Image formation, transformations, etc.

Sensors and Image Formation

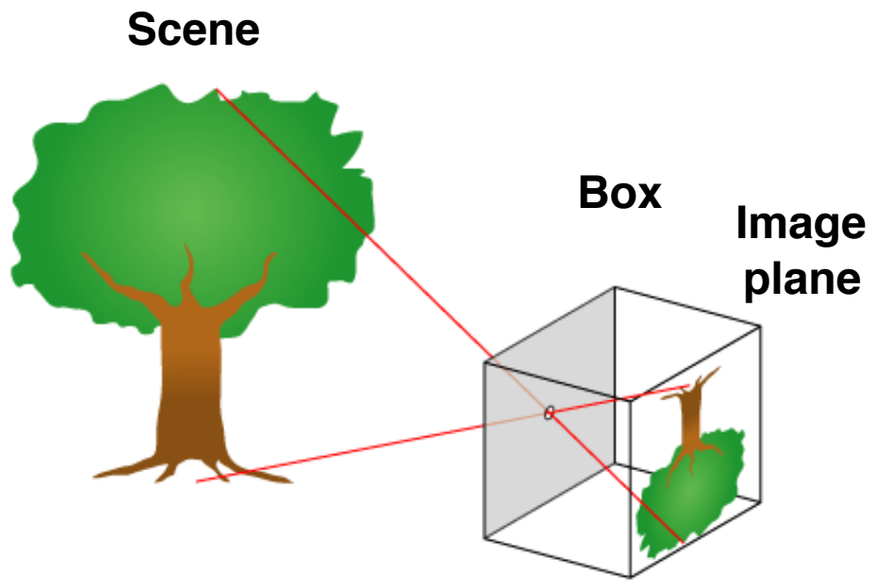
- Imaging sensors and models of image formation
- Coordinate systems

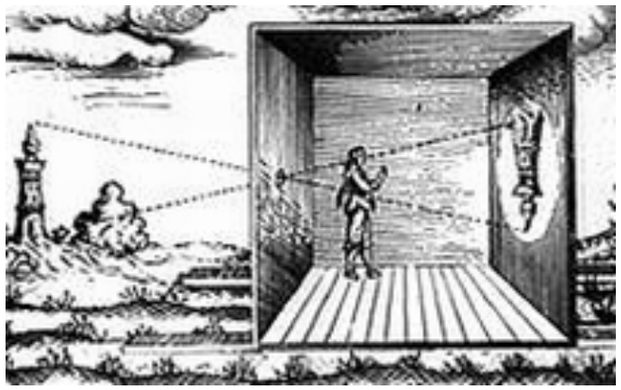
Pinhole Camera Model

- A good lens can be modeled by a pinhole camera; ie., each ray from the scene passes undeflected to the image plane

- Simple equations describe projection of a scene point onto the image plane (“perspective projection”)

- We will use the pinhole camera model exclusively, except for a little later in the course where we model lens distortion in real cameras



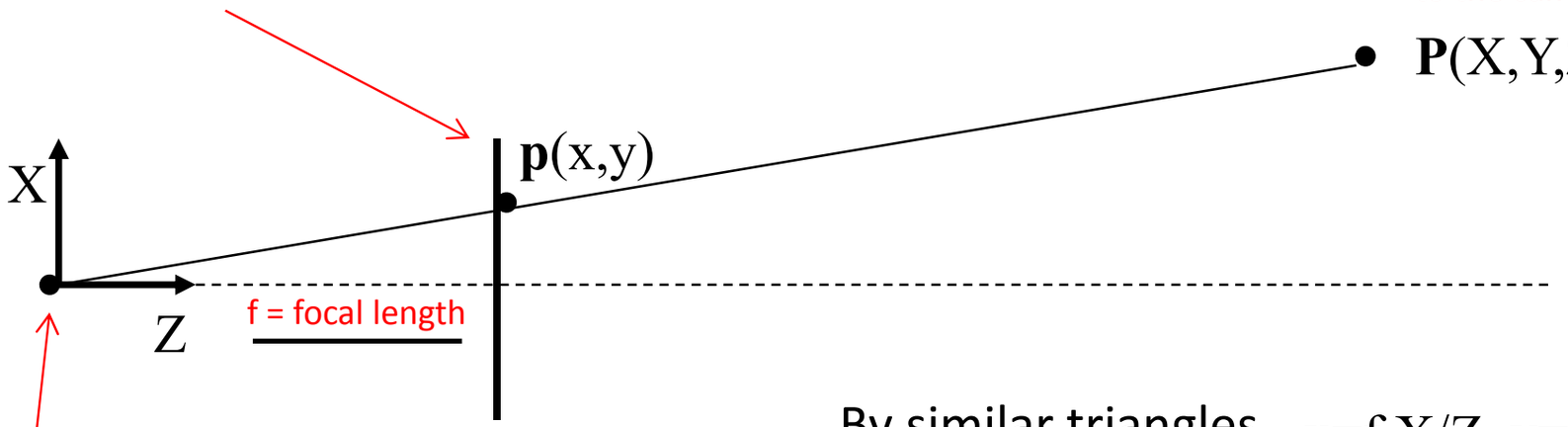


The pinhole camera (“camera obscura”) was used by Renaissance painters to help them understand perspective projection

Perspective Projection Equations

For convenience (to avoid an inverted image) we treat the image plane as if it were in front of the pinhole

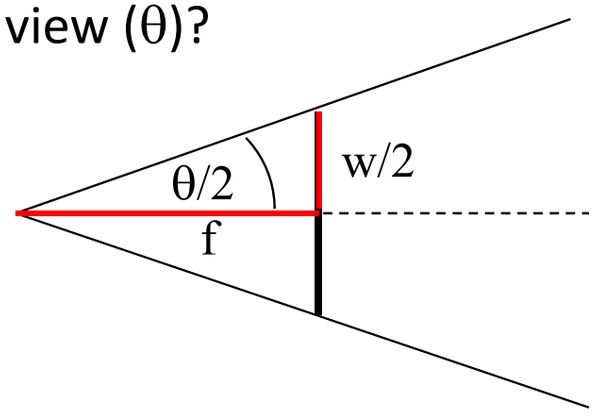
The XYZ coordinates of the point are with respect to the camera origin



We define the origin of the camera's coordinate system at the pinhole (note – this is a 3D XYZ coordinate frame

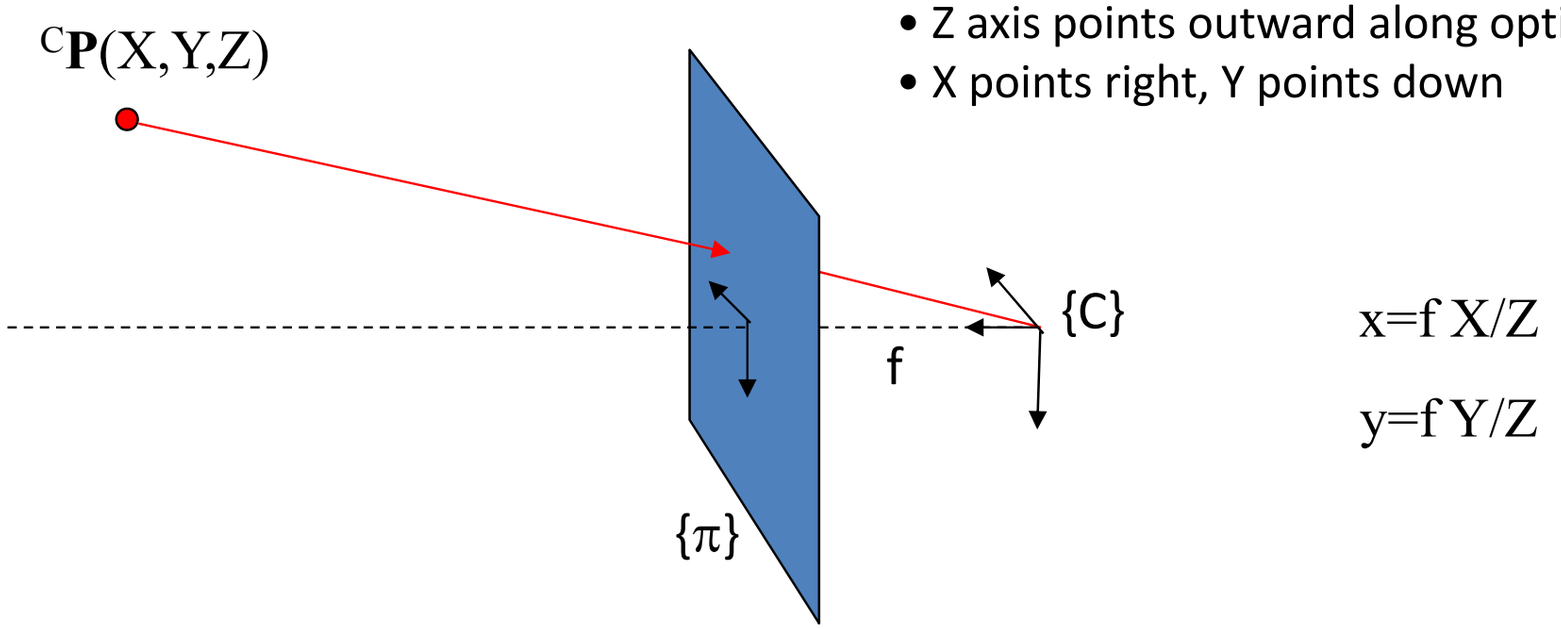
By similar triangles, $x=f\,X/Z,\ y=f\,Y/Z$

Field of view (θ)?



$\tan(\theta/2) = (w/2)/f$

Camera vs Image Plane Coords



Camera coordinate system {C}

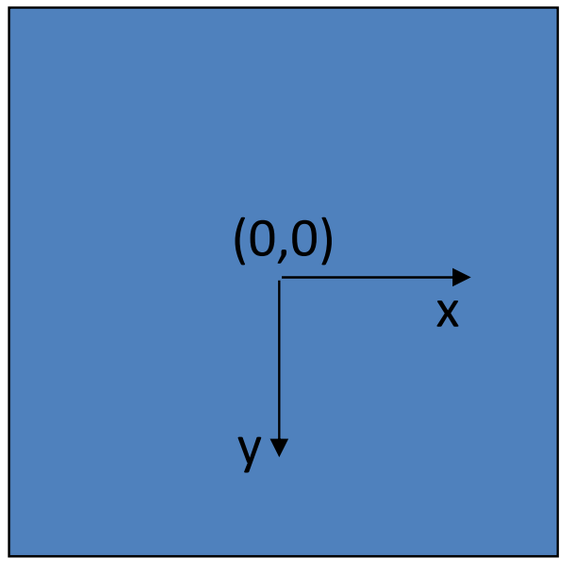
- A 3D coordinate system (X,Y,Z) – units say, in meters
- Origin at the center of projection
- Z axis points outward along optical axis
- X points right, Y points down

Image plane coordinate system {π}

- A 2D coordinate system (x,y) – units in mm
- Origin at the intersection of the optical axis with the image plane
- In real systems, this is where the CCD or CMOS plane is

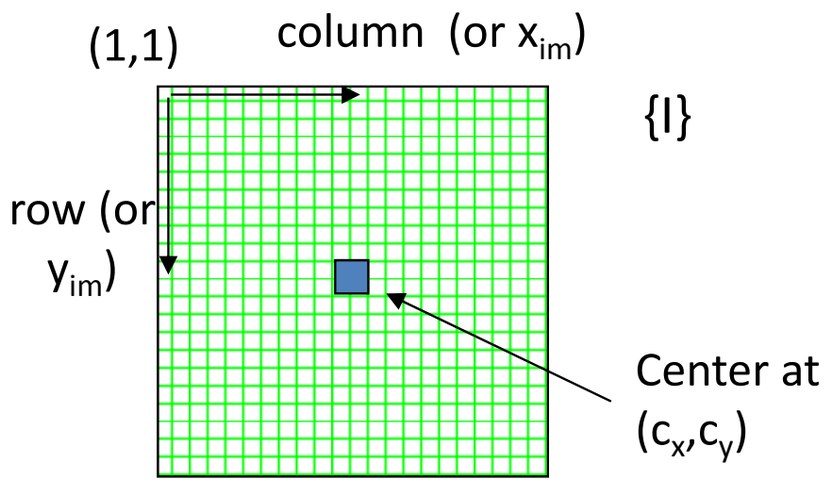
Image Buffer

- Image plane
  - The real image is formed on the CCD plane
  - (x,y) units in mm
  - Origin in center (principal point)



{π}

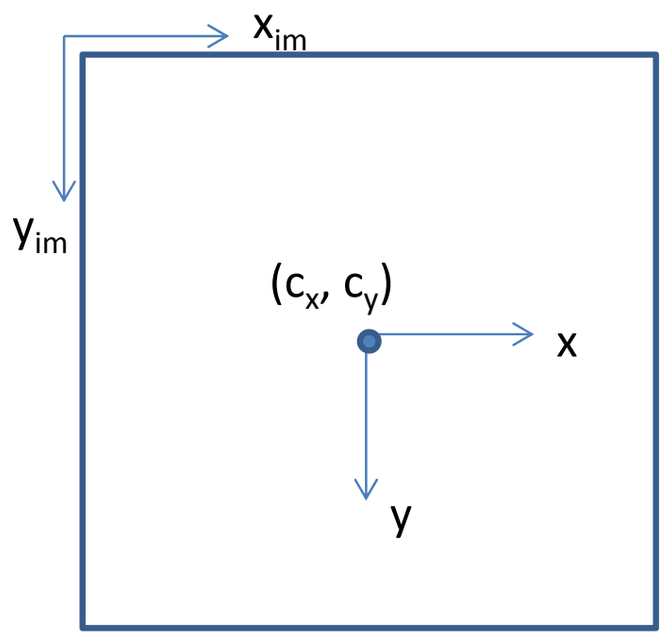
- Image buffer
  - Digital (or pixel) image
  - (row, col) indices
  - We can also use (x_{im}, y_{im})
  - Origin in upper left



{I}

Conversion between real image and pixel image coordinates

- Assume
  - The image <u>center</u> (principal point) is located at pixel (c_x,c_y) in the pixel image
  - The <u>spacing</u> of the pixels is (s_x,s_y) in millimeters



- Then

$$x = (x_{im} - c_x)\, s_x \qquad x_{im} = x/s_x + c_x$$
$$y = (y_{im} - c_y)\, s_y \qquad y_{im} = y/s_y + c_y$$

Note on focal length

- Recall

  $x = (x_{im} - c_x)\, s_x$

  $y = (y_{im} - c_y)\, s_y$

- or

  $x_{im} = x/s_x + c_x$

  $y_{im} = y/s_y + c_y$

- and

  $x = f\, X/Z$

  $y = f\, Y/Z$

- So

  $x_{im} = (f / s_x)\, X/Z + c_x$

  $y_{im} = (f / s_y)\, Y/Z + c_y$

- All we really need is

  $f_x = (f / s_x)$

  $f_y = (f / s_y)$

- We don't need to know the actual values of f and s_x,s_y; just their ratios

- We can alternatively express focal length in units of pixels

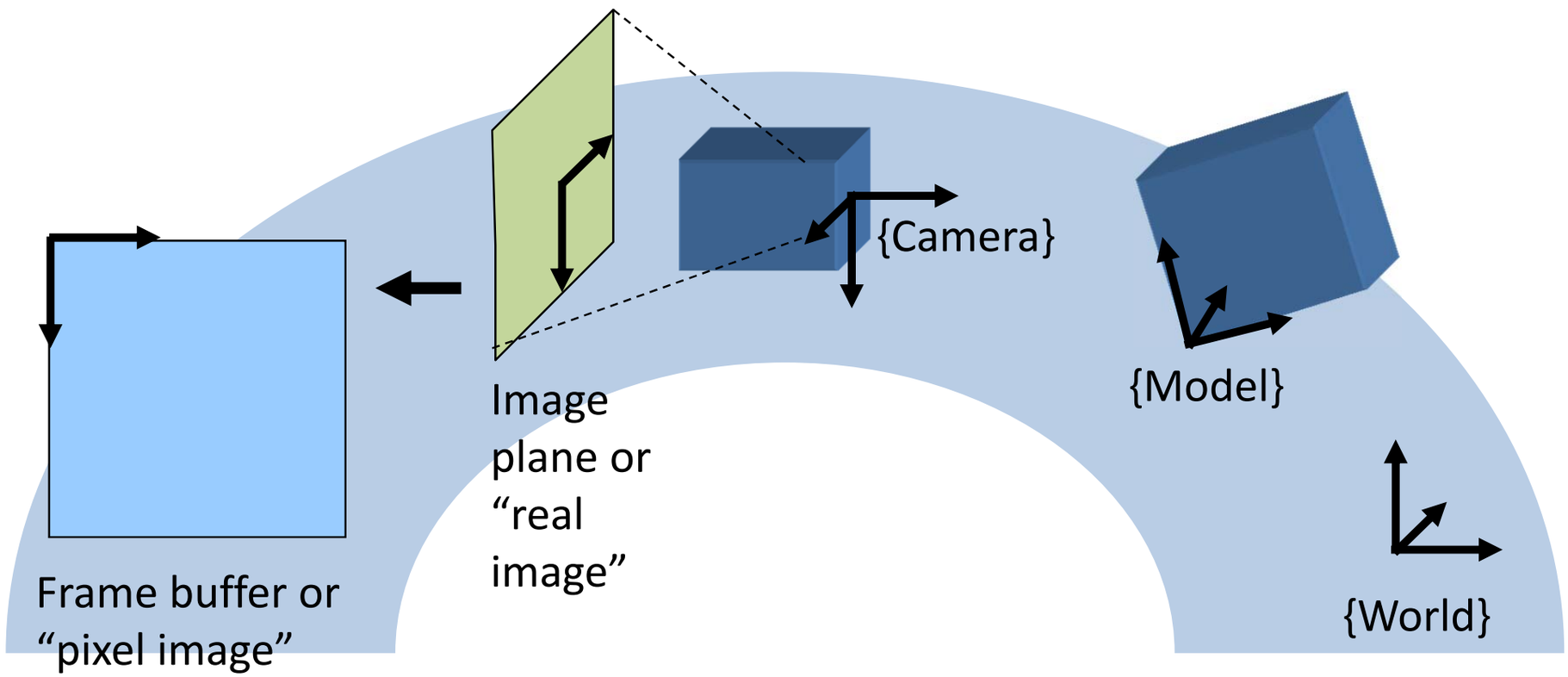
Camera Parameters

- Intrinsic parameters
  - Those parameters needed to relate an image point (in pixels) to a direction in the camera frame
  - f_x, f_y, c_x, c_y

- Extrinsic parameters
  - Define the position and orientation (pose) of the camera in the world

Frames of Reference

- Image frames are 2D; others are 3D
- The “pose” (position and orientation) of a 3D rigid body has 6 degrees of freedom



Transformations

«From» frame {B} «to» frame {A}

$$ {}^{A}\mathbf{P} = {}^{A}_{B}\mathbf{R}\ {}^{B}\mathbf{P} + \mathbf{t} $$

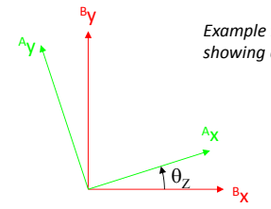
$$ {}^{A}\mathbf{P} = {}^{A}_{B}\mathbf{H}\ {}^{B}\mathbf{P} $$

Two ways to think about a transformation:

- **A transform mapping a point in the «from» frame {B} to its representation in the «to» frame {A}.**
- **A description of the «from» frame {B} relative to the «to» frame {A}**

Note on Direction of Rotation

- The rotation matrix ${}^{B}_{A}\mathbf{R}$ describes the orientation of the {A} frame with respect to the {B} frame
- Example:
  - Consider a rotation about the Z axis
  - We start with the {A} frame aligned with the {B} frame
  - Then rotate about Z until you get to the desired orientation



Example rotation about the Z axis, showing a rotation of +20 degrees

$$ \begin{pmatrix} {}^{B}x \\ {}^{B}y \\ {}^{B}z \end{pmatrix} = \begin{pmatrix} \cos 20° & -\sin 20° & 0 \\ \sin 20° & \cos 20° & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} {}^{A}x \\ {}^{A}y \\ {}^{A}z \end{pmatrix} $$

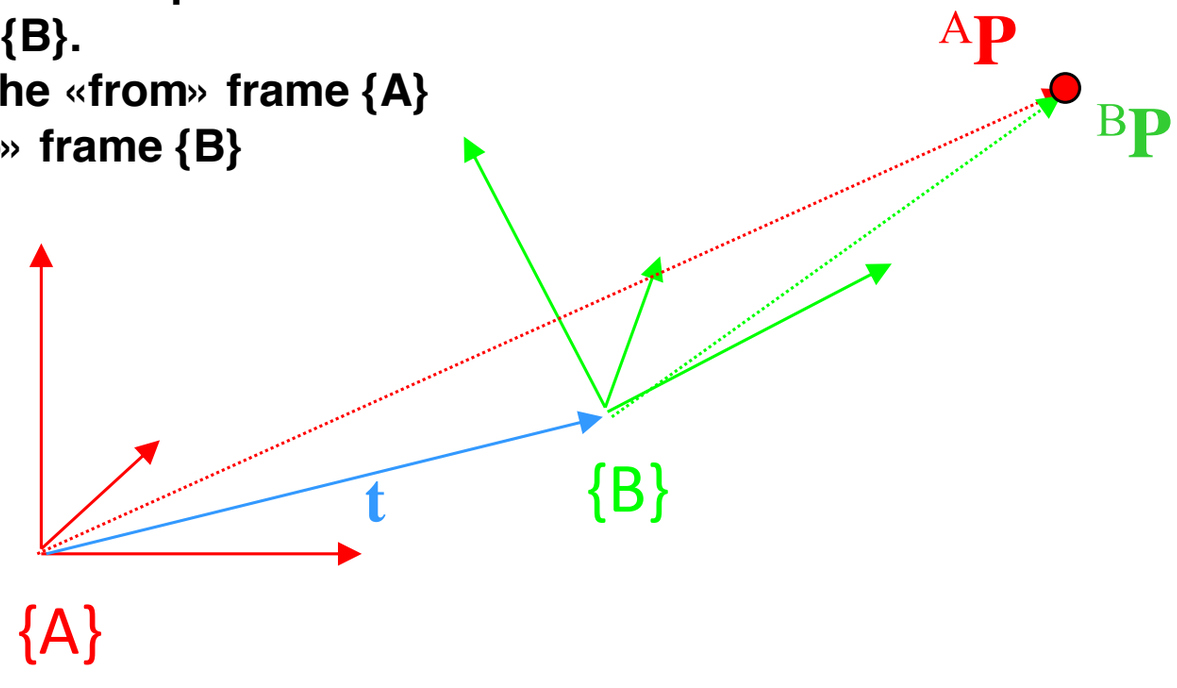
«From» frame {A} «to» frame {B}

$$ {}^{B}\mathbf{P} = {}^{B}_{A}\mathbf{R}\ {}^{A}\mathbf{P} + {}^{B}\mathbf{t}_{Aorg} $$

$$ {}^{B}\mathbf{P} = {}^{B}_{A}\mathbf{H}\ {}^{A}\mathbf{P} $$

Two ways to think about a transformation:

