

# Practical Course Robotics

Marc Toussaint

April 28, 2016

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Setting up your work environment</b>	<b>1</b>
<b>3</b>	<b>Plan</b>	<b>3</b>
3.1	Milestone 1: Pick-and-place . . . . .	3
3.1.1	Lecture: Basic Motion revisited . . . . .	3
3.1.2	Subproblem: Basic Motion . . . . .	3
3.1.3	Exercise: Develop your own scripting interface . . . . .	4
3.1.4	Subproblem: Segmenting & tracking objects . . . . .	4
3.1.5	Lecture: Basic perception . . . . .	4
3.1.6	Subproblem: Pick & Place . . . . .	4
3.2	Milestone 2: System Identification, Machine Learning & Compliant Optimal Control .	4
3.2.1	Lecture: Dynamics Basics; and motivation . . . . .	5
3.2.2	Directly communicating ROS . . . . .	5
3.2.3	Subproblem: Collect data, formulate model, ML . . . . .	5
3.2.4	Subproblem: Use the model for (extended/unscented) Kalman filtering of the state . . . . .	6
3.2.5	Subproblem: Use the model to translate desired $q$ -accelerations directly to torques	6
3.3	Define your own project! . . . . .	6

## 1 Introduction

## 2 Setting up your work environment

### Preliminaries

- You need a gitlab account; access to `mlr_students`
- Connect to the local mlr-robotlab WIFI

### Install from a fresh Ubuntu

- install fresh Ubuntu 14.04.4 LTS
- google 'ros install indigo'; copy&paste steps; install package `ros-indigo-desktop`
- install packages:  

```
sudo apt-get install synaptic git qtcreator
```

```
sudo apt-get install ros-indigo-ar-track-alvar-msgs ros-indigo-baxter-core-msgs
```

- create ssh key:

```
cd
ssh-keygen
cat .ssh/id_rsa.pub
```

- enter ssh key in gitlab: gitlab start page; profile settings; ssh keys; copy&paste the key (without linebreaks!!!); 'Add key'
- in gitlab go to the project page; see the ssh URL ending with ...git
- checkout our code

```
cd
mkdir git
cd git
git clone <SSH-GIT-URL>
```

- Install the code dependency ubuntu packages:

```
cd ~/git/mlr/install
./INSTALL_ALL_UBUNTU_PACKAGES.sh
```

Trouble shooting: read the README.md in /git/mlr

- configure code and test make:

```
cd ~/git/mlr/share/
git checkout baxter
cp gofMake/config.mk.default gofMake/config.mk
bin/createMakefileLinks.sh
cd src/Ors
make
```

- goto project page and test make

```
cd ~/git/mlr/share/teaching/RoboticsPractical/01-...
make
```

Test starting to run ./x.exe

## Make the baxter move

- setup the WIFI connection to the baxters ros server
 

```
source ~/git/mlr/share/bin/baxterwifisetup
```
- In a project folder, try to run ./x.exe -useRos 1

## Get comfortable

- put all extra documentation useful for others in text files in ./doc
- Use qtcreator. You need to be able to:
  - Create a new 'project' that uses the makefile: 'New Project' -> 'Import Existing Project' -> select the project path (with the makefile)

- Enable and use auto completion and code browsing: add include paths to PROJECT-NAME.includes, especially ../../../../src. Test it with 'right mouse' on symbols
- Know how to use the debugger
- Create a symbolic link `.gdbinit -> git/mlr/tools/qt_mlr_types.py` That will enable pretty printing of mlr data structures in the debugger
- Optionally, import our coding style: Options `-i C++ -i Import... git/mlr/tools/qt_coding_style.xml`
- create own folder groupX, maybe own branch

## 3 Plan

### 3.1 Milestone 1: Pick-and-place

Target: The robot perceives objects on the table (= segment, localize). The robot grasps them and puts them into a bin.

#### 3.1.1 Lecture: Basic Motion revisited

- Task spaces, general problem:
  - A task space is defined by a task map  $\phi : q \mapsto y$
  - In each task space we have a desired behavior (*linear acceleration laws*, 2nd order differential equation)  $\ddot{y}^* = \dots$ . This is usually a PD behavior, optionally with max velocity and acceleration.
  - The desired task behaviors are 'projected down' to  $q$ -space using the operational space control objective. That defines a desired  $\ddot{q}^*$ .
- There is three ways to send this to the robot
  - directly, using the dynamics equation  $u = M\ddot{q} + F$ . But it is computationally not feasible/desirable to have ALL of the above computations in a 1kHz real-time loop
  - A 1st-order Taylor approximation of  $\ddot{q}^* \doteq -K_p q - K_d \dot{q} + \dot{q}_0$ . (We try this). Both of the above are very hard if the dynamics model is imprecise! Later we will test these, with a well learned model from data.
  - Forward simulate  $\ddot{q}^*$  (just integrating the differential equation). That defines a  $q^{\text{ref}}(t)$ . Send this to the existing position controller of the robot.
- Discuss (practical is later): impedance, stiffness
- How is this reflected in the code?
  - `$\phi$ : TaskMap`
  - `$\ddot{y}^* = \dots$ : CtrlTask`
  - Computing  $\ddot{q}^*$ : `TaskController`
  - Sending it to the robot and threading the computation: `TaskControllerModule::step`

#### 3.1.2 Subproblem: Basic Motion

Learn how to use our code to generate targets in various task spaces. Learn how to create `CtrlTasks` directly in C++. Optionally, have a look at the much more abstract RAP interface.

