Homography

Lecture 05

Computer Vision for Geosciences

2021-03-26

Universidad Nacional Autónoma de México

Image transformations:

$$g(x,y) = T[f(x,y)]$$

where:

- f(x, y) is an input image
- g(x, y) is the output image
- T is an operator

Previous lecture(s):

- **point operators**
  ⇒ point operators act on individual pixels, ignoring surrounding pixels (neighborhood of T=1x1 pixel)
  ⇒ intensity transformation functions (EX: change image contrast with $g(x,y) = f(x,y)^2$)
- **local operators**
  ⇒ local operators transform pixel value f(x,y) based on surrounding pixels (neighborhood of T>1x1 pixel)
  ⇒ linear operators (filtering with convolutions), morphological operators (filtering with morphology)

Today's lecture:

- **geometrical operators**
  ⇒ geometrical operators **do not** change pixel value, instead "**move**" it to a new position

Image transformations:

$$g(x, y) = T[f(x, y)]$$

where:

- f(x, y) is an input image
- g(x, y) is the output image
- T is an operator

Previous lecture(s):

- **point operators**
  ⇒ point operators act on individual pixels, ignoring surrounding pixels (neighborhood of T=1x1 pixel)
  ⇒ intensity transformation functions (EX: change image contrast with $g(x, y) = f(x, y)^2$)
- **local operators**
  ⇒ local operators transform pixel value f(x,y) based on surrounding pixels (neighborhood of T>1x1 pixel)
  ⇒ linear operators (filtering with convolutions), morphological operators (filtering with morphology)

Today's lecture:

- **geometrical operators**
  ⇒ geometrical operators **do not** change pixel value, instead "**move**" it to a new position

Image transformations:

$$g(x, y) = T[f(x, y)]$$

where:

- f(x, y) is an input image
- g(x, y) is the output image
- T is an operator

Previous lecture(s):

- **point operators**

  ⇒ point operators act on individual pixels, ignoring surrounding pixels (neighborhood of T=1x1 pixel)

  ⇒ intensity transformation functions (EX: change image contrast with $g(x, y) = f(x, y)^2$)

- **local operators**

  ⇒ local operators transform pixel value f(x,y) based on surrounding pixels (neighborhood of T>1x1 pixel)

  ⇒ linear operators (filtering with convolutions), morphological operators (filtering with morphology)

Today's lecture:

- **geometrical operators**

  ⇒ geometrical operators **do not** change pixel value, instead "**move**" it to a new position

Image transformations:

$$g(x, y) = T[f(x, y)]$$

where:

- f(x, y) is an input image
- g(x, y) is the output image
- T is an operator

Previous lecture(s):

- **point operators**
  ⇒ point operators act on individual pixels, ignoring surrounding pixels
  (neighborhood of T=1x1 pixel)
  ⇒ intensity transformation functions (EX: change image contrast with $g(x, y) = f(x, y)^2$)
- **local operators**
  ⇒ local operators transform pixel value f(x,y) based on surrounding pixels
  (neighborhood of T>1x1 pixel)
  ⇒ linear operators (filtering with convolutions), morphological operators (filtering with morphology)

Today's lecture:

- **geometrical operators**
  ⇒ geometrical operators **do not** change pixel value, instead "**move**" it to a new position

Homography is used to transform an image from one projective plane to another

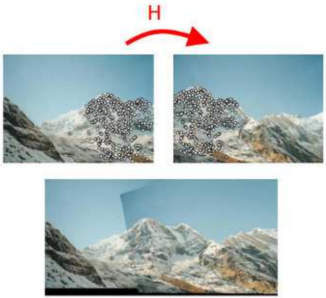
Applications in image processing:

- image stitching (e.g., mosaic and panoramas)
- image registration (e.g., "fuse" datasets in unique coordinate frame)
- image warping (e.g., change image perspective, correct lense distortion, etc.)
- Structure from Motion (SfM) (i.e., 3D reconstruction from multiple images)
- and much more! (e.g., augmented reality, etc.)

Homography is used to transform an image from one projective plane to another

Applications in image processing:

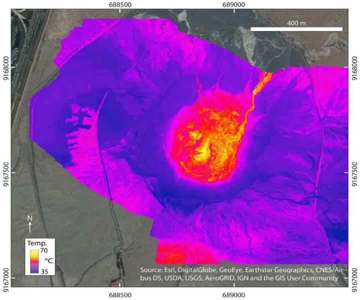
- image stitching (e.g., mosaic and panoramas)
- image registration (e.g., "fuse" datasets in unique coordinate frame)
- image warping (e.g., change image perspective, correct lense distortion, etc.)
- Structure from Motion (SfM) (i.e., 3D reconstruction from multiple images)
- and much more! (e.g., augmented reality, etc.)



Homography is used to transform an image from one projective plane to another

Applications in image processing:

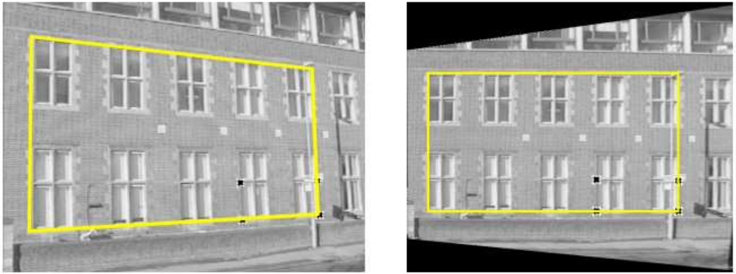
- image stitching (e.g., mosaic and panoramas)
- image registration (e.g., "fuse" datasets in unique coordinate frame)
- image warping (e.g., change image perspective, correct lense distortion, etc.)
- Structure from Motion (SfM) (i.e., 3D reconstruction from multiple images)
- and much more! (e.g., augmented reality, etc.)



Homography is used to transform an image from one projective plane to another

Applications in image processing:

- image stitching (e.g., mosaic and panoramas)
- image registration (e.g., "fuse" datasets in unique coordinate frame)
- image warping (e.g., change image perspective, correct lense distortion, etc.)
- Structure from Motion (SfM) (i.e., 3D reconstruction from multiple images)
- and much more! (e.g., augmented reality, etc.)

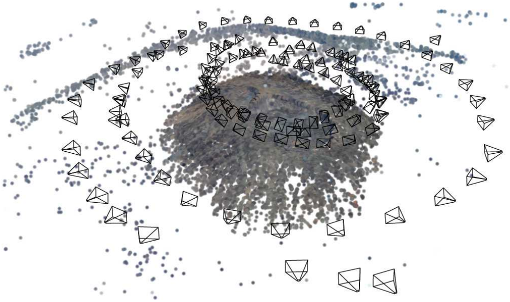


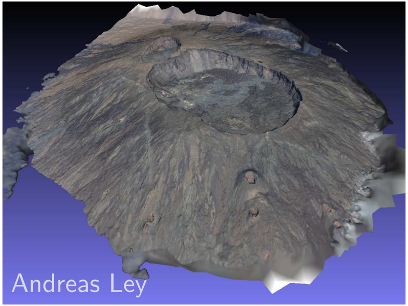
from Hartley & Zisserman

Homography is used to transform an image from one projective plane to another

Applications in image processing:

- image stitching (e.g., mosaic and panoramas)
- image registration (e.g., "fuse" datasets in unique coordinate frame)
- image warping (e.g., change image perspective, correct lense distortion, etc.)
- Structure from Motion (SfM) (i.e., 3D reconstruction from multiple images)
- and much more! (e.g., augmented reality, etc.)