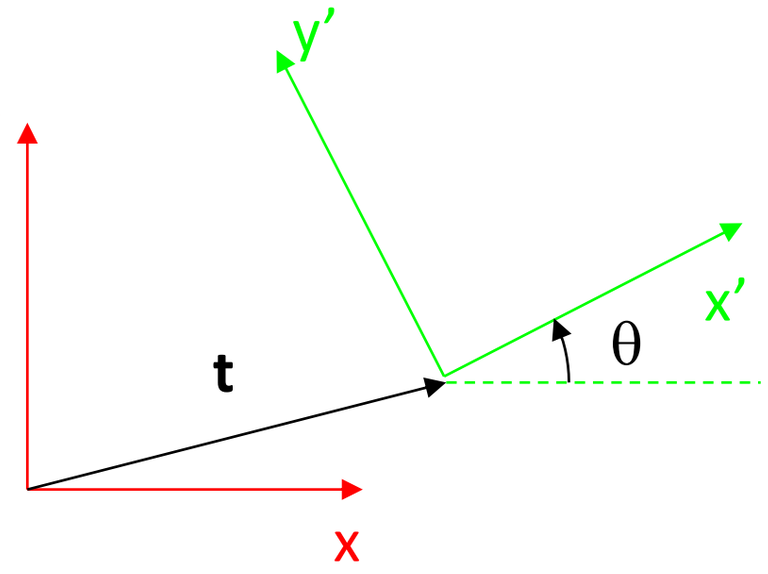
- **A transform mapping a point in the «from» frame {A} to its representation in the «to» frame {B}.**
- **A description of the «from» frame {A} relative to the «to» frame {B}**



2D-2D Coordinate Transforms

2D Rigid Frame Transformations

- The pose of one 2D frame with respect to another is described by
  - Translation vector **t**=$(\Delta x,\Delta y)^T$
  - Rotation angle θ
    - Rotation can also be represented as a 2x2 matrix **R**



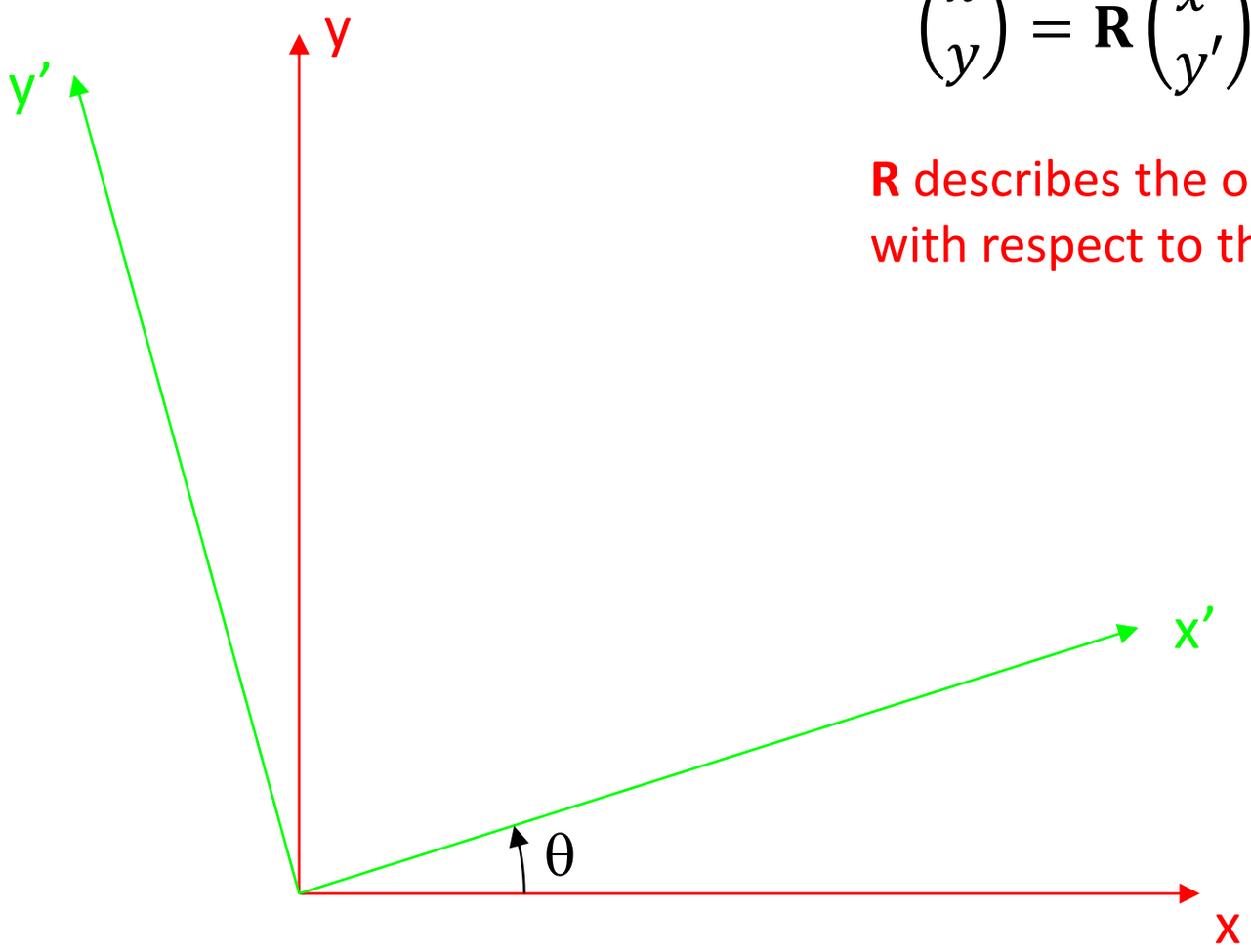
- Object shape and size is preserved

- Number of degrees of freedom for a 2D rigid transformation?

$${}^{A}\mathbf{P} = {}^{A}_{B}\mathbf{H}\,{}^{B}\mathbf{P}$$

B = ‘ (i.e. prime)

Rotations in 2D



$$\begin{pmatrix} x \\ y \end{pmatrix} = \mathbf{R} \begin{pmatrix} x' \\ y' \end{pmatrix} \qquad \mathbf{R} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

R describes the orientation of the “primed” frame (x’,y’) with respect to the “unprimed” frame (x,y)

- **R** is orthonormal
  - Rows, columns are orthogonal ($\mathbf{r}_1 \cdot \mathbf{r}_2 = 0$, $\mathbf{c}_1 \cdot \mathbf{c}_2 = 0$)
  - Transpose is the inverse; $\mathbf{R}\mathbf{R}^T = \mathbf{I}$
  - Determinant is $|\mathbf{R}| = 1$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathbf{R}^T \begin{pmatrix} x \\ y \end{pmatrix}$$

Homogeneous Coordinates

- Points can be represented using homogeneous coordinates
  - This simply means to append a 1 as an extra element
  - If the 3rd element becomes ≠ 1, we divide through by it

$$\tilde{\mathbf{x}} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} sx \\ sy \\ s \end{pmatrix}$$

- Effectively, vectors that differ only by scale are considered to be equivalent
- This simplifies transform equations; instead of

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathbf{R}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} \qquad \mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t} \qquad {}^{B}\mathbf{P} = {}^{B}_{A}\mathbf{R}\ {}^{A}\mathbf{P} + {}^{B}\mathbf{t}_{Aorg}$$

$${}^{B}\mathbf{P} = {}^{B}_{A}\mathbf{H}\ {}^{A}\mathbf{P}$$

- we have

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ 0\ \ 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \qquad \tilde{\mathbf{x}}' = \mathbf{H}\tilde{\mathbf{x}}$$

$\tilde{\mathbf{x}} \in P^2$, where $P^2 = R^3 - (0,0,0)$ is called a 2D projective space

Projective Transform (Homography)

- Most general type of linear 2D-2D transform
- H is an arbitrary 3x3 matrix

$$\tilde{\mathbf{x}}' = \mathbf{H}\,\tilde{\mathbf{x}}$$

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- We still need to divide by the 3rd element

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 / x_3 \\ x_2 / x_3 \\ 1 \end{pmatrix}$$

As we will see later, a homography maps points from the projection of one plane to the projection of another plane

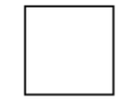
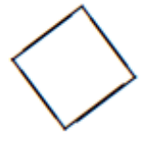
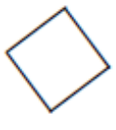
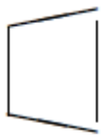
Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} \boldsymbol{I} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	2	orientation	
rigid (Euclidean)	$\begin{bmatrix} \boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	3	lengths	
similarity	$\begin{bmatrix} s\boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	4	angles	
affine	$\begin{bmatrix} \boldsymbol{A} \end{bmatrix}_{2\times3}$	6	parallelism	
projective	$\begin{bmatrix} \tilde{\boldsymbol{H}} \end{bmatrix}_{3\times3}$	8	straight lines	

Table 2.1 Hierarchy of 2D coordinate transformations. Each transformation also preserves the properties listed in the rows below it, i.e., similarity preserves not only angles but also parallelism and straight lines. The 2×3 matrices are extended with a third $[\mathbf{0}^T\ 1]$ row to form a full 3×3 matrix for homogeneous coordinate transformations.

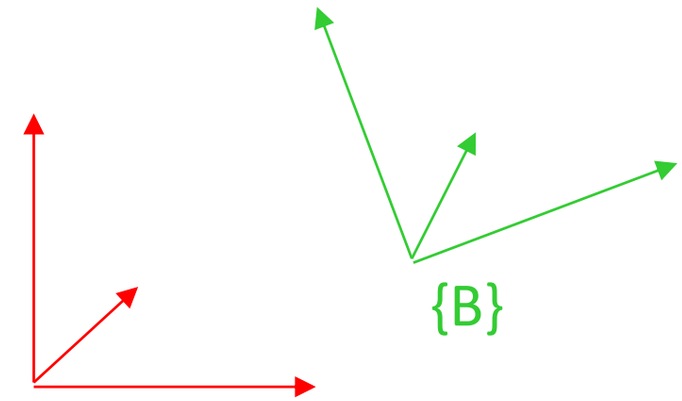
From Szeliski, Computer Vision: Algorithms and Applications

3D-3D Coordinate Transforms

An excellent reference is the book “Introduction to Robotics” by John Craig

3D Coordinate Systems

- Coordinate frames
  - Denote as {A}, {B}, etc
  - Examples: camera, world, model



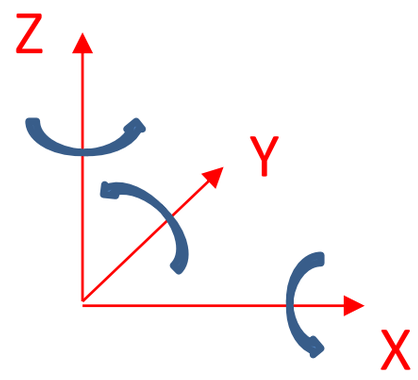
- The pose of {B} with respect to {A} is described by
  - Translation vector **t**
  - Rotation matrix **R**

- Rotation is a 3x3 matrix
  - It represents 3 angles

$$\mathbf{R} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$$

XYZ angles to represent rotations

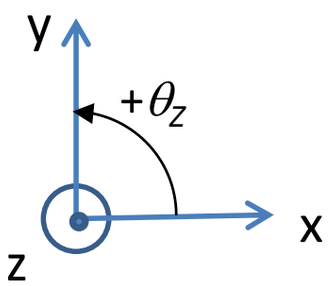
- One way to represent a 3D rotation is by doing successive rotations about the X,Y, and Z axes
- We'll present this first, because it is easy to understand



- However, it is not the best way for several reasons:
  - The result depends on the order in which the transforms are applied
  - Sometimes one or more angles change dramatically in response to a small change in orientation
  - Some orientations have singularities; i.e., the angles are not well defined

XYZ Angles

- Rotation about the Z axis

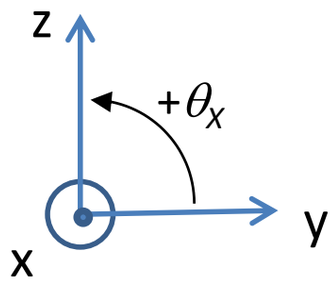


⊙ Points toward me

⊗ Points away from me

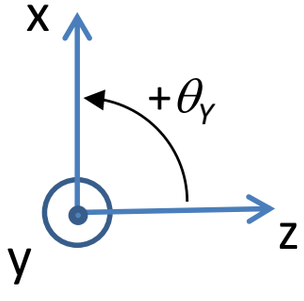
$$\begin{pmatrix} {}^B x \\ {}^B y \\ {}^B z \end{pmatrix} = \begin{pmatrix} \cos\theta_Z & -\sin\theta_Z & 0 \\ \sin\theta_Z & \cos\theta_Z & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} {}^A x \\ {}^A y \\ {}^A z \end{pmatrix}$$

- Rotation about the X axis



$$\begin{pmatrix} {}^B x \\ {}^B y \\ {}^B z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_X & -\sin\theta_X \\ 0 & \sin\theta_X & \cos\theta_X \end{pmatrix} \begin{pmatrix} {}^A x \\ {}^A y \\ {}^A z \end{pmatrix}$$

- Rotation about the Y axis



$$\begin{pmatrix} {}^B x \\ {}^B y \\ {}^B z \end{pmatrix} = \begin{pmatrix} \cos\theta_Y & 0 & \sin\theta_Y \\ 0 & 1 & 0 \\ -\sin\theta_Y & 0 & \cos\theta_Y \end{pmatrix} \begin{pmatrix} {}^A x \\ {}^A y \\ {}^A z \end{pmatrix}$$

Note signs are different than in the other cases

3D Rotation Matrix

- We can concatenate the 3 rotations to yield a single 3x3 rotation matrix; e.g.,

$$\mathbf{R} = \mathbf{R}_Z \mathbf{R}_Y \mathbf{R}_X$$

$$= \begin{pmatrix} \cos\theta_Z & -\sin\theta_Z & 0 \\ \sin\theta_Z & \cos\theta_Z & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_Y & 0 & \sin\theta_Y \\ 0 & 1 & 0 \\ -\sin\theta_Y & 0 & \cos\theta_Y \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_X & -\sin\theta_X \\ 0 & \sin\theta_X & \cos\theta_X \end{pmatrix}$$

- Note: we use the convention that to rotate a vector, we pre-multiply it; i.e., **v**' = **R v**
  - This means that if **R** = $\mathbf{R}_Z \mathbf{R}_Y \mathbf{R}_X$, we actually apply the X rotation first, then the Y rotation, then the Z rotation

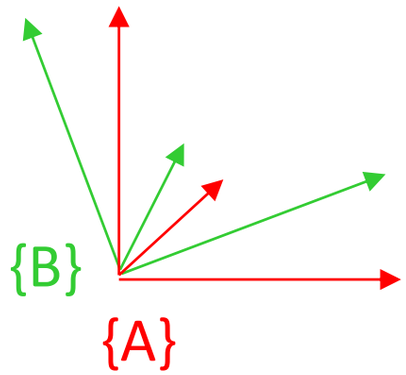
3D Rotation Matrix

- R represents a rotational transformation of frame A to frame B
  - I'll use the leading subscript to indicate "from"
  - I'll use the leading superscript to indicate "to"

$$ {}^{B}_{A}\mathbf{R} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} $$

- We can rotate a vector in frame A to obtain its representation in frame B

$$ {}^{B}\mathbf{v} = {}^{B}_{A}\mathbf{R} \; {}^{A}\mathbf{v} $$

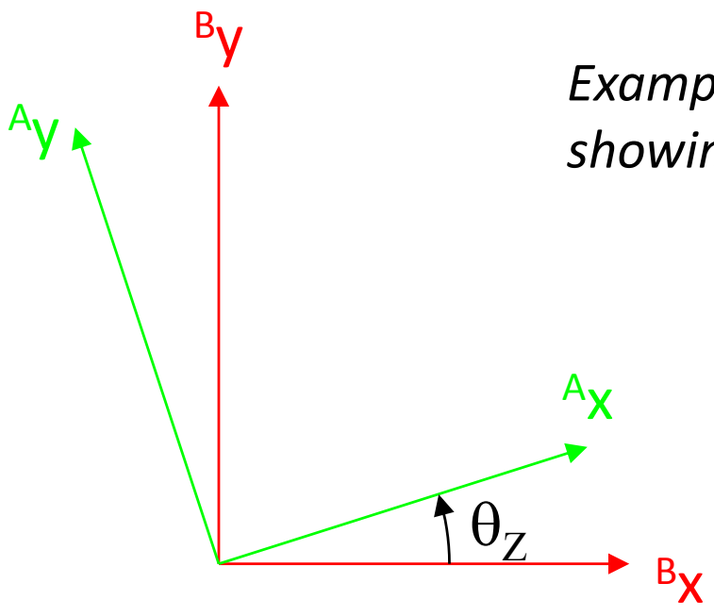


- Note: as in 2D, rotation matrices are orthonormal so the inverse of a rotation matrix is just its transpose

$$ \left({}^{B}_{A}\mathbf{R}\right)^{-1} = \left({}^{B}_{A}\mathbf{R}\right)^{T} = {}^{A}_{B}\mathbf{R} $$

Note on Direction of Rotation

- The rotation matrix ${}^{B}_{A}\mathbf{R}$ describes the orientation of the {A} frame with respect to the {B} frame
- Example:
  - Consider a rotation about the Z axis
  - We start with the {A} frame aligned with the {B} frame
  - Then rotate about Z until you get to the desired orientation



Example rotation about the Z axis, showing a rotation of +20 degrees

$$\begin{pmatrix} {}^{B}x \\ {}^{B}y \\ {}^{B}z \end{pmatrix} = \begin{pmatrix} \cos 20^\circ & -\sin 20^\circ & 0 \\ \sin 20^\circ & \cos 20^\circ & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} {}^{A}x \\ {}^{A}y \\ {}^{A}z \end{pmatrix}$$

3D Rotation Matrix

- The elements of **R** are direction cosines (the projections of unit vectors from one frame onto the unit vectors of the other frame)

$$ {}^{B}_{A}\mathbf{R} = \begin{pmatrix} \hat{\mathbf{x}}_A \cdot \hat{\mathbf{x}}_B & \hat{\mathbf{y}}_A \cdot \hat{\mathbf{x}}_B & \hat{\mathbf{z}}_A \cdot \hat{\mathbf{x}}_B \\ \hat{\mathbf{x}}_A \cdot \hat{\mathbf{y}}_B & \hat{\mathbf{y}}_A \cdot \hat{\mathbf{y}}_B & \hat{\mathbf{z}}_A \cdot \hat{\mathbf{y}}_B \\ \hat{\mathbf{x}}_A \cdot \hat{\mathbf{z}}_B & \hat{\mathbf{y}}_A \cdot \hat{\mathbf{z}}_B & \hat{\mathbf{z}}_A \cdot \hat{\mathbf{z}}_B \end{pmatrix} $$

- To see this, apply **R** to a unit vector

$$ {}^{B}_{A}R\,{}^{A}\hat{x}_A = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} r_{11} \\ r_{21} \\ r_{31} \end{pmatrix} = {}^{B}\hat{x}_A $$

- So the columns of **R** are the unit vectors of A, expressed in the B frame

$$ {}^{B}_{A}\mathbf{R} = \begin{pmatrix} {}^{B}\hat{\mathbf{x}}_A & {}^{B}\hat{\mathbf{y}}_A & {}^{B}\hat{\mathbf{z}}_A \end{pmatrix} $$

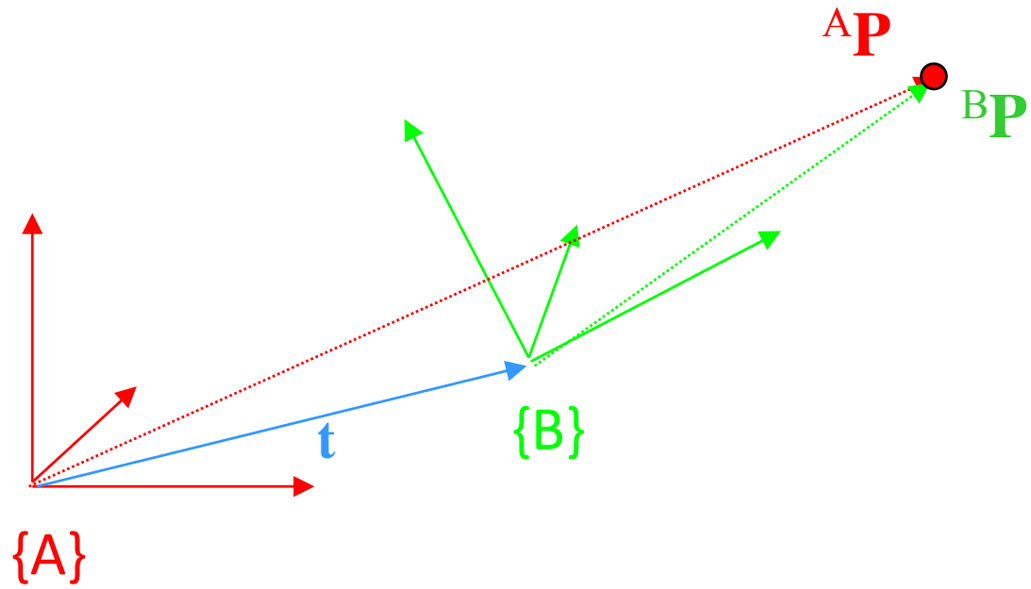
- And the rows of **R** are the unit vectors of {B} expressed in {A}

$$ {}^{B}_{A}\mathbf{R} = \begin{pmatrix} {}^{A}\hat{\mathbf{x}}_B{}^{T} \\ {}^{A}\hat{\mathbf{y}}_B{}^{T} \\ {}^{A}\hat{\mathbf{z}}_B{}^{T} \end{pmatrix} $$

