Computer Vision 252B Hw2

Hao-en Sung (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Abstract

This is the report for CSE 252 Hw2.

1 Programming: Estimation of the camera projection matrix (45 points)

1.1 Linear estimation (15 points)

To find out the projection matrix P, where $x = P \cdot X$, I need to firstly compute the mean and variance for each coordinate of both 2D and 3D points, i.e. $\mu_x, \mu_y, \mu_z, \sigma_x^2, \sigma_y^2, \sigma_z^2$. After that, I can form 2D and 3D normalization matrix as T and U shown below, respectively.

$$T = \begin{bmatrix} s^{2D} & 0 & -\mu_x^{2D} \cdot s^{2D} \\ 0 & s^{2D} & -\mu_y^{2D} \cdot s^{2D} \\ 0 & 0 & 1 \end{bmatrix},\ U = \begin{bmatrix} s^{3D} & 0 & 0 & -\mu_x^{3D} \cdot s^{3D} \\ 0 & s^{3D} & 0 & -\mu_y^{3D} \cdot s^{3D} \\ 0 & 0 & s^{3D} & -\mu_z^{3D} \cdot s^{3D} \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where $s^{2D} = \sqrt{\frac{2}{\sigma_x^2 + \sigma_y^2}}$ and $s^{3D} = \sqrt{\frac{3}{\sigma_x^2 + \sigma_y^2 + \sigma_z^2}}$.

After applying T and U to normalize original 2D points (x) and 3D points (X), I then need to find out the left null space of projection matrix P, which can help us solve P. To do so, I need to first calculate the left null space for each x_i, which can be retrieved by solving *HOUSEHOLDER MATRIX*:

$$H_v = I - 2\frac{v \cdot v'}{v' \cdot v},$$

where $v = x + \text{sign}(x_i 1) \cdot \|(x_i)\| \cdot e_1$.

Once I calculate the null space for each x_i, I just need to use the *Singular Value Decomposition* to solve matrix $A = U \Sigma V'$ and retrieve the last row of V', where

$$A = \begin{bmatrix} [x_1]^{\perp} \otimes X_1^T \\ [x_2]^{\perp} \otimes X_2^T \\ \vdots \\ [x_n]^{\perp} \otimes X_n^T \end{bmatrix}.$$

At the end, I just need to reshape matrix P from shape $(12, 1)$ to $(3, 4)$ and normalized by $\|P\|$. The final result of projection matrix $\frac{P}{\|P\|}$ is recorded in Table 1.

To conclude, in my experiment, the norm of difference between x and $\hat{x} = PX$ is around 9.1697 in denormalized space and 0.0332 in normalized space. For more details, please refer to Code 1 and Code 2.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

-0.0060	0.0048	-0.0088	-0.8405
-0.0091	0.0023	0.0062	-0.5416
-0.0000	-0.0000	-0.0000	-0.0013

Table 1: P_{DLT}

1.2 Nonlinear estimation (30 points)

Similar to what I did in linear estimation algorithm, I firstly calculate the mean and variance for all 2D and 3D points and normalize them accordingly. It it noticeable that projection matrix obtained from *DLT* also needs to be transformed to normalized space, i.e. $P_n = T \cdot P_d \cdot U^{-1}$.

After that I need to calculate the inverse of covariance matrix Σ, where

$$\Sigma = \begin{bmatrix} \begin{bmatrix} (s^{2D})^2 & 0 \\ 0 & (s^{2D})^2 \end{bmatrix} & & \\ & \begin{bmatrix} (s^{2D})^2 & 0 \\ 0 & (s^{2D})^2 \end{bmatrix} & \\ & & \ddots \end{bmatrix}.$$

Later, I will run a for loop to iteratively improve my projection matrix P until there isn't any significant improvement in error measurement, i.e. $\epsilon'_{t+1}\Sigma\epsilon_{t+1} \sim \epsilon'_t\Sigma\epsilon_t$, where

$$\begin{aligned} \epsilon &= x - \hat{x} \\ &= x - P_n X. \end{aligned}$$

.

