Robotics HW4

Yunfan Gao

October 2020

Problem 1

Given $R = R_y(\beta_1) R_x(\beta_2) R_y(\beta_3)$
Multiply both side with ey to get: $Rey = R_y(\beta_1)R_x(\beta_2)ey$
Multiply $R_y(-\beta_1)$ to both side: $R_y(-\beta_1)Rey = R_y(\beta_1)R_x(\beta_2)ey$
The above equation can be solved using subproblem 2, and record the outputs:
$[\alpha_1, \alpha_2] = sub2(Rey, ey, ey, ex)$, where $\alpha_1 = -\beta_1$ and $\alpha_2 = \beta_2$
Using the subproblem 2 will give you two results
The equation can also be modified as following:
$R_x(-\beta_2)R_y(-\beta_1)Re_x = R_y(\beta_3)e_x$
Use the two value results from the subproblem 2 to find the β_3
$R_x(-\alpha_2)R_y(\alpha_1)Re_x = R_y(\beta_3)e_x$
Use the subproblem 1: first β_3
$\beta_3(1) = subprob1(ex, rotation_x(-\alpha_2(1)) * rotation_y(\alpha_1(1)) * R * ex, ey)$
and second β_3:
$\beta_3(2) = subprob1(ex, rotation_x(-\alpha_2(2)) * rotation_y(\alpha_1(2)) * R * ex, ey)$

b)

Given $R = R_y(\beta_1) R_x(\beta_2) R_y(\beta_3)$
Denote y as k_1, x as k_2, and the second y as k_3, thus R becomes:
$R = \mathrm{R}(k_1, \beta_1)\,\mathrm{R}(k_2, \beta_2)\,\mathrm{R}(k_3, \beta_3)$, and take the derivative:
$\dot{R} = \left(\dot{\beta}_3 \mathrm{R}(k_1, \beta_1)\,\mathrm{R}(k_2, \beta_2)\,k_3 + \dot{\theta}_2 \mathrm{R}(k_1, \beta_1)\,k_2 + \dot{\beta}_1 k_1\right)^{\times} R$
which is equal to
$\dot{R} = \left(\dot{\beta}_1 k_1 + \dot{\beta}_2 \mathrm{R}(k_1, \beta_1)\,k_2 + \dot{\beta}_3 \mathrm{R}(k_1, \beta_1)\,\mathrm{R}(k_2, \theta_2)\,k_3\right)^{\times} R$

$$\omega = \dot{\beta}_1 k_1 + \dot{\beta}_2 \mathrm{R}(k_1, \beta_1)\,k_2 + \dot{\beta}_3 \mathrm{R}(k_1, \beta_1)\,\mathrm{R}(k_2, \theta_2)\,k_3 = J^{-1} \begin{bmatrix} \dot{\beta}_1 \\ \dot{\beta}_2 \\ \dot{\beta}_3 \end{bmatrix}$$

Thus $J^{-1} = [k_1, \quad \mathrm{R}(k_1, \beta_1)\,k_2, \quad \mathrm{R}(k_1, \beta_1)\,\mathrm{R}(k_2, \theta_2)\,k_3]$
since $k_1 = e_y$, $k_2 = e_x$, $k_3 = e_y$
$J^{-1} = R(e_y, \beta_1)[e_y, \quad e_x, \quad R(e_x, \beta_2)e_y]$

$$= \begin{bmatrix} 0 & \cos(\beta_1) & \sin(\beta_1)\cdot\sin(\beta_2) \\ 1 & 0 & \cos(\beta_2) \\ 0 & -\sin(\beta_1) & \cos(\beta_1)\cdot\sin(\beta_2) \end{bmatrix}$$

Take the inverse of J^{-1} to get the J:

$$\mathrm{J} = \begin{bmatrix} A & 1 & D \\ B & 0 & E \\ C & 0 & F \end{bmatrix}$$

where

$A = -(\cos(b2) * \sin(b1))/\left(\sin(b2) * \cos(b1)\text{^}2 + \sin(b2) * \sin(b1)\text{^}2\right)$
$B = \cos(\mathrm{b1})/\left(\cos(\mathrm{b1})\text{^}2 + \sin(\mathrm{b1})\text{^}2\right)$
$C = \sin(\mathrm{b1})/\left(\sin(\mathrm{b2})^* \cos(\mathrm{b1})\text{^}2 + \sin(\mathrm{b2})^* \sin(\mathrm{b1})\text{^}2\right)$
$D = -(\cos(b1) * \cos(b2))/\left(\sin(b2) * \cos(b1)\text{^}2 + \sin(b2) * \sin(b1)\text{^}2\right)$
$E = -\sin(b1)/\left(\cos(b1)\text{^}2 + \sin(b1)\text{^}2\right)$
$F = \cos(b1)/\left(\sin(b2) * \cos(b1)\text{^}2 + \sin(b2) * \sin(b1)\text{^}2\right)$

As you can see from J^{-1}, when β_2 is 0 or $n\pi$, it is not invertible, so the matrix is singular, and thus the J is not defined.

