the plan.

The main goal of this paper, as outlined by the authors, was to address the limitations present in earlier techniques and to develop an alternative solution. The rest of this report will be split into three parts: Problem Statement, Technical Approach, and my Simulation Results.

4 Problem Statement

The main objective is to find a set of control signals that will drive a robot over a path with minimum cost. The authors assume that the path for each joint is pre-parameterized to avoid collisions:

$$q^i = f^i(\lambda) = a_4\lambda^3 + a_3\lambda^2 + a_2\lambda + a_1 \tag{1}$$

Where q^i is the ith joint. In addition to the parameterized path, it is assumed that the robot has joint torque constraints, dynamics defined and a cost function associated with the path. Using these pieces of information, an informal problem statement can be stated as: What control signals will direct a robot to follow a pre-specified curve in joint space with minimum cost while satisfying constraints on joint torques, robot dynamics and boundary conditions? Stated formally:

$$\begin{cases} find: \\ \min\limits_{u_i} \int_0^{\lambda_{max}} L(\lambda, \mu, u_i)d\lambda \\ subject\ to \\ u_i = J_{ij}\ddot{q}^j + C_{ijk}q^j\dot{q}^k + R_{ij}\dot{q}^j + G_i \quad \text{(Robot Dynamics)} \\ \forall_i:\ u_{i\ min} < u_i < u_{i\ max} \quad \text{(Realizable Torques)} \end{cases} \tag{2}$$

Symbol	Meaning
L	Cost Function
μ	$\frac{d}{dt}\lambda$
u_i	Torque for joint i
J_{ij}	Mass Inertia Matrix
R_{ij}	Viscous Friction Matrix
C_{ijk}	Centrifugal and Coriolis coefficients
G_i	Gravitational Loading vector
$E(q, \dot{q})$	Realizable Torques

Table 1: Meaning of symbols in Equation 2

5 Technical Approach

In order to apply dynamic programming to the problem statement developed in 2, it is necessary to:

1. Discretize the problem
2. Develop an incremental cost function
3. Apply dynamic programming

5.1 Discretizing the problem

Parameterizing the path reduces the dimensionality of the dynamic programming down to two state variables (λ, $\frac{d}{dt}\lambda = \dot{\lambda} = \mu$) regardless of the number of joints. Thus, discretization applies to these two state variables. This is a convenient representation because it is possible to reconstruct the path through time directly from the path through the phase plane that is constructed through dynamic programming. The phase plane is broken up into $n\lambda$ and $n\mu$ divisions to form a grid as in Figure 5.1.

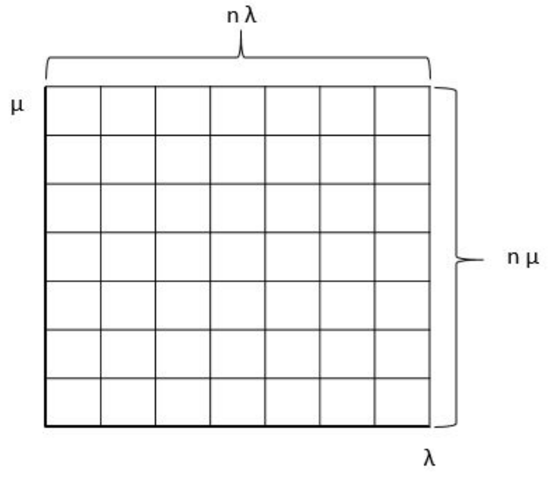

Figure 1: Discretized phase plane

5.2 Develop Incremental cost function

In order to apply dynamic programming to the grid developed in the section above, it is necessary to develop an incremental cost function that is solely a function of phase coordinates. The cost function formulated in the problem statement was a function of lambda (state variable), μ and u (joint torques). Thus, it is necessary to solve for the joint torques as a function of discrete points on the grid. The authors begin this derivation with the state dynamics and path parameterization. Given the system dynamics and the parameterization of a path:

$$u_i = J_{ij}\ddot{q}^j + C_{ijk}q^j\dot{q}^k + R_{ij}\dot{q}^j + G_i \quad \text{(Dynamics)} \tag{3}$$

$$q^i = f^i(\lambda) \quad \text{(Paramaterized Path)} \tag{4}$$

We can substitute equation (4) into equation (3) to find joint torques as a function of path parameters to get:

$$u_i = J_{ij}\frac{df^j}{d\lambda}\dot{\mu} + (J_{ij}\frac{d^2f^j}{d\lambda^2} + C_{ijk}\frac{df^j}{d\lambda}\frac{df^k}{d\lambda})\mu^2 + R_{ij}\frac{df^j}{d\lambda}\mu + G_i \tag{5}$$

