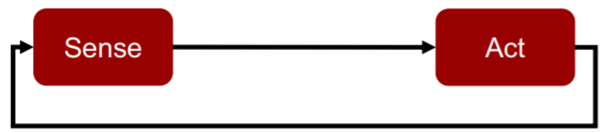
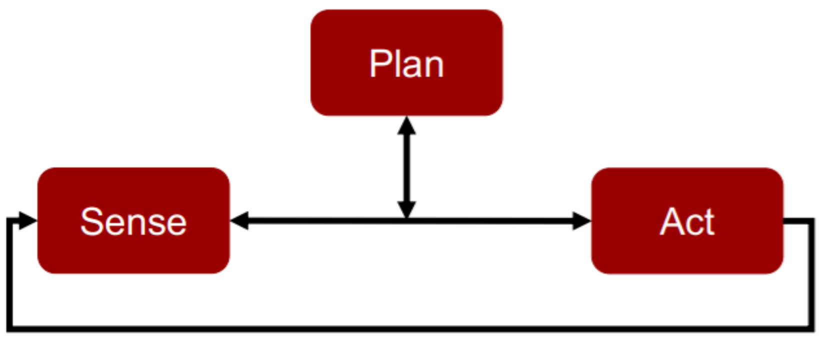
Lecture 1

- Hierarchical (deliberative) paradigm

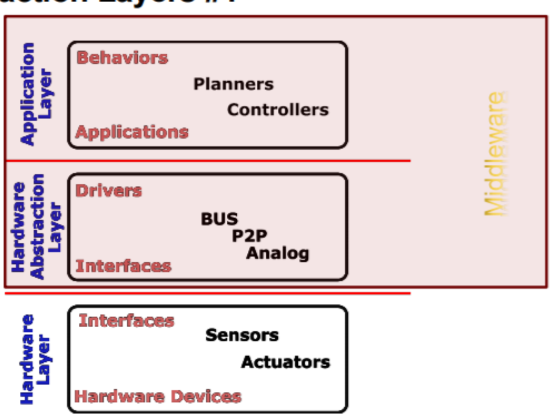
- Reactive paradigm



- Hybrid paradigm / Three Layer Architecture



Hardware Abstraction Layers #1



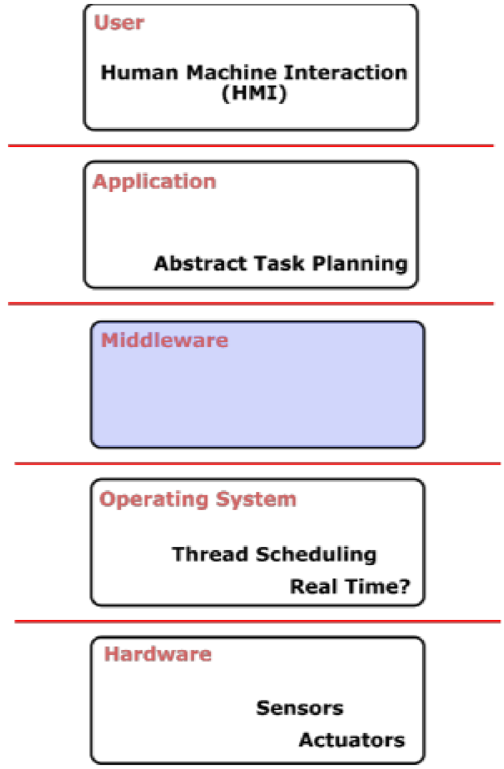
Towards Software Architecture

Must have characteristics:

- Support for multiple components
- Communication between components
- Easy way to write own components
- Possibility to replace individual components
- Easy to extend
- Means for data logging and debugging
- Support for decentralized components

Hardware Abstraction Layers #2

- Let's add a little detail..
- A class of technologies in order to handle the complexity of distributed systems



Lecture 2

What are some Middleware?

- Orocos
- Pyro
- Player
- Orca
- Miro
- OpenRTMaist
- ASEBA
- MARIE
- RSCA

- MRDS
- OPROS
- CLARAty
- ROS
- SmartSoft
- ERSP
- Webots
- RoboFrame

- Is there any reason there're so many?

- Different Scope
- Different Functional Architecture
- Different Communication Architectures

Let's see the most important ones!

