# The Duckietown Book (Duckiebook)

From kits of parts to an autonomous fleet
in 857 easy steps
without hiding anything

The last version of this "duckiebook" and other related documents are available at the URL
http://book.duckietown.org/

For searching, see the one-page version.

# Table of contents


**Part A - The Duckietown project** ... 29
**Unit A-1 - What is Duckietown?** ... 30
Section 1.1 - Goals and objectives ... 30
Section 1.2 - Learn about the Duckietown educational experience ... 30
Section 1.3 - Learn about the platform ... 31
**Unit A-2 - Duckietown history and future** ... 32
Section 2.1 - The beginnings of Duckietown ... 32
Section 2.2 - University-level classes in 2016 ... 33
Section 2.3 - University-level classes in 2017 ... 33
Section 2.4 - Chile ... 34
Section 2.5 - Duckietown High School ... 34
**Unit A-3 - Duckietown classes** ... 37
Section 3.1 - 2016 ... 37
Section 3.2 - 2017 ... 38
**Unit A-4 - First steps** ... 40
Section 4.1 - How to get started ... 40
Section 4.2 - Duckietown for instructors ... 40
Section 4.3 - Duckietown for self-guided learners ... 40
Section 4.4 - Introduction for companies ... 40
Section 4.5 - How to keep in touch ... 40
Section 4.6 - How to contribute ... 40
Section 4.7 - Frequently Asked Questions ... 40

**Part B - Duckumentation documentation** ... 42
**Unit B-1 - Contributing to the documentation** ... 43
Section 1.1 - Where the documentation is ... 43
Section 1.2 - Editing links ... 43
Section 1.3 - Installing the documentation system ... 43
Section 1.4 - Compiling the documentation (updated Sep 12) ... 45
Section 1.5 - The workflow to edit documentation (updated Sep 12) ... 46
Section 1.6 - Reporting problems ... 46
**Unit B-2 - Basic Markduck guide** ... 47
Section 2.1 - Markdown ... 47
Section 2.2 - Variables in command lines and command output ... 47
Section 2.3 - Character escapes ... 47
Section 2.4 - Keyboard keys ... 47
Section 2.5 - Figures ... 48
Section 2.6 - Subfigures ... 49
Section 2.7 - Shortcut for tables ... 50
Section 2.8 - Linking to documentation ... 50
**Unit B-3 - Special paragraphs and environments** ... 53
Section 3.1 - Special paragraphs tags ... 53
Section 3.2 - Other `div` environments ... 56
Section 3.3 - Notes and questions ... 57
**Unit B-4 - Using LaTeX constructs in documentation** ... 59
Section 4.1 - Embedded LaTeX ... 59
Section 4.2 - LaTeX equations ... 61
Section 4.3 - LaTeX symbols ... 61
Section 4.4 - Bibliography support ... 61
Section 4.5 - Embedding Latex in Figures through SVG ... 62
**Unit B-5 - Advanced Markduck guide** ... 64
Section 5.1 - Embedding videos ... 64
Section 5.2 - `move-here` tag ... 64

Section 5.3 - Comments ... 65
Section 5.4 - Referring to Github files ... 65
Section 5.5 - Putting code from the repository in line ... 66
**Unit B-6 - *Compiling the PDF version** ... 68
Section 6.1 - Installing `nodejs` ... 68
Section 6.2 - Installing Prince ... 68
Section 6.3 - Installing fonts ... 68
Section 6.4 - Compiling the PDF ... 69
**Unit B-7 - Markduck troubleshooting** ... 70
Section 7.1 - Changes don't appear on the website ... 70
Section 7.2 - Troubleshooting errors in the compilation process ... 70
Section 7.3 - Common mistakes with Markdown ... 70
**Unit B-8 - The Duckuments bot** ... 72
Section 8.1 - Documentation deployment ... 72
Section 8.2 - Understand what's going on ... 72
Section 8.3 - Logging ... 72
Section 8.4 - Debugging Github Pages problems ... 73
**Unit B-9 - Documentation style guide** ... 74
Section 9.1 - General guidelines for technical writing ... 74
Section 9.2 - Style guide for the Duckietown documentation ... 74
Section 9.3 - Writing command lines ... 75
Section 9.4 - Frequently misspelled words ... 75
Section 9.5 - Other conventions ... 75
Section 9.6 - Troubleshooting sections ... 76
**Unit B-10 - Learning in Duckietown** ... 77
Section 10.1 - The learning feedback loop ... 77
Section 10.2 - Knowledge, Skills and Competences ... 78
**Unit B-11 - Knowledge graph** ... 81
Section 11.1 - Formalization ... 81
Section 11.2 - Atoms properties ... 82
Section 11.3 - Markdown format for text-like atoms ... 82
Section 11.4 - How to describe the semantic graphs of atoms ... 83
Section 11.5 - How to describe modules ... 84
**Unit B-12 - Translations** ... 85
Section 12.1 - File organization ... 85
Section 12.2 - Guidelines for English writers ... 85
Section 12.3 - File format ... 85

