CZECH TECHNICAL UNIVERSITY IN PRAGUE

LAR 2023, Depth Estimation

Vladimír Petrík

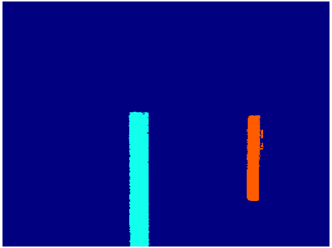
vladimir.petrik@cvut.cz

March 14, 2023

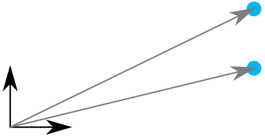
Problem Formulation

- Goal: Compute position of obstacles in **Cartesian** coordinates / Task Space
- Inputs:
  - RGB image with segmentation/labeling (see previous lecture)
  - Depth map
  - Robot odometry (integrated measurements of wheels rotation)

(a) RGB image



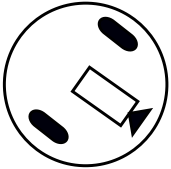
(b) Segmentation



(c) Position of obstacle

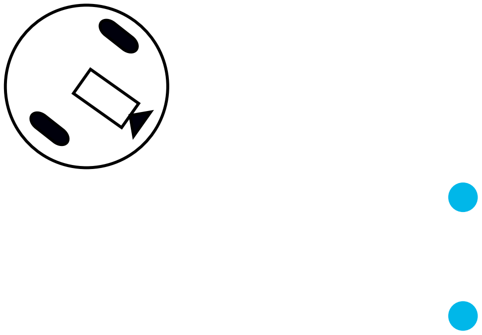
Coordinate frames

- robot is equipped with RGBD camera



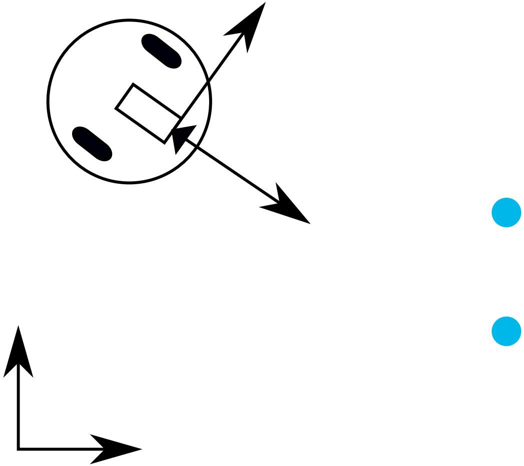
Coordinate frames

- robot is equipped with RGBD camera
- camera sees the obstacles



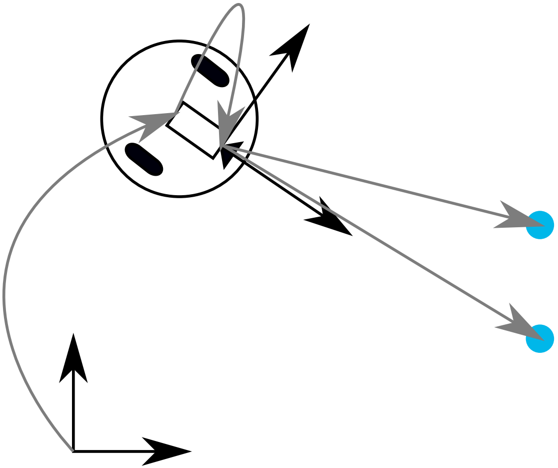
Coordinate frames

- robot is equipped with RGBD camera
- camera sees the obstacles
- multiple coordinate frames



Coordinate frames

- robot is equipped with RGBD camera
- camera sees the obstacles
- multiple coordinate frames
- transformations:
  - robot has moved from the initial position (T_o)
  - camera is not mounted exactly in the middle of robot (T_c)
  - obstacles are at position $\boldsymbol{x}_1, \boldsymbol{x}_2$ w.r.t. camera frame



Transformations

- Transformation in 2D can be represented by 3×3 matrix (in homogeneous coordinates)
- $T = \begin{pmatrix} R(\theta) & & x \\ & & y \\ 0 & 0 & 1 \end{pmatrix}$, $R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$