Transforming a Point

- We can use **R,t** to transform a point from coordinate frame {B} to frame {A}

$$ {}^{A}\mathbf{P} = {}^{A}_{B}\mathbf{R}\ {}^{B}\mathbf{P} + \mathbf{t} $$



- Where
  - ${}^{A}\mathbf{P}$ is the representation of **P** in frame {A}
  - ${}^{B}\mathbf{P}$ is the representation of **P** in frame {B}

- Note

$\mathbf{t}$ is the translation of B's origin in the A frame, ${}^{A}\mathbf{t}_{Borg}$

Homogeneous Coordinates

- We can represent the transformation with a single matrix multiplication if we write **P** in homogeneous coordinates
  - This simply means to append a 1 as a 4th element
  - If the 4th element becomes ≠ 1, we divide through by it

The leading superscript indicates what coordinate frame the point is represented in

$$ {}^{A}\mathbf{P} = \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} sX \\ sY \\ sZ \\ s \end{pmatrix} $$

- Then

$$ {}^{B}\mathbf{P} = \mathbf{H}\ {}^{A}\mathbf{P}, \quad \text{where } \mathbf{H} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

General Rigid Transformation

- A general rigid transformation is a rotation followed by a translation

$$ {}^{B}\mathbf{P} = {}^{B}_{A}\mathbf{R}\ {}^{A}\mathbf{P} + {}^{B}\mathbf{t}_{Aorg} $$

- Can be represented by a single 4x4 homogeneous transformation matrix

$$ {}^{B}_{A}\mathbf{H} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & x_0 \\ r_{21} & r_{22} & r_{23} & y_0 \\ r_{31} & r_{32} & r_{33} & z_0 \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

- A note on notation:

$$ {}^{A}\mathbf{P} = {}^{A}_{B}\mathbf{H}\ {}^{B}\mathbf{P} $$

Cancel leading subscript with trailing superscript

$$ {}^{B}\mathbf{P} = {}^{B}_{A}\mathbf{H}\ {}^{A}\mathbf{P} = \begin{pmatrix} r_{11}x + r_{12}y + r_{13}z + x_0 \\ r_{21}x + r_{22}y + r_{23}z + y_0 \\ r_{31}x + r_{32}y + r_{33}z + z_0 \\ 1 \end{pmatrix} $$

Inverse Transformations

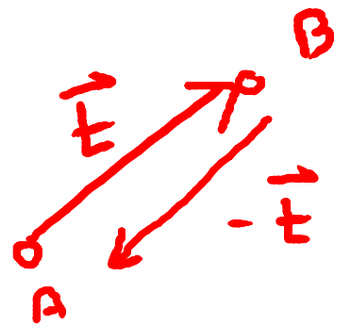
- The matrix inverse is the inverse transformation

$$ {}^{A}_{B}\mathbf{H} = \left({}^{B}_{A}\mathbf{H} \right)^{-1} $$

- Note – unlike rotation matrices, the inverse of a full 4x4 homogeneous transformation matrix is not the transpose

$$ {}^{A}_{B}\mathbf{H} \neq \left({}^{B}_{A}\mathbf{H} \right)^{T} $$

- What is the transformation inverse?



$$ {}^{A}_{B}H = \begin{bmatrix} {}^{A}_{B}R & {}^{A}t_{BORG} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} {}^{B}_{A}R^{T} & {}^{A}_{B}R\left(-{}^{B}t_{AORG}\right) \\ 0 & 1 \end{bmatrix} $$

Transformations

$$^{B}\mathbf{P} = {}^{B}_{A}\mathbf{R}\ {}^{A}\mathbf{P} + {}^{B}\mathbf{t}_{Aorg}$$

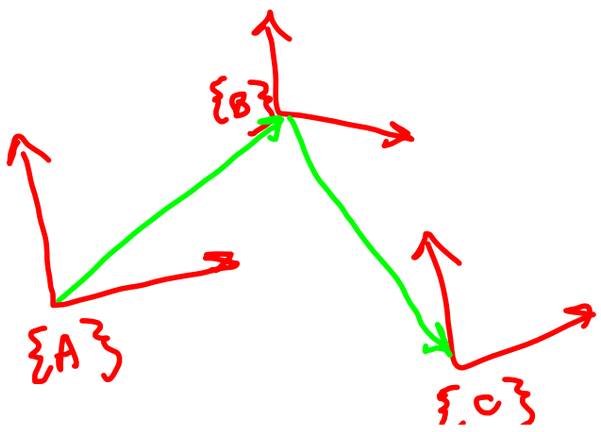
$$^{B}\mathbf{P} = {}^{B}_{A}\mathbf{H}\ {}^{A}\mathbf{P}$$

from {A} to {B}

- Can think of a transformation as:
  - A description of frame {A} relative to frame {B}
  - A transform mapping a point in the {A} frame to its representation in the {B} frame

- Can concatenate transformations together
  - Leading subscripts cancel trailing superscripts

$$^{C}_{A}\mathbf{H} = {}^{C}_{B}\mathbf{H}\ {}^{B}_{A}\mathbf{H} \qquad {}^{D}_{A}\mathbf{H} = {}^{D}_{C}\mathbf{H}\ {}^{C}_{B}\mathbf{H}\ {}^{B}_{A}\mathbf{H}, \quad \text{etc}$$



Possible Combinations for Rotations

- There are 12 possible combinations for rotations about the fixed axes:
  - $R_X R_Y R_Z$
  - $R_X R_Z R_Y$
  - $R_Y R_X R_Z$
  - $R_Y R_Z R_X$
  - $R_Z R_X R_Y$
  - $R_Z R_Y R_X$ We will use this convention in this course
  - $R_X R_Y R_X$
  - $R_X R_Z R_X$
  - $R_Y R_X R_Y$
  - $R_Y R_Z R_Y$
  - $R_Z R_X R_Z$
  - $R_Z R_Y R_Z$

- So for a given 3D rotation, the values of the 3 angles depends on the rotation convention you use
- However, a given 3D rotation always has a unique rotation matrix

Order of Rotations

- XYZ fixed angles
  - Start with {B} coincident with {A}. First rotate {B} about $\mathbf{x}_A$ by angle θ_X, then rotate it about $\mathbf{y}_A$ by θ_Y, then rotate about $\mathbf{z}_A$ by θ_Z.
  - Each rotation takes place relative to the fixed frame {A}
- The order matters
  - Matrices are multiplied in the order **R**z **R**y **R**x
  - **R**z **R**y **R**x order not same as **R**x **R**y **Rz**, etc

$$ {}^{A}_{B}R_{XYZ}(\theta_X,\theta_Y,\theta_Z) = R_Z(\theta_Z)\,R_Y(\theta_Y)\,R_X(\theta_X) $$

$$ = \begin{pmatrix} cz & -sz & 0 \\ sz & cz & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} cy & 0 & sy \\ 0 & 1 & 0 \\ -sy & 0 & cy \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & cx & -sx \\ 0 & sx & cx \end{pmatrix} $$

where

$$ cx = \cos(\theta_X),\ sy = \sin(\theta_Y),\ \text{etc} $$

$$ {}^{A}_{B}R_{XYZ}(\theta_X,\theta_Y,\theta_Z) = \begin{pmatrix} cz\,cy & cz\,sy\,sx - szcx & cz\,sy\,cx + sz\,sx \\ sz\,cy & sz\,sy\,sx + czcx & sz\,sy\,cx - cz\,sx \\ -sy & cy\,sx & cy\,cx \end{pmatrix} $$

$$ {}^{A}_{B}R_{XYZ}(\theta_X,\theta_Y,\theta_Z) \neq {}^{A}_{B}R_{ZYX}(\theta_X,\theta_Y,\theta_Z) $$

3D-2D Coordinate Transforms

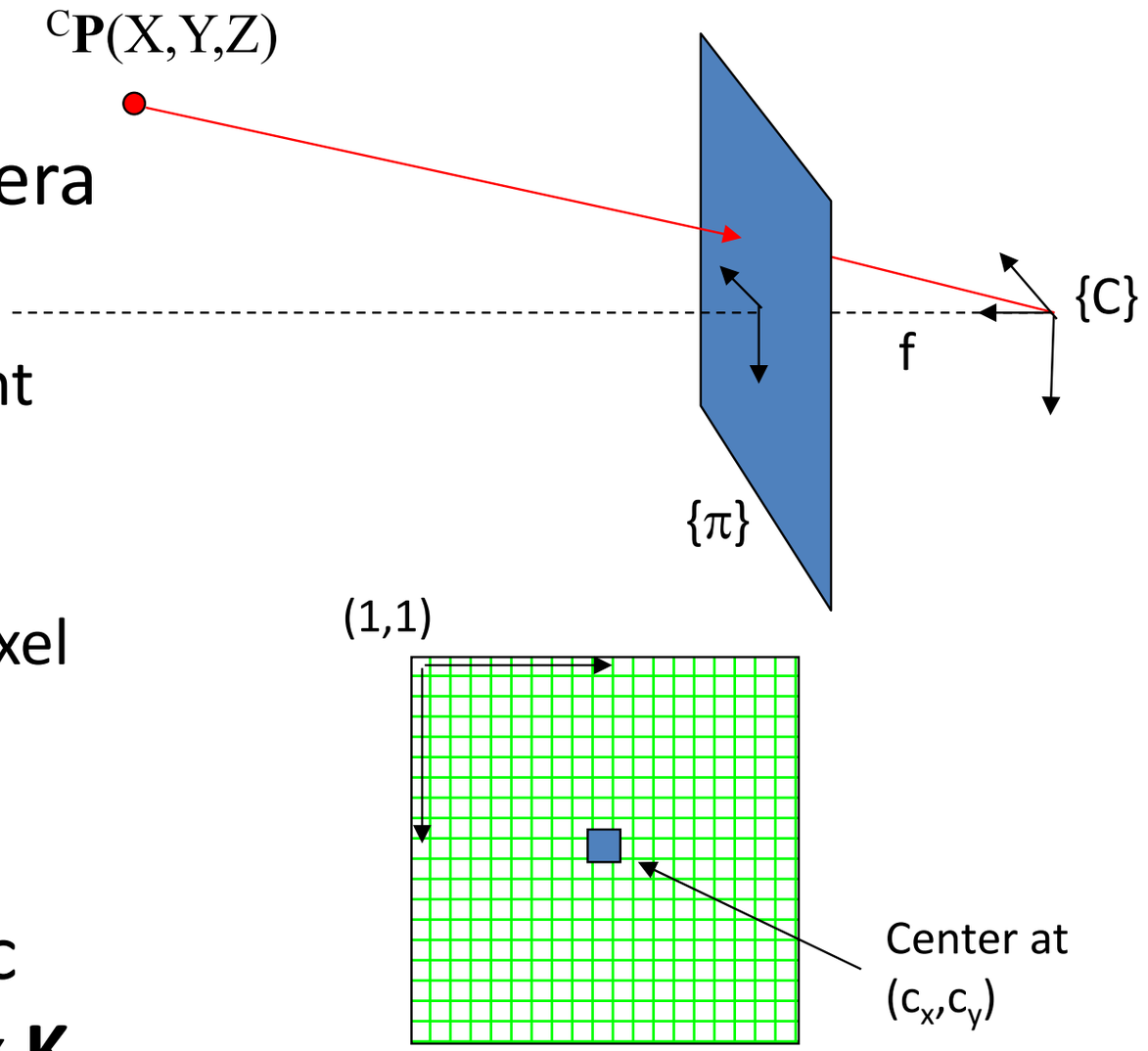
3D to 2D Projections

- We have already seen how to project 3D points onto a 2D image
  - We used the pinhole camera model
  - Also the geometry of similar triangles
- Now we will look at how to model this using matrix multiplication

- This will help us better understand and model:
  - Perspective projection
  - Other projection types, such as weak perspective projection
  - Special cases such as the projection of a planar surface

Intrinsic Camera Parameters

- Recall the intrinsic camera parameters, for a pinhole camera model
  - Focal length *f* and sensor element sizes *sx,sy*
    - Or, just focal lengths in pixels *fx,fy*
  - Optical center of the image at pixel location *cx, cy*



- We can capture all the intrinsic camera parameters in a matrix **K**

$$\mathbf{K} = \begin{pmatrix} f/s_x & 0 & c_x \\ 0 & f/s_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \quad \text{or} \quad \mathbf{K} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

3D to 2D Perspective Transformation

- We can project 3D points onto 2D with a matrix multiplication

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

- We treat the result as a 2D point in homogeneous coordinates. So we divide through by the last element.

$$\widetilde{\mathbf{x}} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} X/Z \\ Y/Z \\ 1 \end{pmatrix}$$

- Recall perspective projection is: $x = f\,X/Z + cx,\ y = f\,Y/Z + cy$
  - If f=1 and (cx,cy) = (0,0), then this is the image projection of the point
  - As we will see later, it is often useful to consider the projection of a point as if it were taken by a camera with f=1 and (cx,cy) = (0,0)
  - These are called “normalized image coordinates”

Complete Perspective Projection

- To project 3D points *represented in the coordinate system attached to the camera*, to the 2D image plane:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \mathbf{K} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} {}^{\mathrm{C}}\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}, \qquad \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 / x_3 \\ x_2 / x_3 \\ 1 \end{pmatrix} \qquad \mathbf{K} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

- To see this:

$$\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} {}^{\mathrm{C}}\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} f_x X + c_x Z \\ f_y Y + c_y Z \\ Z \end{pmatrix} \sim \begin{pmatrix} f_x X / Z + c_x \\ f_y Y / Z + c_y \\ 1 \end{pmatrix}$$

Extrinsic Camera Matrix

- If 3D points are in world coordinates, we first need to transform them to camera coordinates

$$ {}^{C}\mathbf{P} = {}^{C}_{W}\mathbf{H}\,{}^{W}\mathbf{P} = \begin{pmatrix} {}^{C}_{W}\mathbf{R} & {}^{C}\mathbf{t}_{Worg} \\ \mathbf{0} & 1 \end{pmatrix} {}^{W}\mathbf{P} $$

- We can write this as an extrinsic camera matrix, that does the rotation and translation, then a projection from 3D to 2D

$$ \mathbf{M}_{ext} = \begin{pmatrix} {}^{C}_{W}\mathbf{R} & {}^{C}\mathbf{t}_{Worg} \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_X \\ r_{21} & r_{22} & r_{23} & t_Y \\ r_{31} & r_{32} & r_{33} & t_Z \end{pmatrix} $$

Complete Perspective Projection

- Projection of a 3D point ${}^{W}\mathbf{P}$ in the world to a point in the pixel image (x_{im}, y_{im})

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \mathbf{K}\,\mathbf{M}_{ext} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}^{W}, \qquad x_{im} = x_1 / x_3, \;\; y_{im} = x_2 / x_3$$

  - where **K** is the intrinsic camera parameter matrix
  - and $\mathbf{M}_{ext}$ is the 3x4 matrix given by

$$\mathbf{M}_{ext} = \begin{pmatrix} {}^{C}_{W}\mathbf{R} & {}^{C}\mathbf{t}_{Worg} \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_X \\ r_{21} & r_{22} & r_{23} & t_Y \\ r_{31} & r_{32} & r_{33} & t_Z \end{pmatrix}$$

Summary: Transformations

«From» frame {B} «to» frame {A}

$$^{A}\mathbf{P} = {}^{A}_{B}\mathbf{R}\ {}^{B}\mathbf{P} + \mathbf{t}$$

$$^{A}\mathbf{P} = {}^{A}_{B}\mathbf{H}\ {}^{B}\mathbf{P}$$

Two ways to think about a transformation:

- **A transform mapping a point in the «from» frame {B} to its representation in the «to» frame {A}.**
- **A description of the «from» frame {B} relative to the «to» frame {A}**

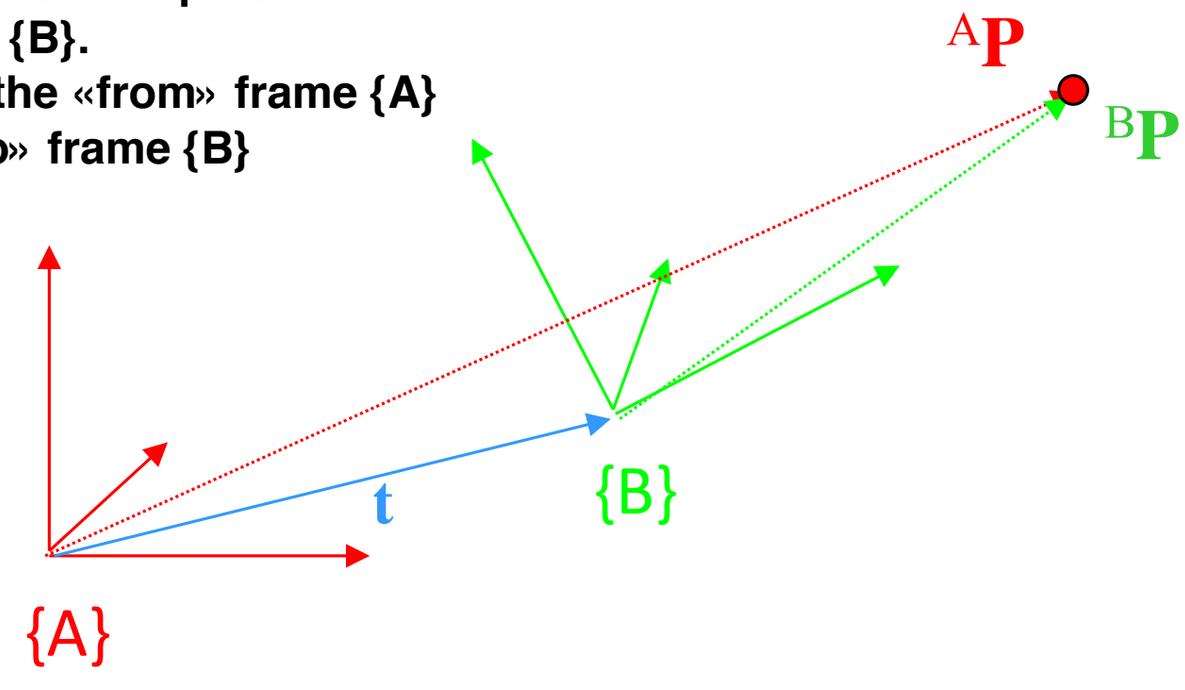
«From» frame {A} «to» frame {B}

$$^{B}\mathbf{P} = {}^{B}_{A}\mathbf{R}\ {}^{A}\mathbf{P} + {}^{B}\mathbf{t}_{Aorg}$$

$$^{B}\mathbf{P} = {}^{B}_{A}\mathbf{H}\ {}^{A}\mathbf{P}$$

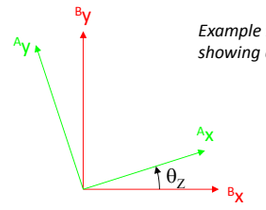
Two ways to think about a transformation:

- **A transform mapping a point in the «from» frame {A} to its representation in the «to» frame {B}.**
- **A description of the «from» frame {A} relative to the «to» frame {B}**



Note on Direction of Rotation

- The rotation matrix ${}^{B}_{A}\mathbf{R}$ describes the orientation of the {A} frame with respect to the {B} frame
- Example:
  - Consider a rotation about the Z axis
  - We start with the {A} frame aligned with the {B} frame
  - Then rotate about Z until you get to the desired orientation



Example rotation about the Z axis, showing a rotation of +20 degrees

$$\begin{pmatrix} {}^{B}x \\ {}^{B}y \\ {}^{B}z \end{pmatrix} = \begin{pmatrix} \cos 20° & -\sin 20° & 0 \\ \sin 20° & \cos 20° & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} {}^{A}x \\ {}^{A}y \\ {}^{A}z \end{pmatrix}$$

Alignment and Pose Estimation

- Alignment using Linear Least Squares
- Orientation, not very relevant for the exam:
  - Alignment using Nonlinear Least Squares
  - Nonlinear Pose Estimation
  - Linear Pose Estimation