**Part C - Operation manual - Duckiebot** ... 87
**Unit C-1 - Duckiebot configurations** ... 88
Section 1.1 - Duckiebot Configurations: Fall 2017 ... 88
Section 1.2 - Configuration functionality ... 88
Section 1.3 - Branch configuration releases: Fall 2017 ... 91
**Unit C-2 - Acquiring the parts (`DB17-jwd`)** ... 93
Section 2.1 - Bill of materials ... 93
Section 2.2 - Chassis ... 94
Section 2.3 - Raspberry Pi 3 - Model B ... 94
Section 2.4 - Camera ... 96
Section 2.5 - DC Motor HAT ... 97
Section 2.6 - Battery ... 98
Section 2.7 - Standoffs, Nuts and Screws ... 98
Section 2.8 - Zip Tie ... 99
Section 2.9 - Configuration `DB17-w` ... 99
Section 2.10 - Configuration `DB17-j` ... 100
Section 2.11 - Configuration `DB17-d` ... 100
**Unit C-3 - Soldering boards (`DB17`)** ... 102
Section 3.1 - General tips ... 102
**Unit C-4 - Preparing the power cable (`DB17`)** ... 105
Section 4.1 - Video tutorial ... 105
Section 4.2 - Step-by-step guide ... 105

**Unit C-5 - Assembling the Duckiebot (`DB17-jwd`)** ... 110
Section 5.1 - Chassis ... 110
Section 5.2 - Assembling the Raspberry Pi, camera, and HATs ... 118
Section 5.3 - FAQ ... 127
**Unit C-6 - Assembling the Duckiebot (`DB17-wjd` TTIC)** ... 129
Section 6.1 - Motors ... 129
Section 6.2 - Wheels ... 132
Section 6.3 - Omni-directional wheel ... 134
Section 6.4 - Chassis standoffs ... 136
Section 6.5 - Camera kit ... 138
Section 6.6 - Heat sinks ... 141
Section 6.7 - Raspberry Pi 3 ... 142
Section 6.8 - Top plate ... 145
Section 6.9 - USB Power cable ... 147
Section 6.10 - DC Stepper Motor HAT ... 147
Section 6.11 - Battery ... 151
Section 6.12 - Upgrade to `DB17-w` ... 153
Section 6.13 - Upgrade to `DB17-j` ... 154
Section 6.14 - Upgrade to `DB17-d` ... 156
Section 6.15 - FAQ ... 157
**Unit C-7 - Reproducing the image** ... 159
Section 7.1 - Download and uncompress the Ubuntu Mate image ... 159
Section 7.2 - Burn the image to an SD card ... 159
Section 7.3 - Raspberry Pi Config ... 160
Section 7.4 - Install packages ... 160
Section 7.5 - Install Edimax driver ... 161
Section 7.6 - Install ROS ... 162
Section 7.7 - Wireless configuration (old version) ... 162
Section 7.8 - Wireless configuration ... 163
Section 7.9 - SSH server config ... 165
Section 7.10 - Create swap Space ... 165
Section 7.11 - Passwordless sudo ... 166
Section 7.12 - Clean up ... 166
Section 7.13 - Ubuntu user configuration ... 167
Section 7.14 - Check that all required packages were installed ... 168
Section 7.15 - Creating the image ... 168
Section 7.16 - TODO: Git LFS ... 169
**Unit C-8 - Installing Ubuntu on laptops** ... 170
Section 8.1 - Install Ubuntu ... 170
Section 8.2 - Install useful software ... 170
Section 8.3 - Install ROS ... 171
Section 8.4 - Other suggested software ... 171
Section 8.5 - Passwordless `sudo` ... 171
Section 8.6 - SSH and Git setup ... 171
Section 8.7 - Installation of the duckuments system ... 172
**Unit C-9 - Duckiebot Initialization** ... 173
Section 9.1 - Acquire and burn the image ... 173
Section 9.2 - Turn on the Duckiebot ... 174
Section 9.3 - Connect the Duckiebot to a network ... 174
Section 9.4 - Ping the Duckiebot ... 174
Section 9.5 - SSH to the Duckiebot ... 175
Section 9.6 - Setup network ... 175
Section 9.7 - Update the system ... 175
Section 9.8 - Give a name to the Duckiebot ... 175
Section 9.9 - Change the hostname ... 175
Section 9.10 - Expand your filesystem ... 176
Section 9.11 - Create your user ... 177
Section 9.12 - Other customizations ... 179
Section 9.13 - Hardware check: camera ... 179
Section 9.14 - Final touches: duckie logo ... 180

