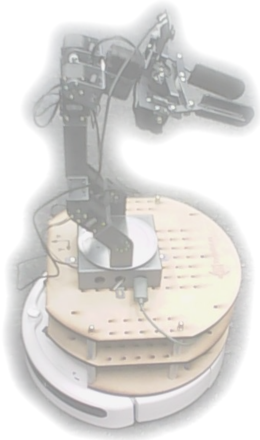
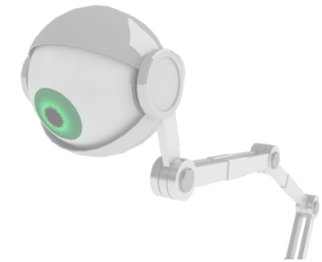
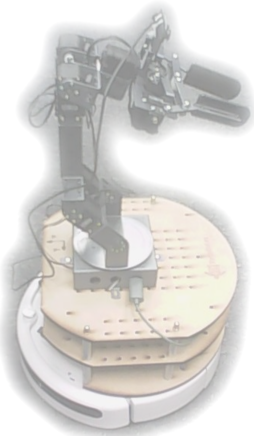
# Armed-Turtlebot



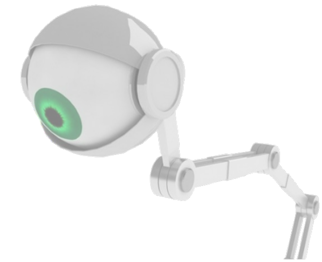
# Outline



1) Introduction
2) Architecture
3) Smart Arm
4) Detection
5) Tracking
6) Final implementation
7) Conclusions

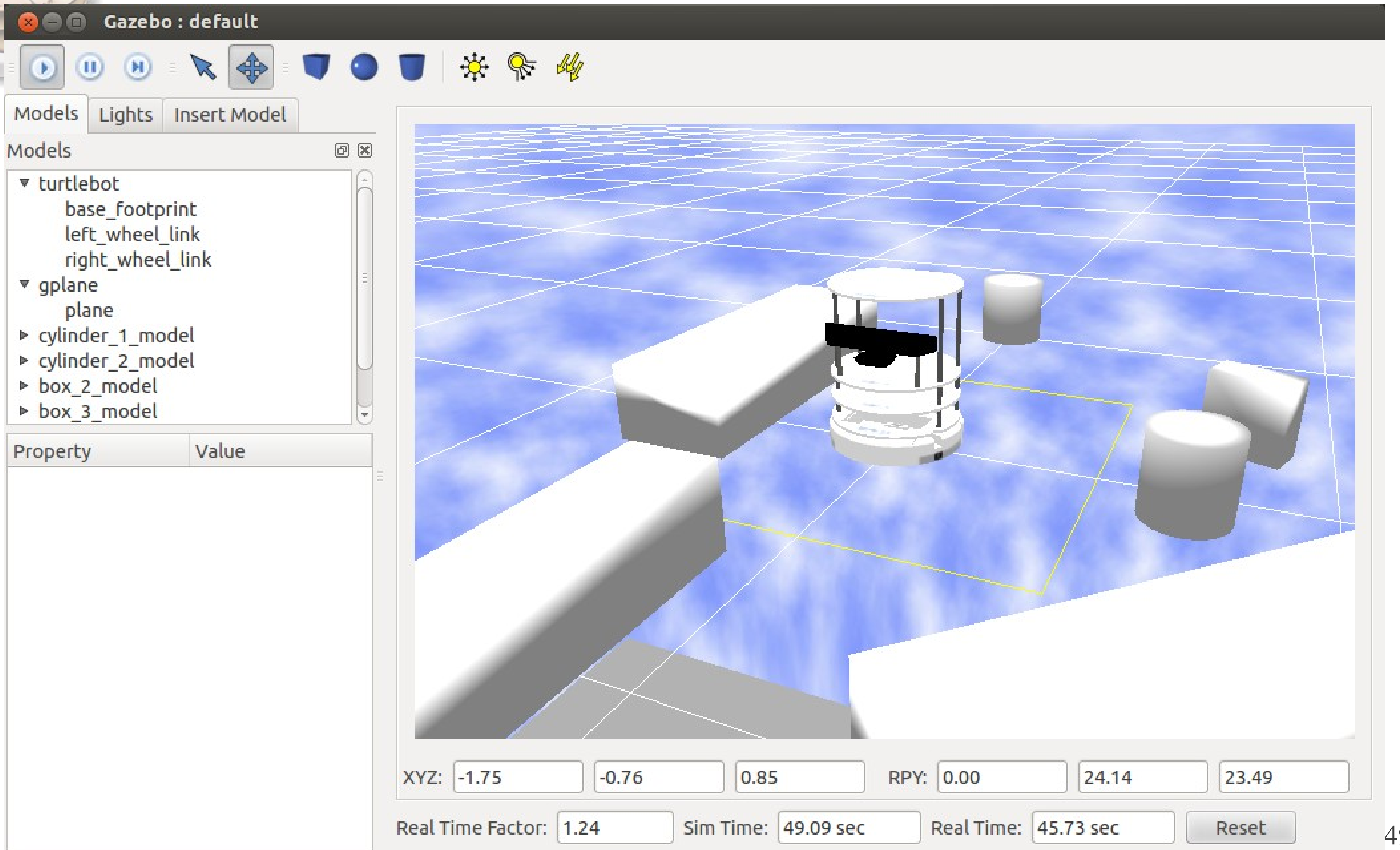


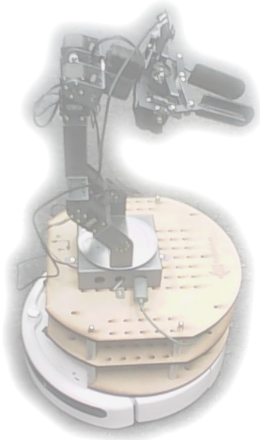
# Introduction

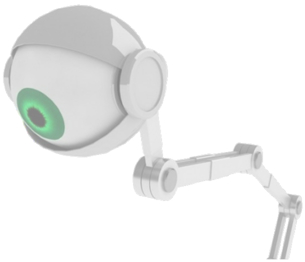


Willow Garage

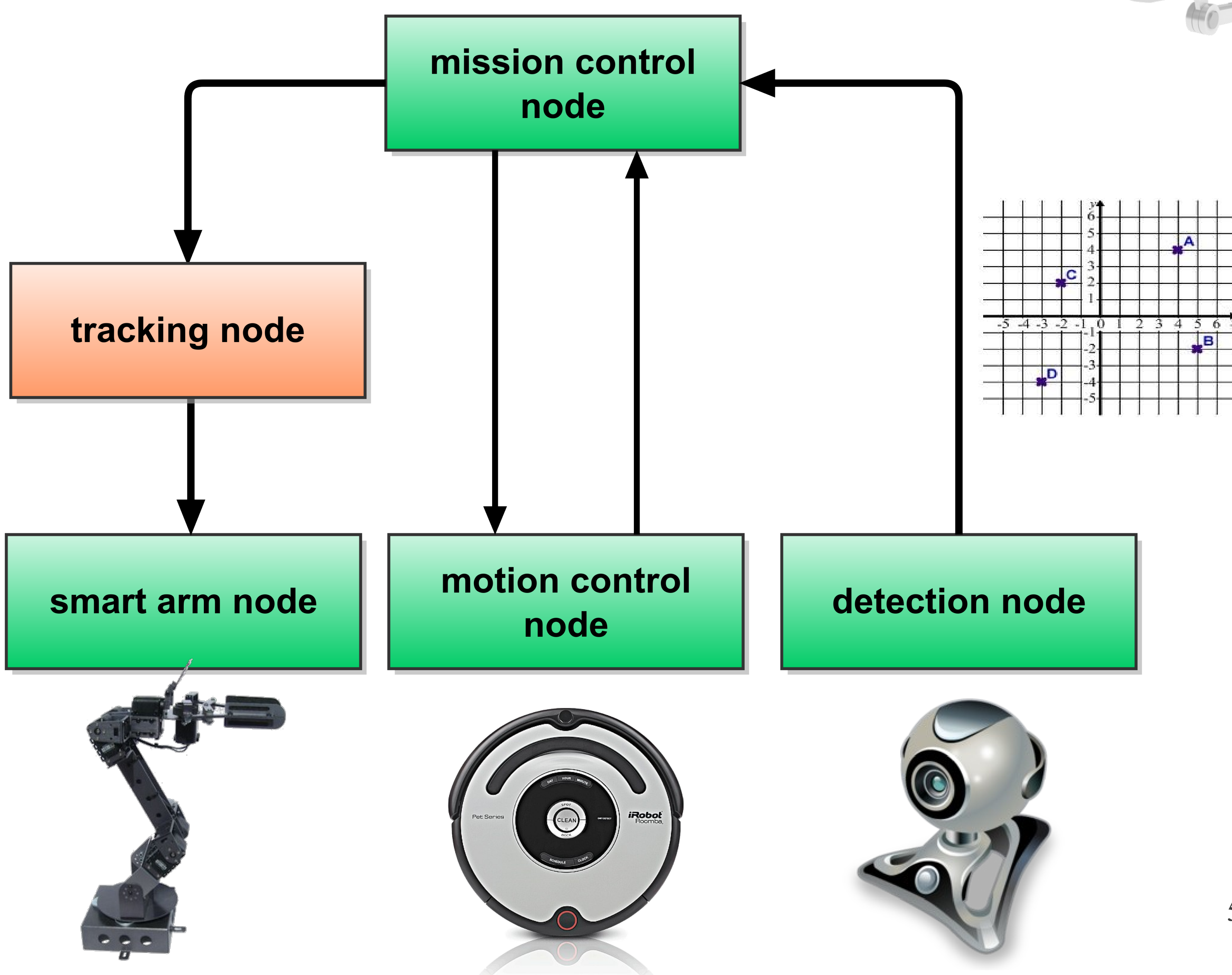
ROS.org

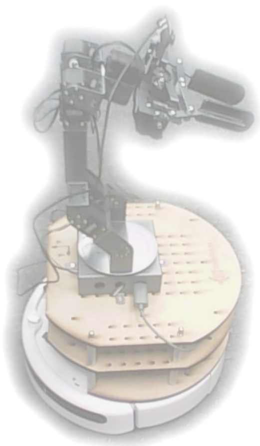

Gazebo : default
Models
Lights
Insert Model
Models
turtlebot
base_footprint
left_wheel_link
right_wheel_link
gplane
plane
cylinder_1_model
cylinder_2_model
box_2_model
box_3_model
Property
Value
XYZ: -1.75 -0.76 0.85
RPY: 0.00 24.14 23.49
Real Time Factor: 1.24
Sim Time: 49.09 sec
Real Time: 45.73 sec
Reset



