2D Template Tracking with Iterative Least Squares

Patrick Mahoney

September 2016

1 Reading

Read sections 6.1 (2D and 3D feature-based alignment) and 6.2 (Pose Estimation) from Szeliski for this tutorial. Section 6.2 is a good explanation of the extrinsic pose recovery (3D Location / Orientations). Section 6.3 covers intrinsic camera calibration.

2 Problem

It is helpful to have a method to track the movement of an element from frame to frame. One approach for template tracking, to rely on an image of the element being tracked which is then searched for in the new frame. Let us define I_0 as the template image (eg. 100×100 pixels) and I_f as an image frame in a sequence (eg. 1280×720 pixels)

In order to determine the location of the template in the new frame we seek to evaluate how closely the template matches a specific location in the frame. For this we will use the sum of squared differences between the template and the image. This provides a measure for how good a match is.

3 Tracking Translation

To match the template to the image we need a transform $\boldsymbol{H}$ that allows us to map a point $\boldsymbol{w} = \left[\begin{smallmatrix} u \\ v \end{smallmatrix}\right]$ in the template to a point $\boldsymbol{w'}$ in the frame. This is done by some function $\boldsymbol{w'} = f(\boldsymbol{w}, \boldsymbol{H})$.

If we consider that the element is moving in the frame with translation only then we can simply define $\boldsymbol{H} = \boldsymbol{t} = \left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right] : (x, y) \in \mathbb{R}$ [1]. In this case, we can state the mapping $\boldsymbol{w} \to \boldsymbol{w'}$ as $\boldsymbol{w'} = \boldsymbol{w} + \boldsymbol{H} = \boldsymbol{w} + \boldsymbol{t}$.

As $\boldsymbol{t} \in \mathbb{R}$ we may need to interpolate in 2D between points when evaluating a mapping between the original template and a subsequent view. For the purpose of this guide we will be using bilinear interpolation. This gives us, $\boldsymbol{I}_n(\boldsymbol{w}) = \text{interp2}(\boldsymbol{I}_n[\boldsymbol{w}])$.

For implementation, we vectorise both $\boldsymbol{I_f}$ and $\boldsymbol{I_0}$ as:

$$\boldsymbol{\alpha}[n \cdot m, 1] = \text{vec}(\boldsymbol{I}) = \text{vec}\left(\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}\right) = \begin{bmatrix} a_{11} \\ a_{12} \\ \vdots \\ a_{1n} \\ \vdots \\ a_{mn} \end{bmatrix} \tag{1}$$

Therefore,

$$\boldsymbol{\alpha_f}[\boldsymbol{w}] = \text{vec}(\text{interp}(\boldsymbol{I_f}[\boldsymbol{w}])) \tag{2}$$
$$\boldsymbol{\alpha_0}[\boldsymbol{w}] = \text{vec}(\boldsymbol{I_0}[\boldsymbol{w}]) \tag{3}$$

From this we can evaluate the vector of differences as $\boldsymbol{r}[n] = \boldsymbol{\alpha_f}[\boldsymbol{w'}] - \boldsymbol{\alpha_0}[\boldsymbol{w}]$.

Secondly we also want to know the derivative of the residual wrt. each of the variables. Analytically this can be evaluated as seen in Equation 4.

[1] Note: We allow $\boldsymbol{t}$ to be a Real ($\mathbb{R}$) value and not an Integer ($\mathbb{Z}$). This will allow for 'sub-pixel' motion estimation as well as expansion to more general transforms.

$$\begin{aligned}\frac{\partial \boldsymbol{r}}{\partial t_x} &= \frac{\partial}{\partial t_x}(\boldsymbol{\alpha_f}(\boldsymbol{w'}) - \boldsymbol{\alpha_0}(\boldsymbol{w})) \\ &= \frac{\partial \boldsymbol{\alpha_f}}{\partial \boldsymbol{w_f}} \cdot \frac{\partial \boldsymbol{w_f}}{\partial t_x} \\ &= \nabla_x \boldsymbol{\alpha_f}(\boldsymbol{w'}) \cdot 1\end{aligned} \tag{4}$$

We then want to scale the residual based on the expected change due to each variable giving us Equation 5. As per Szeliski Section 6.1.3 we use an iterative algorithm to determine the translation.

$$q_x^k[n] = \frac{r_n}{\frac{\partial \boldsymbol{r}}{\partial t_x}}, q_y^k[n] = \frac{r_n}{\frac{\partial \boldsymbol{r}}{\partial t_y}} \tag{5}$$

We can now outline the update equation, shown in Equation 6.

