Image-Based Situation Awareness Audit 28.2.2018

Sakari Lampola

Previous Audit 11.1.2018

Previous Audit

Open questions:

- Role of classical object tracking alrorithms?
- What to do with multiple bounding boxes around one object?
- Appropriate minimum confidence level?
- What to do with false detections inside other objects?
- What to do with false detections from the background?
- How to set Kalman filter parameters for image object filtering?
- Hungarian algorithms special case for hidden objects

To do:

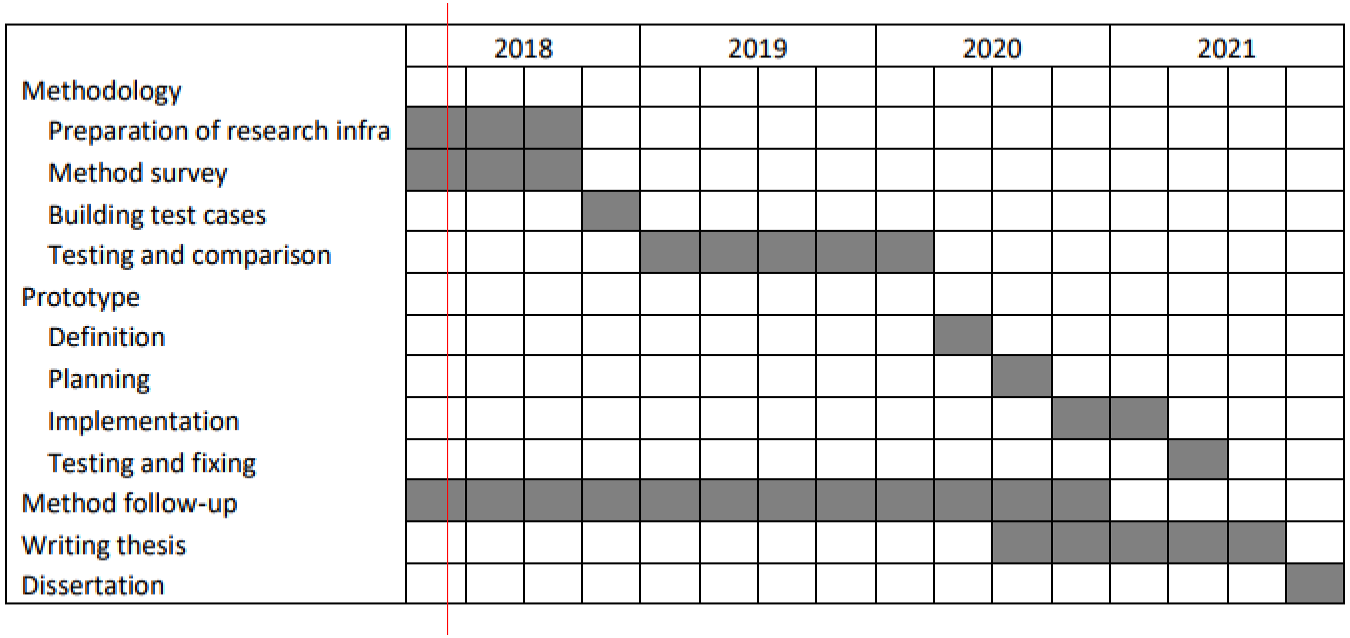
- Close open questions
- Image object status
- Image object velocity estimation
- Probabilistic approach for matching detected and image objects
- 2d -> 3d transformation
- World object state estimation

Other:

- Semantic segmentation
- Organisations to follow: ICCV, ICRA, NIPS, IROS, arXiv
- Camera motion (yaw, pitch)
- Grid or continuos presentation?
- Class specific attributes
- Object history

Project Plan

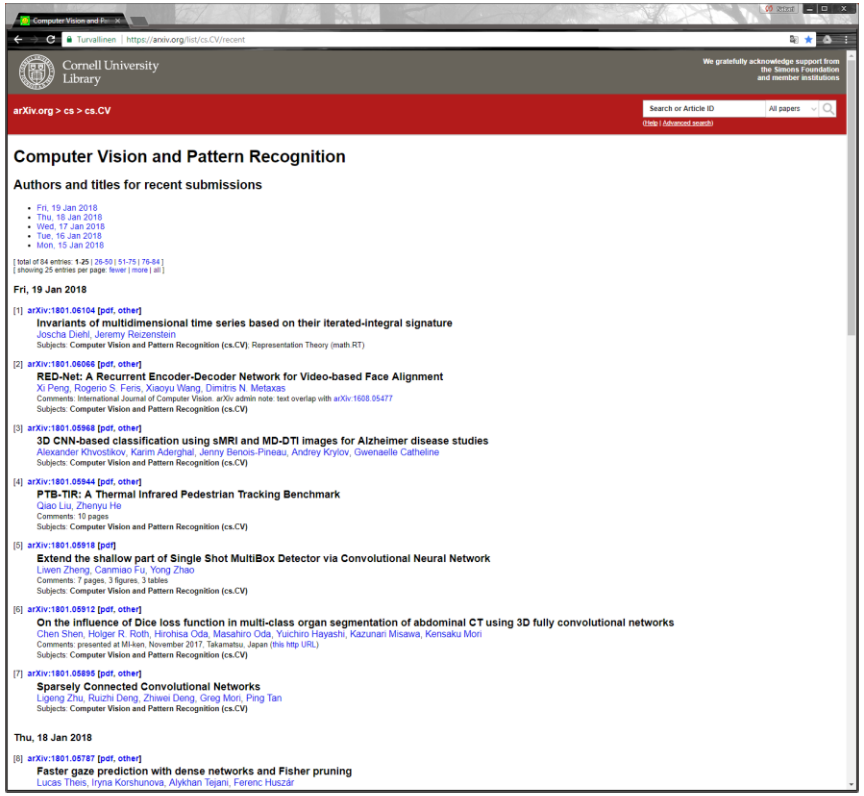
Project Plan



1. Methodology / Preparation of research infra
   a. Software platforms are constructed and tested
   b. Off-the-shelf models are acquired and tested
   c. Necessary skills on platforms are learned
2. Methodology / Method survey
   a. Current state-of-art methods are studied
   b. Methods are constructed and tested on the software platforms
3. Method follow-up
   a. Screening of conference papers related to the subject
   b. Possibly integrating new methods to the project

Work Done

Method Follow-Up



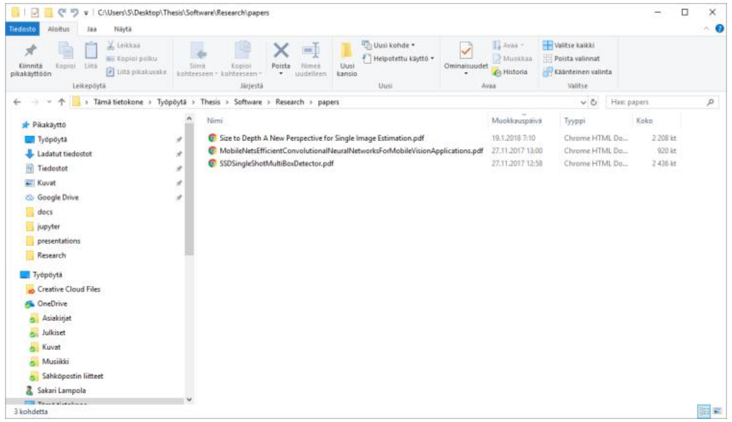


Image Object Velocity Estimation

Image object velocity is necessary for:

- predicting image object locations when matching new measurements
- identifying image objects
- predicting image object locations for hidden objects

Image object

- id
- status
- x_min
- x_max
- y_min
- y_max
- vx_min
- vx_max
- vy_min
- vy_max
- class
- confidence
- appearance

Estimation algorithm

Image Object Kalman Filtering

Bounding box corner location

State vector s:

$$s = \begin{bmatrix} l \\ v \end{bmatrix}$$

where

l = location coordinate (x_{min}, x_{max}, y_{min}, y_{max}) of the bounding box corner in the image
v = velocity (vx_{min}, vx_{max}, vy_{min}, vy_{max}) of the bounding box corner in the image

State equation in differential form:

$$\frac{ds(t)}{dt} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} * s(t) + \epsilon(t) = A_1 * s$$

State equation in difference form:

$$s(k+1) = (I + \Delta * A_1) * s(k) + \epsilon(k)$$

$$= \begin{bmatrix} 1 & \Delta \\ 0 & 1 \end{bmatrix} * s(k) + \varepsilon(k) = A * s(k) + \varepsilon(k)$$

where Δ is the time increment and ε Gaussian noise with co-variance R.

Measurement equation

$$z(k) = \begin{bmatrix} 1 & 0 \end{bmatrix} * s(k) + \delta(k) = C * s(k) + \delta(k)$$

Where δ is Gaussian noise with covariance matrix Q.

Kalman filter initialization:

$$\mu(0) = \begin{bmatrix} l(0) \\ 0 \end{bmatrix}$$

where l(0) is the first location measurement.

$$\Sigma(0) = \begin{bmatrix} 10.0 & 0 \\ 0 & 10000.0 \end{bmatrix}$$

where 10.0 and 10000.0 are believed initial error variances of location and velocity.

$$R = \begin{bmatrix} 1.0 & 0 \\ 0 & 1.0 \end{bmatrix}$$

where diagonal elements are believed state equation vari-ances of location and velocity.

$$Q = [10.0]$$

Where 10.0 is the believed measurement variance.

Kalman filter update:

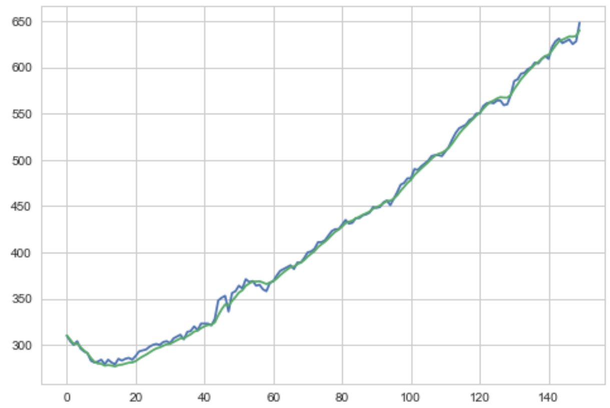
$$\mu_1(k) = A * \mu(k-1)$$
$$\Sigma_1(k) = A * \Sigma(k-1) * A^T + R$$
$$K(k) = \Sigma_1(k) * C^T * (C * \Sigma_1(k) * C^T + Q)^{-1}$$
$$\mu(k) = \mu_1(k) + K(k) * (z(k) - C * \mu_1(k))$$
$$\Sigma(k) = (I - K(k) * C) * \Sigma_1(k)$$

Asiakirjan loppu

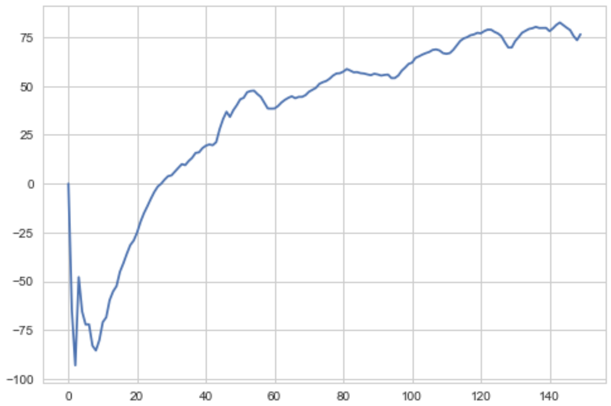
Numerical values are estimated using grid search and 10 step ahead mean prediction error. Values rounded. Later adjusted by experiments.

Image Object Velocity Estimation

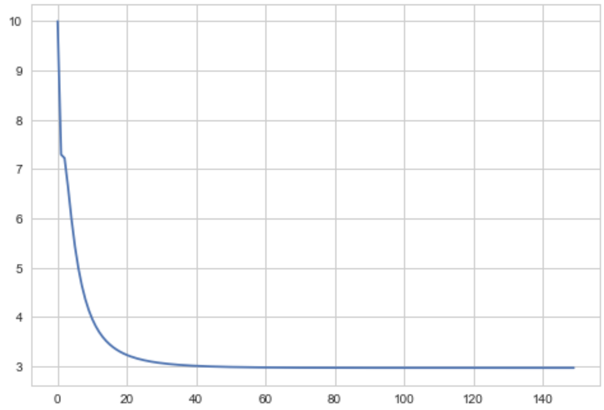
Moving object (car)



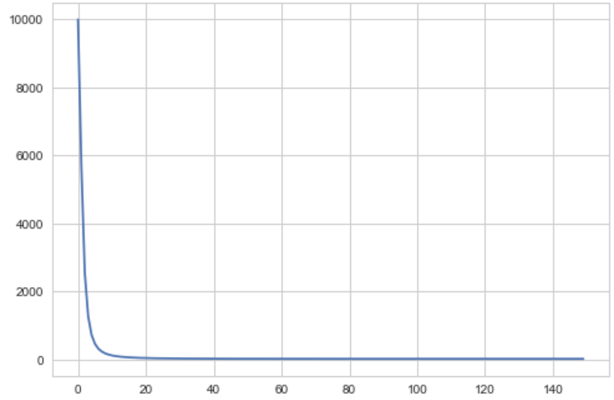
Measured and filtered location (upper left corner)



Estimated velocity



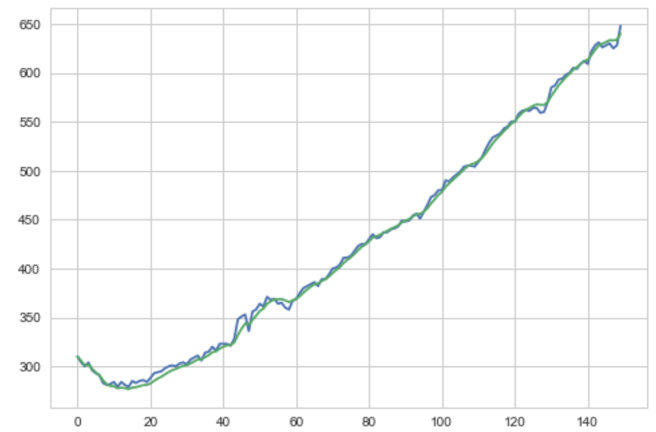
Location variance

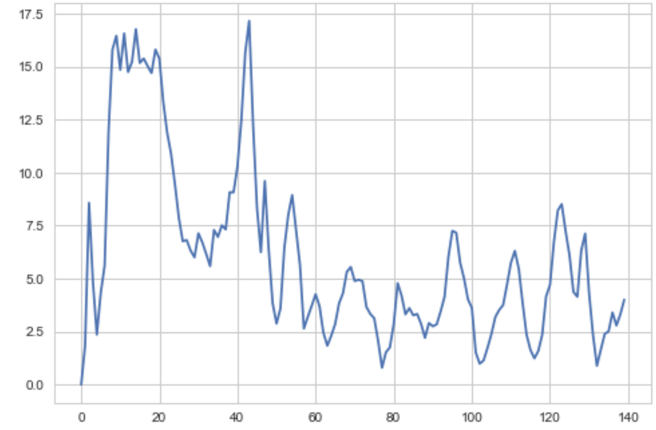


Velocity variance

Image Object Velocity Estimation

Moving object (car)

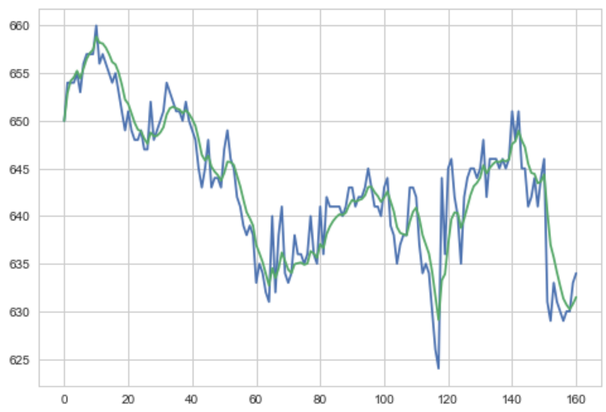




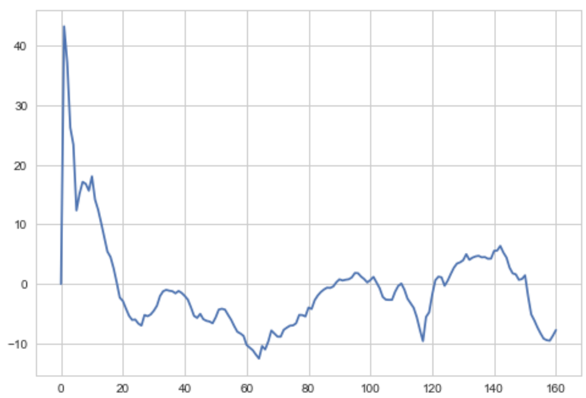
10 step ahead mean prediction error

Image Object Velocity Estimation

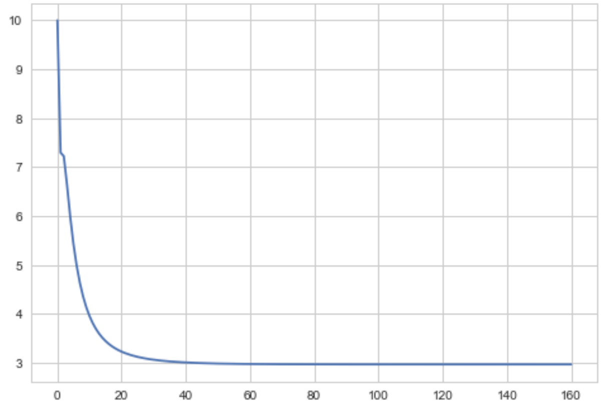
Static object (calf)



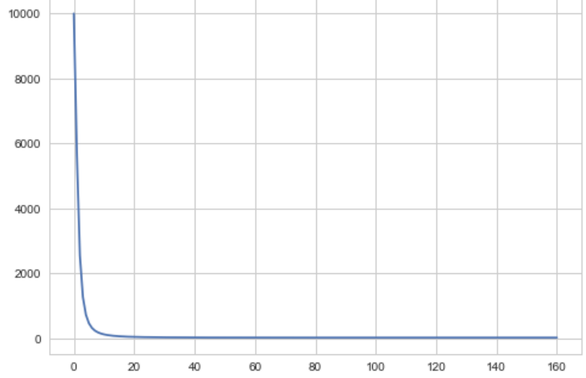
Measured and filtered location (upper left corner)



Estimated velocity



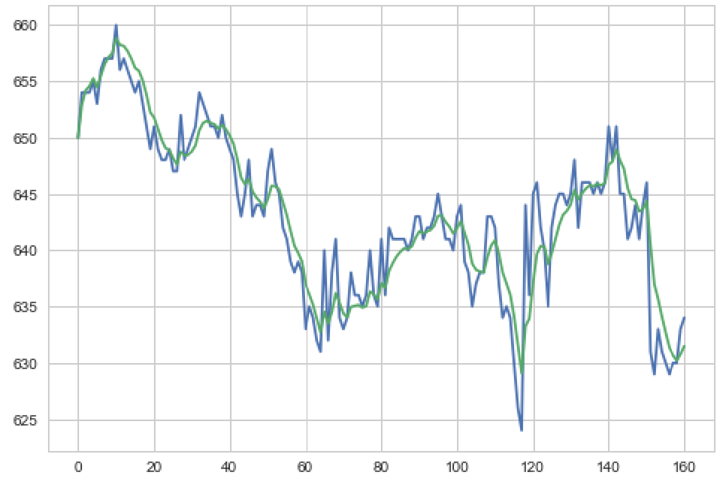
Location variance

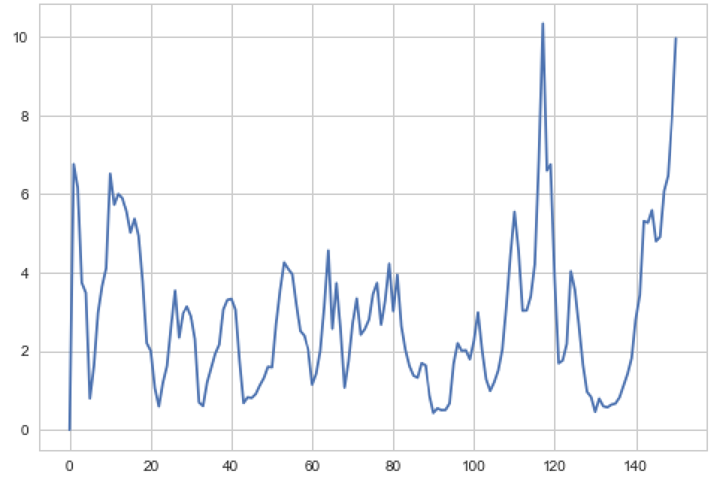


Velocity variance

Image Object Velocity Estimation

Static object (calf)

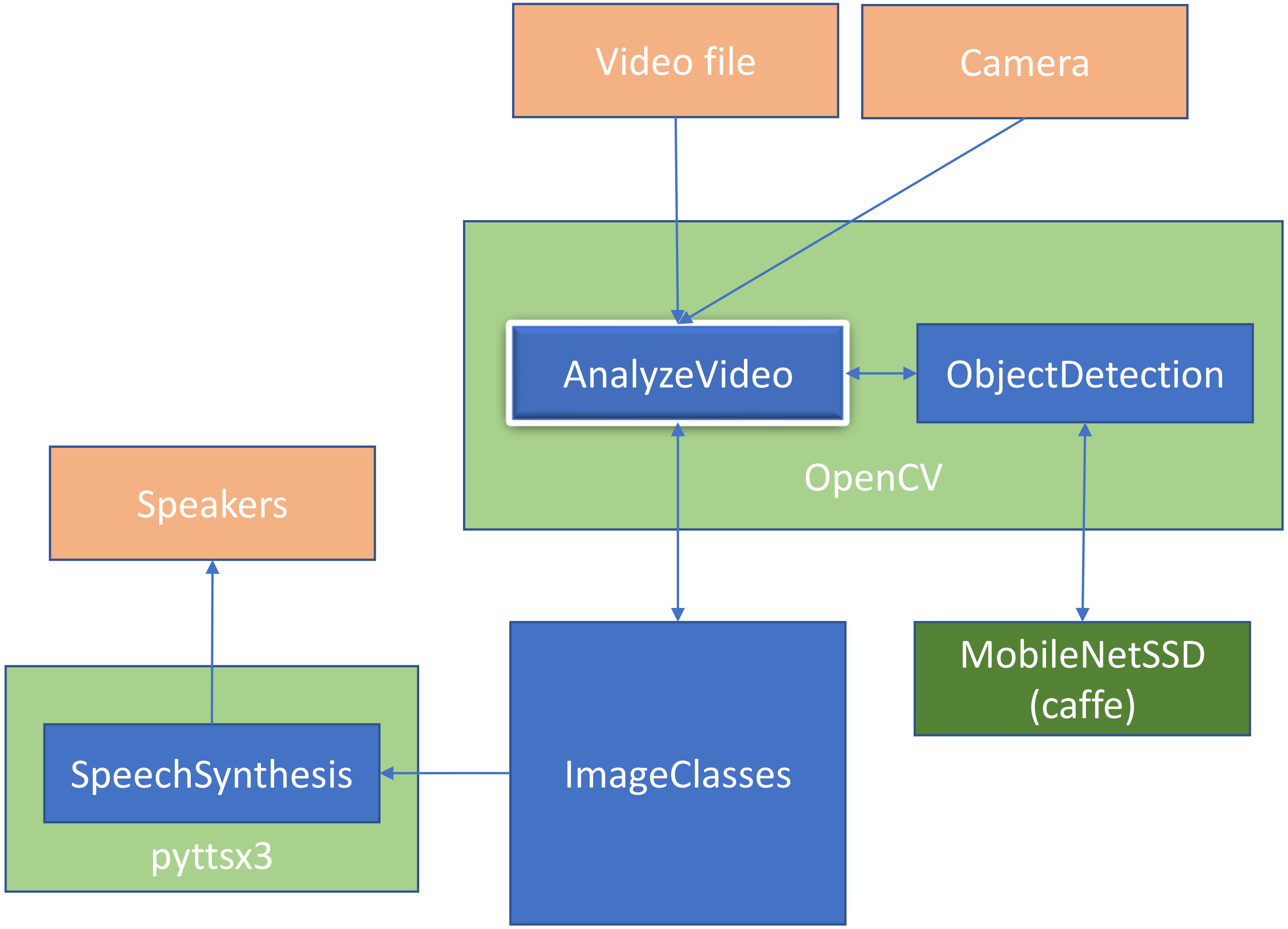




10 step ahead mean prediction error

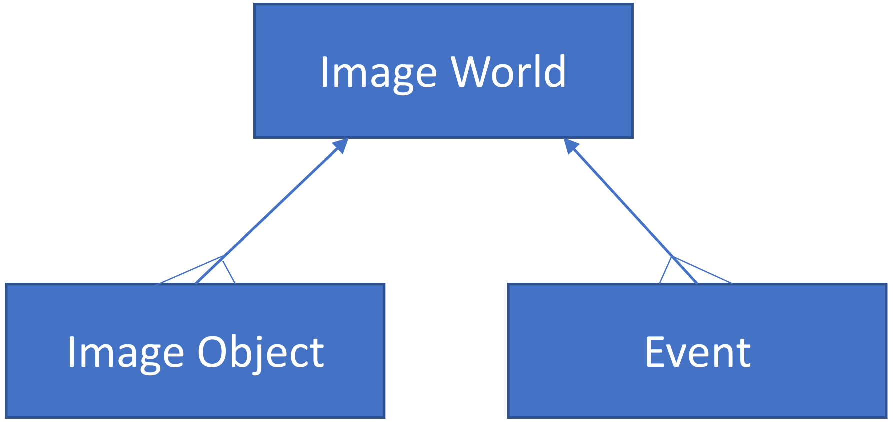
Speech Synthesis

Software Architecture



Speech Synthesis

Entities



Event
- time
- text
- priority

- Event is generated when
  - new image object is created
  - image object status is changed
- Event will pause the video for the duration of speech (not in the final version)
- Events are collected (history)