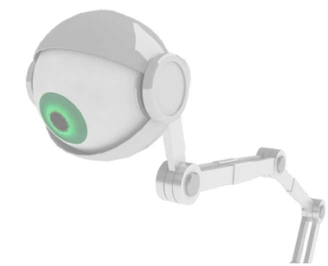


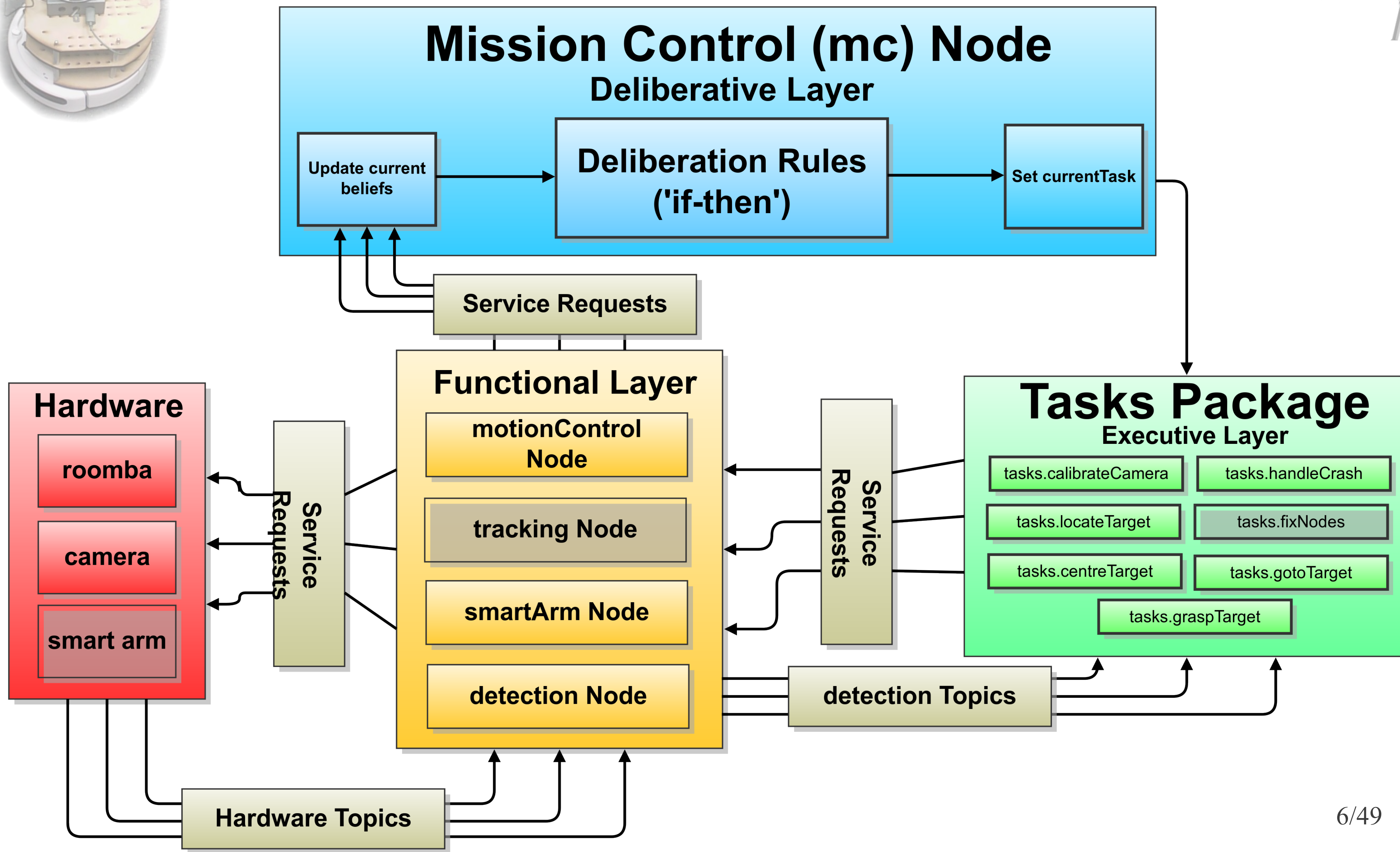
# Basic Architecture

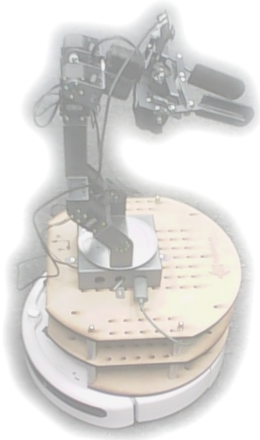


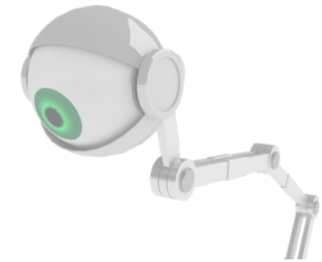


# Detailed Architecture







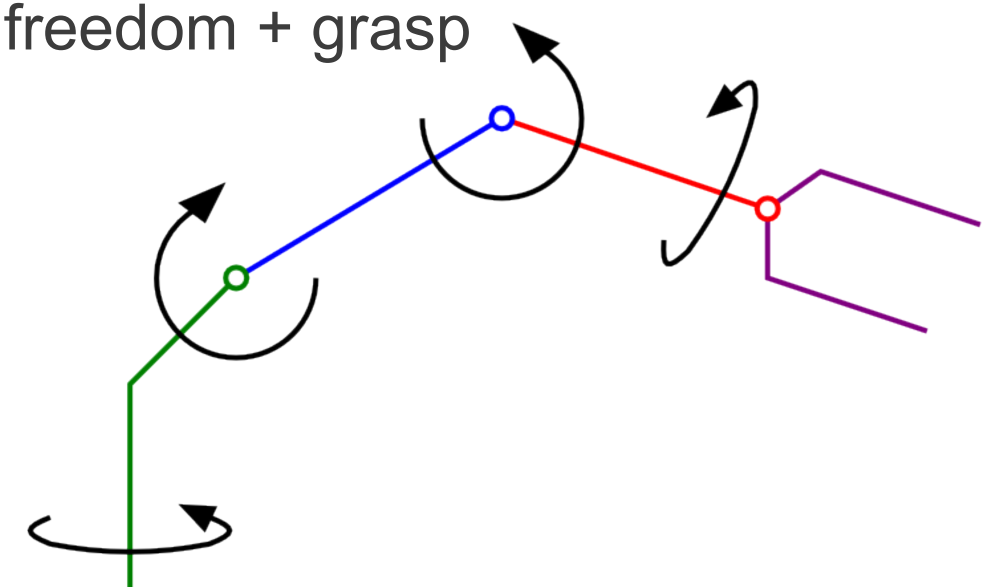


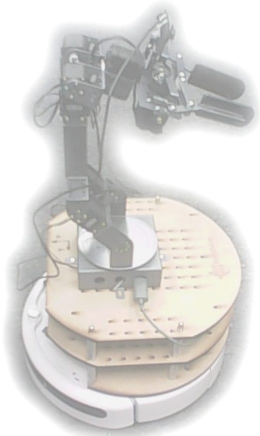
# Robotic Arm

# Robotic Arm

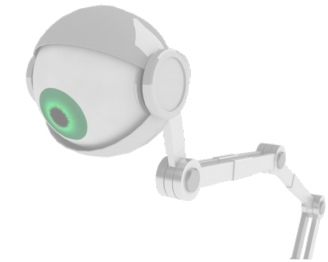
## Smart Arm

- ✔ Aluminum link arm system.
- ✔ Powerful AX-12 servo motors (15 kg/cm and 59 rpm).
- ✔ Voltage, current, position and temperature feedback.
- ✔ Automatic thermal shutdown capability.
- ✔ 4 degrees of freedom + grasp

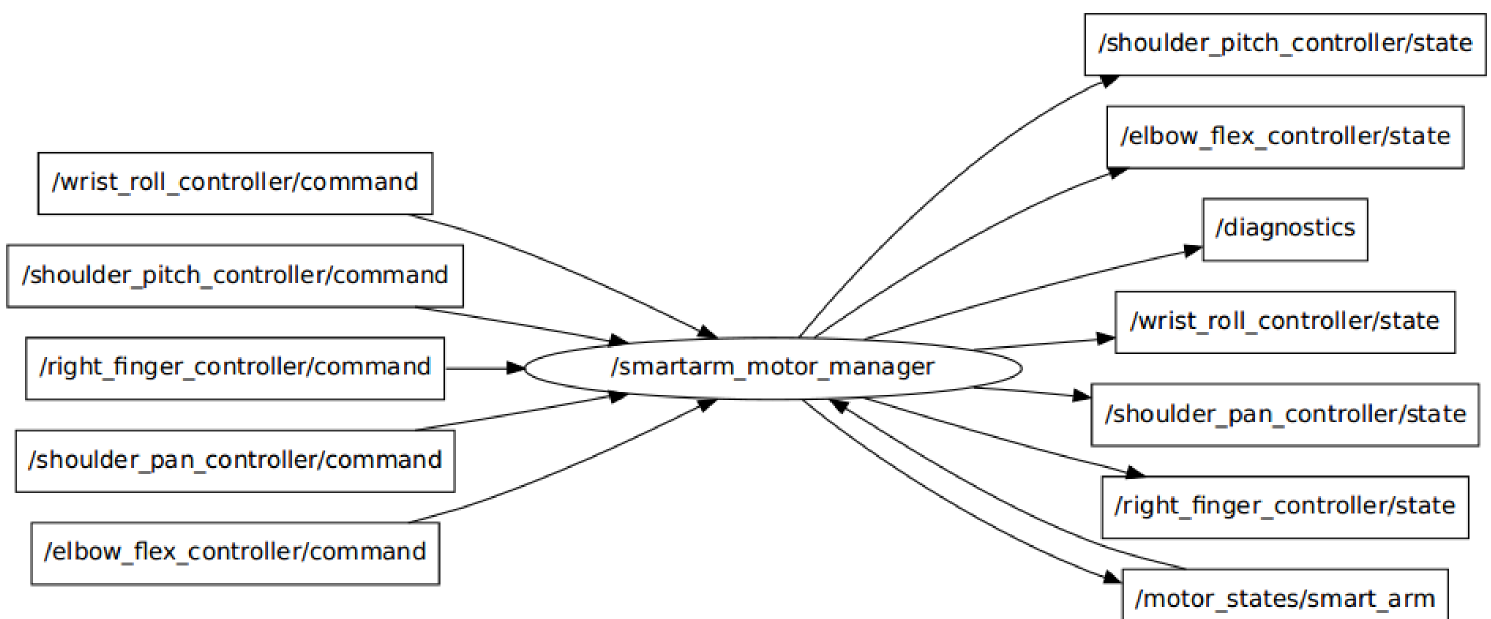


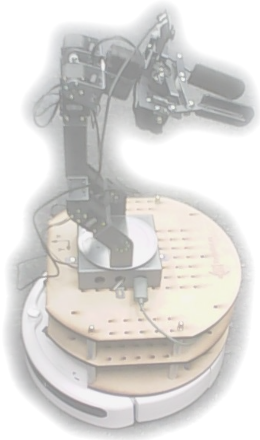


# smart_arm_controller

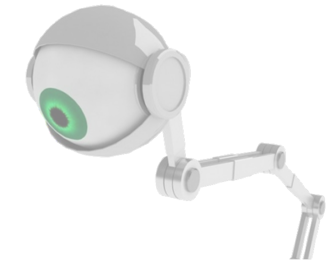


- ✔ Built-in controller for the arm.
- ✔ Provides basic functionality to move each joint and joint status.
- ✔ We need more functionality (FK, IK, move_all, grasp sensing...)

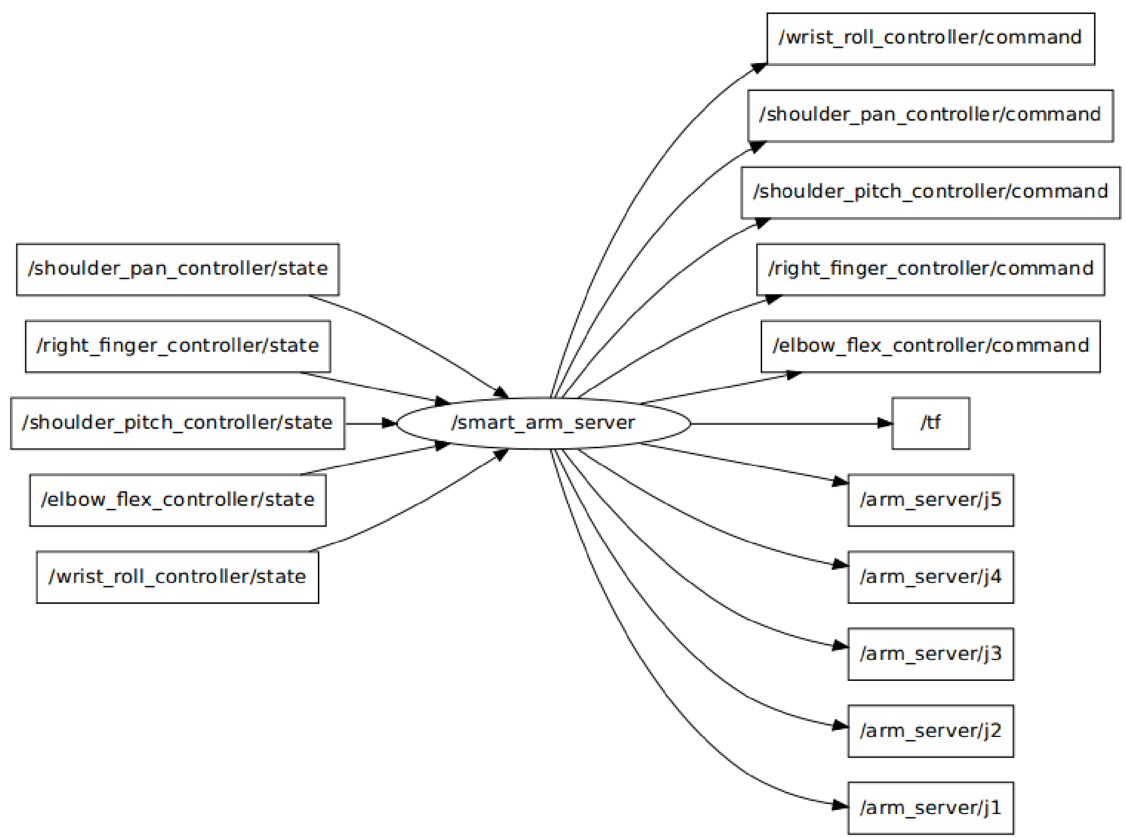


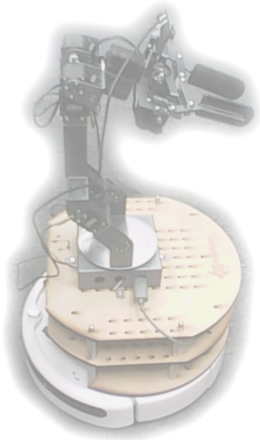


# smart_arm_server

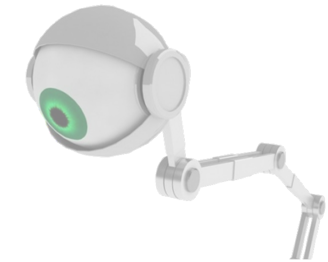


- ✔ Our custom High-level API to control the arm.
- ✔ Provides advanced functionality
- ✔ FK, IK, easy joint movement, grasp sensing...