- CLARAty
  - NASA Jet Propulsion Laboratory
  - Functional Layer:
    - Navigation, Mapping
    - Terrain evaluation, path planning
    - Estimation, simulation
  - Decision Layer:
    - General Planners
      Schedulers, Databases
  - Client ←→Server Scheme

- Orocos
  - RealTime
  - Orocos Components Library (OCL)
  - Orocos Kinematics and Dynamics Library (KDL)
  - Orocos Bayesian Filtering Library (BFL)
  - OMG's CORBA

- Orocos
- Player

Kind of Opposite

- Player
  - Shared Libraries among devices
  - Player Core
    - Drivers, Libraries
    - Configuration Parsing
  - Transport Layer
    - Independent of Drivers
    - TCP communication using web sockets

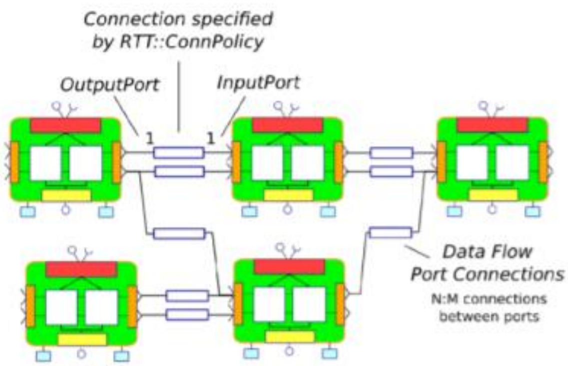
- Player
- ROS

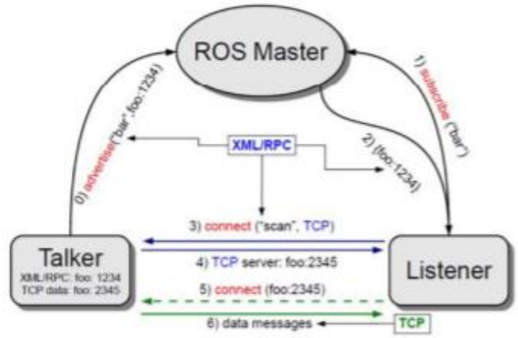
Logical Extension

- ROS
  - Master
  - Nodes
  - Topics
    - Messages
  - Publish/Subscribe Scheme

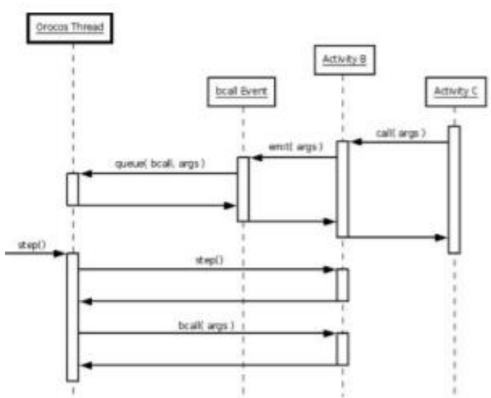
Main Differences to remember

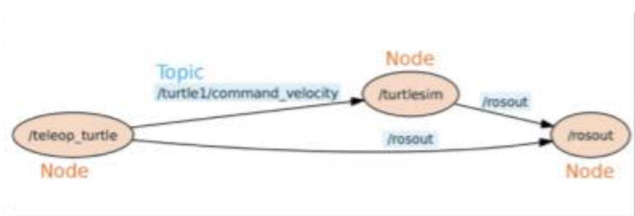
- Peer to Peer **Vs** "Open" Publish Subscribe Communication





- State Driven **Vs** Message Driven





Intro to ROS!

- What is ROS:
  - Distributed Computation
  - Software Reuse
  - Rapid Testing
- What it isn't:
  - A programming Language
  - A library
  - An IDE

- Master
  - "roscore"
- Let's run something, quickly!
- "rosrun"
- Packages
  - "rospack"
- What did just happen?
- "rqt_graph"

- Nodes
  - "rosnode"
- Topics & Messages
  - "rostopic"
  - "rosmsg"

Lecture 3

Transformations
High-level approach

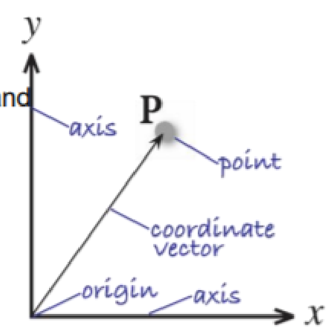
- Attach a separate coordinate system, or frame, for each rigid body
- Relate these frames to each other

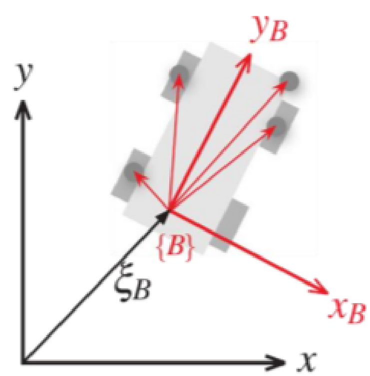
- Why? It makes everything so much easier!