Inside the for loop, I need to parametrize the projection matrix $P_n \in \mathbb{R}^{12\times 1}$ to $P_p \in \mathbb{R}^{11\times 1}$, where

$$P_p = \frac{2}{\text{sinc}(\cos^{-1}(P_n[:1]))} \cdot P_n[1:].$$

To avoid the singularity, which will cause the numerical calculation issue, I need to deal with angle normalization once $\|P_p\| > \pi$ as follows.

$$P_p = \left(1 - \frac{2\pi}{\|P_p\|} \cdot \left\lceil \frac{\|P_p\| - \pi}{2\pi} \right\rceil\right) \cdot P_p$$

Next step, I need to calculate the Jacobian Matrix as

$$\begin{aligned} J &= \frac{\partial E}{\partial P_p} \\ &= \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_n \end{bmatrix}, \end{aligned}$$

where

$$\begin{aligned} A_i &= \frac{\partial x_i}{\partial P_p} = \frac{\partial x_i}{\partial P_n} \cdot \frac{\partial P_n}{\partial P_p} \\ &= \frac{1}{P_n^3 \cdot X_i} \cdot \begin{bmatrix} X'_i & 0' & -x_i^1 X'_i \\ 0' & X'_i & -x_i^2 X'_i \end{bmatrix} \cdot \begin{bmatrix} -0.5P_n[1:] \\ \frac{\text{sinc}(\frac{\|P_p\|}{2})}{2} \cdot I + \frac{1}{4\|P_p\|} \cdot \frac{d\text{sinc}(\frac{\|P_p\|}{2})}{d\frac{\|P_p\|}{2}} \cdot P_n \cdot P'_n \end{bmatrix}. \end{aligned}$$

At the end, I need to solve δ in the following equations:

$$(J'\Sigma^{-1}J + \lambda I) \cdot \delta = J'\Sigma^{-1} \cdot (x - \hat{x}),$$

where $\hat{x} = P_n \cdot X$.

If we find that the new derived δ cannot improve our model performance, we will try to reduce λ by scalar 10; otherwise, we will accept current δ, update both P_n and P_p, and iterate the update procedure once again. The final result of projection matrix $\frac{P}{\|P\|}$ is recorded in Table 2.

To conclude, after this non-linear approximation, the norm of difference between x and $\hat{x} = PX$ drops from 9.1697 to 9.0996 in denormalized space as well as from 0.0332 to 0.0329 in normalized space. The error measurement, i.e. $\epsilon'\Sigma\epsilon$ also drops as

$$84.0826 \to 82.8036 \to 82.8032 \to 82.8032$$

(last two estimations differ smaller than 0.0001). For moe details, please refer to Code 1 and Code 3.

0.0061	−0.0047	0.0088	0.8439
0.0090	−0.0023	−0.0061	0.5363
0.0000	0.0000	0.0000	0.0012

Table 2: P_{LM}

Appendix

Main Function

Code Listing 1: Main Function

```
% Read data
x = dlmread('../dat/hw2_points2D.txt');
X = dlmread('../dat/hw2_points3D.txt');
n = size(x,1);

% inhomogeneous -> homogeneous
x(:,3) = ones(n,1);
X(:,4) = ones(n,1);

% DLT Procedure
P = DLT(x, X, n);
disp(P ./ norm(P,'fro'));

% LEVENBERG-MARQUARDT
P = LM(x, X, n, P);
disp(P ./ norm(P,'fro'));
```

DLT Implementation

Code Listing 2: Direct Linear Transformation (DLT)

```
function P = DLT(x, X, n)

% Data Normalization
xm = mean(x);
xv = var(x);
xs = sqrt(2 / (xv(1)+xv(2)));
T = [xs, 0, -xm(1)*xs; 0, xs, -xm(2)*xs; 0, 0, 1];
x = x * T';

Xm = mean(X);
```

```
Xv = var(X);
Xs = sqrt(3 / (Xv(1)+Xv(2)+Xv(3)));
U = [Xs, 0, 0, -Xm(1)*Xs; 0, Xs, 0, -Xm(2)*Xs; ...
    0, 0, Xs, -Xm(3)*Xs; 0, 0, 0, 1];
X = X * U';

% Left Null Space of P
A = zeros(2*n, 12);
mmax = 0;
for i = 1:n
    v = [x(i,1) + sign(x(i,1)) * norm(x(i,:)), x(i,2), x(i,3)]';
    H_v = eye(3) - 2 * (v * v') / (v' * v);
    mmax = max(mmax, H_v(2,:) * x(i,:)');
    mmax = max(mmax, H_v(3,:) * x(i,:)');
    A(2*i-1,:) = [H_v(2,1)*X(i,:), H_v(2,2)*X(i,:), H_v(2,3)*X(i,:)];
    A(2*i  ,:) = [H_v(3,1)*X(i,:), H_v(3,2)*X(i,:), H_v(3,3)*X(i,:)];
end

% Solve for P
[~, ~, V] = svd(A, 'econ');
P = V(:,end);
P = reshape(P, 4, 3)';

% Data Denormalization
P = T \ P * U;
```