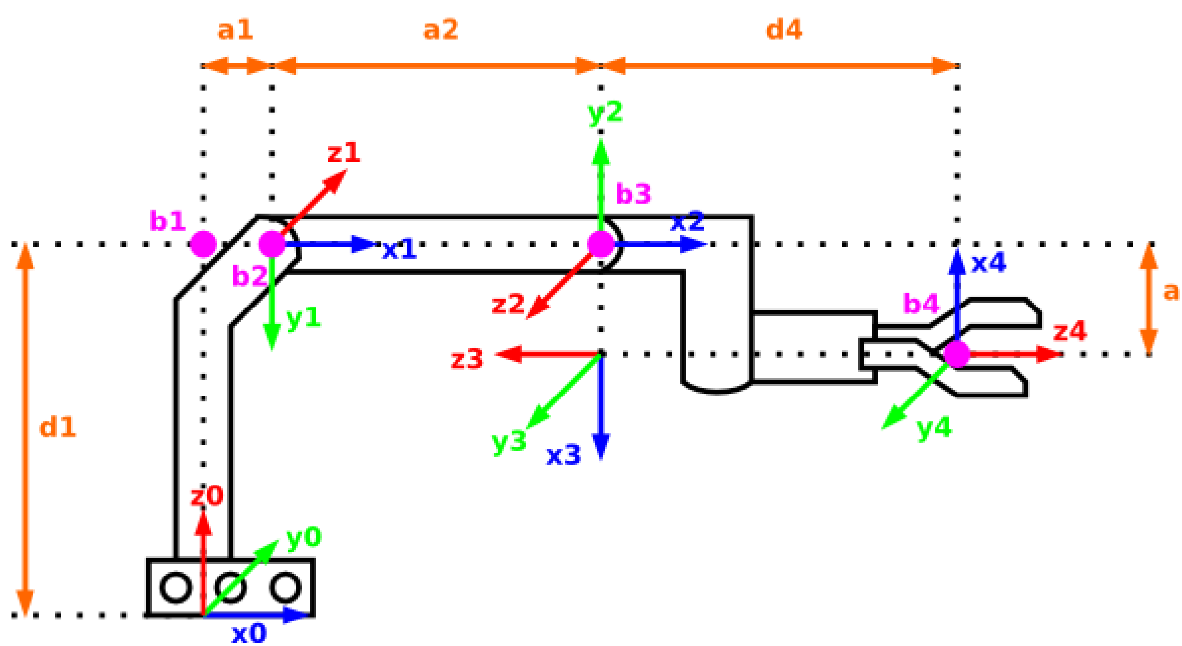


# smart_arm_server



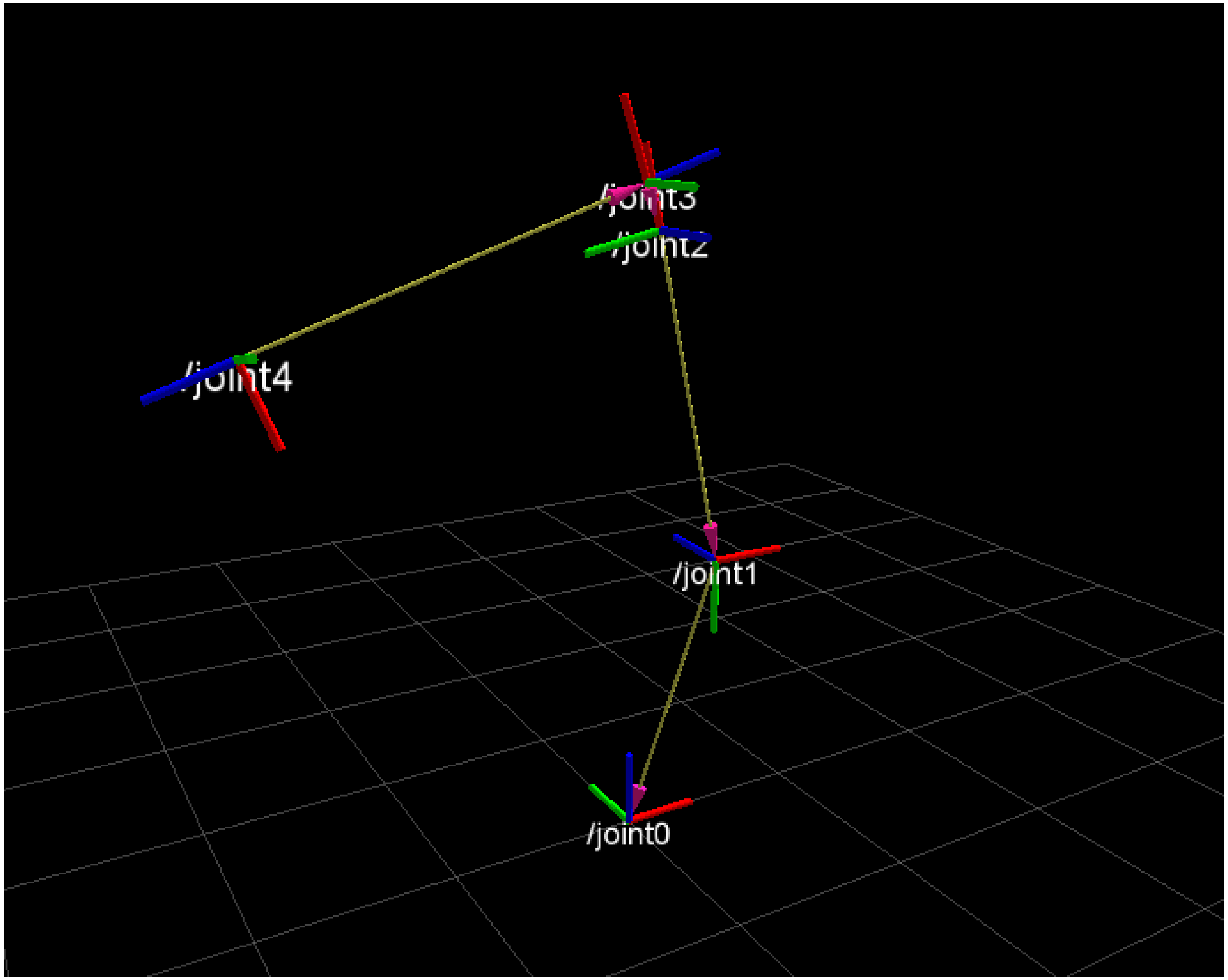
## Forward kinematics

- Follow the algorithm of Denavit-Hartemberg to find the relationship between the arm links.
- Gives the transformation between each pair of links.
- Allows to know the position of the end effector, given the joint positions.
- Allows visualization of the arm in RViz



# smart_arm_server

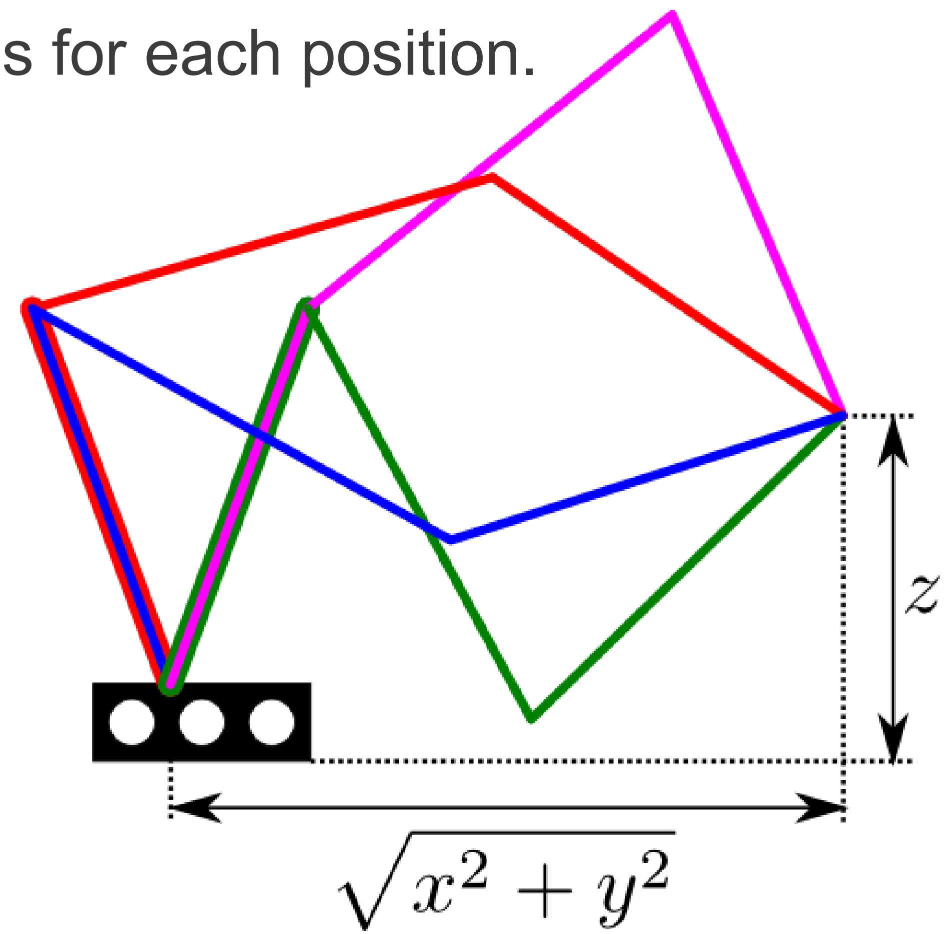
## Forward kinematics



# smart_arm_server

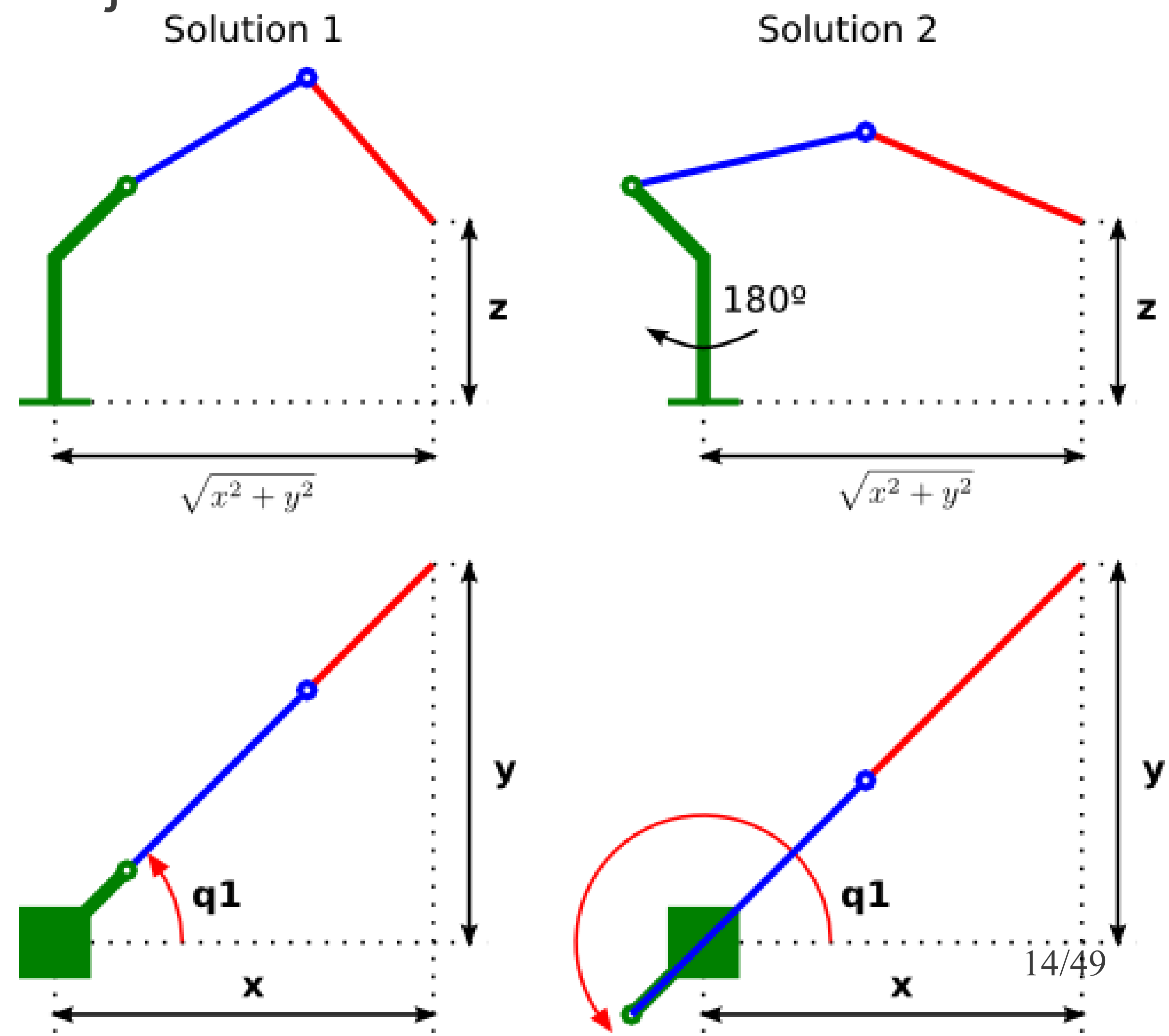
## Inverse kinematics

- To obtain joint positions from an (x,y,z) position of the end effector.
- Up to 4 solutions for each position.