Homography is used to transform an image from one projective plane to another

Applications in image processing:

- image stitching (e.g., mosaic and panoramas)
- image registration (e.g., "fuse" datasets in unique coordinate frame)
- image warping (e.g., change image perspective, correct lense distortion, etc.)
- Structure from Motion (SfM) (i.e., 3D reconstruction from multiple images)
- and much more! (e.g., augmented reality, etc.)

Geometric transformations map points from one space to another:

$$(x', y') = f(x, y)$$

$\Rightarrow$ in linear algebra, linear transformations can be represented by matrix operations:

$$\boxed{X' = MX} \tag{1}$$

where:

- $X = \begin{bmatrix} x \\ y \end{bmatrix}$ = original pixel coordinates
- $X' = \begin{bmatrix} x' \\ y' \end{bmatrix}$ = transformed pixel coordinates
- $\mathrm{M} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ = **transformation matrix**

Geometric transformations map points from one space to another:

$$(x', y') = f(x, y)$$

$\Rightarrow$ in linear algebra, linear transformations can be represented by matrix operations:

$$\boxed{X' = MX} \tag{1}$$

where:

- $X = \begin{bmatrix} x \\ y \end{bmatrix}$ = original pixel coordinates
- $X' = \begin{bmatrix} x' \\ y' \end{bmatrix}$ = transformed pixel coordinates
- $\mathrm{M} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ = **transformation matrix**

The matrix equation:

$$X' = MX$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Can we written as a linear system of equations:

$$\begin{cases} x' = ax + by \\ y' = cx + dy \end{cases}$$

The matrix equation:

$$X' = MX$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Can we written as a linear system of equations:

$$\begin{cases} x' = ax + by \\ y' = cx + dy \end{cases}$$

The matrix equation:

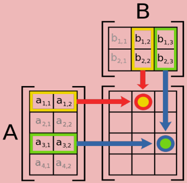
$$X' = MX$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Can we written as a linear system of equations:

$$\begin{cases} x' = ax + by \\ y' = cx + dy \end{cases}$$

Reminder: matrix multiplication



The matrix equation:

$$X' = MX$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

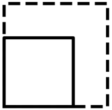
Can we written as a linear system of equations:

$$\begin{cases} x' = ax + by \\ y' = cx + dy \end{cases}$$

The transformation matrix M will determine the type of geometric transformation.

Example 1: scale points?

scaling

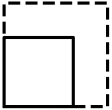


⇒ in Python this translates as:

```
import numpy as np
X = np.array([1, 1]).T            # original coordinates (x, y)
M = np.array([[2,0], [0,2]])      # transformation matrix
X_prime = M @ X                   # transformed coordinates (x', y')
# returns: X_prime = array([2, 2])
```

Example 1: scale points?

scaling



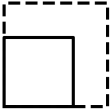
$$\begin{cases} x' = s_x * x \\ y' = s_y * y \end{cases}$$

⇒ in Python this translates as:

```
import numpy as np
X = np.array([1, 1]).T            # original coordinates (x, y)
M = np.array([[2,0], [0,2]])      # transformation matrix
X_prime = M @ X                   # transformed coordinates (x', y')
# returns: X_prime = array([2, 2])
```

Example 1: scale points?

scaling



$$\begin{cases} x' = s_x * x \\ y' = s_y * y \end{cases}$$

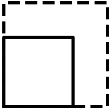
$$M = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

⇒ in Python this translates as:

```
import numpy as np
X = np.array([1, 1]).T              # original coordinates (x, y)
M = np.array([[2,0], [0,2]])        # transformation matrix
X_prime = M @ X                     # transformed coordinates (x', y')
# returns: X_prime = array([2, 2])
```

Example 1: scale points?

scaling



$$\begin{cases} x' = s_x * x \\ y' = s_y * y \end{cases} \qquad M = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

$\Rightarrow$ the point with coordinates $\begin{bmatrix} x \\ y \end{bmatrix}$ is transformed to coordinates $\begin{bmatrix} x' \\ y' \end{bmatrix}$ using the matrix multiplication:

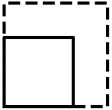
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x * x \\ s_y * y \end{bmatrix}$$

$\Rightarrow$ in Python this translates as:

```
import numpy as np
X = np.array([1, 1]).T           # original coordinates (x, y)
M = np.array([[2,0], [0,2]])     # transformation matrix
X_prime = M @ X                  # transformed coordinates (x', y')
# returns: X_prime = array([2, 2])
```

Example 1: scale points?

scaling



$$\begin{cases} x' = s_x * x \\ y' = s_y * y \end{cases} \qquad M = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

$\Rightarrow$ the point with coordinates $\begin{bmatrix} x \\ y \end{bmatrix}$ is transformed to coordinates $\begin{bmatrix} x' \\ y' \end{bmatrix}$ using the matrix multiplication:

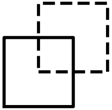
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x * x \\ s_y * y \end{bmatrix}$$

$\Rightarrow$ in Python this translates as:

```python
import numpy as np
X = np.array([1, 1]).T          # original coordinates (x, y)
M = np.array([[2,0], [0,2]])    # transformation matrix
X_prime = M @ X                 # transformed coordinates (x', y')
# returns: X_prime = array([2, 2])
```

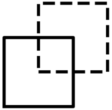
Example 2: translate points?

translation



Example 2: translate points?

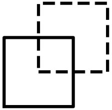
translation



$$\begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases}$$

Example 2: translate points?

translation

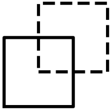


$$\begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases}$$

$$M = \left[\quad ? \quad \right]$$

Example 2: translate points?

translation

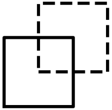


$$\begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases} \qquad M = \begin{bmatrix} \quad ? \quad \end{bmatrix}$$

$\Rightarrow$ add a component to the coordinates: redefine $X = \begin{bmatrix} x \\ y \end{bmatrix}$ as $\overline{X} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ = "augmented vector"

Example 2: translate points?

translation



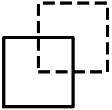
$$\begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases} \qquad M = \begin{bmatrix} \quad ? \quad \end{bmatrix}$$

$\Rightarrow$ add a component to the coordinates: redefine $X = \begin{bmatrix} x \\ y \end{bmatrix}$ as $\overline{X} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ = "augmented vector"

$\Rightarrow$ the transformation matrix to translate can now be defined as: $M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$

Example 2: translate points?

translation



$$\begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases} \qquad M = \begin{bmatrix} & ? & \end{bmatrix}$$

$\Rightarrow$ add a component to the coordinates: redefine $X = \begin{bmatrix} x \\ y \end{bmatrix}$ as $\overline{X} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ = "augmented vector"

$\Rightarrow$ the transformation matrix to translate can now be defined as: $M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$

$\Rightarrow$ hence the transformation coordinates can be calculated from:

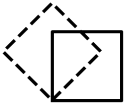
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1x + 0y + 1t_x \\ 0x + 1y + 1t_y \\ 0x + 0y + 1 \end{bmatrix}$$

$$= \begin{bmatrix} x + t_x \\ y + t_y \end{bmatrix}$$

Example 3: other simple transformations?

rotation

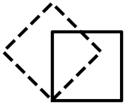


$$\begin{cases} x' = x * cos\theta - y * sin\theta \\ y' = x * cos\theta + y * sin\theta \end{cases}$$

$$M = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Example 3: other simple transformations?