**Unit C-10 - Networking aka the hardest part** ... 182
Section 10.1 - (For `DB17-w`) Configure the robot-generated network ... 182
Section 10.2 - Setting up wireless network configuration ... 183
**Unit C-11 - Software setup and RC remote control** ... 190
Section 11.1 - Clone the Duckietown repository ... 190
Section 11.2 - Update the system ... 190
Section 11.3 - Set up the ROS environment on the Duckiebot ... 191
Section 11.4 - Clone the duckiefleet repository ... 191
Section 11.5 - Add your vehicle data to the robot database ... 191
Section 11.6 - Test that the joystick is detected ... 192
Section 11.7 - Run the joystick demo ... 192
Section 11.8 - The proper shutdown procedure for the Raspberry Pi ... 193
**Unit C-12 - Reading from the camera** ... 195
Section 12.1 - Check the camera hardware ... 195
Section 12.2 - Create two windows ... 195
Section 12.3 - First window: launch the camera nodes ... 195
Section 12.4 - Second window: view published topics ... 196
**Unit C-13 - RC control launched remotely** ... 198
Section 13.1 - Two ways to launch a program ... 198
Section 13.2 - Download and setup `Software` repository on the laptop ... 198
Section 13.3 - Rebuild the machines files ... 198
Section 13.4 - Start the demo ... 198
Section 13.5 - Watch the program output using `rqt_console` ... 199
Section 13.6 - Troubleshooting ... 199
**Unit C-14 - RC+camera remotely** ... 201
Section 14.1 - Assumptions ... 201
Section 14.2 - Terminal setup ... 201
Section 14.3 - First window: launch the joystick demo ... 201
Section 14.4 - Second window: launch the camera nodes ... 202
Section 14.5 - Third window: view data flow ... 202
Section 14.6 - Fourth window: visualize the image using `rviz` ... 202
Section 14.7 - Proper shutdown procedure ... 203
**Unit C-15 - Interlude: Ergonomics** ... 204
Section 15.1 - SSH aliases ... 204
Section 15.2 - `set_ros_master.sh` ... 205
**Unit C-16 - Wheel calibration** ... 206
Section 16.1 - Introduction ... 206
Section 16.2 - What is the Calibration step? ... 206
Section 16.3 - Perform the Calibration ... 207
**Unit C-17 - Camera calibration** ... 213
Section 17.1 - Intrinsic calibration ... 213
Section 17.2 - Extrinsic calibration ... 216
**Unit C-18 - Updated camera calibration and validation** ... 220
Section 18.1 - Check out the experimental branch ... 220
Section 18.2 - Place the robot on the pattern ... 220
Section 18.3 - Extrinsic calibration procedure ... 221
Section 18.4 - Camera validation by simulation ... 222
Section 18.5 - Camera validation by running one-shot localization ... 223
Section 18.6 - The importance of validation ... 224
**Unit C-19 - Taking and verifying a log** ... 226
Section 19.1 - Preparation ... 226
Section 19.2 - Run something on the Duckiebot ... 226
Section 19.3 - View images on the laptop ... 226
Section 19.4 - Record the log ... 226
Section 19.5 - Verify a log ... 227
**Unit C-20 - Troubleshooting** ... 228
Section 20.1 - The Raspberry Pi does not turn ON ... 228
Section 20.2 - I cannot access my Duckiebot via SSH ... 228
Section 20.3 - The Duckiebot does not move ... 229

**Part D - Operation manual - Duckietown** ... 231
**Unit D-1 - Duckietown parts** ... 232
Section 1.1 - Bill of materials ... 232
Section 1.2 - Duckies ... 232
Section 1.3 - Floor Mats ... 233
Section 1.4 - Duck Tape ... 233
Section 1.5 - Traffic Signs ... 234
Section 1.6 - Traffic lights Parts ... 234
**Unit D-2 - Duckietown Appearance Specification** ... 236
Section 2.1 - Version history ... 236
Section 2.2 - Overview ... 236
Section 2.3 - Layer 1 - The Tile Layer ... 236
Section 2.4 - Layer 2 - Signage and Lights ... 242
Section 2.5 - Traffic Signs ... 242
Section 2.6 - Street Name Signs ... 244
Section 2.7 - Traffic Lights ... 245
**Unit D-3 - Signage** ... 246
Section 3.1 - Build a map ... 246
Section 3.2 - Making New Signage ... 246
Section 3.3 - Assembly ... 247
Section 3.4 - Placement ... 247
**Unit D-4 - Duckietown Assembly** ... 248
**Unit D-5 - Traffic lights Assembly** ... 249
**Unit D-6 - Semantics of LEDS** ... 250

**Part E - Duckiebot - DB17-1c configurations** ... 251
**Unit E-1 - Acquiring the parts (DB17-1c)** ... 252
Section 1.1 - Bill of materials ... 252
Section 1.2 - LEDs ... 253
Section 1.3 - Bumpers ... 255
Section 1.4 - Headers, resistors and jumper ... 255
Section 1.5 - Caster (DB17-c) ... 256
**Unit E-2 - Soldering boards (DB17-1)** ... 258
Section 2.1 - General rules ... 258
Section 2.2 - 16-channel PWM/Servo HAT ... 258
Section 2.3 - LSD board ... 259
Section 2.4 - LED connection ... 260
Section 2.5 - Putting everything together! ... 261
**Unit E-3 - Assembling the Duckiebot (DB17-1c)** ... 262
Section 3.1 - Assembly the Servo/PWM hat (DB17-11) ... 262
Section 3.2 - Assembling the Bumper Set (DB17-12) ... 264
Section 3.3 - Assembling the LED HAT and LEDs (DB17-13) ... 264
**Unit E-4 - Bumper Assembly** ... 265
Section 4.1 - Locate all required parts ... 265
Section 4.2 - Remove protective paper ... 266
Section 4.3 - Assembly Rear Spacers ... 266
Section 4.4 - Mount Front Bumper ... 271
**Unit E-5 - DB17-1 setup** ... 274
Section 5.1 - Locate all required parts ... 274
Section 5.2 - Connecting Wires to LEDs ... 274
Section 5.3 - Connecting LEDs to LSD Hat ... 276
Section 5.4 - Running the Wires Through the Chassis ... 277
Section 5.5 - Final tweaks ... 278

**Part F - Operation Manual - demos** ... 279
**Unit F-1 - Demo template** ... 280
Section 1.1 - Video of expected results ... 280
Section 1.2 - Duckietown setup notes ... 280
Section 1.3 - Duckiebot setup notes ... 280

