3D Computer Vision

Exercise 5: Dense Matching

ThreeDCV-TEAM
Prof Dr. Didier Stricker
Kaiserlautern University
http://ags.cs.uni-kl.de/

DFKI – Deutsches Forschungszentrum für Künstliche Intelligenz
http://av.dfki.de
Outline

- Organizational issues
- Theoretical questions
- Pixel and window based matching
  - Naive approach
  - Faster approach
  - SSD and NCC
- Questions
Theory Questions
1. What is the advantage of a rectified image pair regarding correspondence search?

- Epipolar geometry reduces correspondence search to a 1D-search problem
1. What is the advantage of a rectified image pair regarding correspondence search?

- Epipolar geometry reduces correspondence search to a 1D-search problem
- When the two cameras are parallel: the search gets simplified even further
1. What is the advantage of a rectified image pair regarding correspondence search?

- Epipolar geometry reduces correspondence search to a 1D-search problem
- When the two cameras are parallel: the search gets simplified even further:
  - All epipolar lines will be horizontal, thus parallel to one another as they converge at infinity.
  - Matching algorithm becomes easier to implement.
  - Parallel-implementations become very simple. B/c the computation at each pixel is identical.
2. What is the advantage of a rectified image pair regarding triangulation?

- Triangulation is becomes simpler
  - no need to solve homogeneous equations with SVD

\[ Z = \frac{f t_x}{d}, d = x' - x \]
3. Is image rectification also a good approach in case of multi-view dense reconstruction? Why?

- **No.** Rectification works well when the poses of the two views are already very close to a rectified stereo pair.
- Several images were taken around a statue.
  - Leads to distortions in the rectified image
  - The image might have very small overlap, which leads to poor correspondences
- Apply rectification when your cameras are already close to being parallel
Implementation
Datasets
Implementation: Datasets

Medieval Port Dataset

Baseline and intrinsic matrix are given.
Implementation: Datasets

KITTI Dataset

Baseline and intrinsic matrix are given.
Implementation: Datasets

Office Dataset (Optional)

Baseline and intrinsic matrix are given.
I1. Pixel-based dense matching

Left Camera

Right Camera
I1. Pixel-based dense matching

- Let’s search for the pixel in Red using pixel intensity metric
- We search for the corresponding pixel along the Red line
- We can use
  - Absolute pixel intensity difference (gray scale)
  - L1 distance in RGB space

Pixel intensity difference (gray scale)

Pixel location
11. Pixel-based dense matching

Pixel-Based Matching Result

Disparity

Left Image
12. Window-based dense matching

- Compare the green patch with blue patches along the red line
  - Use SSD, NCC etc
- Apply padding around the image
  - padding_size = patch_width//2
I2. Window-based dense matching

- Compare the green patch with blue patches along the red line
  - Use SSD, NCC etc
- Apply padding around the image
  - padding_size = patch_width//2

- disparity = \(x - x'\)
  \[\text{loc}(x,y)\]  \[\text{loc}(x',y)\]
I2. Window-based dense matching

- **Naive** stereo matching approach
  - We will use multiple for loops

- This will be very slow
  - later we will discuss speed up tricks
I2. Window-based dense matching: slow implementation

```python
left_patches, right_patches = extract_patches(left_img, right_img)
# shape of left_patches and right_patches -> [h, w, k*k]
```
left_patches, right_patches = extract_patches(left_img, right_img)
# shape of left_patches and right_patches -> [h, w, k*k]
for h in [0, img_height]:
    for w in [0, img_width]:
        for every pixel in the left image
I1. Window-based dense matching: slow implementation

```python
left_patches, right_patches = extract_patches(left_img, right_img)
# shape of left_patches and right_patches -> [h, w, k*k]
for h in [0, img_height]:
    for w in [0, img_width]:
        left_crop = left_patches[h, w, :]
```

*for every pixel in the left image*
I1. Window-based dense matching: slow implementation

```python
left_patches, right_patches = extract_patches(left_img, right_img)
# shape of left_patches and right_patches -> [h, w, k*k]
for h in [0, img_height]:
    for w in [0, img_width]:
        left_crop = left_patches[h, w, :]
        for x in [0, img_width]:
            right_crop = right_patches[h, x, :]
```

- For every pixel in the left image
- For each location along the epipolar line
I1. Window-based dense matching: slow implementation

```
left_patches, right_patches = extract_patches(left_img, right_img)
# shape of left_patches and right_patches -> [h, w, k*k]
for h in [0, img_height]:
    for w in [0, img_width]:
        left_crop = left_patches[h, w, :]
        right_crop = right_patches[h, x, :]
        matching_error = SSD(left_crop, right_crop)
```
left_patches, right_patches = extract_patches(left_img, right_img)
# shape of left_patches and right_patches -> [h, w, k*k]
for h in [0, img_height]:
    for w in [0, img_width]:
        left_crop = left_patches[h, w, :]
        for x in [0, img_width]:
            right_crop = right_patches[h, x, :]
            matching_error = SSD(left_crop, right_crop)
            matching_costs[h, w, x] = matching_error