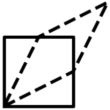
rotation



$$\begin{cases} x' = x * cos\theta - y * sin\theta \\ y' = x * cos\theta + y * sin\theta \end{cases}$$

$$M = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

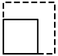
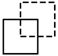
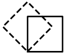
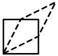
shear



$$\begin{cases} x' = x + s_v * y \\ y' = x * s_h + y \end{cases}$$

$$M = \begin{bmatrix} 1 & s_h & 0 \\ s_v & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

"primary" 2D transformations:

Transformation Type	Transformation Matrix M	Pixel Mapping Equation	
Identity	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x$ $y' = y$	
Scaling	$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = s_x * x$ $y' = s_y * y$	
Translation	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x + t_x$ $y' = y + t_y$	
Rotation (counter-clockwise about origin)	$\begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x * cos\theta - y * sin\theta$ $y' = x * cos\theta + y * sin\theta$	
Shear	$\begin{bmatrix} 1 & s_h & 0 \\ s_v & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x + s_v * y$ $y' = x * s_h + y$	

"composite" 2D transformations $\Rightarrow$ concatenation of "primary" transformations

Example 1: **Euclidean transformation** (a.k.a. "rigid transform", or "motion")
$\Rightarrow$ *rotation* followed by a *translation*

$\Rightarrow$ the transformation matrix is therefore defined as: (**read from right to left**, think like $f(g(x))$)

"composite" 2D transformations $\Rightarrow$ concatenation of "primary" transformations

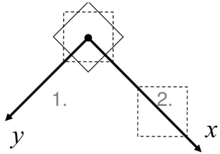
Example 1: **Euclidean transformation** (a.k.a. "rigid transform", or "motion")
$\Rightarrow$ *rotation* followed by a *translation*

$\Rightarrow$ the transformation matrix is therefore defined as: (**read from right to left**, think like $f(g(x))$)

"composite" 2D transformations $\Rightarrow$ concatenation of "primary" transformations

Example 1: **Euclidean transformation** (a.k.a. "rigid transform", or "motion")
$\Rightarrow$ *rotation* followed by a *translation*
$\Rightarrow$ the transformation matrix is therefore defined as: (**read from right to left**, think like $f(g(x))$)

$$\begin{aligned} M &= M_{translation} \cdot M_{rotation} \\ &= \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} cos\theta & -sin\theta & t_x \\ sin\theta & cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$



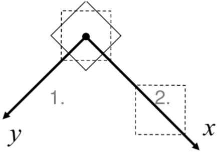
"composite" 2D transformations $\Rightarrow$ concatenation of "primary" transformations

Example 1: **Euclidean transformation** (a.k.a. "rigid transform", or "motion")

$\Rightarrow$ *rotation* followed by a *translation*

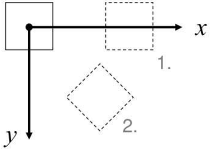
$\Rightarrow$ the transformation matrix is therefore defined as: (**read from right to left**, think like $f(g(x))$)

$$
\begin{aligned}
M &= M_{translation} \cdot M_{rotation} \\
&= \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} cos\theta & -sin\theta & t_x \\ sin\theta & cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$



order matters !

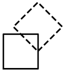
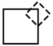
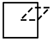
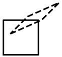
$$
\begin{aligned}
M &\neq M_{rotation} \cdot M_{translation} \\
&\neq \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \\
&\neq \begin{bmatrix} cos\theta & -sin\theta & t_x\,cos\theta - t_y\,sin\theta \\ sin\theta & cos\theta & t_y\,sin\theta\,cos\theta \\ 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$



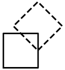
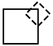
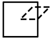
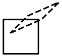
⇒ in Python this translates as:

```
import numpy as np

# set rotation transformation matrix
angle = np.deg2rad(45)
R = np.array([
    [np.cos(angle), -np.sin(angle), 0],
    [np.sin(angle),  np.cos(angle), 0],
    [0, 0, 1]])

# set translation transformation matrix
tx, ty = 1, .5
T = np.array([
    [1, 0, tx],
    [0, 1, ty],
    [0, 0, 1]])

# set original coordinates
X = np.array([
    [0, 0, 1],  # point 1 (x,y,w)
    [1, 0, 1],  # point 2 (x,y,w)
    [1, 1, 1],  # point 3 (x,y,w)
    [0, 1, 1]]) # point 4 (x,y,w)

# get euclidean transformation matrix as (1) rotation followed by (2) translation
M = T @ R

# get transformed coordinates (x', y')
X_prime = M @ X.T
```

"composite" 2D transformations:

Transformation Type	Transformation Matrix M	Pixel Mapping Equation	
Euclidean transformation (a.k.a. "rigid transform", or "motion") = rotation $\rightarrow$ translation	$\begin{bmatrix} cos\theta & -sin\theta & tx \\ sin\theta & cos\theta & ty \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x * cos\theta - y * sin\theta + tx$ $y' = x * sin\theta + y * cos\theta + ty$	
Similarity transformation = rotation $\rightarrow$ translation $\rightarrow$ scale	$\begin{bmatrix} a & -b & tx \\ b & a & ty \\ 0 & 0 & 1 \end{bmatrix}$	$x' = s * x * cos\theta - s * y * sin\theta + tx$ $y' = s * x * sin\theta + s * y * cos\theta + ty$	
Affine transformation = similarity $\rightarrow$ shear	$\begin{bmatrix} a & b & tx \\ c & d & ty \\ 0 & 0 & 1 \end{bmatrix}$	$x' = sx * x * cos(\theta) - sy * y * sin(\theta + shear) + tx$ $y' = sx * x * sin(\theta) + sy * y * cos(\theta + shear) + ty$	
Projective transformation (a.k.a. **homography**)	$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix}$	encompasses rotation, scaling, skew and perspective	

"composite" 2D transformations:

Transformation Type	Transformation Matrix M	Pixel Mapping Equation	
Euclidean transformation (a.k.a. "rigid transform", or "motion") = rotation → translation	$\begin{bmatrix} cos\theta & -sin\theta & tx \\ sin\theta & cos\theta & ty \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x * cos\theta - y * sin\theta + tx$ $y' = x * sin\theta + y * cos\theta + ty$	
Similarity transformation = rotation → translation → scale	$\begin{bmatrix} a & -b & tx \\ b & a & ty \\ 0 & 0 & 1 \end{bmatrix}$	$x' = s * x * cos\theta - s * y * sin\theta + tx$ $y' = s * x * sin\theta + s * y * cos\theta + ty$	
Affine transformation = similarity → shear	$\begin{bmatrix} a & b & tx \\ c & d & ty \\ 0 & 0 & 1 \end{bmatrix}$	$x' = sx * x * cos(\theta) - sy * y * sin(\theta + shear) + tx$ $y' = sx * x * sin(\theta) + sy * y * cos(\theta + shear) + ty$	
Projective transformation (a.k.a. **homography**)	$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix}$	encompasses rotation, scaling, skew and perspective	