- If we have all the coordinate systems, and their relations, we can easily **transform** a **pose** in one **frame** to **any other frame**

Transformations: Graphical overview

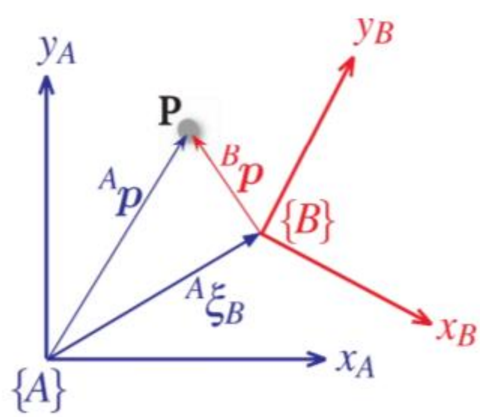
- **Point** is defined by a vector
  - The **frame** matters!
- **Frame** is defined by a changed in **pose ξ**
  - Describes both translation and rotation





Transformations: Graphical overview

- We use indices to indicate the relevant frames
  - For points, in which frame we have defined the point
  - For transformations, the **pose** of a frame with respect to another
    - Transform: "From A to B"
    - Pose: "B relative to A"



Transformations: Rotation around one axis

- Rotation around X axis

$$R^x_\phi = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & \sin\phi & 0 \\ 0 & -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation around X axis

$$R^y_\theta = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation around X axis

$$R^z_\psi = \begin{bmatrix} \cos & \sin\psi & 0 & 0 \\ -\sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Translation X,Y,Z axis

$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformations: Euler

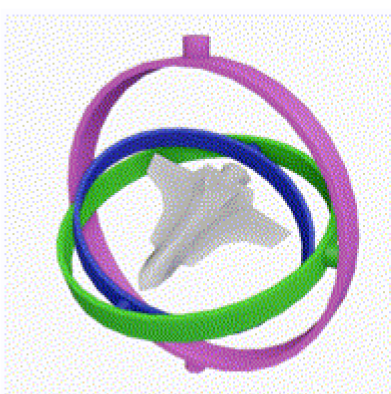
- 12 possible representations

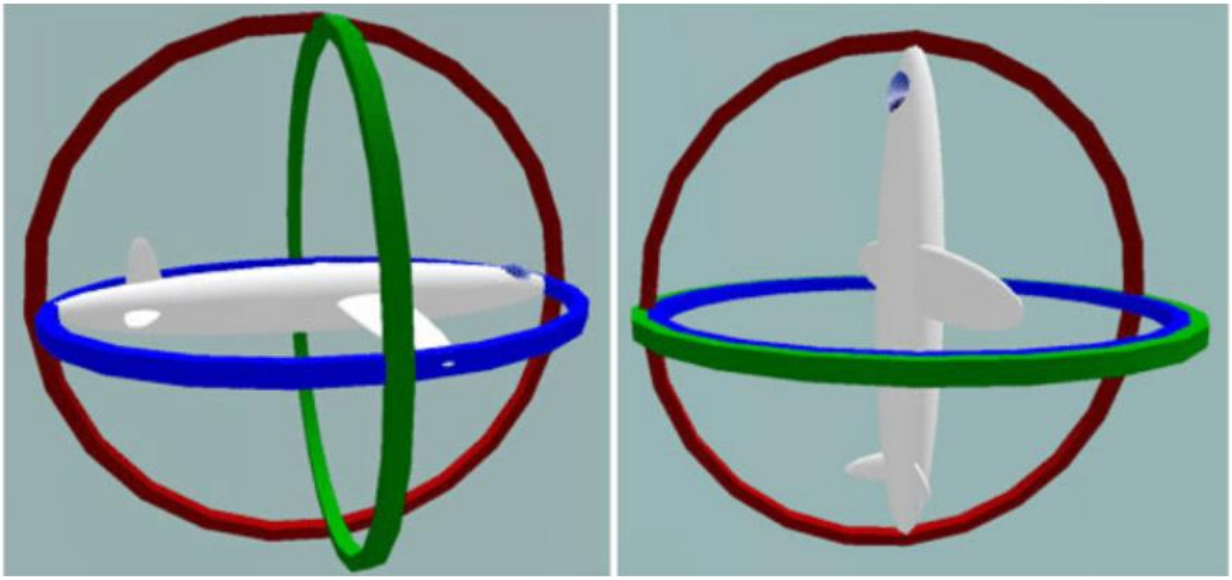
xyz	yzx	zxy
xzy	yxz	zyx
xyx	yzy	zxz
xzx	yxy	zyz

- The most popular is the roll, pitch, yaw one:

$$\begin{aligned} R &= R^x_\phi R^y_\theta R^z_\psi \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Gimbal lock is the loss of one degree of freedom in a three-dimensional, three-gimbal mechanism that occurs when the axes of two of the three gimbals are driven into a parallel configuration, "locking" the system into rotation in a degenerate two-dimensional space.





17.: On the left a normal situation: the three gimbals are independent. On the right the Gimbal lock phenomenon: two out of the three gimbals are on the same plane.