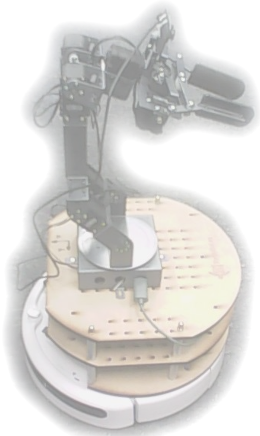


# Inverse kinematics

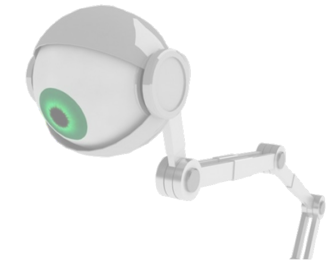
- 2 solutions for the first joint.




q1 = atan2(y,x)
q1 = atan2(y,x) + 180º



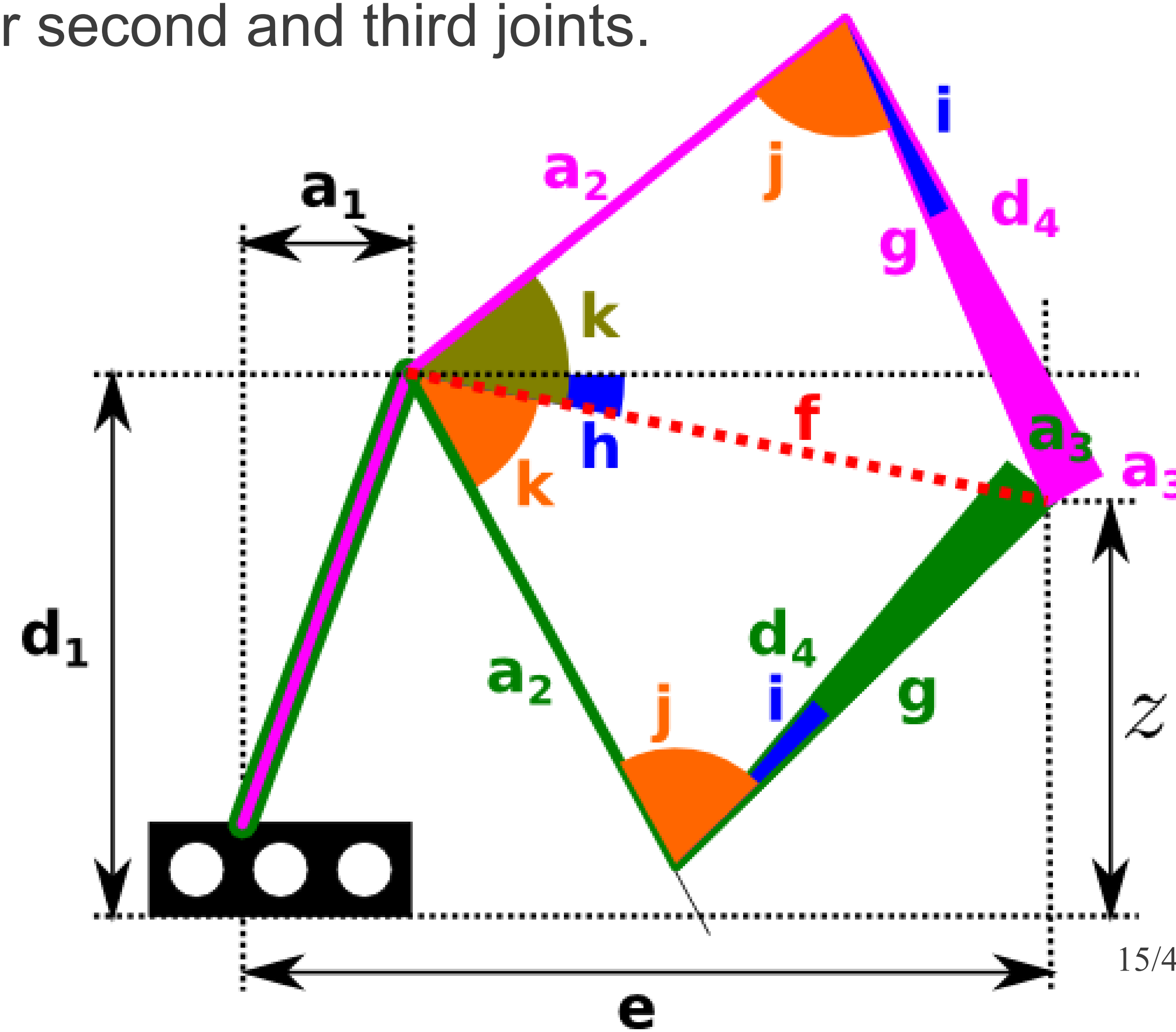
# smart_arm_server



## Inverse kinematics

- 2 solutions for second and third joints.

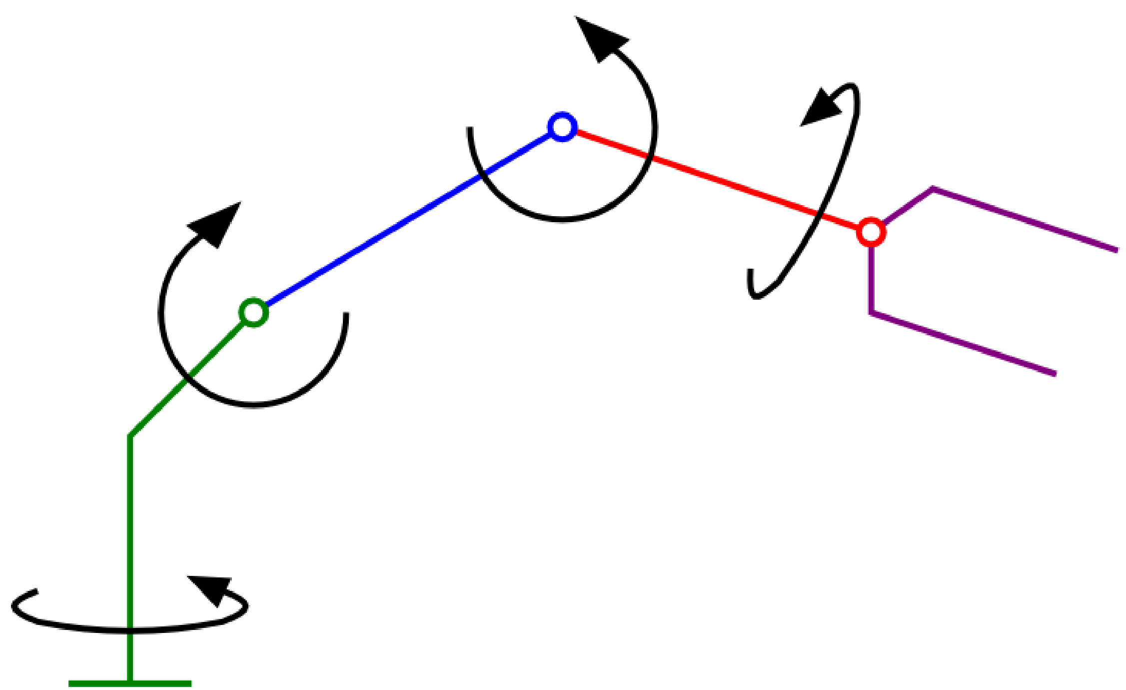
```
e = sqrt(x**2 + y**2)
f = sqrt((e-a1)**2 + (d1-z)**2)
g = sqrt(d4**2 + a3**2)
h = arctan2(d1-z,e-a1)
i = arctan2(a3,d4)
j = arccos(-(f**2 - a2**2 - g**2)/
(2*a2*g))
k = arctan2(g*sin(pi-j),
a2+g*cos(pi-j))

# First solution
q1 = arctan2(y,x)
q2 = h + k
q3 = pi - j + i

# Second solution
q1 = arctan2(y,x)
q2 = -(k - h)
q3 = -(pi-j) + i
```

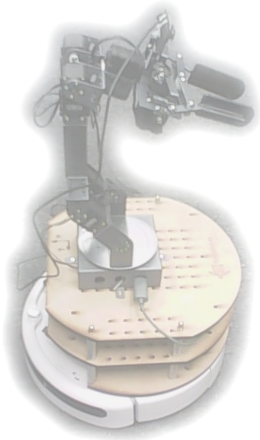


# smart_arm_server

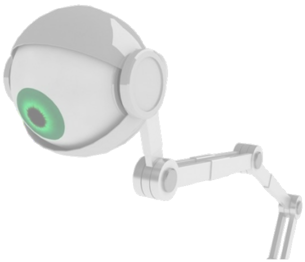
## Inverse kinematics

- The fourth joint is set to zero.
- Its rotation does not affect result.





# smart_arm_server

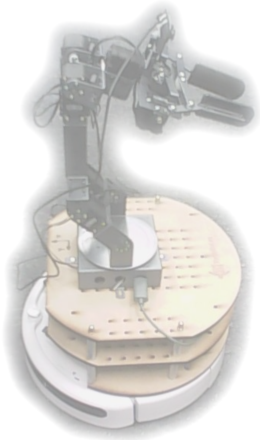


## Sensible grasping

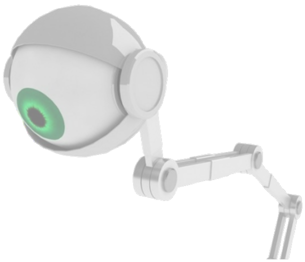
- The feedback of motors is used.
- The motor is stopped when certain load is detected.

## Implementation

- Also easy methods for moving a joint, move all, move to an (x,y,z) position, go home and retrieve joint status.
- All in a Arm() class.
- Arm() inside a ROS node : smart_arm_server
- Contains also a service to easy access all methods.



# smart_arm_server



## Service

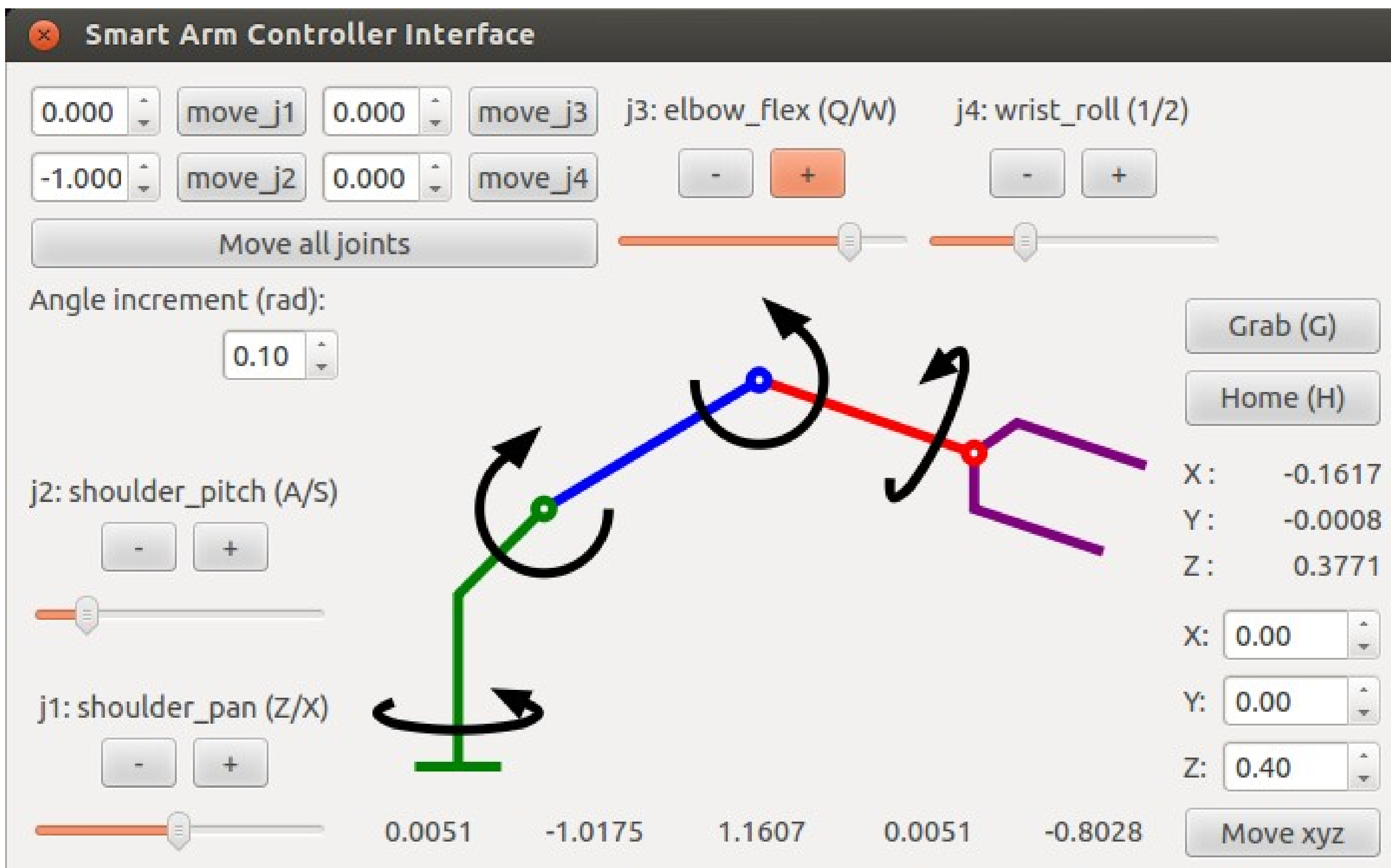
- A service to easy access all methods.

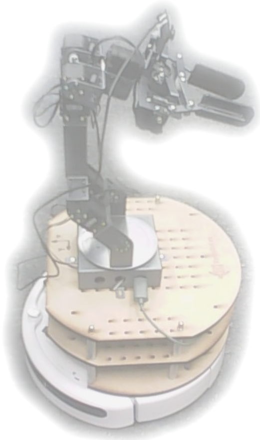
```
int8 what
float64[] data
---
float64[] values



Accepts external requests about status and movements, depending on the  'req.what' value.
          0 : joint status
        1-5 : move a single joint
          6 : move the four first joints to desired joint angles
          7 : move the four first joints to desired (x,y,z) using IK solver
          8 : grab/ungrab with the hand
          9 : ask for (x,y,z) position of end effector
         10 : go to home position
    Data should be provided for some of the requests.
    A response according to what is accomplished is returned.
```

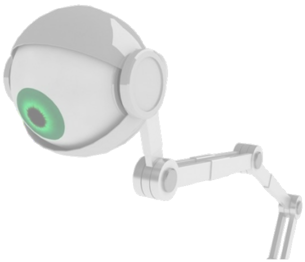
# smart_arm_gui

## GUI

- Implemented in Qt.
- Easy access to all the methods provided by the smart_arm_server service.