Problem 2

It is very important to assume that the ω is a constant, such that $\theta(t) - \theta_0 = wt$ Since $R(t) = e^{k^\times \theta(t)}$ and $\dot{R} = \omega^\times R$ where $\omega^\times = k\dot{\theta}$, $\dot{R} = \dot{\theta}k^\times R$. The initial rotation matrix and the rotation matrix at T gives the expressions: $R(T) = e^{k^\times \theta(T)} = R_f$, $R(0) = e^{k^\times \theta(0)} = R_0$. $\theta(T) = \omega T + \theta_0$, $\theta(0) = \theta_0$.
$e^{k^\times \theta(T)} = e^{k^\times(\omega T + \theta_0)} = e^{k^\times(\omega T)} e^{k^\times \theta_0} = e^{k^\times(\omega T)} R_0 = R_f$, thus we can get $R_f R_0^{-1} = e^{k^\times(\omega T)}$. Since $R_f R_0^{-1}$ is a rotation matrix, we can use matlab function R2kth to find the corresponding rotation vector and the ωT, and then divide it by T to get the ω

Problem 3

Listing 1: inverse kinematics using Levenberg–Marquardt(LM) algoritm

```
function normv = hw4_3() %output is the error
    r = -3 + (6)*rand(2,1);
    p0T = [r(1);r(2);0]; %target value
    %p0T = [-1.8663;1.1207;0]
    ez = [0;0;1];
    H = [ez, ez, ez, ez];
    n = 4; % n link
    qk = [0;0;0;pi/2]; %intial q0, making it at 3
    type = [0;0;0;0];
    a = 0.2; %learn rate
```

```
    e = 2; %regularization constant
    I = eye(4);
    p12_1=[1;0;0]; p23_2=[1;0;0]; p34_3=[1;0;0]; p4T_4=[1;0;0];
    p10 = [0;0;0];
    P = [p10, p12_1, p23_2, p34_3, p4T_4];
    dots = [];
    list = zeros(100:1);
    for i = 1:100
        J = jacob4(qk);%need updated qk as paramenters
        JT = transpose(J); %for fwdkin
        [R,p] = generalfwdkin(qk, type, H, P, n); %P is fixed
        normv = norm(p - p0T);
        qk = qk-a*(e*I+JT*J)^(-1)*JT*(p-p0T);
        dots(i,:) = p;
        list(i) = normv;
    end
    figure(1);
    scatter(dots(:,1),dots(:,2));
    figure(2);
    plot(list(1:end));
end
```

Listing 2: jacob function matlab code

```
function J =jacob4(qk) %need a updated qk everytime
    s = [0 -1 0; 1 0 0; 0 0 0];
    ez = [0;0;1];
    q1 = qk(1);q2=qk(2);q3=qk(3);q4=qk(4);
    R01 = rot(ez, q1); R02 = rot(ez, q1+q2); R03 = rot(ez, q1+q2+
        ↪ q3);
    R04 = rot(ez, q1+q2+q3+q4);

    p12_1=[1;0;0]; p23_2=[1;0;0];
    p34_3=[1;0;0]; p4T_4=[1;0;0];

    p1t_0 = R01*p12_1+R02*p23_2+R03*p34_3+R04*p4T_4;
    p2t_0 = R02*p23_2+R03*p34_3+R04*p4T_4;
    p3t_0 = R03*p34_3+R04*p4T_4;
    p4t_0 = R04*p4T_4;
    J = [s*p1t_0, s*p2t_0, s*p3t_0, s*p4t_0]; %output J to use
end
```

Notice: the generalfwdkin and rot function are from the RPI robotic toolbox. With a randomly generated targeted position, the function outputs the norm of the difference between the current position and the desired position,with the learning rate $\alpha = 0.2$ and the intial qk is [0; 0; 0; pi/2], so that the intial P0T

is located at (3, 1). Run the program:

```
p0T =

    2.1567
    1.8329
         0


ans =

   8.0161e-06
```