For simplicity rewrite:

$$u_i = J_{ij}\frac{df^j}{d\lambda}\dot{\mu} + (J_{ij}\frac{d^2f^j}{d\lambda^2} + C_{ijk}\frac{df^j}{d\lambda}\frac{df^k}{d\lambda})\mu^2 + R_{ij}\frac{df^j}{d\lambda}\mu + G_i \tag{5}$$

into

$$u_i = M_i\dot{\mu} + Q_i\mu^2 + R_i\mu + G_i \tag{6}$$

Equations in 6 can then be made into a single equation by projecting u_i onto the velocity vector $\frac{df^i}{d\lambda}$

$$U = u_i\frac{df^i}{d\lambda} = M\dot{\mu} + Q\mu^2 + R\mu + G$$
$$where\ M = M_i\frac{df^i}{d\lambda}, Q = Q_i\frac{df^i}{d\lambda}, G = G_i\frac{df^i}{d\lambda} \tag{7}$$

If Equation 7 is then divided by μ we can get a dynamics expression that is not directly dependent on time:

$$M\frac{d\mu}{d\lambda} + Q\mu + R + \frac{1}{\mu}(G - U) = 0 \tag{8}$$

This is significant because the dynamic programming algorithm can use λ,μ as the state variables for dynamic programming. A solution that satisfies our previously developed dynamics

$$U = u_i\frac{df^i}{d\lambda} = M\dot{\mu} + Q\mu^2 + R\mu + G\frac{df^i}{d\lambda} \tag{7}$$

at boundary conditions $\mu(\lambda_k) = \mu_0,\ \mu(\lambda_{k+1}) = \mu_1$ can take the form of:

$$u_i = Q_i\mu^2 + R_i\mu + V_i \tag{9}$$

It is important to note that Equation 9 is a description of the torques required to get between two discrete points on a phase grid. Our newly developedu can be plugged into

$$M\frac{d\mu}{d\lambda} + Q\mu + R + \frac{1}{\mu}(G - U) = 0 \tag{8}$$

to generate:

$$\frac{d\mu}{d\lambda} = -\frac{1}{\mu}\frac{(S - V)}{M} \tag{10}$$

which can be solved to get

$$\lambda = K - \frac{M}{2(S - V)}\mu^2 \tag{11}$$

We solve for the constants of integration K and V to meet boundary conditions $(\mu(\lambda_k) = \mu_0,\ \mu(\lambda_k + 1) = \mu_1)$ to get

$$\lambda = \frac{\lambda_k(\mu_1^2 - \mu^2) + \lambda_{k+1}(\mu^2 - \mu_0^2)}{\mu_1^2 - \mu_0^2} \tag{12}$$

Finally, we can solve for μ and u_i

Fuction for μ in terms of λ

$$\mu = \sqrt{\frac{(\lambda_{k+1} - \lambda){\mu_0}^2 + (\lambda_k - \lambda){\mu_1}^2}{\lambda_{k+1} - \lambda_k}} \tag{13}$$

Fuction for u_i in terms of λ

$$u_i = Q_i\mu^2 + R_i\mu + S_i + M_i\frac{\mu_1^2 - \mu_0^2}{2(\lambda_{k+1} - \lambda_k)} \tag{14}$$

We have μ and u_i solved for in terms of boundary conditions and λ! So we can now find the incremental cost:

$$C_{inc} = \int_{\lambda_k}^{\lambda_{k+1}} L(\lambda, \mu, u_i)d\lambda \tag{15}$$

A segment of a path in phase space can have its cost calculated with Equation 15. Lastly, the joint torque constraints are checked by solving for each u_i over the trajectory and checking to see if they are within acceptable bounds.