Alignment using Linear Least Squares

Outline

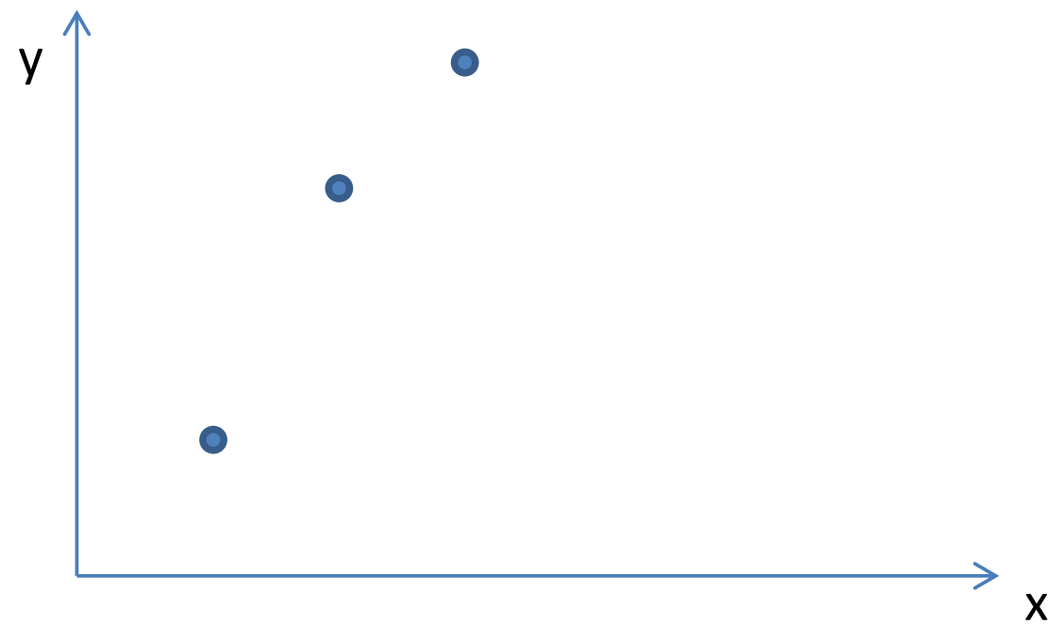
- First, a review of least squares fitting
- Finding an image transform using least squares
- The transform caused by a rotating camera
- Applying a transform to an input image to get an output image
  - Example: generating an orthophoto
  - Example: stitching two images together to make a “panorama”

Least Squares Fitting to a Line

- We have some measurement data (*xi,yi*)
- We want to fit the data to *y = f(x) = mx + b*
- We will find the parameters (*m,b*) that minimize the objective function

$$E = \sum_i |y_i - f(x_i)|^2$$

measured - predicted



Example
(x1,y1) = (1,1)
(x2,y2) = (2,3)
(x3,y3) = (3,4)

3 points -> fit line

Linear Least Squares

- In general
  - The input data can be vectors
  - The function can be a linear combination of the input data
- We write **A x** = **b**
  - The parameters to be fit are in the vector **x**
  - The input data is in **A,b**
- Example of a line
  - Parameter vector

$$\mathbf{x} = \begin{pmatrix} m \\ b \end{pmatrix}$$ **unknown parameters**

  - Linear equations

$$\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ x_3 & 1 \end{pmatrix} \begin{pmatrix} m \\ b \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

- So for a line

$$\mathbf{A} = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \\ x_N & 1 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}$$

known input data

Solving Linear Least Squares

- Want to minimize

$$E = \left|\mathbf{Ax} - \mathbf{b}\right|^2$$

0 if no noise, (ax-b) squared in scalar form

- Expanding we get

$$E = \mathbf{x}^T\left(\mathbf{A}^T\mathbf{A}\right)\mathbf{x} - 2\mathbf{x}^T\left(\mathbf{A}^T\mathbf{b}\right) + \left|\mathbf{b}\right|^2$$

- To find the minimum, take derivative wrt x and set to zero, getting

$$\left(\mathbf{A}^T\mathbf{A}\right)\mathbf{x} = \mathbf{A}^T\mathbf{b}$$

Called the "normal equations"

- To solve, can do

$$\mathbf{x} = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{b}$$

"pseudo inverse"

$$\mathbf{A}^+ = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T$$

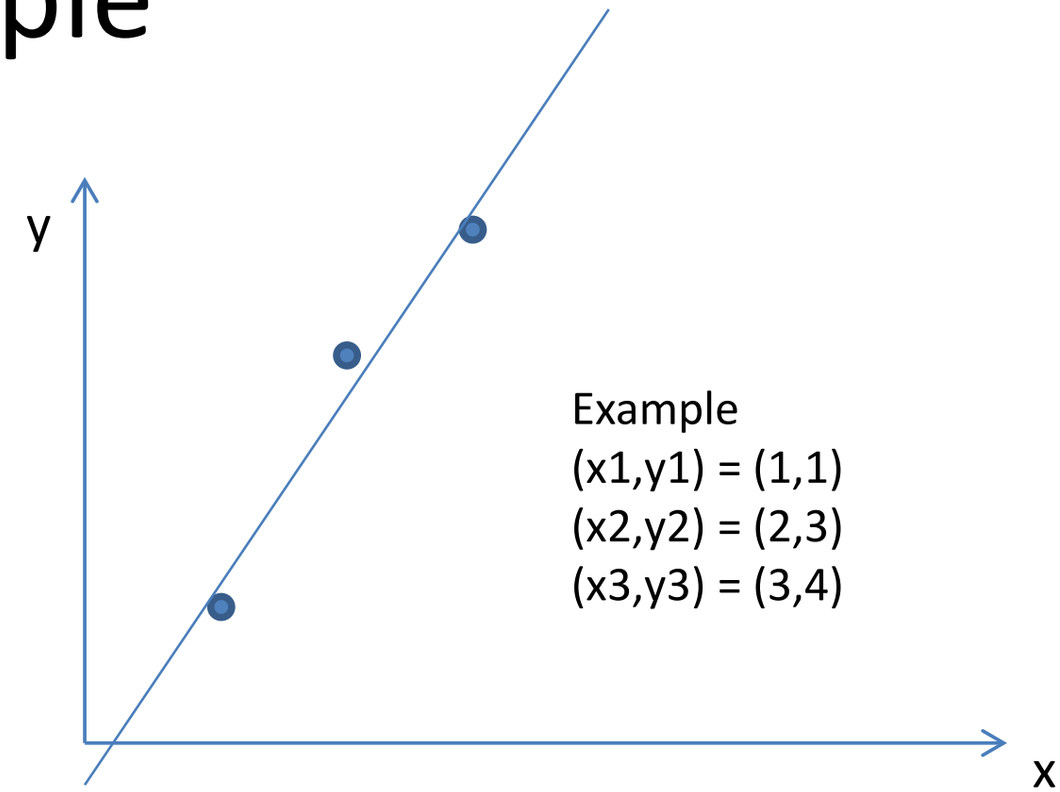
- In Matlab can do
  - x = pinv(A)*b;
  - or x = A\b;

- Note – it is preferable to solve the normal equations using Cholesky decomposition

Example

- The linear system for the line example earlier is **Ax=b**, where

$$\mathbf{A} = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ x_3 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix}$$



- Normal equations $\left(\mathbf{A}^T\mathbf{A}\right)\mathbf{x} = \mathbf{A}^T\mathbf{b}$

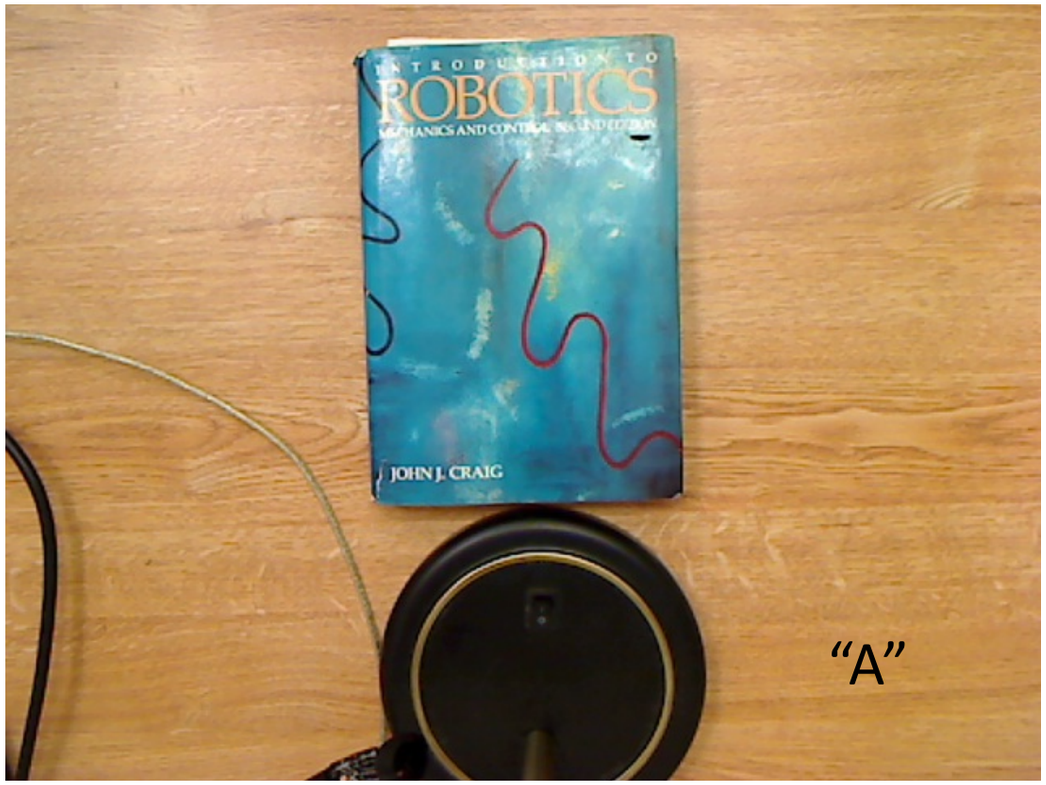
$$\mathbf{A}^T\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{pmatrix} = \begin{pmatrix} 14 & 6 \\ 6 & 3 \end{pmatrix}, \quad \mathbf{x} = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{b} = \begin{pmatrix} 1.5 \\ -0.333 \end{pmatrix}$$

- So the best fit line is $y = 1.5x - 0.333$

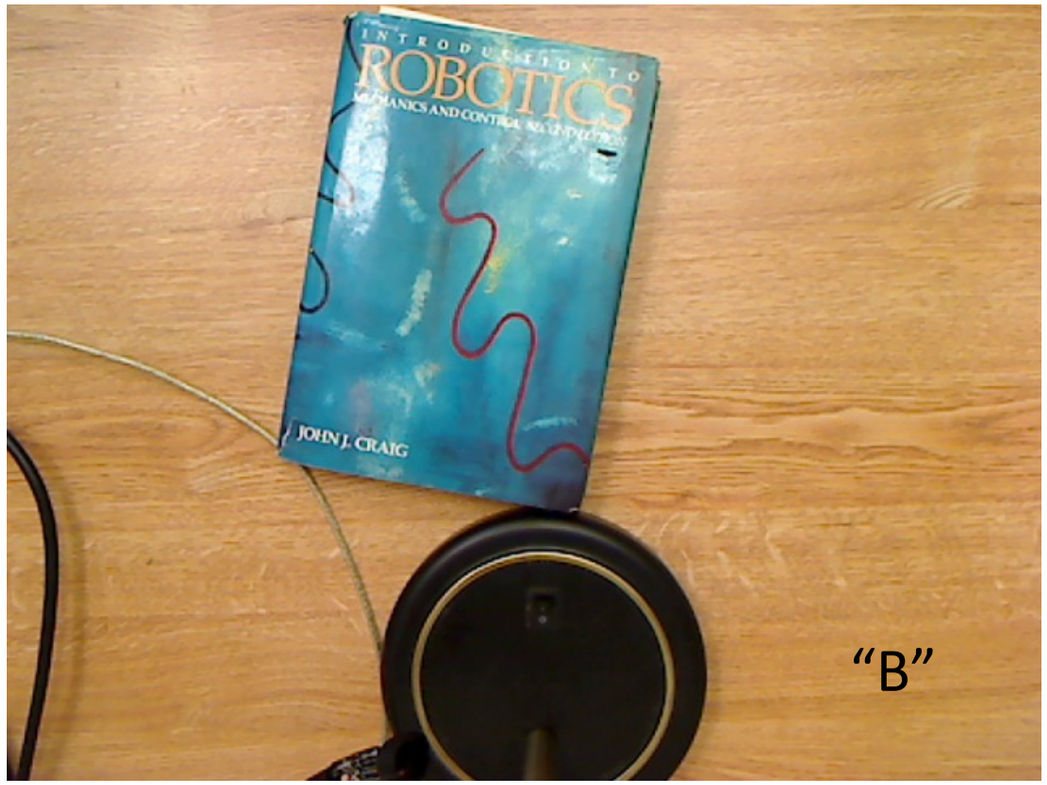
Finding an image transform

- If you know a set of point correspondences, you can estimate the parameters of the transform
- Example – find the rotation and translation of the book in the images below

```
% Using imtool, we manually find
% corresponding points (x;y), which are
% the four corners of the book
pA = [
    221 413  416  228;
    31   20  304  308];
pB = [
    214 404  352  169;
    7    34  314  280];
```



"A"



"B"

Example (continued)

- A 2D rigid transform is

$$\begin{pmatrix} x_B \\ y_B \\ 1 \end{pmatrix} = \begin{pmatrix} c & -s & t_x \\ s & c & t_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_A \\ y_A \\ 1 \end{pmatrix}, \quad \text{where } c = \cos\theta,\ s = \sin\theta$$

- Or

$$x_B = cx_A - sy_A + t_x$$
$$y_B = sx_A + cy_A + t_y$$

- We put into the form **Ax = b**, where

$$\mathbf{A} = \begin{pmatrix} x_A^{(1)} & -y_A^{(1)} & 1 & 0 \\ y_A^{(1)} & x_A^{(1)} & 0 & 1 \\ \hline \vdots & & & \\ y_A^{(N)} & x_A^{(N)} & 0 & 1 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} c \\ s \\ t_x \\ t_y \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} x_B^{(1)} \\ y_B^{(1)} \\ \hline \vdots \\ y_B^{(N)} \end{pmatrix}$$

Note: c and s are not really independent variables; however we treat them as independent so that we get a system of linear equations

Matlab

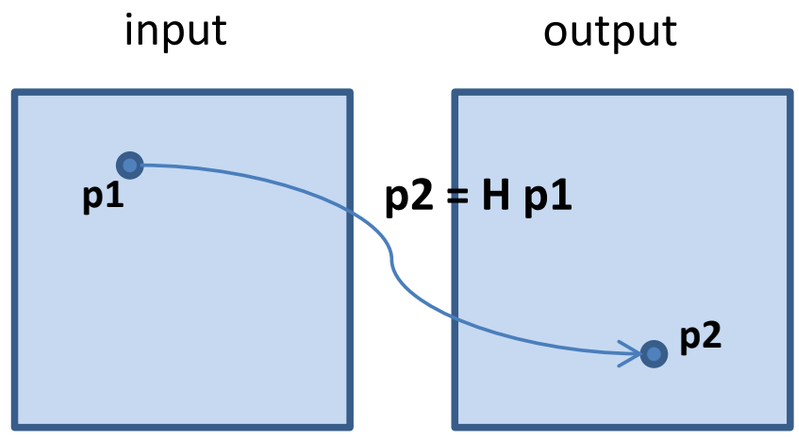
```
% Here are corresponding points (x;y)
pA = [
    221 413  416  228;
    31   20  304  308];
pB = [
    214 404  352  169;
    7    34  314  280];
N = size(pA,2);

A = zeros(2*N,4);
for i=1:N
    A( 2*(i-1)+1, :) = [ pA(1,i) -pA(2,i)  1  0];
    A( 2*(i-1)+2, :) = [ pA(2,i)  pA(1,i)  0  1];
end
b = reshape(pB, [], 1);

x = A\b;

theta = acos(x(1));
tx = x(3);
ty = x(4);
```

Note: you might get slightly different values of theta, from c and s. You could average them to get a better estimate.

Generating another image using a transform

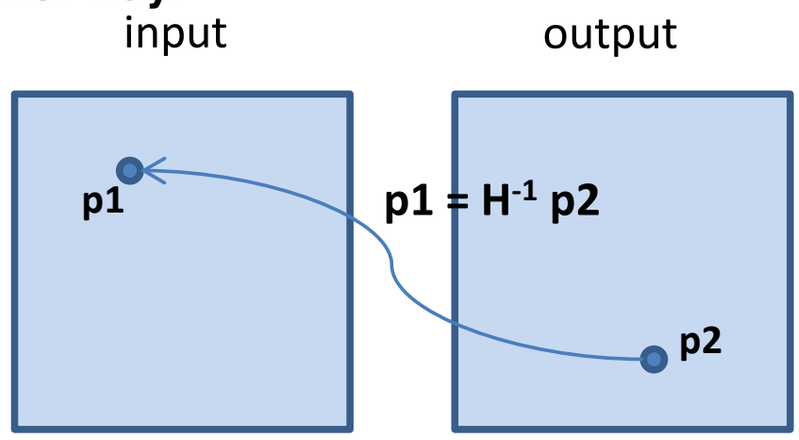
- We know the transformation from input image to output image, **p2 = H p1**
- But instead we use the inverse, **p1 = H^{-1} p2**
- Then we scan through every point **p2** in the output image, and calculate the point **p1** in the input image where we should get the intensity value to use
  - This makes sure that we don't miss assigning any pixels in the output image
  - If p1 falls at a non-integer location, we just take the value at the nearest integer point (a better way to do it is to interpolate among the neighbors)

One way:



for every input pixel -> not filled out

Another way:



for every output pixel

Matlab

Homography pre-calculated

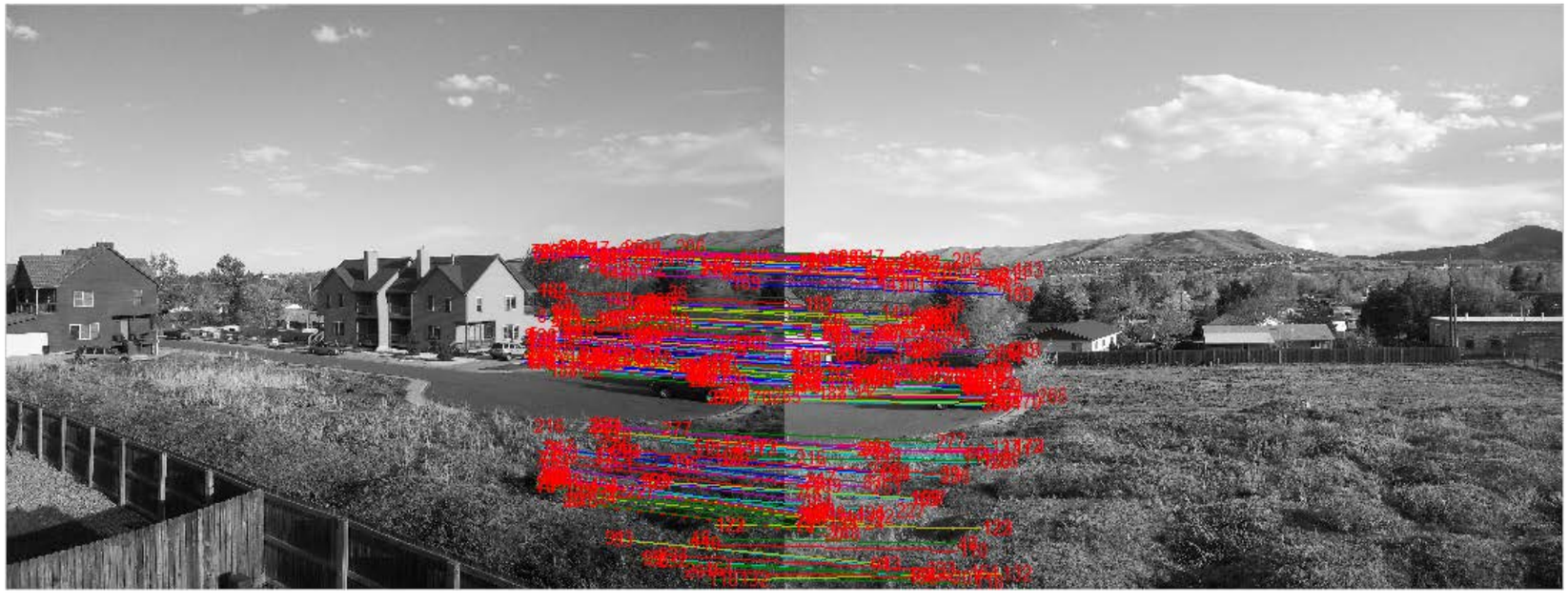
Homogeneous coord.

```
Hinv = inv(H);
for x2=1:size(I2,2)
    for y2=1:size(I2,1)
        p2 = [x2; y2; 1];
        p1 = Hinv * p2;
        p1 = p1/p1(3);

        % We'll just pick the nearest point to p1 (better way is to
        % interpolate).
        x1 = round(p1(1));
        y1 = round(p1(2));

        if x1>0 && x1<=size(I1,2) && y1>0 && y1<=size(I1,1)
            I2(y2,x2) = I1(y1,x1);
        end
    end
end
```

Example Application - Building Mosaics

- Assume we have two images of the same scene from the same position but different camera angles
- The mapping between the two image planes is a homography
- We find a set of corresponding points between the left and the right image
  - Since the homography matrix has 8 degrees of freedom, we need at least 4 corresponding point pairs
  - We solve for the homography matrix using least squares fitting
- We then apply the homography transform to one image, to map it into the plane of the other image

Matching points



Registered images

Note – blending should be done so that “seams” are not visible where the images are joined.