Section 1.4 - Pre-flight checklist ..... 280
Section 1.5 - Demo instructions ..... 280
Section 1.6 - Troubleshooting ..... 281
Section 1.7 - Demo failure demonstration ..... 281
**Unit F-2 - Lane following** ..... 282
Section 2.1 - Video of expected results ..... 282
Section 2.2 - Duckietown setup notes ..... 282
Section 2.3 - Duckiebot setup notes ..... 282
Section 2.4 - Pre-flight checklist ..... 282
Section 2.5 - Demo instructions ..... 282
Section 2.6 - Troubleshooting ..... 283
Section 2.7 - Demo failure demonstration ..... 284
**Unit F-3 - Demo Saviors** ..... 285
Section 3.1 - Video of expected results ..... 285
Section 3.2 - Duckietown setup notes ..... 285
Section 3.3 - Duckiebot setup notes ..... 285
Section 3.4 - Pre-flight checklist ..... 285
Section 3.5 - Demo instructions ..... 286
Section 3.6 - Troubleshooting ..... 287
Section 3.7 - Further Reading ..... 287
**Unit F-4 - Intersection Navigation** ..... 288
**Unit F-5 - Indefinite Navigation** ..... 289
Section 5.1 - Video of expected results ..... 289
Section 5.2 - Duckietown setup notes ..... 289
Section 5.3 - Duckiebot setup notes ..... 289
Section 5.4 - Pre-flight checklist ..... 289
Section 5.5 - Demo instructions ..... 290
Section 5.6 - Troubleshooting ..... 291
**Unit F-6 - Coordination** ..... 292
Section 6.1 - Duckietown setup notes ..... 292
Section 6.2 - Duckiebot setup notes ..... 292
Section 6.3 - Demo instructions ..... 292
Section 6.4 - Troubleshooting ..... 293
**Unit F-7 - Implicit Coordination** ..... 295
Section 7.1 - Video of expected results ..... 295
Section 7.2 - Duckietown setup notes ..... 295
Section 7.3 - Duckiebot setup notes ..... 295
Section 7.4 - Laptop setup notes ..... 295
Section 7.5 - Pre-flight checklist ..... 295
Section 7.6 - Demo instructions ..... 296
Section 7.7 - Troubleshooting ..... 296
Section 7.8 - Demo failure demonstration ..... 296
**Unit F-8 - Explicit Coordination** ..... 297
Section 8.1 - Duckietown setup notes ..... 297
Section 8.2 - Duckiebot setup notes ..... 297
Section 8.3 - Demo instructions ..... 297
Section 8.4 - Troubleshooting ..... 298
**Unit F-9 - Parallel Autonomy** ..... 300
**Unit F-10 - Follow Leader** ..... 301
Section 10.1 - Video of expected results ..... 301
Section 10.2 - Duckietown setup notes ..... 301
Section 10.3 - Duckiebot setup notes ..... 301
Section 10.4 - Pre-flight checklist ..... 301
Section 10.5 - Demo instructions ..... 301
Section 10.6 - Troubleshooting ..... 302
Section 10.7 - Demo failure demonstration ..... 302

**Part G - Preliminaries** ..... 303
**Unit G-1 - Chapter template** ..... 304
Section 1.1 - Example Title: PID control ..... 304

Section 1.2 - Problem Definition ... 304
Section 1.3 - Introduced Notions ... 305
Section 1.4 - Examples ... 306
Section 1.5 - Pointers to Exercises ... 307
Section 1.6 - Conclusions ... 307
Section 1.7 - Next Steps ... 307
Section 1.8 - References ... 307
**Unit G-2 - Symbols and conventions** ... 308
Section 2.1 - Conventions ... 308
Section 2.2 - Spaces ... 309
**Unit G-3 - Sets** ... 310
Section 3.1 - Definition ... 310
Section 3.2 - Maps ... 310
Section 3.3 - Definition ... 310
Section 3.4 - Properties of maps ... 310
**Unit G-4 - Numbers** ... 311
Section 4.1 - Natural numbers ... 311
Section 4.2 - Integers ... 311
Section 4.3 - Rationals ... 311
Section 4.4 - Irrationals ... 312
Section 4.5 - Reals ... 312
Section 4.6 - Complex ... 312
**Unit G-5 - Complex numbers** ... 313
Section 5.1 - Powers of $i$ ... 313
Section 5.2 - Complex conjugate ... 313
**Unit G-6 - Linearity and Vectors** ... 314
Section 6.1 - Linearity ... 314
Section 6.2 - Vectors ... 315
Section 6.3 - Linear dependance ... 320
Section 6.4 - Pointers to Exercises ... 320
Section 6.5 - Conclusions ... 320
**Unit G-7 - Matrices basics** ... 322
Section 7.1 - Matrix dimensions ... 322
Section 7.2 - Matrix diagonals ... 323
Section 7.3 - Diagonal matrix ... 323
Section 7.4 - Identity matrix ... 323
Section 7.5 - Null matrix ... 323
Section 7.6 - Determinant ... 323
Section 7.7 - Rank of a matrix ... 323
Section 7.8 - Trace of a matrix ... 324
**Unit G-8 - Matrix inversions** ... 325
Section 8.1 - Adjugate matrix ... 325
Section 8.2 - Matrix Inverse ... 325
Section 8.3 - Nonsingularity of a matrix ... 325
**Unit G-9 - Matrices and vectors** ... 326
Section 9.1 - Matrix as representation of linear (vector) spaces ... 326
**Unit G-10 - Matrix operations (basic)** ... 327
**Unit G-11 - Matrix operations (complex)** ... 328
**Unit G-12 - Matrix diagonalization** ... 329
Section 12.1 - Left and Right Inverse (topic for advanced-linear-algebra?) ... 329
**Unit G-13 - Norms** ... 330
Section 13.1 - Definition ... 330
Section 13.2 - Properties ... 331
**Unit G-14 - From ODEs to LTI systems** ... 332
Section 14.1 - LTI Systems ... 332
**Unit G-15 - Linearization** ... 333
Section 15.1 - Linearization of a nonlinear system ... 333
**Unit G-16 - Probability basics** ... 334
Section 16.1 - Random Variables ... 334
**Unit G-17 - Kinematics** ... 341