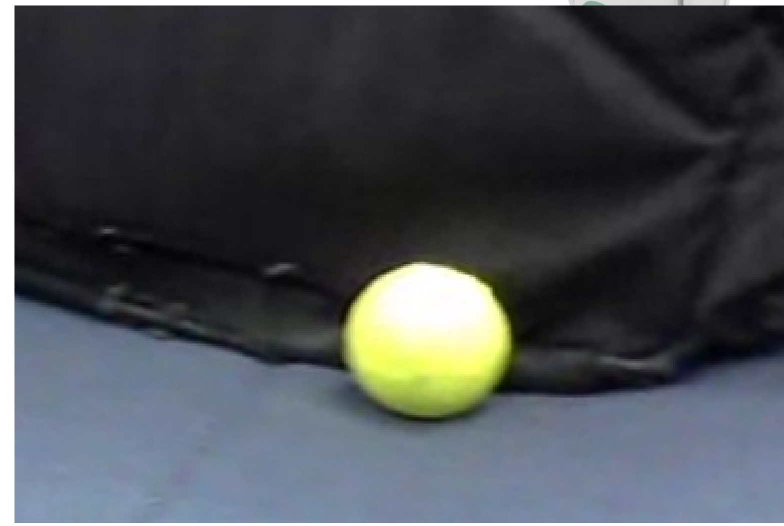






# Object detection

# Object detection



**Assumptions:**

- The object is a ball.
- Detection is based on color and shape

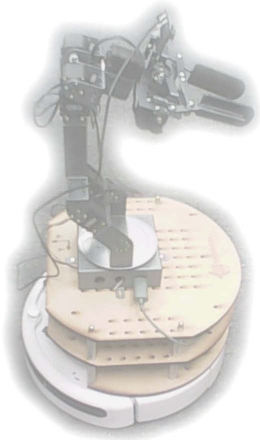
**Input / Output:**

**Subscriber:**

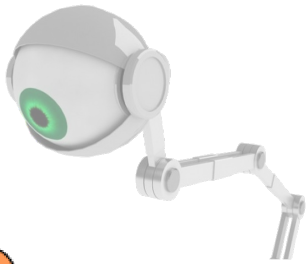
- /camera/image_raw : RGB image from webcam

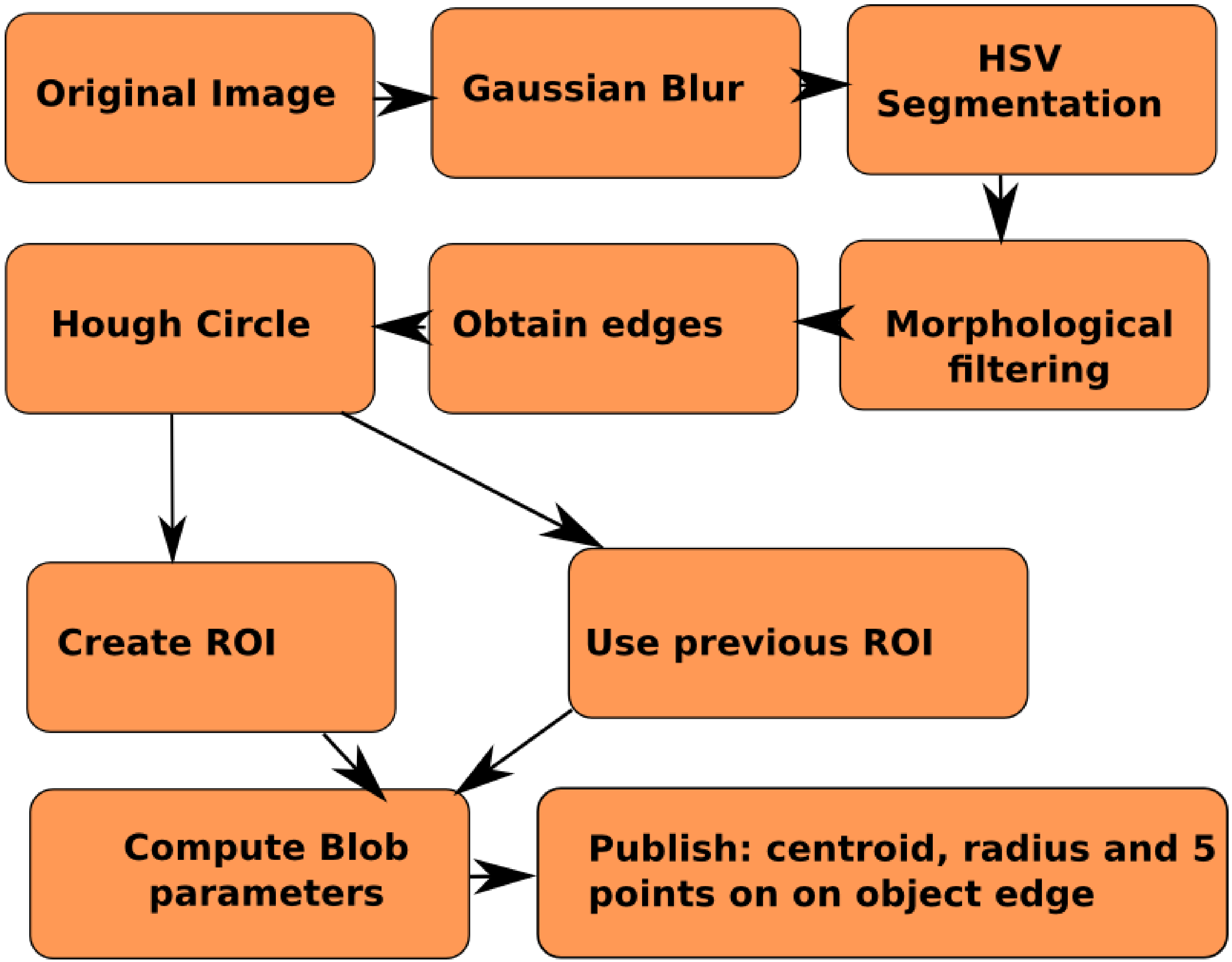
**Publisher:**

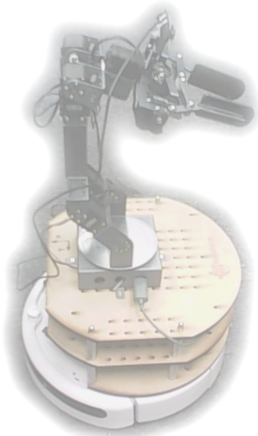
- /detection/posrad : “posx posy radius”
- /detection/5point : “p1x p1y p2x p2y …. p5x p5y “



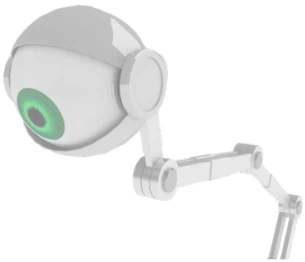
# Object detection







# Object detection



**Debug mode:**

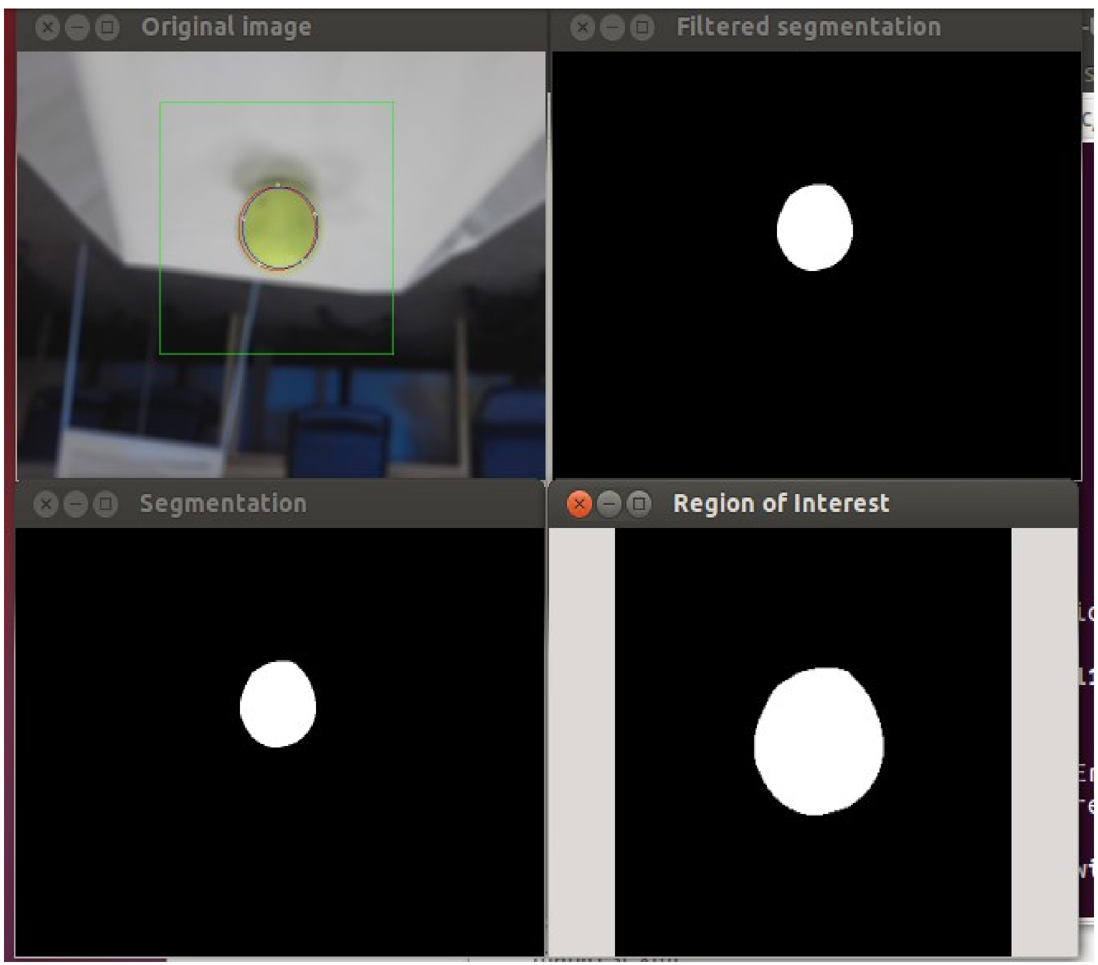
- Option 0 – No debug

- Option 1 – Show windows: Original Image, Segmented, filtered segmentation and ROI

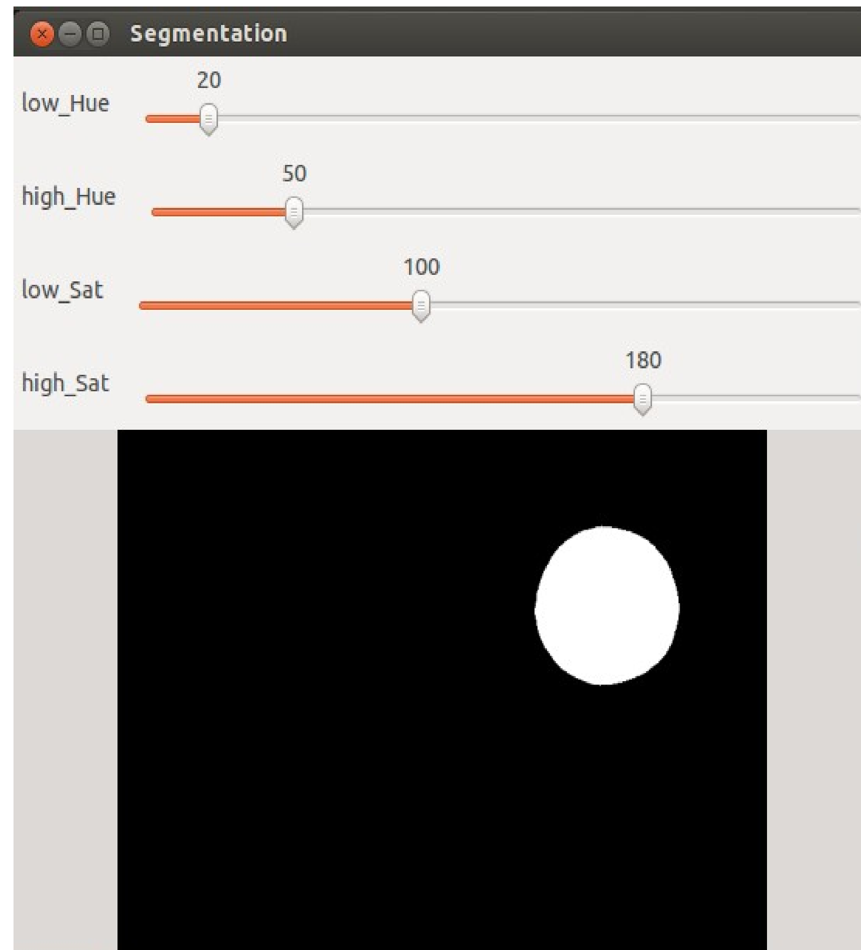
- Option 2 – Windows + Segmentation window with sliders for Hue and Saturation ranges