Orientation, not very relevant for the exam

Alignment using Nonlinear Least Squares

Recall 2D Rigid Transformation

- A 2D rigid transform (rotation + translation) is

$$\begin{pmatrix} x_B \\ y_B \\ 1 \end{pmatrix} = \begin{pmatrix} c & -s & t_x \\ s & c & t_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_A \\ y_A \\ 1 \end{pmatrix}, \quad \text{where } c = \cos\theta,\ s = \sin\theta$$

- Or

$$x_B = cx_A - sy_A + t_x$$
$$y_B = sx_A + cy_A + t_y$$

- We put into the form **Ax = b**, to get a system of linear equations

$$\mathbf{A} = \begin{pmatrix} x_A^{(1)} & -y_A^{(1)} & 1 & 0 \\ y_A^{(1)} & x_A^{(1)} & 0 & 1 \\ \vdots & & & \\ y_A^{(N)} & x_A^{(N)} & 0 & 1 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} c \\ s \\ t_x \\ t_y \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} x_B^{(1)} \\ y_B^{(1)} \\ \vdots \\ y_B^{(N)} \end{pmatrix}$$

Note: c and s are not really independent variables; however we treat them as independent so that we get a system of linear equations

Linear Least Squares

- Want to minimize

$$E = \left|\mathbf{A}\mathbf{x} - \mathbf{b}\right|^2$$

- Expanding we get

$$E = \mathbf{x}^T\left(\mathbf{A}^T\mathbf{A}\right)\mathbf{x} - 2\mathbf{x}^T\left(\mathbf{A}^T\mathbf{b}\right) + \left|\mathbf{b}\right|^2$$

- To find the minimum, take derivative wrt x and set to zero, getting

$$\left(\mathbf{A}^T\mathbf{A}\right)\mathbf{x} = \mathbf{A}^T\mathbf{b}$$

Called the "normal equations"

- To solve, can do

$$\mathbf{x} = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{b}$$

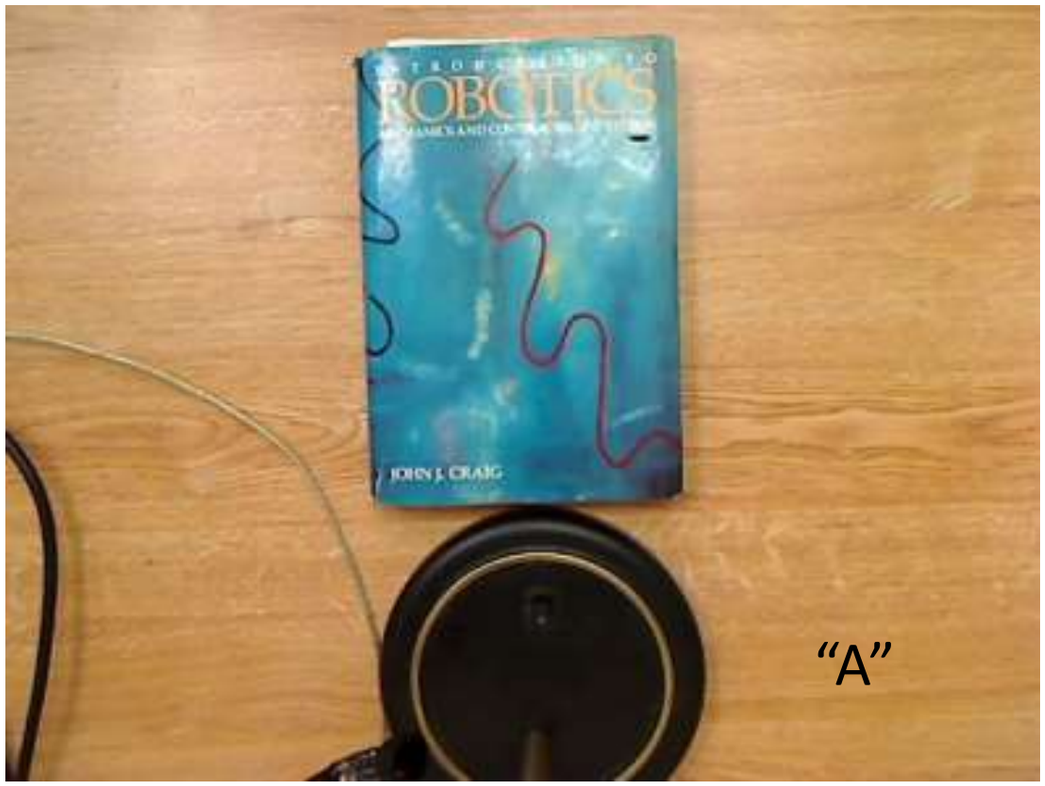
"pseudo inverse"

$$\mathbf{A}^+ = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T$$

- In Matlab can do
  - x = pinv(A)*b;
  - or x = A\b;

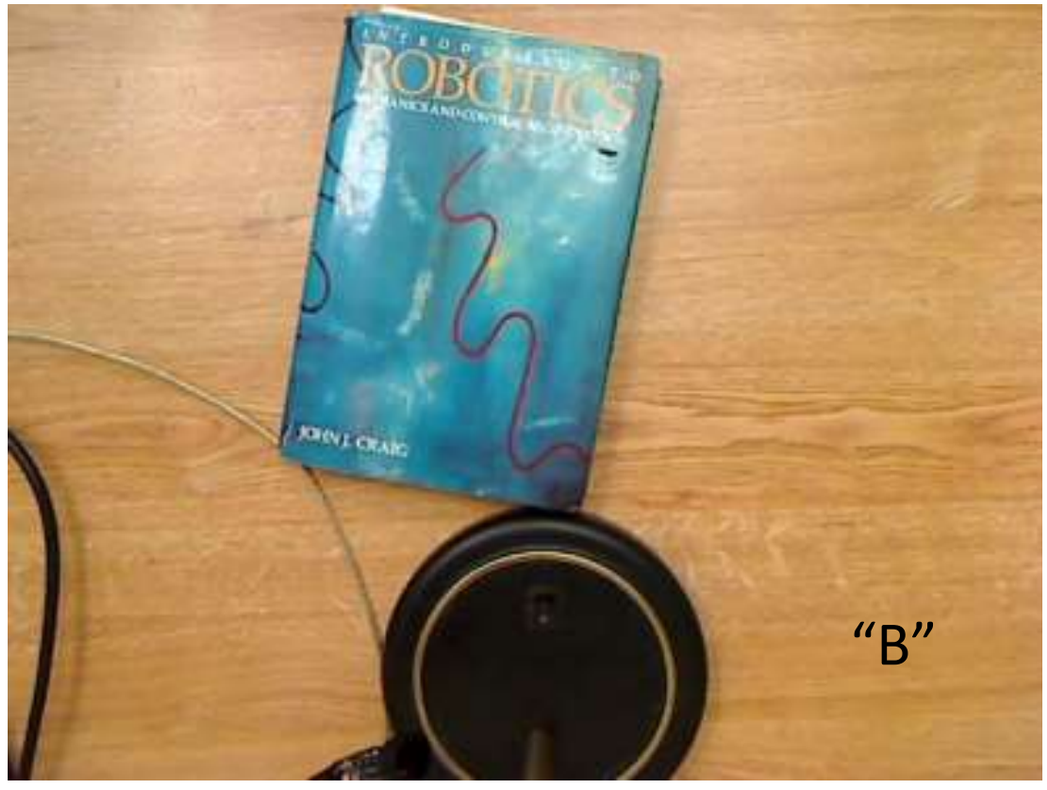
Example

```
% Using imtool, we find corresponding
% points (x;y), which are the four
% corners of the book
pA = [
    213 398  401  223;
    29   20  293  297];
pB = [
    207 391  339  164;
    7    34  302  270];
N = size(pA,2);
```



"A"

```
A = zeros(2*N,4);
for i=1:N
    A( 2*(i-1)+1, :) = [ pA(1,i) -pA(2,i)  1  0];
    A( 2*(i-1)+2, :) = [ pA(2,i)  pA(1,i)  0  1];
end
b = reshape(pB, [], 1);

x = A\b;

theta = acos(x(1));
tx = x(3);
ty = x(4);
```



"B"

System of Nonlinear Equations

- Since we treated $c = \cos\theta$ and $s = \sin\theta$ as independent variables, we got a system of linear equations

$$x_B = cx_A - sy_A + t_x$$
$$y_B = sx_A + cy_A + t_y$$

- But c,s are not independent – we should really just solve for 3 variables (t_x, t_y, θ), not 4 variables
- But this gives us a system of non linear equations

$$x_B = \cos\theta\, x_A - \sin\theta\, y_A + t_x$$
$$y_B = \sin\theta\, x_A + \cos\theta\, y_A + t_y$$

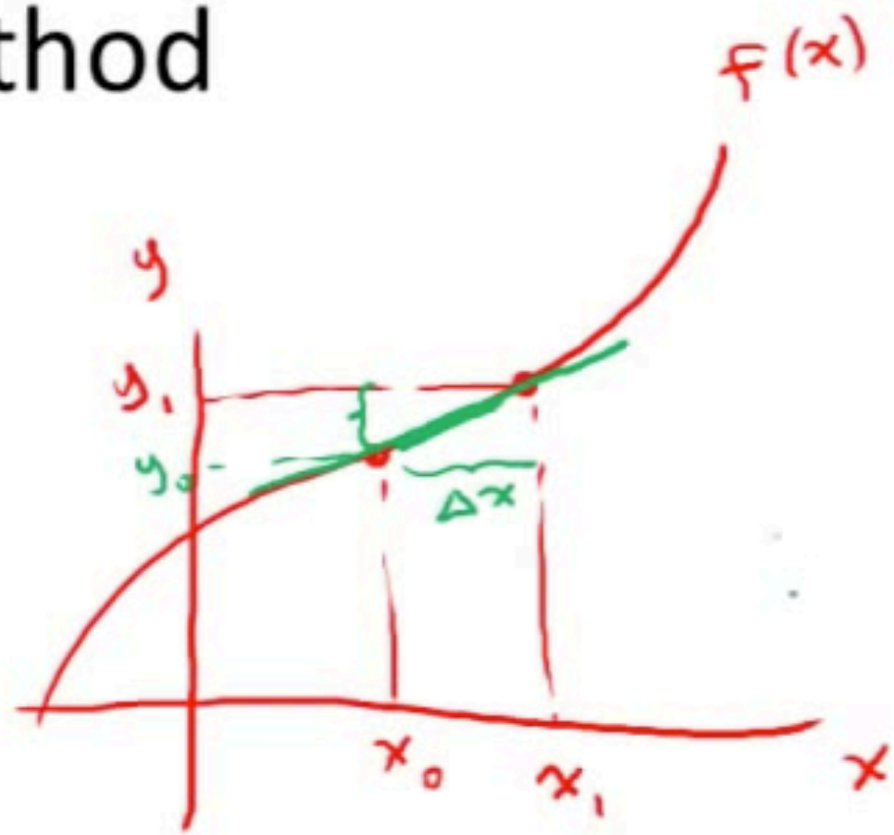
- We can still solve for the unknowns using least squares, but it requires an iterative algorithm

Newton's Method

- Example for a scalar function
  - Given
    - A known function $y = f(x)$
    - A value of y, call it y_1
  - Find x_1 such that $y_1 = f(x_1)$
    - We need a starting guess for x, call it x_0

Newton's Method

- Example for a scalar function
  - Given
    - A known function $y = f(x)$
    - A value of y, call it y_1
  - Find x_1 such that $y_1 = f(x_1)$
    - We need a starting guess for x, call it x_0



total derivative

$y = f(x)$ $\quad y_0 = f(x_0)$

$dy = \frac{df}{dx} dx$ $\quad \Delta y = y_1 - y_0$ ERROR, RESIDUAL

$\Delta y = \frac{df}{dx} \Delta x$ $\rightarrow$ $\Delta x = \frac{\Delta y}{df/dx}$

$x \leftarrow x + \Delta x$

REPEAT

Nonlinear Least Squares

- We have a nonlinear function **y** = **f**(**x**)
  - **x** is a vector of our unknowns
  - **y** is a vector of our observations
- We start with a guess for **x**, call it **x0**
- We linearize (take the Taylor series expansion) about that point
  dy = $[\partial \mathbf{f}/\partial \mathbf{x}]_{\mathbf{x0}}$ **dx**
- The matrix of partial derivatives of **f** with respect to **x** is called the Jacobian matrix

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_M \end{pmatrix}$$

$$\mathbf{J} = \left[\frac{\partial f_i}{\partial x_j}\right] = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \cdots & \partial f_1/\partial x_M \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \cdots & \partial f_2/\partial x_M \\ \vdots & \vdots & & \vdots \\ \partial f_N/\partial x_1 & \partial f_N/\partial x_2 & \cdots & \partial f_N/\partial x_M \end{pmatrix}$$

Nonlinear Iterative Least Squares Algorithm

We have

- **y0** = observations or measurements
- **x0** = a guess for **x**
- **y** = **f(x)** is a non linear function

1. Initialize **x** to **x0**
2. Compute **y** = **f(x)**. Residual error is **dy** = **y**-**y0**
3. Calculate Jacobian of **f**, evaluate it at **x**. We now have **dy** = **J dx**

   A x = b
4. Solve for **dx** using pseudo inverse $\mathbf{dx} = (\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T\mathbf{dy}$ $\mathbf{x} = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{b}$
5. Set **x** <= **x** + **dx**
6. Repeat steps 2-5 until convergence (no more change in **x**)

Example – 2D rigid transformation

- Recall
$$x_B = \cos\theta\, x_A - \sin\theta\, y_A + t_x$$
$$y_B = \sin\theta\, x_A + \cos\theta\, y_A + t_y$$

- We have N corresponding points

$$\left(x_B^{(i)}, y_B^{(i)}\right) \leftrightarrow \left(x_A^{(i)}, y_A^{(i)}\right), \quad i = 1 \ldots N$$

- Let **y** = **f(x)**, where

$$\mathbf{y} = \begin{pmatrix} x_B^{(1)} \\ y_B^{(1)} \\ \hline x_B^{(2)} \\ y_B^{(2)} \\ \vdots \\ x_B^{(N)} \\ y_B^{(N)} \end{pmatrix}_{2Nx1} \qquad \mathbf{x} = \begin{pmatrix} \theta \\ t_x \\ t_y \end{pmatrix}_{3x1}, \quad \mathbf{f}(\mathbf{x}) = \begin{pmatrix} \cos\theta\, x_A^{(1)} - \sin\theta\, y_A^{(1)} + t_x \\ \sin\theta\, x_A^{(1)} + \cos\theta\, y_A^{(1)} + t_y \\ \hline \cos\theta\, x_A^{(2)} - \sin\theta\, y_A^{(2)} + t_x \\ \sin\theta\, x_A^{(2)} + \cos\theta\, y_A^{(2)} + t_y \\ \vdots \\ \cos\theta\, x_A^{(N)} - \sin\theta\, y_A^{(N)} + t_x \\ \sin\theta\, x_A^{(N)} + \cos\theta\, y_A^{(N)} + t_y \end{pmatrix}_{2Nx1}$$

Example

- Jacobian?

$$\mathbf{J} = \left[\frac{\partial f_i}{\partial x_j}\right] = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \partial f_1/\partial x_3 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \partial f_2/\partial x_3 \\ \vdots & \vdots & \vdots \\ \partial f_{2N}/\partial x_1 & \partial f_{2N}/\partial x_2 & \partial f_{2N}/\partial x_3 \end{pmatrix}_{2Nx3} \qquad \mathbf{x} = \begin{pmatrix} \theta \\ t_x \\ t_y \end{pmatrix}_{3x1}, \quad \mathbf{f}(\mathbf{x}) = \begin{pmatrix} \cos\theta\ x_A^{(1)} - \sin\theta\ y_A^{(1)} + t_x \\ \sin\theta\ x_A^{(1)} + \cos\theta\ y_A^{(1)} + t_y \\ \cos\theta\ x_A^{(2)} - \sin\theta\ y_A^{(2)} + t_x \\ \sin\theta\ x_A^{(2)} + \cos\theta\ y_A^{(2)} + t_y \\ \vdots \\ \cos\theta\ x_A^{(N)} - \sin\theta\ y_A^{(N)} + t_x \\ \sin\theta\ x_A^{(N)} + \cos\theta\ y_A^{(N)} + t_y \end{pmatrix}_{2Nx1}$$

Example

- Jacobian?

$$\mathbf{J}=\left[\frac{\partial f_i}{\partial x_j}\right]=\begin{pmatrix}\partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \partial f_1/\partial x_3\\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \partial f_2/\partial x_3\\ \vdots & \vdots & \vdots\\ \partial f_{2N}/\partial x_1 & \partial f_{2N}/\partial x_2 & \partial f_{2N}/\partial x_3\end{pmatrix}_{2Nx3} \qquad \mathbf{x}=\begin{pmatrix}\theta\\ t_x\\ t_y\end{pmatrix}_{3x1}, \quad \mathbf{f}(\mathbf{x})=\begin{pmatrix}\cos\theta\, x_A^{(1)}-\sin\theta\, y_A^{(1)}+t_x\\ \sin\theta\, x_A^{(1)}+\cos\theta\, y_A^{(1)}+t_y\\ \cos\theta\, x_A^{(2)}-\sin\theta\, y_A^{(2)}+t_x\\ \sin\theta\, x_A^{(2)}+\cos\theta\, y_A^{(2)}+t_y\\ \vdots\\ \cos\theta\, x_A^{(N)}-\sin\theta\, y_A^{(N)}+t_x\\ \sin\theta\, x_A^{(N)}+\cos\theta\, y_A^{(N)}+t_y\end{pmatrix}_{2Nx1}$$

$$\frac{\partial f_1}{\partial x_1}=\frac{\partial}{\partial\theta}\left[\cos\theta\, x_A^{(1)}-\sin\theta\, y_A^{(1)}+t_x\right]=-\sin\theta\, x_A^{(1)}-\cos\theta\, y_A^{(1)}$$

$$J=\begin{bmatrix}-\sin\theta\, x_A^{(1)}-\cos\theta\, y_A^{(1)} & 1 & 0\\ \cos\theta\, x_A^{(1)}-\sin\theta\, y_A^{(1)} & 0 & 1\\ \vdots & & \\ & & \end{bmatrix}_{2N\times 3}$$

Algorithm for 2D Rigid Transform

We have

- **y0** = observations or measurements
- **x0** = a guess for **x**
- **y** = **f(x)** is a non linear function

1. Initialize **x** to **x0**
2. Compute **y** = **f(x)**. Residual error is **dy** = **y**-**y0**
3. Calculate Jacobian of **f**, evaluate it at **x**. We now have **dy** = **J dx**
4. Solve for **dx** using pseudo inverse **dx** = $\mathbf{(J^T J)^{-1} J^T}$ **dy**
5. Set **x** <= **x** + **dx**
6. Repeat steps 2-5 until convergence (no more change in **x**)

```
clear all
close all

IA = imread('book_A.jpg');
IB = imread('book_B.jpg');
figure, imshow(IA,[]);
figure, imshow(IB,[]);

% Using imtool, we find corresponding
% points (x;y), which are the four
% corners of the book
pA = [
    213 398  401  223;
    29   20  293  297;
    1    1   1    1];
pB = [
    207 391  339  164;
    7    34  302  270];
N = size(pA,2);

theta = 0;
tx = 0;
ty = 0;
x = [theta; tx; ty];    % initial guess
```

Algorithm for 2D Rigid Transform

1. Initialize **x** to **x0**
2. Compute **y** = **f(x)**. Residual error is **dy** = **y**-**y0**
3. Calculate Jacobian of **f**, evaluate it at **x**. We now have **dy** = **J dx**
4. Solve for **dx** using pseudo inverse $\mathbf{dx} = (\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T\,\mathbf{dy}$
5. Set **x** <= **x** + **dx**
6. Repeat steps 2-5 until convergence (no more change in **x**)

```
while true
    disp('Parameters (theta; tx; ty):'), disp(x);

    y = fRigid(x, pA); % Call function to compute expected measurements

    dy = reshape(pB,[],1) - y;       % new residual

    J = zeros(2*N,3);
    % Fill in values of J …

    dx = pinv(J)*dy;

    % Stop if parameters are no longer changing
    if abs( norm(dx)/norm(x) ) < 1e-6
        break;
    end

    x = x + dx;     % add correction
end
```

Algorithm for 2D Rigid Transform

```
while true
    disp('Parameters (theta; tx; ty):'), disp(x);

    y = fRigid(x, pA); % Call function to compute expected measurements

    dy = reshape(pB,[],1) - y;       % new residual

    J = zeros(2*N,3);
    % Fill in values of J …
```

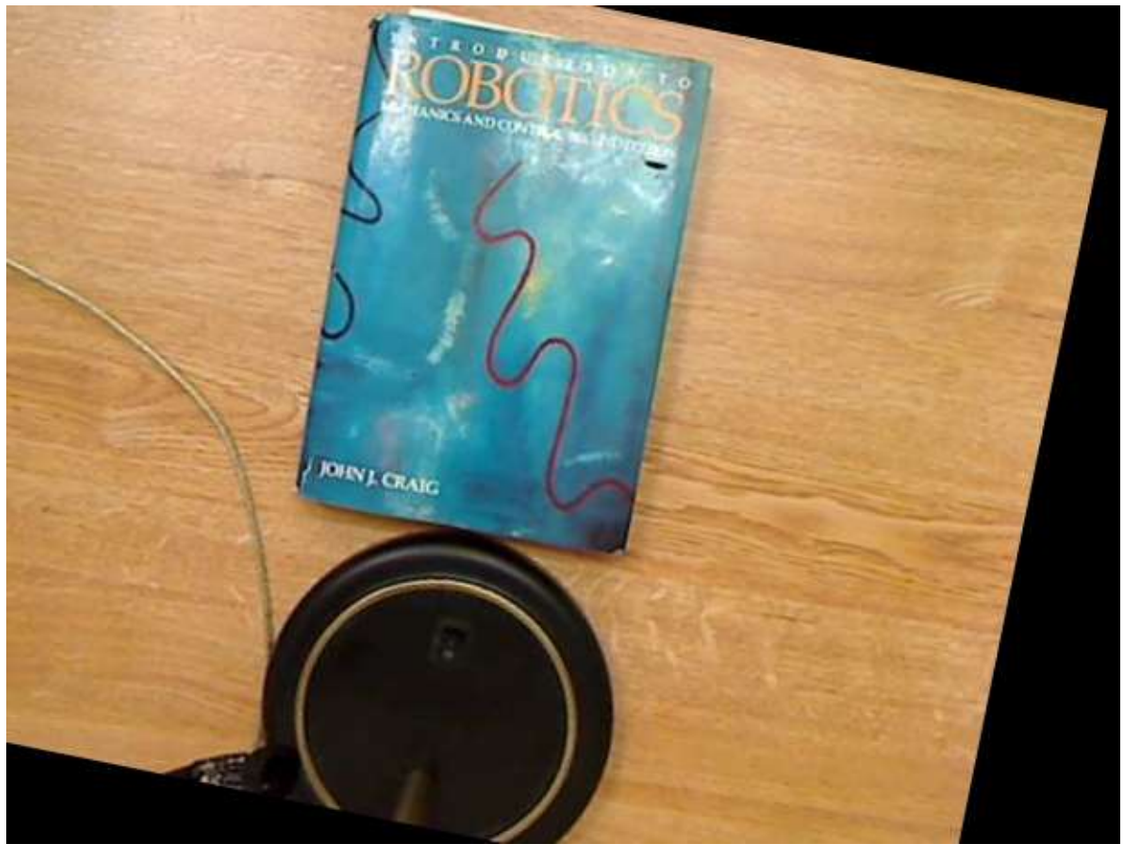
```
    dx = pinv(J)*dy;

    % Stop if parameters are no longer changing
    if abs( norm(dx)/norm(x) ) < 1e-6
        break;
    end

    x = x + dx;     % add correction
end
```