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \\ \vdots & \vdots \\ 0 & 1 \\ 0 & 1 \\ \vdots & \vdots \end{bmatrix} \begin{bmatrix} t_x^{k+1} \\ t_y^{k+1} \end{bmatrix} = \begin{bmatrix} t_x^k - q_x[0] \\ t_x^k - q_x[1] \\ \vdots \\ t_y^k - q_y[0] \\ t_y^k - q_y[1] \\ \vdots \end{bmatrix} \tag{6}$$

As this fits the form of $\boldsymbol{Ax} = \boldsymbol{b}$ this can be solved with a linear least squares approach. In Matlab consider using 'mldivide' or '
' to solve for $\boldsymbol{x}$. By comparing the update difference $\boldsymbol{t}^{k+1} - \boldsymbol{t}^k$ it is possible to determine when updates have reached a stable solution as the difference tends towards zero.

4 Example solution

4.1 Use

The example code provided should allow you to test the performance of the provided tracked. It uses the provided dataset of linear domino movement however this can quickly be swapped out for existing data or a camera feed.

Running the Matlab script 'track_main.m' will allow you to test the tracker. It loads in the data set to a cell array and prompts the user to select the region to track from the first frame. The code then progresses to track the region through consecutive images.

Some things of note:

- The tracking function takes images in grayscale double format. This means that images that are being tracked on must be converted first.
- When using the provided dataset it is of note that tracking is lost after the domino has progressed across about half the image. Consider why this is and possible solutions to this behaviour.
- Currently the tracking location is signified by a point plotted at the top left corner of the tracked region. A better method of this is using the returned solution to draw a box or polygon around the tracked region.

5 Extensions

5.1 Illumination Compensation

As can be seen with the demo set, the current solution does not compensate for illumination variation, globally or locally. How does this affect the tracking ability and why we need brightness constancy here. Consider what changes are required required to solve for this, and implement these changes.

5.2 Tracking Transforms

The solution only only considers changes in translation for the element tracked. To robustly track an element it is necessary to solve for at least a similarity transform across the element. Consider the changes required for this correction. Szeliski Table 6.1 may help as a reference for this.

5.3 Performance Improvements

There are a number of changes that can be made to improve performance in both the solution and the supplied example code. What changes can be made to increase the efficiency of tracking and overhead required for performing the tracking operation.

Take a look at the variables defined each time the tracking function is run. Are there any variables that are constant or do not need to be defined new each round. Change the scope of this definition.

It is also useful to consider the scale at which the tracking is performed. Is the full resolution as used in the demo required. It may be useful to consider Matlabs 'impyramid' for this.