Transformations

- Transformation in 2D can be represented by 3×3 matrix (in homogeneous coordinates)
- $T = \begin{pmatrix} R(\theta) & & x \\ & & y \\ 0 & 0 & 1 \end{pmatrix}, \ R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$
- For our coordinates: $\boldsymbol{x}_w = T_o T_c \boldsymbol{x}_c$
  - $\boldsymbol{x}_w$ position of gate in world coordinate system
  - $\boldsymbol{x}_c$ position of gate in camera coordinate system
  - T_o computed from odometry data
  - T_c **approximated** by unit transformation
    - $\theta = 0$, $x = 0$, $y = 0$
    - optionally can be calibrated

Odometry Computation

- You define where the world coordinate is placed by resetting the odometry
- Robot computes relative wheels rotation and integrate it to obtain position w.r.t. call of reset
- Integration is **not robust**, i.e. the errors are integrated too

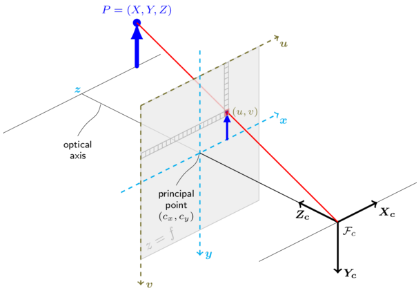
```
reset_odometry() -> None # sets world coordinate to the
# current robot position
get_odometry() -> [x,y,a]  # gives relative distance travelled from
# the last call of reset
```

Gate Position in Camera Frame

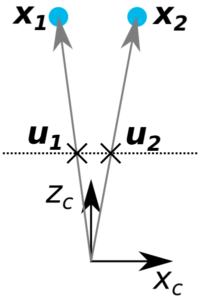
- We will compute gate positions in camera frame, hereinafter
- It simplifies some of the equations
- You can then transform them into world coordinates using: $\boldsymbol{x}_w = T_o T_c \boldsymbol{x}_c$

Camera Model

- camera is approximated by pinhole camera model
  - all points on a ray project to the same pixel
  - from given pixel, you cannot compute Cartesian point (without additional prior knowledge)



(a) Projection of point[1]



(b) Top view

[1] https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html

Pinhole Camera Model

- $\boldsymbol{u}_H = K\boldsymbol{x}$
  - $\boldsymbol{u}_H$ is pixel in homogeneous coordinates
  - if $\boldsymbol{u}_H = \begin{pmatrix} u_H & v_H & w_H \end{pmatrix}^\top$, then pixel coordinates are $\begin{pmatrix} u_H/w_H & v_H/w_H \end{pmatrix}^\top$

Pinhole Camera Model

- $\boldsymbol{u}_H = K\boldsymbol{x}$
  - $\boldsymbol{u}_H$ is pixel in homogeneous coordinates
  - if $\boldsymbol{u}_H = \begin{pmatrix} u_H & v_H & w_H \end{pmatrix}^\top$, then pixel coordinates are $\begin{pmatrix} u_H/w_H & v_H/w_H \end{pmatrix}^\top$
  - alternatively, we can represent it as: $\lambda \left(u, v, 1\right)^\top = \lambda \boldsymbol{u} = K\boldsymbol{x}$

Pinhole Camera Model

- $\boldsymbol{u}_H = K\boldsymbol{x}$
  - $\boldsymbol{u}_H$ is pixel in homogeneous coordinates
  - if $\boldsymbol{u}_H = \begin{pmatrix} u_H & v_H & w_H \end{pmatrix}^\top$, then pixel coordinates are $\begin{pmatrix} u_H/w_H & v_H/w_H \end{pmatrix}^\top$
  - alternatively, we can represent it as: $\lambda \left(u, v, 1\right)^\top = \lambda\boldsymbol{u} = K\boldsymbol{x}$
- K is camera matrix
  - `get_rgb_K(self) -> K`
  - $K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$

Pinhole Camera Model

- $\boldsymbol{u}_H = K\boldsymbol{x}$
  - $\boldsymbol{u}_H$ is pixel in homogeneous coordinates
  - if $\boldsymbol{u}_H = \begin{pmatrix} u_H & v_H & w_H \end{pmatrix}^\top$, then pixel coordinates are $\begin{pmatrix} u_H/w_H & v_H/w_H \end{pmatrix}^\top$
  - alternatively, we can represent it as: $\lambda \left(u, v, 1\right)^\top = \lambda\boldsymbol{u} = K\boldsymbol{x}$
- K is camera matrix
  - `get_rgb_K(self) -> K`
  - $K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$
  - what does λ represent?