for every pixel in the left image
for each location along the epipolar line
left_patches, right_patches = extract_patches(left_img, right_img)
# shape of left_patches and right_patches -> [h, w, k*k]
for h in range(0, img_height):
    for w in range(0, img_width):
        left_crop = left_patches[h, w, :]
        for x in range(0, img_width):
            right_crop = right_patches[h, x, :]
            matching_error = SSD(left_crop, right_crop)
            matching_costs[h, w, x] = matching_error
            matching_idxs = argmin(matching_costs, axis=2)  # along the disp axis

for every pixel in the left image
for each location along the epipolar line
I1. Window-based dense matching: slow implementation

```python
left_patches, right_patches = extract_patches(left_img, right_img)
# shape of left_patches and right_patches -> [h, w, k*k]
for h in [0, img_height]:
    for w in [0, img_width]:
        left_crop = left_patches[h, w, :]
    for x in [0, img_width]:
        right_crop = right_patches[h, x, :]
        matching_error = SSD(left_crop, right_crop)
        matching_costs[h, w, x] = matching_error

matching_idxs = argmin(matching_costs, axis=2)# along the disp axss
# Calculate disparity
disparity = zeros(img_height, img_width)
for h in [0, img_height]:
    for w in [0, img_width]:
        disparity[h, w] = abs(matching_idxs - w)
```
left_patches, right_patches = extract_patches(left_img, right_img)
# shape of left_patches and right_patches -> [h, w, k*k]
for h in [0, img_height]:
    for w in [0, img_width]:
        left_crop = left_patches[h, w, :]
        for x in [0, img_width]:
            right_crop = right_patches[h, x, :]
            matching_error = SSD(left_crop, right_crop)
            matching_costs[h, w, x] = matching_error
matching_idxs = argmin(matching_costs, axis=2)# along the disp axss
# Calculate disparity
disparity = zeros(img_height, img_width)
for h in [0, img_height]:
    for w in [0, img_width]:
        disparity[h, w] = abs(matching_idxs - w)
I1. Window-based dense matching: SSD and NCC

```python
def ssd(feature_1, feature_2):
    # Shape -> [k, k]
    patch_width = feature_1.shape[1]
    feature_1 = feature_1.reshape(patch_width**2)
    feature_2 = feature_2.reshape(patch_width**2)
    ssd_val = np.square(feature_1 - feature_2).sum()
    return ssd_val
```
I1. Window-based dense matching: SSD and NCC

```python
def ssd(feature_1, feature_2):
    # Shape -> [k, k]
    patch_width = feature_1.shape[1]
    feature_1 = feature_1.reshape(patch_width**2)
    feature_2 = feature_2.reshape(patch_width**2)
    ssd_val = np.square(feature_1 - feature_2).sum()
    return ssd_val

def ncc(feature_1, feature_2):
    patch_width = feature_1.shape[1]
    feature_1 = feature_1.reshape(patch_width**2)
    feature_2 = feature_2.reshape(patch_width**2)
    norm_1 = np.linalg.norm(feature_1) + 10e-06
    norm_2 = np.linalg.norm(feature_2) + 10e-06
    ncc_val = np.dot(feature_1/norm_1, feature_2/norm_2)
    return ncc_val
```
Faster Implementation
I1. Faster implementation

- We know that pixels move only one direction*
- We can restrict the search to pixels in the left side of the right image

* Might not hold for some special cameras
11. Faster implementation

- We know that pixels move only one direction
- We can restrict the search to pixels in the left side of the right image
11. Faster implementation

- Points will move only by a limited amount
- We can set maximum disparity
  - by making an assumption about the depth of closest point to the camera

\[
\text{max}_\text{disp} = \frac{\text{baseline} \times \text{focal len}}{\text{min}_\text{depth}}
\]

* Might not hold for some special cameras
I1. Faster implementation

- Points will move only by a limited amount
  - We can set minimum and maximum disparity
- We can set maximum disparity
  - by making an assumption about the depth of closest point to the camera
    \[ \text{max\_disp} = \frac{\text{baseline} \times \text{focal\_len}}{\text{min\_depth}} \]
  - !!! If there is a point with depth<min\_depth, it will not be reconstructed
11. Faster implementation

```python
l_patches, r_patches = extract_patches(left_img, right_img)
# shape of left_patches and right_patches -> [h, w, k*k]
num_disps = disp_max - disp_min
sim_scores = 1000*np.ones([height, width, num_disps])
```
11. Faster implementation

```python
l_patches, r_patches = extract_patches(left_img, right_img)
# shape of left_patches and right_patches -> [h, w, k*k]
disp_min = 0
num_disps = disp_max - disp_min
sim_scores = 1000*np.ones([height, width, num_disps])

for i in range(disp_min, disp_max):
    shifted_r_patches = shift(r_patches, i)
    ssd_score = SSD(left_patches, shifted_r_patches)
    sim_scores[:, :, i-disp_min] = ssd_scores

disparity = np.argmin(sim_scores, axis=2)
depth = f_length*baseline/disparity
```
11. Faster implementation

```python
def shift(feature, d):
    assert feature.ndim==3, 'inp shape should be [h, w, f]'
    s_feats = 100*np.ones(feature.shape)
    if d==0:
        s_feats = feature
    else:
        s_feats[:, d:, :] = feature[:, :-d, :]
    return s_feats
```