Figure 1: the desired position and output error

As shown above, $\|p - p_{0T}\| < 0.01$. Here are the plot of the movement of the arm and the plot of error between current position and desired position:

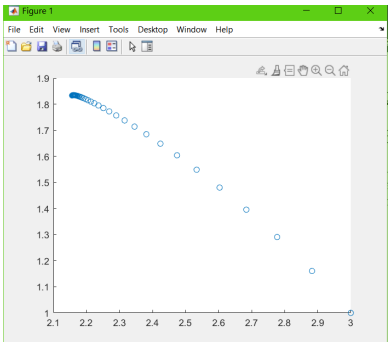

Figure 2: the arm moves to the desired position

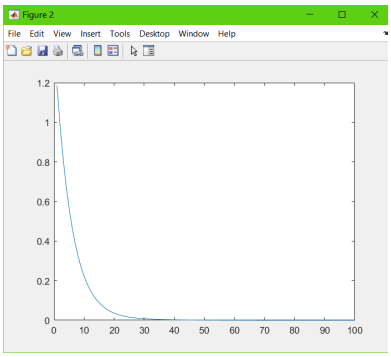

Figure 3: the difference between the end point and the target position approaching to zero

b)

In order to see what happen if P_{0T} is not feasible, the P_{0T} can set to an unreachable position such as (3,4,0), and thus the distance of the targeted position from the origin is 5, while the total length of all joints is 4. Set the initial qk to $[0; 0; 0; \frac{\pi}{2}]$. The movement of the arms with 100 iterations goes like this:

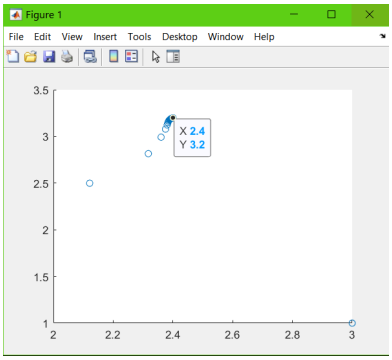

Figure 4: when P_{0T} is not feasible, with target value (3,4)

The final point after the 100 iterations reach to (2.4, 3.2), by Pythagorean theorem, the distance from the the target value is $(4-2.4)^2+(4-3.2)^2 = 1$, which is the closest position from the target value. This algorithm will find the nearest point P_{0T}, if it is not feasible, and the error will reduce to the smallest. For this example, the error reach to 1 as time increases:

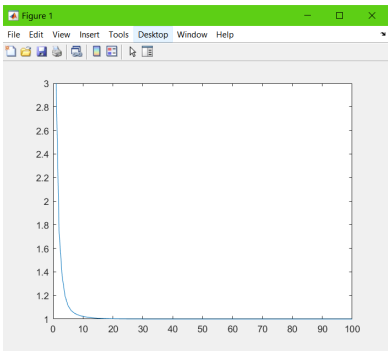

Figure 5: error approaching to 1, the smallest error achieved

c)

The α is the learning rate in such gradient descent, it is the updating value for every step. If the learning rate is too fast, each step is very large, the value is difficult to converge the desired value, because it might take a large step when only a small step is needed to reach to the targeted position. If the learning rate is too small, it takes a longer time to converge to the desired point. Choosing an appropriate learning rate is very important.
To compare a large learning rate with a small learning rate, the target position has been fixed at the (-1.8663 1.1207). When the learning rate is large ($\alpha = 2.8$), and the red arrow points to the targeted position P_{0T}, it has a problem of converging:

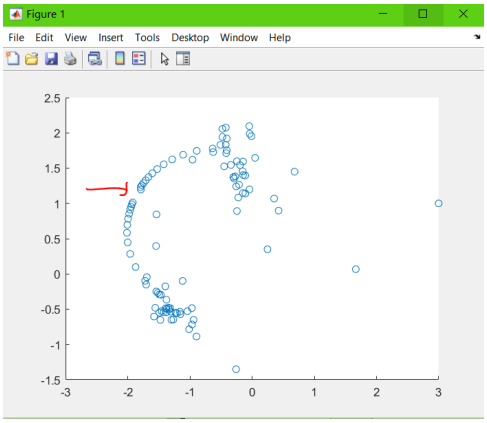

Figure 6: when learning rate is too large

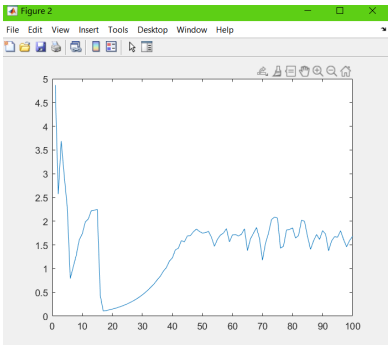

Figure 7: the error fails to converge with a large learning rate

A small learning rate (0.01) of 100 iterations with target position (-1.8663, 1.1207):

