DTU

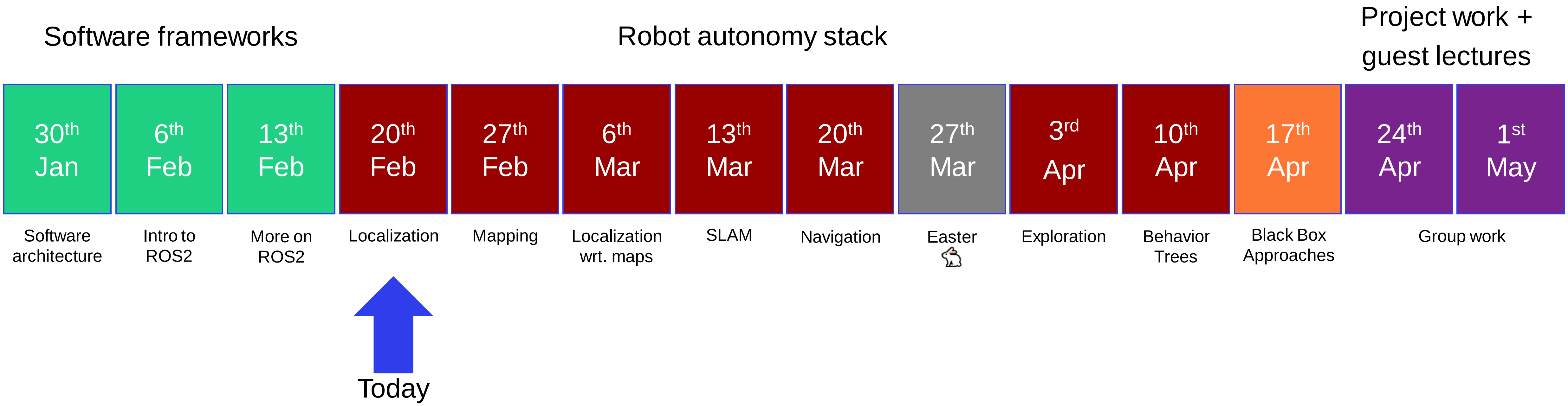
Rasmus Andersen

34761 – Robot Autonomy

(Robot) Localization

Overview of 34761 – Robot Autonomy

- 3 lectures on software frameworks
- 7 lectures on building your own autonomy stack for a mobile robot
- 1 lecture on DL/RL – an overview of black-box approaches to what you have done
- 2 lectures of project work before hand in + guest lectures



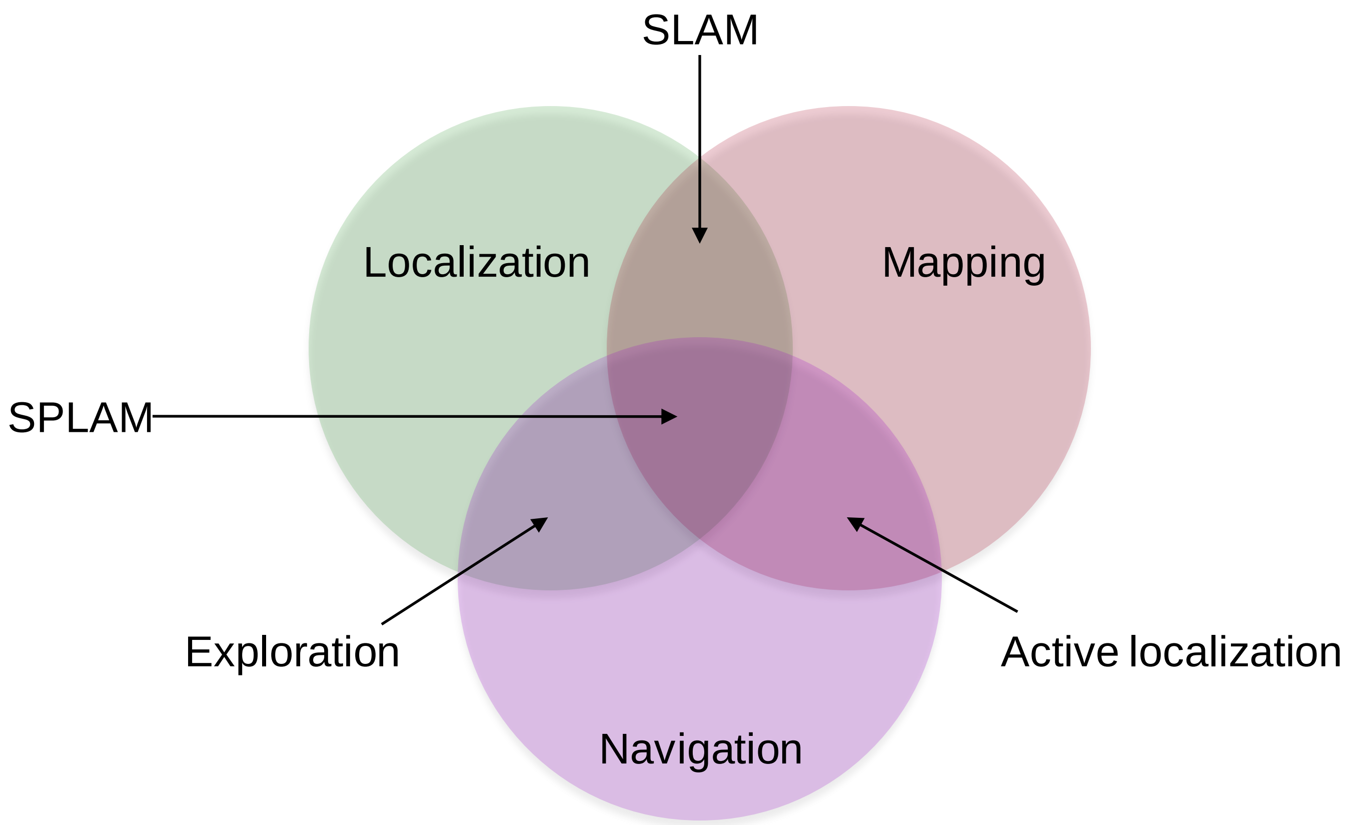
Software frameworks			Robot autonomy stack									Project work + guest lectures	
30th Jan	6th Feb	13th Feb	20th Feb	27th Feb	6th Mar	13th Mar	20th Mar	27th Mar	3rd Apr	10th Apr	17th Apr	24th Apr	1st May
Software architecture	Intro to ROS2	More on ROS2	Localization	Mapping	Localization wrt. maps	SLAM	Navigation	Easter 🐇	Exploration	Behavior Trees	Black Box Approaches	Group work	

Outline for the next 7 weeks

- Our own autonomy stack:

Topic of today

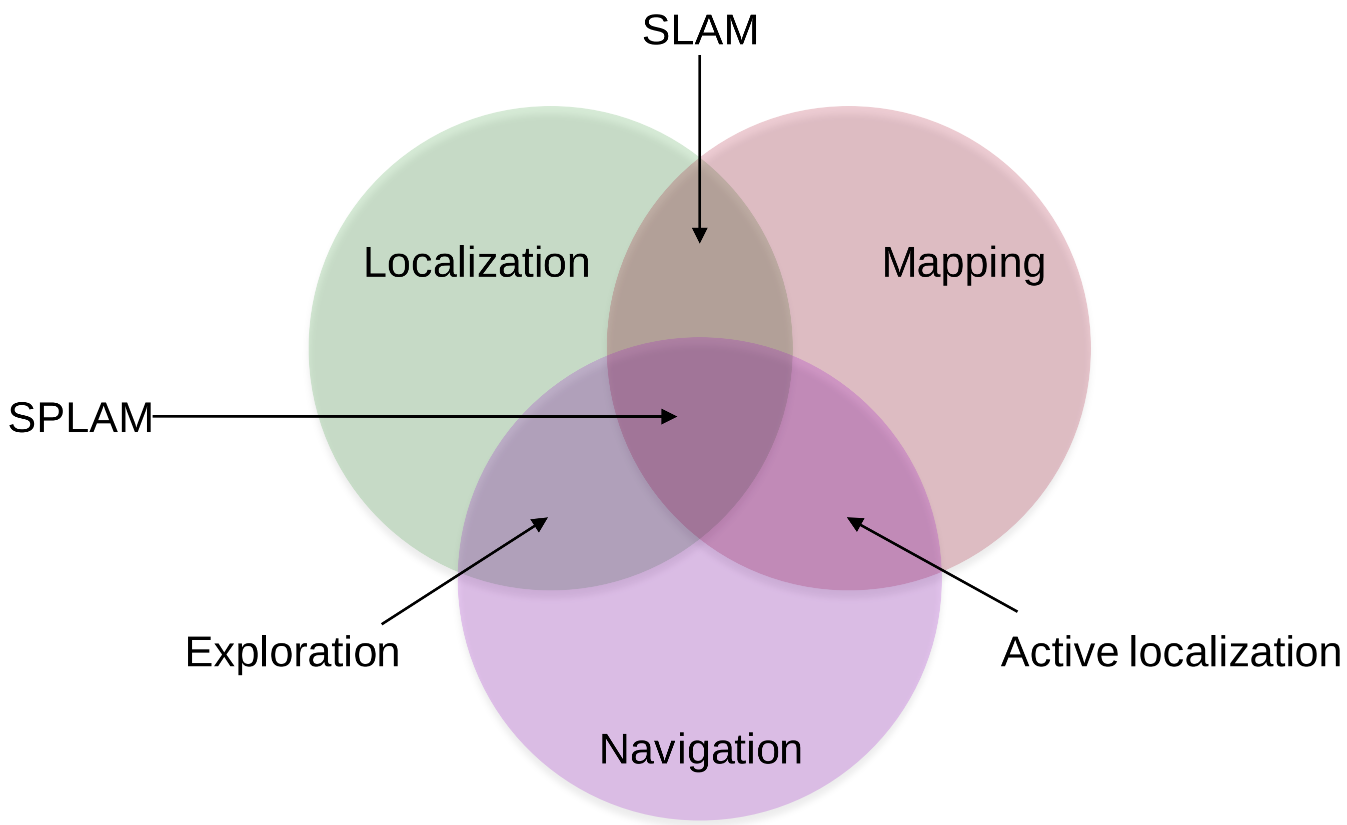
1. Localization
2. Mapping
3. Navigation
4. Exploration
5. Behaviour trees