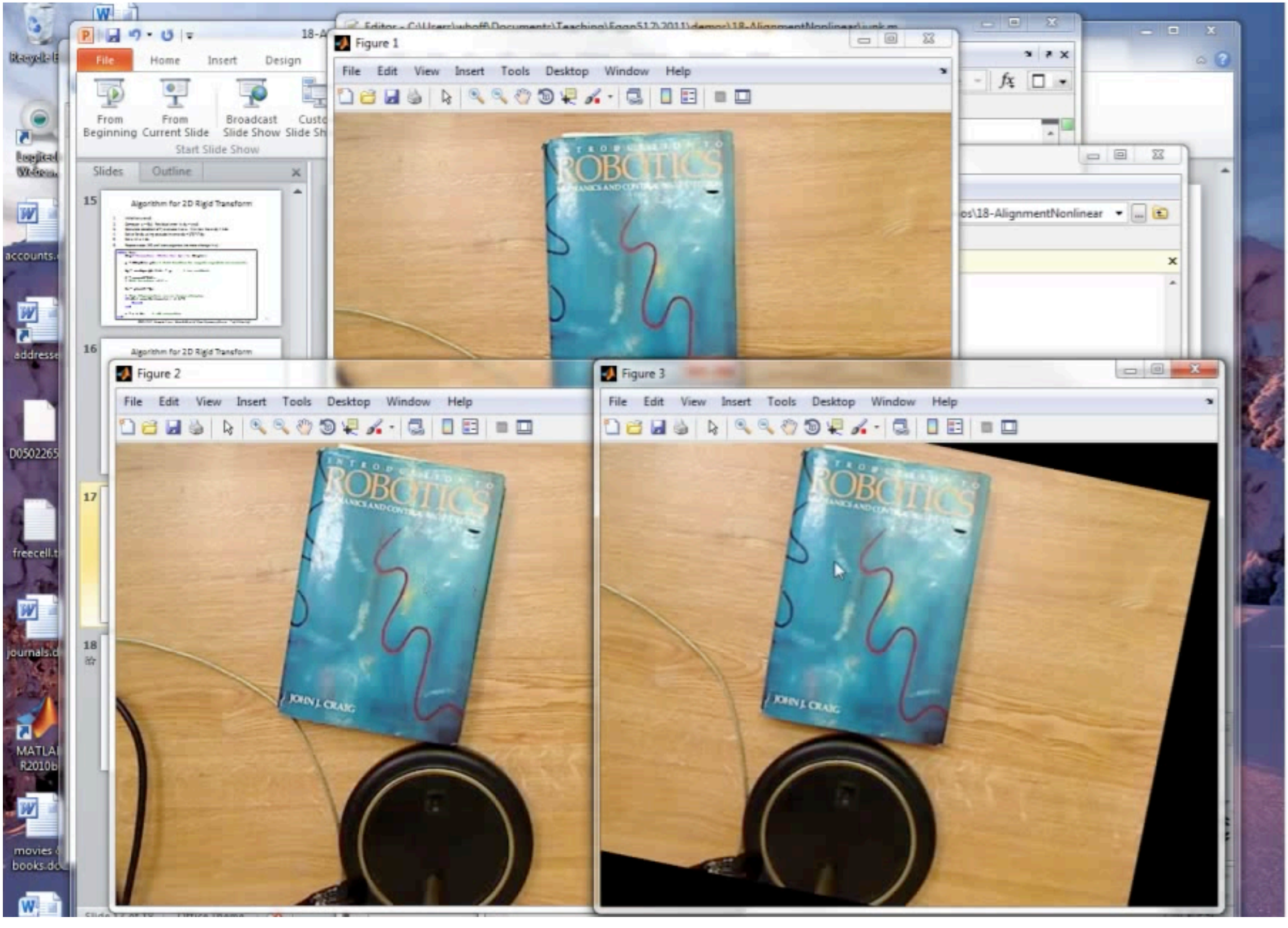
Algorithm for 2D Rigid Transform

```
while true
    disp('Parameters (theta; tx; ty):'), disp(x);

    y = fRigid(x, pA); % Call function to compute expected measurements

    dy = reshape(pB,[],1) - y;      % new residual

    J = zeros(2*N,3);
    % Fill in values of J …
    theta = x(1);
    for i=1:N
        J( 2*(i-1)+1, :) = [ -sin(theta)*pA(1,i)-cos(theta)*pA(2,i)  1   0 ];
        J( 2*(i-1)+2, :) = [  cos(theta)*pA(1,i)-sin(theta)*pA(2,i)  0   1 ];
    end
    dx = pinv(J)*dy;

    % Stop if parameters are no longer changing
    if abs( norm(dx)/norm(x) ) < 1e-6
        break;
    end

    x = x + dx;     % add correction
end
```

Apply final transform to image

```
% Apply transform to image
theta = x(1);
tx = x(2);
ty = x(3);

A = [cos(theta)  -sin(theta) tx;
     sin(theta)  cos(theta) ty;
     0 0 1];
T = maketform('affine', A');

I2 = imtransform(IA,T, ...
    'XData', [1 size(IA,2)], ...
    'YData', [1 size(IA,1)]  );
figure, imshow(I2,[]);
```




File Home Insert Design
From Beginning
From Current Slide
Broadcast Slide Show
Start Slide Show
Slides
Outline
15
Algorithm for 2D Rigid Transform
16
Algorithm for 2D Rigid Transform
17
18
Figure 1
File Edit View Insert Tools Desktop Window Help
ROBOTICS
Figure 2
File Edit View Insert Tools Desktop Window Help
JOHN J. CRAIG
Figure 3
File Edit View Insert Tools Desktop Window Help
JOHN J. CRAIG

Computing Jacobian Numerically

- An alternative way to compute the Jacobian matrix
- It's easier, but more computation
- Recall definition of derivative

$$\mathbf{J} = \left[\frac{\partial f_i}{\partial x_j}\right] = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \cdots & \partial f_1/\partial x_M \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \cdots & \partial f_2/\partial x_M \\ \vdots & \vdots & & \vdots \\ \partial f_N/\partial x_1 & \partial f_N/\partial x_2 & \cdots & \partial f_N/\partial x_M \end{pmatrix}$$

$$\frac{\partial f(x_1, x_2, \ldots x_N)}{\partial x_i} \approx \frac{f(x_1, \ldots, x_i + \varepsilon, \ldots x_N) - f(x_1, x_2, \ldots x_N)}{\varepsilon}$$

$$\frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_i} \approx \frac{\mathbf{f}(\mathbf{x} + \varepsilon \hat{\mathbf{u}}_i) - \mathbf{f}(\mathbf{x})}{\varepsilon}$$

$$\mathbf{J} = \begin{pmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \frac{\partial \mathbf{f}}{\partial x_2} & \vdots & \frac{\partial \mathbf{f}}{\partial x_N} \end{pmatrix}$$

Column vectors

- Matlab code:

```
% Estimate J numerically
e = 1e-6;    % A tiny number
J(:,1) = (fRigid(x+[e;0;0],pA) - y)/e;
J(:,2) = (fRigid(x+[0;e;0],pA) - y)/e;
J(:,3) = (fRigid(x+[0;0;e],pA) - y)/e;
```

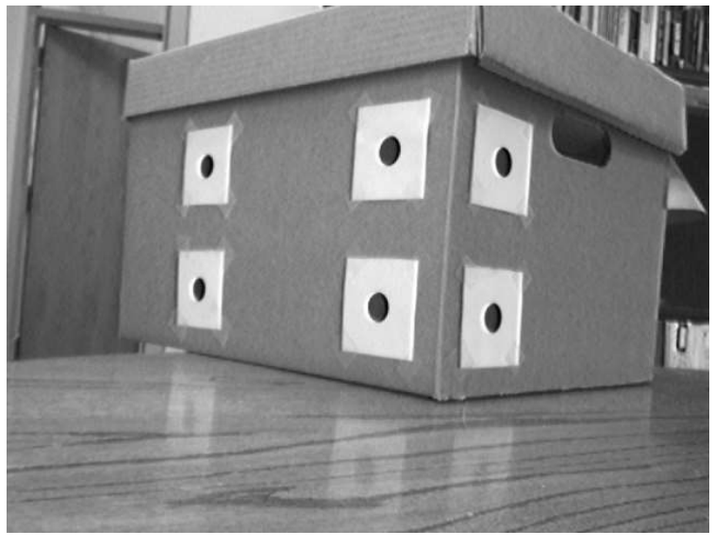
Orientation, not very relevant for the exam

Nonlinear Pose Estimation

Model-based Pose Estimation

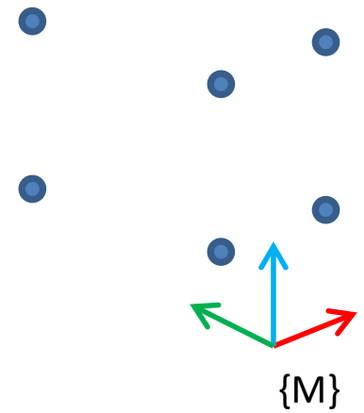
- Problem Statement
  - Given
    - We have an image of an object
    - We know the model geometry of the object (specifically, the location of features on the object)
    - We have found the corresponding features in the image
  - Find
    - The position and orientation (pose) of the object with respect to the camera

 # points needed?



- Assumptions
  - Object is rigid (so 6 only degrees of freedom)
  - Camera intrinsic parameters are known

- We will find the pose that minimizes the squared error of the predicted locations of the image features, to the measured locations



we want ${}^{C}_{M}\mathbf{H}$

Least Squares Pose Estimation

- Let **y** = **f**(**x**)
  - **x** is a vector of the unknown pose parameters
  - **f** is a function that returns the predicted image points **y**, given the pose **x**
  - **y0** is a vector of the actual observed image points

$$\mathbf{y} = \begin{pmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \vdots \\ x_N \\ y_N \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} \theta_x \\ \theta_y \\ \theta_z \\ t_x \\ t_y \\ t_z \end{pmatrix}$$

- We want to find **x** to minimize $E = |\mathbf{f}(\mathbf{x}) - \mathbf{y0}|^2$

- Algorithm:

1. We start with a guess for **x**, call it **x0**
2. Compute **y** = **f(x)**. Residual error is **dy** = **y**-**y0**
3. Calculate Jacobian of **f**, **J** = [∂**f**/ ∂**x**], and evaluate it at **x**. We now have **dy** = **J dx**
4. Solve for **dx** using pseudo inverse $\mathbf{dx} = (\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T\,\mathbf{dy}$
5. Set **x** <= **x** + **dx**
6. Repeat steps 2-5 until convergence (no more change in **x**)

Recall Perspective Projection

- Projection of a 3D point ${}^{W}\mathbf{P}$ in the world to a point in the pixel image (x_{im},y_{im})

$$\tilde{\mathbf{p}} = \mathbf{K}\,\mathbf{M}_{ext}\,{}^{W}\mathbf{P} \qquad \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \mathbf{K}\,\mathbf{M}_{ext}\,{}^{W}\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}, \qquad x_{im} = x_1 / x_3,\ \ y_{im} = x_2 / x_3$$

- Where the extrinsic parameter matrix is

$$\mathbf{M}_{ext} = \begin{pmatrix} {}^{C}_{W}\mathbf{R} & {}^{C}\mathbf{t}_{Worg} \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_X \\ r_{21} & r_{22} & r_{23} & t_Y \\ r_{31} & r_{32} & r_{33} & t_Z \end{pmatrix}$$

- And the intrinsic parameter matrix

$$\mathbf{K} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

- Or, if we use “model” instead of “world” frame for the point:

$$\tilde{\mathbf{p}} = \mathbf{K}\,\mathbf{M}_{ext}\,{}^{M}\mathbf{P} = \mathbf{K}\begin{pmatrix} {}^{C}_{M}\mathbf{R} & {}^{C}t_{Morg} \end{pmatrix}{}^{M}\mathbf{P}$$

Recall XYZ fixed angle convention

- R = Rz Ry Rx, where

$$R_Z = \begin{pmatrix} \cos\theta_Z & -\sin\theta_Z & 0 \\ \sin\theta_Z & \cos\theta_Z & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad R_Y = \begin{pmatrix} \cos\theta_Y & 0 & \sin\theta_Y \\ 0 & 1 & 0 \\ -\sin\theta_Y & 0 & \cos\theta_Y \end{pmatrix} \quad R_X = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_X & -\sin\theta_X \\ 0 & \sin\theta_X & \cos\theta_X \end{pmatrix}$$

- Matlab

```
Rx = [ 1 0 0; 0 cos(ax) -sin(ax); 0 sin(ax) cos(ax)];
Ry = [ cos(ay) 0 sin(ay); 0 1 0; -sin(ay) 0 cos(ay)];
Rz = [ cos(az) -sin(az) 0; sin(az) cos(az) 0; 0 0 1];
R = Rz * Ry * Rx
```

Note – in general, the "angle-axis" convention would be better to use than XYZ angles

Function to project one point

- Write a function to project a 3D point P_M in model coordinates to image point p, given the model-to-camera pose x = (ax,ay,az,tx,ty,tz)
  - P_M = [X;Y;Z;1] is the input point
  - x = [ax;ay;az;tx;ty;tz] is the vector of model-to-camera pose parameters
  - K = intrinsic camera matrix
  - p = [x;y] is the output point

```
function p = fProject(x, P_M, K)
% Project 3D point onto image

% Get pose params
ax = x(1); ay = x(2); az = x(3);
tx = x(4); ty = x(5); tz = x(6);

% Rotation matrix, model to camera
Rx = [ 1 0 0; 0 cos(ax) -sin(ax); 0 sin(ax) cos(ax)];
Ry = [ cos(ay) 0 sin(ay); 0 1 0; -sin(ay) 0 cos(ay)];
Rz = [ cos(az) -sin(az) 0; sin(az) cos(az) 0; 0 0 1];
R = Rz * Ry * Rx;

% Extrinsic camera matrix
Mext = [ R  [tx;ty;tz] ];

% Project point
ph = K*Mext*P_M;
ph = ph/ph(3);

p = ph(1:2);
return
```

Function to transform a set of points

- Now modify the function to transform a set of points
  - P_M = is a set of input points
  - **x** = [ax;ay;az;tx;ty;tz] is the pose
  - K = intrinsic camera matrix
  - p = [x1;y1;x2;y2; …] are the output points

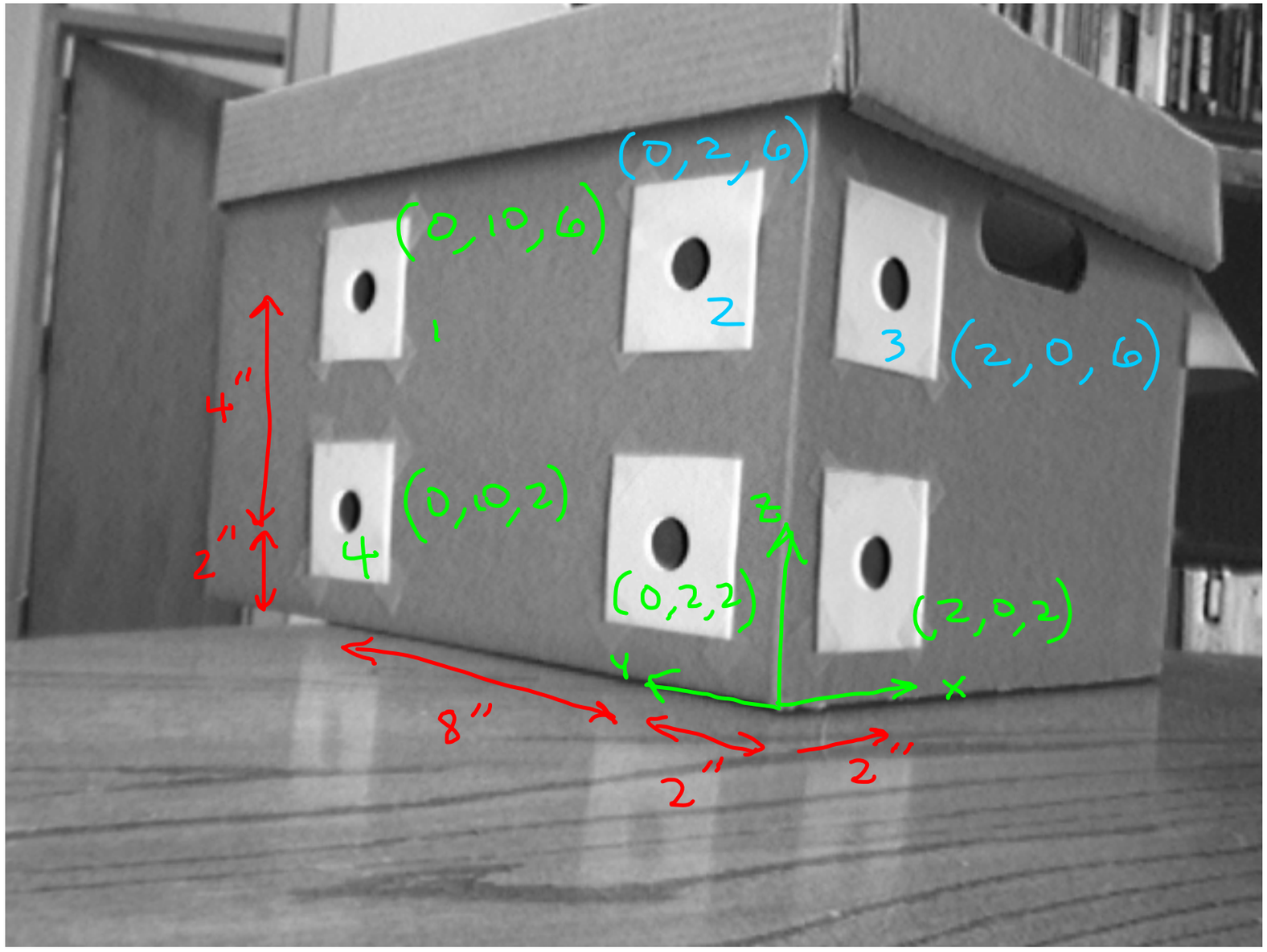
$$ {}^{M}\mathbf{P} = \begin{pmatrix} X_1 & X_2 & \cdots & X_N \\ Y_1 & Y_2 & \cdots & Y_N \\ Z_1 & Z_2 & \cdots & Z_N \\ 1 & 1 & \cdots & 1 \end{pmatrix} \qquad \mathbf{p} = \begin{pmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \vdots \\ x_N \\ y_N \end{pmatrix} $$

```
function p = fProject(x, P_M, K)
```

Function to transform a set of points

```
function p = fProject(x, P_M, K)
% Project 3D points onto image

% Get pose params
ax = x(1); ay = x(2); az = x(3);
tx = x(4); ty = x(5); tz = x(6);

% Rotation matrix, model to camera
Rx = [ 1 0 0; 0 cos(ax) -sin(ax); 0 sin(ax) cos(ax)];
Ry = [ cos(ay) 0 sin(ay); 0 1 0; -sin(ay) 0 cos(ay)];
Rz = [ cos(az) -sin(az) 0; sin(az) cos(az) 0; 0 0 1];
R = Rz * Ry * Rx;

% Extrinsic camera matrix
Mext = [ R  [tx;ty;tz] ];

% Project points
ph = K*Mext*P_M;

% Divide through 3rd element of each column
ph(1,:) = ph(1,:)./ph(3,:);
ph(2,:) = ph(2,:)./ph(3,:);
ph = ph(1:2,:); % Get rid of 3rd row

p = reshape(ph, [], 1); % reshape into 2Nx1 vector
return
```

Box Model



Focal length in pixels: 715

Image center (x,y): (354, 245)

Example

```
clear all
close all

I = imread('img1_rect.tif');
imshow(I, [])

% These are the points in the model's coordinate system (inches)
P_M = [  0      0       2       0       0       2;
        10      2       0      10       2       0;
         6      6       6       2       2       2;
         1      1       1       1       1       1 ];

% Define camera parameters
f = 715;      % focal length in pixels
cx = 354;
cy = 245;

K = [ f 0 cx; 0 f cy; 0 0 1 ];   % intrinsic parameter matrix

y0 = [ 183; 147;     % 1
    350; 133;        % 2
    454; 144;        % 3
    176; 258;        % 4
    339; 275;        % 5
    444; 286 ];      % 6

% Make an initial guess of the pose [ax ay az tx ty tz]
x = [1.5; -1.0; 0.0; 0; 0; 30];

% Get predicted image points by substituting in the current pose
y = fProject(x, P_M, K);

for i=1:2:length(y)
    rectangle('Position', [y(i)-8 y(i+1)-8 16 16], 'FaceColor', 'r');
end
```

File Home Insert Design Transitions Animations Slide Show Review View Format

From Beginning | From Current Slide | Broadcast Slide Show | Custom Slide Show | Set U Slide Sh

Start Slide Show

Slides Outline

9 Example

10 Example

11 Computing Jacobian Numerically

12

13

14 Overlaying Graphical Model

Editor - C:\Users\whoff\Documents\Teaching\Eggn512\2011\demos\19-PoseEstimation\junk.m

MATLAB 7.11.0 (R2010b)