Transformations: Quaternions

- Provide Orientations
- Invented by Hamilton in 1843
- The governing rule is:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{i}\,\mathbf{j}\,\mathbf{k} = -1$$

- A quaternion is defined as:

$$q = q_0 + \mathbf{q} \quad = \quad q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

,where $q0$ is the scalar and q is called the vector part.
i,j,z is the common orthonormal bases of R^3

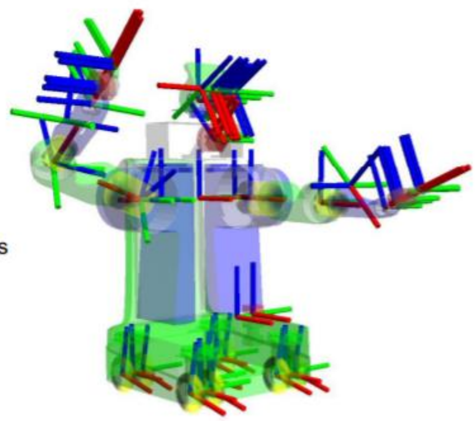
- Quaternions solve all the problems with euler angles

Transformations (Rotations): Overall

Task/Property	Matrix	Euler Angles	Quaternion
Rotating points between coordinate spaces (object and internal)	Possible	Impossible (must convert to matrix)	Impossible (must convert to matrix)
Concatenation or incremental rotation	Possible but usually slower than quaternion form	Impossible	Possible, and usually faster than matrix form
Interpolation	Basically impossible	Possible, but aliasing causes Gimbal lock and other problems	Provides smooth interpolation
Human interpretation	Difficult	Easy	Difficult
Storing in memory	Nine numbers	Three numbers	Four numbers
Representation is unique for a given orientation	Yes	No - an infinite number of Euler angle triples alias to the same orientation	Exactly two distinct representations for any orientation
Possible to become invalid	Can be invalid	Any three numbers form a valid orientation	Can be invalid

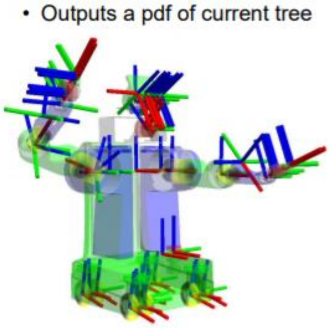
tf Package

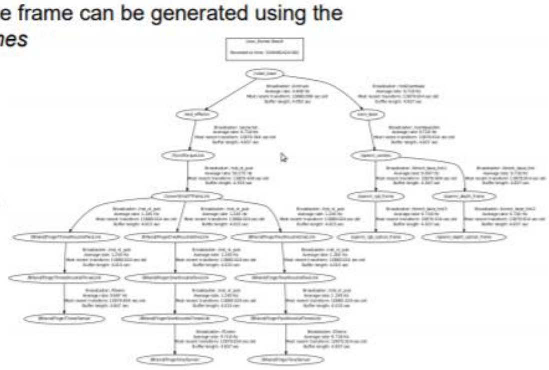
- The *tf* package allows the tracking over time of coordinate systems tree(s)
- Allows the easily creation of new frames (static or dynamic)
- Eases the process of transforming points, vectors, etc.
- Distributed system – no centralized storage
- Caches the past information on the transforms



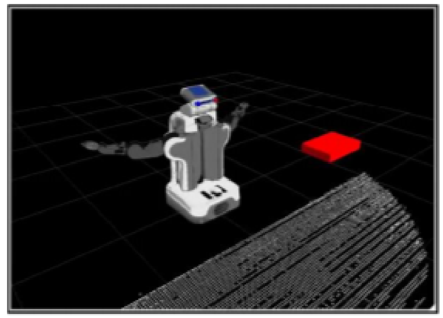
The *tf* coordinate frame tree

- A tree of the current coordinate frame can be generated using the command: *rosrun tf view_frames*
- Outputs a pdf of current tree





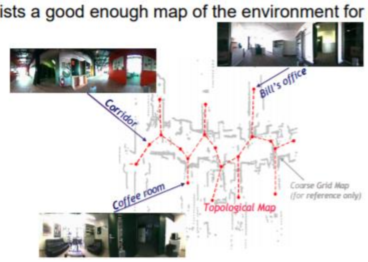
Robot Planning



Ok now we're more realistic…
… however, still autonomous robots are more complex than this
- The robots have to autonomously find their way through..
  - Robotic Manipulators need:
    - To solve the inverse kinematics and dynamics problems
    - Find the correct trajectories to avoid obstacles
    - …
  - Mobile Robots need to use path planning to navigate the environment

The Planning Problem (case of Mobile Robots 1/2)

- The problem: find a path in the work space (physical space) from the initial position to the goal position avoiding all collisions with the obstacles
- Assumption: there exists a good enough map of the environment for navigation.
  - Topological
  - Metric
  - Hybrid methods