Outline for the next 7 weeks

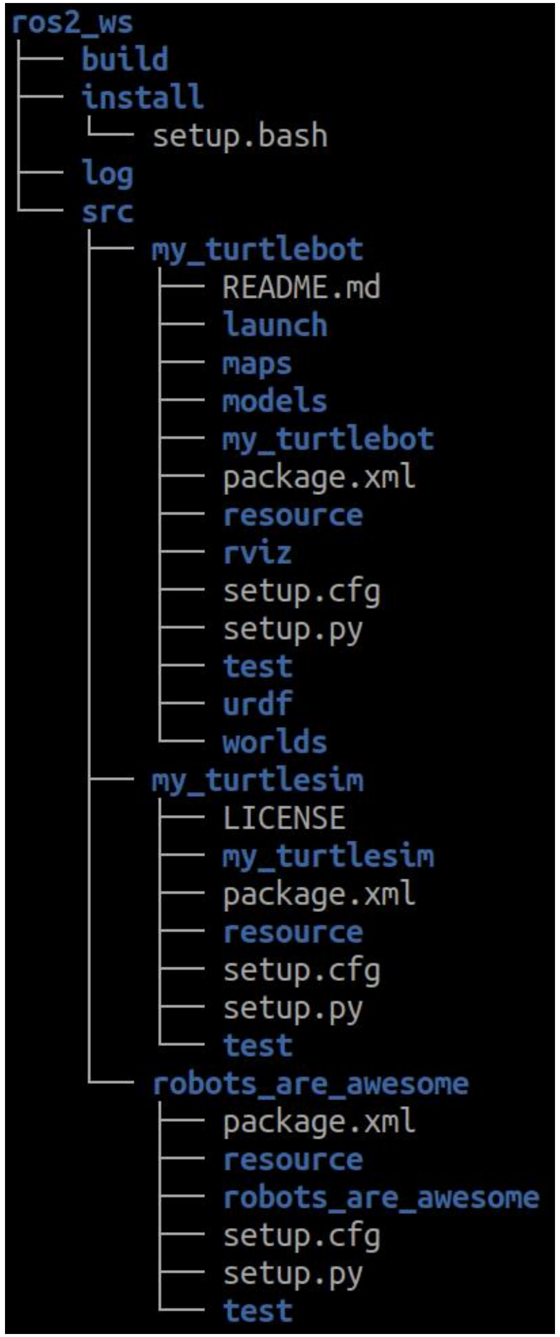
- Our own autonomy stack:
  1. Localization
     - Wheel Odometry
     - Visual Odometry
     - Visual Inertial Odometry
     - Iterative closest point
     - LIDAR Odometry
  2. Mapping
  3. Navigation
  4. Exploration
  5. Behaviour trees

Topic of today



Recall from last lecture

- A workspace consists of a **build**, **install**, **log**, and **src** folder
  - **The build directory**
    - Intermediate/temporary build files
  - **The install directory**
    - Where the final binaries, libraries, resources, etc. are stored
    - Never modify anything in this directory – it is intended to only be modified by the build system
  - **The log directory**
    - Default location for ROS2 logs
  - **The src directory**
    - Where we put everything to create nodes
    - The only directory we should modify anything in



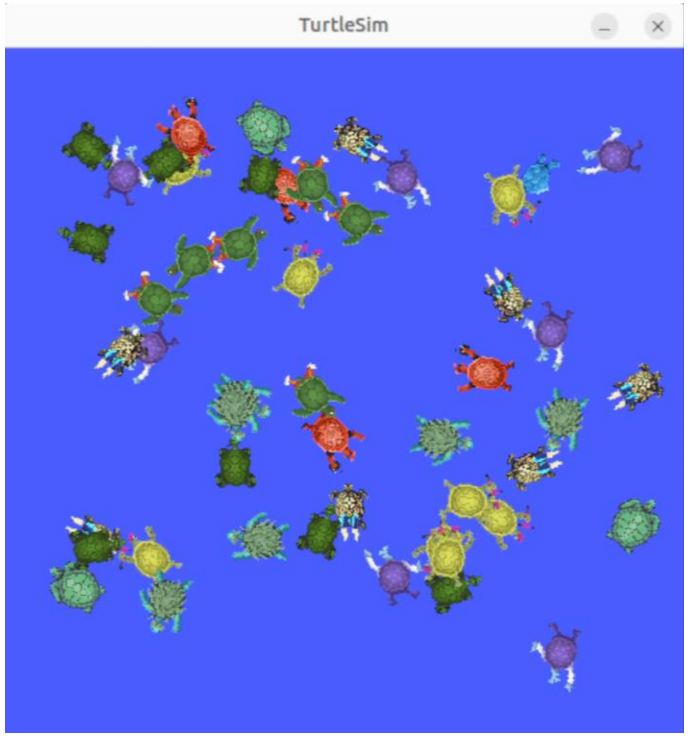
```
ros2_ws
├── build
├── install
│   └── setup.bash
├── log
└── src
    ├── my_turtlebot
    │   ├── README.md
    │   ├── launch
    │   ├── maps
    │   ├── models
    │   ├── my_turtlebot
    │   ├── package.xml
    │   ├── resource
    │   ├── rviz
    │   ├── setup.cfg
    │   ├── setup.py
    │   ├── test
    │   ├── urdf
    │   └── worlds
    ├── my_turtlesim
    │   ├── LICENSE
    │   ├── my_turtlesim
    │   ├── package.xml
    │   ├── resource
    │   ├── setup.cfg
    │   ├── setup.py
    │   └── test
    └── robots_are_awesome
        ├── package.xml
        ├── resource
        ├── robots_are_awesome
        ├── setup.cfg
        ├── setup.py
        └── test
```

Recall from last lecture

- Creating ROS nodes to control turtles
  - Anyone up for showing their solution?