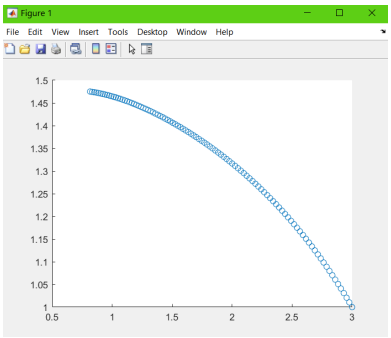

Figure 8: the arm moves very slow with small learning rate

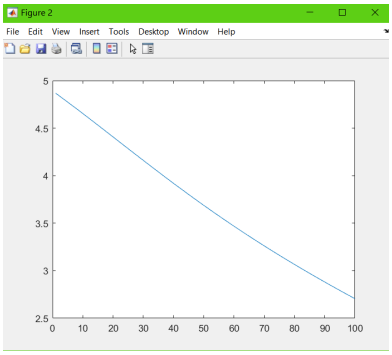

Figure 9: the error graph for the small learning rate

d)

The ϵ term is called the regularization term. With a large regularization, which will prevent the curve from overfitting, the curvature will not be too complex. For this example, the learning rate stays at a constant and we just change the regularization parameter. Same as previous question, the target position doesn't change. First, let the regularization term being $\epsilon = 8$:

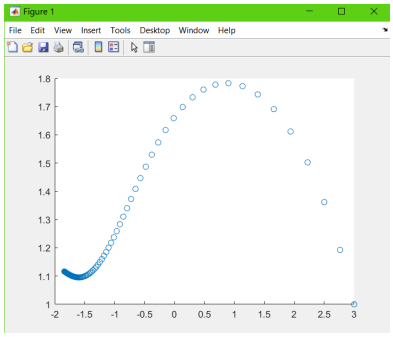

Figure 10: the path of the arm with regularization term being 8

The curve is very smooth and the movement for the robot arm seems to be very natural. However, although the curve is smooth, the route of this path with this large curve is a little redundant, as it goes to somewhere far from the target at beginning and then bends towards the targeted position. This path is not very efficient because some position seems to be unnecessary to be at.
With a relatively small regularization term $\epsilon = 0.5$: the curve become less

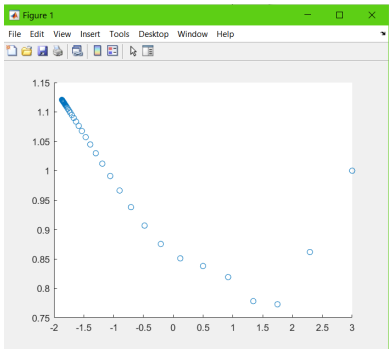

Figure 11: the path of the arm with regularization term being 0.5

smooth, as the curve model is more complicated than the previous one, but its doesn't travel to the point far from the targeted point.

If the path curve is being viewed as a polynomial with orders, then regularization is a way to regularize the polynomial to a lower order and thus the

curvature is not too complicated to predict. However, this simplification might cause the arm to move into somewhere unnecessarily far from the desired position. A regularization term with $\epsilon = 50$:

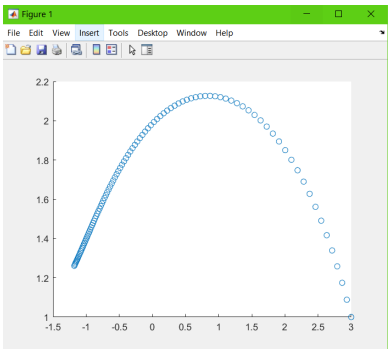

Figure 12: the path of the arm with regularization term being 50

In this case, if there are two robot arms working together, this large curvature might cause the arms to run into each other and thus it requires a large space, which will not be efficient.