**Unit G-18 - Dynamics** ... 342
**Unit G-19 - Coordinate systems** ... 343
Section 19.1 - Motivation ... 343
Section 19.2 - Definitions ... 343
Section 19.3 - Examples ... 343
Section 19.4 - Further reading ... 347
**Unit G-20 - Reference frames** ... 348
Section 20.1 - Motivation ... 348
Section 20.2 - Definition ... 348
Section 20.3 - Example ... 348
Section 20.4 - Translating motions in one frame to another ... 348
**Unit G-21 - Time series** ... 350
Section 21.1 - Definition of time series ... 350
Section 21.2 - Operations on time series ... 350
Section 21.3 - Further reading ... 350
**Unit G-22 - Transformations** ... 351
Section 22.1 - Introduction ... 351
Section 22.2 - Definitions and important examples ... 351
Section 22.3 - Applications ... 351
**Unit G-23 - Types** ... 352
Section 23.1 - Types vs sets ... 352
Section 23.2 - Example: product types ... 352
Section 23.3 - In Duckietown ... 352
Section 23.4 - Historical notes ... 352
Section 23.5 - Further reading ... 353
**Unit G-24 - Computer science concepts** ... 354
Section 24.1 - Real-time Operating system ... 354

**Part H - A course in autonomy** ... 355
**Unit H-1 - Autonomous Vehicles** ... 356
Section 1.1 - Autonomous Vehicles in the News ... 356
Section 1.2 - Levels of Autonomy ... 356
**Unit H-2 - Autonomy overview** ... 358
Section 2.1 - Basic Building Blocks of Autonomy ... 358
Section 2.2 - Advanced Building Blocks of Autonomy ... 363
**Unit H-3 - Modern Robotic Systems** ... 365
Section 3.1 - No robot is an island ... 365
Section 3.2 - Development of modern robotic systems ... 365
Section 3.3 - Example of a typical data processing pipeline ... 366
Section 3.4 - Vignettes from an optimistic future ... 367
Section 3.5 - Take-away points ... 367
Section 3.6 - Further reading ... 367
**Unit H-4 - System architectures basics** ... 369
Section 4.1 - Logical and physical architecture ... 369
Section 4.2 - Logical architecture ... 369
Section 4.3 - Actor model ... 369
Section 4.4 - Physical architecture ... 369
Section 4.5 - Mapping logical architecture onto physical ... 370
Section 4.6 - Example in Duckietown ... 370
Section 4.7 - Take-away points ... 370
Section 4.8 - Further reading ... 370
**Unit H-5 - Autonomy architectures** ... 372
Section 5.1 - The state of the art in autonomy architectures ... 372
Section 5.2 - Further reading ... 372
Section 5.3 - Figures ... 372
**Unit H-6 - Representations** ... 375
Section 6.1 - Robot Representations ... 376
Section 6.2 - Environment Representations ... 377
**Unit H-7 - Modern signal processing** ... 379
Section 7.1 - Event-based processing ... 379