- Launching the turtlebot simulation environment (for Mac users with ARM processors)
  - *docker run -p 6080:80 --security-opt seccomp=unconfined --shm-size=512m* **--platform linux/amd64** *tiryoh/ros2-desktop-vnc:humble*
  - Lecture 1 “Before the lecture” section has been updated
  - Your environment should rebuild
  - Be sure to backup your progress and copy it over to the new container

- Transforms

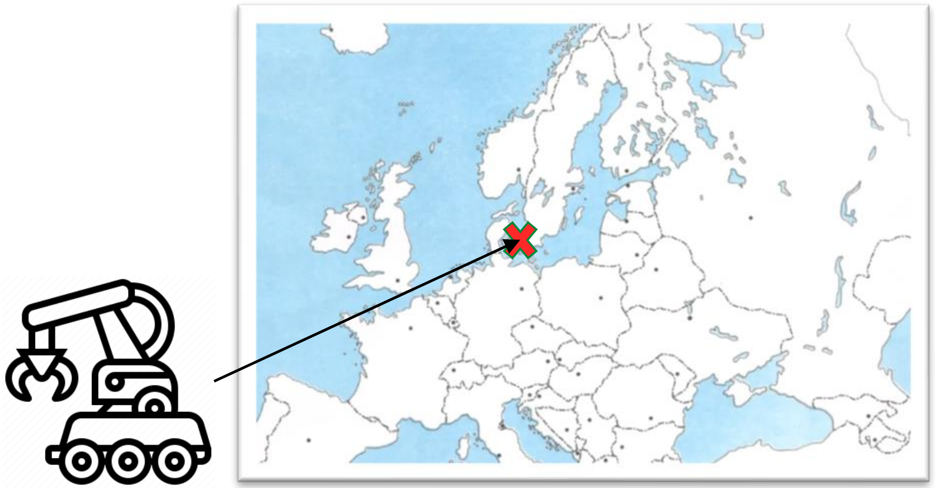


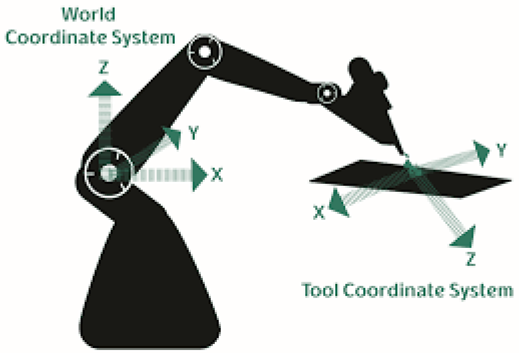
Three students, scripts awry, forgot to heed,
Setup.bash neglected, a crucial feed.
Before the class, they stand with dread,
Singing Rihanna's "Work" in hues of red.

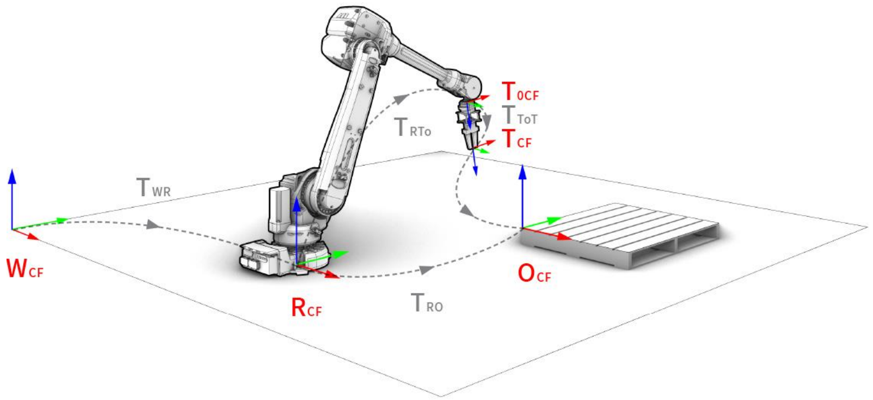
Karaoke♬ Work ft. Drake - Rihanna 【No Guide Melody】 Instrumental, Lyric, BGM

What does it mean to localize

- It depends..
  - What do we want to know the location of?
    - Ourselves
    - Our tool
    - An object
  - Localize with respect to what?
    - Do we need global or relative localization?
      - What kind of robot do we have?
      - What kind of sensors do we have?





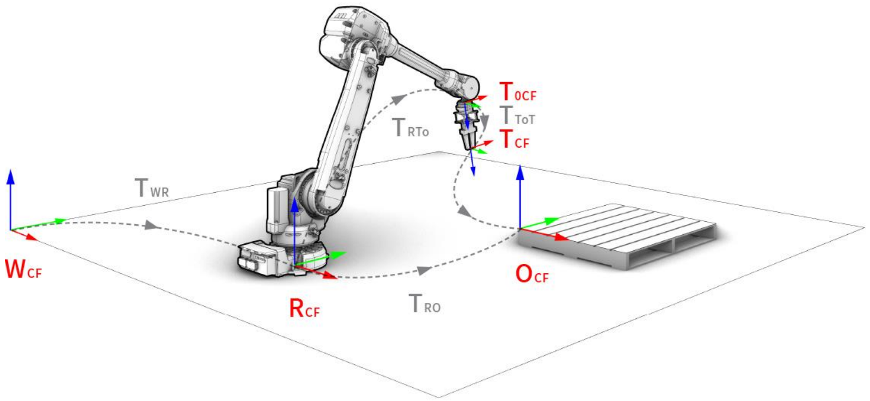


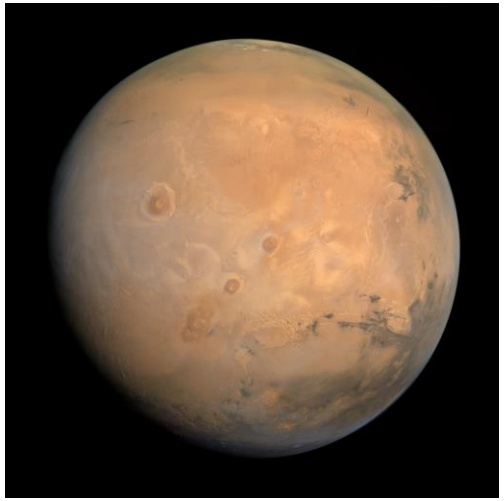
Global localization

- Just use a GPS

Global localization

- Just use a GPS?
  - What about orientation then?
  - Accuracy?
  - What if this is our robot setup?
  - Or if this is our environment?



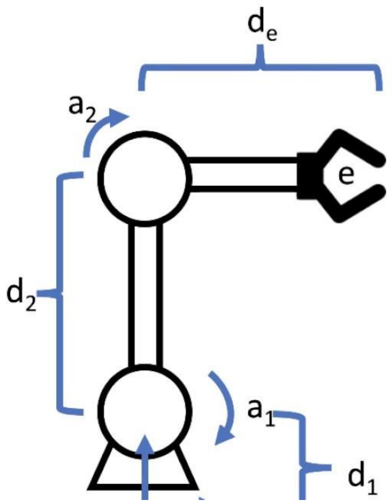


- What does global localization even mean?
  - We usually call it **ABSOLUTE** localization, because it’s with respect to some defined world reference frame

Forward kinematics of a robot arm

- Use sensor measurement to determine the location of the TPC
- Standard construction for robot manipulators: Denavit-Hartenberg
  - The location is given by a series of matrix multiplications

$$
{}^{i-1}T_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_{i,i+1} & \sin\theta_i\sin\alpha_{i,i+1} & a_{i,i+1}\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_{i,i+1} & -\cos\theta_i\sin\alpha_{i,i+1} & a_{i,i+1}\sin\theta_i \\ 0 & \sin\alpha_{i,i+1} & \cos\alpha_{i,i+1} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$



d_1	Distance between ground and Joint 1
a_1	Angle of rotation around Joint 1
d_2	Distance between Joint 1 and Joint 2
a_2	Angle of rotation around Joint 2
d_e	Distance between Joint 2 and the end effector

Forward kinematics of a robot arm

- Use sensor measurement to determine the location of the TCP
- Standard construction for robot manipulators: Denavit-Hartenberg
  - The location is given by a series of matrix multiplications

Forward kinematics of a robot arm

- Use sensor measurement to determine the location of the TCP
- Standard construction for robot manipulators: Denavit-Hartenberg
  - The location is given by a series of matrix multiplications

- What are the assumptions for this to work?
  - Static environment
  - Objects have the exact same location every time

How does this transfer to mobile robots?