⇒ the homography matrix H has 8 degrees of freedom (DOF):

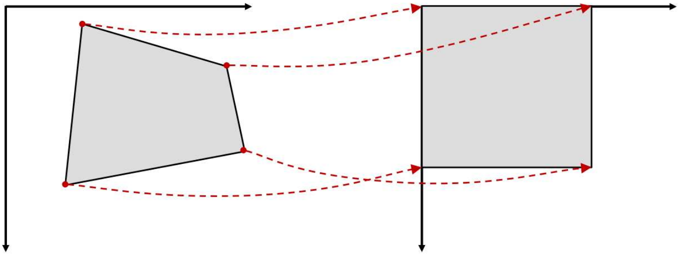
$$H = \begin{bmatrix} H_{00} & H_{01} & H_{02} \\ H_{10} & H_{11} & H_{12} \\ H_{20} & H_{21} & 1 \end{bmatrix}$$

⇒ estimating these parameters is key to transforming from one coordinate system to another

⇒ the homography matrix H has 8 degrees of freedom (DOF):

$$H = \begin{bmatrix} H_{00} & H_{01} & H_{02} \\ H_{10} & H_{11} & H_{12} \\ H_{20} & H_{21} & 1 \end{bmatrix}$$

⇒ <u>estimating these parameters is key to transforming from one coordinate system to another</u>



EX1: digital planar rectification

⇒ the homography matrix H has 8 degrees of freedom (DOF):

$$H = \begin{bmatrix} H_{00} & H_{01} & H_{02} \\ H_{10} & H_{11} & H_{12} \\ H_{20} & H_{21} & 1 \end{bmatrix}$$

⇒ <u>estimating these parameters is key to transforming from one coordinate system to another</u>

Original

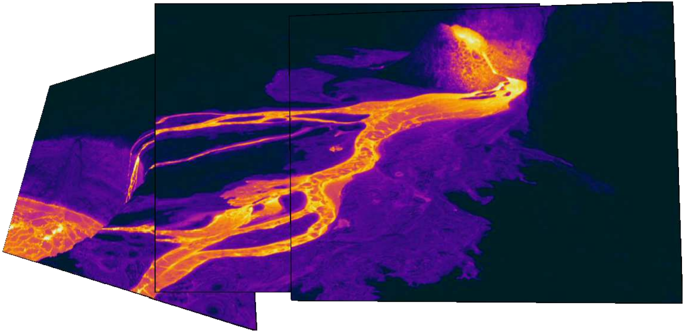
Transformed

EX1: digital planar rectification

⇒ the homography matrix H has 8 degrees of freedom (DOF):

$$H = \begin{bmatrix} H_{00} & H_{01} & H_{02} \\ H_{10} & H_{11} & H_{12} \\ H_{20} & H_{21} & 1 \end{bmatrix}$$

⇒ estimating these parameters is key to transforming from one coordinate system to another



EX2: panorama creation

How do we estimate these 8 parameters?

⇒ the **Direct Linear Transformation** (DLT) is an algorithm for computing H given ⩾ 4 correspondences

- Given: at least n ⩾ 4 point pairs $X_i \rightarrow X_i'$
- Wanted: 3×3 homography matrix H (8 DOF), for which $X_i' = HX_i$ holds

1. Reformulate the general projective transformation into a linear homogeneous equation system, i.e. Ah = 0

General projective transformation:

$$X' = HX$$

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} H_{00} & H_{01} & H_{02} \\ H_{10} & H_{11} & H_{12} \\ H_{20} & H_{21} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Write as linear equation system:

$$\begin{cases} x' = H_{00}x + H_{01}y + H_{02}w \\ y' = H_{10}x + H_{11}y + H_{12}w \\ w' = H_{20}x + H_{21}y + H_{22}w \end{cases}$$

Convert back from homogeneous to Euclidean coordinates by dividing with w', and move all terms to the left:

$$\frac{x'}{w'} - \frac{H_{00}x + H_{01}y + H_{02}}{H_{20}x + H_{21}y + H_{22}} = 0$$

$$\frac{y'}{w'} - \frac{H_{10}x + H_{11}y + H_{12}}{H_{20}x + H_{21}y + H_{22}} = 0$$

How do we estimate these 8 parameters?

$\Rightarrow$ the **Direct Linear Transformation** (DLT) is an algorithm for computing H given $\geqslant$ 4 correspondences

- Given: at least n $\geqslant$ 4 point pairs $X_i \rightarrow X_i'$
- Wanted: 3×3 homography matrix H (8 DOF), for which $X_i' = HX_i$ holds

1. Reformulate the general projective transformation into a linear homogeneous equation system, i.e. $Ah = 0$

General projective transformation:

$$X' = HX$$

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} H_{00} & H_{01} & H_{02} \\ H_{10} & H_{11} & H_{12} \\ H_{20} & H_{21} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Write as linear equation system:

$$\begin{cases} x' = H_{00}x + H_{01}y + H_{02}w \\ y' = H_{10}x + H_{11}y + H_{12}w \\ w' = H_{20}x + H_{21}y + H_{22}w \end{cases}$$

Convert back from homogeneous to Euclidean coordinates by dividing with w', and move all terms to the left:

$$\frac{x'}{w'} - \frac{H_{00}x + H_{01}y + H_{02}}{H_{20}x + H_{21}y + H_{22}} = 0$$

$$\frac{y'}{w'} - \frac{H_{10}x + H_{11}y + H_{12}}{H_{20}x + H_{21}y + H_{22}} = 0$$

How do we estimate these 8 parameters?

⇒ the **Direct Linear Transformation** (DLT) is an algorithm for computing H given ⩾ 4 correspondences

- Given: at least n ⩾ 4 point pairs $X_i \rightarrow X_i'$
- Wanted: 3×3 homography matrix H (8 DOF), for which $X_i' = HX_i$ holds

1. Reformulate the general projective transformation into a linear homogeneous equation system, i.e. $Ah = 0$

General projective transformation:

$$X' = HX$$

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} H_{00} & H_{01} & H_{02} \\ H_{10} & H_{11} & H_{12} \\ H_{20} & H_{21} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Write as linear equation system:

$$\begin{cases} x' = H_{00}x + H_{01}y + H_{02}w \\ y' = H_{10}x + H_{11}y + H_{12}w \\ w' = H_{20}x + H_{21}y + H_{22}w \end{cases}$$

Convert back from homogeneous to Euclidean coordinates by dividing with w', and move all terms to the left:

$$\frac{x'}{w'} - \frac{H_{00}x + H_{01}y + H_{02}}{H_{20}x + H_{21}y + H_{22}} = 0$$

$$\frac{y'}{w'} - \frac{H_{10}x + H_{11}y + H_{12}}{H_{20}x + H_{21}y + H_{22}} = 0$$

How do we estimate these 8 parameters?