The Planning Problem (case of Mobile Robots 2/2)

- We can generally distinguish between
  - (global) path planning and
  - (local) obstacle avoidance.
- First step:
  - Transformation of the map into a representation useful for planning
  - This step is planner-dependent
- Second step:
  - Plan a path on the transformed map
- Third step:
  - Send motion commands to controller
  - This step is planner-dependent (e.g. Model based feed forward, path following

Sampling-based Path Planning (or Randomized graph search)

- When the state space is large complete solutions are often infeasible.
- In practice, most algorithms are only resolution complete, i.e., only complete if the resolution is ne-grained enough
- Sampling-based planners create possible paths by randomly adding points to a tree until some solution is found

RRT

- RRT is a good example of a Sampling-based algorithm:

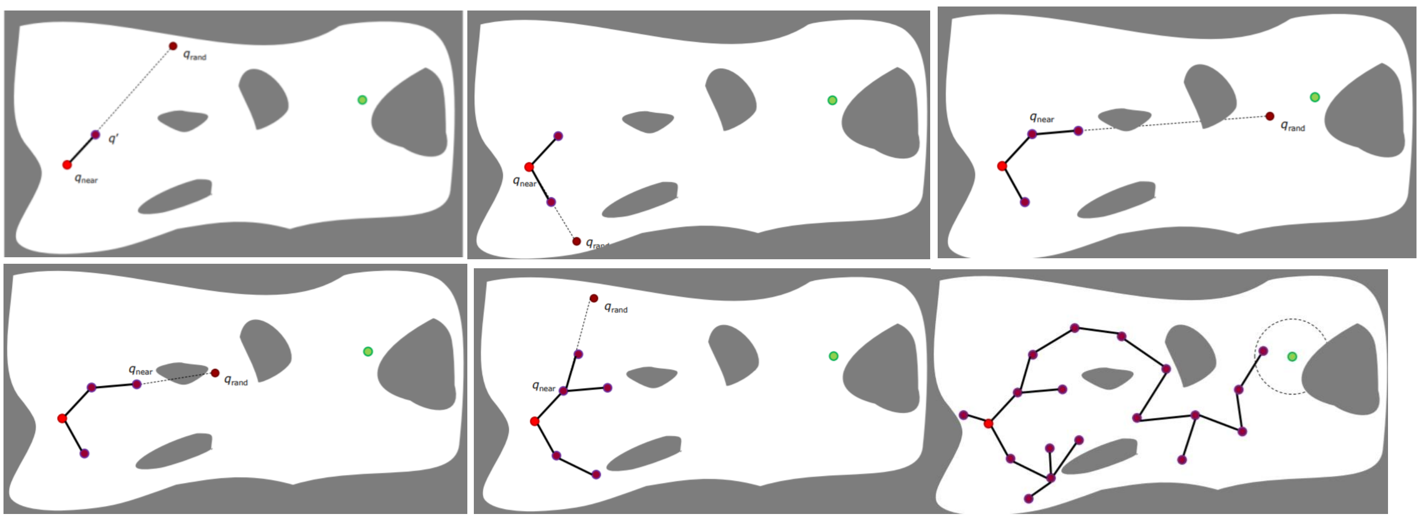
```
RRT(q_0)
  G.init(q_0);
  for i = 1 to k do
      q_n ← NEAREST(S, α(i));
      q_s ← STOPPING-CONFIGURATION(q_n, α(i));
      if q_s ≠ q_n then
          G.add_vertex(q_s);
          G.add_edge(q_n, q_s);
```

- Several additional algorithms are worth exploring
  - RRT*
  - Informed RRT
  - …

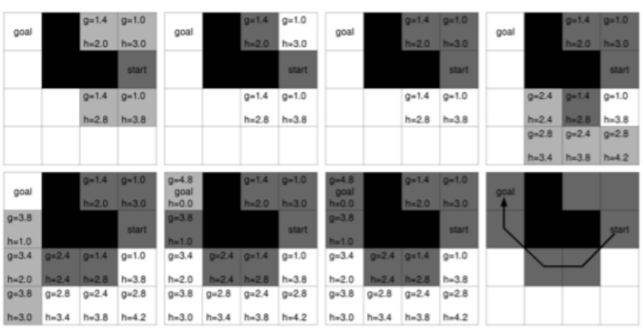
Forward Search Agorithms

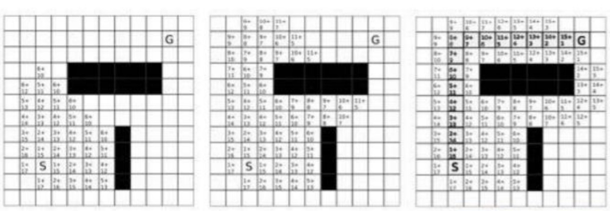
- Forward Search Methods:
  - Breadth first
  - Dijkstra's algorithm
  - A*

```
FORWARD_SEARCH
  Q.Insert(x_I) and mark x_I as visited
  while Q not empty do
      x ← Q.GetFirst()
      if x ∈ X_G
          return SUCCESS
      forall u ∈ U(x)
          x' ← f(x, u)
          if x' not visited
              Mark x' as visited
              Q.Insert(x')
          else
              Resolve duplicate x'
  return FAILURE
```