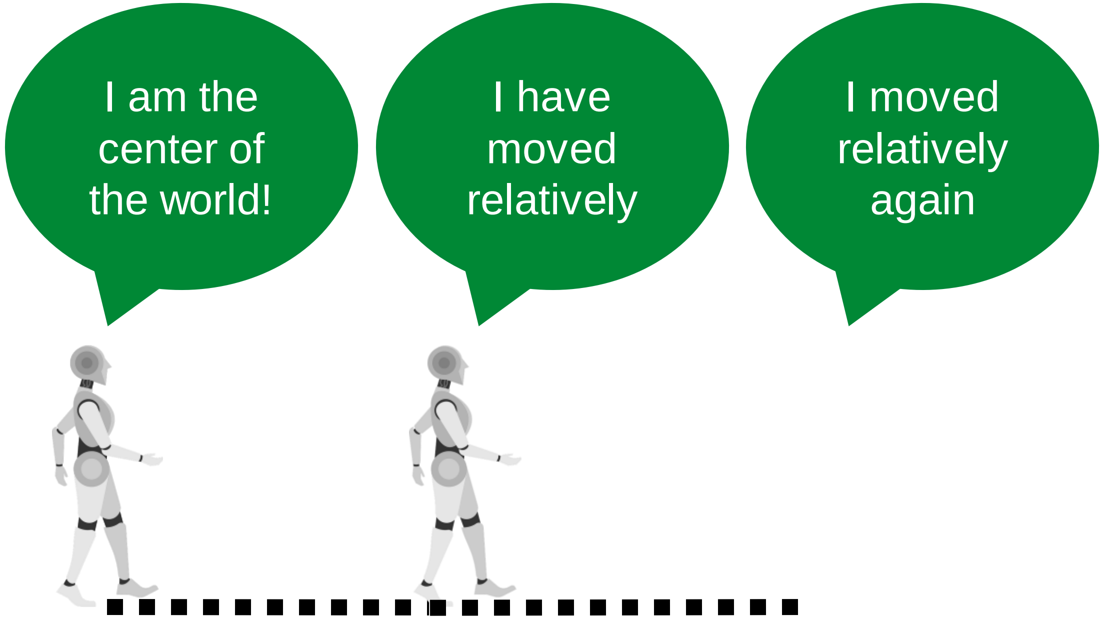
- Use sensor measurement to determine the location of ~~the TPC~~ mobile robots
- The location is given by a series of matrix multiplications

$$
{}^{i-1}T_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_{i,i+1} & \sin\theta_i \sin\alpha_{i,i+1} & a_{i,i+1}\cos\theta_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_{i,i+1} & -\cos\theta_i \sin\alpha_{i,i+1} & a_{i,i+1}\sin\theta_i \\ 0 & \sin\alpha_{i,i+1} & \cos\alpha_{i,i+1} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

- So, we just need to keep track of how much we have moved relatively

Relative localization for mobile robots

- Assume no global information about the environment (no map)
- Keep track of movement by integrating over the sensor inputs
  - This is called odometry
- What would the equivalent of measuring joint sensors be for a mobile robot?



Wheel Odometry

- Use wheel encoders to measure how many revolutions a wheel has done
- The size of the wheel determines the distance driven per revolution

- For a differential drive robot:
  – The current position of the robot $\boldsymbol{p} = [x, y, \theta]^T$

$$\Delta x = \Delta s \cos\left(\theta + \frac{\Delta\theta}{2}\right)$$

$$\Delta y = \Delta s \sin\left(\theta + \frac{\Delta\theta}{2}\right)$$

$$\Delta\theta = \frac{\Delta s_r + \Delta s_l}{b}$$

$$\Delta \mathrm{s} = \frac{\Delta \mathrm{s}_\mathrm{r} + \Delta s_l}{2}$$

Where b is the distance between the two wheels

Wheel Odometry

- For a differential drive robot:
  - The current position of the robot $\boldsymbol{p} = [x, y, \theta]^T$

$$\Delta x = \Delta s \cos\left(\theta + \frac{\Delta\theta}{2}\right)$$

$$\Delta y = \Delta s \sin\left(\theta + \frac{\Delta\theta}{2}\right)$$

$$\Delta\theta = \frac{\Delta s_r + \Delta s_l}{b}$$

$$\Delta \mathrm{s} = \frac{\Delta \mathrm{s}_{\mathrm{r}} + \Delta s_l}{2}$$

Where b is the distance between the two wheels

Wheel Odometry

- For a differential drive robot:
  - The current position of the robot $\boldsymbol{p} = [x, y, \theta]^T$

$$\Delta x = \Delta s \cos\left(\theta + \frac{\Delta\theta}{2}\right)$$

$$\Delta y = \Delta s \sin\left(\theta + \frac{\Delta\theta}{2}\right)$$

$$\Delta\theta = \frac{\Delta s_r + \Delta s_l}{b}$$

$$\Delta \mathrm{s} = \frac{\Delta \mathrm{s}_\mathrm{r} + \Delta s_l}{2}$$

Where b is the distance between the two wheels

$$\boldsymbol{p}' = \boldsymbol{p} + \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\theta \end{bmatrix}$$

By using the relationship between Δs and $\Delta\theta$, we can substitute this further

Wheel Odometry

- For a differential drive robot:
  - The current position of the robot $\boldsymbol{p} = [x, y, \theta]^T$

$$\Delta x = \Delta s \cos\left(\theta + \frac{\Delta\theta}{2}\right)$$

$$\Delta y = \Delta s \sin\left(\theta + \frac{\Delta\theta}{2}\right)$$

$$\Delta\theta = \frac{\Delta s_r - \Delta s_l}{b}$$

$$\Delta \mathrm{s} = \frac{\Delta s_{\mathrm{r}} + \Delta s_l}{2}$$

Where b is the distance between the two wheels

By using the relationship between Δs and $\Delta\theta$, we can substitute this further:

$$\boldsymbol{p}' = \boldsymbol{p} + \begin{bmatrix} \frac{\Delta s_{\mathrm{r}} + \Delta s_l}{2} \cos\left(\theta + \frac{\Delta s_r - \Delta s_l}{2b}\right) \\ \frac{\Delta s_{\mathrm{r}} + \Delta s_l}{2} \sin\left(\theta + \frac{\Delta s_r - \Delta s_l}{2b}\right) \\ \frac{\Delta s_r - \Delta s_l}{b} \end{bmatrix}$$

Wheel Odometry – errors and noise

- Wheel odometry is extremely sensitive to noise
  - Slipping wheels
  - Uneven terrain

  - Misalignment of wheels
  - Shifting contact point of the wheel
  - Error propagation

- In practice, we don't use this method a lot when other methods are available
  - If used, it's often combined with other approaches

Visual Odometry

- Utilize visual features to track changes in localization
- Camera independent – Can be used with a large range of sensors
  - Though, depending on the type of camera, we might have to do some extra steps
- Requires a sequence of images from a rigidly mounted camera
  - Match features in two subsequent images to generate the transformation

- We are specifically looking for the transformation between two subsequent frames:
  - $T_k = \begin{bmatrix} R_{k-1,k} & t_{k-1,k} \\ 0 & 1 \end{bmatrix}$

- The relative motion between a set of subsequent transformations is then
  - $T_{0:N} = \{T_0, \dots, T_N\}$

- We can get then get the current pose through simple matrix multiplications
  - $C_N = C_{N-1} T_N$

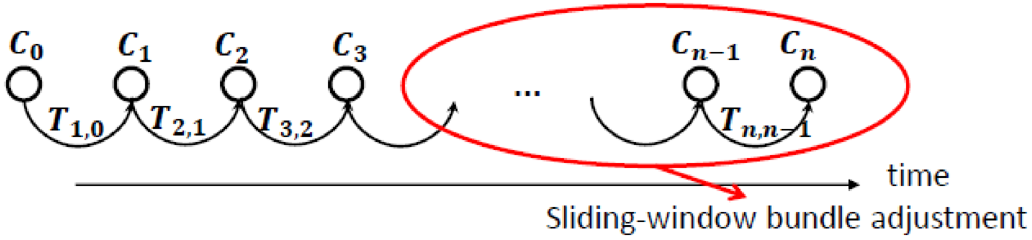
Visual Odometry

- We are specifically looking for the transformation between two subsequent frames:
  - $T_k = \begin{bmatrix} R_{k-1,k} & t_{k-1,k} \\ 0 & 1 \end{bmatrix}$

- The relative motion between a set of subsequent transformations is then
  - $T_{0:N} = \{T_0, \ldots, T_N\}$

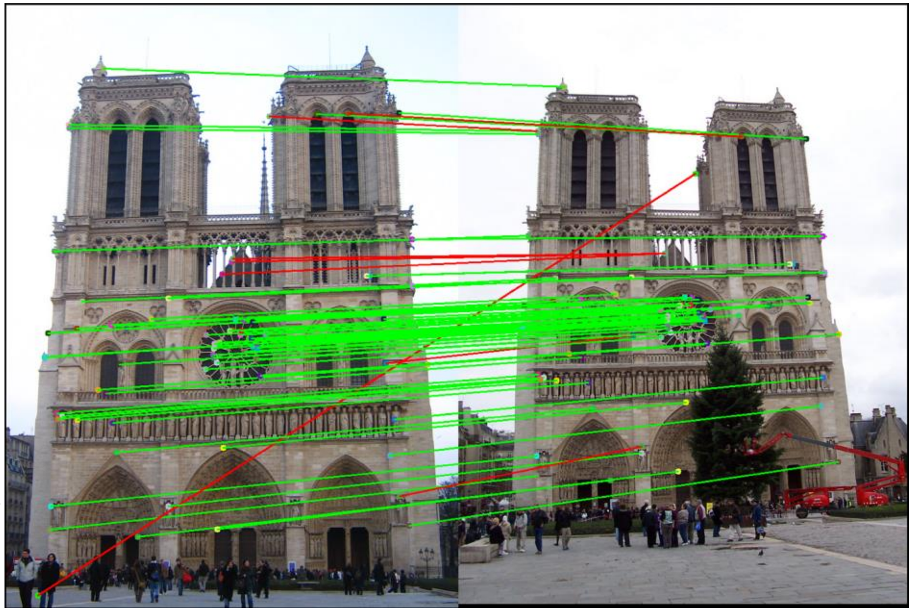
- We can get then get the current pose simply through simple matrix multiplications
  - $C_N = C_{N-1} T_N$