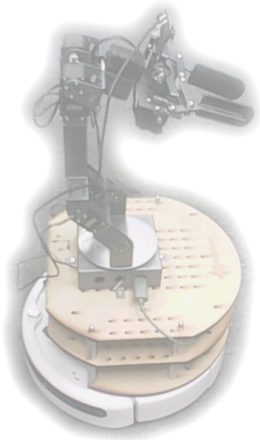
# Object detection

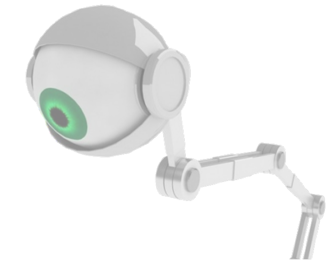
**Debug 1**




**Debug 2**



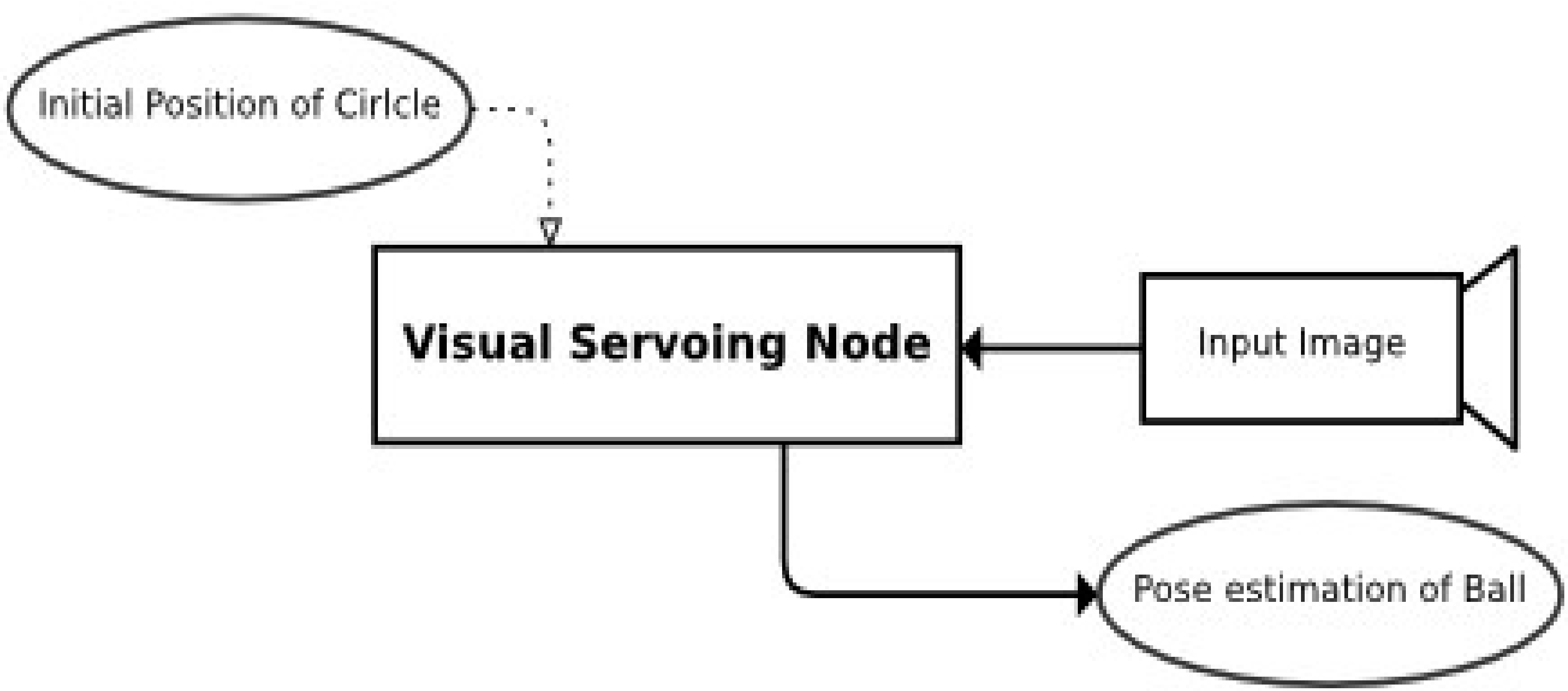




# Visual Servoing

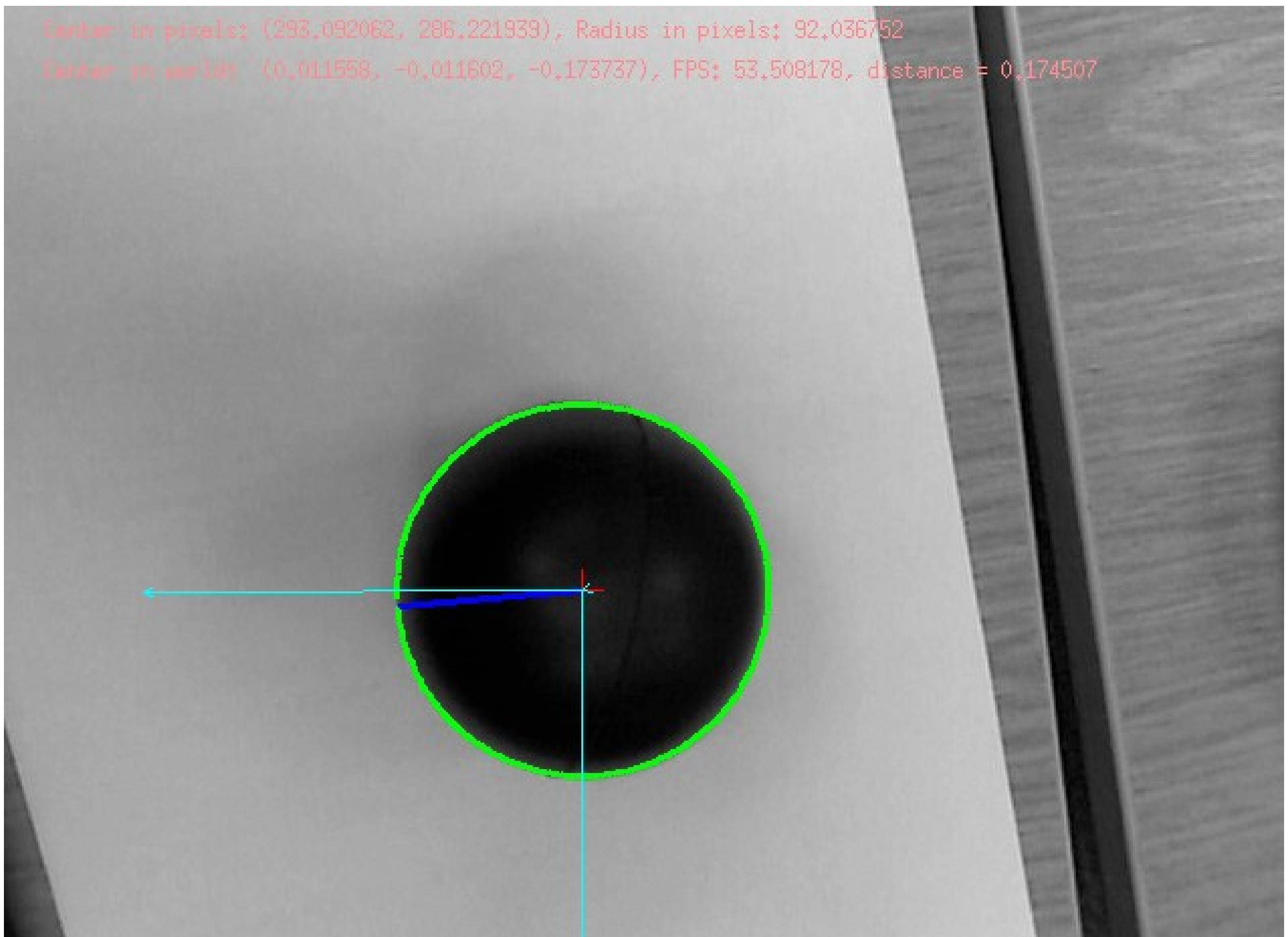
# Visual Servoing

Main Idea:



# Visual Servoing

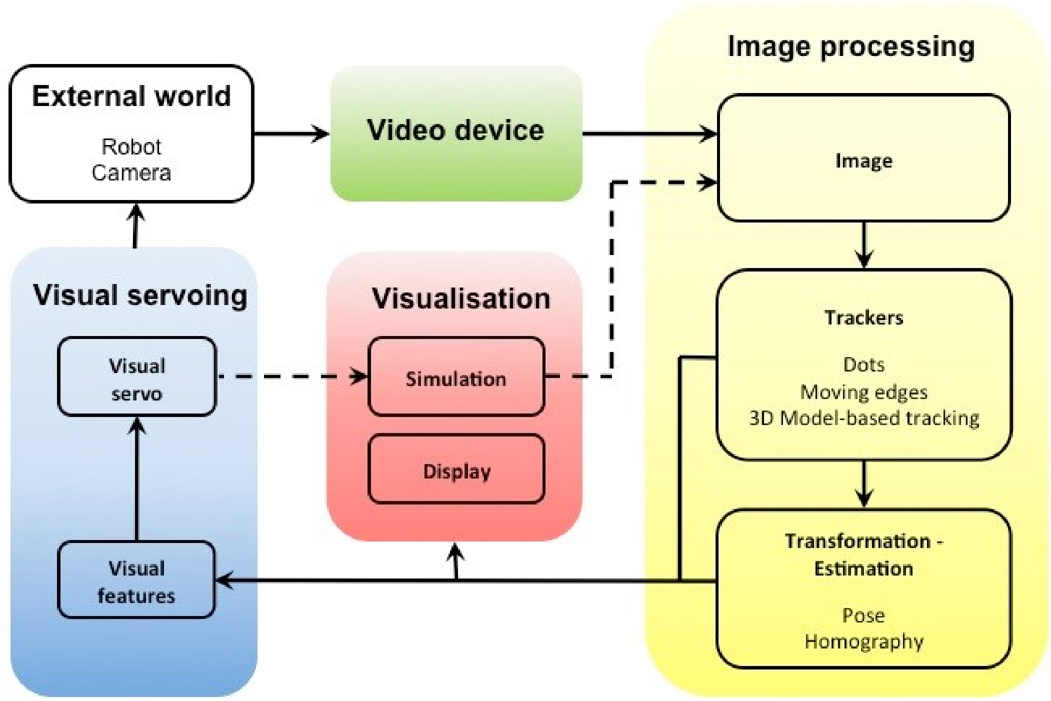
From initial detection we can track using edges



# Visual Servoing

Tracking edges allows us to:

- Have a much higher accuracy
- Have a more stable estimation of the circle
- Much higher frame rate
- Much of the tracking is already in the ViSP library.

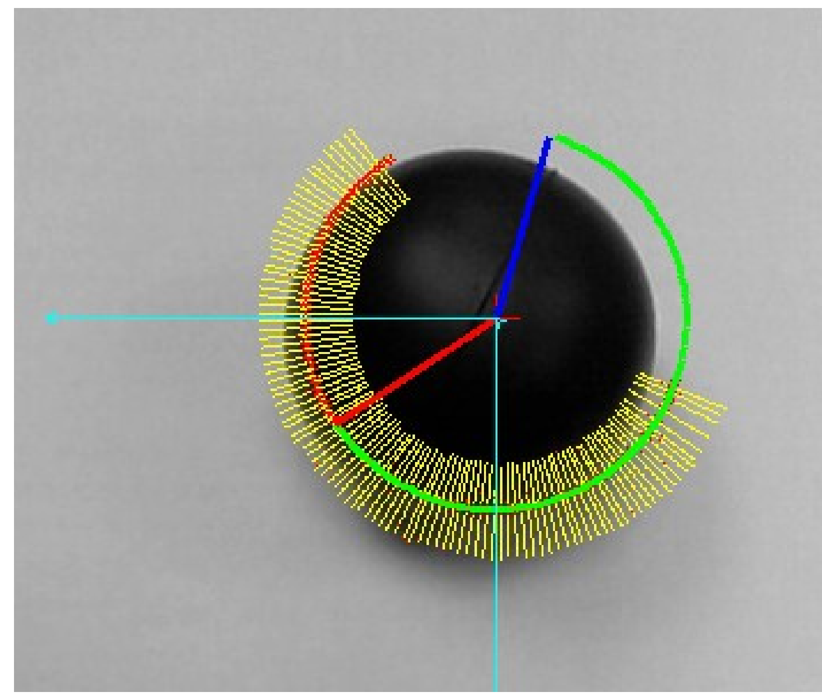


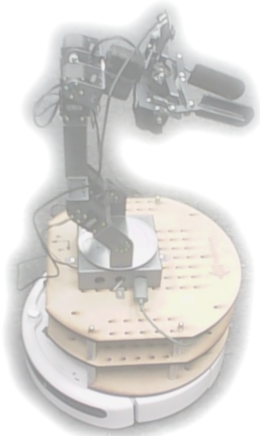
# Visual Servoing: Steps for Estimation

1. Draw perpendicular lines to previous circle estimate

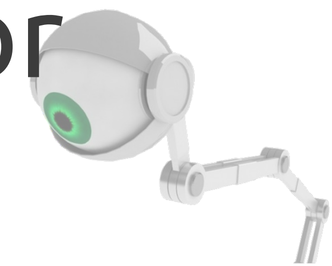
2. Find best edges along that line (1D problem)