File Edit Debug Desktop Window Help

:uments\Teaching\Eggn512\2011\demos\19-PoseEstimation

Shortcuts How to Add What's New

New to MATLAB? Watch this Video, see Demos, or read Getting Started.

```
>> junk
```

Figure 1

File Edit View Insert Tools Desktop Window Help

```
cle
clo
I =
ims
% D
% M
f =
cx
cy
K =
y0
% M
x =
% G
y =
for
end
```

Start

OVR

Animation Pane

Play

Computing Jacobian Numerically

- We approximate the derivatives

$$\frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_i} \approx \frac{\mathbf{f}(\mathbf{x}+\varepsilon\hat{\mathbf{u}}_i)-\mathbf{f}(\mathbf{x})}{\varepsilon}$$

$$\mathbf{J} = \left[\frac{\partial f_i}{\partial x_j}\right] = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \cdots & \partial f_1/\partial x_M \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \cdots & \partial f_2/\partial x_M \\ \vdots & \vdots & & \vdots \\ \partial f_N/\partial x_1 & \partial f_N/\partial x_2 & \cdots & \partial f_N/\partial x_M \end{pmatrix}$$

- Matlab code:

```
y = fProject(x,P_M,K);

e = 0.000001;   % a tiny number
J(:,1) = ( fProject(x+[e;0;0;0;0;0],P_M,K) - y )/e;
J(:,2) = ( fProject(x+[0;e;0;0;0;0],P_M,K) - y )/e;
J(:,3) = ( fProject(x+[0;0;e;0;0;0],P_M,K) - y )/e;
J(:,4) = ( fProject(x+[0;0;0;e;0;0],P_M,K) - y )/e;
J(:,5) = ( fProject(x+[0;0;0;0;e;0],P_M,K) - y )/e;
J(:,6) = ( fProject(x+[0;0;0;0;0;e],P_M,K) - y )/e;
```

We have

- **y0** = observations or measurements
- **x0** = a guess for **x**
- **y** = **f(x)** is a non linear function

1. Initialize **x** to **x0**
2. Compute **y** = **f(x)**. Residual error is **dy** = **y**-**y0**
3. Calculate Jacobian of **f**, evaluate it at **x**. We now have **dy** = **J dx**
4. Solve for **dx** using pseudo inverse **dx** = $(J^T J)^{-1} J^T$ **dy**
5. Set **x** <= **x** + **dx**
6. Repeat steps 2-5 until convergence (no more change in **x**)

```
for i=1:10
    fprintf('\nIteration %d\nCurrent pose:\n', i);
    disp(x);

    % Get predicted image points
    y = fProject(x, P_M, K);

    imshow(I, [])
    for i=1:2:length(y)
        rectangle('Position', [y(i)-8 y(i+1)-8 16 16], ...
            'FaceColor', 'r');
    end
    pause(1);

    % Estimate Jacobian
    e = 0.00001;    % a tiny number
    :

    % Error is observed image points - predicted image points
    dy = y0 - y;
    fprintf('Residual error: %f\n', norm(dy));

    % Ok, now we have a system of linear equations   dy = J dx
    % Solve for dx using the pseudo inverse
    dx = pinv(J) * dy;

    % Stop if parameters are no longer changing
    if abs( norm(dx)/norm(x) ) < 1e-6
        break;
    end

    x = x + dx;    % Update pose estimate
end
```

```
for i=1:10
    fprintf('\nIteration %d\nCurrent pose:\n', i);
    disp(x);

    % Get predicted image points
    y = fProject(x, P_M, K);

    imshow(I, [])
    for i=1:2:length(y)
        rectangle('Position', [y(i)-8 y(i+1)-8 16 16], ...
            'FaceColor', 'r');
    end
    pause(1);

    % Estimate Jacobian
    e = 0.00001;    % a tiny number
    J(:,1) = ( fProject(x+[e;0;0;0;0;0],P_M,K) - y )/e;
    J(:,2) = ( fProject(x+[0;e;0;0;0;0],P_M,K) - y )/e;
    J(:,3) = ( fProject(x+[0;0;e;0;0;0],P_M,K) - y )/e;
    J(:,4) = ( fProject(x+[0;0;0;e;0;0],P_M,K) - y )/e;
    J(:,5) = ( fProject(x+[0;0;0;0;e;0],P_M,K) - y )/e;
    J(:,6) = ( fProject(x+[0;0;0;0;0;e],P_M,K) - y )/e;


    % Error is observed image points - predicted image points
    dy = y0 - y;
    fprintf('Residual error: %f\n', norm(dy));

    % Ok, now we have a system of linear equations   dy = J dx
    % Solve for dx using the pseudo inverse
    dx = pinv(J) * dy;

    % Stop if parameters are no longer changing
    if abs( norm(dx)/norm(x) ) < 1e-6
        break;
    end

    x = x + dx;    % Update pose estimate
end
```

File Home Insert Design Transitions Animations Slide Show Review View Format

From Beginning | From Current Slide | Broadcast Slide Show | Custom Slide Show | Start Slide Show

Slides Outline

9 Example

10 Example

11 Computing Jacobian Numerically

12

13

14 Overlaying Graphical Model

Figure 1

File Edit View Insert Tools Desktop Window Help

Animation Pane

Play

end

File Home Insert Design Transitions Animations Slide Show Review View Format

From Beginning | From Current Slide | Broadcast Slide Show | Custom Slide Show | Start Slide Show

Slides Outline

9 Example

10 Example

11 Computing Jacobian Numerically

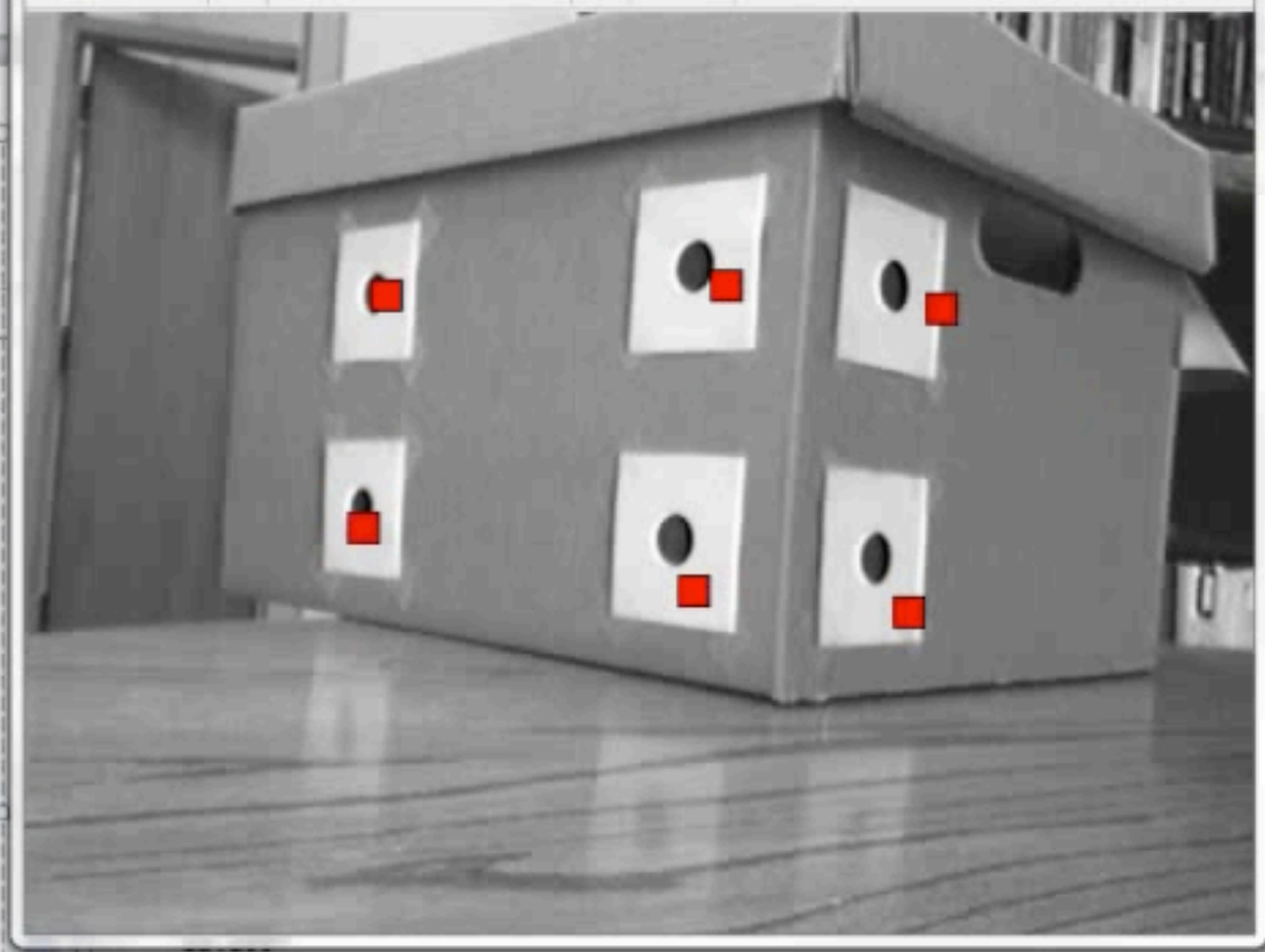
12

13

14 Overlaying Graphical Model

Figure 1

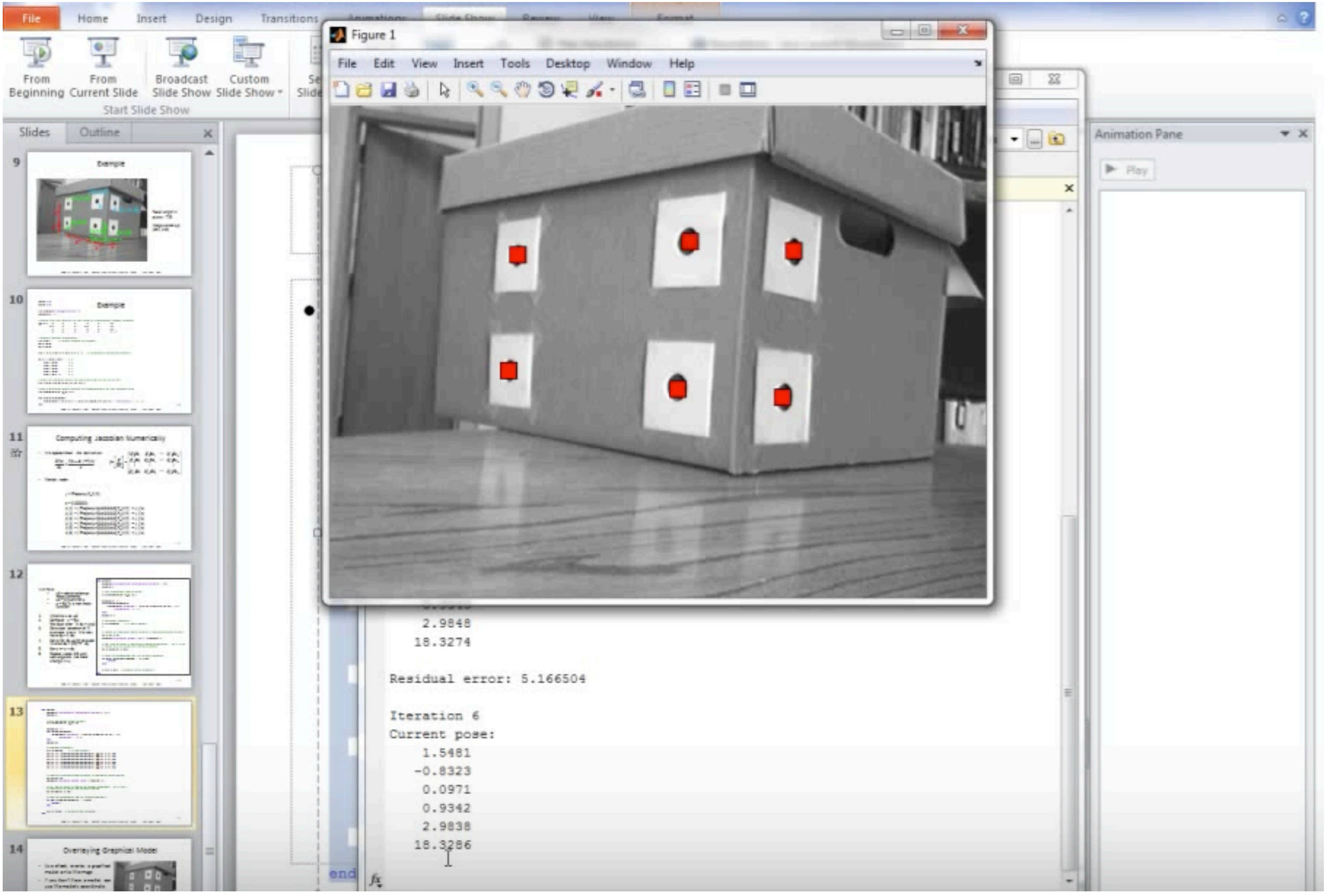
File Edit View Insert Tools Desktop Window Help



Animation Pane

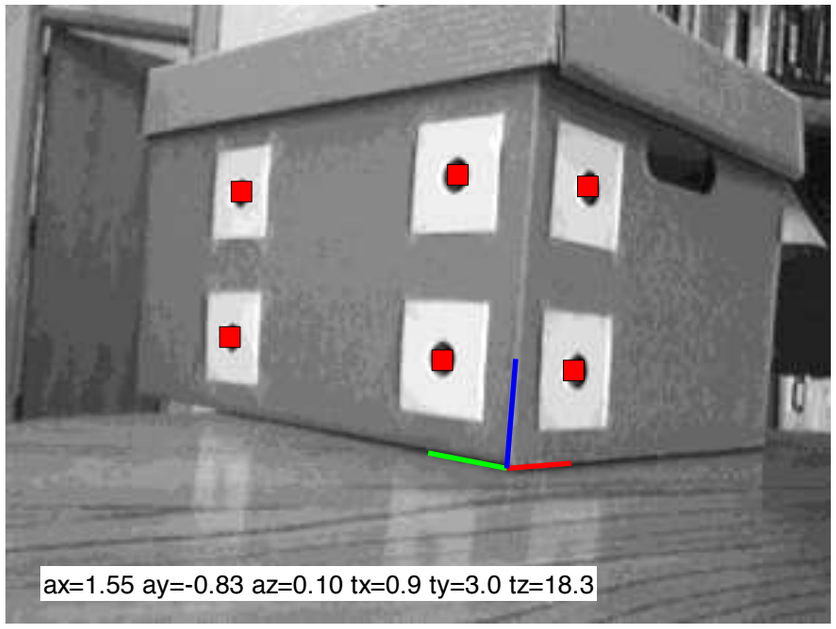
Play

```
Residual error: 230.074282

Iteration 3
Current pose:
    1.5809
   -0.7999
    0.0969
    1.0954
    3.5669
   16.5158

fx

end
```


File
Home
Insert
Design
Transitions
From Beginning
From Current Slide
Broadcast Slide Show
Custom Slide Show
Start Slide Show
Slides
Outline
Figure 1
File Edit View Insert Tools Desktop Window Help
Animation Pane
Play
2.9848
18.3274
Residual error: 5.166504
Iteration 6
Current pose:
1.5481
-0.8323
0.0971
0.9342
2.9838
18.3286
end

Overlaying Graphical Model

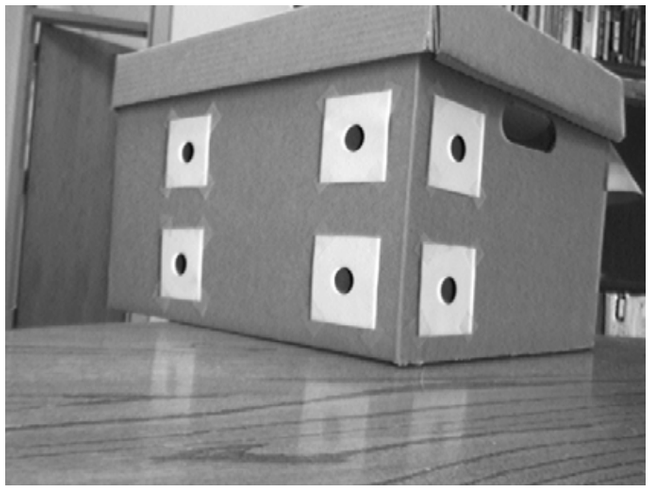
- As a check, overlay a graphical model onto the image
- If you don't have a model, you can display the model's coordinate axes



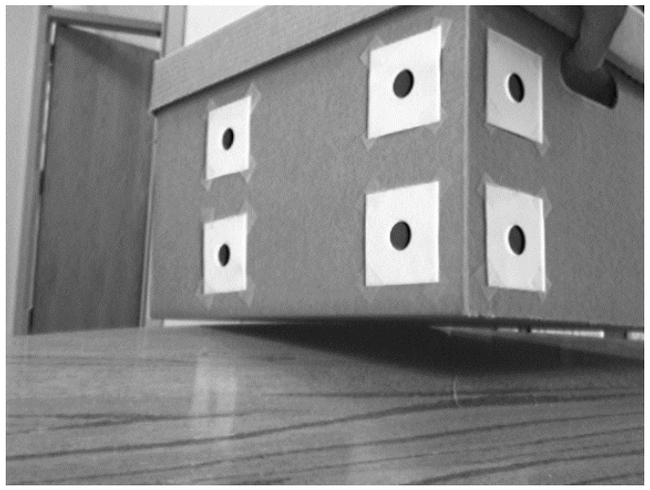

```
% Draw coordinate axes onto the image.  Scale the length of the axes
% according to the size of the model, so that the axes are visible.
W = max(P_M,[],2) - min(P_M,[],2);  % Size of model in X,Y,Z
W = norm(W);   % Length of the diagonal of the bounding box

u0 = fProject(x, [0;0;0;1], K);  % origin
uX = fProject(x, [W/5;0;0;1], K);  % unit X vector
uY = fProject(x, [0;W/5;0;1], K);  % unit Y vector
uZ = fProject(x, [0;0;W/5;1], K);  % unit Z vector

line([u0(1) uX(1)], [u0(2) uX(2)], 'Color', 'r', 'LineWidth', 3);
line([u0(1) uY(1)], [u0(2) uY(2)], 'Color', 'g', 'LineWidth', 3);
line([u0(1) uZ(1)], [u0(2) uZ(2)], 'Color', 'b', 'LineWidth', 3);

% Also print the pose onto the image.
text(30,450,sprintf('ax=%.2f ay=%.2f az=%.2f tx=%.1f ty=%.1f tz=%.1f', ...
    x(1), x(2), x(3), x(4), x(5), x(6)), ...
    'BackgroundColor', 'w', 'FontSize', 15);
```

Box Example

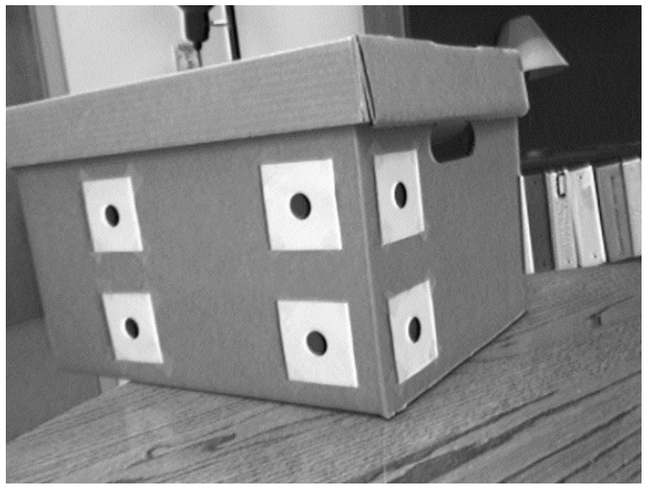
- Find the pose of the box with respect to the camera in all three images



img1_rect.tif



img2_rect.tif



img3_rect.tif

- How close does the initial guess have to be, for the solution to converge?

Orientation, not very relevant for the exam

Linear Pose Estimation

Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD)

- SVD is a matrix technique that has some important uses in computer vision
- These include:
  - Solving a set of homogeneous linear equations
    - Namely we solve for the vector **x** in the equation **Ax** = **0**
  - Guaranteeing that the entries of a matrix estimated numerically satisfy some given constraints (e.g., orthogonality)
    - For example, we have computed **R** and now want to make sure that it is a valid rotation matrix

Singular Value Decomposition (SVD)

- Any (real) *mxn* matrix **A** can be written as the product of three matrices

$$\mathbf{A} = \mathbf{U}\,\mathbf{D}\,\mathbf{V}^T$$

  - **U** (*mxm*) and **V**(*nxn*) have columns that are mutually orthogonal unit vectors
  - **D** (*mxn*) is diagonal; its diagonal elements σ_i are called singular values, and $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_n \geq 0$

p=min(M,N)

$$\boldsymbol{A}_{M\times N} = \boldsymbol{U}_{M\times P}\boldsymbol{\Sigma}_{P\times P}\boldsymbol{V}^T_{P\times N}$$

$$= \left[\begin{array}{c|c|c} \boldsymbol{u}_0 & \cdots & \boldsymbol{u}_{p-1} \end{array}\right] \left[\begin{array}{ccc} \sigma_0 & & \\ & \ddots & \\ & & \sigma_{p-1} \end{array}\right] \left[\begin{array}{c} \boldsymbol{v}_0^T \\ \hline \cdots \\ \hline \boldsymbol{v}_{p-1}^T \end{array}\right],$$

- If only the first *r* singular values are positive, the matrix **A** is of rank *r* and we can drop the last *p-r* columns of **U** and **V**

$$\mathbf{U}^T\mathbf{U} = \mathbf{I},\ \mathbf{V}^T\mathbf{V} = \mathbf{I}$$

$$\mathbf{u}_i \cdot \mathbf{u}_j = \mathbf{v}_i \cdot \mathbf{v}_j = \delta_{ij}$$

Some properties of SVD

- We have

  $\mathbf{A} = \mathbf{U}\,\mathbf{D}\,\mathbf{V}^T$

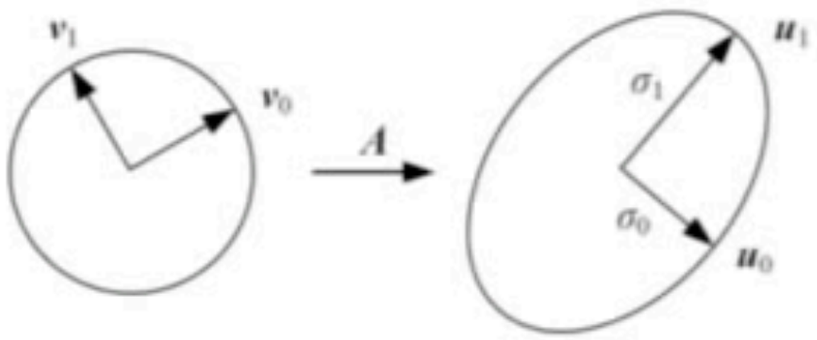
- Multiplying by V on the right on each side yields

  $\mathbf{AV} = \mathbf{U}\,\mathbf{D}$

- or

  $\mathbf{A}\,\mathbf{v}_j = \sigma_j\,\mathbf{u}_j$

- In other words, matrix **A** takes any basis vector $\mathbf{v}_j$ and maps it to a direction $\mathbf{u}_j$ with length σ_j



from "Computer Vision: Algorithms and Applications", Richard Szeliski, 2010, Springer

Figure A.1 The action of a matrix A can be visualized by thinking of the domain as being spanned by a set of orthonormal vectors v_j, each of which is transformed to a new orthogonal vector u_j with a length σ_j. When A is interpreted as a covariance matrix and its eigenvalue decomposition is performed, each of the u_j axes denote a principal direction (component) and each σ_j denotes one standard deviation along that direction.