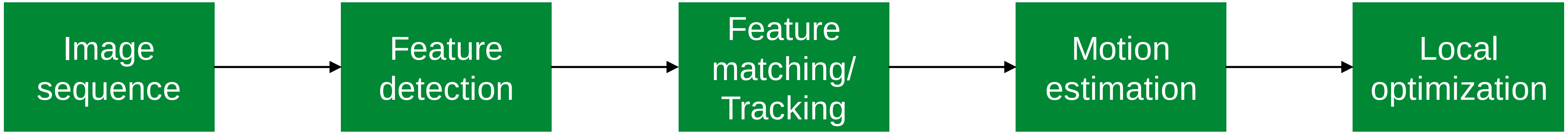
- Producing this relative transformation is the core of VO
  - Which means we recover the path incrementally
  - (though we might perform bundle adjustment as a refinement step)



So how do we produce this relative transform?

- Feature correspondence
  - Use a feature detector on both images and generate correspondence
  - The features must be robust for best performance!
- Appearance based approach
  - Match pixel intensities instead of features
  - Computationally heavier and generally performs worse
  - Not used a lot



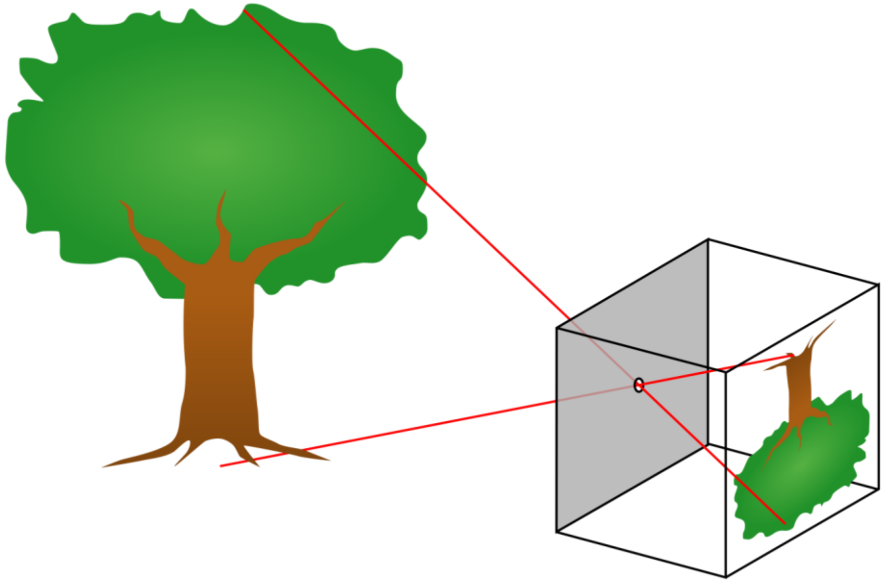


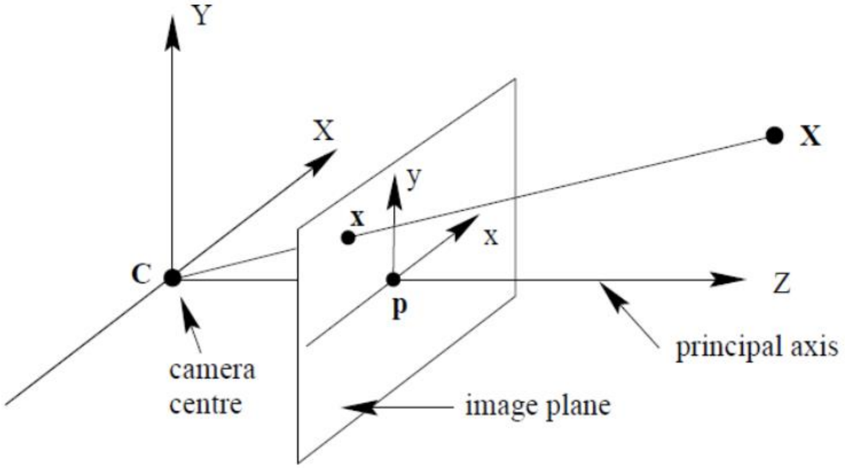
So how do we produce this relative transform?

- There are three options for feature-based approaches:
  - 2D to 2D
    - All features are produced in 2D image coordinates
  - 3D to 2D
    - Features of one image plane are given in 3D coordinates and projected to the second image plane
  - 3D to 3D
    - All features are given in 3D – this requires triangulation, e.g., with a stereo camera

The trigonometry

- Assuming a pinhole-model
  - i.e. the mapping from world coordinates to image coordinates is a linear projection
  - $[X, Y, Z] \rightarrow \left[\frac{fX}{Z}, \frac{fY}{Z}\right]$





2D to 2D

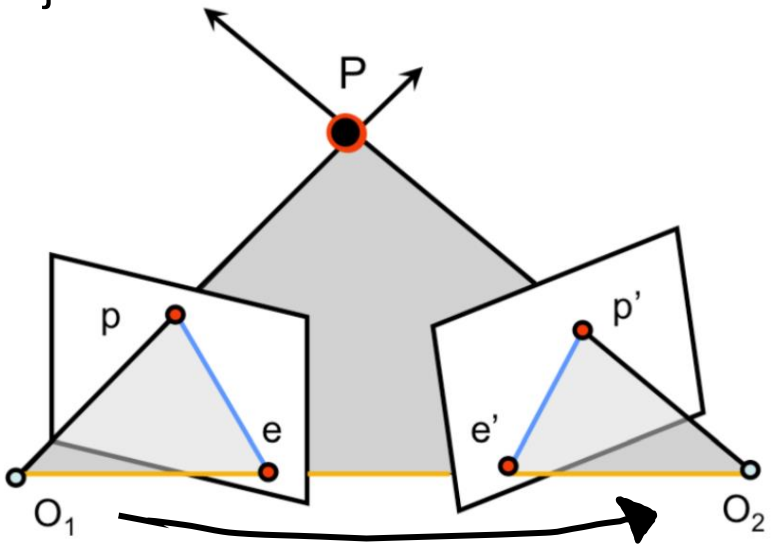
- Find the transformation T_k that minimizes the reprojection error:

$$T_k = \underset{X_i, C_k}{\operatorname{argmin}} \left|\left| p_k - g(X_i, C_k) \right|\right|^2$$

- Where p_k are our features from image k and $g(X_i, C_k)$ is the reprojection of our features into image plane $k-1$

- Use the essential matrix to get the reprojection:
  - A minimum of 5 points needed – more = more better
  - p' in image plane O_1 is $Rp' + t$
  - Since $Rp' + T$ and T are on the same (epipolar) plane, the crossproduct $\mathrm{t} \times (Rp' + t)$ is normal to the plane
  - p also lies on the epipolar plane! Its dot product should be zero:

$$p \cdot \left(t \times (Rp' + t)\right) = 0$$



$$T_k = \begin{bmatrix} R_{k-1,k} & t_{k-1,k} \\ 0 & 1 \end{bmatrix}$$

2D to 2D

- Use the essential matrix to get the reprojection:
  - p' in image plane O_1 is $Rp' + t$
  - Since $Rp' + t$ and t are on the same (epipolar) plane, the crossproduct $\mathrm{t} \times (Rp' + t)$ is normal to the plane
  - p also lies on the epipolar plane! Its dot product should be zero:

$$p \cdot \big(t \times (Rp' + t)\big) = 0$$
$$p \cdot \big(t \times (Rp')\big) = 0$$

  - We can express cross product using matrix multiplication:

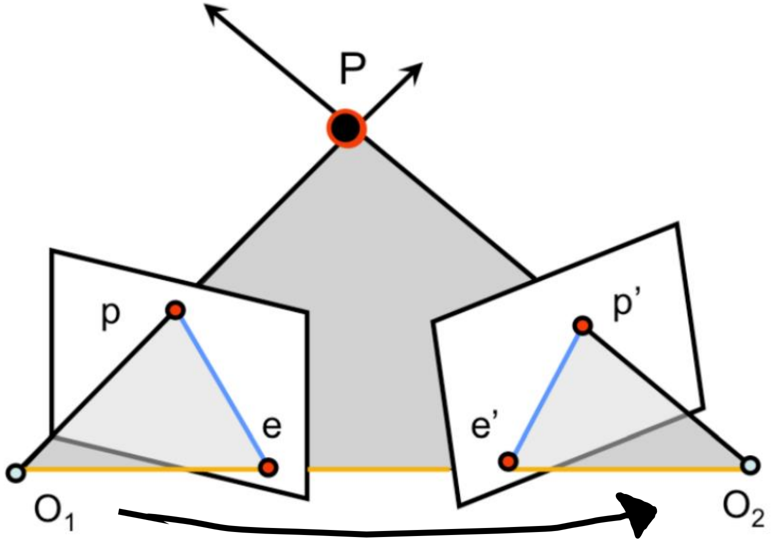
$$a \times b = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ a_y & a_x & 0 \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = [a_\times]b$$

  - We get

$$p^T \cdot [t_\times](Rp') = 0$$
$$p^T [t_\times] R p' = 0$$

  - The matrix $E = [t_\times]R$ is the essential matrix!

$$p^T E p' = 0$$



$$T_k = \begin{bmatrix} R_{k-1,k} & t_{k-1,k} \\ 0 & 1 \end{bmatrix}$$

2D to 2D

- So the matrix $E = [t_\times]R$ is the essential matrix, and it can be decomposed to a translation and rotation
- Why isn't $E = T_k$ then?

- Scale ambiguity! – we are only looking at epipolar LINES
  - The Essential matrix is a 3×3 matrix that contains 5 degrees of freedom. It has rank 2 and is singular
  - One solution is to use triangulation with previous frames and compute the scale using

$$r = \frac{\left|\left|X_{k-1}^1 - X_{k-1}^2\right|\right|}{\left|\left|X_k^1 - X_k^2\right|\right|}$$

Where X are the triangulated features in 3D

  - Do this for many points to increase robustness

2D to 2D

and it can be decomposed to a translation

If you found this confusing, don’t worry there’s a 10ECTS course on the topic

ar LIN

Perception for Autonomous Systems

tains 5 degrees of freedom. It has rank 2

us frames and compute the scale using

Where X are features in an image plane

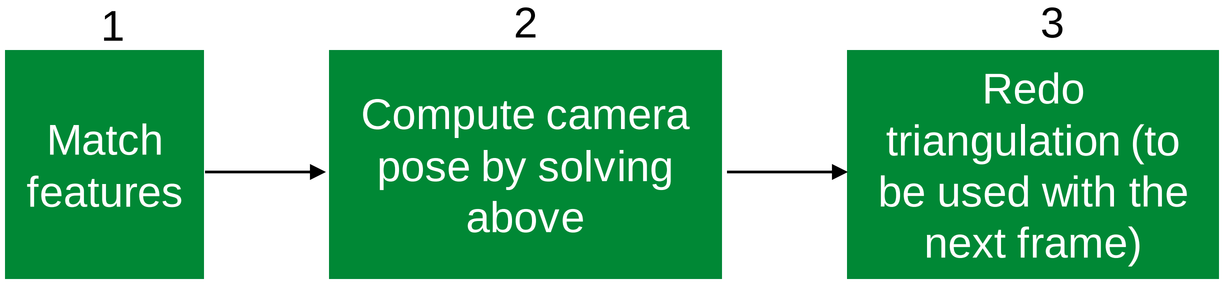
– Do this for many points to increase robustness

3D to 2D

- This problem is known as camera resection or PnP (perspective from n points)
- Determine the transformation that minimizes the reprojection error

$$T_k = \begin{bmatrix} R_{k-1,k} & t_{k-1,k} \\ 0 & 1 \end{bmatrix} = \underset{T_k}{\operatorname{argmin}} \left|\left|p_k - p'_{k_1}\right|\right|^2$$

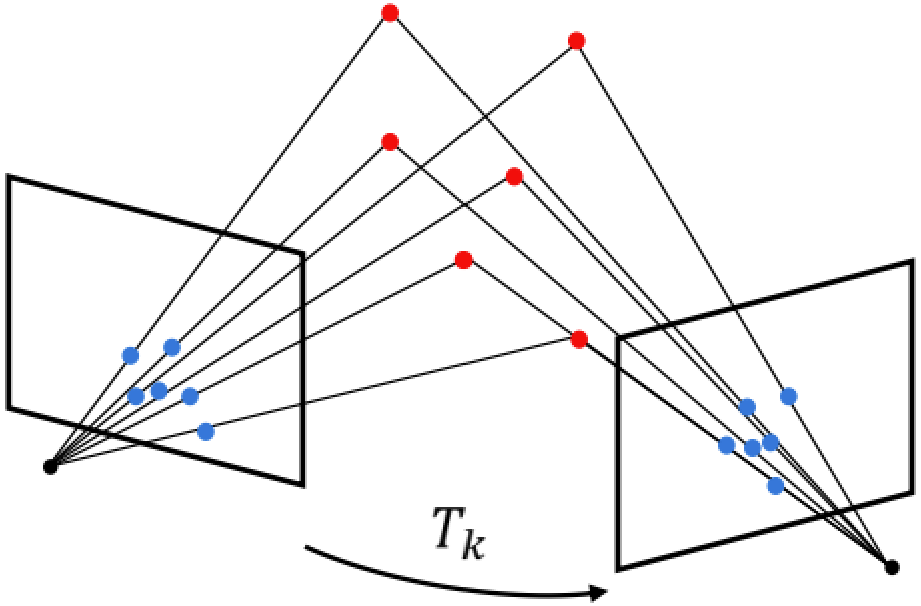
- Basically boils down to



1 Match features

2 Compute camera pose by solving above

3 Redo triangulation (to be used with the next frame)



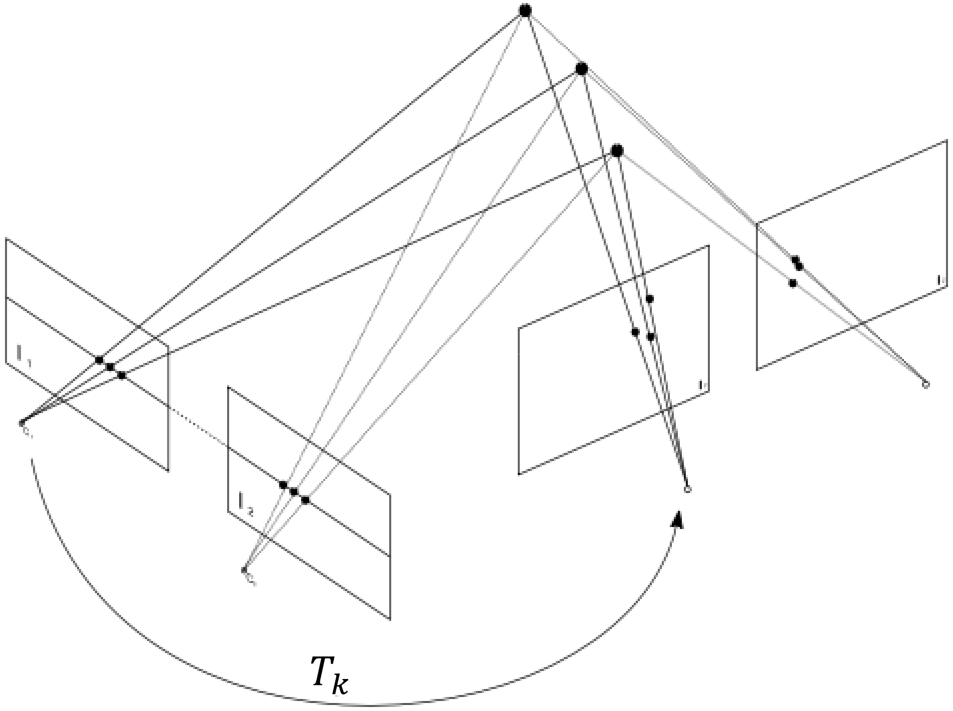
3D to 3D

- To do this, we need two stereo cameras
- Match the two pointclouds by minimizing the error

$$T_k = \begin{bmatrix} R_{k-1,k} & t_{k-1,k} \\ 0 & 1 \end{bmatrix} = \underset{T_k}{\operatorname{argmin}} \left|\left|X_k^1 - T_k X_k^2\right|\right|^2$$