Confidence Level

SSD Mobilenet implementation:
extract the confidence (i.e., probability) associated with the prediction

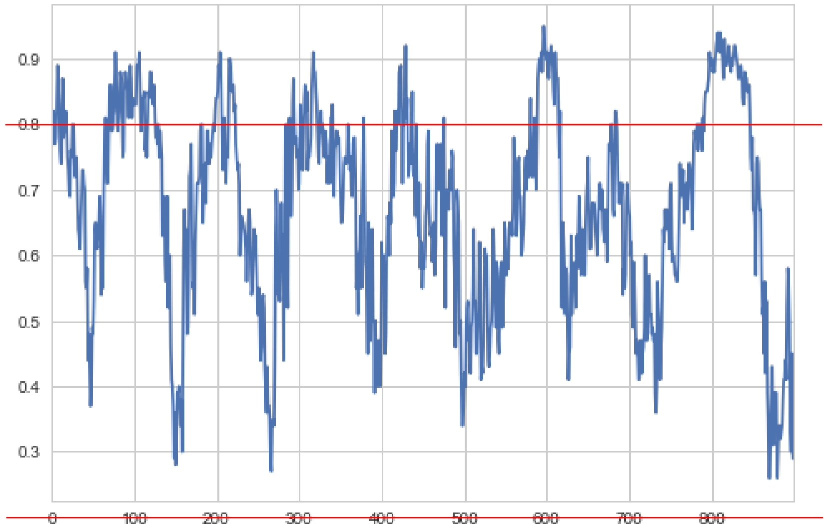
Objects detected		Confidence level							
Video	Correct	0,00	0,20	0,40	0,60	0,80	0,90	0,95	1,00
CarsOnHighway001.mpg	39	49	49	39	36	34	32	32	0
Calf-2679.mp4	1	2	2	2	2	1	1	1	0
Dunes-7238.mp4	1	7	7	6	5	2	2	2	0
Sofa-11294.mp4	1	2	2	1	1	1	1	1	0
Cars133.mp4	5	9	9	6	5	5	5	5	0
BlueTit2975.mp4	1	3	3	2	1	1	1	1	0
Railway-4106.mp4	1	10	10	5	3	3	1	1	0
Hiker1010.mp4	1	4	4	0	0	0	0	0	0
Cat-3740.mp4	1	3	3	2	2	1	1	1	0
SailingBoat6415.mp4	1	1	1	1	1	1	1	1	0
AWomanStandsOnTheSeashore-10058.mp4	1	1	1	1	1	1	1	1	0
Dog-4028.mp4	1	4	4	2	1	1	1	1	0
Boat-10876.mp4	1	2	2	1	1	1	1	0	0
Horse-2980.mp4	1	3	3	3	2	2	1	1	0
Sheep-12727.mp4	1	1	1	1	1	1	1	1	1

Good value for creating a new image object is between 0.8 and 0.9.

The ’good’ value also depends on other hyperparameters.

Confidence Level

Confidence level has dynamics



create

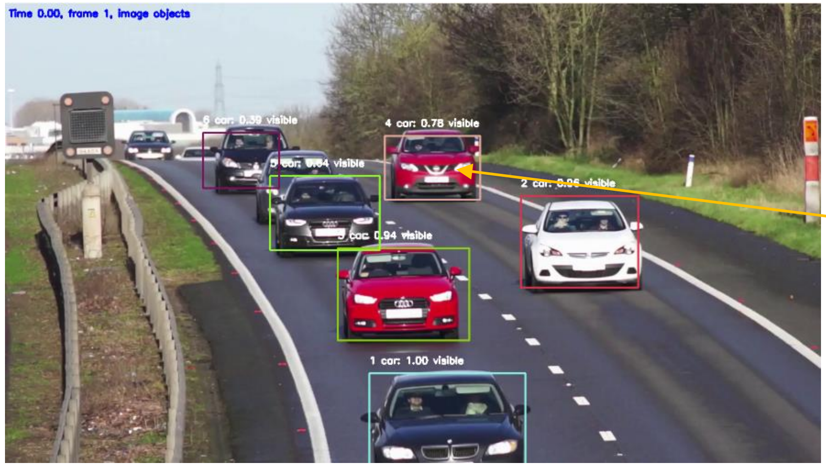
Update (not class)

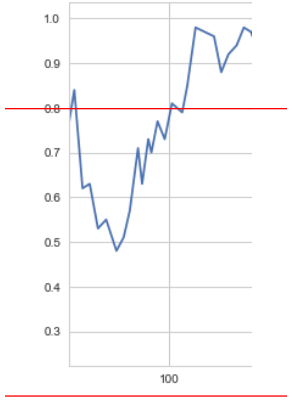
ignore

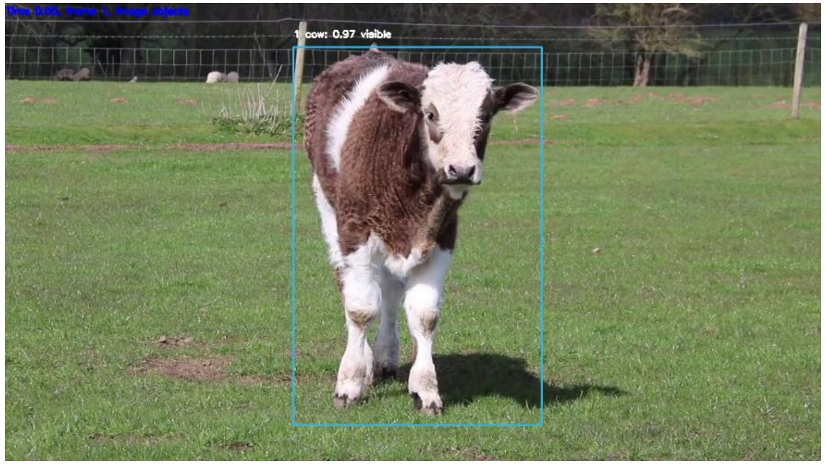
Different levels for creating and updating image object. Hyperparameters:

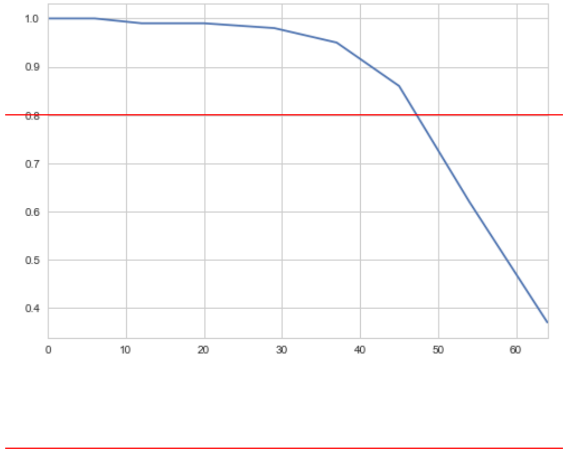
- CONFIDENCE_LEVEL_CREATE (0.8)
- CONFIDENCE_LEVEL_UPDATE (0.2)

Confidence Level

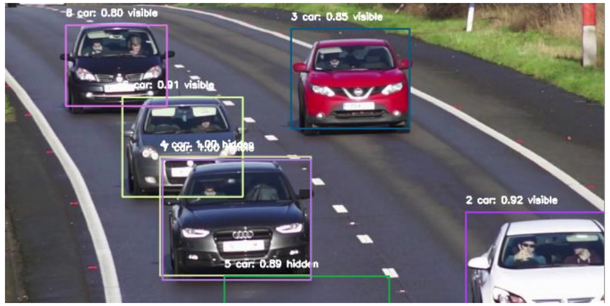








Border Behaviour



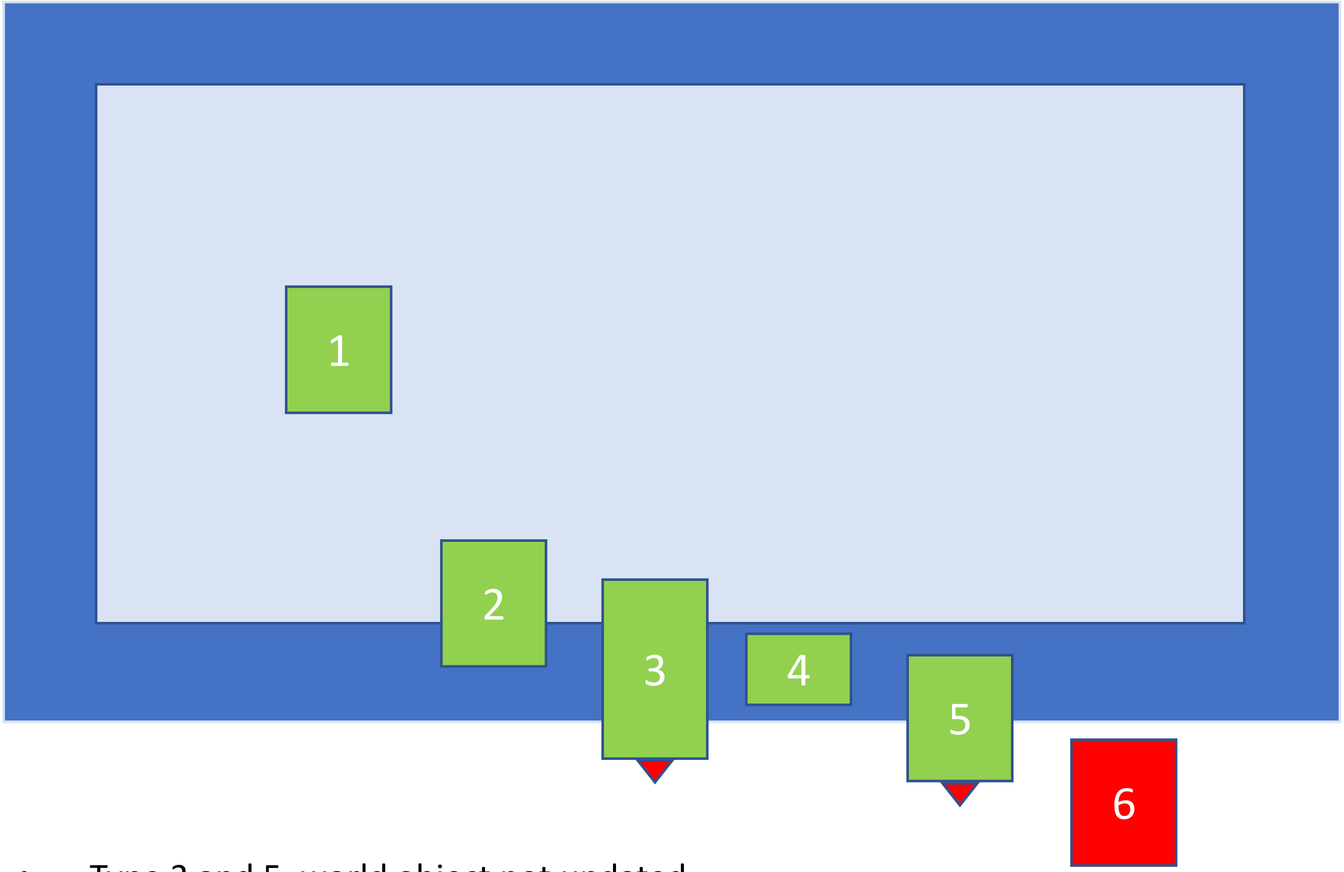
Box size and form distorded

time	x_max_c	x_max_m	x_max_p	y_max_c	y_max_m	y_max_p
1.48	1208.859	1209.0	1205.616	646.300	652.0	640.731
1.52	1221.500	1236.0	1212.044	653.697	656.0	649.501
1.56	1232.488	1242.0	1224.941	660.427	661.0	656.939
1.60	1241.599	1246.0	1236.095	668.758	673.0	663.679
1.64	1251.081	1256.0	1245.282	677.391	682.0	672.083
1.68	1258.430	1258.0	1254.848	687.143	694.0	680.794
1.72	1265.965	1266.0	1262.190	694.428	695.0	690.663
1.76	1272.740	1271.0	1269.725	704.340	711.0	697.956
1.80	1280.741	1282.0	1276.471	711.433	711.0	707.979
1.84	1287.573	1286.0	1284.493	717.291	714.0	715.066
1.88	1292.323	1286.0	1291.299	722.517	718.0	720.869
1.92	1292.517	1276.0	1295.946	728.172	725.0	726.022
1.96	1291.385	1273.0	1295.873	731.168	722.0	731.626
2.00	1291.974	1279.0	1294.445	732.465	720.0	734.474
2.04	1291.500	1277.0	1294.826	732.500	718.0	735.572
2.08	1290.547	1276.0	1294.121	733.994	724.0	735.375
2.12	1289.259	1275.0	1292.938	736.016	728.0	736.711
2.16	1289.533	1280.0	1291.424	736.959	727.0	738.606
2.20	1290.113	1282.0	1291.548	737.402	727.0	739.392
2.24	1290.640	1283.0	1292.000	735.994	722.0	739.671

```
In [10]: # image size 1280 * 720
```

Hyperparameter BORDER_WIDTH (30)

Border Behaviour

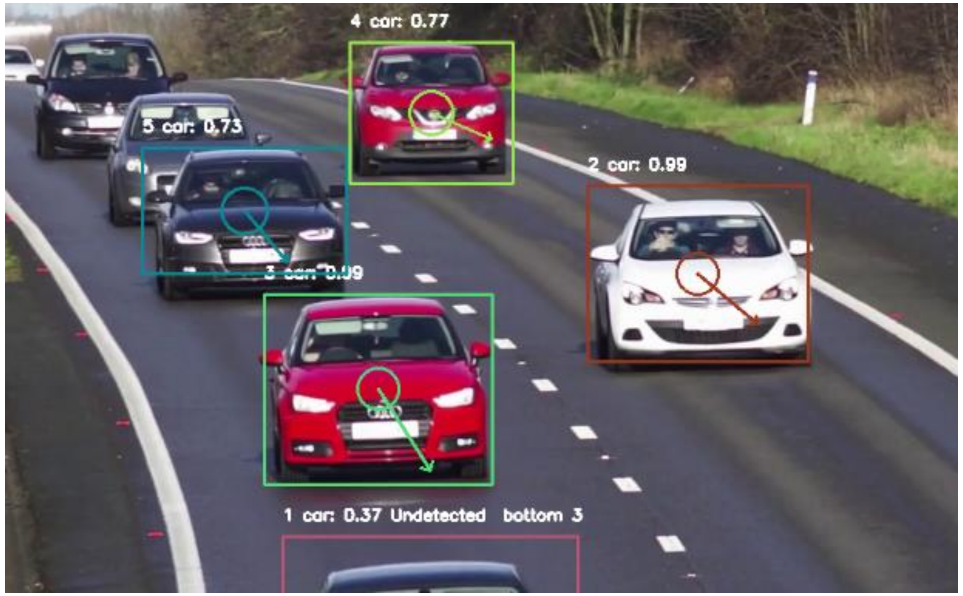


- Type 3 and 5: world object not updated
- Type 6: removed, world object acceleration fixed
- If an object touches 3 borders, it is removed

Done for:

- left
- right
- top
- bottom

Visual Presentation



- Ellipse axes proportional to the standard deviation of the location (2*std, corresponding to 95% probability)
- Arrow direction and length proportional to velocity (measured in pixels/second)

Object Retention

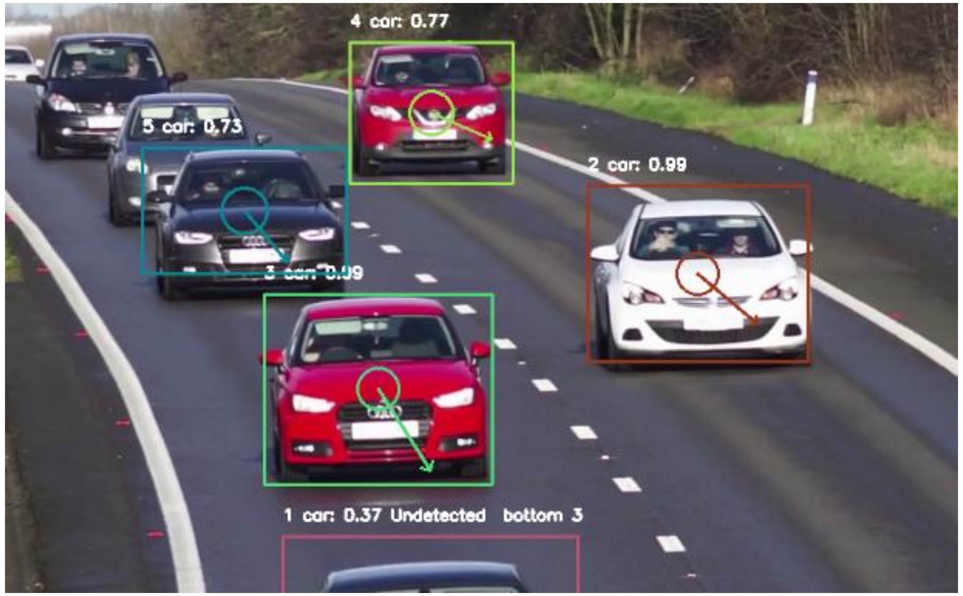
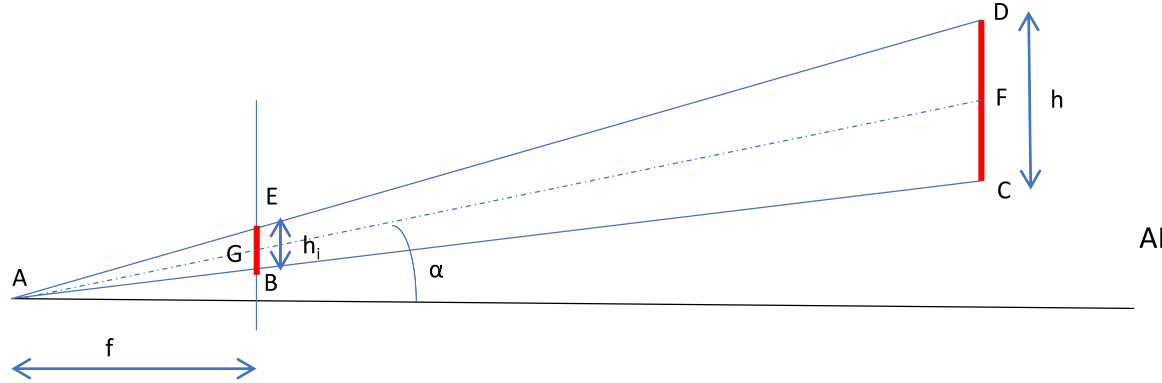


Image objects are removed if not detected in RETENTION_COUNT_MAX (30) successive frames.

Distance Estimation



Similar triangles AGE and AFD:

$$\frac{0.5 * h_i}{0.5 * h} = \frac{AG}{d} = \frac{\frac{f}{\cos(\alpha)}}{d} = \frac{f}{d * \cos(\alpha)}$$

$$\boxed{d = \frac{f * h}{\cos(\alpha) * h_i}}$$

Similar equations for horizontal direction (β=azimuth)

Distance Estimation

$$d = \frac{f * h}{\cos(\alpha) * \cos(\beta) * h_i} = \frac{f * h}{\cos(\alpha) * \cos(\beta) * h_i * s_h / p_h}$$

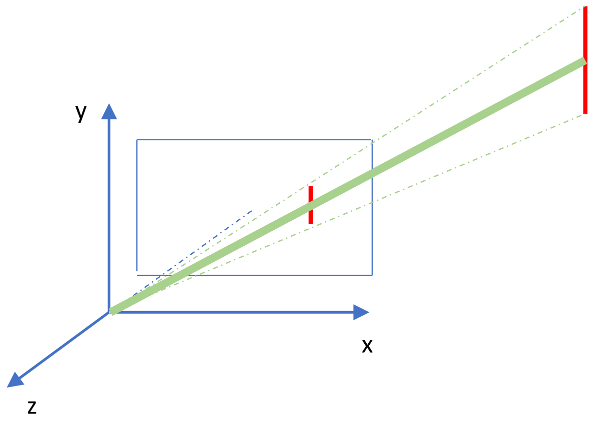
s_w= sensor width (m)
s_h= sensor height (m)
p_w= image width (pixels)
p_h= image height (pixels)
h_i = object height (pixels)
h = object height (m)
f = focal length (m)
α = altitude (rad)
β = azimuth (rad)

Example (Nikon D800E):

- s_w= sensor width (m) = 0.0359 m
- s_h= sensor height (m) = 0.0240 m
- p_w= image width (pixels) = 7360
- p_h= image height (pixels) = 4912
- h_i = object height (pixels) = 100
- h = object height (m) = 1.0 m
- f = focal length (m) = 0.050 m
- α = *altitude (rad) = 0.0*
- β = *azimuth (rad) = 0.0*

$$d = \frac{0.050m * 1m}{1.0 * 1.0 * 100 * 0.024m/4912} = 102.33\ m$$

2d -> 3d Transformation



From pixel coordinates (sensor plane) to 3d camera coordinates:

$$(x_c, y_c, z_c) = \left(-\frac{s_w}{2} + xp * \frac{s_w}{p_w},\ \frac{s_h}{2} - yp * \frac{s_h}{p_h},\ -f\right)$$

Object center will be on the line:

$$(x_o, y_o, z_o) = t * (x_c, y_c, z_c)$$

The length of the line is:

$$d = \frac{f * h}{\cos(\alpha) * \cos(\beta) * h_i * s_h/p_h}$$

$$\alpha = \text{arc tan}(y_c/f)$$
$$\beta = \text{arc tan}(x_c/f)$$

s_w= *sensor width (m)*
s_h= *sensor height (m)*
p_w= *image width (pixels)*
p_h= *image height (pixels)*
h_i = *object height (pixels)*
h = object height (m)
f = focal length (m)
α = altitude (rad)

2d -> 3d Transformation

$$t^2 * ({x_c}^2 + {y_c}^2 + {z_c}^2) = d^2$$

$$t = \frac{d}{\sqrt{{x_c}^2 + {y_c}^2 + {z_c}^2}}$$

2d -> 3d Transformation

Example:

- s_w= sensor width (m) = 0.0359 m
- s_h= sensor height (m) = 0.0240 m
- p_w= image width (pixels) = 7360
- p_h= image height (pixels) = 4912
- h_i = object height (pixels) = 100
- h = object height (m) = 1.0 m
- f = focal length (m) = 0.050 m
- x_p = 1200
- y_p = 2000

$$(x_c, y_c, z_c) = \left(-\frac{s_w}{2} + xp*\frac{s_w}{p_w},\ \frac{s_h}{2} - yp*\frac{s_h}{p_h},\ -f\right)$$

$$= \left(-\frac{0.0359}{2} + 1200*\frac{0.0359}{7360},\ \frac{0.0240}{2} - yp*\frac{0.0240}{4912},\ -0.050\right) = (-0.0121, 0.0022, -0.0500)$$

α = arc tan(y_c/f) = 0.0445 $\qquad$ β = arc tan(x_c/f) = -0.2374

$$d = \frac{f * h}{\cos(\alpha) * \cos(\beta) * h_i * s_h/p_h}$$

$$= \frac{0.050*1}{\cos(0.0445)*\cos(-0.2374)*100*0.0240/4912} = 105.39$$

$$t = \frac{105.39}{\sqrt{-0.0121^2 + 0.0022^2 + -0.0500^2}} = 2.0468e+03$$

2d -> 3d Transformation

Object location in 3d camera coordinates:

$(x_o, y_o, z_o) = t^* (x_c, y_c, z_c)$

$= 2.0468e+03 * (-0.0121, 0.0022, -0.0500)$

$(-24.7593, 4.5602, -102.3389)$

Open questions:

- Derivation ok?
- Assumptions ok?
  - Optical axis in sensor center?