Pinhole Camera Model

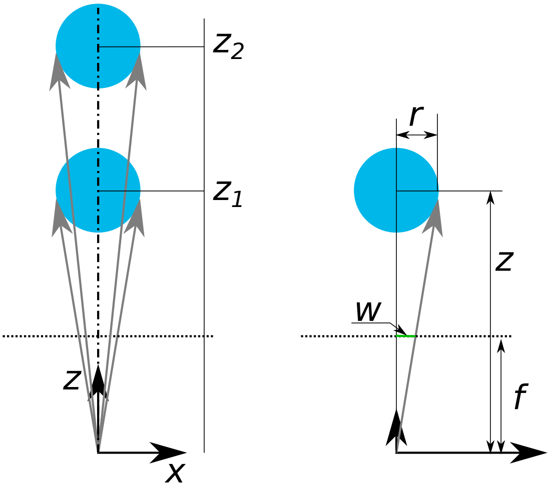
- $\boldsymbol{u}_H = K\boldsymbol{x}$
  - $\boldsymbol{u}_H$ is pixel in homogeneous coordinates
  - if $\boldsymbol{u}_H = \begin{pmatrix} u_H & v_H & w_H \end{pmatrix}^\top$, then pixel coordinates are $\begin{pmatrix} u_H/w_H & v_H/w_H \end{pmatrix}^\top$
  - alternatively, we can represent it as: $\lambda \left(u, v, 1\right)^\top = \lambda\boldsymbol{u} = K\boldsymbol{x}$
- K is camera matrix
  - `get_rgb_K(self) -> K`
  - $K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$
  - what does λ represent?
    - λ is non-zero real number
    - if you know λ value, you can compute Cartesian coordinate $\boldsymbol{x} = \lambda K^{-1}\boldsymbol{u}$
    - otherwise, only ray is computable

How to Get Depth Information?

- We need either prior knowledge of the scene or depth map
- Example of prior knowledge
  - width of the obstacle in pixels and corresponding z-coordinate for several positions
  - width of the obstacle in meters
  - height of the obstacle
  - etc.

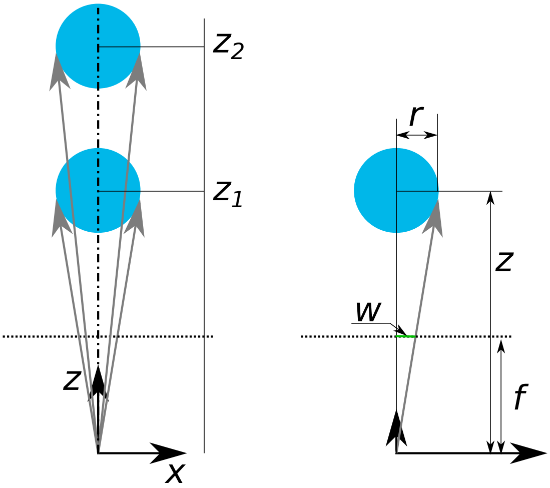
Using Regression

- what is relation between width in the image (px) and distance in meters?



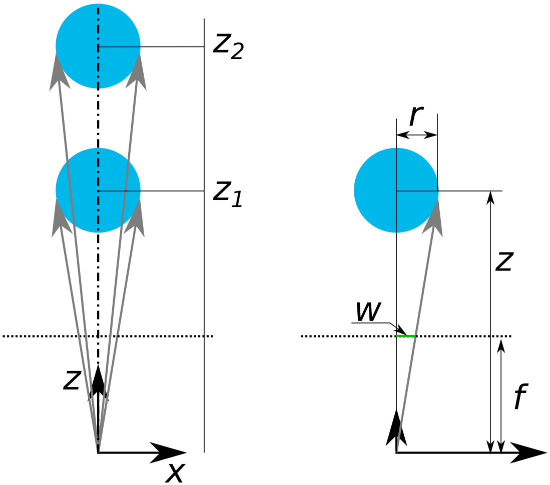
Using Regression

- what is relation between width in the image (px) and distance in meters?
  - $f : w = z : r$
  - $z = rf\frac{1}{w} = k\frac{1}{w}$