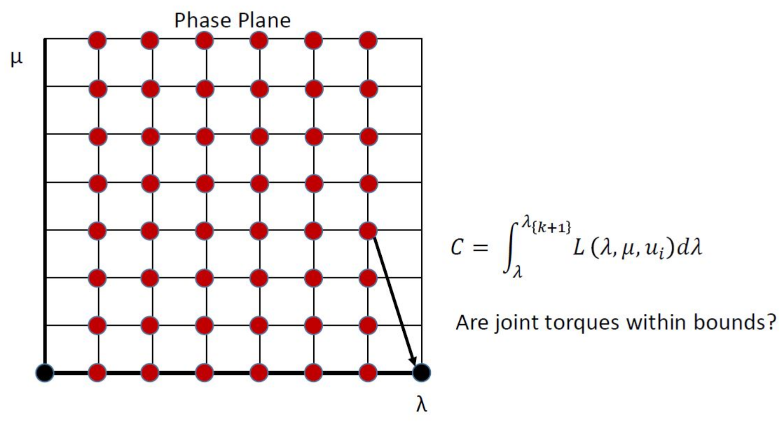
Phase Plane
μ
λ
$C = \int_{\lambda}^{\lambda\{k+1\}} L(\lambda, \mu, u_i)d\lambda$
Are joint torques within bounds?

Figure 2: Applying incremental cost to phase grid

5.3 Apply Dynamic Programming

The authors used the incremental cost function and the discretized grid to perform dynamic programming. The exact algorithm they used is outlined below:

SI: Determine the derivatives df'/dh of the parametric functions y(A), and from these quantities and the dynamic equations determine the coefficients of 6.

S2: Divide the (A, p) phase plane into a rectangular grid with NA divisions on the A-axis and N, divisions on the p-axis. Associate with each point (A,,,, p,) on the grid a cost C,,,, and a "next row" pointer P,,,,. Set all costs C,, to infinity, except for the cost of the desired final state, which should be set to zero. Set all the pointers P,, to null, i.e., make them point nowhere. Set the column counter a to NA.

S3: If the column counter a is zero, then stop.

S4: Otherwise, set the current-row counter fl to 0.

S5: If fl = N,, go to S12.

S6: Otherwise, set the next-row counter y to 0.

S7: If y = N,, go to S11.

S8: For rows fl and y, generate the curve that connects the (a - 1,fl) entry to the (a, y) entry. For this curve, test, as described in the previous paragraphs, to see if the required joint torques are in the range given by inequalities (3.7). If they are not, go to S10.

S9: Compute the cost of the curve by adding the cost C, to the incremental cost of joining point (a - 1,fl) to point (a, y). If this cost is less than the cost Ca-l,a, then set Ca-l,s to this cost, and set the pointer P,-l,a to point to that grid entry (a, y) that produced the minimum cost, i.e., set Pa-I, to y.
SIO: Increment the next-row counter y and go to S7.
SII: Increment the current-row counter fl and go to S5.
S12: Decrement the column counter a and go to S3. It is then possible to find the optimal trajectory from the grid by starting at the first column, reading off the next row pointer, and then moving to the row specified in the next column and repeating the process. Given the optimal trajectory, it is possible to directly[13] calculate the joint angles, velocities and torques. [14]

6 Simulation Results

6.1 Code Description

I implemented my interpretation of the paper's algorithm in Matlab. Additionally, I used Peter Corke's Matlab robotics toolbox as a simulation environment. The robotics toolbox contains methods to calculate components of a robot's dynamics given the robot's joint angles and velocities. I broke my implementation down into two parts: the main.m script and the getJointTorques.m script. The main.m script discretized a phase space grid and performed dynamic programming on it to get a trajectory in phase space. The getJointTorques.m function was setup to provide joint torques to the dynamics simulator as a function of joint position and velocity. The getJointTorques.m was responsible for taking the trajectory found in phase space by main.m into joint torques. This entailed finding the boundary conditions associated with a particular set of joint angles and the exact value of the path parameter, λ that generated the joint angles. A PUMA560 arm was chosen as the robotic manipulator to simulate throughout this simulation. Additionally, the parameterized path for each joint was chosen to be simple:

$$q^i = \lambda \tag{16}$$

where lambda ranged from 0 to 1. The net result is that each joint would aim to linearly move from a starting position of 0 to 1 radians. This parameterization is convenient because the function is invertible, thus for every lambda, there is a q associated with it. This made it easier to find where exactly on the grid the current joint angle was. If a non-invertible parameterization were used, it would be necessary to keep track of state to uniquely solve for q as a function of λ.