2d -> 3d Transformation

Parameters:

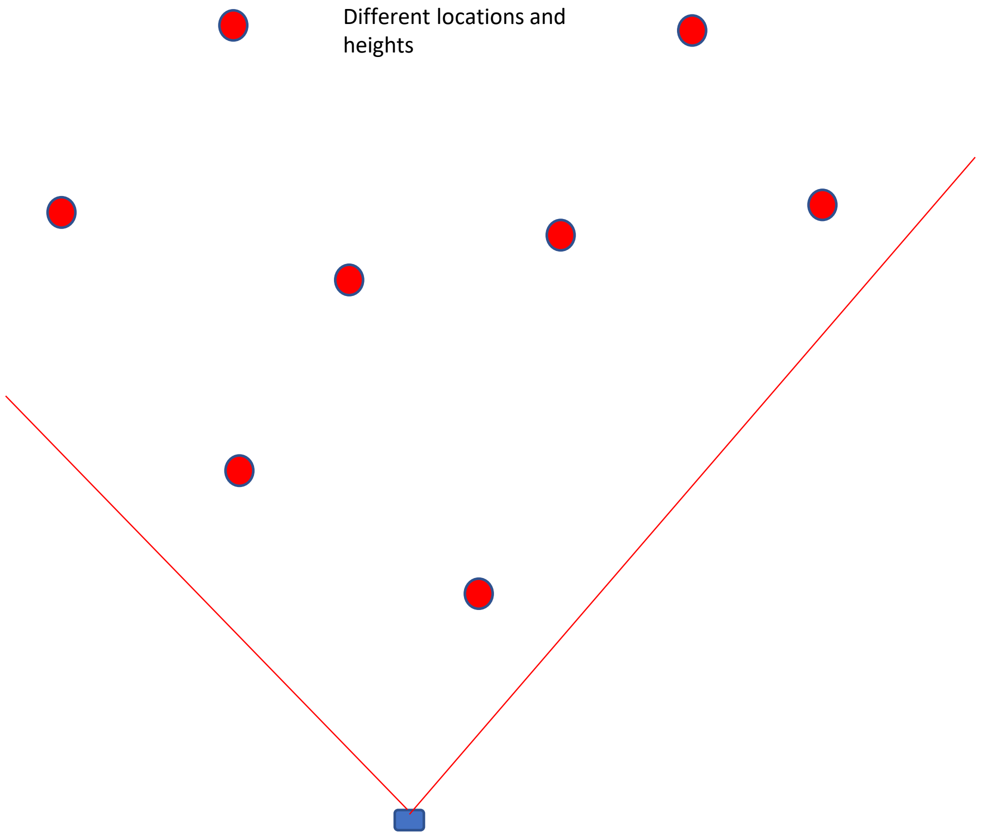
s_w = sensor width (m)
s_h = sensor height (m)
p_w = image width (pixels)
p_h = image height (pixels)
f = focal length (m)

Open questions:

- Video metadata often lacks sensor and focal parameters
- Focal length can change during shooting (zooming)

2d -> 3d Transformation

Experiment 1 in the wild (locations)

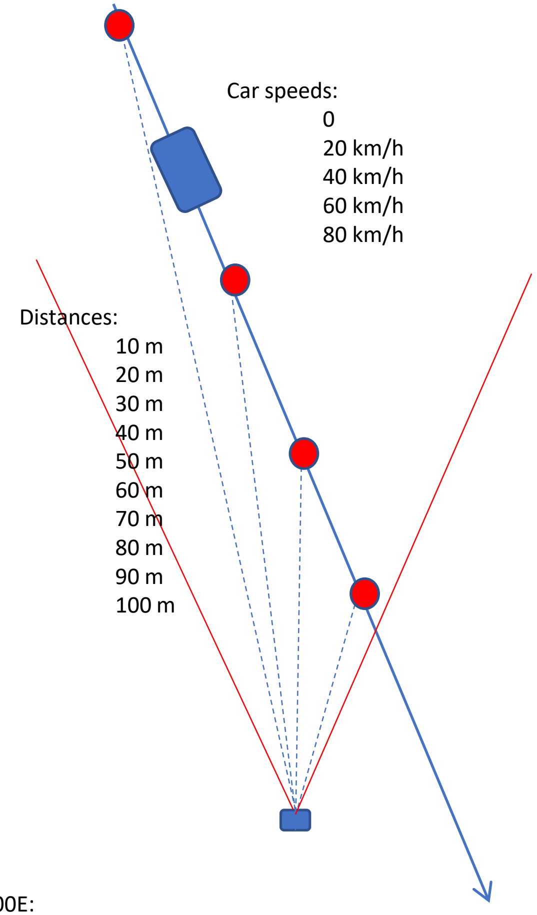


Nikon D800E:

- s_w= sensor width (m) = 0.0359 m
- s_h= sensor height (m) = 0.0240 m
- p_w= image width (pixels) = 7360
- p_h= image height (pixels) = 4912
- h = object height (m) = 1.5 m
- f = focal length (m) = 0.020 m
- fov = $2 * \mathrm{atan}(\frac{s_w}{2*f})$ = 83.81 ^{o}

2d -> 3d Transformation

Experiment 2 in the wild (moving car)



Nikon D800E:

s_w= sensor width (m) = 0.0359 m

s_h= sensor height (m) = 0.0240 m

p_w= image width (pixels) = 7360

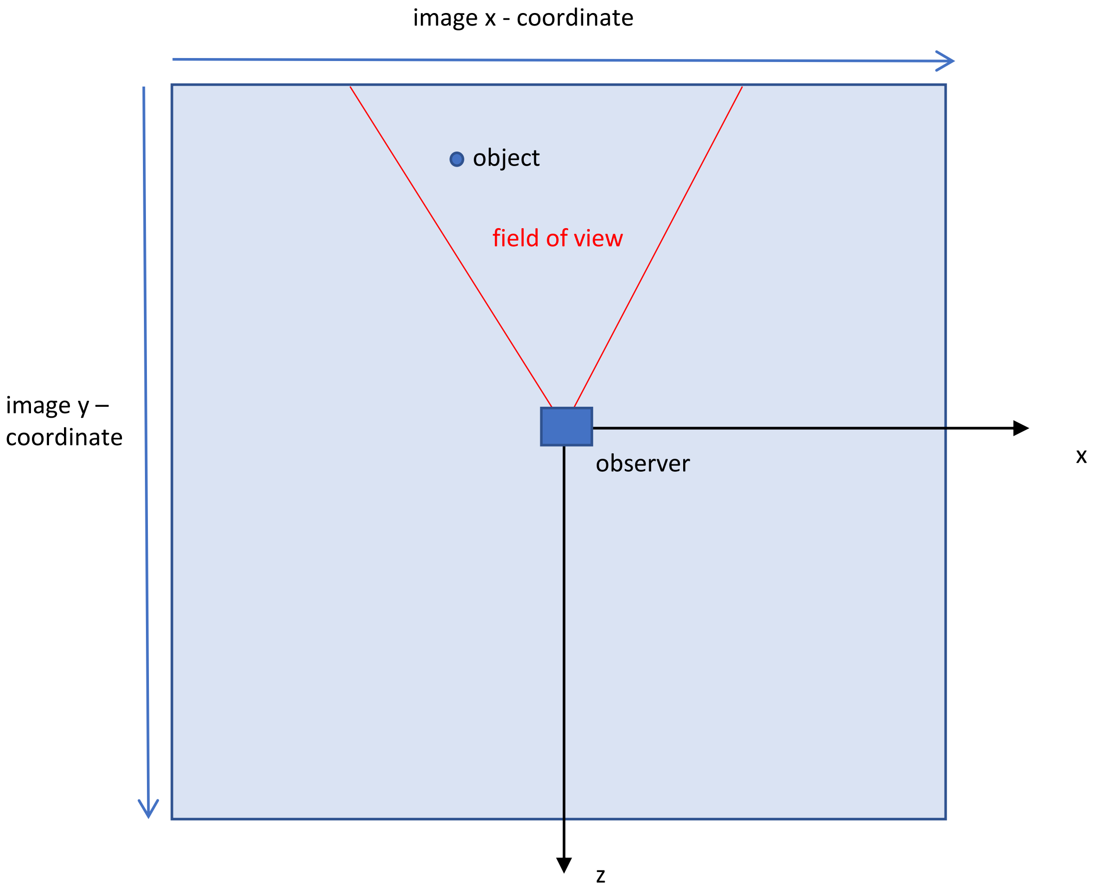
p_h= image height (pixels) = 4912

h = object height (m) = 1.5 m

f = focal length (m) = 0.050 m

fov = $2 * \mathrm{atan}(\frac{s_w}{2*f})$ = 39.49 º

Map Presentation



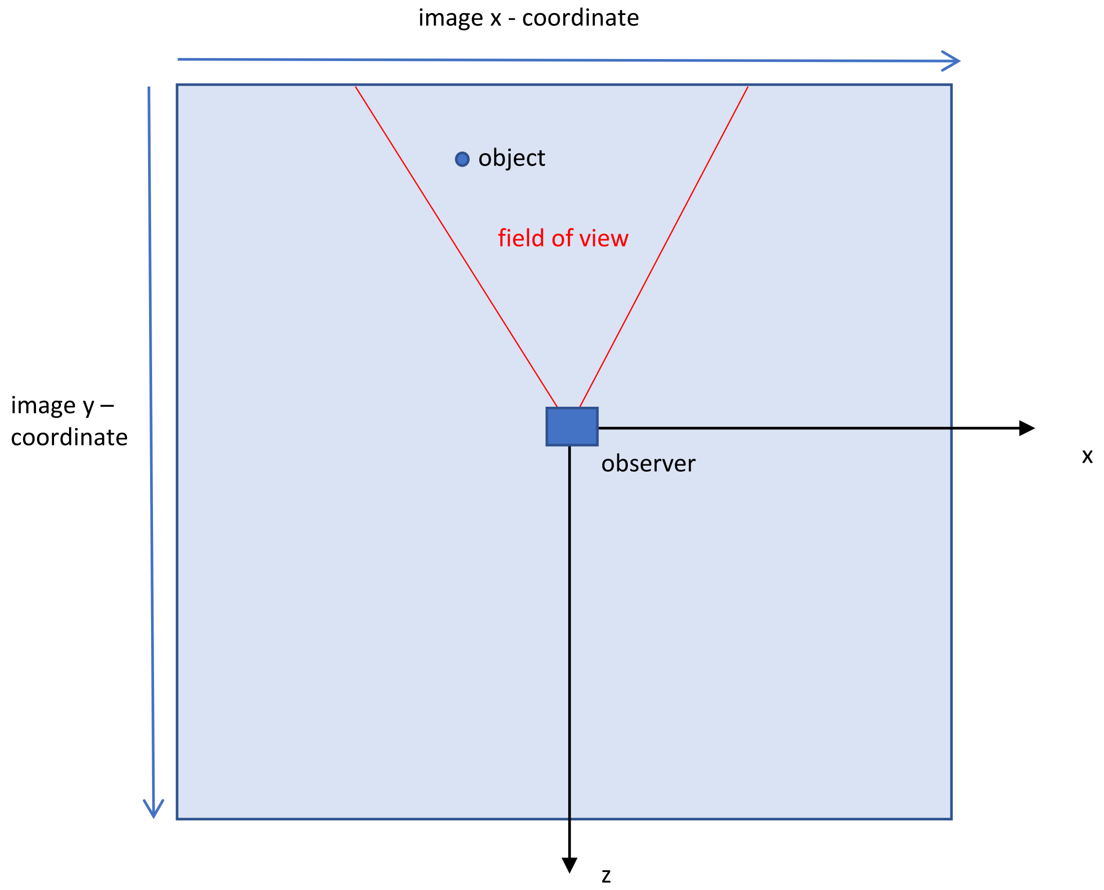
x_w= *object world x coordinate (m)*
z_w= *object world z coordinate(m)*
x_i= *object image x coordinate (pixel)*
y_i= *object image y coordinate (pixel)*
h_w= *image area world height (m)*
h_i = *image area image height (pixels)*
w_w= *image area world width (m)*
w_i = *image area image width (pixels)*

$$(x_i, y_i) = \left(\frac{w_i}{2} + x_w * \frac{w_i}{w_w}, \quad \frac{h_i}{2} + zw * \frac{h_i}{h_w} \right)$$

$$\frac{w_i}{w_w} = \frac{h_i}{h_w} = \text{p (pixel/meter ratio)}$$

$$(x_i, y_i) = \left(\frac{w_i}{2} + x_w * p, \quad \frac{h_i}{2} + zw * \text{p} \right)$$

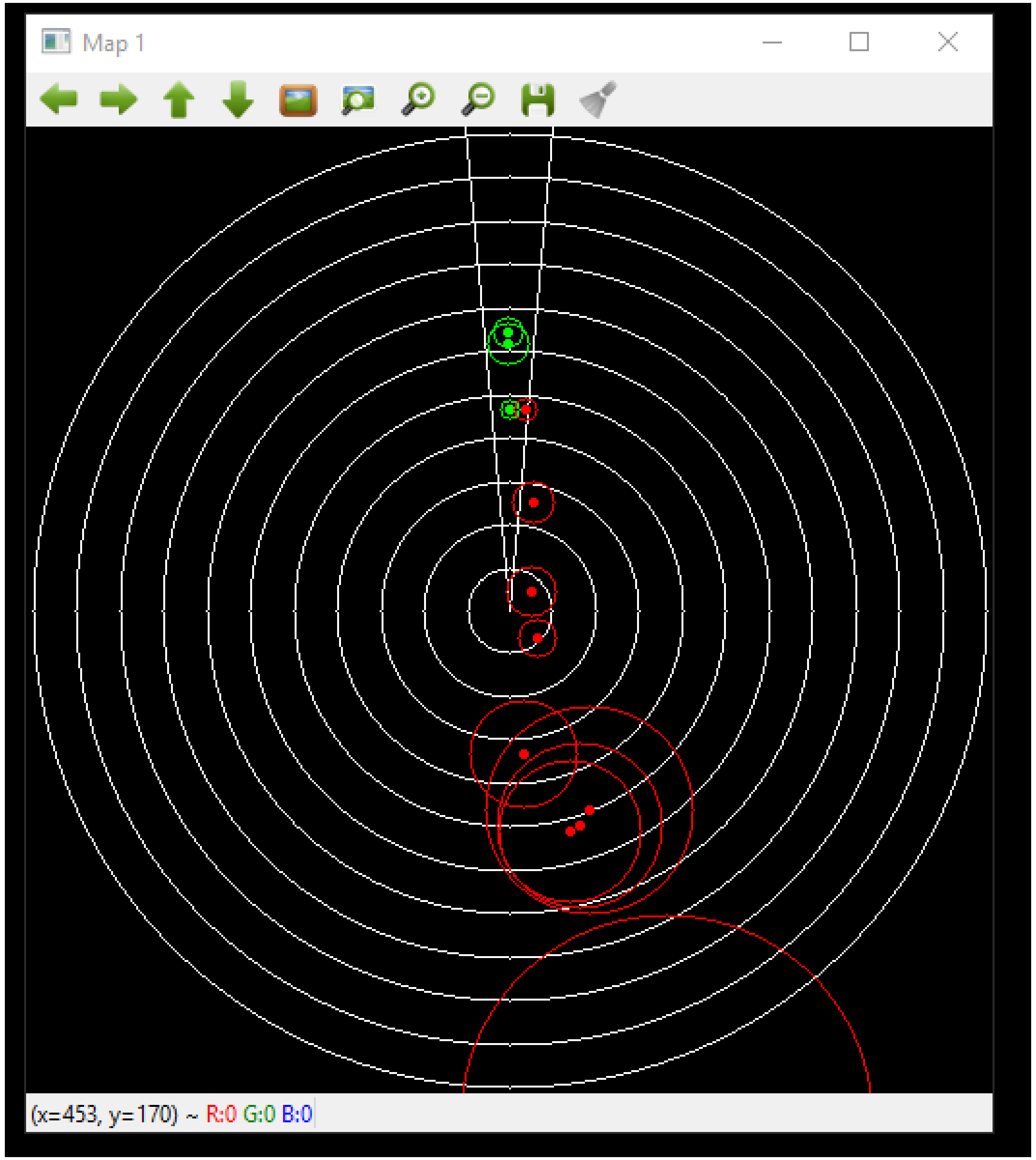
Map Presentation



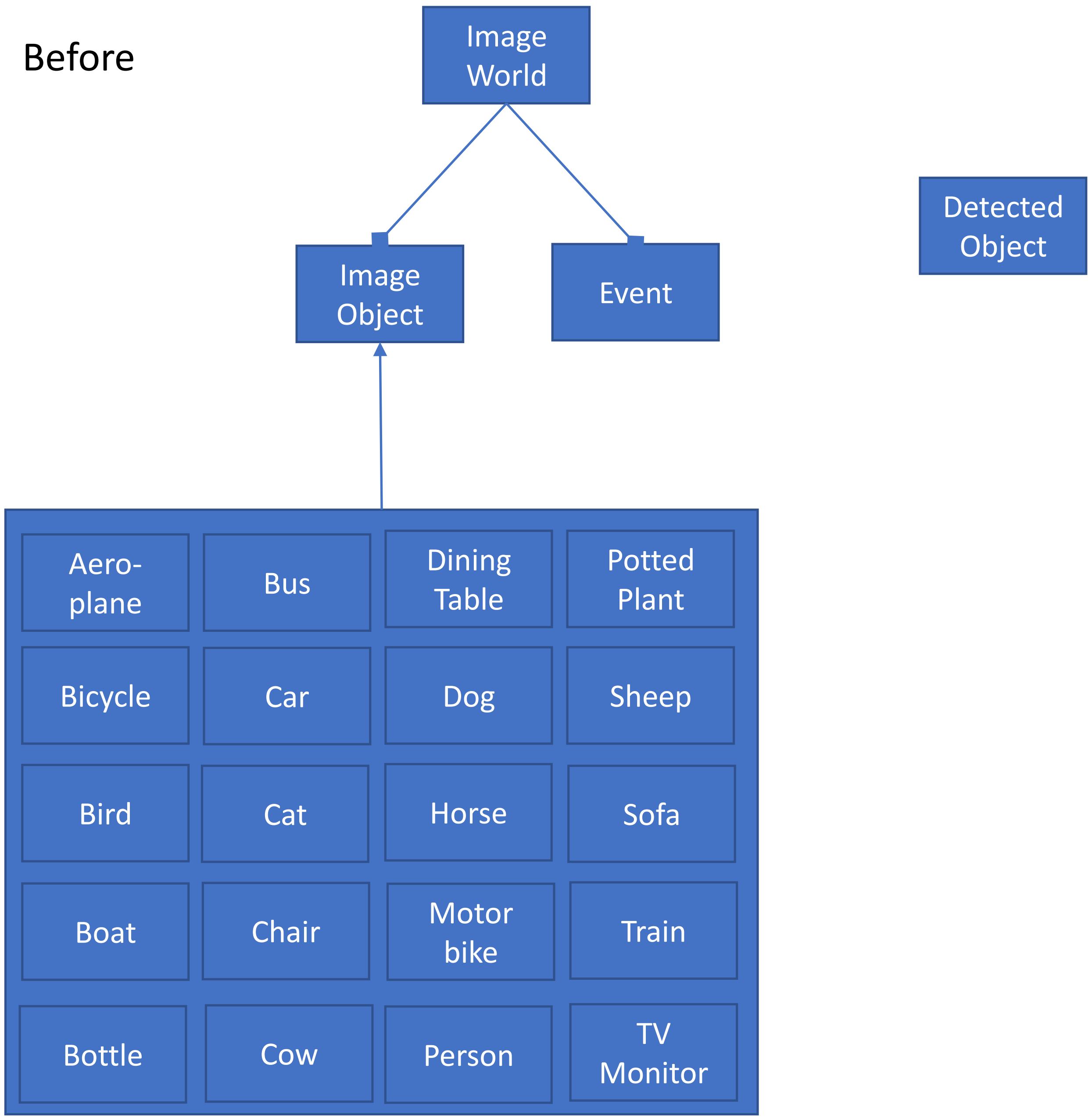
$$FOV = 2 * \text{atan}\left(\frac{s_w}{2 * f}\right)$$

$s_w = sensor\ width\ (m)$
f = focal length (m)

Map Presentation



Entity Diagram



Entity Diagram

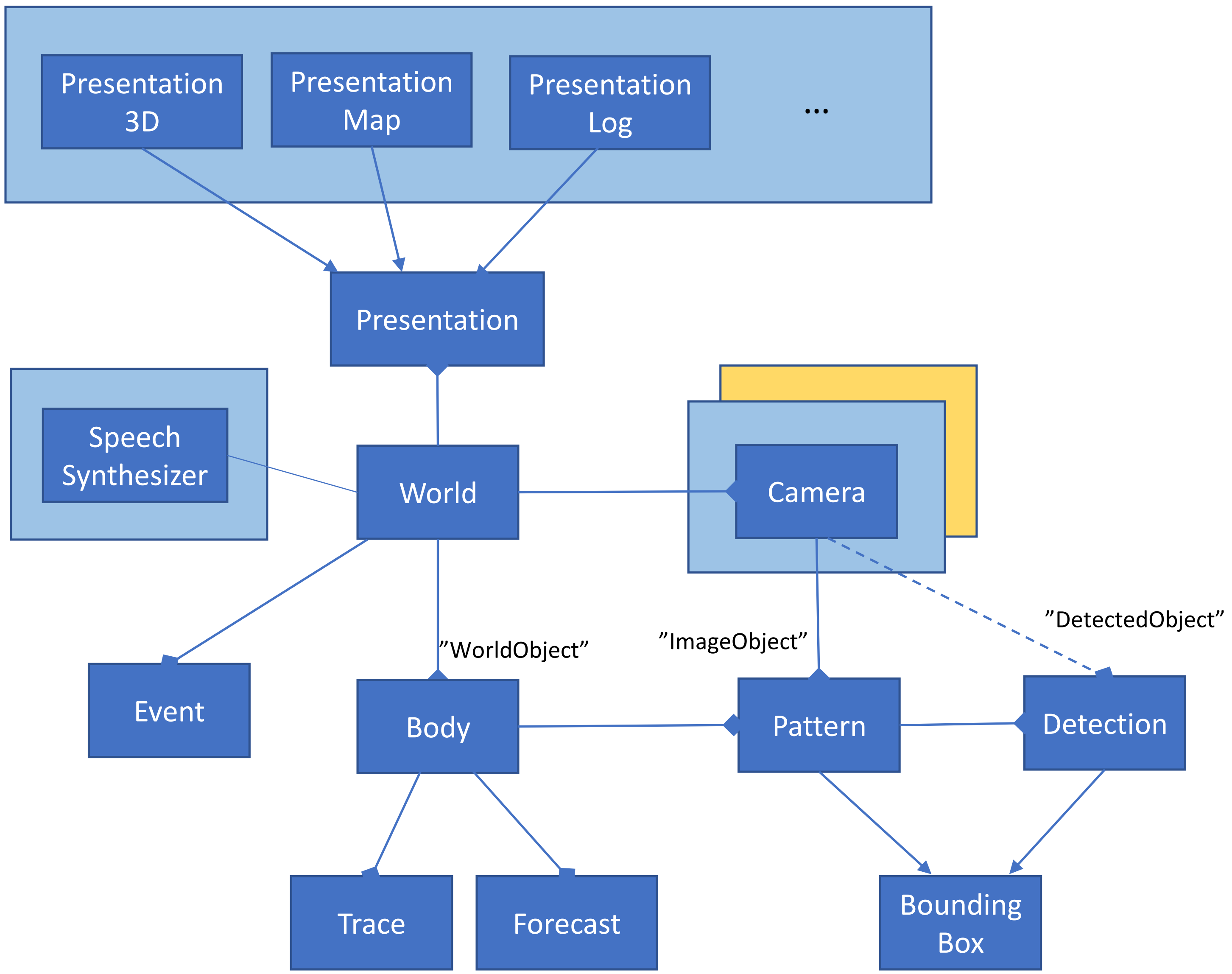
V2.0 goal

- Detected classes not hardcoded
- Object class may change
- Support for many cameras , rotations
- Names less awkward
- Cleaning
- Python style guide followed, excluding line length
- Code optimization
- One package

Name of the software package: ShadowWorld
(Plato: Allegory of the Cave)

Entity Diagram

V2.0



Body Kalman Filtering

- Second order model does not work, constant acceleration makes bodies bounce back or get enormous velocities.
- In world, constant acceleration for several (tens) of seconds is not common
- First order model works! (No wonder it’s popular in robotics…)
- When measurement is lost, the body is switched into constant velocity mode
- In filtering terminology: velocity mode = only prediction, no correction
- Algorithms described in [docs] folder documents

Body Kalman Filtering

State vector s:

$$s = \begin{bmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{bmatrix}$$

where

(x, y, z) = location of the world object center point

(v_x , v_y , v_z) = velocity of the object

State equation in differential form:

$$\frac{ds(t)}{dt} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * s(t) + \epsilon(t) = A_1 * s + \epsilon(t)$$

State equation in difference form:

$$s(k+1) = (I + \Delta * A_1) * s(k) + \epsilon(k)$$

$$= \begin{bmatrix} 1 & 0 & 0 & \Delta & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} * s(k) + \varepsilon(k) = A * s(k) + \varepsilon(k)$$

where Δ is the time increment and ε Gaussian noise with covariance R.