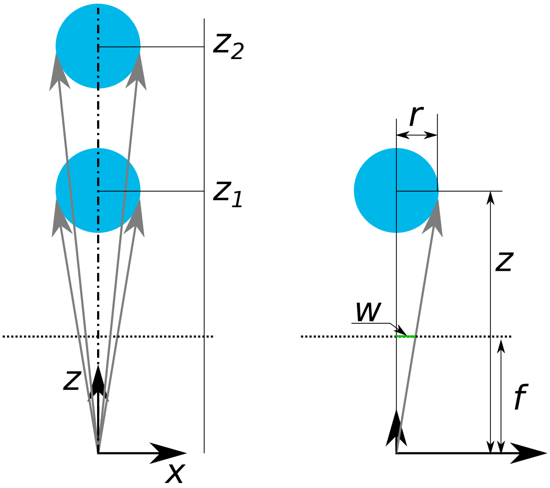
Using Regression

- what is relation between width in the image (px) and distance in meters?
  - $f : w = z : r$
  - $z = rf\frac{1}{w} = k\frac{1}{w}$
- How to estimate unknown constant?
  - calibration
  - measure (at least) two different positions
  - use least square estimation



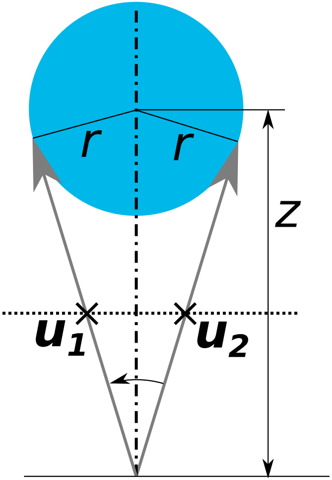
Using Regression

- what is relation between width in the image (px) and distance in meters?
  - $f : w = z : r$
  - $z = rf\frac{1}{w} = k\frac{1}{w}$
- How to estimate unknown constant?
  - calibration
  - measure (at least) two different positions
  - use least square estimation
- This is an approximated computation (ignoring viewing angle)



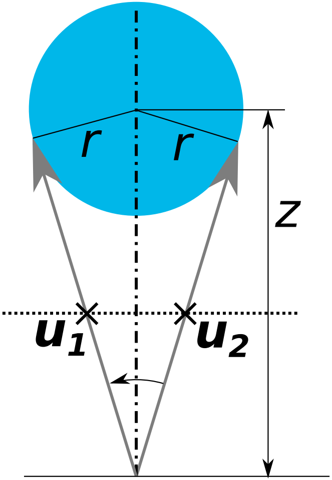
Using Prior Knowledge of Fixed Width

- We know radius of gate is fixed



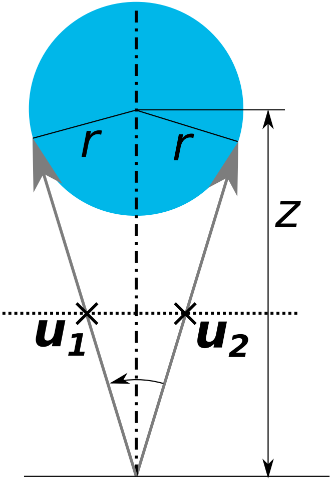
Using Prior Knowledge of Fixed Width

- We know radius of gate is fixed
- From detected pixels $\boldsymbol{u}_1, \boldsymbol{u}_2$, we can compute rays $\boldsymbol{x}_1, \boldsymbol{x}_2$: $\frac{1}{\lambda_i}\boldsymbol{x}_i = K^{-1}\boldsymbol{u}_i$



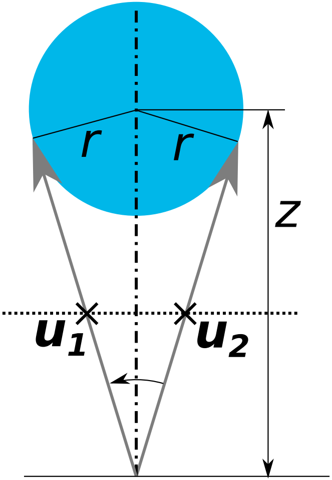
Using Prior Knowledge of Fixed Width

- We know radius of gate is fixed
- From detected pixels $\boldsymbol{u}_1, \boldsymbol{u}_2$, we can compute rays $\boldsymbol{x}_1, \boldsymbol{x}_2$:
  $\frac{1}{\lambda_i}\boldsymbol{x}_i = K^{-1}\boldsymbol{u}_i$
- Angle between vectors: $\cos\alpha = \frac{\frac{1}{\lambda_1\lambda_2}}{\frac{1}{\lambda_1\lambda_2}} \frac{\boldsymbol{x}_1 \cdot \boldsymbol{x}_2}{\|\boldsymbol{x}_1\|\|\boldsymbol{x}_2\|}$