- Similar to Dijkstra's algorithm, except that it uses a heuristic function h(n)
- $f(n) = g(n) + \varepsilon\, h(n)$

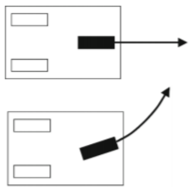


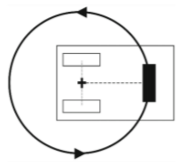


Lecture 4

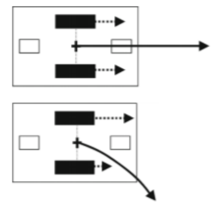
Types of Mobile Robot Locomotion - Steering

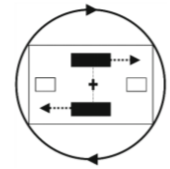
- Single Wheel
  - Working Principle
    - Single wheel for Driving and Steering
  - Pros
    - Linear and Angular Velocities Decoupled
  - Cons
    - Cannot handle Complex Terrains



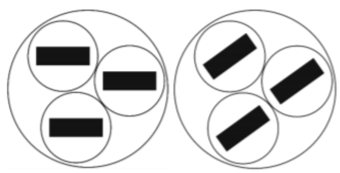


- Single Wheel
- Differential
  - Working Principle
    - Two fixed driving wheels
    - One caster wheel
  - Pros
    - Simplicity
  - Cons
    - Difficulty of calculating odometry/ Driving Straight

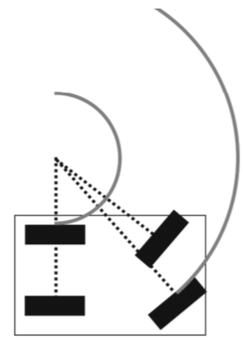




- Single Wheel
- Differential
- Synchro Drive
  - Working Principle
    - Three wheels rotating and driving identically
  - Pros
    - Almost holonomic
  - Cons
    - Has to stop to rotate

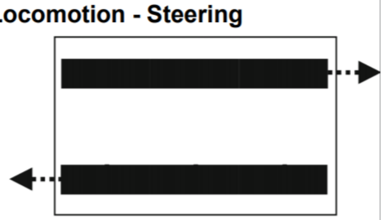


- Single Wheel
- Differential
- Synchro Drive
- Ackerman
  - Working Principle
    - All wheels on the tangent of circle
  - Pros
    - Different motor for drive and steering
  - Cons
    - Planning is difficult, non holonomic

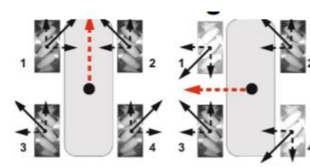


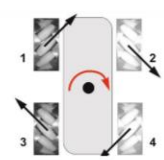
Types of Mobile Robot Locomotion - Steering

- Single Wheel
- Differential
- Synchro Drive
- Ackerman
- Skid Steering
  - Working Principle
    - Differential, no caster
  - Pros
    - Rugged, Robust
  - Cons
    - Wheel Odometry is veeery hard



- Single Wheel
- Differential
- Synchro Drive
- Ackerman
- Skid Steering
- Omni-Directional - Mecanum Wheels
  - Working Principle
    - Perpendicular vector
  - Pros
    - Holonomic
  - Cons
    - Hard to get grip(friction)





Differential Drive Kinematics

- Forward:

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = 2\pi r \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{d} & \frac{1}{d} \end{bmatrix} \begin{bmatrix} \dot{\theta}_L \\ \dot{\theta}_R \end{bmatrix}$$

where:

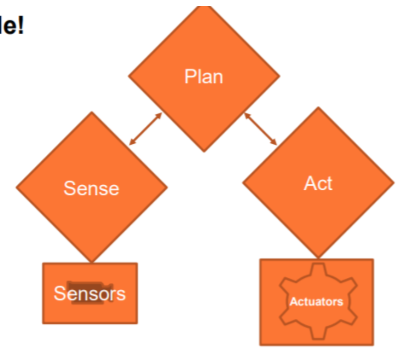
v is the vehicle's linear speed (equals ds/dt or $\dot{s}$),
ω is the vehicle's rotational speed (equals $d\varphi/dt$ or $\dot{\varphi}$),
$\dot{\theta}_{L,R}$ are the individual wheel speeds in revolutions per second,
r is the wheel radius,
d is the distance between the two wheels.