Where X are the triangulated features in 3D

- This can be done in many ways
  - RANSAC, ICP, Robust Point matching, kernel correlation, and many more



Visual Inertial Odometry

- Visual odometry is great, but…
  - .. we are bound to low-framerates and highly depending on visual features
    - i.e. we can’t move fast because of motion blur
  - .. we cannot estimate velocities

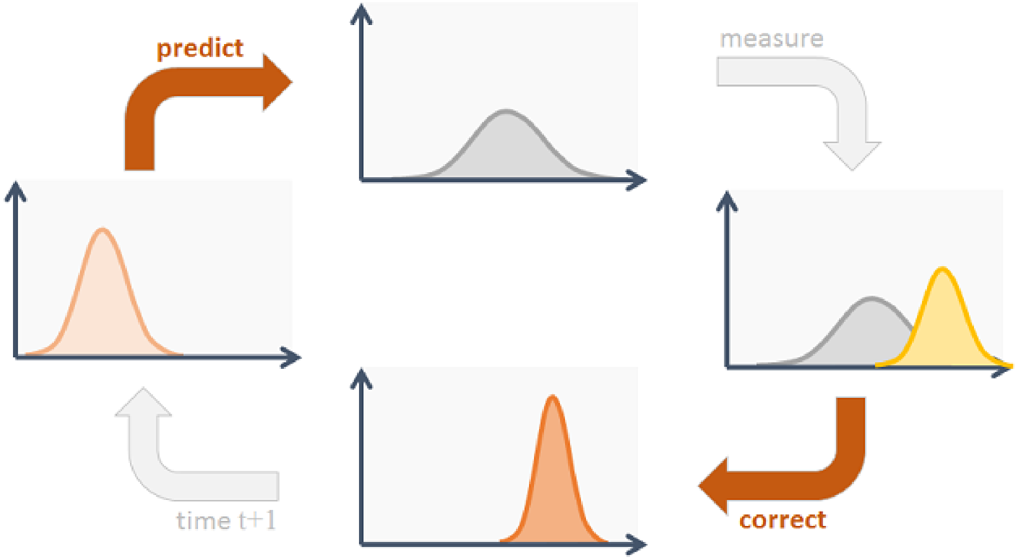
- The solution is to fuse our VO with other sensors
  - Inertial Measurement Units provides
    - 6DOF (3DOF acceleration and 3DOF gyroscope) + sometimes a magnetometer
    - Fast sampling rate (can be almost a magnitude faster than the camera FPS)
  - However, they suffer from
    - Sensitive to vibrations
    - Drifts over time – not suitable for localization on its own

- Do you see the combined advantage here?

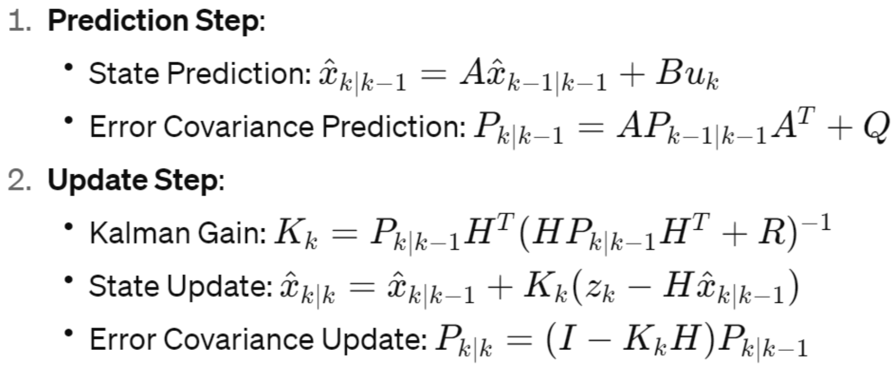
Visual Inertial Odometry – sensor fusion

- How do we fuse the visual odometry and the IMU?

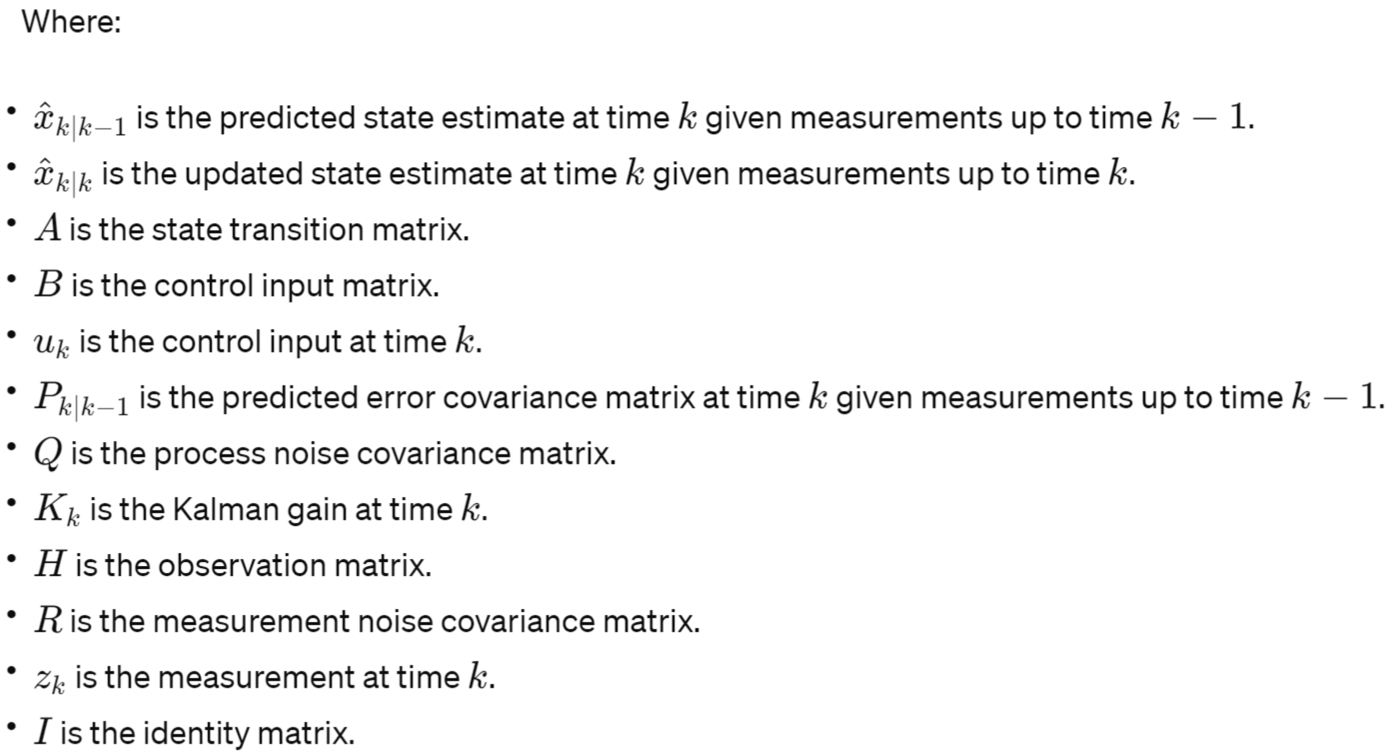
- We use a Kalman filter
  - – Predict the motion from sensor data
  - – Update/correct the prediction based on new sensor data



Visual Inertial Odometry – Kalman filter



1. **Prediction Step**:
   - State Prediction: $\hat{x}_{k|k-1} = A\hat{x}_{k-1|k-1} + Bu_k$
   - Error Covariance Prediction: $P_{k|k-1} = AP_{k-1|k-1}A^T + Q$
2. **Update Step**:
   - Kalman Gain: $K_k = P_{k|k-1}H^T(HP_{k|k-1}H^T + R)^{-1}$
   - State Update: $\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(z_k - H\hat{x}_{k|k-1})$
   - Error Covariance Update: $P_{k|k} = (I - K_kH)P_{k|k-1}$



Where:

- $\hat{x}_{k|k-1}$ is the predicted state estimate at time k given measurements up to time $k-1$.
- $\hat{x}_{k|k}$ is the updated state estimate at time k given measurements up to time k.
- A is the state transition matrix.
- B is the control input matrix.
- u_k is the control input at time k.
- $P_{k|k-1}$ is the predicted error covariance matrix at time k given measurements up to time $k-1$.
- Q is the process noise covariance matrix.
- K_k is the Kalman gain at time k.
- H is the observation matrix.
- R is the measurement noise covariance matrix.
- z_k is the measurement at time k.
- I is the identity matrix.