6.2 Code Results and Analysis

A simple discretization was run over $\lambda = 0, 1$ and $\mu = 0, 1$ to obtain a grid.

```
% divide mu,lambda phase space
% nMu is # of mu divisions
% nL is # of lambda divisions
nMu = 5; nL = 5;

% initialize cost matrices to infinity
% set last cost matrix entry to 0
costMatrix = ones(nMu,nL)*100000;
costMatrix(1,nL) = 0;

% initialize pointer matrices
pointerMatrix = ones(nMu,nL)*-1;
```

Additionally, the algorithm described by the paper was followed in that a cost and a next row pointer was associated with each element in the grid. The cost function was chosen to be a minimum time function defined as:

$$C = \int_0^\lambda \frac{1}{\mu} d\lambda \tag{17}$$

Dynamic programming was then run on this grid with simulated dynamics values to find the optimal trajectory. For full implementation details, see attached code in the appendix. The dynamic programming algorithm was able to find a reasonable trajectory for the minimum time criteria:

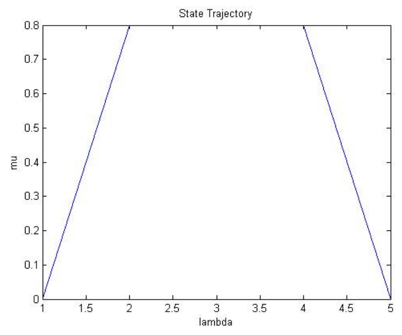

Figure 3: Simulation Results

Given this state trajectory, it is possible to map joint positions to joint torques to create a control policy to drive the robot arm along the pre-specified path. This was implemented in the form of findJointTorques.m (see appendix). Unfortunately, it was found that the use of symbolic math in such a function grinds Corke's dynamics simulator down to a halt. Using the tic and toc functions in Matlab, it was measured that a function call to findJointTorques.m took .5 seconds as opposed to the .002 seconds it took example functions. This resulted in prohibitively long simulation as the simulation would need to make many function calls to obtain joint torque.

7 Conclusion

In conclusion, this course project focused on examining the work of [3] and, to some extent, replicating its results. The simulation that I implemented was able to apply dynamic programming to a phase grid to find a minimum time state trajectory for a simple parameterized path. However, the generated path could not be fed back into the robot arm to generate a time-indexed torque history due to computational limitations.

References

[1] Tomas Lozano-Perez. Spatial planning: A configuration space approach. *Computers, IEEE Transactions on*, 100(2):108–120, 1983.

[2] Johnson YS Luh and Chyuan S Lin. Optimum path planning for mechanical manipulators. *Journal of Dynamic Systems, Measurement, and Control*, 103(2):142–151, 1981.

[3] Kang G Shin and Neil D McKay. A dynamic programming approach to trajectory planning of robotic manipulators. *Automatic Control, IEEE Transactions on*, 31(6):491–500, 1986.