Using Prior Knowledge of Fixed Width

- We know radius of gate is fixed
- From detected pixels $\boldsymbol{u}_1, \boldsymbol{u}_2$, we can compute rays $\boldsymbol{x}_1, \boldsymbol{x}_2$: $\frac{1}{\lambda_i}\boldsymbol{x}_i = K^{-1}\boldsymbol{u}_i$
- Angle between vectors: $\cos\alpha = \frac{\frac{1}{\lambda_1\lambda_2}}{\frac{1}{\lambda_1\lambda_2}} \frac{\boldsymbol{x}_1 \cdot \boldsymbol{x}_2}{\|\boldsymbol{x}_1\| \|\boldsymbol{x}_2\|}$
- Depth: $z = \frac{r}{\sin(\alpha/2)}$



Using Depth Sensor

- Turtlebots are equipped with RGB**D** sensors
- In addition to RGB image they provide depth information
- `get_depth_image() -> numpy array size depends on the sensor`
- Depth corresponds to distance in meters (x, y need to be computed from ray)

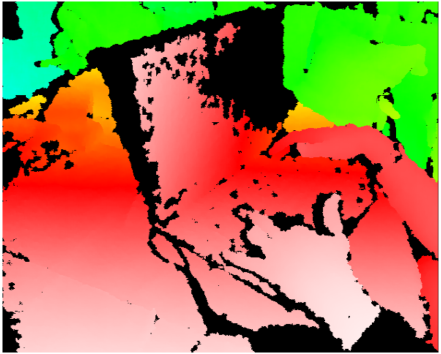
(a) RGB (b) Depth

Point Cloud

- Our library:
  - We also provide point cloud with topology
  - `get_point_cloud()` `numpy 480x640x3`
  - Array has the same dimensions as an RGB image
  - Channels correspond to x, y, z-coordinates in camera frame
- In general:
  - Point clouds are without topology
  - Set of points

Troubles with Depth Maps and Point Clouds

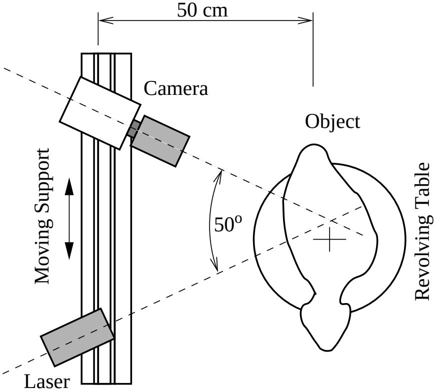
- Depth reconstruction is not perfect (black areas in the image[2])
- In python represented by NaN
- Not every pixel in RGB has reconstructed depth value
- RGB and Depth data are not aligned (you need to calibrate them)

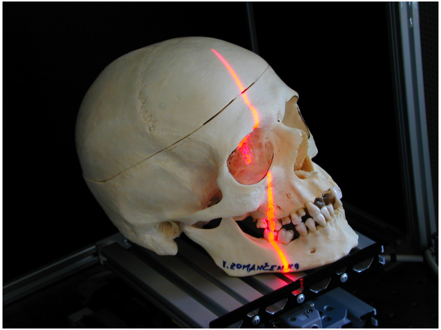



[2] https://commons.wikimedia.org, User:Kolossos

How Depth Sensors Work

- Laser projects pattern and camera recognizes it
- Depth information is computed using triangulation





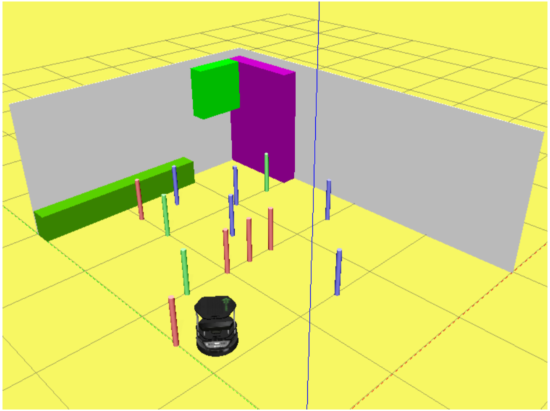
Kinect/Astra/Realsense

- Structured light based sensors
- Projects 2d infra red patterns
- There is one projector and two cameras (RGB + IR)