Concretely:

- What are task spaces? Read the `share/doc/taskSpaces.pdf`!

- Make the robot do funny things, like point the hands at each other, etc.
- Think of positioning and orienting the gripper to grasp a box. Define the grasp center, and grasp orientation.

### 3.1.3 Exercise: Develop your own scripting interface

The reason I chose to let you first use the direct interface (directly creating `CtrlTasks`) is:

- This interface is very close to the maths. There is a rather literal relation between the things you define and objectives in the optimization problem.
- This interface is rather transparent: the `CtrlTasks` define the objective, the `TaskController` computes the reference acceleration, this is integrated and send to the robot.

However, using the interface in practice will lead to ugly spaghetti code.

**Exercise:** Develop your own personal abstraction interface; maybe some class that contains methods like `homing()` or `moveToRelative(char* obj, char* relativeObj, const arr& off` or something like that. Think about how to deal with the real time issue (e.g., waiting till convergence or timeout) and whether the interface allows to do multiple things at the same time.

### 3.1.4 Subproblem: Segmenting & tracking objects

Understand how the `tabletop` ROS packages can extract planes (the table) and point cloud clusters on top of the plane. Learn how the objects are imported in our system.

### 3.1.5 Lecture: Basic perception

- The pain of computer vision...
- Keep it simple: point clouds, planes, clusters, markers
- Practical packages

### 3.1.6 Subproblem: Pick & Place

Realize the whole pick-and-place scenario. Core issues are

- Designing the motion tasks
- Sequencing, ideally failure detection & reaction

## 3.2 Milestone 2: System Identification, Machine Learning & Compliant Optimal Control

Target: The robot is controlled on the lowest level, sending direct 'torques' (or alike). Using system identification (ML) we learnt a perfect model of both, the dynamics and the observations. Using Bayesian filtering we can perfectly track the state—giving nice and smooth velocity estimates. The robot 'intelligently' explores its state-space to collect data for the previous tasks.

### 3.2.1 Lecture: Dynamics Basics; and motivation

- Dynamics & optimal control revisited
- Compliance, impedance control, manipulation & teleoperation
- (Do we have F/T sensors?)
- caveats of real robots: 'non-Markovian', sticktion, time lag, gear clearance

### 3.2.2 Directly communicating ROS

**Command line tools** `rostopic list` to list all topics in the network

`rostopic echo <name>` to show what is currently published in a topic. In our case, an interesting topic name is `/robot/limb/left/joint...`

`rostopic ???` to actually send a topic from command line [[Matt, explicitly give an example]]

**From a simple main.cpp** Here is a minimalistic program that sends a command [[Matt: add includes, check that it works]]

```
int main(int argn, char** argv){
    ros::init(argn, argv, "my_node", ros::init_options::NoSigintHandler);

    ros::NodeHandle nh;
    ros::Publisher pub = nh.advertise<baxter_core_msgs::JointCommand>(
        "robot/limb/left/joint_command", 1);

    baxter_core_msgs::JointCommand msg;
    msg.mode = 1;
    msg.command = { 0., 0,0,0,0,0,0};
    msg.names = { "left_s0", "left_s1", "left_e0", "left_e1",
        "left_w0", "left_w1", "left_w2" };

    pub.publish(msg);
    ros::spin();
}
```

**Interferences with our code** In our code, `ros::init` is done by the little `RosInit` object; `ros::spin` is done by a thread called `RosSpinner`. So these you have to drop.

Another issues is that, in our old code, `SendPositionCommandsToBaxter` permanently sends position commands to the baxter (with 100Hz is sends the pose computed by the `TaskControllerModule` and displayed in green). One has to turn this of, e.g., by adding a flag to `SendPositionCommandsToBaxter` on whether to really send.

### 3.2.3 Subproblem: Collect data, formulate model, ML

Think about motion patterns to collect data. Formulate models for the robot dynamics as well as observation model. Apply ML.

- 3.2.4 Subproblem: Use the model for (extended/unscented) Kalman filtering of the state
- 3.2.5 Subproblem: Use the model to translate desired  $q$ -accelerations directly to torques
- 3.3 Define your own project!