- Inverse

$$\begin{bmatrix} \dot{\theta}_L \\ \dot{\theta}_R \end{bmatrix} = \frac{1}{2\pi r} \begin{bmatrix} 1 & -\frac{d}{2} \\ 1 & \frac{d}{2} \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}$$

AGV System as a whole!

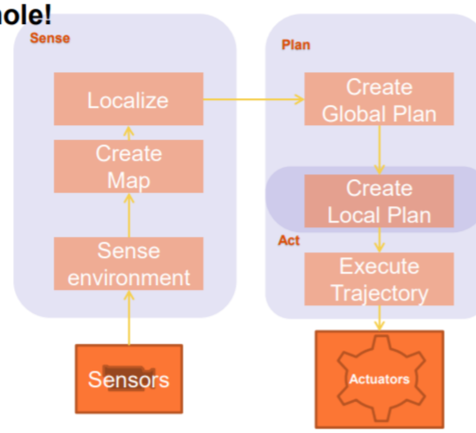
- Robot Software Architecture (Sense→Plan→Act)



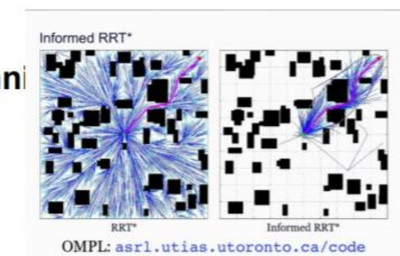
AGV System as a whole!

- Robot Software Architecture (Sense→Plan→Act)

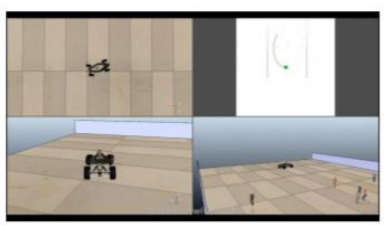
1. Use Sensors
   a) Mapping
   b) Localization
2. Plan Global Route
3. Execute Path
   a) Plan Local Trajectory
   b) Execute Velocity Commands



AGV Navigation - Path Planni

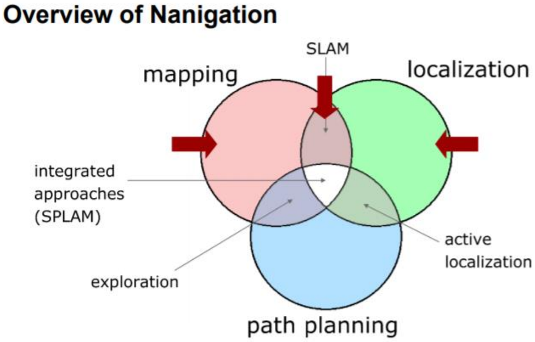


- Global Path Planning
  - Operates on Global Map
  - Find the best Global trajectory



- Local Path Planning
  - Operates on Local Map
  - Generate small navigation spline to overcome obstacles
  - Send arc velocity commands to robot

Overview of Nanigation



Overview of Navigation

To navigate a robot we need:
- A map
- A localization module
- A path planning module

These components are sufficient if:
- The map fully reflects the environment
- The environment is static
- There are no errors in the estimate

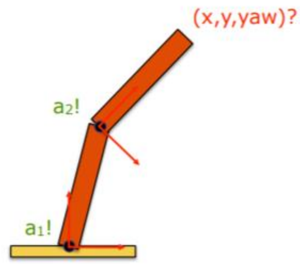
However:
- The environment changes (e.g. opening/closing doors)
- It is dynamic (things might appear/disappear from the perception range of the robot)
- The estimate is "noisy"

Thus we need to complement our ideal design with other components that address these issues, namely:
- Obstacle-Detection/Avoidance
- Local Map Refinement, based on the most recent sensor reading.

Lecture X

One-slide kinematics

- Kinematics is the "equations of motion"
- No regard to forces that cause the motion
- Our goal is to be able to use the robot in *Cartesian* coordinates
- Let's consider a simple robot
  - If we know the joint angles, where is the end-effector? How is the end-effector oriented?
    = **Forward Kinematics**
- Forward kinematics only have 1 solution
  Why?