Body Kalman Filtering

Measurement equation

$$z(k) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} * s(k) + \delta(k) = C * s(k) + \delta(k)$$

Where δ is Gaussian noise with covariance matrix Q.

Kalman filter initialization:

$$\mu(0) = \begin{bmatrix} x(0) \\ y(0) \\ z(0) \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

where x(0), y(0), z(0) is the first location measurement.

$$\Sigma(0) = \begin{bmatrix} \alpha & 0 & 0 & 0 & 0 & 0 \\ 0 & \alpha & 0 & 0 & 0 & 0 \\ 0 & 0 & \alpha & 0 & 0 & 0 \\ 0 & 0 & 0 & \beta & 0 & 0 \\ 0 & 0 & 0 & 0 & \beta & 0 \\ 0 & 0 & 0 & 0 & 0 & \beta \end{bmatrix}$$

where α and β are believed variances of location and velocity.

$$R = \begin{bmatrix} r_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & r_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & r_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & r_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & r_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & r_2 \end{bmatrix}$$

where r_1 and r_2 are believed variances of location and velocity.

$$Q = \begin{bmatrix} q & 0 & 0 \\ 0 & q & 0 \\ 0 & 0 & q \end{bmatrix}$$

Where q is the believed measurement variance.

Body Kalman Filtering

Kalman filter update:

$$\mu_1(k) = A * \mu(k-1)$$

$$\Sigma_1(k) = A * \Sigma(k-1) * A^T + R$$

$$K(k) = \Sigma_1(k) * C^T (C * \Sigma_1(k) * C^T + Q)^{-1}$$

$$\mu(k) = \mu_1(k) + K(k) * (z(k) - C * \mu_1(k))$$

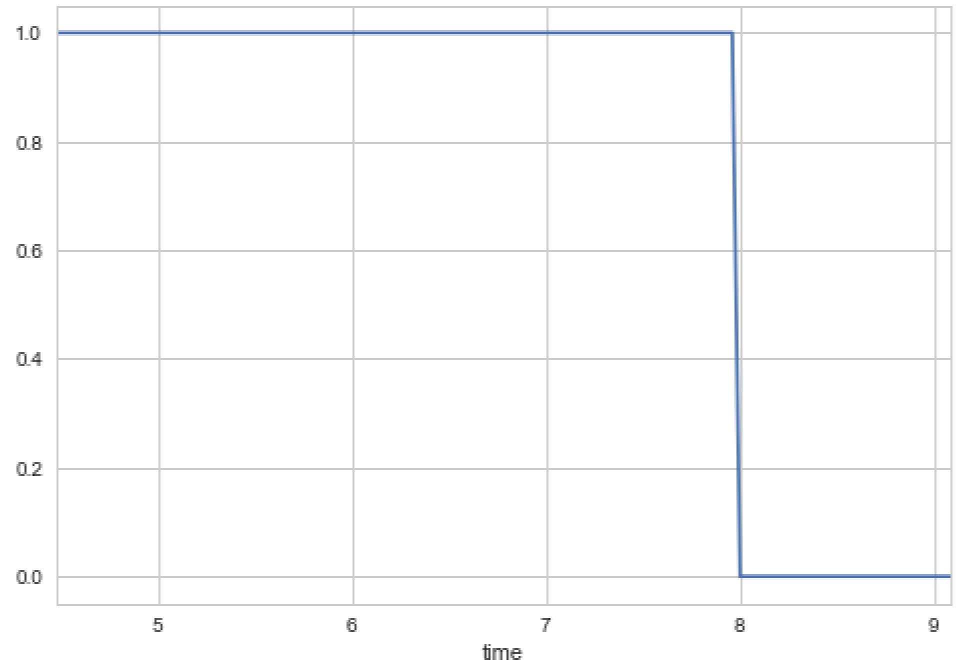
$$\Sigma(k) = (I - K(k) * C) * \Sigma_1(k)$$

Example: Car

Body Kalman Filtering

```
In [606]: data_one['status'].plot()
```

Out[606]: <matplotlib.axes._subplots.AxesSubplot at 0x1f5dbe24be0>



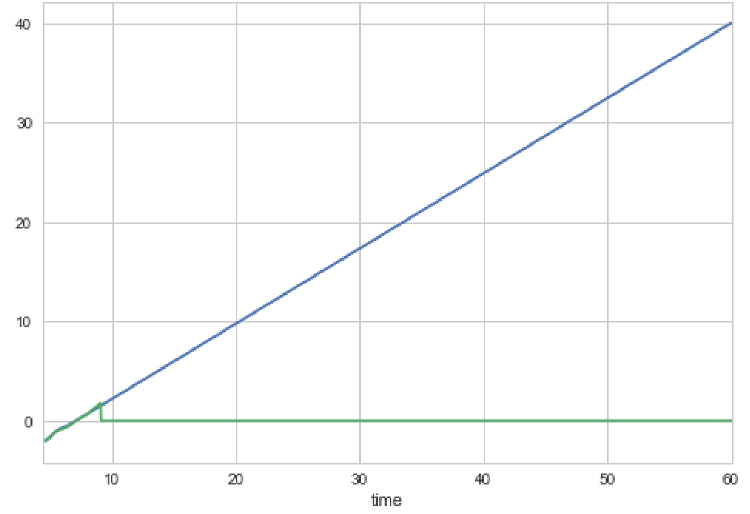
Measurement status:

1=measurement

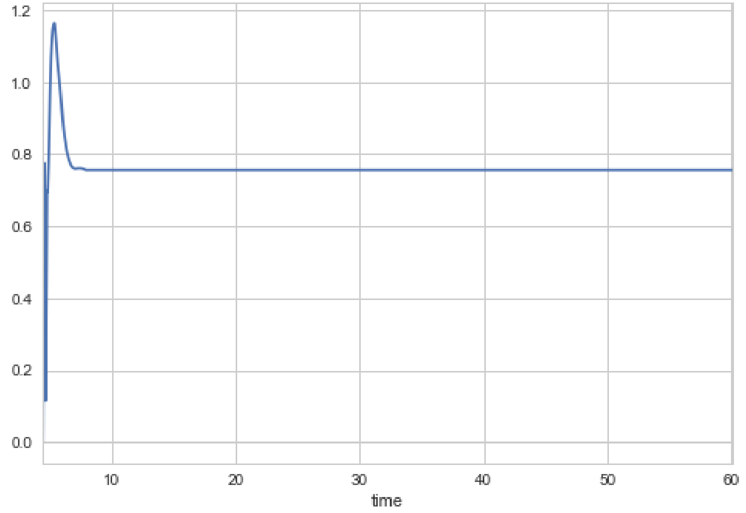
0=no measurement

Body Kalman Filtering

```
In [612]: data_one['x'].plot() # estimated, blue
          data_one['x_pattern'].plot() # measured, green
Out[612]: <matplotlib.axes._subplots.AxesSubplot at 0x1f5defad828>
```



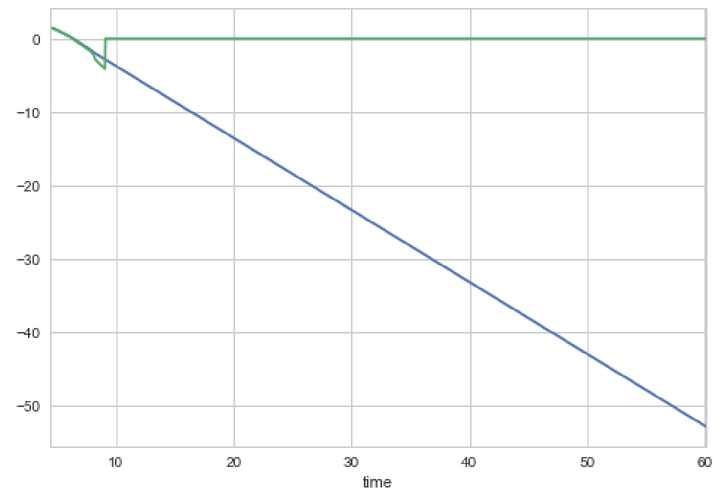

```
In [613]: data_one['vx'].plot() # blue
Out[613]: <matplotlib.axes._subplots.AxesSubplot at 0x1f5dbf720f0>
```



Body Kalman Filtering

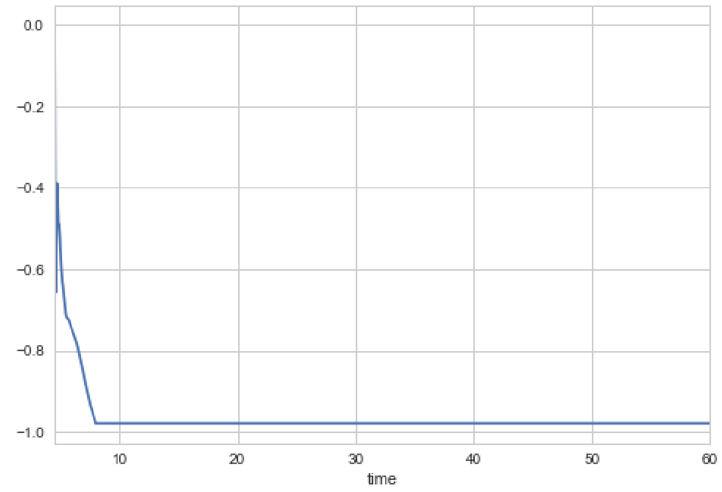
```
In [614]: data_one['y'].plot() # estimated, blue
          data_one['y_pattern'].plot() # measured, green
```

```
Out[614]: <matplotlib.axes._subplots.AxesSubplot at 0x1f5e003d0b8>
```




```
In [615]: data_one['vy'].plot() # blue
```

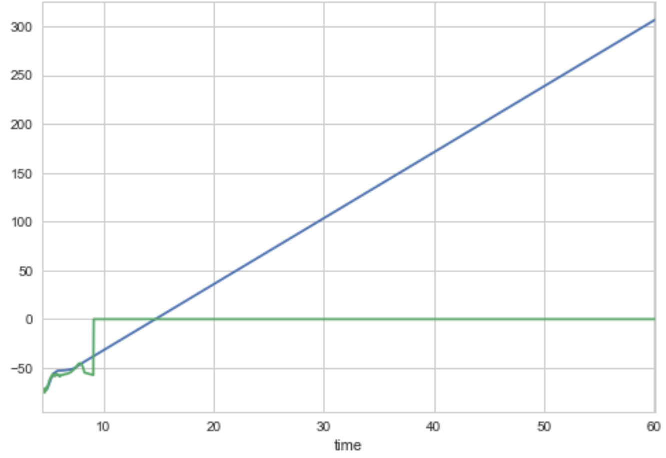
```
Out[615]: <matplotlib.axes._subplots.AxesSubplot at 0x1f5e00cd908>
```



Body Kalman Filtering

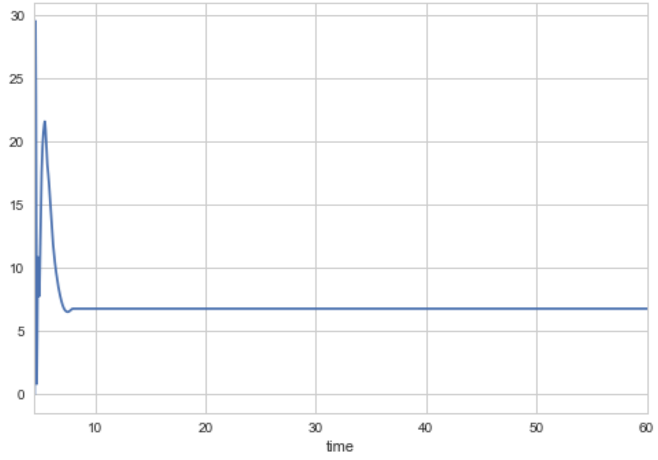
In [616]:
```
data_one['z'].plot() # estimated, blue
data_one['z_pattern'].plot() # measured, green
```

Out[616]: <matplotlib.axes._subplots.AxesSubplot at 0x1f5def60780>



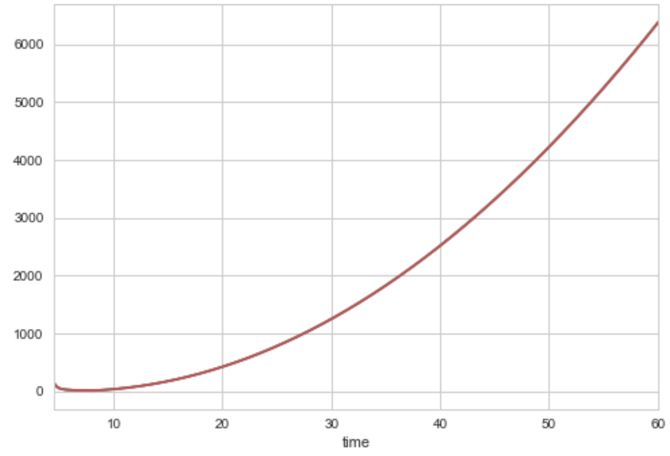
In [617]:
```
data_one['vz'].plot() # blue
```

Out[617]: <matplotlib.axes._subplots.AxesSubplot at 0x1f5d827f0b8>

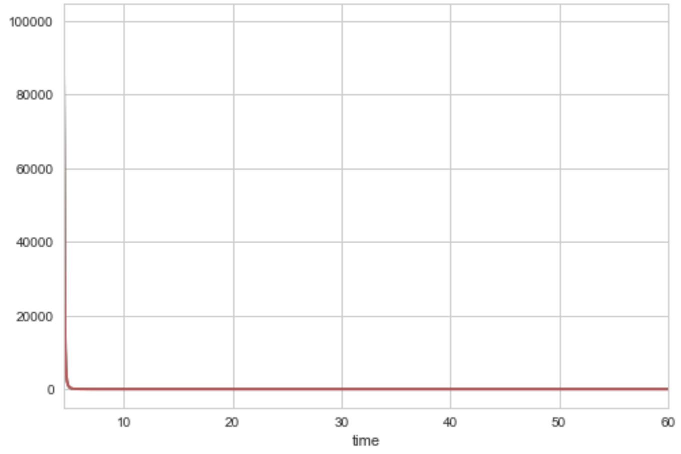


Body Kalman Filtering

```
In [621]: data_one['sigma_00'].plot() # x variance, blue
          data_one['sigma_11'].plot() # y variance, green
          data_one['sigma_22'].plot() # y variance, green

Out[621]: <matplotlib.axes._subplots.AxesSubplot at 0x1f5e0280d68>
```

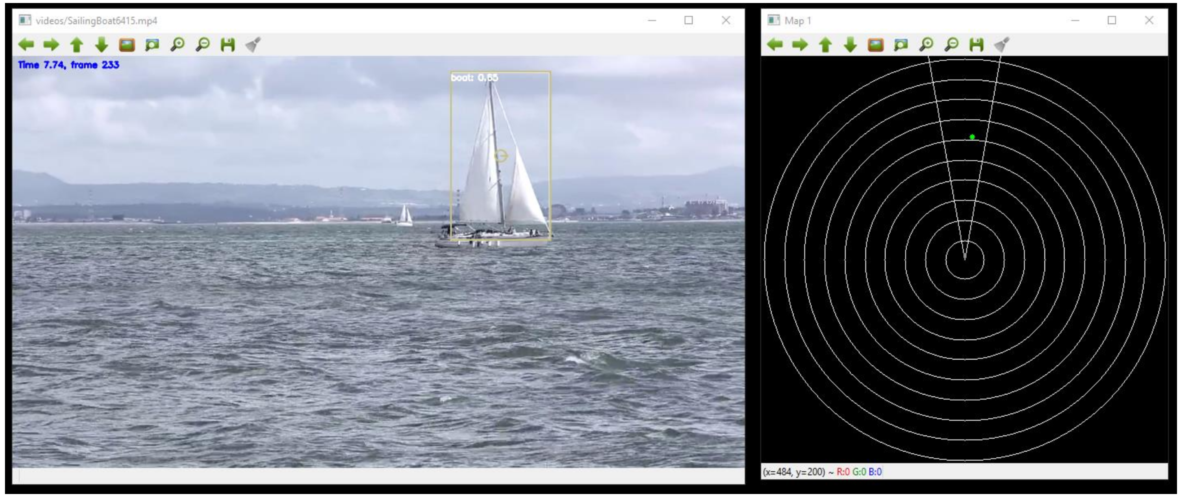



```
In [622]: data_one['sigma_33'].plot() # vx variance, blue
          data_one['sigma_44'].plot() # vy variance, green
          data_one['sigma_55'].plot() # vy variance, green

Out[622]: <matplotlib.axes._subplots.AxesSubplot at 0x1f5e03d60b8>
```

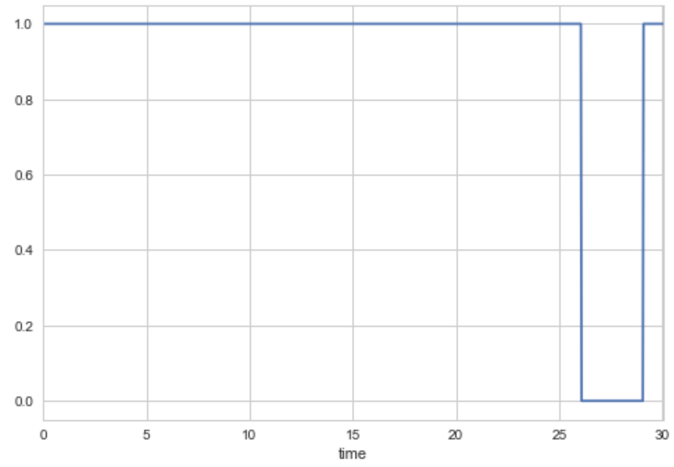


Body Kalman Filtering

Example 2: Slow moving object

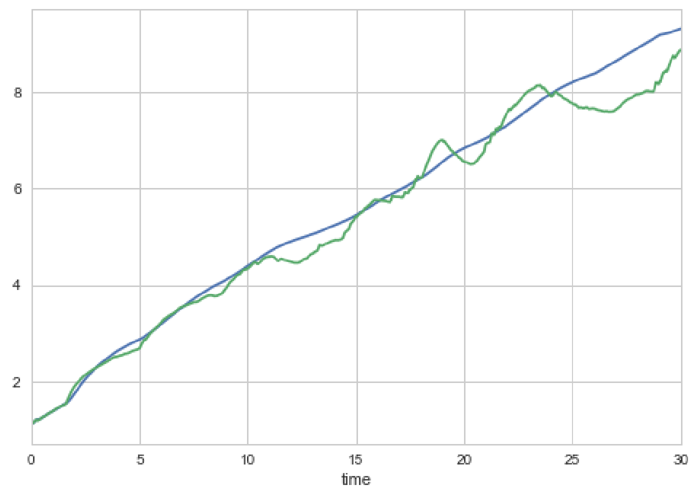



```
In [8]: data_one['status'].plot()
Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x13a6ba38470>
```

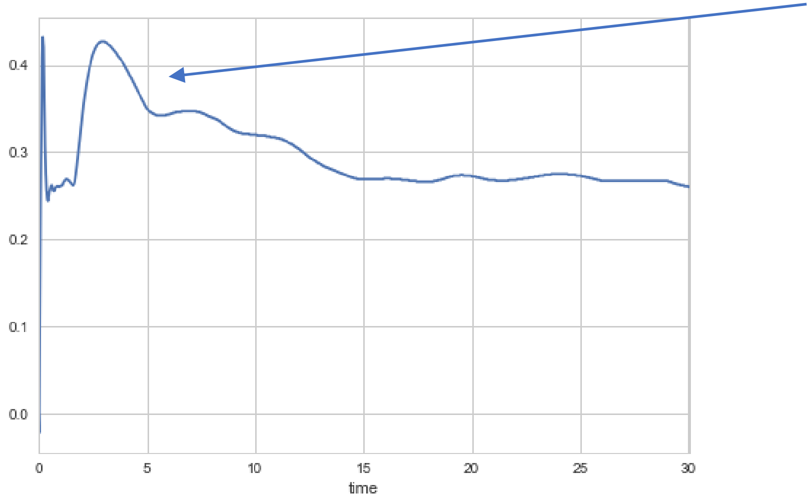


Body Kalman Filtering

In [9]:
```
data_one['x'].plot() # estimated, blue
data_one['x_pattern'].plot() # measured, green
```
Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x13a6b725e80>