⇒ the **Direct Linear Transformation** (DLT) is an algorithm for computing H given $\geqslant 4$ correspondences

- Given: at least n $\geqslant 4$ point pairs $X_i \rightarrow X_i'$
- Wanted: 3×3 homography matrix H (8 DOF), for which $X_i' = HX_i$ holds

1. Reformulate the general projective transformation into a linear homogeneous equation system, i.e. $Ah = 0$

General projective transformation:

$$X' = HX$$

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} H_{00} & H_{01} & H_{02} \\ H_{10} & H_{11} & H_{12} \\ H_{20} & H_{21} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Write as linear equation system:

$$\begin{cases} x' = H_{00}x + H_{01}y + H_{02}w \\ y' = H_{10}x + H_{11}y + H_{12}w \\ w' = H_{20}x + H_{21}y + H_{22}w \end{cases}$$

Convert back from homogeneous to Euclidean coordinates by dividing with w', and move all terms to the left:

$$\frac{x'}{w'} - \frac{H_{00}x + H_{01}y + H_{02}}{H_{20}x + H_{21}y + H_{22}} = 0$$

$$\frac{y'}{w'} - \frac{H_{10}x + H_{11}y + H_{12}}{H_{20}x + H_{21}y + H_{22}} = 0$$

How do we estimate these 8 parameters?

⇒ the **Direct Linear Transformation** (DLT) is an algorithm for computing H given $\geqslant 4$ correspondences

- Given: at least n $\geqslant 4$ point pairs $X_i \rightarrow X_i'$
- Wanted: 3×3 homography matrix H (8 DOF), for which $X_i' = HX_i$ holds

1. Reformulate the general projective transformation into a linear homogeneous equation system, i.e. Ah = 0

General projective transformation:

$$X' = HX$$

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} H_{00} & H_{01} & H_{02} \\ H_{10} & H_{11} & H_{12} \\ H_{20} & H_{21} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Write as linear equation system:

$$\begin{cases} x' = H_{00}x + H_{01}y + H_{02}w \\ y' = H_{10}x + H_{11}y + H_{12}w \\ w' = H_{20}x + H_{21}y + H_{22}w \end{cases}$$

Convert back from homogeneous to Euclidean coordinates by dividing with w', and move all terms to the left:

$$\frac{x'}{w'} - \frac{H_{00}x + H_{01}y + H_{02}}{H_{20}x + H_{21}y + H_{22}} = 0$$

$$\frac{y'}{w'} - \frac{H_{10}x + H_{11}y + H_{12}}{H_{20}x + H_{21}y + H_{22}} = 0$$

How do we estimate these 8 parameters?

$\Rightarrow$ the **Direct Linear Transformation** (DLT) is an algorithm for computing H given $\geqslant 4$ correspondences

- Given: at least n $\geqslant$ 4 point pairs $X_i \rightarrow X_i'$
- Wanted: 3×3 homography matrix H (8 DOF), for which $X_i' = HX_i$ holds

1. Reformulate the general projective transformation into a linear homogeneous equation system, i.e. Ah = 0

General projective transformation:

$$X' = HX$$

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} H_{00} & H_{01} & H_{02} \\ H_{10} & H_{11} & H_{12} \\ H_{20} & H_{21} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Write as linear equation system:

$$\begin{cases} x' = H_{00}x + H_{01}y + H_{02}w \\ y' = H_{10}x + H_{11}y + H_{12}w \\ w' = H_{20}x + H_{21}y + H_{22}w \end{cases}$$

Convert back from homogeneous to Euclidean coordinates by dividing with w', and move all terms to the left:

$$\frac{x'}{w'} - \frac{H_{00}x + H_{01}y + H_{02}}{H_{20}x + H_{21}y + H_{22}} = 0$$

$$\frac{y'}{w'} - \frac{H_{10}x + H_{11}y + H_{12}}{H_{20}x + H_{21}y + H_{22}} = 0$$

1. (continued)

Multiplying by the denominator yields:

$$\frac{x'}{w'}(H_{20}x + H_{21}y + H_{22}) - H_{00}x - H_{01}y - H_{02} = 0$$
$$\frac{y'}{w'}(H_{20}x + H_{21}y + H_{22}) - H_{10}x - H_{11}y - H_{12} = 0$$

Which can be written as the system:

$$\begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & \frac{x'x}{w'} & \frac{x'y}{w'} & \frac{x'}{w'} \\ 0 & 0 & 0 & -x & -y & -1 & \frac{y'x}{w'} & \frac{y'y}{w'} & \frac{y'}{w'} \end{bmatrix} \begin{bmatrix} H_{01} \\ H_{02} \\ \vdots \\ H_{21} \\ H_{22} \end{bmatrix} = 0$$

We now have to solve the homogeneous set of linear equations:

$$Ah = 0$$

where:

- A is the "design matrix", in which each point correspondence n fills 2 rows (2 observations per point: x and y coordinates), so that n point correspondences yields a 2n × 9 matrix:

$$A = \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & \frac{x_1'x_1}{w'} & \frac{x_1'y_1}{w'} & \frac{x_1'}{w'} \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & \frac{y_1'x_1}{w'} & \frac{y_1'y_1}{w'} & \frac{y_1'}{w'} \\ & \vdots & & \vdots & & \vdots & & \vdots & \end{bmatrix}$$

- h is the vector of unknowns: $h = \begin{bmatrix} H_{01} & H_{02} & H_{02} & H_{10} & H_{11} & H_{12} & H_{20} & H_{21} & H_{22} \end{bmatrix}^T$

1. (continued)

Multiplying by the denominator yields:

$$\frac{x'}{w'}(H_{20}x + H_{21}y + H_{22}) - H_{00}x - H_{01}y - H_{02} = 0$$
$$\frac{y'}{w'}(H_{20}x + H_{21}y + H_{22}) - H_{10}x - H_{11}y - H_{12} = 0$$

Which can be written as the system:

$$\begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & \frac{x'x}{w'} & \frac{x'y}{w'} & \frac{x'}{w'} \\ 0 & 0 & 0 & -x & -y & -1 & \frac{y'x}{w'} & \frac{y'y}{w'} & \frac{y'}{w'} \end{bmatrix} \begin{bmatrix} H_{01} \\ H_{02} \\ \vdots \\ H_{21} \\ H_{22} \end{bmatrix} = 0$$

We now have to solve the homogeneous set of linear equations:

$$Ah = 0$$

where:

- A is the "design matrix", in which each point correspondence n fills 2 rows (2 observations per point: x and y coordinates), so that n point correspondences yields a 2n $\times$ 9 matrix:

$$A = \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & \frac{x_1'x_1}{w'} & \frac{x_1'y_1}{w'} & \frac{x_1'}{w'} \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & \frac{y_1'x_1}{w'} & \frac{y_1'y_1}{w'} & \frac{y_1'}{w'} \\ & \vdots & & \vdots & & \vdots & & \vdots & \end{bmatrix}$$

- h is the vector of unknowns: $h = \begin{bmatrix} H_{01} & H_{02} & H_{02} & H_{10} & H_{11} & H_{12} & H_{20} & H_{21} & H_{22} \end{bmatrix}^T$

1. (continued)

Multiplying by the denominator yields:

$$\frac{x'}{w'}(H_{20}x + H_{21}y + H_{22}) - H_{00}x - H_{01}y - H_{02} = 0$$
$$\frac{y'}{w'}(H_{20}x + H_{21}y + H_{22}) - H_{10}x - H_{11}y - H_{12} = 0$$

Which can be written as the system:

$$\begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & \frac{x'x}{w'} & \frac{x'y}{w'} & \frac{x'}{w'} \\ 0 & 0 & 0 & -x & -y & -1 & \frac{y'x}{w'} & \frac{y'y}{w'} & \frac{y'}{w'} \end{bmatrix} \begin{bmatrix} H_{01} \\ H_{02} \\ \vdots \\ H_{21} \\ H_{22} \end{bmatrix} = 0$$