Section 7.2 - Periodic vs event-based processing ... 379
Section 7.3 - Why working with events ... 379
Section 7.4 - Definition of message statistics ... 380
Section 7.5 - On the independence of latency and frequency ... 380
Section 7.6 - Design considerations and trade-offs ... 380
Section 7.7 - Further reading ... 381
**Unit H-8 - Middleware** ... 382
Section 8.1 - Why using middleware ... 382
Section 8.2 - What middleware provides ... 382
Section 8.3 - A few alternatives of robotics middleware ... 382
Section 8.4 - Trade-offs and design considerations ... 382
**Unit H-9 - Modularization and Contracts** ... 384
Section 9.1 - Monolithic vs modular design ... 384
Section 9.2 - Contracts ... 384
Section 9.3 - Level 0: API (Types and signatures) ... 384
Section 9.4 - Level 1: Timing: latency, frequency, availability ... 384
Section 9.5 - Level 2: Resources: Computation and memory ... 384
Section 9.6 - Level 3: Semantics ... 384
Section 9.7 - Level 4: Quality ... 384
**Unit H-10 - Configuration management** ... 386
Section 10.1 - Motivation ... 386
Section 10.2 - Stages of configuration management: from clueless to pro ... 386
Section 10.3 - The primordial stage (clueless developer) ... 386
Section 10.4 - Recognition that parameters are important ... 386
Section 10.5 - Parameters in interfaces ... 387
Section 10.6 - Encapsulation of functionality objects ... 387
Section 10.7 - Convenient interfaces ... 388
Section 10.8 - Abstracting the configuration ... 388
Section 10.9 - Duckietown solution ... 389
Section 10.10 - Beyong: Customization, History tracking ... 389
**Unit H-11 - Duckiebot modeling** ... 390
Section 11.1 - Preliminaries ... 390
Section 11.2 - Dynamics ... 392
Section 11.3 - Kinematics ... 395
Section 11.4 - Differential drive robot kinematic model ... 396
Section 11.5 - Simplified dynamic model ... 397
Section 11.6 - DC motor dynamic model ... 397
Section 11.7 - Conclusions ... 398
**Unit H-12 - Odometry Calibration** ... 400
**Unit H-13 - Computer vision basics** ... 401
**Unit H-14 - Camera geometry** ... 402
**Unit H-15 - Camera calibration** ... 403
**Unit H-16 - Image filtering** ... 404
**Unit H-17 - Illumination invariance** ... 405
**Unit H-18 - Line Detection** ... 406
**Unit H-19 - Feature extraction** ... 407
**Unit H-20 - Place recognition** ... 408
**Unit H-21 - Filtering 1** ... 409
**Unit H-22 - Filtering 2** ... 410
**Unit H-23 - Mission planning** ... 411
**Unit H-24 - Planning in discrete domains** ... 412
**Unit H-25 - Motion planning** ... 413
**Unit H-26 - RRT** ... 414
**Unit H-27 - Feedback control** ... 415
**Unit H-28 - PID Control** ... 416
**Unit H-29 - MPC Control** ... 417
**Unit H-30 - Object detection** ... 418
**Unit H-31 - Object classification** ... 419
**Unit H-32 - Object tracking** ... 420
**Unit H-33 - Reacting to obstacles** ... 421

Unit H-34 - Semantic segmentation ..... 422
Unit H-35 - Text recognition ..... 423
Unit H-36 - SLAM - Problem formulation ..... 424
Unit H-37 - SLAM - Broad categories ..... 425
Unit H-38 - VINS ..... 426
Unit H-39 - Advanced place recognition ..... 427
Unit H-40 - Fleet level planning (placeholder) ..... 428
Unit H-41 - Fleet level planning (placeholder) ..... 429
Unit H-42 - Bibliography ..... 430

Part I - Exercises ..... 432
Unit I-1 - Exercise: Basic image operations ..... 433
Section 1.1 - Skills learned ..... 433
Section 1.2 - Instructions ..... 433
Section 1.3 - Specification of `dt-image-flip0` ..... 433
Section 1.4 - Useful APIs ..... 433
Unit I-2 - Exercise: Basic image operations, adult version ..... 435
Section 2.1 - Skills learned ..... 435
Section 2.2 - Instructions ..... 435
Section 2.3 - `dt-image-flip` specification ..... 435
Section 2.4 - Useful APIs ..... 435
Section 2.5 - Testing it works with `image-ops-tester` ..... 436
Section 2.6 - Bottom line ..... 436
Unit I-3 - Exercise: Simple data analysis from a bag ..... 437
Section 3.1 - Skills learned ..... 437
Section 3.2 - Instructions ..... 437
Section 3.3 - Specification for `dt-bag-analyze` ..... 437
Section 3.4 - Useful APIs ..... 437
Section 3.5 - Test that it works ..... 438
Unit I-4 - Exercise: Bag in, bag out ..... 439
Section 4.1 - Skills learned ..... 439
Section 4.2 - Instructions ..... 439
Section 4.3 - Specification of `dt-bag-decimate` ..... 439
Section 4.4 - Useful new APIs ..... 439
Section 4.5 - Check that it works ..... 439
Unit I-5 - Exercise: Bag thumbnails ..... 441
Section 5.1 - Skills learned ..... 441
Section 5.2 - Instructions ..... 441
Section 5.3 - Specification for `dt-bag-thumbnails` ..... 441
Section 5.4 - Test data ..... 441
Section 5.5 - Useful APIs ..... 441
Unit I-6 - Exercise: Instagram filters ..... 443
Section 6.1 - Skills learned ..... 443
Section 6.2 - Instructions ..... 443
Section 6.3 - Specification for `dt-instagram` ..... 443
Section 6.4 - Useful new APIs ..... 444
Unit I-7 - Exercise: Bag instagram ..... 445
Section 7.1 - Instructions ..... 445
Section 7.2 - Specification for `dt-bag-instagram` ..... 445
Section 7.3 - Test data ..... 445
Section 7.4 - Useful new APIs ..... 445
Section 7.5 - Check that it works ..... 445
Unit I-8 - Exercise: Live Instagram ..... 447
Section 8.1 - Skills learned ..... 447
Section 8.2 - Instructions ..... 447
Section 8.3 - Specification for the node `dt_live_instagram_`*robot name*`_node` ..... 447
Section 8.4 - Check that it works ..... 447
Unit I-9 - Exercise: Augmented Reality ..... 448
Section 9.1 - Skills learned ..... 448
Section 9.2 - Introduction ..... 448