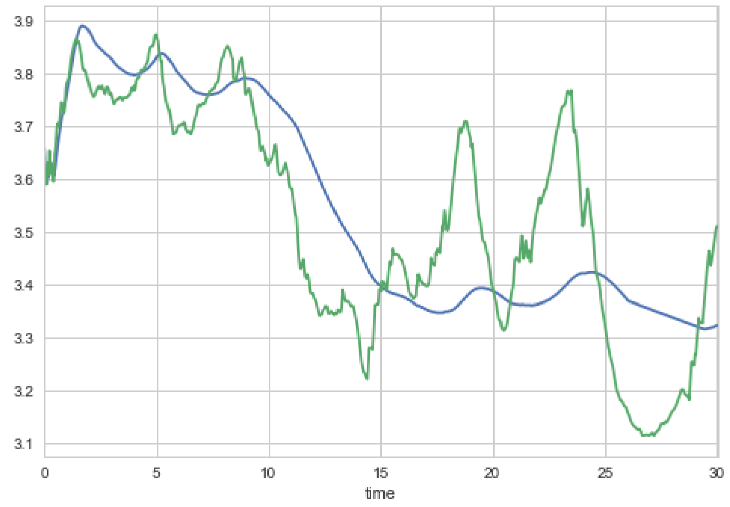
In [10]:
```
data_one['vx'].plot() # blue
```
Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x13a6b7be780>



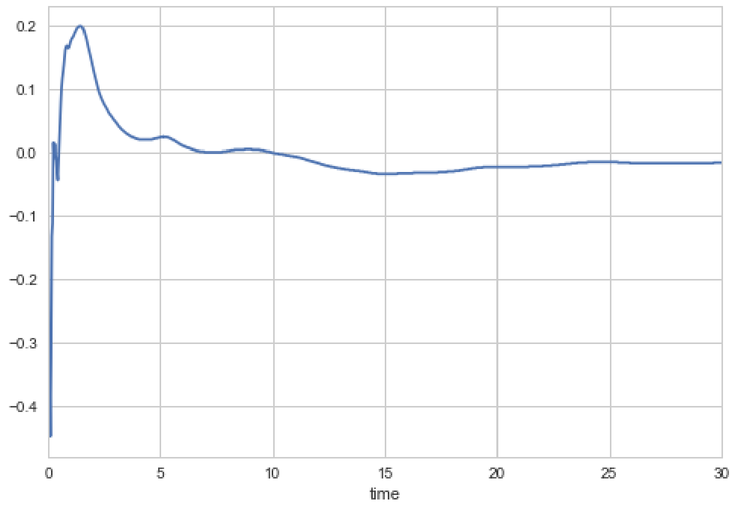
Takes long time to settle

Body Kalman Filtering

```
In [11]: data_one['y'].plot() # estimated, blue
         data_one['y_pattern'].plot() # measured, green

Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x13a6b88ceb8>
```



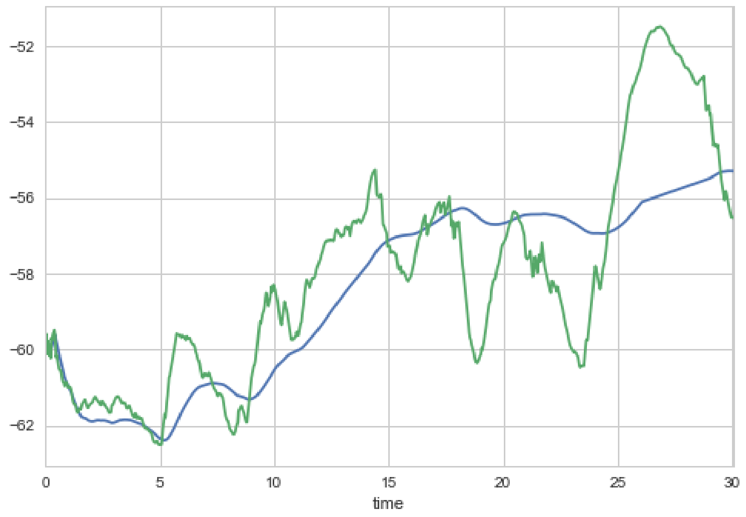

```
In [12]: data_one['vy'].plot() # blue

Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x13a6b9c1b00>
```



Body Kalman Filtering

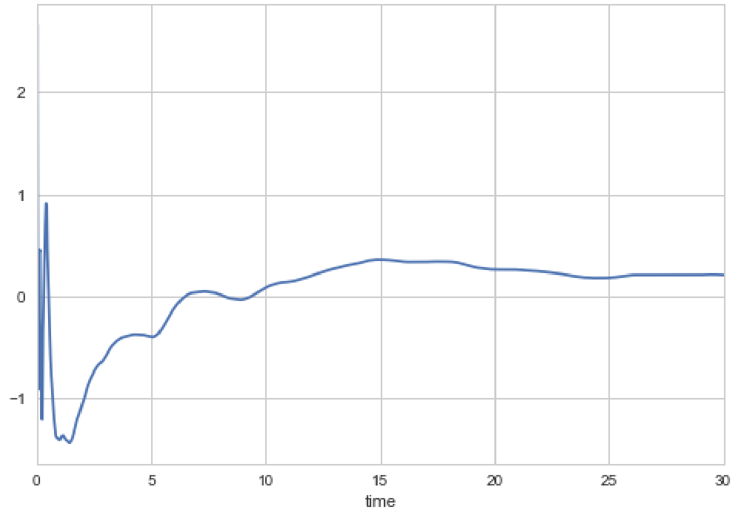
In [13]:
```
data_one['z'].plot() # estimated, blue
data_one['z_pattern'].plot() # measured, green
```

Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x13a6b95dcc0>



In [14]:
```
data_one['vz'].plot() # blue
```

Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0x13a6cd225f8>



Body Kalman Filtering

```python
# Hyperparameters-----------------------------------------------------------
BODY_ALFA = 100000.0 # Body initial location error variance 200
BODY_BETA = 100000.0 # Body initial velocity error variance 10000
BODY_C = np.array([[1.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 1.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 1.0, 0.0, 0.0, 0.0]
                  ]) # Body measurement matrix
BODY_DATA_COLLECTION_COUNT = 30 # How many frames until notification
BODY_Q = np.array([[200.0, 0.0,   0.0],
                   [0.0, 200.0,   0.0],
                   [0.0,   0.0, 200.0]]) # Body measurement variance 200
BODY_R = np.array([[0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
                  ]) # Body state equation covariance
```

Kalman filter update:

$$\mu_1(k) = A * \mu(k-1)$$

$$\Sigma_1(k) = A * \Sigma(k-1) * A^T + R$$

$$K(k) = \Sigma_1(k) * C^T (C * \Sigma_1(k) * C^T + Q)^{-1}$$

$$\mu(k) = \mu_1(k) + K(k) * (z(k) - C * \mu_1(k))$$

$$\Sigma(k) = (I - K(k) * C) * \Sigma_1(k)$$

- Filtering requires several seconds to achieve reliable results
- Kalman filter parameters are not optimized
- Optimization will be done later

Paper

Image-based situation awareness: Estimating location and velocity using single camera object detection

?????

Contents:

1. Abstract
2. Introduction
3. Related work
4. Our approach
    1. Definitions
    2. Object identity
    3. Distance estimation
    4. Location and velocity estimation
    5. Movement prediction
5. Experiments
    1. Test setup
    2. Evaluation metric
    3. Results
    4. Other experiments
6. Conclusion

detection, pattern…

Hungarian algorithm, distance metrics, movement prediction

geometry

Kalman filtering

Difference equation, uncertanties

In the wild. See previous slides. More?

Existing datasets? Which?

Paper

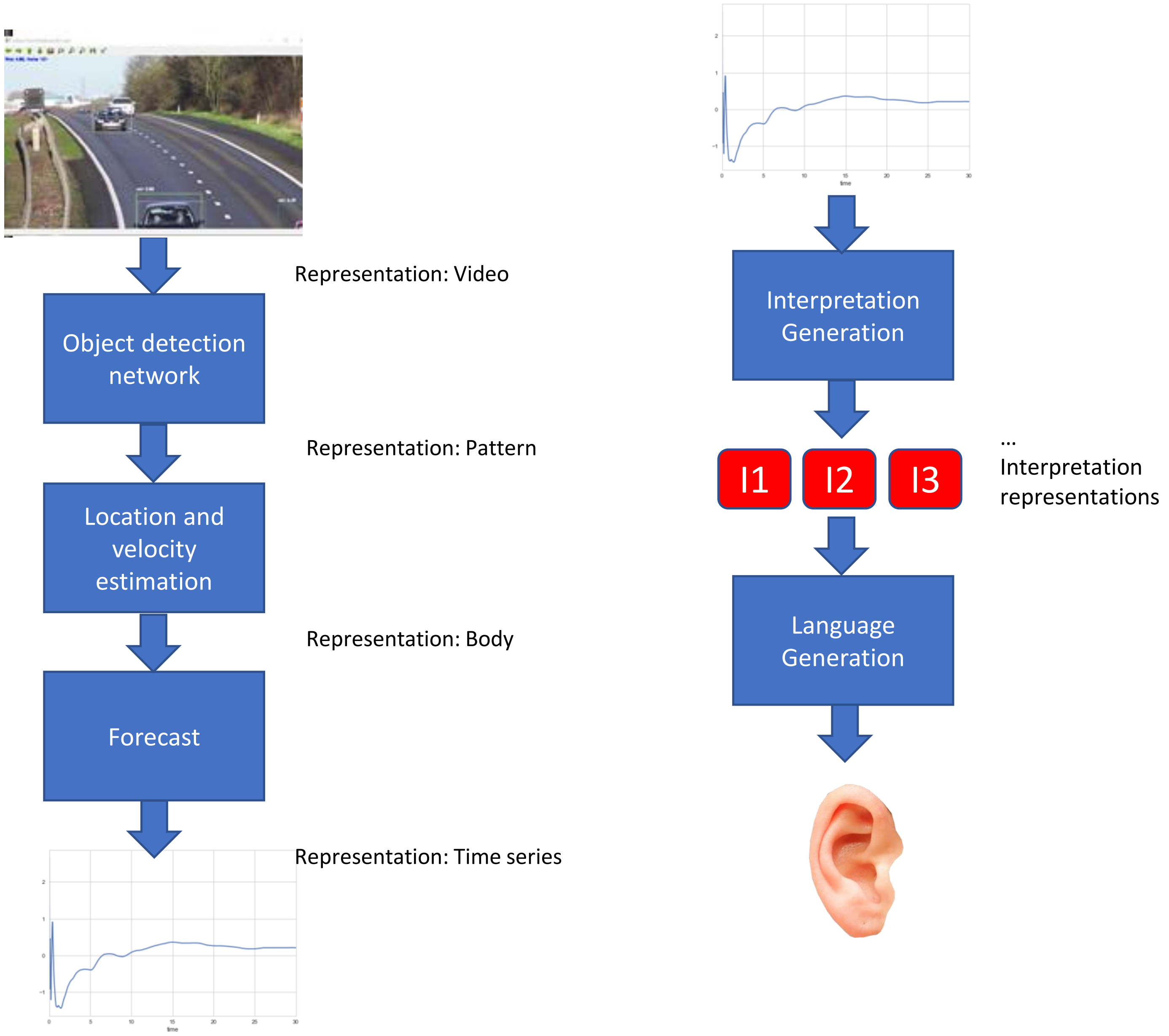
- One solution for location estimation and movement prediction for video detected object (nearly) solved

- Work needed:
  - Kalman filter parameter adjustment*
  - Experiments in the wild (see previous slides)
  - Other tests? (using existing (tracking) dataset)

- Where to publish?
  - ECCV 2018 (8-14.9, Munich, deadline 14.3.2018, too soon)
  - CVPR 2019 (6/2019, Long Beach, deadline 11/2018)
  - ICCV 2019 (29.10.-3.11.2019, Seoul, deadline 1/2019)

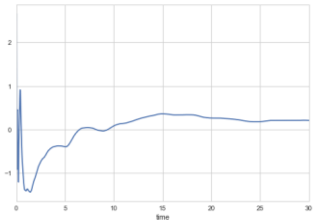
*Locations and especially velocities take too much time to settle at the moment.

Representations for Interpretation



Representations for Interpretation

Example for interpretation representation: collision detection



Time series forecast for body locations

Interpretation Generation

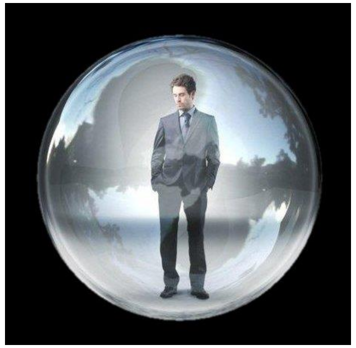
$$
\begin{bmatrix}
0 & p_{12} & p_{13} & p_{14} & \cdots & p_{1n} \\
p_{21} & 0 & p_{23} & p_{24} & \cdots & p_{2n} \\
p_{31} & p_{32} & 0 & p_{34} & \cdots & p_{3n} \\
p_{41} & p_{42} & p_{43} & 0 & \cdots & p_{4n} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
p_{n1} & p_{n2} & p_{n3} & p_{n4} & \cdots & 0
\end{bmatrix}
$$

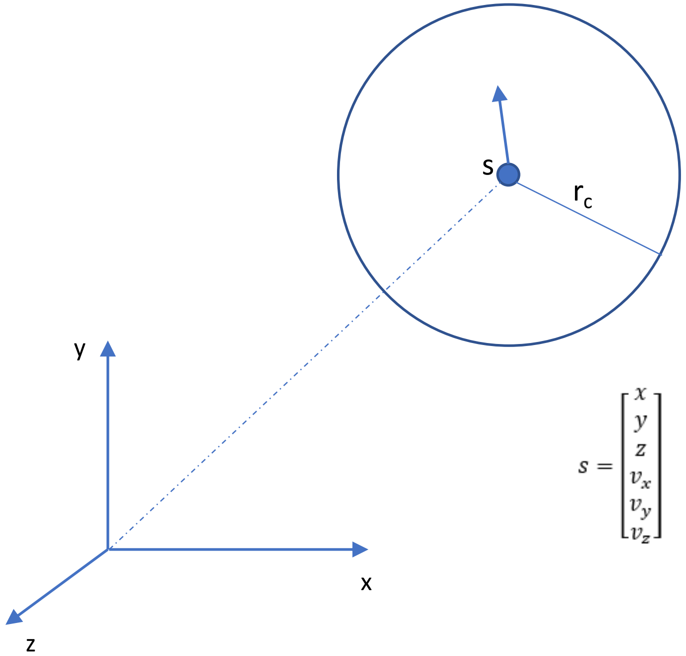
(Symmetric) collision probability matrix
p_{nm} = probability that bodies n and m will collide

Probabilistic Model for Body

Some form of spatial simplification is needed, like

- cube
- rectangular prism
- cylinder
- sphere (probably the easiest math)





From state estimation

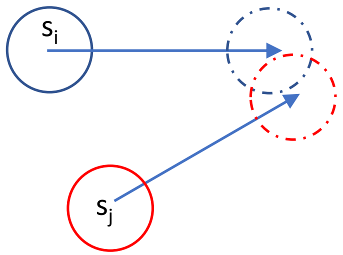
$$P(s) \sim N(s \mid \mu_s, Vs)$$

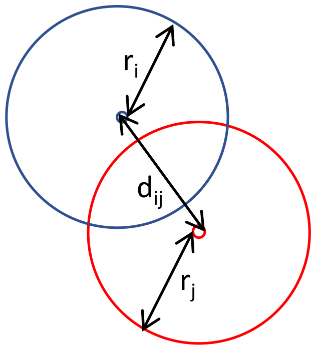
$$P(r_c) \sim N(rc \mid \mu_{r,c}\, Vr_{,c})$$

Class specific, a priori

$$s = \begin{bmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{bmatrix}$$

Collision Detection





Collision C_{ij} if $d_{ij} < r_i + r_j$

$$\begin{bmatrix} x_i(t) \\ y_i(t) \\ z_i(t) \end{bmatrix} = \begin{bmatrix} x_i(0) \\ y_i(0) \\ z_i(0) \end{bmatrix} + t * \begin{bmatrix} v_{i,x}(0) \\ v_{i,y}(0) \\ v_{i,z}(0) \end{bmatrix}$$

$$d_{ij}(t) = \sqrt{(x(t)_i - x(t)_j)^2 + (y(t)_i - y(t)_j)^2 + (z(t)_i - z(t)_j)^2}$$

Random variable $$r = \begin{bmatrix} x_1(0) \\ y_1(0) \\ z_1(0) \\ v_{1,x}(0) \\ v_{1,y}(0) \\ v_{1,z}(0) \\ r_1 \\ \cdots \\ x_n(0) \\ y_n(0) \\ z_n(0) \\ v_{n,x}(0) \\ v_{n,y}(0) \\ v_{n,z}(0) \\ r_n \\ t \end{bmatrix}$$

t: uniform distribution on $[0, t_{end}]$

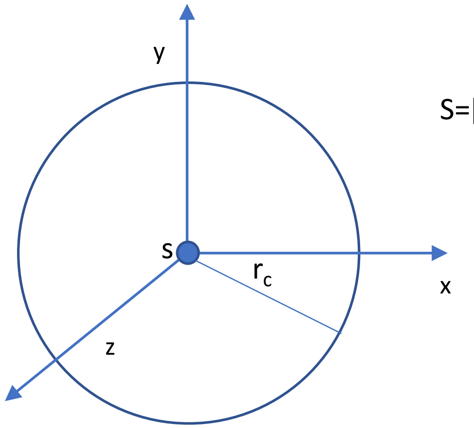
Sampling: $$p_{ij} = E\{C_{ij}\} = \sum_{k=1}^{m} \frac{\delta(C_{i,j})}{m}$$

Open question:

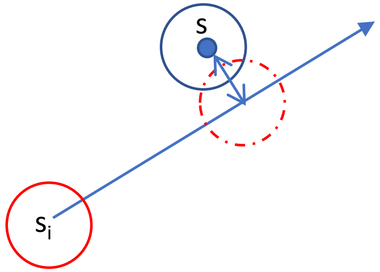
1) Too many dimensions?
2) More efficient sampling with MCMC / Metropolis-Hastings?

Collision Detection

Simpler case: Collision with the observer



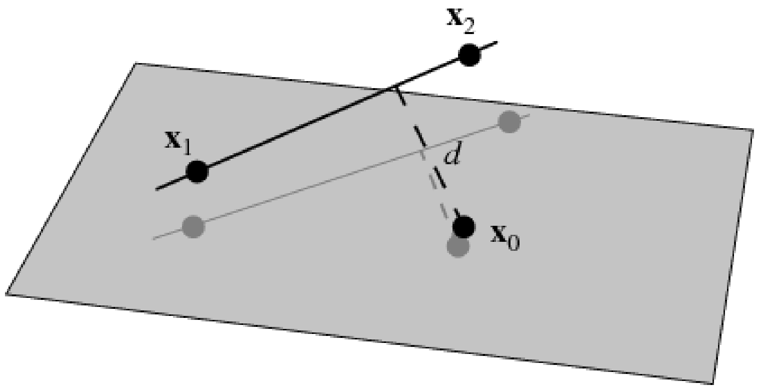
$S=[0,0,0,0,0,0]^T$, deterministic, r = r[class person]



$$r = \begin{bmatrix} x_i(0) \\ y_i(0) \\ z_i(0) \\ v_{i,x}(0) \\ v_{i,y}(0) \\ v_{i,z}(0) \\ r_i \\ r \end{bmatrix}$$

For each body i, a random vector r is sampled and minimum distance to the observer calculated. If the distance is less than r1+r, collision occurred. The proportion on collisions to all cases is the probability estimate for the body/observer collision.

Collision Detection



$$t = -\frac{(\mathbf{x}_1 - \mathbf{x}_0)\cdot(\mathbf{x}_2 - \mathbf{x}_1)}{|\mathbf{x}_2 - \mathbf{x}_1|^2}$$

$$d = \frac{|(\mathbf{x}_2 - \mathbf{x}_1)\times(\mathbf{x}_1 - \mathbf{x}_0)|}{|\mathbf{x}_2 - \mathbf{x}_1|} = \frac{|(\mathbf{x}_0 - \mathbf{x}_1)\times(\mathbf{x}_0 - \mathbf{x}_2)|}{|\mathbf{x}_2 - \mathbf{x}_1|}$$

In our case:
x_1=P(0)
x_2=P(0)+V(0) (loc after one sec)
x_0=0

- ?

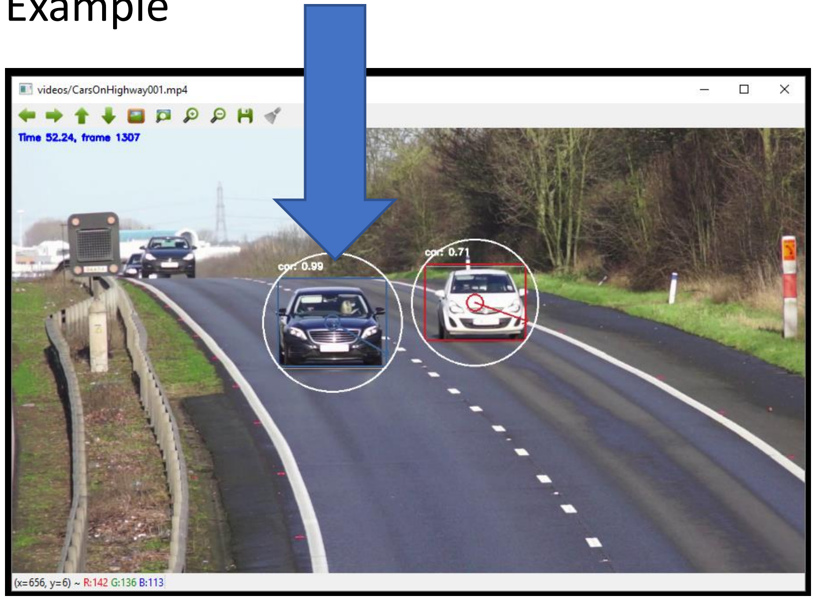
$$t = \frac{(P(0)-0)\cdot(P(0)+V(0)-P(0))}{|P(0)+V(0)-P(0)|^2} = \frac{P(0)\cdot V(0)}{|V(0)|^2}$$

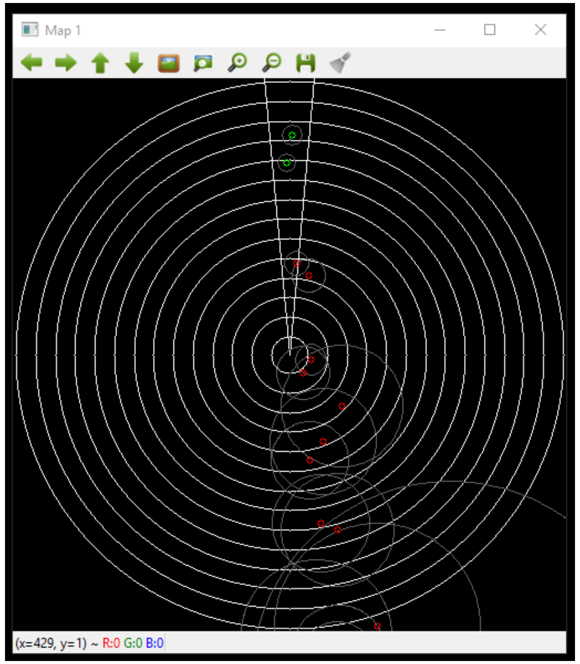
If t<=0, nearest point is P(0)

$$d = |P(0) + t * V(0)|$$

Collision Detection

Example

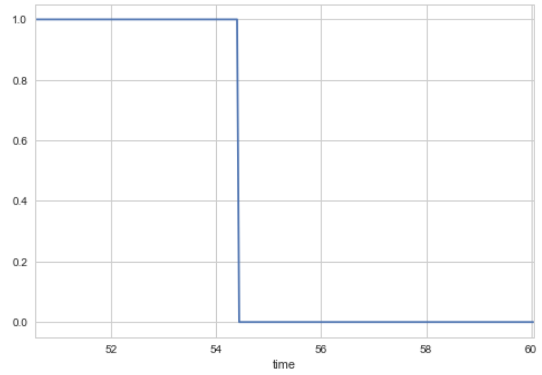




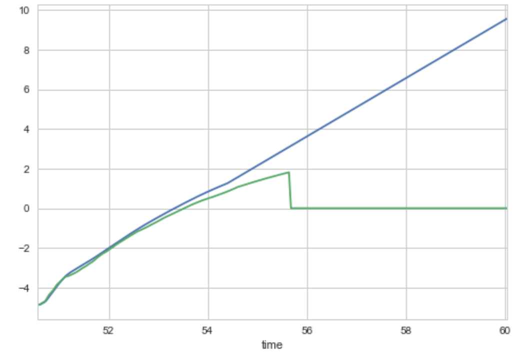
Collision Detection

Example

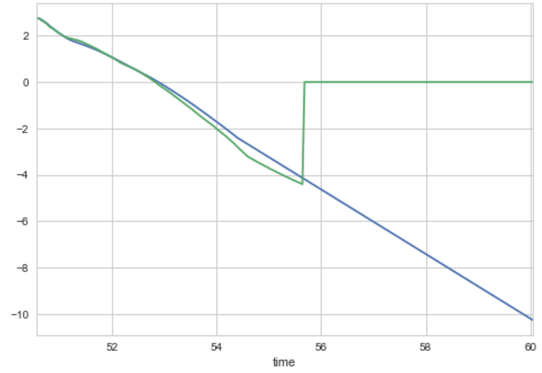
In [177]:
```
data_one['status'].plot()
```
Out[177]: <matplotlib.axes._subplots.AxesSubplot at 0x1b238f06588>