Levenberg Marquardt Implementation

Code Listing 3: Levenberg Marquardt

```
function [P, log] = LM(x, X, n, P)

% Data Normalization
xm = mean(x);
xv = var(x);
xs = sqrt(2 / (xv(1)+xv(2)));
T = [xs, 0, -xm(1)*xs; 0, xs, -xm(2)*xs; 0, 0, 1];
x = x * T';

Xm = mean(X);
Xv = var(X);
Xs = sqrt(3 / (Xv(1)+Xv(2)+Xv(3)));
U = [Xs, 0, 0, -Xm(1)*Xs; 0, Xs, 0, -Xm(2)*Xs; ...
    0, 0, Xs, -Xm(3)*Xs; 0, 0, 0, 1];
X = X * U';

P = T * P / U;

% Covariance Matrix
Z = diag(repmat(xs^2, 1, 2*n));

% Initialization
lambda = 0.001;
ex = calEpsilon(x, X, P);
perr = 10000000;
err = ex'*inv(Z)*ex;

% Error Log
log = err;

while abs(perr-err) > 0.0001
    vP = vector(P);
    v = parameterize(vP);
```

```
    % Angle Normalization
    if norm(v) > pi
        v = (1 - 2*pi/norm(v) * ceil((norm(v)-pi)/(2*pi))) * v;
    end
    vP = deparameterize(v);
    P = reshape(vP, 4, 3)';

    partial_vv = [-0.5 * vP(2:end)'; ...
                 my_sinc(norm(v)/2)/2 * eye(11) + ...
                 1/(4*norm(v)) * my_dsinc(norm(v)/2) * (v * v')];

    % Calculate J
    J = zeros(2*n,11);
    for i = 1:n
        w = X(i,:)*P(3,:)';
        partial_xp = 1/w * [X(i,:), zeros(1,4), -x(i,1)*X(i,:);
                           zeros(1,4), X(i,:), -x(i,2)*X(i,:)];
        J(2*i-1:2*i,:) = partial_xp * partial_vv;
    end

    while true
        % Solve delta
        d = (J'*inv(Z)*J + lambda*eye(11)) \ (J'*inv(Z)*ex);
        nv = v + d;

        % Angle Normalization
        if norm(nv) > pi
            nv = (1 - 2*pi/norm(nv) * ceil((norm(nv)-pi)/(2*pi))) * nv
                                                  ;
        end
        nvP = deparameterize(nv);
        nP = reshape(nvP, 4, 3)';

        nex = calEpsilon(x, X, nP);
        if nex'*inv(Z)*nex < ex'*inv(Z)*ex
            P = nP;
            ex = nex;
            lambda = 0.1 * lambda;
            break;
        else
            lambda = 10 * lambda;
        end
    end

    % Update error
    perr = err;
    err = nex'*inv(Z)*nex;
    log = [log, err];
end

% Data Denormalization
P = T \ P * U;
```

Helper Functions

Code Listing 4: Parameterize

```
function Pp = parameterize(Pn)
    a = Pn(1);
    b = Pn(2:end);
    Pp = 2 / my_sinc(acos(a)) * b;
end
```

Code Listing 5: Deparameterize

```
function Pn = deparameterize(Pp)
    Pn = [cos(norm(Pp)/2), my_sinc(norm(Pp)/2)/2 * Pp']';
end
```

Code Listing 6: Calculate *sinc*

```
function ret = my_sinc(x)
    if x == 0
        ret = 1;
    else
        ret = sin(x) / x;
    end
end
```

Code Listing 7: Calculate derivative of *sinc*

```
function ret = my_dsinc(x)
    if x == 0
        ret = 0;
    else
        ret = cos(x) / x - sin(x) / x^2;
    end
end
```

Code Listing 8: Calculate ϵ

```
function ex = calEpsilon(x, X, P)
    px = X * P';
    px = px ./ px(:,3);
    ex = vector(x(:,1:2)) - vector(px(:,1:2));
end
```

Code Listing 9: Vectorize Matrix to Vector

```
function vx = vector(x)
    vx = x';
    vx = vx(:);
end
```