6 Code

6.1 Solver

```
function [result, it, res] = track_template(im, im_template, init, max_steps, epsilon)
%TRACK_TEMPLATE Locates a template in an image
%   Locates a template in an image based on iterative least
%   squares over the residual image intensities.
%
%   Input ::
%       im : NxM Double Matrix of image to track template in
%       im_template : PxQ Double Matrix of template to track in im
%       init : Initial transform to use for search
%       max_steps : Maximum number of iterations to use on optimisation
%           (Default 50)
%       epsilon : Minimum difference between iteration solutions
%           (Default 1e-1)
%
%   Output ::
%       result : Updated solution
%       it : Iterations required to reach solution
%       res : Residual at solution

%% Initialise default parameter values
if ~exist('max_steps', 'var')
    max_steps = 50;
end

if ~exist('epsilon', 'var')
    epsilon = 1e-1;
end

%% Calculate image gradient

[gx, gy] = imgradientxy(im, 'central');

%% Initialise

% Optimisation variables

t_kp = init;
t_k = zeros(size(init));

% Construct template index vectors
template_size = size(im_template);
[u, v] = meshgrid(0:template_size(2)-1, 0:template_size(1)-1);
u = u(:);
v = v(:);

% Vectorise template
I_t = im_template(:);

% A matrix
tplt_vec_size = size(I_t);
ones_array = ones(tplt_vec_size);
zeros_array = zeros(tplt_vec_size);

A_x = vertcat(ones_array, zeros_array);
A_y = vertcat(zeros_array, ones_array);

A = horzcat(A_x, A_y);


%% Iterate towards solution
for it = 1:max_steps

    % Calculate point mapping
    ud = u + t_kp(1);
    vd = v + t_kp(2);

    % Calculate residuals
    r = interp2(im, ud, vd) - I_t;
    r = vertcat(r, r);

    % Calculate residual pde
    dr_x = interp2(gx, ud, vd);
    dr_y = interp2(gy, ud, vd);

    dr = vertcat(dr_x, dr_y);

    % Initialise weightings
    % Weight to remove small gradients
    w = true(size(dr));
    w(abs(dr) < 0.01) = false;

    % Construct t vector for b
    t_x = t_kp(1) * ones_array;
    t_y = t_kp(2) * ones_array;

    t = vertcat(t_x, t_y);

    % Construct q (b) vector
    q = t - r./dr;
    %fprintf('%g\n', min(abs(dr)));

    % Weight A for solving
    Aw = bsxfun(@times, A, w);
    rw = q.*w;

    % Update solution
    t_k = Aw\rw;

    % Calculate solution difference
    t_diff = t_k - t_kp;

    % Check if solution settled
    % If settled escape
    if norm(t_diff) < epsilon
        break
    end

    % If not settled
    % Update solution for next iteration
    t_kp = t_k;

end

result = t_k;
res = norm(t_diff);

end
```
Script 1: track_template.m

6.2 Main

```
%% Load In Images
imagefiles = dir('data/*.png');
nfiles = length(imagefiles);    % Number of files found

images = cells(1, nfiles);
for i=1:nfiles
   currentfilename = fullfile('data', imagefiles(ii).name);
   currentimage = imread(currentfilename);
   images{ii} = im2double(currentimage);
end

%% Get and display 200 frames from vi devices
figure('Position', [100, 100, 1000, 700]);

% There are many ways to plot an image
% 'imshow' tends to be the easiest how ever it is slow
% the following constructs an image object that can
% have its data overwritten directly improving image display
% performance

% Setup plot
set(gca,'units','pixels');
% Aquire size of video image format
sz = size(images{1});

% Construct image display object
cim = image(...
    [1 sz(2)],...
    [1 sz(1)],...
    zeros(sz(1), sz(2), 1),...
    'CDataMapping', 'scaled'...
);

% Ensure axis is set to improve display
colormap gray;
axis image;

%% Setup tracking

im = imgaussfilt(images{1}, 4);

% Update data in image display objects
set(cim, 'cdata', images{1});

drawnow;

template_rect = round(getrect);

template = im(...
    template_rect(2):(template_rect(2) + template_rect(4)), ...
    template_rect(1):(template_rect(1) + template_rect(3)) ...
    );

%Initialise Result from first frame
[result, it, res] = track_template(im, template, [template_rect(1); template_rect(2)]);
fprintf('Initialised track location to X: %g, Y: %g, it: %i, r: %g\n', result(1),
    result(2), it, res);


%% Begin tracking
result_handle = 0;
for i = 2:nfiles

    % Get the frame and metadata.
    im = imgaussfilt(images{i}, 4);

    if result_handle ~= 0
        delete(result_handle);
    end

    % Update data in image display objects
    set(cim, 'cdata', images{i});

    [new_result, it, res] = track_template(im, template, result);

    % Output tracking results
    if isnan(res)
        fprintf('Lost tracking... initialising with previous result');
    else
        fprintf('Tracked template to X: %g, Y: %g, it: %i, r: %g\n', result(1),
    result(2), it, res);
    end

    % Update if result is ok
    if ~any(isnan(new_result))
        result=new_result;
    end

    % Draw result position on image
    hold on;
    result_handle = scatter(result(1), result(2));
    hold off;

    % Force a draw update
    drawnow;
end
```

Script 2: track_main.m