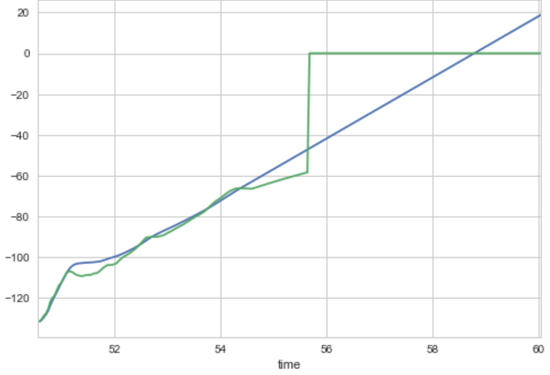
In [446]:
```
data_one['x'].plot() # estimated, blue
data_one['x_pattern'].plot() # measured, green
```
Out[446]: <matplotlib.axes._subplots.AxesSubplot at 0x1b241357278>



In [448]:
```
data_one['y'].plot() # estimated, blue
data_one['y_pattern'].plot() # measured, green
```
Out[448]: <matplotlib.axes._subplots.AxesSubplot at 0x1b241111898>



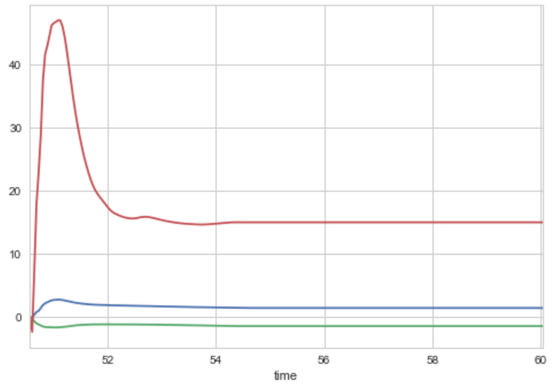
In [449]:
```
data_one['z'].plot() # estimated, blue
data_one['z_pattern'].plot() # measured, green
```
Out[449]: <matplotlib.axes._subplots.AxesSubplot at 0x1b2390bf0f0>



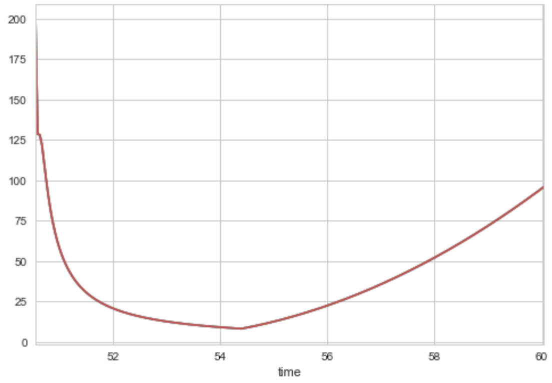
Collision Detection

Example

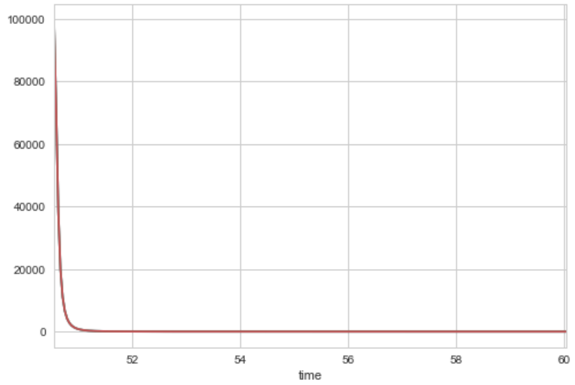
In [450]:
```
data_one['vx'].plot() # estimated, blue
data_one['vy'].plot() # estimated, green
data_one['vz'].plot() # estimated, red
```
Out[450]: <matplotlib.axes._subplots.AxesSubplot at 0x1b2391b7c50>



In [451]:
```
data_one['sigma_00'].plot() # x, estimated, blue
data_one['sigma_11'].plot() # y, estimated, green
data_one['sigma_22'].plot() # z, estimated, red
```
Out[451]: <matplotlib.axes._subplots.AxesSubplot at 0x1b2391a0e10>



In [452]:
```
data_one['sigma_33'].plot() # vx, estimated, blue
data_one['sigma_44'].plot() # vy, estimated, green
data_one['sigma_55'].plot() # vz, estimated, red
```
Out[452]: <matplotlib.axes._subplots.AxesSubplot at 0x1b2392fb780>



Collision Detection

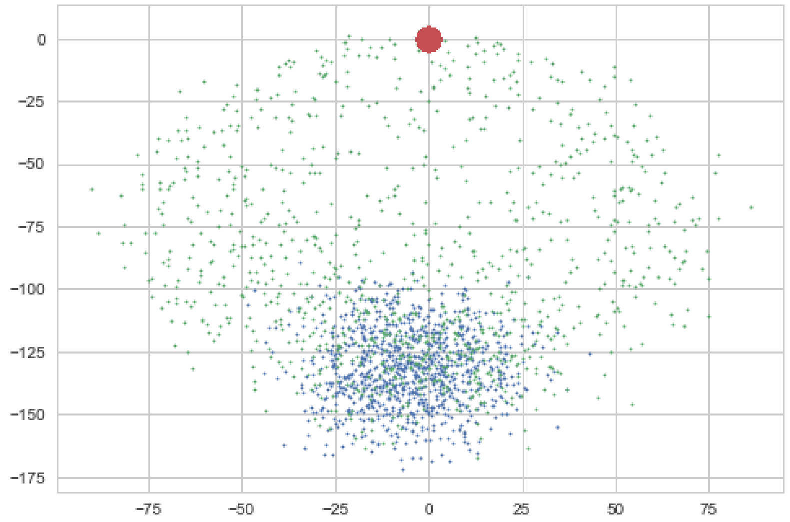
Example

T=50.56 sec

```
In [506]: mu # mean for location and velocity
Out[506]: array([  -4.868,    2.767, -131.242,    0.   ,    0.   ,    0.   ])
```

```
In [508]: sigma # covariance for location and velocity
Out[508]: array([[  1.99601000e+02,   0.00000000e+00,   0.00000000e+00,
                    7.97100000e+00,   0.00000000e+00,   0.00000000e+00],
                 [  0.00000000e+00,   1.99601000e+02,   0.00000000e+00,
                    0.00000000e+00,   7.97100000e+00,   0.00000000e+00],
                 [  0.00000000e+00,   0.00000000e+00,   1.99601000e+02,
                    0.00000000e+00,   0.00000000e+00,   7.97100000e+00],
                 [  7.97100000e+00,   0.00000000e+00,   0.00000000e+00,
                    9.98405740e+04,   0.00000000e+00,   0.00000000e+00],
                 [  0.00000000e+00,   7.97100000e+00,   0.00000000e+00,
                    0.00000000e+00,   9.98405740e+04,   0.00000000e+00],
                 [  0.00000000e+00,   0.00000000e+00,   7.97100000e+00,
                    0.00000000e+00,   0.00000000e+00,   9.98405740e+04]])
```

```
In [537]: plt.scatter(x,z,alpha=1.0,s=2) # blue, start point
          plt.scatter(x_end,z_end,alpha=1.0,s=2) # green, end point
          plt.scatter(xo,zo,alpha=1.0,s=220) # red, observer
Out[537]: <matplotlib.collections.PathCollection at 0x1b243a07a20>
```



P=0.000

Collision Detection

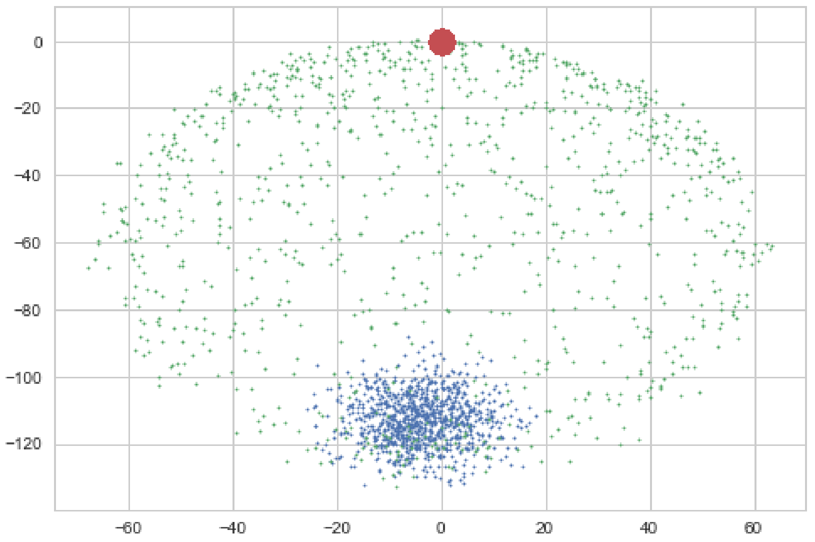
Example

T=51.00 sec

```
In [547]: mu # mean for location and velocity
Out[547]: array([  -3.806,    2.093, -112.909,    2.742,   -1.606,   46.543])
```

```
In [549]: sigma # covariance for location and velocity
Out[549]: array([[  58.597,    0.   ,    0.   ,  190.567,    0.   ,    0.   ],
                 [   0.   ,   58.597,    0.   ,    0.   ,  190.567,    0.   ],
                 [   0.   ,    0.   ,   58.597,    0.   ,    0.   ,  190.567],
                 [ 190.567,    0.   ,    0.   ,  866.044,    0.   ,    0.   ],
                 [   0.   ,  190.567,    0.   ,    0.   ,  866.044,    0.   ],
                 [   0.   ,    0.   ,  190.567,    0.   ,    0.   ,  866.044]])
```

```
In [605]: plt.scatter(x,z,alpha=1.0,s=2) # blue, start point
          plt.scatter(x_end,z_end,alpha=1.0,s=2) # green, end point
          plt.scatter(xo,zo,alpha=1.0,s=220) # red, observer
Out[605]: <matplotlib.collections.PathCollection at 0x1b24597a780>
```



P=0.000

Collision Detection

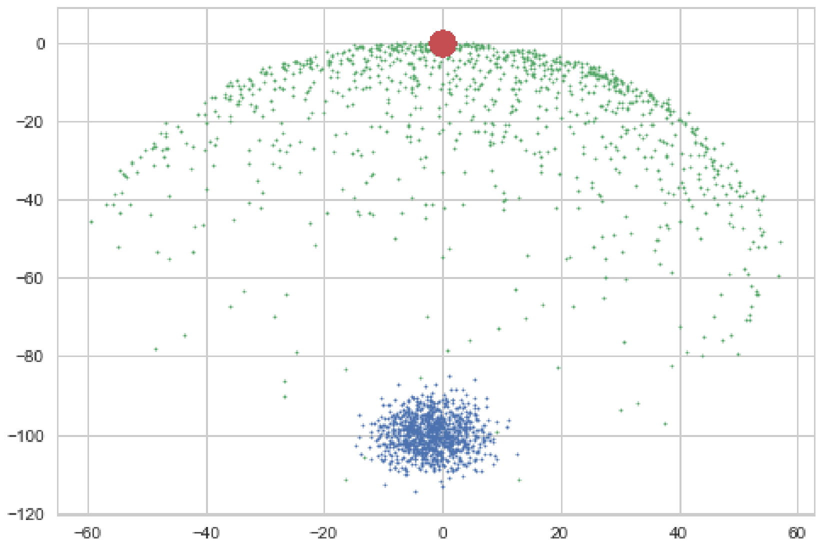
Example

T=52.00 sec

```
In [615]: mu # mean for location and velocity
Out[615]: array([ -2.018,   1.065, -99.876,   1.916,  -1.118,  17.396])
```

```
In [617]: sigma # covariance for location and velocity
Out[617]: array([[ 20.762,   0.   ,   0.   ,  21.328,   0.   ,   0.   ],
                 [  0.   ,  20.762,   0.   ,   0.   ,  21.328,   0.   ],
                 [  0.   ,   0.   ,  20.762,   0.   ,   0.   ,  21.328],
                 [ 21.328,   0.   ,   0.   ,  29.621,   0.   ,   0.   ],
                 [  0.   ,  21.328,   0.   ,   0.   ,  29.621,   0.   ],
                 [  0.   ,   0.   ,  21.328,   0.   ,   0.   ,  29.621]])
```

```
In [639]: plt.scatter(x,z,alpha=1.0,s=2) # blue, start point
          plt.scatter(x_end,z_end,alpha=1.0,s=2) # green, end point
          plt.scatter(xo,zo,alpha=1.0,s=220) # red, observer
Out[639]: <matplotlib.collections.PathCollection at 0x1b245cb0eb8>
```



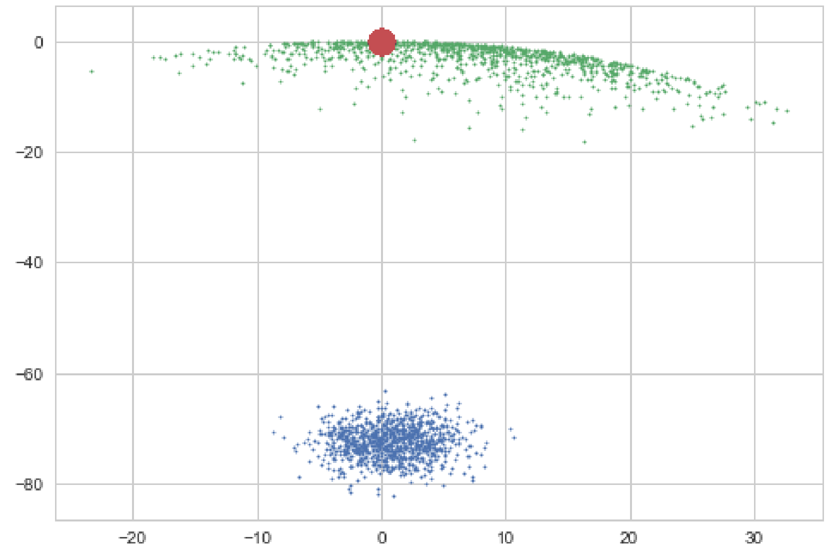
P=0.003

Collision Detection

Example

T=54.00 sec

```
In [661]: mu # mean for location and velocity
Out[661]: array([  0.83 ,  -1.716, -72.372,   1.544,  -1.332,  14.808])

In [663]: sigma # covariance for location and velocity
Out[663]: array([[ 9.038,  0.   ,  0.   ,  3.918,  0.   ,  0.   ],
                 [ 0.   ,  9.038,  0.   ,  0.   ,  3.918,  0.   ],
                 [ 0.   ,  0.   ,  9.038,  0.   ,  0.   ,  3.918],
                 [ 3.918,  0.   ,  0.   ,  2.278,  0.   ,  0.   ],
                 [ 0.   ,  3.918,  0.   ,  0.   ,  2.278,  0.   ],
                 [ 0.   ,  0.   ,  3.918,  0.   ,  0.   ,  2.278]])

In [685]: plt.scatter(x,z,alpha=1.0,s=2) # blue, start point
          plt.scatter(x_end,z_end,alpha=1.0,s=2) # green, end point
          plt.scatter(xo,zo,alpha=1.0,s=220) # red, observer
Out[685]: <matplotlib.collections.PathCollection at 0x1b2455e9f28>
```



P=0.019

Collision Detection

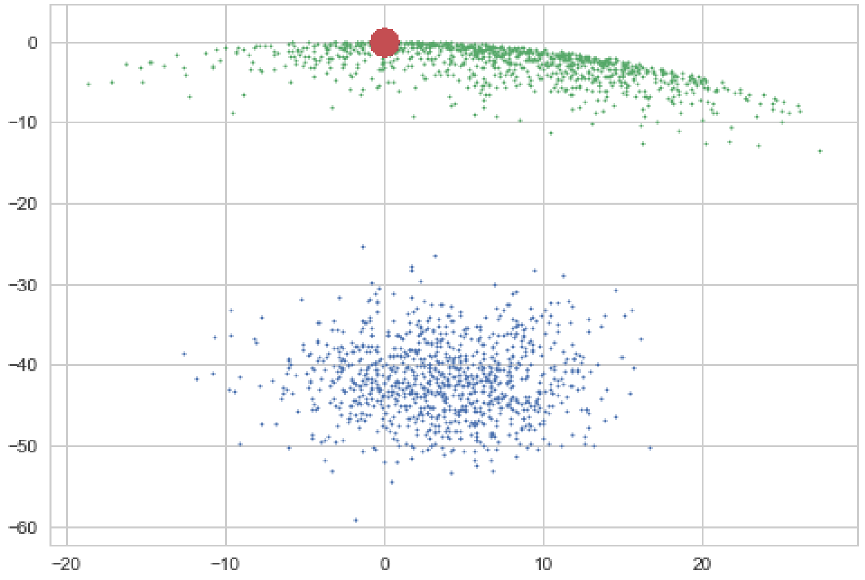
Example

T=56.00 sec

```
In [695]: mu # mean for location and velocity

Out[695]: array([  3.616,  -4.639, -41.828,   1.473,  -1.393,  15.026])
```

```
In [697]: sigma # covariance for location and velocity

Out[697]: array([[ 22.427,   0.   ,   0.   ,   5.785,   0.   ,   0.   ],
                 [  0.   ,  22.427,   0.   ,   0.   ,   5.785,   0.   ],
                 [  0.   ,   0.   ,  22.427,   0.   ,   0.   ,   5.785],
                 [  5.785,   0.   ,   0.   ,   1.644,   0.   ,   0.   ],
                 [  0.   ,   5.785,   0.   ,   0.   ,   1.644,   0.   ],
                 [  0.   ,   0.   ,   5.785,   0.   ,   0.   ,   1.644]])
```

```
In [719]: plt.scatter(x,z,alpha=1.0,s=2) # blue, start point
          plt.scatter(x_end,z_end,alpha=1.0,s=2) # green, end point
          plt.scatter(xo,zo,alpha=1.0,s=220) # red, observer

Out[719]: <matplotlib.collections.PathCollection at 0x1b245e5a7f0>
```



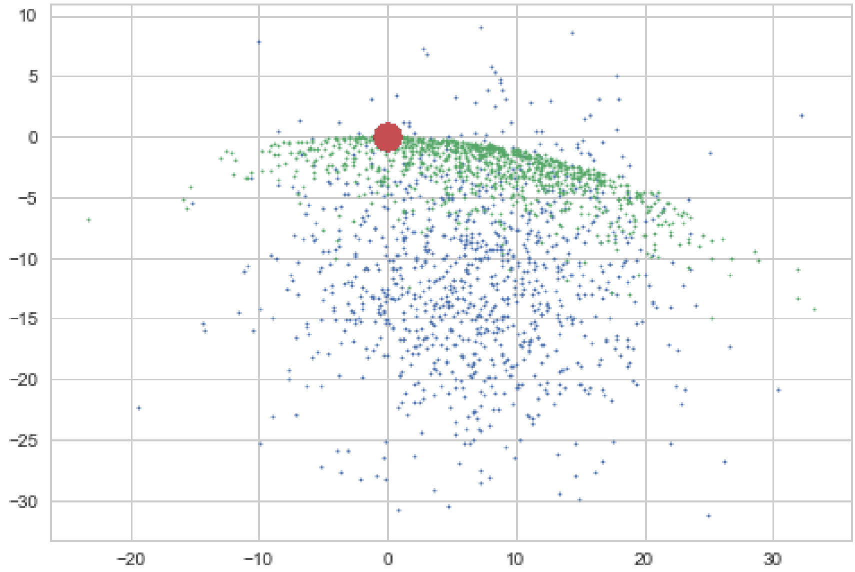
P=0.027

Collision Detection

Example

T=58.00 sec

```
In [729]: mu # mean for location and velocity

Out[729]: array([  6.562,  -7.425, -11.777,   1.473,  -1.393,  15.026])

In [731]: sigma # covariance for location and velocity

Out[731]: array([[ 52.143,   0.   ,   0.   ,   9.073,   0.   ,   0.   ],
                 [  0.   ,  52.143,   0.   ,   0.   ,   9.073,   0.   ],
                 [  0.   ,   0.   ,  52.143,   0.   ,   0.   ,   9.073],
                 [  9.073,   0.   ,   0.   ,   1.644,   0.   ,   0.   ],
                 [  0.   ,   9.073,   0.   ,   0.   ,   1.644,   0.   ],
                 [  0.   ,   0.   ,   9.073,   0.   ,   0.   ,   1.644]])

In [753]: plt.scatter(x,z,alpha=1.0,s=2) # blue, start point
          plt.scatter(x_end,z_end,alpha=1.0,s=2) # green, end point
          plt.scatter(xo,zo,alpha=1.0,s=220) # red, observer

Out[753]: <matplotlib.collections.PathCollection at 0x1b24716e4a8>
```



P=0.019

Collision Detection

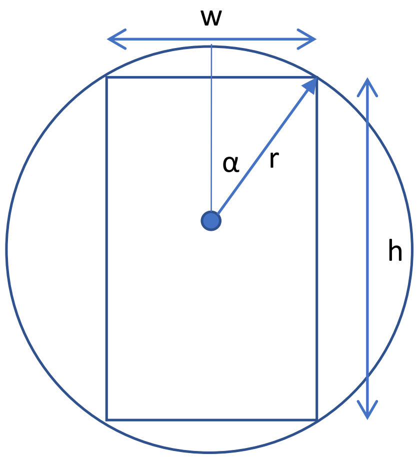
Open questions:

- How to make uncertainty smaller?
- Collision matrix reachable?