3. Fit a circle model into the edges found (SVD).





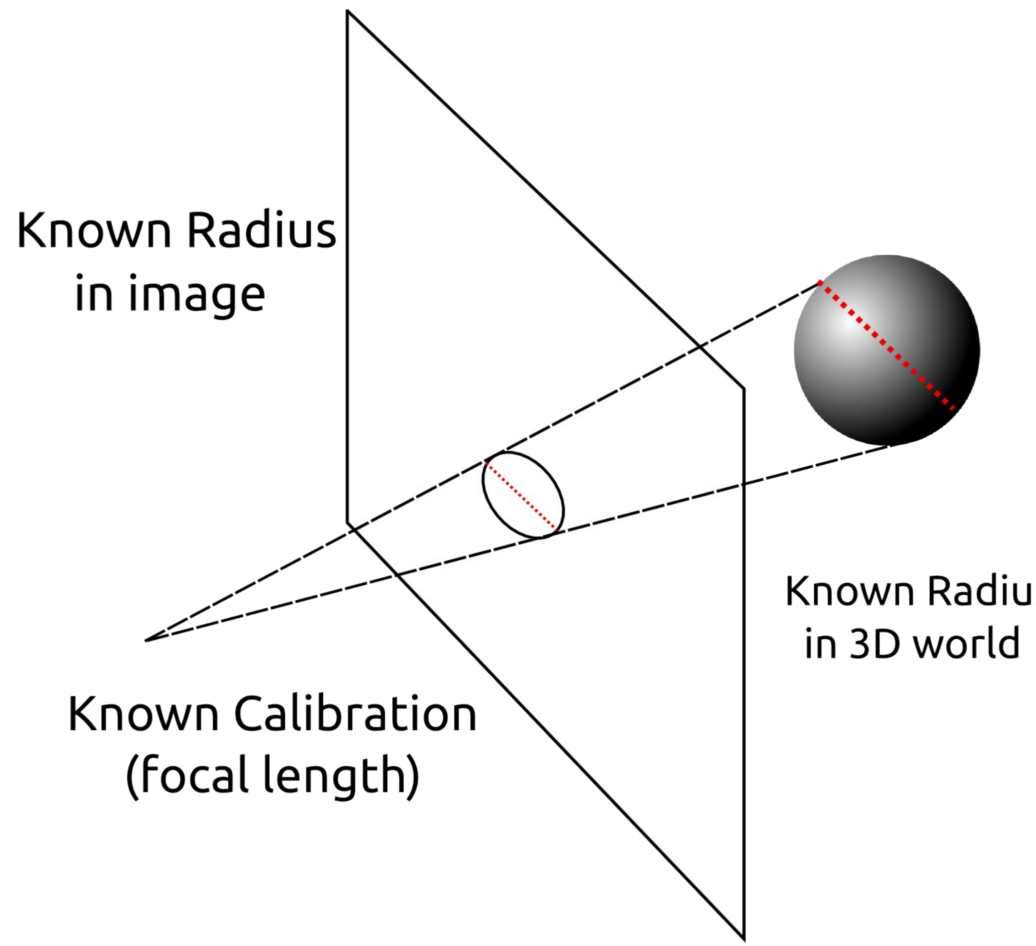
# Visual Servoing: Steps for Estimation



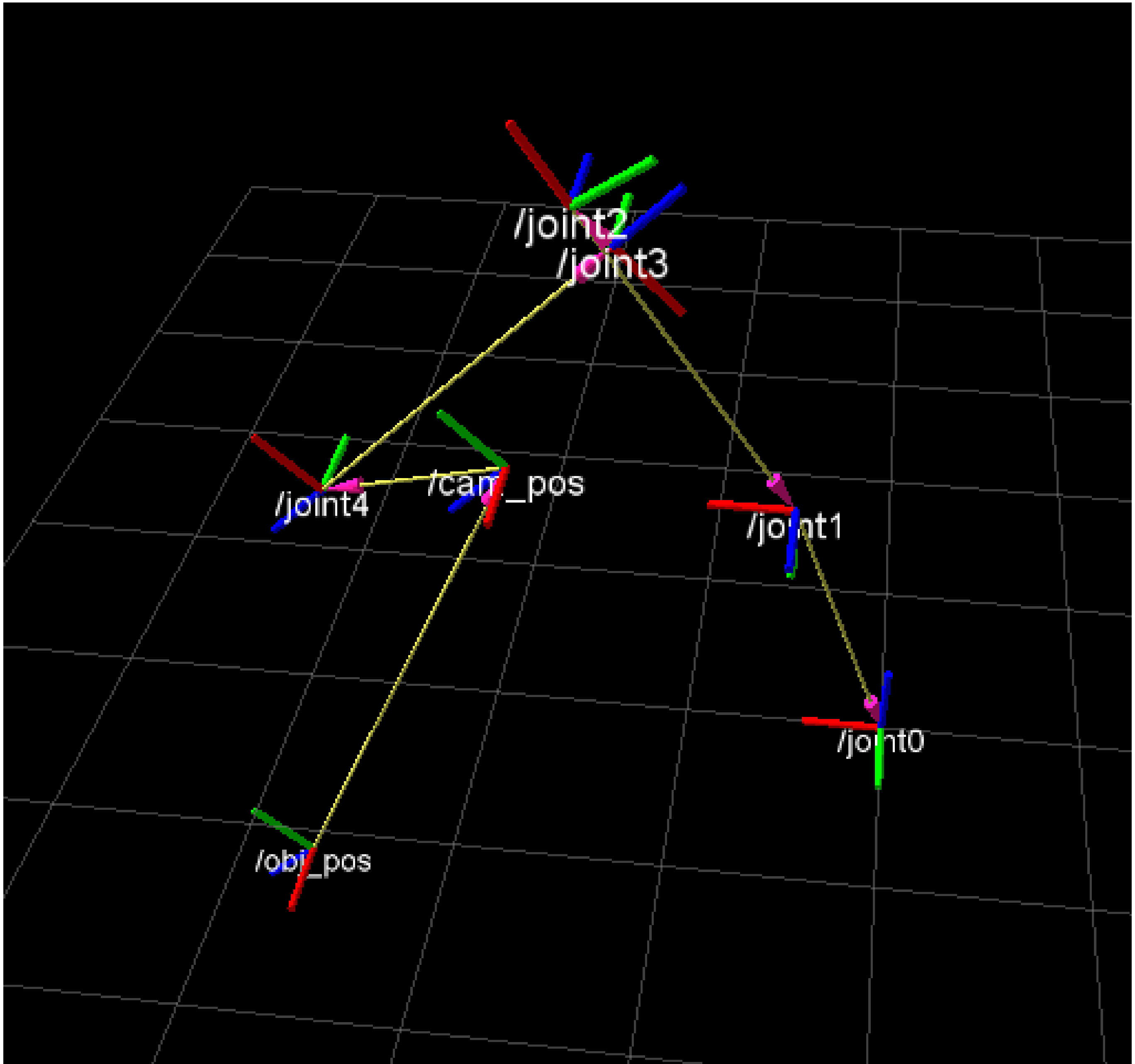
4. Extract the pose vector in pixels (x,y,r)

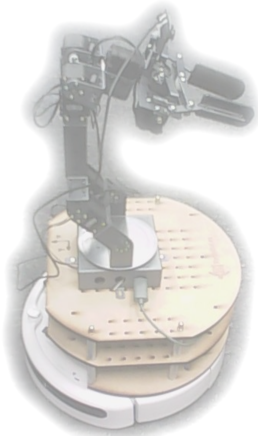
5. Convert the pose in pixels to a 3D coordinate using camera calibration.

6. Publish state to RViz



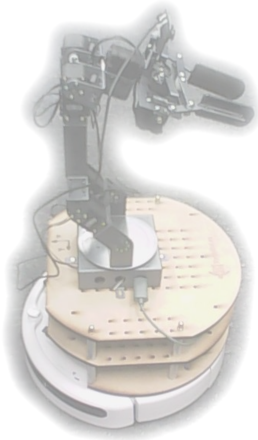
# RViz





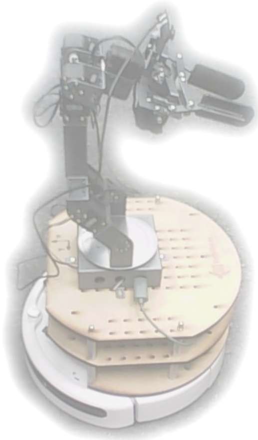
# Visual Servoing: Parallelization

- Motivation
  - To increase the frames per second
  - More frame per second, more robustness to fast robot motion
  - More fps also allows more time for more post-processing of the image to increase stability of output



# Visual Servoing: Parallelization

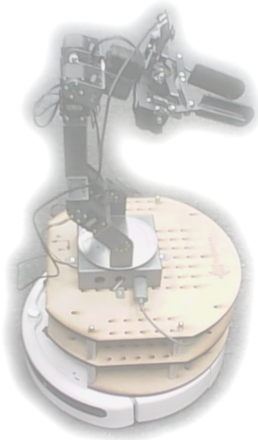
1. Draw perpendicular lines to previous circle estimate
2. Find best edges along that line (1D problem)
3. Fit a circle model into the edges found (SVD).
4. Extract the pose vector in pixels (x,y,r)
5. Convert the pose in pixels to a 3D coordinate using camera calibration.



# Visual Servoing: Parallelization

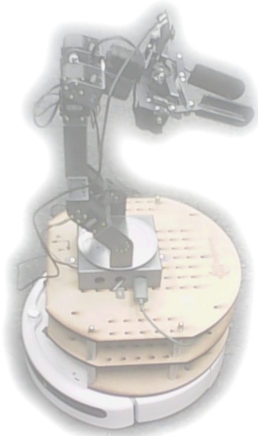
1. Draw perpendicular lines to previous circle estimate
2. Find best edges along that line (1D problem)
3. Fit a circle model into the edges found (SVD).
4. Extract the pose vector in pixels (x,y,r)
5. Convert the pose in pixels to a 3D coordinate using camera calibration.

?



# Visual Servoing: Parallelization

1. Draw perpendicular lines to previous circle estimate
2. Find best edges along that line (1D problem)
3. Fit a circle model into the edges found (SVD).
4. Extract the pose vector in pixels (x,y,r)
5. Convert the pose in pixels to a 3D coordinate using camera calibration.



# Visual Servoing: Parallelization

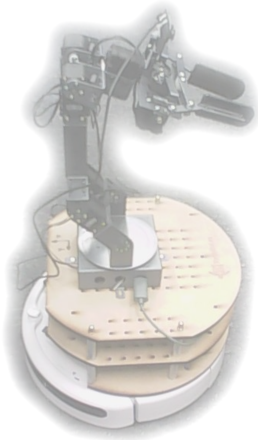
## This was found using Intel's Profiler: VTune

3. Fit a circle model into edges found (SVD).

| Call Stack | CPU Time:Total by Utilization (Idle / Poor / Ok / Ideal / Over) |
| --- | --- |
| ▾vpMeEllipse::leastSquare | 28.1% |
| ▾vpMatrix::operator* | 20.5% |
| vpMatrix::mult2Matrices | 20.5% |
| ▸vpMatrix::pseudoInverse | 3.3% |
| ▸vpMatrix::setIdentity | 1.7% |
| ▸vpRobust::MEstimator | 0.9% |
| ▸vpMatrix::operator* | 0.7% |

2. Find best edges along that line (1D problem)

| | |
| --- | --- |
| ▸vpColVector::vpColVector | 0.1% |
| ▾vpMeTracker::track | 17.7% |
| ▾vpMeSite::track | 17.5% |
| ▾vpMeSite::convolution | 14.2% |
| ▸vpImage<unsigned char>::operat | 1.1% |
| vpMatrix::operator[] | 1.1% |
| ▸vpMe::getMask | 0.6% |
| ▸vpMath::round | 0.2% |
| ▸vpMe::getMaskSize | 0.2% |



# Visual Servoing: Parallelization

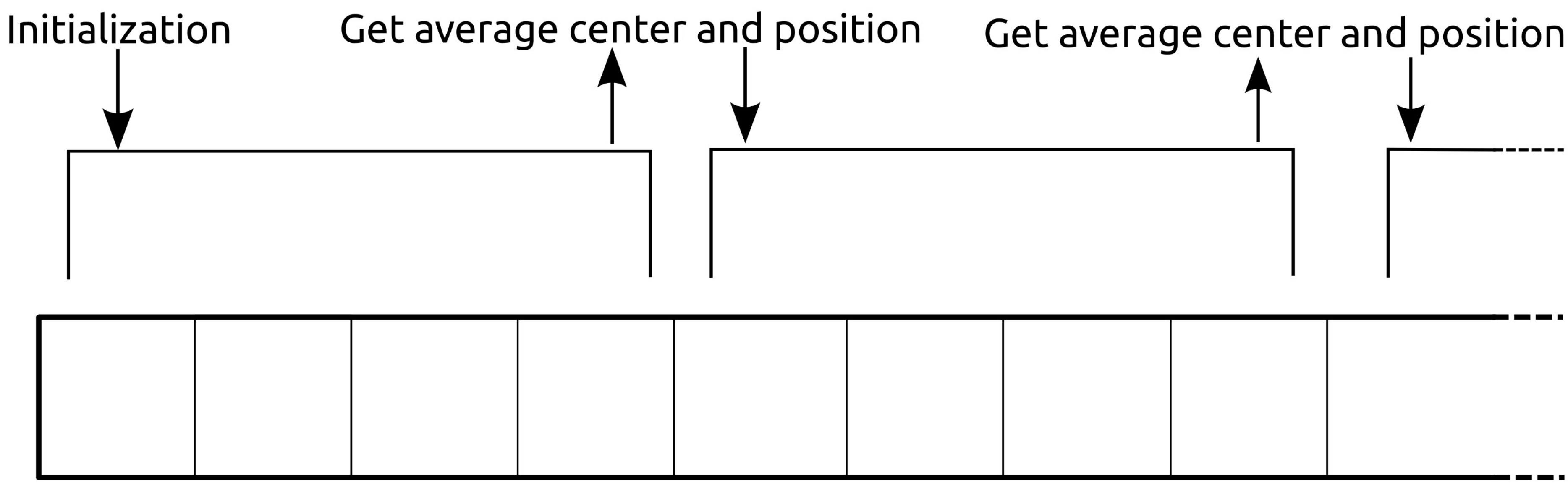
Status of First Approach:

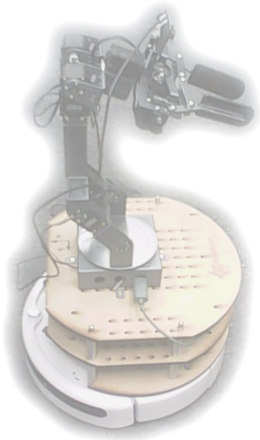
- Code for matrix multiplication is parallelizable and a 50% gain was observed so far.
- Convolution no yet tested.

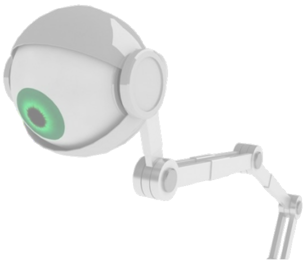
Second Proposed Approach:

- Make a 4 frame buffer, run tracking in parallel, average radius and position: more stability from estimation!