We now have to solve the homogeneous set of linear equations:

$$\boxed{Ah = 0}$$

where:

- A is the "design matrix", in which each point correspondence n fills 2 rows (2 observations per point: x and y coordinates), so that n point correspondences yields a 2n $\times$ 9 matrix:

$$A = \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & \frac{x'_1 x_1}{w'} & \frac{x'_1 y_1}{w'} & \frac{x'_1}{w'} \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & \frac{y'_1 x_1}{w'} & \frac{y'_1 y_1}{w'} & \frac{y'_1}{w'} \\ & \vdots & & \vdots & & \vdots & & \vdots & \end{bmatrix}$$

- h is the vector of unknowns: $h = \begin{bmatrix} H_{01} & H_{02} & H_{02} & H_{10} & H_{11} & H_{12} & H_{20} & H_{21} & H_{22} \end{bmatrix}^T$

1. (continued)

Multiplying by the denominator yields:

$$\frac{x'}{w'}(H_{20}x + H_{21}y + H_{22}) - H_{00}x - H_{01}y - H_{02} = 0$$
$$\frac{y'}{w'}(H_{20}x + H_{21}y + H_{22}) - H_{10}x - H_{11}y - H_{12} = 0$$

Which can be written as the system:

$$\begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & \frac{x'x}{w'} & \frac{x'y}{w'} & \frac{x'}{w'} \\ 0 & 0 & 0 & -x & -y & -1 & \frac{y'x}{w'} & \frac{y'y}{w'} & \frac{y'}{w'} \end{bmatrix} \begin{bmatrix} H_{01} \\ H_{02} \\ \vdots \\ H_{21} \\ H_{22} \end{bmatrix} = 0$$

We now have to solve the homogeneous set of linear equations:

$$\boxed{Ah = 0}$$

where:

- A is the "design matrix", in which each point correspondence n fills 2 rows (2 observations per point: x and y coordinates), so that n point correspondences yields a 2n $\times$ 9 matrix:

$$A = \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & \frac{x_1'x_1}{w'} & \frac{x_1'y_1}{w'} & \frac{x_1'}{w'} \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & \frac{y_1'x_1}{w'} & \frac{y_1'y_1}{w'} & \frac{y_1'}{w'} \\ & \vdots & & \vdots & & \vdots & & \vdots & \end{bmatrix}$$

- h is the vector of unknowns: $h = \begin{bmatrix} H_{01} & H_{02} & H_{02} & H_{10} & H_{11} & H_{12} & H_{20} & H_{21} & H_{22} \end{bmatrix}^T$

2. Solve the homogeneous equation system with Singular Value Decomposition (SVD)

Note: SVD is generally used for finding solutions of over-determined systems.

The "singular value decomposition" of matrix A is a factorization of the form:

$$A = UDV^T$$

where:
- the diagonal elements of D (arranged to be non-negative and in decreasing order of magnitude), are called **singular values**
- the matrices U and V are called left and right **singular vectors** respectively

$\Rightarrow$ **the least squares solution is found as the last row of the matrix V of the SVD**

$\Rightarrow$ this translate in Python as:

```
import numpy as np
U,S,V = np.linalg.svd(A)     # singular value decomposition of A
h = V[8]                     # least squares solution found as the last row of V
H = h.reshape((3,3))         # reshape into 3x3 homography matrix
```

2. Solve the homogeneous equation system with Singular Value Decomposition (SVD)

Note: SVD is generally used for finding solutions of over-determined systems.

The "singular value decomposition" of matrix A is a factorization of the form:

$$A = UDV^T$$

where:
- the diagonal elements of D (arranged to be non-negative and in decreasing order of magnitude), are called **singular values**
- the matrices U and V are called left and right **singular vectors** respectively

⇒ **the least squares solution is found as the last row of the matrix V of the SVD**

⇒ this translate in Python as:

```
import numpy as np
U,S,V = np.linalg.svd(A)     # singular value decomposition of A
h = V[8]                     # least squares solution found as the last row of V
H = h.reshape((3,3))         # reshape into 3x3 homography matrix
```

2. Solve the homogeneous equation system with Singular Value Decomposition (SVD)

Note: SVD is generally used for finding solutions of over-determined systems.

The "singular value decomposition" of matrix A is a factorization of the form:

$$A = UDV^T$$

where:
- the diagonal elements of D (arranged to be non-negative and in decreasing order of magnitude), are called **singular values**
- the matrices U and V are called left and right **singular vectors** respectively

⇒ **the least squares solution is found as the last row of the matrix V of the SVD**

⇒ this translate in Python as:

```
import numpy as np
U,S,V = np.linalg.svd(A)     # singular value decomposition of A
h = V[8]                     # least squares solution found as the last row of V
H = h.reshape((3,3))         # reshape into 3x3 homography matrix
```

2. Solve the homogeneous equation system with Singular Value Decomposition (SVD)

Note: SVD is generally used for finding solutions of over-determined systems.

The "singular value decomposition" of matrix A is a factorization of the form:

$$A = UDV^T$$

where:
- the diagonal elements of D (arranged to be non-negative and in decreasing order of magnitude), are called **singular values**
- the matrices U and V are called left and right **singular vectors** respectively

$\Rightarrow$ **the least squares solution is found as the last row of the matrix V of the SVD**

$\Rightarrow$ this translate in Python as:

```
import numpy as np
U,S,V = np.linalg.svd(A)     # singular value decomposition of A
h = V[8]                     # least squares solution found as the last row of V
H = h.reshape((3,3))         # reshape into 3x3 homography matrix
```

3. Conditioning & Unconditioning of points

In order to stabilize the solution, the points need to be conditioned before creating the design matrix A and solving for H

$\Rightarrow$ the points are conditioned by normalizing so that they have zero mean and unit standard deviation

3. Conditioning & Unconditioning of points

In order to stabilize the solution, the points need to be conditioned before creating the design matrix A and solving for H

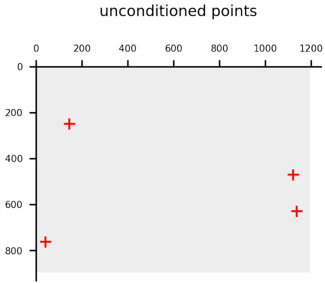
$\Rightarrow$ the points are conditioned by normalizing so that they have zero mean and unit standard deviation

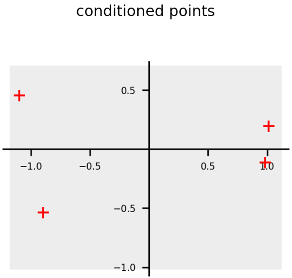
point selection

3. Conditioning & Unconditioning of points

In order to stabilize the solution, the points need to be conditioned before creating the design matrix A and solving for H

⇒ the points are conditioned by normalizing so that they have zero mean and unit standard deviation



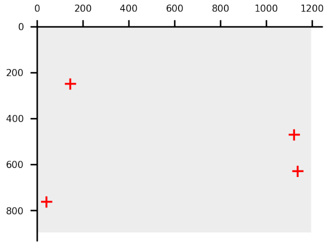


3. Conditioning & Unconditioning of points

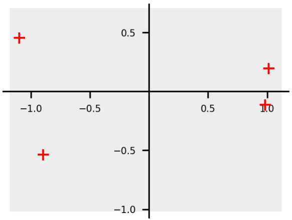
In order to stabilize the solution, the points need to be conditioned before creating the design matrix A and solving for H