Collision Detection

Side effect: instead of using height in distance estimation, radius of enclosing circle will be used:

From bounding box coordinates to radius:



$$r = \sqrt{(\frac{w}{2})^2 + (\frac{h}{2})^2}$$

$$h = (ymax - ymin) \quad c_y = (ymax + ymin)/2$$
$$w = (xmax - xmin) \quad c_x = (xmax + xmin)/2$$

Collision Detection

Distance estimation using object radius:

$$d = \frac{f * h}{\cos(\alpha) * \cos(\beta) * h_i * s_h/p_h}$$

s_w= *sensor width* (m)
s_h= *sensor height* (m)
p_w= *image width (pixels)*
p_h= *image height (pixels)*
h_i = *object height (pixels)*
h = object height (m)
f = focal length (m)
α = *altitude (rad)*
β = *azimuth (rad)*

Before

$$d = \frac{f * r}{\cos(\alpha) * \cos(\beta) * r_i * s_h/p_h}$$

s_w= *sensor width* (m)
s_h= *sensor height* (m)
p_w= *image width (pixels)*
p_h= *image height (pixels)*
r_i = *object radius (pixels)*
r = object radius (m)
f = focal length (m)
α = *altitude (rad)*
β = *azimuth (rad)*

Now

Collision Detection

Distance estimation using object radius:

$$t = \frac{d}{\sqrt{{x_c}^2 + {y_c}^2 + {z_c}^2}}$$

$$(x_c, y_c, z_c) = \left(-\frac{s_w}{2} + xp * \frac{s_w}{p_w},\ \ \frac{s_h}{2} - yp * \frac{s_h}{p_h},\ -f\right)$$

$$(x_o, y_o, z_o) = t * (x_c, y_c, z_c)$$

Collision Detection

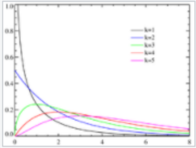
Body radius distribution

Distibution should:

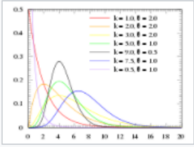
- be defined in [0,∞]
- mode > 0
- simple
- skew controllable

Supported on semi-infinite intervals, usually [0,∞) [edit]

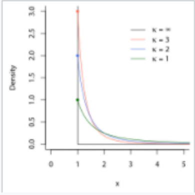
- The Beta prime distribution
- The Birnbaum–Saunders distribution, also known as the fatigue life distribution, is a probability distribution used extensively in reliability applications to model failure times.
- The chi distribution
  - The noncentral chi distribution
- The chi-squared distribution, which is the sum of the squares of *n* independent Gaussian random variables. It is a special case of the Gamma distribution, and it is used in goodness-of-fit tests in statistics.
  - The inverse-chi-squared distribution
  - The noncentral chi-squared distribution
  - The Scaled inverse chi-squared distribution
- The Dagum distribution
- The exponential distribution, which describes the time between consecutive rare random events in a process with no memory.
- The Exponential-logarithmic distribution
- The F-distribution, which is the distribution of the ratio of two (normalized) chi-squared-distributed random variables, used in the analysis of variance. It is referred to as the beta prime distribution when it is the ratio of two chi-squared variates which are not normalized by dividing them by their numbers of degrees of freedom.
  - The noncentral F-distribution
- The folded normal distribution
- The Fréchet distribution
- The Gamma distribution, which describes the time until *n* consecutive rare random events occur in a process with no memory.
  - The Erlang distribution, which is a special case of the gamma distribution with integral shape parameter, developed to predict waiting times in queuing systems
  - The inverse-gamma distribution
- The Generalized gamma distribution
- The generalized Pareto distribution
- The Gamma/Gompertz distribution
- The Gompertz distribution
- The half-normal distribution
- Hotelling's T-squared distribution
- The inverse Gaussian distribution, also known as the Wald distribution
- The Lévy distribution
- The log-Cauchy distribution
- The log-Laplace distribution
- The log-logistic distribution
- The log-normal distribution, describing variables which can be modelled as the product of many small independent positive variables.
- The Lomax distribution
- The Mittag-Leffler distribution
- The Nakagami distribution
- The Pareto distribution, or "power law" distribution, used in the analysis of financial data and critical behavior.
- The Pearson Type III distribution
- The Phase-type distribution, used in queueing theory
- The phased bi-exponential distribution is commonly used in pharmokinetics
- The phased bi-Weibull distribution
- The Rayleigh distribution
- The Rayleigh mixture distribution
- The Rice distribution
- The shifted Gompertz distribution
- The type-2 Gumbel distribution
- The Weibull distribution or Rosin Rammler distribution, of which the exponential distribution is a special case, is used to model the lifetime of technical devices and is used to describe the particle size distribution of particles generated by grinding, milling and crushing operations.



Chi-squared distribution



Gamma distribution

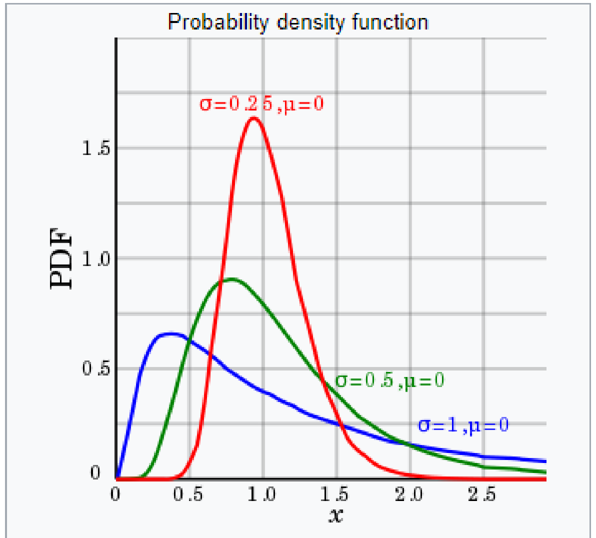


Pareto distribution

Collision Detection

Log-normal distribution for body radius:

Log-normal



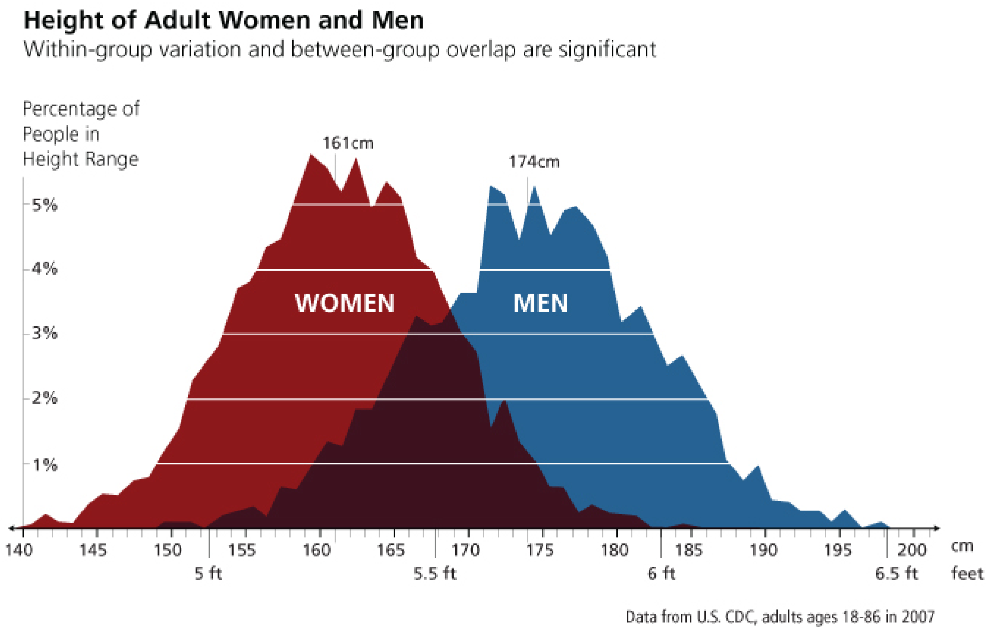
Some log-normal density functions with identical parameter μ but differing parameters σ

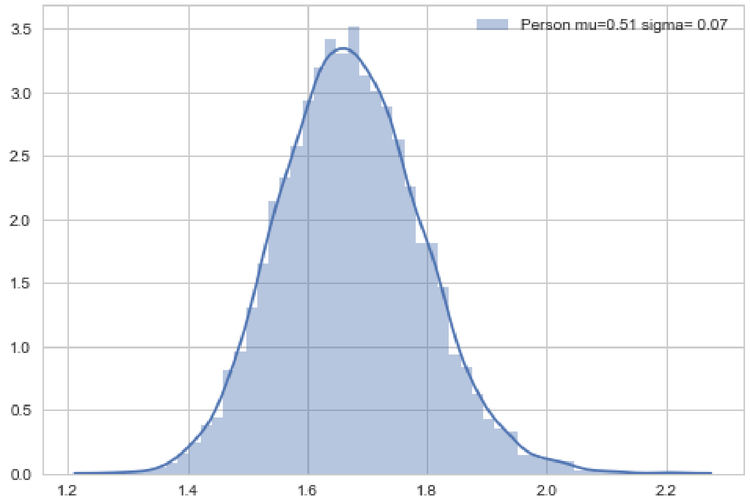
Used in the context of describing human height distribution

Notation	$\mathrm{Lognormal}(\mu,\,\sigma^2)$
Parameters	$\mu \in (-\infty, +\infty)$, $\sigma > 0$
Support	$x \in (0, +\infty)$
PDF	$\frac{1}{x\sigma\sqrt{2\pi}}\, e^{-\frac{(\ln x-\mu)^2}{2\sigma^2}}$
CDF	$\frac{1}{2} + \frac{1}{2}\,\mathrm{erf}\Big[\frac{\ln x - \mu}{\sqrt{2}\sigma}\Big]$
Mean	$\exp\left(\mu + \frac{\sigma^2}{2}\right)$
Median	$\exp(\mu)$
Mode	$\exp(\mu - \sigma^2)$
Variance	$[\exp(\sigma^2) - 1]\exp(2\mu + \sigma^2)$
Skewness	$(e^{\sigma^2} + 2)\sqrt{e^{\sigma^2} - 1}$
Ex. kurtosis	$\exp(4\sigma^2) + 2\exp(3\sigma^2) + 3\exp(2\sigma^2) - 6$
Entropy	$\log(\sigma e^{\mu+\frac{1}{2}}\sqrt{2\pi})$
MGF	defined only for numbers with a non-positive real part, see text
CF	representation $\sum_{n=0}^{\infty}\frac{(it)^n}{n!}e^{n\mu+n^2\sigma^2/2}$ is asymptotically divergent but sufficient for numerical purposes
Fisher information	$\begin{pmatrix} 1/\sigma^2 & 0 \\ 0 & 1/2\sigma^4 \end{pmatrix}$

Collision Detection

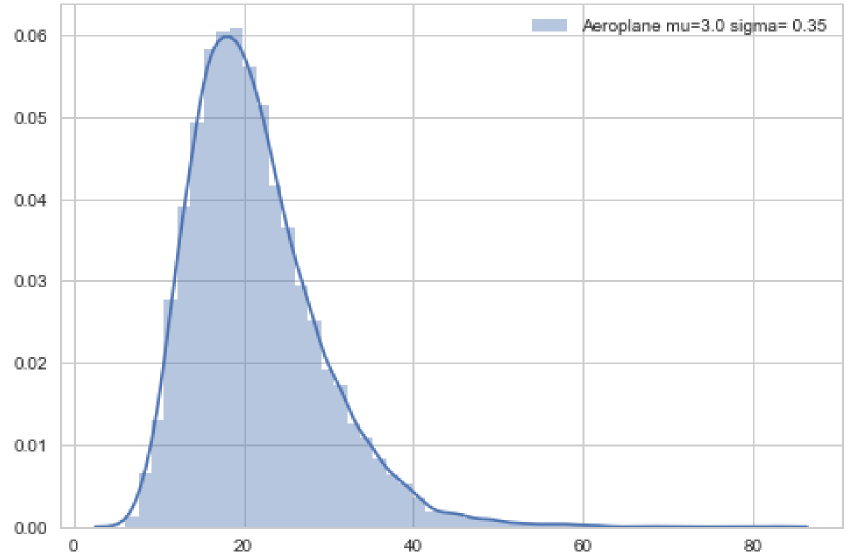
Example: Person

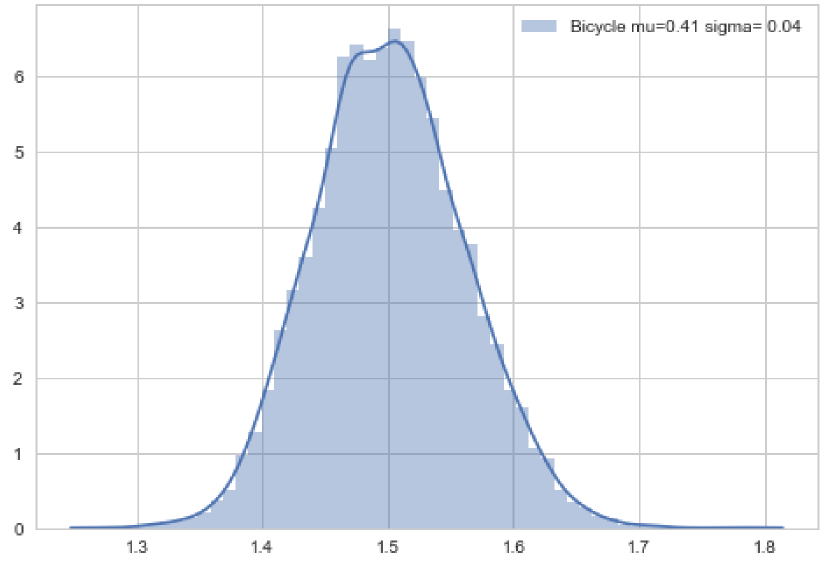




Collision Detection

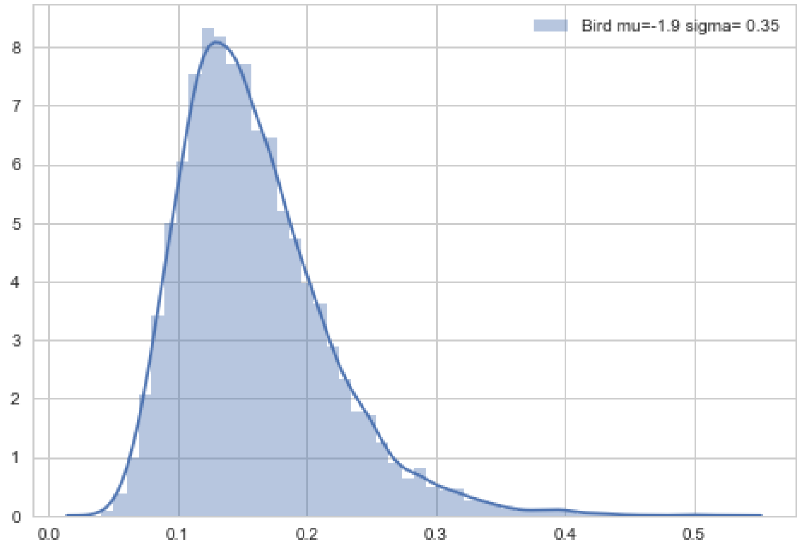
Bubble diameters (2*radius)

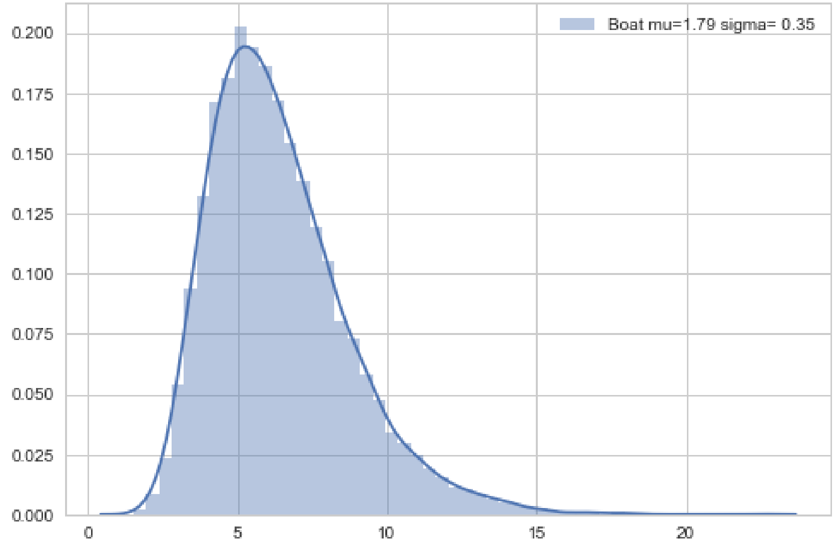




Collision Detection

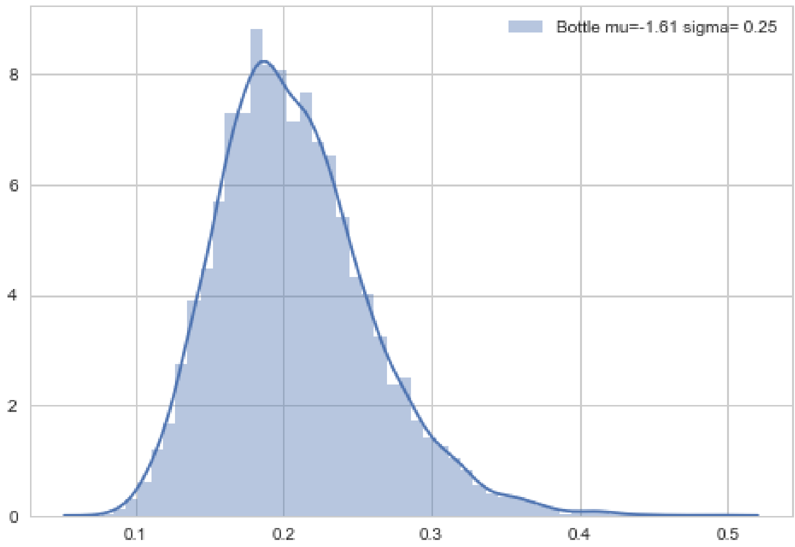
Bubble diameters (2*radius)

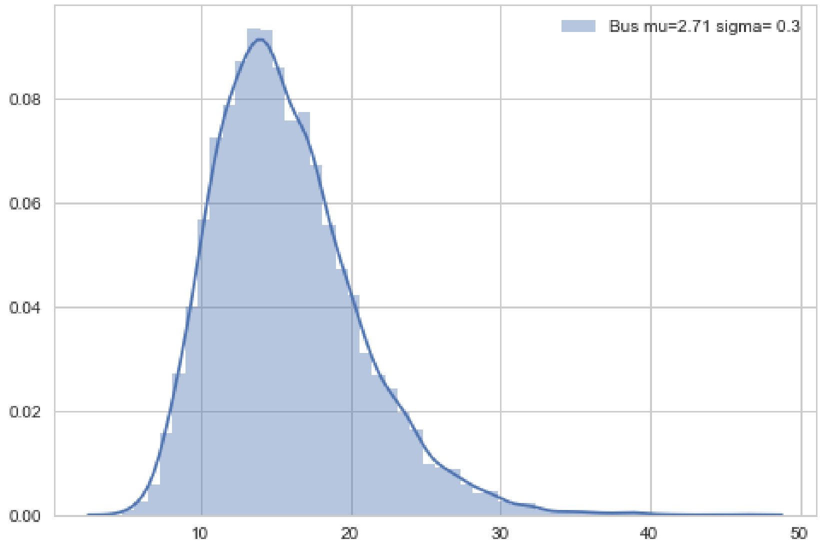




Collision Detection

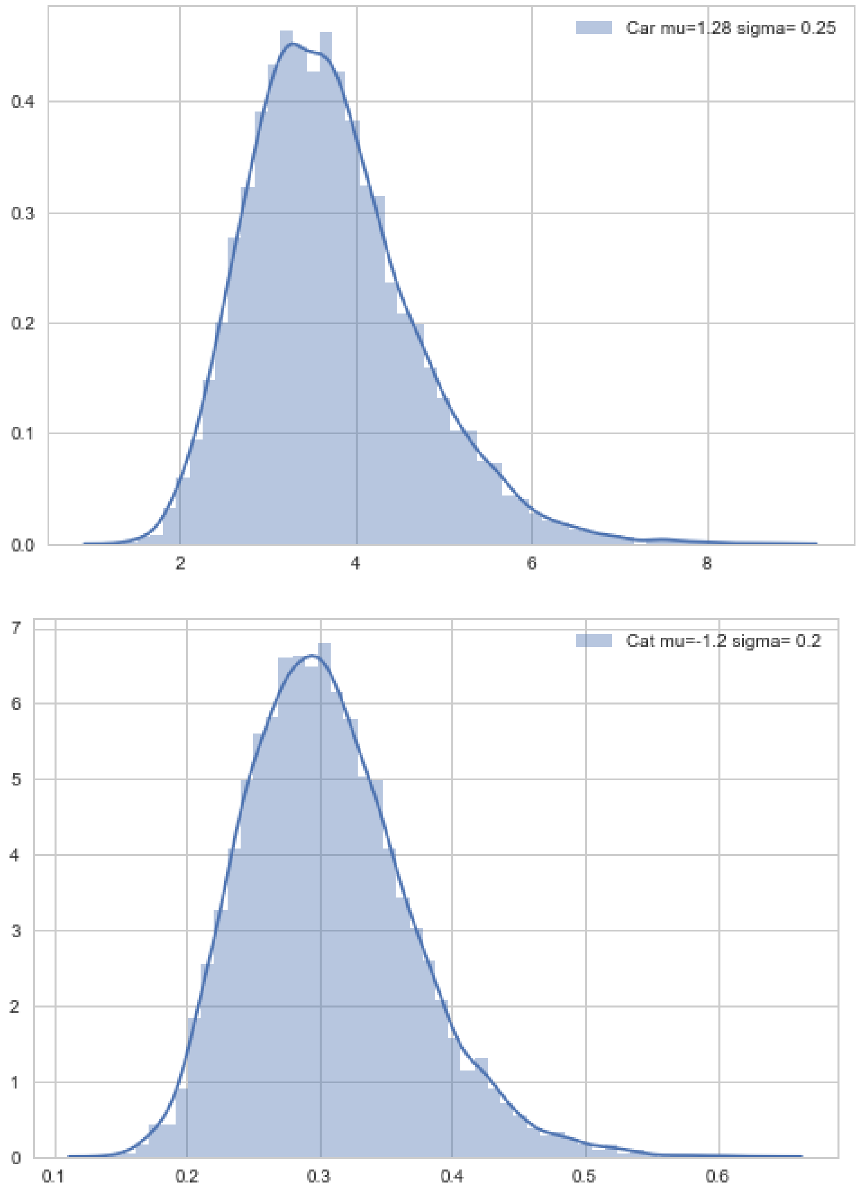
Bubble diameters (2*radius)





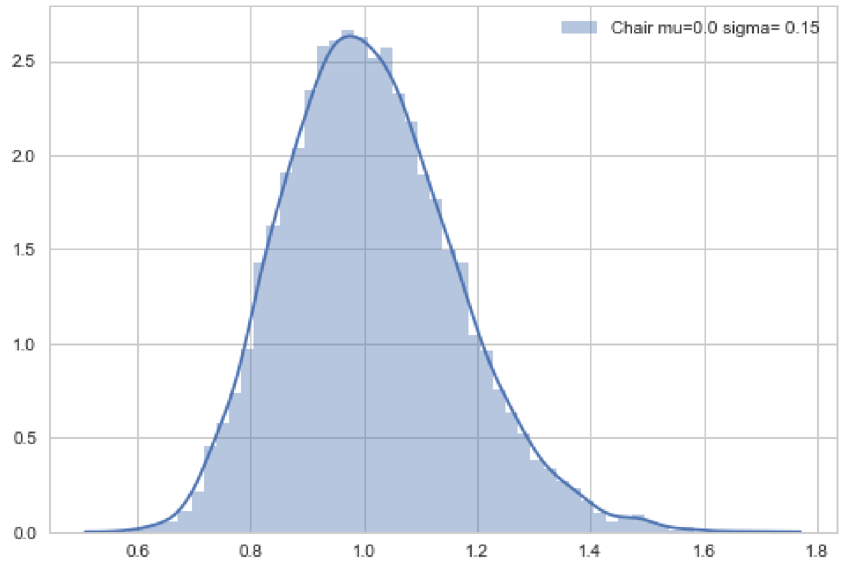
Collision Detection

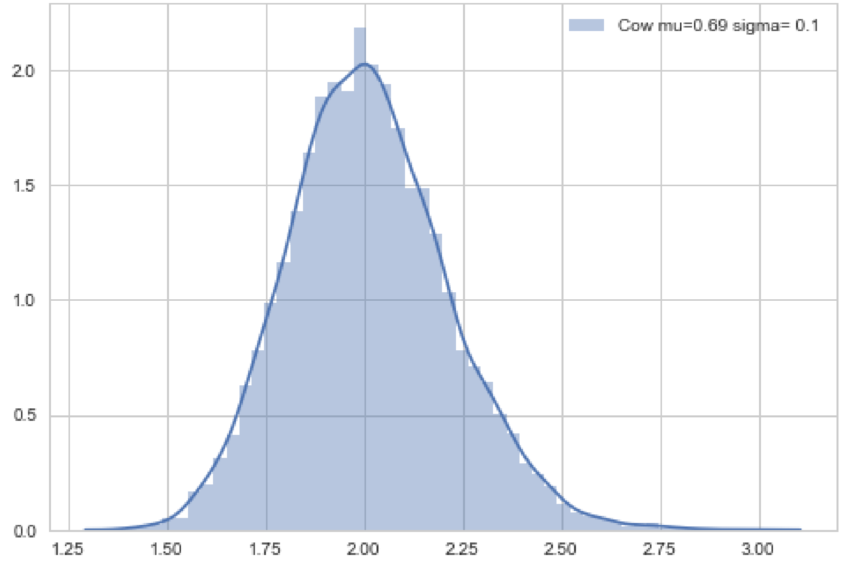
Bubble diameters (2*radius)