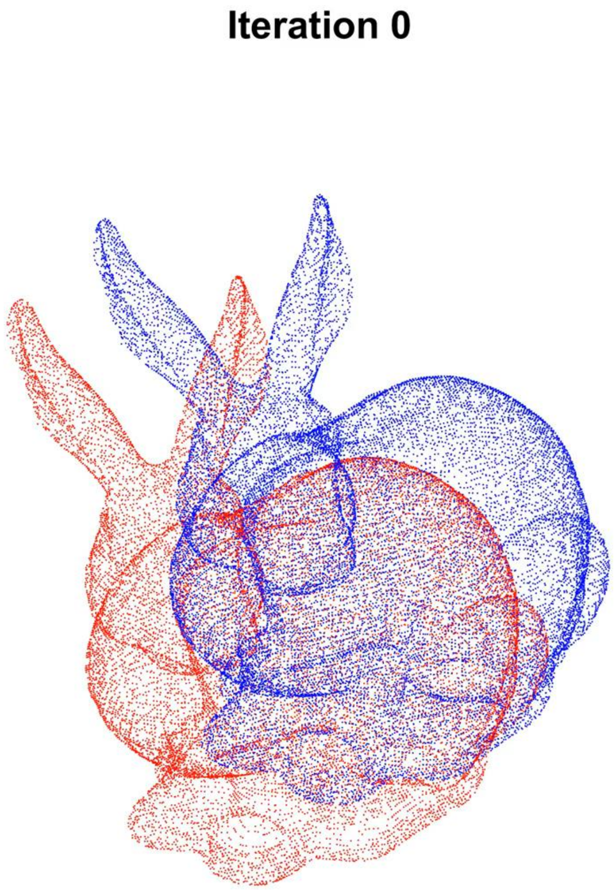
Iterative closest point

- Matching two pointclouds (like we did for the 3D-3D VO case)

- So we are trying to find T_k

$$T_k = \begin{bmatrix} R_{k-1,k} & t_{k-1,k} \\ 0 & 1 \end{bmatrix} = \underset{T_k}{\operatorname{argmin}} \left|\left|X_k - T_k X_k\right|\right|^2$$

- This can be solved iteratively for two pointclouds P^1, P^2
  1. For each point p_i^2, find the nearest point p_j^1
  2. Use all point correspondences to compute T_k^n
  3. Apply T_k^n to P^1
  4. Repeat from step 1 until convergence $(n \rightarrow n+1)$



Computing the transformation for ICP

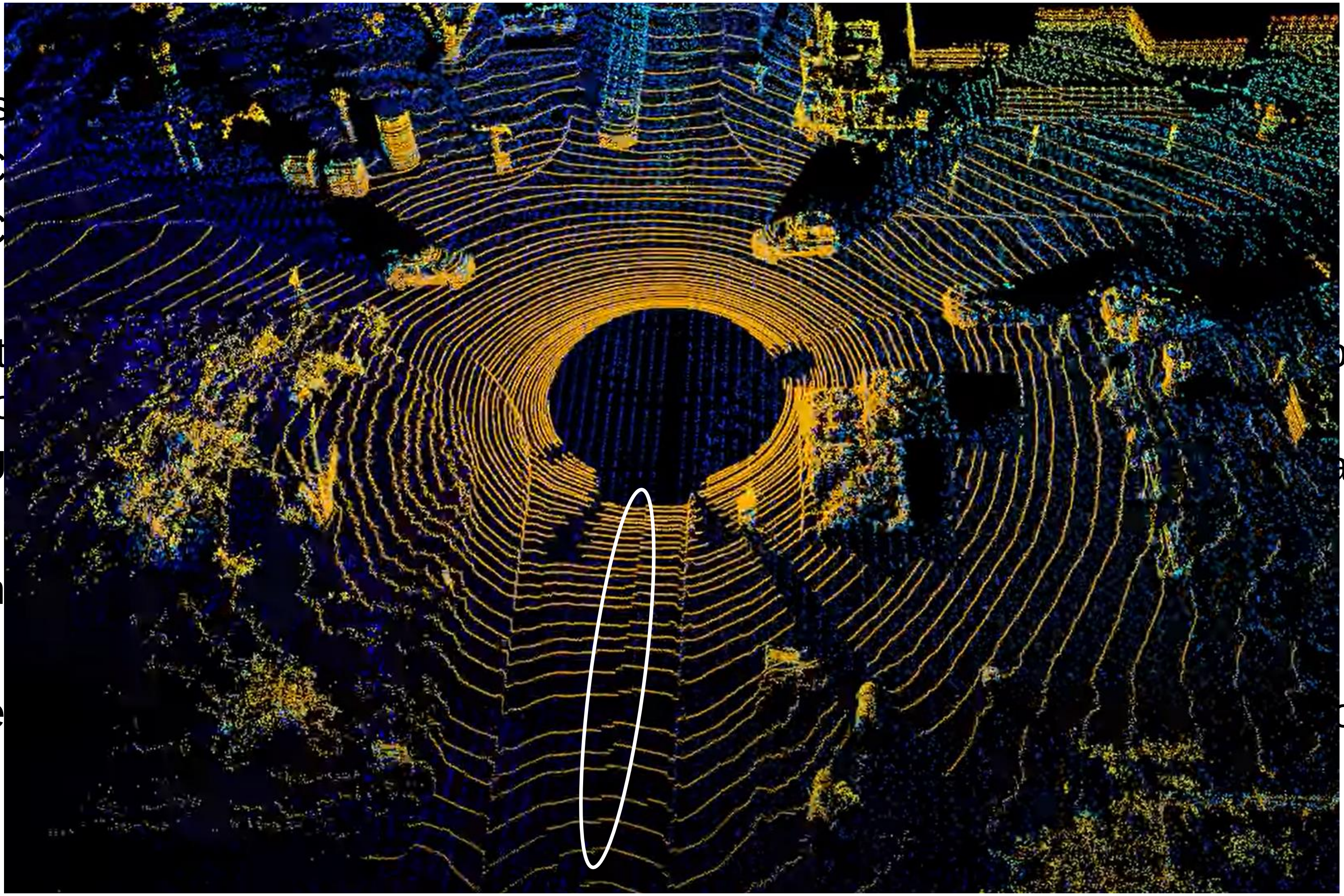
- So how do we actually get the transformation given only the pairs $\left(p_i^1, p_j^2\right)$ - Ideas?

- Center the two pointclouds by mean subtraction μ_1, μ_2
- Compute the covariance matrix
  - $C = cov\left(P'^1, P'^2\right)$
- Compute the rotation
  - $R = UV^T$ where U and V are SVD components of the covariance matrix
- Compute the translation
  - $t = \mu_1 - R\mu_2$

LIDAR Odometry – KISS ICP

- Keep it small and simple ICP (KISS ICP)
  - Assume a motion/velocity model
    - Can be constant or estimated from previous movement
    - Can also be based on wheel encoders or IMU integration
  - Instead of matching previous LIDAR scan cloud to the current, we try to correct the error from our motion model
    - Use our motion model to predict how much the robot has moved since last scan
  - What are the advantages of doing it this way?
  - We can correct for movements during data acquisition (relevant for fast moving robots like cars)

LIDAR Odometry – KISS ICP



- Keep
  - Ass
    - C
    - C
  - Inst orrect the erro
    - U ast scan
  - Wh
  - We oving robots like

LIDAR Odometry

- Performing the KISS ICP correction
  - Move the pointcloud $\hat{P}^*$ according to your motion model $T_{pred,t}$

$$S = \left\{s_i = T_{t-1}T_{pred,t}\,\boldsymbol{p} \middle| \boldsymbol{p} \in \hat{P}^*\right\}$$

  - Match the updated scan S with the new pointcloud q

$$\Delta T_{est,j} = \underset{T}{\operatorname{argmin}} \left|\left|T s_i - q\right|\right|^2$$

$$\left\{s_i \leftarrow \Delta T_{est,j} s_i \middle| s_i \in S\right\}$$

  - The result is the error of your motion model

$$\Delta T_t = \left(T_{t-1}T_{pred,t}\right)^{-1} \Delta T_{icp,t} T_{t-1} T_{pred,t}$$

  - Note how we applied $T_{pred,t}$ in the local reference frame, we apply $\Delta T_{icp,t}$ in the global reference frame

Exercises

- Create a localization ROS node for your turtlebot
  - Use the LIDAR scanner
  - Assume a constant velocity model
  - Publish a TF with the result of your odometry
  - Compare with the one provided by ROS