Input image

Padding: d pixels

Shifted image
1. Faster implementation

- Sum of Square Differences

```python
def ssp(feature_1, feature_2):
    ...
    inputs: feature_1 and feature_2 with shape[w, h, f]
    f = patch_width*patch_width
    return: per pixel distance, shape[w, h]
    ...
    sq_diff = np.square(feature_1 - feature_2)
    ssp_val = np.sum(sq_diff, axis=2)
    return ssp_val
```
11. Faster implementation

- Normalized Cross Correlation

```python
def ncc(feature_1, feature_2):
    ...
    inputs: feature_1 and feature_2 -> shape[h, w, k*k]
k= is the patch width/or height/
return: per pixel patch similarity -> , shape[h, w]
    ...

h,w,f = feature_1.shape
feat_norm_1 = normalise(feature_1)
feat_norm_2 = normalise(feature_2)
ncc_vals = (feat_norm_1*feat_norm_2).sum(axis=2)
return ncc_vals
```
11. Faster implementation

- Normalized Cross Correlation

```python
def ncc(feature_1, feature_2):
    ...
    inputs: feature_1 and feature_2 -> shape[h, w, k*k]
    k= is the patch width/or height/
    return: per pixel patch similarity -> , shape[h, w]
    ...

    h,w,f = feature_1.shape
    feat_norm_1 = normalise(feature_1)
    feat_norm_2 = normalise(feature_2)
    ncc_vals = (feat_norm_1*feat_norm_2).sum(axis=2)
    return ncc_vals
```

```python
def normalise(feature):
    eps = 1e-06
    h,w,d = feature.shape
    l2_norm = np.linalg.norm(feature, axis=2, keepdims=True)
    return feature/(l2_norm + eps)
```
Results
I2. Window-based dense matching: SSD Results

Window-Based Matching Result  \textbf{patch size = 1}

Disparity  \hspace{1cm}  Left Image
I2. Window-based dense matching: SSD Results

Window-Based Matching Result

patch size = 3

Disparity

Left Image
II2. Window-based dense matching: SSD Results

Window-Based Matching Result

patch size = 5

Disparity

Left Image
I2. Window-based dense matching: SSD Results

Patch size = 7

Window-Based Matching Result

Disparity

Left Image
12. Window-based dense matching: SSD Results

Window-Based Matching Result  
patch size = 9

Disparity  
Left Image
I2. Window-based dense matching: SSD Results

Window-Based Matching Result

patch size = 11

Disparity

Left Image
I2. Window-based dense matching: SSD Results

Window-Based Matching Result  \textbf{patch size} = 13

Disparity  Left Image
I2. Window-based dense matching: SSD Results
Effect of patch size on computation time and quality
Effect of patch size on computation time and quality

- Small patch sizes require small computation time
- Small patch sizes lead to noisy reconstruction

- However, larger window size doesn’t always give higher quality
  - very large window sizes tend over smooth outputs

![SSD](image1)

![NCC](image2)
Outlier Filtering
Outlier Filtering

- When a pixel has too many good matches in the other image it is probably an outlier
- In the image below we have colored outliers as red pixels
Outlier Filtering

Larger patch sizes lead to lower outlier percentage

SSD, threshold=3 patch_width=3

Outliers percentage = 5.2
Outlier Filtering

Larger patch sizes lead to lower outlier percentage

SSD, threshold=3 patch_width=5

Outliers percentage = 4.69
Outlier Filtering and Patch Size

Larger patch sizes lead to lower outlier percentage

SSD, threshold=3 patch_width=9

Outliers percentage = 3.63
12. Window-based dense matching: NCC Results

Window-Based Matching Result  \( \text{patch size} = 1 \)

Disparity  \hspace{1cm} \text{Left Image}
12. Window-based dense matching: NCC Results

Window-Based Matching Result  \text{patch size} = 3

Disparity  Left Image
12. Window-based dense matching: NCC Results

Window-Based Matching Result

patch size = 5
12. Window-based dense matching: NCC Results

Window-Based Matching Result  \textbf{patch size} = 7
I2. Window-based dense matching: NCC Results

Window-Based Matching Result  \textit{patch size} = 9

Disparity  \hspace{1cm}  Left Image
I2. Window-based dense matching: NCC Results

Window-Based Matching Result  \textbf{patch size} = 11

\begin{itemize}
\item Disparity
\item Left Image
\end{itemize}
12. Window-based dense matching: NCC Results

Window-Based Matching Result  

**patch size = 13**

Disparity  

Left Image
12. Window-based dense matching: NCC Results

Window-Based Matching Result  \textbf{patch size} = 15

Disparity  \hspace{1cm}  Left Image
NCC KITTI

Patch width = 1, NCC
NCC KITTI

Patch width = 3, NCC
Patch width = 5, NCC
NCC KITTI

Patch width = 7, NCC
NCC KITTI

Patch width = 9, NCC
NCC KITTI

Patch width = 11, NCC
Thank you!