$\Rightarrow$ the points are conditioned by normalizing so that they have zero mean and unit standard deviation

unconditioned points



conditioned points



$\Rightarrow$ can be done with the conditioning matrix C (consisting of scaling & translation to origin):

$$C = \begin{bmatrix} s & 0 & tx \\ 0 & s & ty \\ 0 & 0 & 1 \end{bmatrix}$$

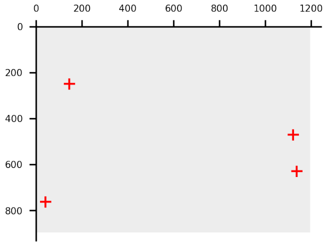
where: $s = \frac{1}{max([std_X, std_X y])}$, $t_x = \frac{-mean_x}{max([std_X, std_X y])}$, and $t_y = \frac{-mean_y}{max([std_X, std_X y])}$

3. Conditioning & Unconditioning of points

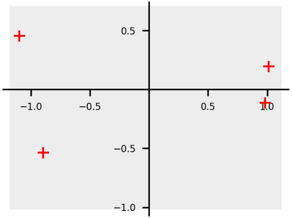
In order to stabilize the solution, the points need to be conditioned before creating the design matrix A and solving for H

$\Rightarrow$ the points are conditioned by normalizing so that they have zero mean and unit standard deviation

unconditioned points



conditioned points



$\Rightarrow$ can be done with the conditioning matrix C (consisting of scaling & translation to origin):

$$C = \begin{bmatrix} s & 0 & tx \\ 0 & s & ty \\ 0 & 0 & 1 \end{bmatrix}$$

where: $s = \frac{1}{max([std_x, std_x y])}$, $t_x = \frac{-mean_x}{max([std_x, std_x y])}$, and $t_y = \frac{-mean_y}{max([std_x, std_x y])}$

$\Rightarrow$ conditioned coordinates are then calculated (for points pairs in original & new coordinate systems) as: $\widetilde{X} = C_1 X$ and $\widetilde{X'} = C_2 X'$

3. (continued)

The solved H matrix is in conditioned coordinates, so it must be decondition before it can be used:

$\Rightarrow$ conditioned homography matrix: $\widetilde{H} = \begin{bmatrix} \widetilde{H}_{00} & \widetilde{H}_{01} & \widetilde{H}_{02} \\ \widetilde{H}_{10} & \widetilde{H}_{11} & \widetilde{H}_{12} \\ \widetilde{H}_{20} & \widetilde{H}_{21} & 1 \end{bmatrix}$

$\Rightarrow$ unconditioned homography matrix: $H = C_2^{-1}\widetilde{H}C_1 = \begin{bmatrix} H_{00} & H_{01} & H_{02} \\ H_{10} & H_{11} & H_{12} \\ H_{20} & H_{21} & 1 \end{bmatrix}$

Then what?

$\Rightarrow$ applying the transformation matrix H on an image is called **warping**

Case examples:

Then what?

⇒ applying the transformation matrix H on an image is called **warping**

Case examples:

1. Projection rectification

⇒ use the estimated homography to change the projection of an image

Original

Transformed

Then what?
⇒ applying the transformation matrix H on an image is called **warping**

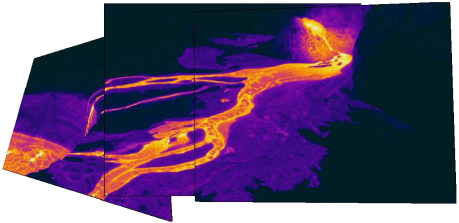
Case examples:

1. Projection rectification

⇒ use the estimated homography to change the projection of an image

2. Panorama stitching

⇒ use the estimated homography(ies) to adapt image(s) to a central image



We have seen that homographies can be computed directly from corresponding points in two images:

> $\Rightarrow$ *since a full projective transformation (homography) has 8 degrees of freedom, and since each point correspondence gives two equations, (one each for the x and y coordinates), $\geqslant 4$ points correspondences are needed to compute H*

However manually selecting corresponding points is cumbersome and not scalable!

Solution? Identify **interest points** in image(s)

> $\Rightarrow$ provide distinctive image points
> $\Rightarrow$ used in tracking (optical flow), object recognition, Structure from Motion

Example of most common interest points:

- Corner Detectors (e.g., Harris, Shi-Tomasi, Förstner, etc.)
- Blob and Ridge Detectors (e.g., LoG, DoG, Hessian, etc.)
- Features: SIFT, HOG, ORB, etc.

We have seen that homographies can be computed directly from corresponding points in two images:

> $\Rightarrow$ *since a full projective transformation (homography) has 8 degrees of freedom, and since each point correspondence gives two equations, (one each for the x and y coordinates), $\geqslant 4$ points correspondences are needed to compute H*

However manually selecting corresponding points is cumbersome and not scalable!

Solution? Identify **interest points** in image(s)

> $\Rightarrow$ provide distinctive image points
> $\Rightarrow$ used in tracking (optical flow), object recognition, Structure from Motion

Example of most common interest points:

- Corner Detectors (e.g., Harris, Shi-Tomasi, Förstner, etc.)
- Blob and Ridge Detectors (e.g., LoG, DoG, Hessian, etc.)
- Features: SIFT, HOG, ORB, etc.

We have seen that homographies can be computed directly from corresponding points in two images:

> $\Rightarrow$ *since a full projective transformation (homography) has 8 degrees of freedom, and since each point correspondence gives two equations, (one each for the x and y coordinates), $\geqslant 4$ points correspondences are needed to compute H*

However manually selecting corresponding points is cumbersome and not scalable!

Solution? Identify **interest points** in image(s)

> $\Rightarrow$ provide distinctive image points
> $\Rightarrow$ used in tracking (optical flow), object recognition, Structure from Motion

Example of most common interest points:

- Corner Detectors (e.g., Harris, Shi-Tomasi, Förstner, etc.)
- Blob and Ridge Detectors (e.g., LoG, DoG, Hessian, etc.)
- Features: SIFT, HOG, ORB, etc.

We have seen that homographies can be computed directly from corresponding points in two images:

> $\Rightarrow$ *since a full projective transformation (homography) has 8 degrees of freedom, and since each point correspondence gives two equations, (one each for the x and y coordinates), $\geqslant 4$ points correspondences are needed to compute H*

However manually selecting corresponding points is cumbersome and not scalable!

Solution? Identify **interest points** in image(s)

> $\Rightarrow$ provide distinctive image points
> $\Rightarrow$ used in tracking (optical flow), object recognition, Structure from Motion

Example of most common interest points:

- Corner Detectors (e.g., Harris, Shi-Tomasi, Förstner, etc.)
- Blob and Ridge Detectors (e.g., LoG, DoG, Hessian, etc.)
- Features: SIFT, HOG, ORB, etc.

We have seen that homographies can be computed directly from corresponding points in two images:

> $\Rightarrow$ *since a full projective transformation (homography) has 8 degrees of freedom, and since each point correspondence gives two equations, (one each for the x and y coordinates), $\geqslant 4$ points correspondences are needed to compute H*

However manually selecting corresponding points is cumbersome and not scalable!