A Appenix A: Code

Listing 1: main.m

```
mdl_puma560;
lambdaMax = 1;
muMax = 1;

% define parametric paths
syms l mu u li  lf
f1  = -2*l^3+3*l^2; f2 = -2*l^3+3*l^2;
f3  = -2*l^3+3*l^2; f4 = -2*l^3+3*l^2;
f5  = -2*l^3+3*l^2; f6 = -2*l^3+3*l^2;
f1(l)  = l;  f2(l)  = l;
f3(l)  = l;  f4(l)  = l;
f5(l)  = l;  f6(l)  = l;
 if1  = finverse(f1);  if2  = finverse(f2);
 if3  = finverse(f3);  if4  = finverse(f4);
 if5  = finverse(f5);  if6  = finverse(f6);

f(l)  = [f1;f2;f3;f4;f5;f6];
dfdl(l)  = diff(f,l);
 inversef(l)  = [if1 ; if2 ; if3 ; if4 ; if5 ; if6 ];

% define cost  function
L = 1/mu;
costFunction(li , lf)  = int(L,l, li , lf);

% divide mu,lambda phase space
% nMu is # of mu divisions
% nL is # of lambda divisions
nMu = 5; nL = 5;

%  initialize  cost  matrices to  infinity
% set last  cost  matrix entry to  0
costMatrix = ones(nMu,nL)*100000;
costMatrix(1,nL) = 0;

%  initialize  pointer  matrices
pointerMatrix = ones(nMu,nL)*-1;

% do dynamic programming!
for  columnCounter=nL:-1:2
    columnCounter
    % Find best way to get from columnCounter-1 to columnCounter
    for  muki=1:nMu %muki is index for mu_i
        for  mukp1i=1:nMu %mukp1i is index for mu_i+1
            muk = indexToMu(muki,nMu,muMax);
            mukp1 = indexToMu(mukp1i,nMu,muMax);
            lkp1 = indexToLambda(columnCounter,nL,lambdaMax);
            lk = indexToLambda(columnCounter−1,nL,lambdaMax);

            % Get robot dynamics parameters
            q = f(lk); q = double(q)';
            M = p560.inertia(q); G = p560.gravload(q);
            C = zeros(size(M));
            Mi=M∗dfdl; Qi = C∗dfdl;

            tempMu = sqrt(((lkp1−l)∗(muk−1)^2+(l+lk)∗(mukp1−1)^2)/(lkp1−lk));
            tempu = Qi∗mu^2+G'+Mi∗((mukp1−1)^2−(muk−1)^2)/(2∗(lkp1−lk));

            incrementalCost = int(tempMu,l,lk,lkp1);

            newCost = double(incrementalCost + costMatrix(mukp1i,columnCounter));

            if newCost < costMatrix(muki,columnCounter−1)
                costMatrix(muki,columnCounter−1) = newCost;
                pointerMatrix(muki,columnCounter−1) = mukp1i;
            end

        end
    end
end

indexTraj = getControlTraj(pointerMatrix)
muTraj = [0]
for i=1:size(indexTraj,2)−1
    muTraj = [muTraj indexToMu(indexTraj(i),nMu,muMax)];
end
```

Listing 2: findJointTorques.m

```
function [ u ] = findJointTorques(t, qd, q, qstar)
    nL = 5;
    muTraj = [0 .8 .8 .8 0];
    lambdaMax = 1;

    % Offset from 0 starting location
    q=q+.01

    mdl_puma560;

    % PARAMETERIZE PATHS
    syms l mu u li lf
    f1(l) = l; f2(l) = l;
    f3(l) = l; f4(l) = l;
    f5(l) = l; f6(l) = l;
    if1(l) = finverse(f1);
    f(l) = [f1;f2;f3;f4;f5;f6];
    dfdl(l) = diff(f,l);
    lCurrent = double(if1(q(1)));

    % GET BOUNDARY CONDITIONS
    mu0 = muTraj(floor(lCurrent/lambdaMax * nL)+1);
    mu1 = muTraj(ceil(lCurrent/lambdaMax * nL)+1);
    lk  = floor(lCurrent/lambdaMax * nL)*lambdaMax+1;
    lkp1  = ceil(lCurrent/lambdaMax * nL)*lambdaMax+1;
    mu(l) = sqrt(((lkp1-l)*(mu0-1)^2+(l+lk)*(mu1-1)^2)/(lkp1-lk));

    % CALCULATE ROBOT DYNAMICS
    M = p560.inertia(q); G = p560.gravload(q);
    C = zeros(size(M));
    Mi=M*dfdl; Qi = C*dfdl;

    % CALCULATE JOINT TORQUES
    u = Qi*mu(lCurrent)^2+G'+Mi*((mu1-1)^2-(mu0-1)^2)/(2*(lkp1-lk));
    u = double(u)'
end
```

Listing 3: getControlTraj.m

```
function [ u ] = getControlTraj( pointerMatrix )
    u = pointerMatrix(1,1);
    for i=2:size(pointerMatrix,2)
        u = [u pointerMatrix(u(end),i)];
    end
end
```