Collision Detection

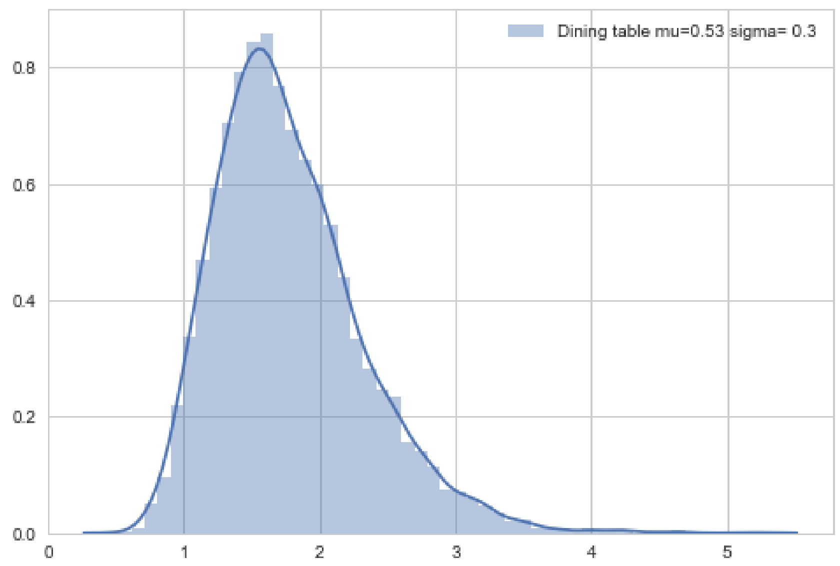
Bubble diameters (2*radius)

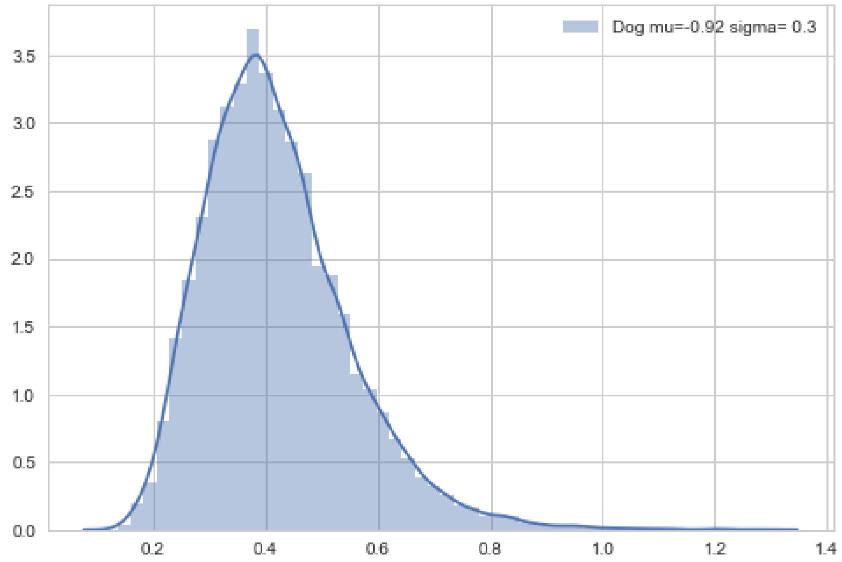




Collision Detection

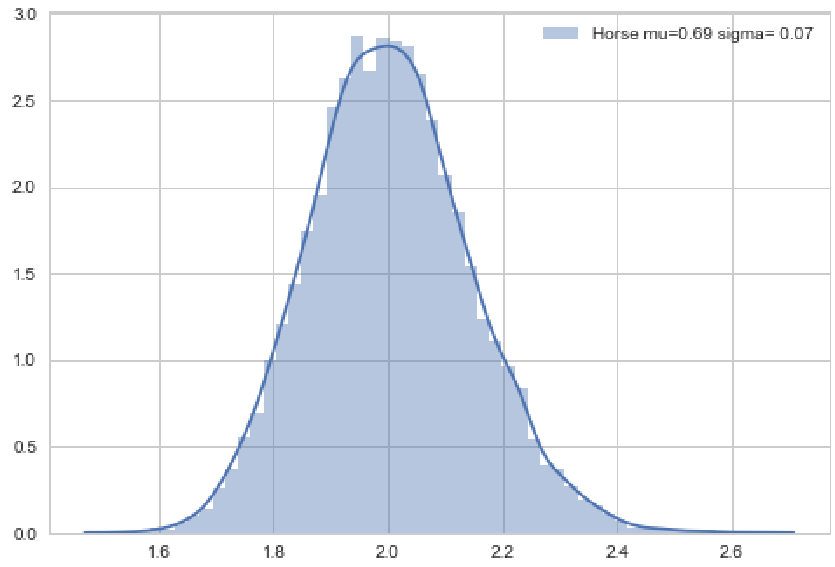
Bubble diameters (2*radius)

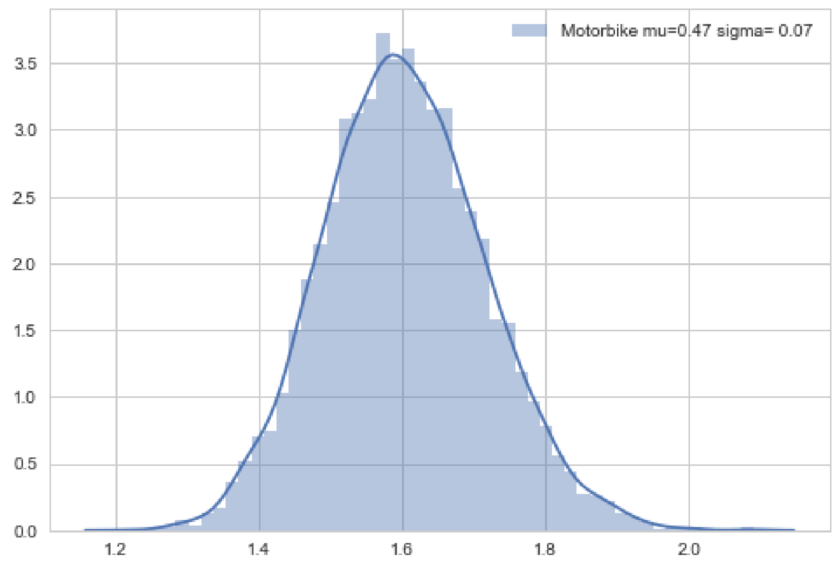




Collision Detection

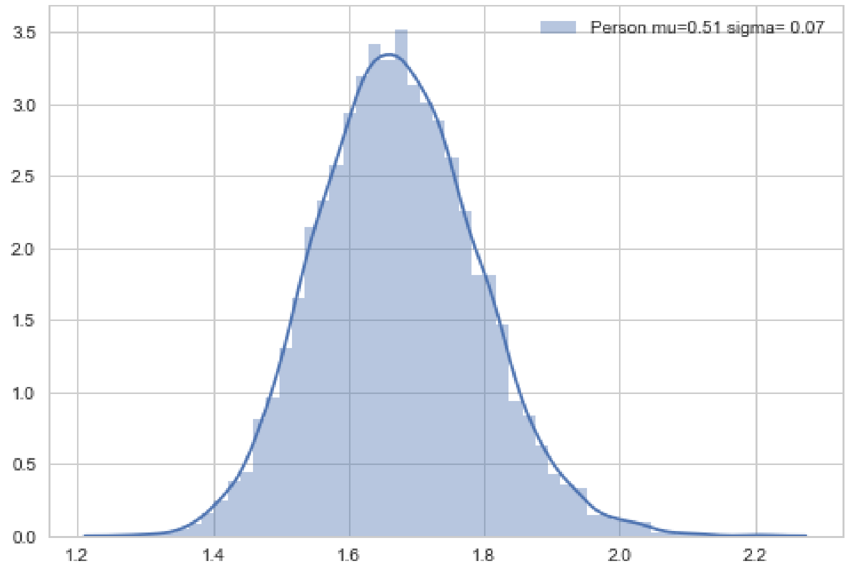
Bubble diameters (2*radius)

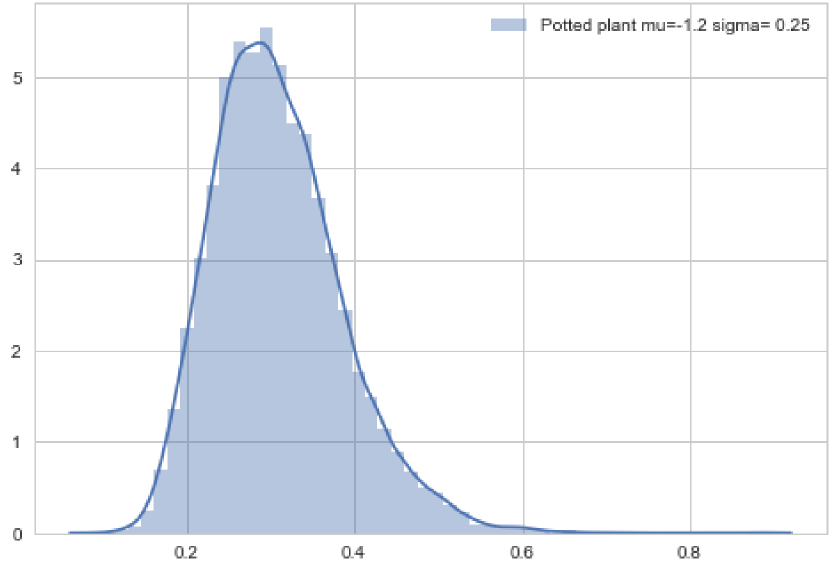




Collision Detection

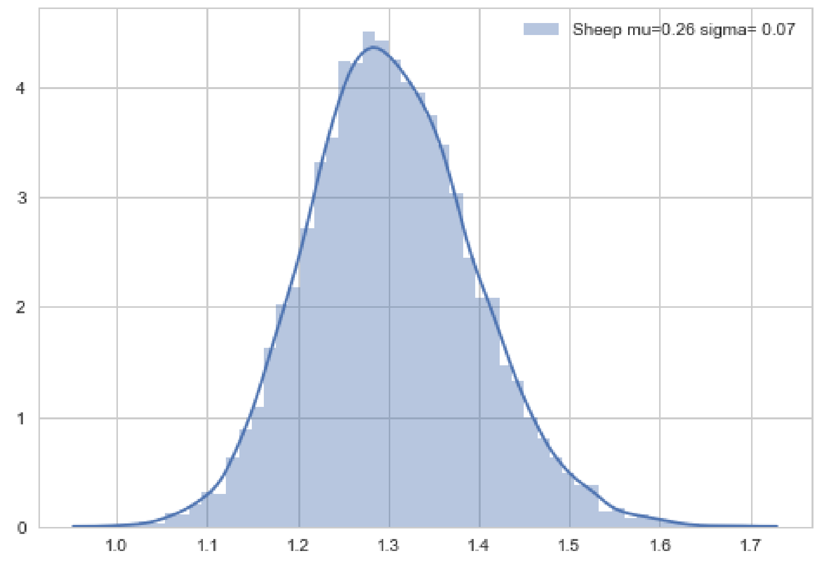
Bubble diameters (2*radius)

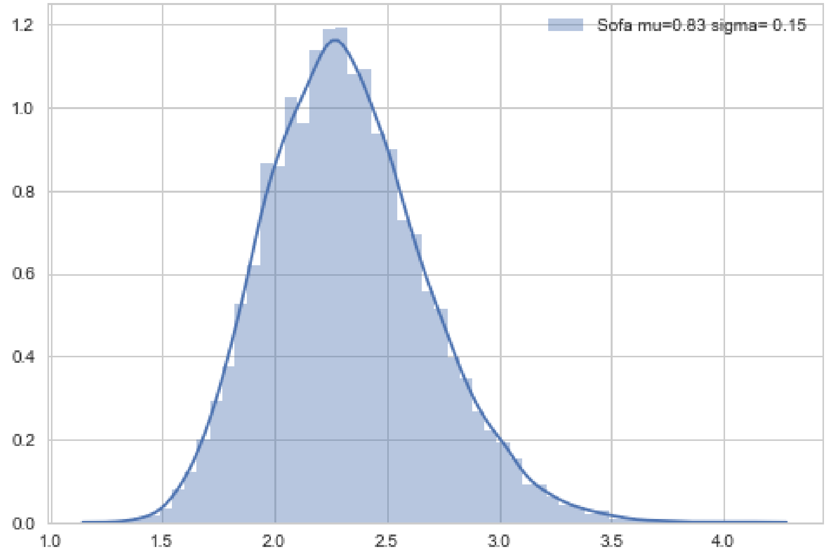




Collision Detection

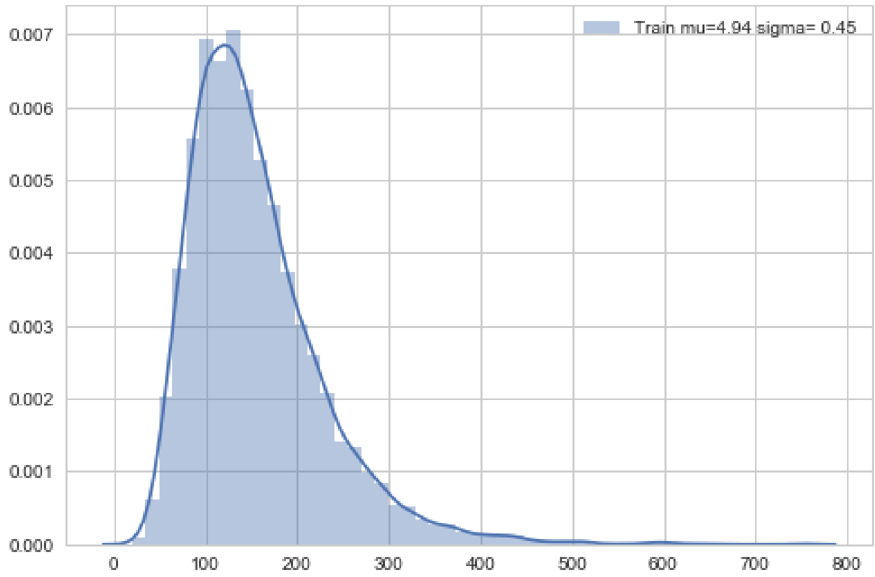
Bubble diameters (2*radius)

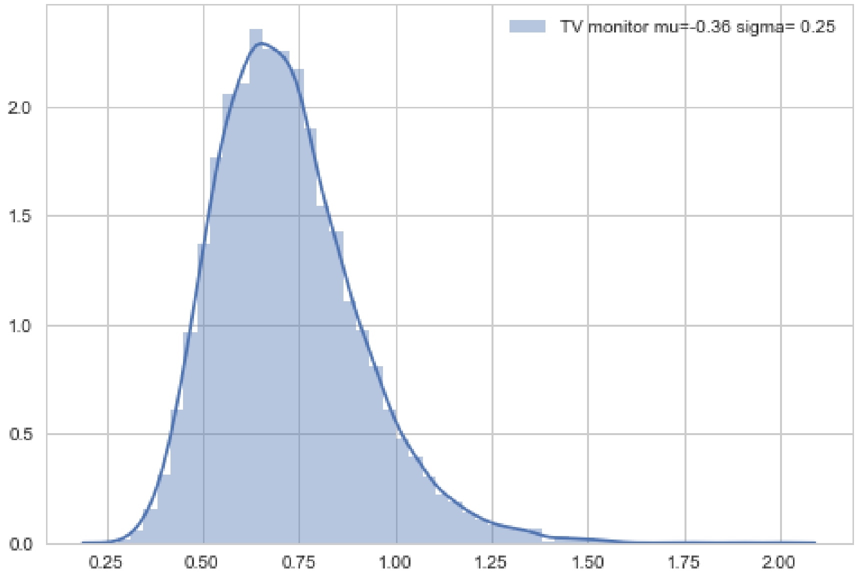




Collision Detection

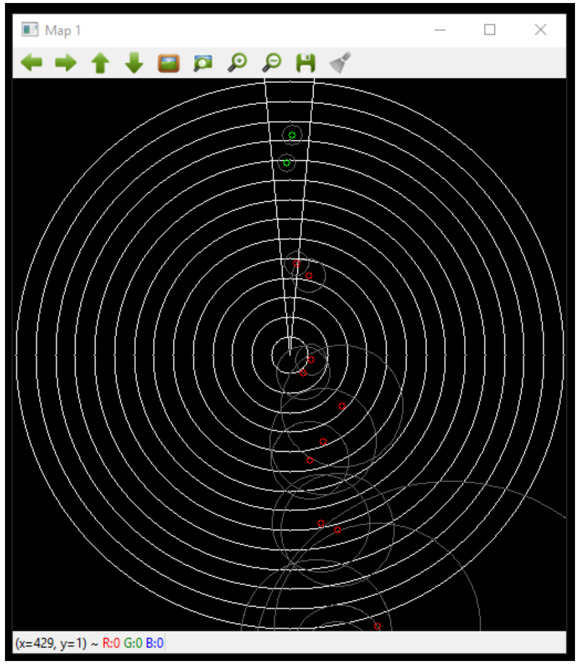
Bubble diameters (2*radius)





Prediction

Path chosen - a simpler solution which provides a lot more information



Kalman filter update:

$$\mu_1(k) = A * \mu(k-1)$$

$$\Sigma_1(k) = A * \Sigma(k-1) * A^T + R$$

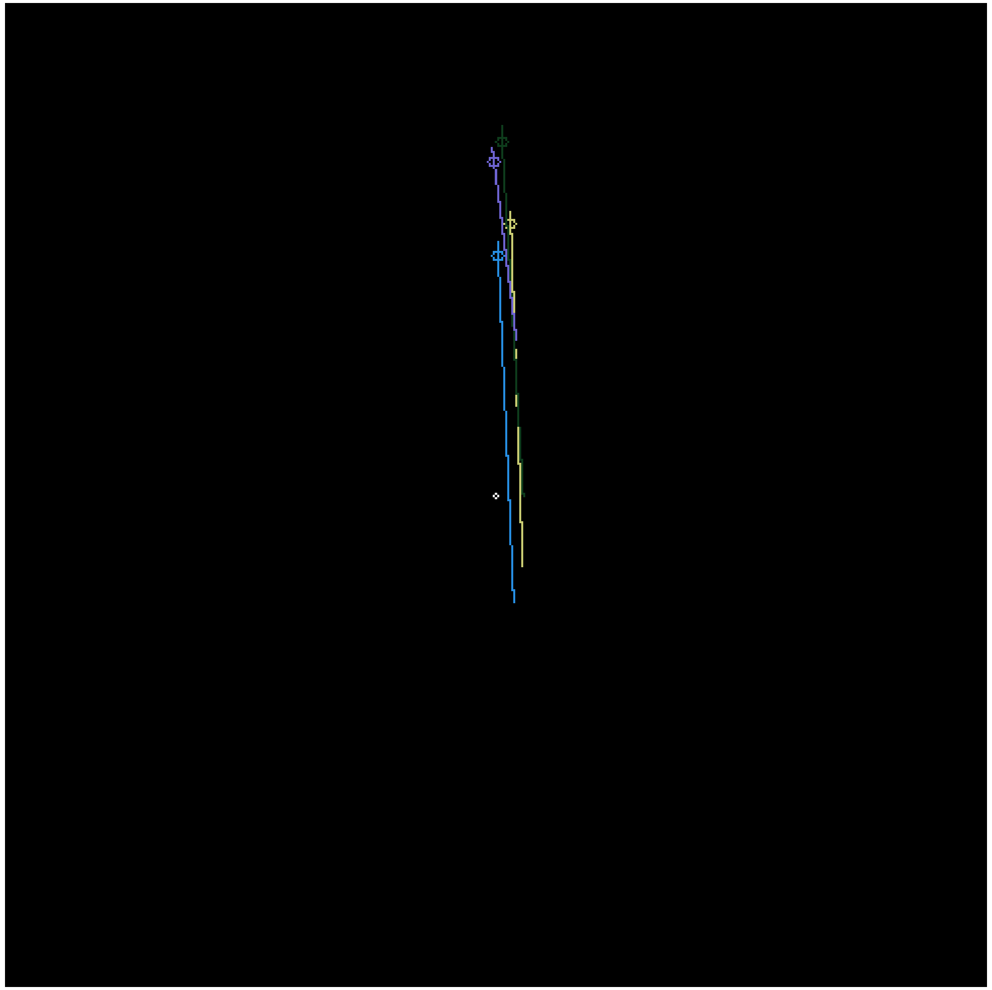
Update is done n times

$$A = \begin{bmatrix} 1 & 0 & 0 & \Delta & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mu = \begin{bmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{bmatrix}$$

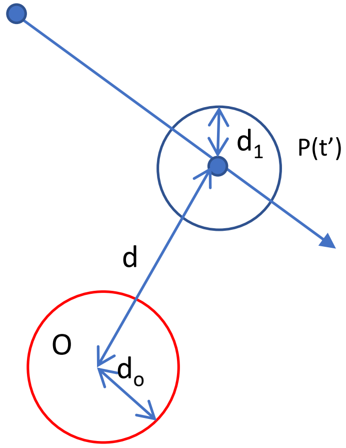
Δ has to be smaller than 1/fps. If fps=25, 1/fps = 0.04 sec. A car driving 120 km/h will proceed 1.33 meters and a collision with an observer might not be detected well enough. A value of Δ = 0.01 corresponds to the movement of 33 cm for an object moving at 120 km/h. This will generate 100 predictions per second. If we want the prediction horizon to be 10 secs, we have 1000 predictions per object. Predicting for every frame will generate fps*1000 ≈ 25 000 predictions per second per object. This is too much, so only current prediction is saved.

Prediction



Collision Detection

Collision with the observer



$$x(t) = x(0) + v_x(0) * t$$
$$y(t) = y(0) + v_y(0) * t$$
$$z(t) = z(0) + v_z(0) * t$$

$$d^2(t) = x(t)^2 + y(t)^2 + z(t)^2$$

$$d^2 = (x(0) + v_x(0) * t\,)^2 + (y(0) + v_y(0) * t\,)^2 + (z(0) + v_z(0) * t)^2$$

$$\frac{d(d(t)^2)}{dt} = 2 * (x(0) + v_x(0) * t\,) * v_x(0) + 2 * \left(y(0) + v_y(0) * t\,\right) * v_y(0) + 2 * (z(0) + v_z(0) * t\,) * v_z(0) = 0$$

$$t' = -\frac{x(0)*v_x(0)+y(0)*v_y(0)+z(0)*v_z(0)}{v_x(0)^2+v_y(0)^2+v_z(0)^2}$$

Work in Progress

Perception

“The first step in achieving SA is to perceive the status, attributes, and dynamics of relevant elements in the environment. Thus, Level 1 SA, the most basic level of SA, involves the processes of monitoring, cue detection, and simple recognition, which lead to an awareness of multiple situational elements (objects, events, people, systems, environmental factors) and their current states (locations, conditions, modes, actions).”

Next Steps

Next steps

Comprehension:

1. Closing the open questions
2. 2d -> 3d transformation
3. World object state estimation

To Be Discussed

To Be Discussed

- Activity recognition?
- Emotion recognition?
- Turning camera, estimation by background movement?

Thank you!

lampola@student.tut.fi
https://github.com/SakariLampola/Thesis