Solution? Identify **interest points** in image(s)

$\Rightarrow$ provide distinctive image points
$\Rightarrow$ used in tracking (optical flow), object recognition, Structure from Motion

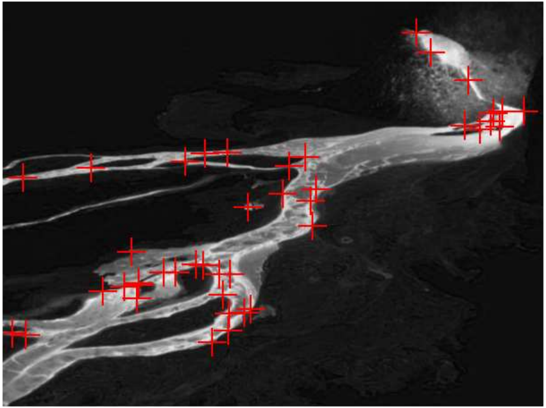
Example of most common interest points:

- Corner Detectors (e.g., Harris, Shi-Tomasi, Förstner, etc.)
- Blob and Ridge Detectors (e.g., LoG, DoG, Hessian, etc.)
- Features: SIFT, HOG, ORB, etc.

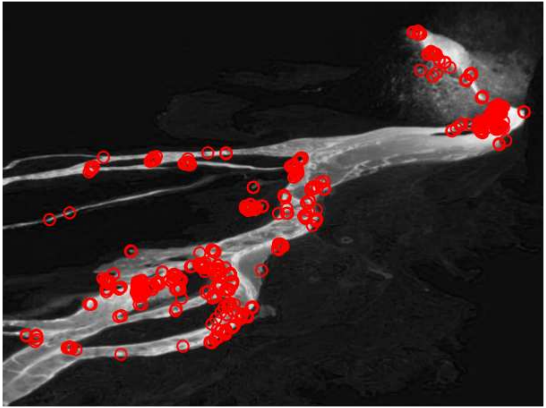
> we will discuss in more detail about *interest points* and *features* during the next lecture

Example: **Harris corners** & **ORB features** detected automatically in an image

Harris corners



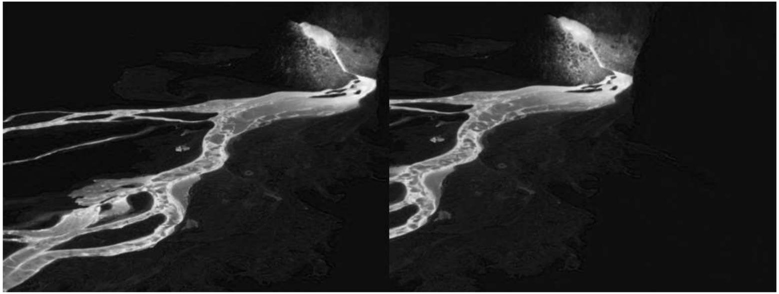
ORB features



How can we use *interest points* to create panoramas?

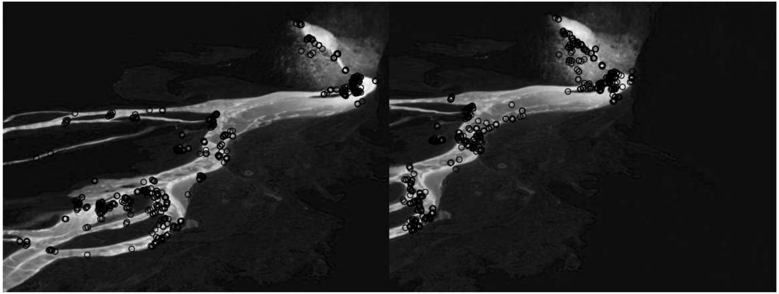
1. take images with overlap
2. detect ORB features in both images seperately
3. detect matching features between both images
4. remove outliers with RANSAC (robust iterative regression algorithm, resistant to outliers)
5. estimate homography and warp

How can we use *interest points* to create panoramas?



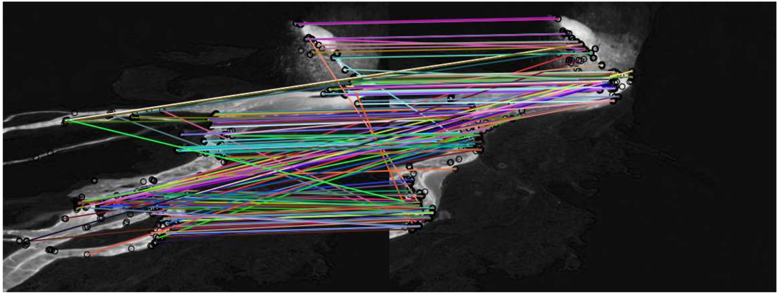
1. take images with overlap
2. detect ORB features in both images seperately
3. detect matching features between both images
4. remove outliers with RANSAC (robust iterative regression algorithm, resistant to outliers)
5. estimate homography and warp

How can we use *interest points* to create panoramas?



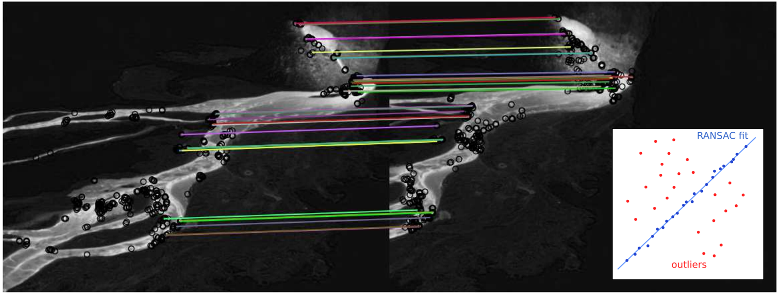
1. take images with overlap
2. detect ORB features in both images seperately
3. detect matching features between both images
4. remove outliers with RANSAC (robust iterative regression algorithm, resistant to outliers)
5. estimate homography and warp

How can we use *interest points* to create panoramas?



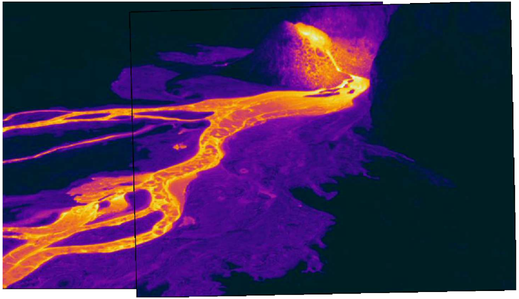
1. take images with overlap
2. detect ORB features in both images seperately
3. detect matching features between both images
4. remove outliers with RANSAC (robust iterative regression algorithm, resistant to outliers)
5. estimate homography and warp

How can we use *interest points* to create panoramas?



1. take images with overlap
2. detect ORB features in both images seperately
3. detect matching features between both images
4. remove outliers with RANSAC (robust iterative regression algorithm, resistant to outliers)
5. estimate homography and warp

How can we use *interest points* to create panoramas?



1. take images with overlap
2. detect ORB features in both images seperately
3. detect matching features between both images
4. remove outliers with RANSAC (robust iterative regression algorithm, resistant to outliers)
5. estimate homography and warp

Exercices !