Section 9.3 - Instructions ... 448
Section 9.4 - Specification of `dt_augmented_reality` ... 449
Section 9.5 - Specification of the map ... 449
Section 9.6 - "Map" files ... 450
Section 9.7 - Suggestions ... 454
Section 9.8 - Useful APIs ... 454
**Unit I-10 - Exercise: Lane Filtering - Particle Filter** ... 456
Section 10.1 - Skills learned ... 456
Section 10.2 - Introduction ... 456
Section 10.3 - Instructions ... 456
Section 10.4 - Submission ... 456
**Unit I-11 - Exercise: Lane Filtering - Extended Kalman Filter** ... 457
Section 11.1 - Skills learned ... 457
Section 11.2 - Introduction ... 457
Section 11.3 - Instructions ... 457
Section 11.4 - Submission ... 457
**Unit I-12 - Exercise: Git and conventions** ... 458
**Unit I-13 - Exercise: Use our API for arguments** ... 459
Section 13.1 - Skills learned ... 459
Section 13.2 - Instructions ... 459
Section 13.3 - Useful new API learned ... 459
**Unit I-14 - Exercise: Bouncing ball** ... 460
Section 14.1 - Skills learned ... 460
Section 14.2 - Instructions ... 460
Section 14.3 - Useful new API learned ... 460
**Unit I-15 - Exercise: Visualizing data on image** ... 461
Section 15.1 - Skills learned ... 461
Section 15.2 - Instruction ... 461
Section 15.3 - Specification for `bag-mark-spots` ... 461
**Unit I-16 - Exercise: Make that into a node** ... 462
Section 16.1 - Learned skills ... 462
Section 16.2 - Instructions ... 462
**Unit I-17 - Exercise: Instagram with EasyAlgo interface** ... 463
Section 17.1 - Learned skills ... 463
Section 17.2 - Instructions ... 463
Section 17.3 - See documentation ... 463
Section 17.4 - Use in your code ... 463
**Unit I-18 - Milestone: Illumination invariance (anti-instagram)** ... 464
**Unit I-19 - Exercise: Launch files basics** ... 465
Section 19.1 - Learned skills ... 465
**Unit I-20 - Exercise: Unit tests** ... 466
Section 20.1 - Learned skills ... 466
Section 20.2 - Unit tests with nosetests ... 466
Section 20.3 - Unite tests with ROS tests ... 466
Section 20.4 - Unite tests with comptests ... 466
**Unit I-21 - Exercise: Parameters (standard ROS api)** ... 467
Section 21.1 - Learned skills ... 467
**Unit I-22 - Exercise: Parameters (our API)** ... 468
Section 22.1 - Learned skills ... 468
**Unit I-23 - Exercise: Place recognition abstraction** ... 469
Section 23.1 - Learned skills ... 469
Section 23.2 - Instructions ... 469
Section 23.3 - Try a basic feature: ... 469
Section 23.4 - Bottom line ... 469
**Unit I-24 - Exercise: Parallel processing** ... 470
Section 24.1 - Learned skills ... 470
Section 24.2 - Instructions ... 470
**Unit I-25 - Exercise: Adding new test to integration tests** ... 471
Section 25.1 - Learned skills ... 471
Section 25.2 - Instructions ... 471

Section 25.3 - Bottom line ... 471
Section 25.4 - Milestone: Lane following ... 471
**Unit I-26 - Exercise: localization** ... 472
**Unit I-27 - Exercise: Ducks in a row** ... 473
**Unit I-28 - Exercise: Comparison of PID** ... 474
**Unit I-29 - Exercise: RRT** ... 475
**Unit I-30 - Exercise: Who watches the watchmen? (optional)** ... 476
Section 30.1 - Skills learned ... 476
Section 30.2 - Instructions ... 476
Section 30.3 - `image-ops-tester-tester` specification ... 476
Section 30.4 - Testing it works with `image-ops-tester-tester-tester` ... 476
Section 30.5 - Bottom line ... 476

**Part J - Software reference** ... 477
**Unit J-1 - Ubuntu packaging with APT** ... 478
Section 1.1 - `apt install` ... 478
Section 1.2 - `apt update` ... 478
Section 1.3 - `apt-key` ... 478
Section 1.4 - `apt-mark` ... 478
Section 1.5 - `apt-get` ... 478
Section 1.6 - `add-apt-repository` ... 478
Section 1.7 - `wajig` ... 478
Section 1.8 - `dpigs` ... 478
**Unit J-2 - GNU/Linux general notions** ... 479
Section 2.1 - Background reading ... 479
**Unit J-3 - Every day Linux** ... 480
Section 3.1 - `man` ... 480
Section 3.2 - `cd` ... 480
Section 3.3 - `sudo` ... 480
Section 3.4 - `ls` ... 480
Section 3.5 - `cp` ... 480
Section 3.6 - `mkdir` ... 480
Section 3.7 - `touch` ... 480
Section 3.8 - `reboot` ... 481
Section 3.9 - `shutdown` ... 481
Section 3.10 - `rm` ... 481
**Unit J-4 - Users** ... 482
Section 4.1 - `passwd` ... 482
**Unit J-5 - UNIX tools** ... 483
Section 5.1 - `cat` ... 483
Section 5.2 - `tee` ... 483
Section 5.3 - `truncate` ... 483
**Unit J-6 - Linux disks and files** ... 484
Section 6.1 - `fdisk` ... 484
Section 6.2 - `mount` ... 484
Section 6.3 - `umount` ... 484
Section 6.4 - `losetup` ... 484
Section 6.5 - `gparted` ... 484
Section 6.6 - `dd` ... 484
Section 6.7 - `sync` ... 484
Section 6.8 - `df` ... 484
Section 6.9 - How to make a partition ... 484
**Unit J-7 - Other administration commands** ... 485
Section 7.1 - `visudo` ... 485
Section 7.2 - `update-alternatives` ... 485
Section 7.3 - `udevadm` ... 485
Section 7.4 - `systemctl` ... 485
**Unit J-8 - Make** ... 486
Section 8.1 - `make` ... 486
**Unit J-9 - Python-related tools** ... 487