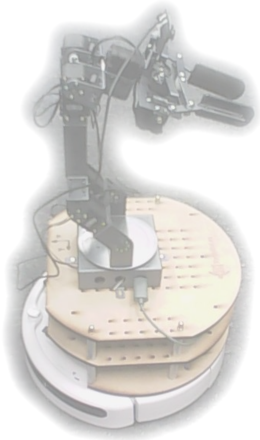
# Visual Servoing: Parallelization



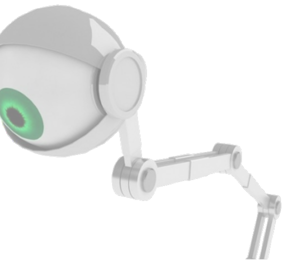




# Final Implementation

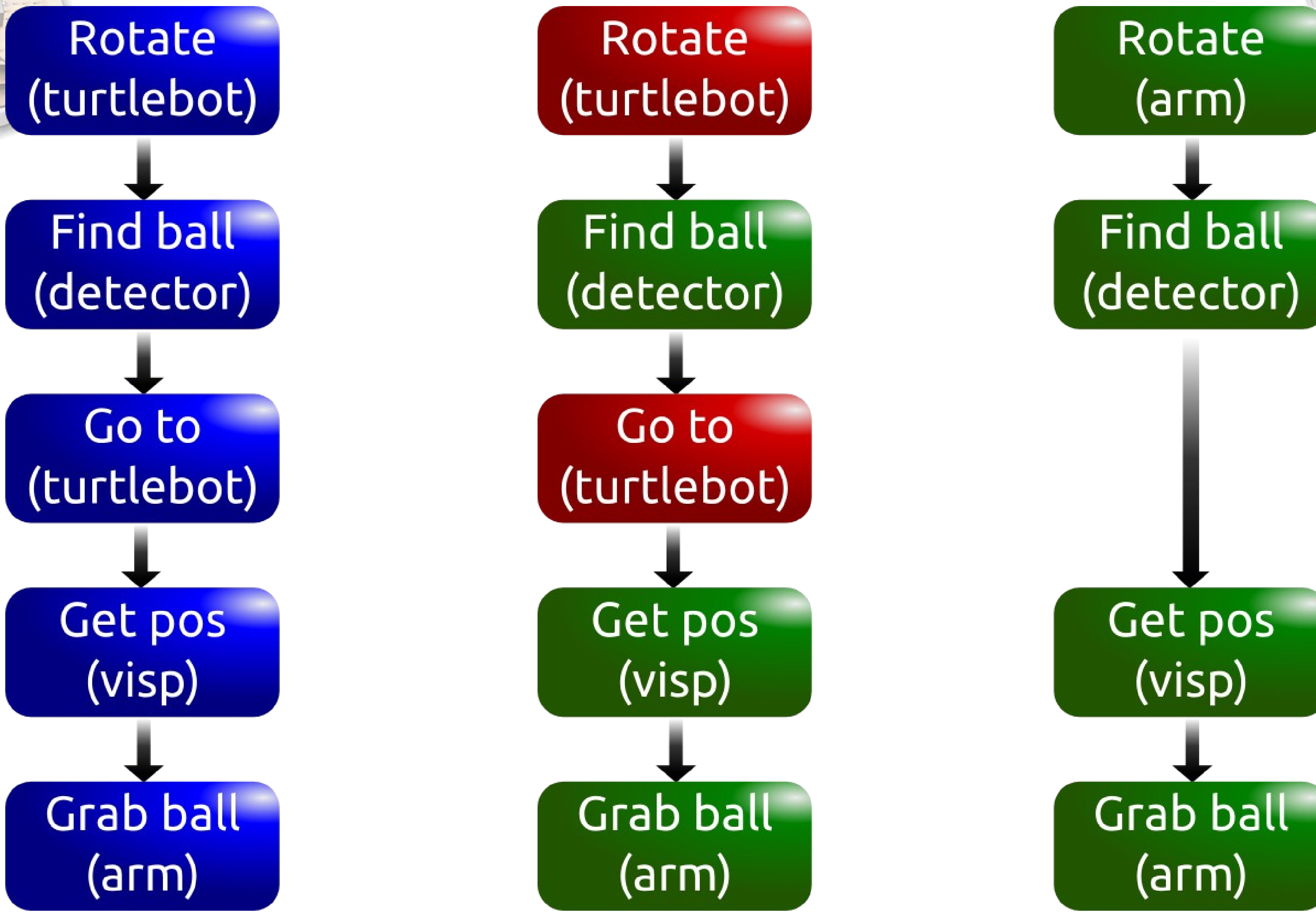


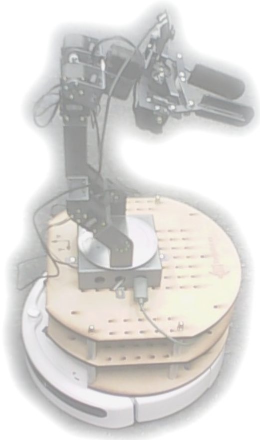
# Project plan



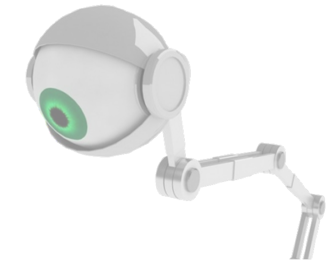
- Guillem Vallicrosa → Arm
- Marc Barnada → Object Detection
- Federico Camposeco → Visual Servoing
- James Rowell → Turtlebot

# Project plan



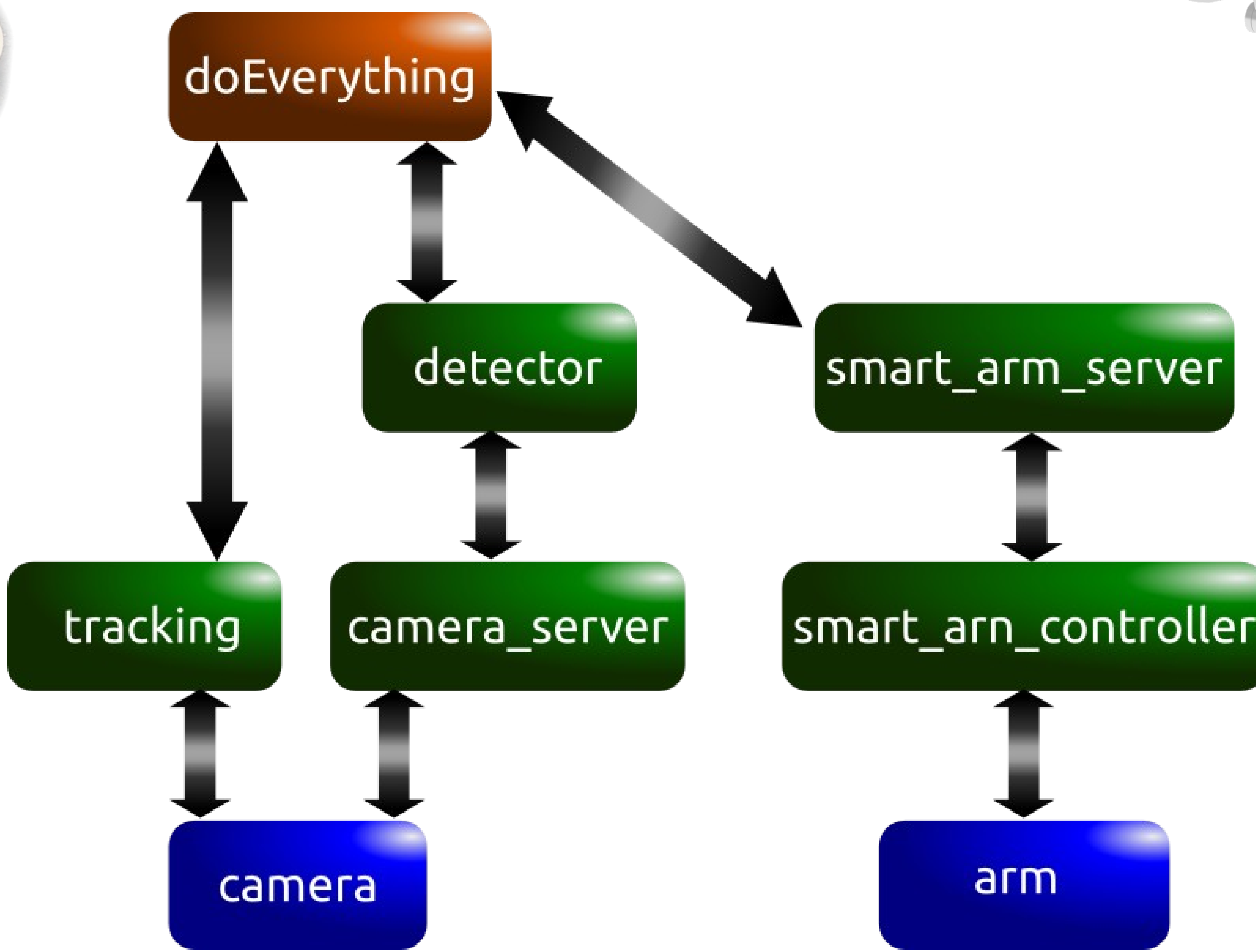


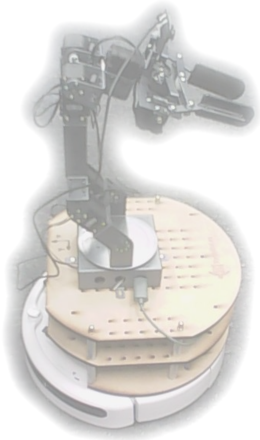
# Project plan



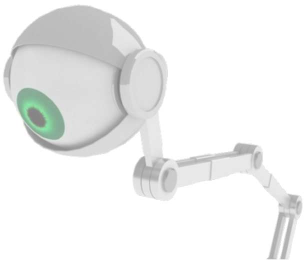
- Move arm to home position
- Subscribe to detector topics
- Rotate arm until ball in center of image
- Stop arm
- Kill detector to free camera
- Launch tracking node
- Listen for 10 positions
- Move arm with security
- Move to ball position
- Grab the ball
- Move arm with security
- Go back home

# Project plan





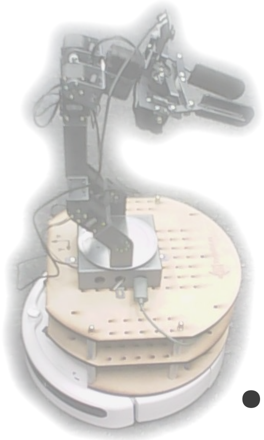
# Project plan

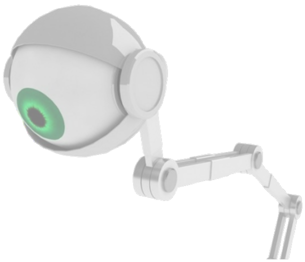


## Video demo

http://www.youtube.com/watch?v=Y_vMQx4dcy4&feature=youtu.be

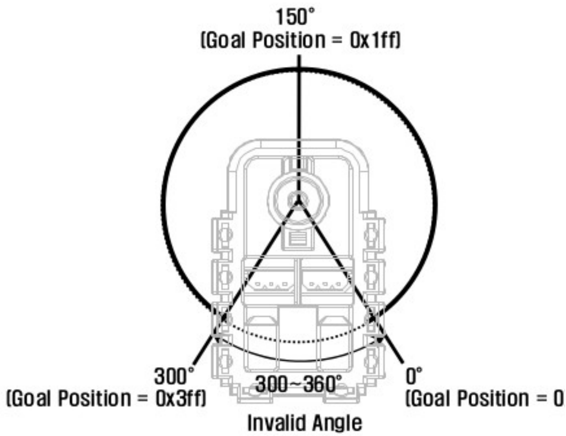
# Conclusions

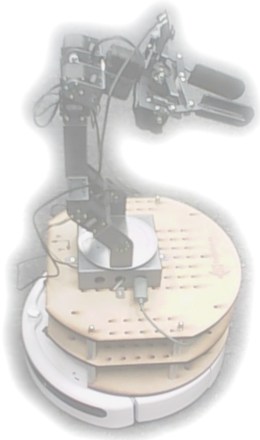




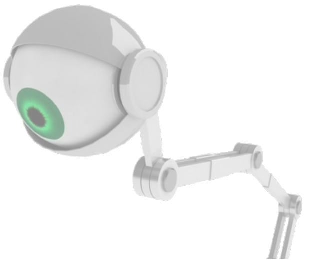
# Limitations

- Most of them are hardware limitations
- Arm has too few degrees of freedom
  - Constrained to simple grasping of spheres
- Arm cannot be controlled by speed
  - No real servoing possible (no control loop)
- Arm shakes too much.
  - Limits robustness from tracking
- Grasper modifies length of end effector
- Servomotors are constrained to 300 degrees of motion
- Detection is color based
- Constrained to one object only

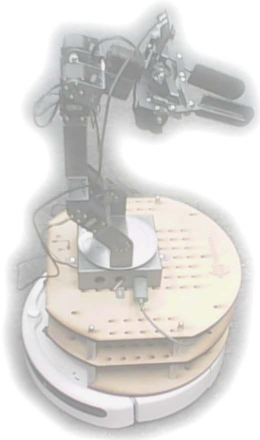




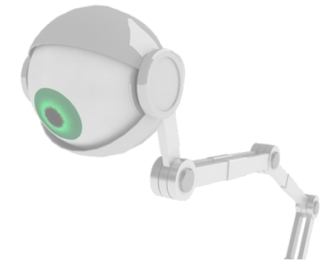
# Future Work



- Add more objects to the scene
  - Object recognition
- Allow for more shapes
- Avoid color for detection
- Use stereo estimation for pose estimation
- Integrate all parts using a nice GUI



# WIKI + Links

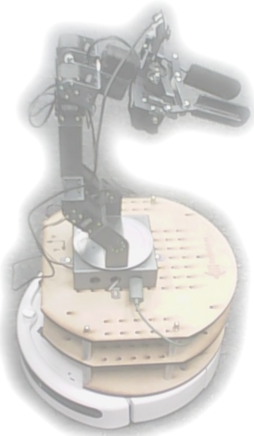


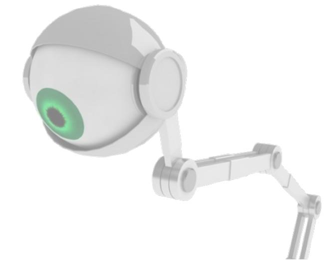
- All work is unified in GitHub

- Link is here (feel free to fork it!):

  https://github.com/FedeCamposeco/Armed-turtlebot

- **We also have a nice installer!!!**





# Questions