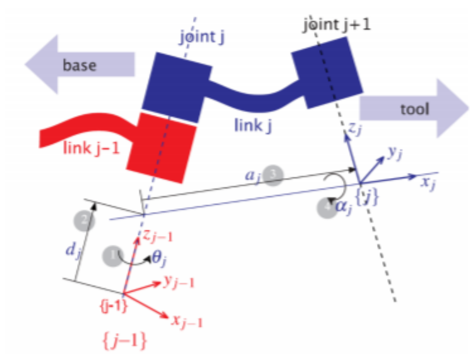


- A more complicated case is this:
  - Given a desired (x,y,yaw), what should the joint angles be?
    = **Inverse Kinematics**
- Inverse kinematics can have multiple solutions!
- How many solutions for this case?
- How about this one?
- Commonly 4 solutions for many robot arms

Describing robot arm kinematics

- Deals with describing the chain of transformations between the links in the robot arm
- Some convention is needed
- The standard is Denavit-Hartenberg parameters (from 1955)
- Describes robot kinematics using 4 parameters for each joint
- Remember there can be different types of joints, and arms
  - "RRRRRR" - articulated robot, with only rotational joints
  - "RRPRRR" - this robot has one prismatic joint
- In the toolbox, one extra parameter defines the type of each joint
  - 0 = rotational
  - 1 = prismatic

DH parameters



- Robot has *N* joints, and *N+1* links
- Joint *j* connects link *j-1* to link *j*
  - Joint *j* moves link *j*
- Link described by two parameters:
  - Length - a_j (sometimes called r_j)
  - Twist - α_j
- Joint described by two parameters:
  - Link offset - d_j
  - Joint angle - θ_j
- Joint 1 connects Link 0 (the base of the robot) to Link 1
- Joint N connects Link N-1 to Link N (the end-effector of the robot)

Forward kinematics

- The goal is to find the end-effector pose, as a function of the joint angles
  - How do we do this?
- Transformation from base to end-effector ${}^{B}T_{E}$
  - How do we find this?
- By joining the transformations for each link ${}^{j-1}A_j$ and multiplying

$$ {}^{0}T_E = {}^{0}A_1\,{}^{1}A_2 \cdots {}^{N-1}A_N $$

- The forward kinematics solution exists and is unique for *any* serial-link robot
- A serial-link robot is a robot where the links are connected in series

- In the toolbox, this is defined as a SerialLink object
- First, define a vector L with the Links
- Then construct the SerialLink to have the complete robot:
  - **my_robot = SerialLink(L, 'name', 'My Cool Robot')**

Inverse kinematics

- The goal is to find the joint angles that locate the end-effector at some desired pose
- A problem of real practical interest, since we usually know where e.g. objects are in Cartesian coordinates
- Solution is not unique, and in some cases no *closed-form* solution exists

Closed-form solution

- Requires that
  - The robot has 6 axes
  - The 3 axes in the wrist intersect at a single point
- Thus, motion of the wrist joints only change the orientation of the end-effector

- What if we don't have this type of robot?
  - We don't for the UR robots

Numerical solution

- Can deal with any number of joints, and any type of robot
- Considerably slower than the closed-form solution

- Basic principle is to model the pose change as a *special spring*
- Spring *forces and torques* (wrench) are proportional to pose change
- Method:
  - Calculate wrench for current pose difference
  - Calculate pose for current estimate of inverse kinematics
  - Resolve wrench to joint torques
  - Calculate joint velocities due to torques
  - Calculate discrete-time update of joint angles
  - Repeat until wrench is small

Trajectories of robot arms

- The same applies as we discussed in the last lecture
  - It's all about smooth motion!

- Two different strategies:
  - Straight lines in joint space - *joint-space motion*
  - Straight lines in Cartesian space - *Cartesian motion*

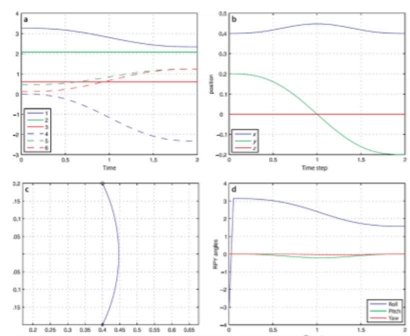
Examining the trajectory

Joint positions

Cartesian positions

Cartesian in xy-plane only

RPY angles



Cartesian trajectories

- In joint space, we cannot guarantee a specific motion of the end-effector
- Cartesian trajectories are useful for
  - Welding, painting, grinding, drawing….
  - Avoiding obstacles

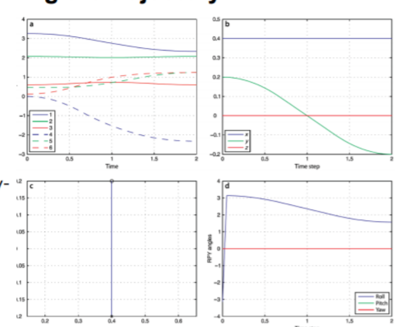
Examining the trajectory

Joint positions

Cartesian positions

Cartesian in xy-plane only

RPY angles



Singularities

- We will now have a look at Cartesian motion through a singularity

- Again, we can generate the Cartesian trajectory, and examine the corresponding joint trajectory for different cases
  - Closed-form solution
  - Numeric solution
  - Pure joint-space trajectory

Joint trajectories

Closed-form solution

Numeric solution

Joint-space trajectory

Manipulability