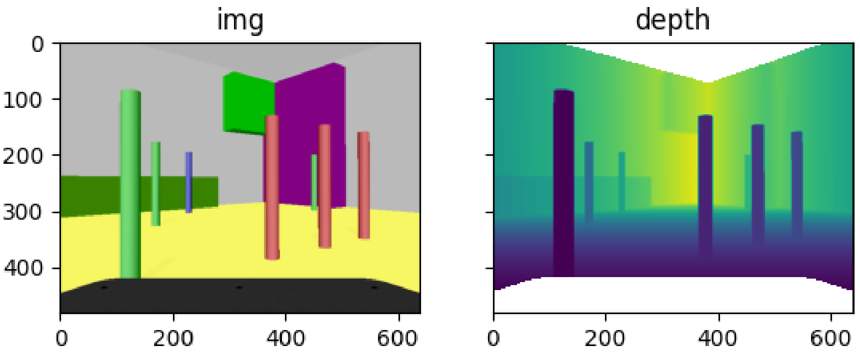
Comparison of Sensors

	 Kinect Xbox 360	 Orbbec Astra	 Realsense R200	 Realsense D435
FOV [deg]:	57 x 45	60 x 49.5	59 x 45.5	69.4 x 42.5
Range [m]:	1.5 . . . 3.5	0.6 . . . 8.0	0.5 . . . 3.5 (4.0)	0.105 . . . 10
Error XY [mm]:	10 (2.5m)	7.2 (3m)	—	–
Error Z [mm]:	10 (2.5m)	12.7 (3m)	10 (2m)	–
Resolution [px]:	640x480	640x480	640x480	1280x720

Our scene

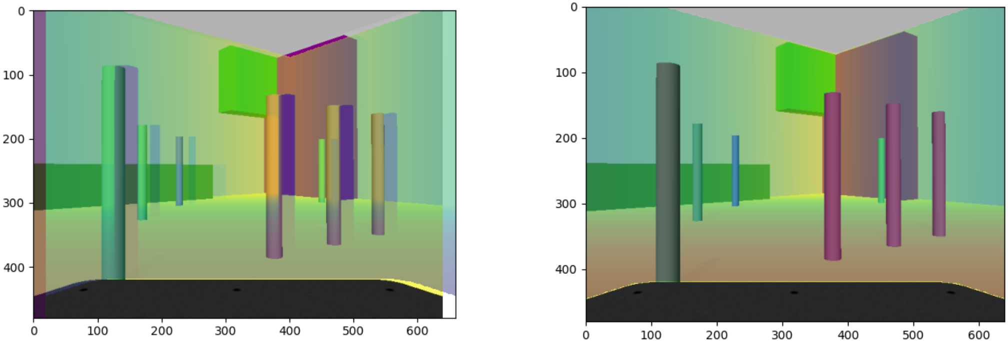


Our RGBD data



- Sensor range is limited - NaNs for too close and too far away points.

Are RGB/DEPTH aligned?



(a) In reality without calibration

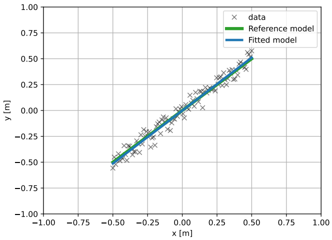
(b) In simulation

Figure: Overlay of DEPTH data over the RGB image.

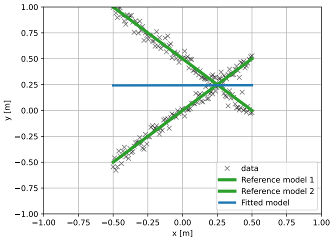
Working with noisy data

- We can fit geometry primitives to our observations
  - Observations are noisy
  - Contains outliers and multiple geometries
- Non-linear least square fitting
  - Using SciPy:

```
def line_model(x, slope, bias):
    return x * slope + bias
(best_slope, best_bias), _ = curve_fit(line_model, xdata, ydata)
```



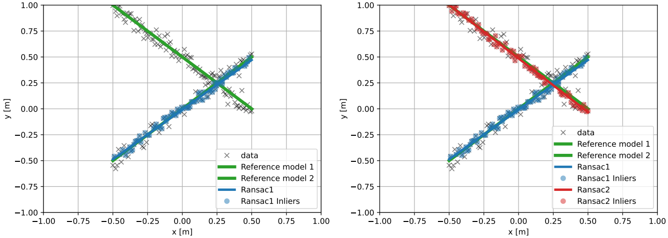
(a) Single geometry



(b) Multiple geometries

RANSAC

- Random sample consensus
- Iterative fitting method robust to outliers
  - Choose a small subset of data points
  - Fit a model to the subset
  - Count number of inliers - (what is inlier?)
  - Repeat many times and select the best model



(a) RANSAC - First fit (b) RANSAC - Second fit