Some properties of SVD

- We can represent A in terms of the vectors u and v

$$\mathbf{A}\,\mathbf{v}_j = \sigma_j\,\mathbf{u}_j$$

- or

$$\mathbf{A} = \sum_{j=0}^{p-1} \sigma_j \mathbf{u}_j \mathbf{v}_j^T$$

- The vectors $\mathbf{u}_j$ are called the "principal components" of **A**

- Sometimes we want to compute an approximation to **A** using fewer principal components
- If we truncate the expansion, we obtain the best possible least squares approximation[1] to the original matrix **A**

$$\mathbf{A} \approx \sum_{j=0}^{t} \sigma_j \mathbf{u}_j \mathbf{v}_j^T$$

[1]In terms of the Frobenius norm, defined as

$$\|\mathbf{A}\|_F = \sum_{i,j} a_{i,j}^2$$

Some properties of SVD (continued)

- We have

  $\mathbf{A} = \mathbf{U}\,\mathbf{D}\,\mathbf{V}^T$

- Look at

  $\mathbf{A}\,\mathbf{A}^T = (\mathbf{U}\,\mathbf{D}\,\mathbf{V}^T)\,(\mathbf{U}\,\mathbf{D}\,\mathbf{V}^T)^T = \mathbf{U}\,\mathbf{D}\,\mathbf{V}^T\,\mathbf{V}\,\mathbf{D}\,\mathbf{U}^T = \mathbf{U}\,\Lambda\,\mathbf{U}^T$

- where $\lambda_i = \sigma_i^2$

- Multiplying by **U** on the right on each side yields

  $(\mathbf{A}\,\mathbf{A}^T)\,\mathbf{U} = \mathbf{U}\,\Lambda$

- or

  $(\mathbf{A}\,\mathbf{A}^T)\,\mathbf{u}_j = \lambda_j\,\mathbf{u}_j$

- So the columns of **U** are the eigenvectors of $\mathbf{A}\,\mathbf{A}^T$

Some properties of SVD (continued)

- Similarly, we have

  $\mathbf{A} = \mathbf{U}\,\mathbf{D}\,\mathbf{V}^T$

- Look at

  $\mathbf{A}^T\mathbf{A} = (\mathbf{U}\,\mathbf{D}\,\mathbf{V}^T)^T(\mathbf{U}\,\mathbf{D}\,\mathbf{V}^T) = \mathbf{V}\,\mathbf{D}\,\mathbf{U}^T\mathbf{U}\,\mathbf{D}\,\mathbf{V}^T = \mathbf{V}\,\Lambda\,\mathbf{V}^T$

- where $\lambda_i = \sigma_i^2$

- Multiplying by **V** on the right on each side yields

  $(\mathbf{A}^T\mathbf{A})\,\mathbf{V} = \mathbf{V}\,\Lambda$

- or

  $(\mathbf{A}^T\mathbf{A})\,\mathbf{v}_j = \lambda_j\,\mathbf{v}_j$

- So the columns of **V** are the eigenvectors of $\mathbf{A}^T\mathbf{A}$

Application: Solving a System of Homogeneous Equations

- We want to solve a system of *m* linear equations in *n* unknowns, of the form **Ax** = 0
  - Assume *m* >= *n*-1 and rank(**A**)=*n*-1

- Any vectors **x** that satisfy **Ax** = 0 are in the "null space" of A
  - **x**=0 is a solution, but it is not interesting
  - If you find a solution **x**, then any scaled version of **x** is also a solution

- As we will see, these equations can arise when we want to solve for
  - The elements of a camera projection matrix
  - The elements of a homography transform

Application: Solving a System of Homogeneous Equations (continued)

- The solution **x** is the eigenvector corresponding to the only zero eigenvalue of $\mathbf{A}^T\mathbf{A}$
  - Proof: We want to minimize

$$\|\mathbf{Ax}\|^2 = (\mathbf{Ax})^T \mathbf{Ax} = \mathbf{x}^T \mathbf{A}^T \mathbf{Ax} \qquad \text{subject to} \quad \mathbf{x}^T\mathbf{x} = 1$$

  - Introducing a Lagrange multiplier λ, this is equivalent to minimizing

$$L(\mathbf{x}) = \mathbf{x}^T \mathbf{A}^T \mathbf{Ax} - \lambda(\mathbf{x}^T\mathbf{x} - 1)$$

  - Take derivative wrt **x** and set to zero

$$\mathbf{A}^T \mathbf{Ax} - \lambda\mathbf{x} = 0$$

  - Thus, λ is an eigenvalue of $\mathbf{A}^T\mathbf{A}$, and $\mathbf{x} = \mathbf{e}_\lambda$ is the corresponding eigenvector. $L(\mathbf{e}_\lambda) = \lambda$ is minimized at λ=0, so $\mathbf{x} = \mathbf{e}_0$ is the eigenvector corresponding to the zero eigenvalue.

Example

- Let

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

- Find solution **x** to **Ax=0**

$$\mathbf{A}^T\mathbf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Eigenvalues and eigenvectors of $\mathbf{A}^T\mathbf{A}$:

$$\lambda_1 = 0, \mathbf{e}_1 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad \lambda_2 = 1, \mathbf{e}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \lambda_3 = 1, \mathbf{e}_3 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

So $\mathbf{x}=\mathbf{e}_1$ is the solution. To verify:

$$\mathbf{Ax} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \mathbf{0}$$

So it does work

Solving Homogeneous Equations with SVD

- Given a system of linear equations **Ax** = 0
- Then the solution **x** is the eigenvector corresponding to the only zero eigenvalue of $\mathbf{A}^T\mathbf{A}$
- Equivalently, we can take the SVD of **A**; ie., $\mathbf{A} = \mathbf{U}\,\mathbf{D}\,\mathbf{V}^T$
  - And **x** is the column of **V** corresponding to the zero singular value of **A**
  - (Since the columns are ordered, this is the rightmost column of **V**)

- Example

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$\text{Svd}: \quad \mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^T = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

So the last column of V is indeed the solution **x**

Solving Homogeneous Equations - Matlab

```
clear all
close all

% Solve the system of equations Ax = 0
A = [ 1   0   0;
      0   1   0 ];

[U,D,V] = svd(A);
x = V(:,end);                    % get last column of V
```

- Output

```
>> U
U =
    1    0
    0    1
>> D
D =
    1    0    0
    0    1    0
```

```
>> V
V =
    1    0    0
    0    1    0
    0    0    1
>> x
x =
    0
    0
    1
```

Another application: Enforcing constraints

- Sometimes you generate a numerical estimate of a matrix **A**
  - The values of **A** are not all independent, but satisfy some algebraic constraints
  - For example, the columns and rows of a rotation matrix should be orthonormal
  - However, the matrix you found, **A'**, does not satisfy the constraints
- SVD can find the closest matrix[1] to **A** that satisfies the constraints exactly
- Procedure:
  - You take the SVD of $\mathbf{A}' = \mathbf{U}\,\mathbf{D}\,\mathbf{V}^T$
  - Create matrix **D'** with singular values equal to those expected when the constraints are satisfied exactly
  - Then $\mathbf{A} = \mathbf{U}\,\mathbf{D}'\,\mathbf{V}^T$ satisfies the desired constraints by construction

[1]In terms of the Frobenius norm

Example – rotation matrix

- The singular values of R should all be equal to 1 ... we will enforce this

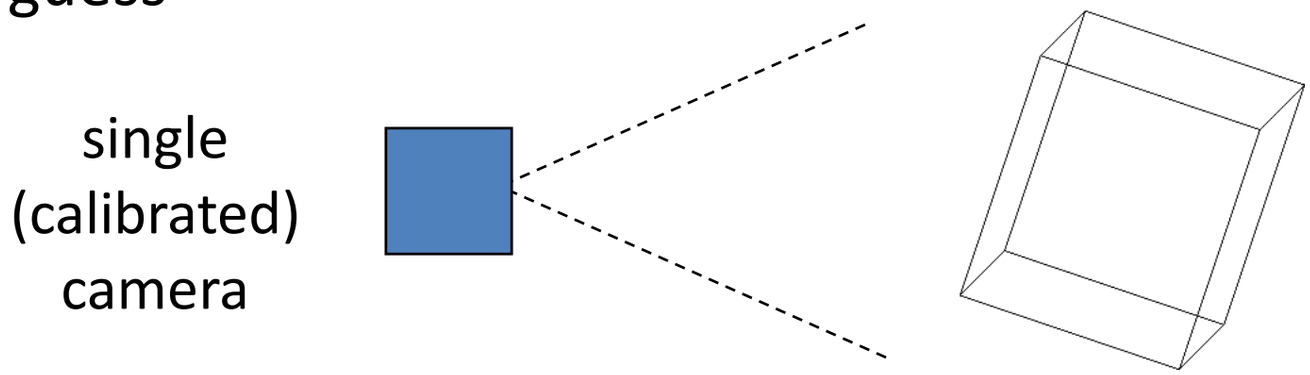
```
clear all
close all

% Make a valid rotation matrix
ax = 0.1;   ay = -0.2;  az = 0.3;    % radians
Rx = [ 1 0 0; 0 cos(ax) -sin(ax); 0 sin(ax) cos(ax)];
Ry = [ cos(ay) 0 sin(ay); 0 1 0; -sin(ay) 0 cos(ay)];
Rz = [ cos(az) -sin(az) 0; sin(az) cos(az) 0; 0 0 1];

R = Rz * Ry * Rx

% Ok, perturb the elements of R a little
Rp = R + 0.01*randn(3,3)

[U,D,V] = svd(Rp);  % Take SVD of Rp

D    % Here is the actual matrix of singular values

% Recover a valid rotation matrix by enforcing constraints
Rc = U * eye(3,3) * V'
```

Direct Linear Transform (DLT)

Linear Pose Estimation

Linear Pose Estimation

- We have seen how to compute pose, from 2D-3D point correspondences, using non-linear least squares
  - This gives the most accurate results; however, it requires a good initial guess



-> Can determine the pose of the model

- Now we will look at how to estimate pose using a linear method, that doesn't require an initial guess
  - The linear method is called "Direct Linear Transform" (DLT)
- For best results, use the linear method to get an initial guess, then refine it with the nonlinear method

Direct Linear Transform (DLT)

- We can directly solve for the elements of the camera projection matrix
- Recall the projection of a 3D point ${}^{W}\mathbf{P}$ in the world to a point in the pixel image (x_{im}, y_{im})

$$\tilde{\mathbf{p}} = \mathbf{K}\,\mathbf{M}_{ext}\ {}^{W}\mathbf{P} \qquad \tilde{\mathbf{p}} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \mathbf{K}\,\mathbf{M}_{ext} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}, \qquad x_{im} = x_1 / x_3, \;\; y_{im} = x_2 / x_3$$

- Where the extrinsic parameter matrix is

$$\mathbf{M}_{ext} = \begin{pmatrix} {}^{C}_{W}\mathbf{R} & {}^{C}\mathbf{t}_{Worg} \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_X \\ r_{21} & r_{22} & r_{23} & t_Y \\ r_{31} & r_{32} & r_{33} & t_Z \end{pmatrix}$$

- And the intrinsic parameter matrix

$$\mathbf{K} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

- We will solve for the 12 elements of $\mathbf{M}_{ext}$ by treating them as independent (of course, they are not independent!)

Normalized Image Coordinates

- We will work with “normalized” image points:
- If we know the intrinsic camera parameter matrix, we can convert the image points to “normalized” image coordinates
  - Origin is in center of image
  - Effective focal length equals 1
  - $x_{normalized} = X/Z$, $y_{normalized} = Y/Z$

Note – Hartley and Zisserman say that you should precondition in the input values; ie., translate and scale the image points so that the centroid of the points is at the origin, and the average distance of the points to the origin is equal to sqrt(2).

- Then
  - $\mathbf{p}_{unnormalized} = \mathbf{K}\,\mathbf{p}_{normalized}$
  - $\mathbf{p}_{normalized} = (\mathbf{K})^{-1}\,\mathbf{p}_{unnormalized}$
- where **K** is the intrinsic parameter matrix

Direct Linear Transform (DLT)

- The projection of a 3D point ${}^{W}\mathbf{P}$ in the world to a normalized image point is

$$\tilde{\mathbf{p}}_n = \mathbf{M}_{ext}\mathbf{P} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \quad \text{or} \quad x = \frac{r_{11}X + r_{12}Y + r_{13}Z + t_x}{r_{31}X + r_{32}Y + r_{33}Z + t_z}, \quad y = \frac{r_{21}X + r_{22}Y + r_{23}Z + t_y}{r_{31}X + r_{32}Y + r_{33}Z + t_z}$$

- Multiplying by the denominator

$$r_{11}X + r_{12}Y + r_{13}Z + t_x - x\left(r_{31}X + r_{32}Y + r_{33}Z + t_z\right) = 0$$
$$r_{21}X + r_{22}Y + r_{23}Z + t_y - y\left(r_{31}X + r_{32}Y + r_{33}Z + t_z\right) = 0$$

- Put into the form **A x** = 0

$$\mathbf{A}\mathbf{x} = \begin{pmatrix} X & Y & Z & 0 & 0 & 0 & -xX & -xY & -xZ & 1 & 0 & -x \\ 0 & 0 & 0 & X & Y & Z & -yX & -yY & -yZ & 0 & 1 & -y \end{pmatrix} \begin{pmatrix} r_{11} \\ r_{12} \\ r_{13} \\ r_{21} \\ r_{22} \\ r_{23} \\ r_{31} \\ r_{32} \\ r_{33} \\ t_x \\ t_y \\ t_z \end{pmatrix} = \mathbf{0}$$

*How many points do we need to solve for **x**?*

Solving a System of Homogeneous Equations

- We want to solve a system of *m* linear equations in *n* unknowns, of the form $\mathbf{Ax} = 0$
  - Note that any scaled version of **x** is also a solution (**x**=0 is not interesting)

- The solution **x** is the eigenvector corresponding to the only zero eigenvalue of $\mathbf{A}^T\mathbf{A}$

- Equivalently, we can take the SVD of **A**; ie., $\mathbf{A} = \mathbf{U}\,\mathbf{D}\,\mathbf{V}^T$
  - And **x** is the column of **V** corresponding to the zero singular value of **A**
  - (Since the columns are ordered, this is the rightmost column of **V**)

DLT Example

```
% Solve for the value of x that satisfies Ax = 0.
% The solution to Ax=0 is the singular vector of A corresponding to the
% smallest singular value; that is, the last column of V in A=UDV'
[U,D,V] = svd(A);
x = V(:,end);                    % get last column of V

% Reshape x back to a 3x4 matrix, M = [R  t]
M = [ x(1)  x(2)  x(3)  x(10);
      x(4)  x(5)  x(6)  x(11);
      x(7)  x(8)  x(9)  x(12) ];
```

- We now have the camera extrinsic matrix M (up to a scale factor).
- Now, we need to extract the rotation and translation from M.

Extracting translation

- The projection matrix is a 3x4 matrix

$$\mathbf{M} = \begin{bmatrix} {}^{c}_{m}\mathbf{R} & {}^{c}\mathbf{t}_{morg} \end{bmatrix}$$

rotation matrix, model to camera

origin of model with respect to camera

- Recall (from lecture on 3D-3D transformations)

$${}^{c}\mathbf{t}_{morg} = -{}^{c}_{m}\mathbf{R}\ {}^{m}\mathbf{t}_{corg}$$

i.e., the origin of the model frame with respect to the camera frame is the (rotated) negative of the origin of the camera frame with respect to the model frame

- So

$$\mathbf{M} = {}^{c}_{m}\mathbf{R} \begin{bmatrix} \mathbf{I}_{3x3} & -{}^{m}\mathbf{t}_{corg} \end{bmatrix}$$

Extracting translation (continued)

- Now if we multiply **M** by the vector representing the camera origin with respect to the model, we get zero:

$$\mathbf{M}\begin{pmatrix} {}^{m}\mathbf{t}_{corg} \\ 1 \end{pmatrix} = {}^{c}_{m}\mathbf{R}\begin{bmatrix}\mathbf{I}_{3x3} & -{}^{m}\mathbf{t}_{corg}\end{bmatrix}\begin{pmatrix} {}^{m}\mathbf{t}_{corg} \\ 1 \end{pmatrix} = \mathbf{0}$$

- So solve the system MX=0; then scale the result so that the 4th element = 1

```
% We can find the camera center, tcorg_m by solving the equation MX=0.
% To see this, write M = [R_m_c tmorg_c].  But tmorg_c = -R_m_c * tcorg_m.
% So M = R_m_c*[ I  -tcorg_m ].  And if we multiply M times tcorg_m, we
% get   R_m_c*[ I  -tcorg_m ] * [tcorg_m; 1] = 0.
[U,D,V] = svd(M);
tcorg_m = V(:,end);      % Get last column of V
tcorg_m = tcorg_m / tcorg_m(4);      % Divide through by last element
```

Extracting the rotation

- The leftmost 3x3 portion of **M** represents the rotation

$$\mathbf{M} = \begin{bmatrix} {}^{c}_{m}\mathbf{R} & {}^{c}\mathbf{t}_{morg} \end{bmatrix}$$

- However, that 3x3 submatrix of **M** (as estimated) may not be a valid rotation matrix:
  - A valid rotation matrix is orthonormal (ie its rows and columns are unit vectors and are orthogonal to each other)
  - A valid rotation matrix has determinant = +1 (i.e., it is a right-handed coordinate system)

- To get a valid rotation matrix, we will do “QR” decomposition

QR Decomposition

- Any real square matrix **A** may be decomposed as **A** = **QR**, where
  - **Q** is an orthonormal matrix
  - **R** is an upper triangular matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} q_{11} & q_{12} & q_{13} \\ q_{21} & q_{22} & q_{23} \\ q_{31} & q_{32} & q_{33} \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \end{pmatrix}$$

A **Q** **R**

- Note the unfortunate clash of terminology ... we have been using "**R**" to represent a rotation matrix. To avoid this, let's use **B** to represent the triangular matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} q_{11} & q_{12} & q_{13} \\ q_{21} & q_{22} & q_{23} \\ q_{31} & q_{32} & q_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ 0 & b_{22} & b_{23} \\ 0 & 0 & b_{33} \end{pmatrix}$$

A = QB

A **Q** **B**

Extracting the rotation

- Assume that the leftmost 3x3 portion of M is the rotation, but multiplied by some scaling matrix (this could be the intrinsic camera parameter matrix)

$$\mathbf{M}_{1:3,1:3} = \mathbf{KR}$$

- The transpose is

$$\left(\mathbf{M}_{1:3,1:3}\right)^T = \left(\mathbf{KR}\right)^T = \mathbf{R}^T\mathbf{K}^T$$

- We take the “QR” decomposition to get

$$\mathbf{R}^T\mathbf{K}^T = \mathbf{QB}$$

- So “**Q**” is the transpose of the rotation matrix that we want

DLT for Motion Capture

- We can use the DLT method for tracking markers for motion capture applications (sports, animation)

- Approach:
  - Set up a calibration grid with known target points
  - Determine the camera projection matrices for multiple cameras

- Run time
  - Each marker must be seen by more than one camera
  - Each marker's 3D position can be reconstructed from the corresponding image points

A dancer wearing a suit used in an optical motion capture system (from Wikepedia article on motion capture)

DLT for Reconstruction

- Recall that the image point projection of a target marker is

$$x = \frac{r_{11}X + r_{12}Y + r_{13}Z + t_x}{r_{31}X + r_{32}Y + r_{33}Z + t_z}, \quad y = \frac{r_{21}X + r_{22}Y + r_{23}Z + t_y}{r_{31}X + r_{32}Y + r_{33}Z + t_z}$$

- or

$$r_{11}X + r_{12}Y + r_{13}Z + t_x - x\left(r_{31}X + r_{32}Y + r_{33}Z + t_z\right) = 0$$
$$r_{21}X + r_{22}Y + r_{23}Z + t_y - y\left(r_{31}X + r_{32}Y + r_{33}Z + t_z\right) = 0$$

- Now, the (X,Y,Z) of the marker point is unknown and everything else is known
- Again, rearrange to put into the form **A x** = 0 where **x** = (X,Y,Z) and solve for **x**
- Note that we will need multiple cameras (how many?)
- Note that each camera has its own parameters (r_{11}, r_{12}, …, t_y, t_z)