Section 9.1 - `virtualenv` ... 487
Section 9.2 - `pip` ... 487
**Unit J-10 - Raspberry-PI commands** ... 488
Section 10.1 - `raspi-config` ... 488
Section 10.2 - `vcgencmd` ... 488
Section 10.3 - `raspistill` ... 488
Section 10.4 - `jstest` ... 488
Section 10.5 - `swapon` ... 488
Section 10.6 - `mkswap` ... 488
**Unit J-11 - Users and permissions** ... 489
Section 11.1 - `chmod` ... 489
Section 11.2 - `groups` ... 489
Section 11.3 - `adduser` ... 489
Section 11.4 - `useradd` ... 489
**Unit J-12 - Downloading** ... 490
Section 12.1 - `curl` ... 490
Section 12.2 - `wget` ... 490
Section 12.3 - `sha256sum` ... 490
Section 12.4 - `xz` ... 490
**Unit J-13 - Shells and environments** ... 491
Section 13.1 - `source` ... 491
Section 13.2 - `which` ... 491
Section 13.3 - `export` ... 491
**Unit J-14 - Other misc commands** ... 492
Section 14.1 - `pgrep` ... 492
Section 14.2 - `npm` ... 492
Section 14.3 - `nodejs` ... 492
Section 14.4 - `ntpdate` ... 492
Section 14.5 - `chsh` ... 492
Section 14.6 - `echo` ... 492
Section 14.7 - `sh` ... 492
Section 14.8 - `fc-cache` ... 492
**Unit J-15 - Mounting USB drives** ... 493
Section 15.1 - Unmounting a USB drive ... 493
**Unit J-16 - Linux resources usage** ... 494
Section 16.1 - Measuring CPU usage using `htop` ... 494
Section 16.2 - Measuring I/O usage using `iotop` ... 494
Section 16.3 - How fast is the SD card? ... 494
**Unit J-17 - SD Cards tools** ... 495
Section 17.1 - Testing SD Card and disk speed ... 495
Section 17.2 - How to burn an image to an SD card ... 495
Section 17.3 - How to shrink an image ... 496
**Unit J-18 - Networking tools** ... 499
Section 18.1 - `hostname` ... 499
Section 18.2 - Visualizing information about the network ... 499
**Unit J-19 - Accessing computers using SSH** ... 500
Section 19.1 - Background reading ... 500
Section 19.2 - Installation of SSH ... 500
Section 19.3 - Local configuration ... 500
Section 19.4 - How to login with SSH and a password ... 501
Section 19.5 - Creating an SSH keypair ... 501
Section 19.6 - How to login without a password ... 503
Section 19.7 - Fixing SSH Permissions ... 504
Section 19.8 - `ssh-keygen` ... 504
**Unit J-20 - Wireless networking in Linux** ... 505
Section 20.1 - `iwconfig` ... 505
Section 20.2 - `iwlist` ... 505
**Unit J-21 - Moving files between computers** ... 507
Section 21.1 - `scp` ... 507
Section 21.2 - `rsync` ... 507

**Unit J-22 - VIM** ... 508
Section 22.1 - External documentation ... 508
Section 22.2 - Installation ... 508
Section 22.3 - `vi` ... 508
Section 22.4 - Suggested configuration ... 508
Section 22.5 - Visual mode ... 508
Section 22.6 - Indenting using VIM ... 508
**Unit J-23 - Atom** ... 510
Section 23.1 - Install Atom ... 510
Section 23.2 - Using Atom to code remotely ... 510
**Unit J-24 - Liclipse** ... 512
Section 24.1 - Why using IDEs ... 512
Section 24.2 - Other alternatives ... 512
Section 24.3 - Installing LiClipse ... 512
Section 24.4 - Set shortcuts for LiClipse ... 513
Section 24.5 - Shortcuts for LiClipse ... 513
Section 24.6 - Other configuration for Liclipse ... 513
**Unit J-25 - Slack** ... 514
Section 25.1 - Installing the Slack app on Linux ... 514
Section 25.2 - Disabling Slack email notifications ... 514
Section 25.3 - Disabling Slack pop-up notification on the desktop ... 514
**Unit J-26 - Byobu** ... 515
Section 26.1 - Advantages of using Byobu ... 515
Section 26.2 - Installation ... 515
Section 26.3 - Documentation ... 515
Section 26.4 - Quick command reference ... 515
Section 26.5 - Commands on OS X ... 516
**Unit J-27 - Source code control with Git** ... 517
Section 27.1 - Background reading ... 517
Section 27.2 - Installation ... 517
Section 27.3 - Setting up global configurations for Git ... 517
Section 27.4 - Git tips ... 517
Section 27.5 - Git troubleshooting ... 518
Section 27.6 - `git` ... 519
Section 27.7 - `hub` ... 519
**Unit J-28 - Git LFS** ... 520
Section 28.1 - Generic installation instructions ... 520
Section 28.2 - Ubuntu 16 installation (laptop) ... 520
Section 28.3 - Raspberry Pi 3 ... 520
Section 28.4 - Troubleshooting ... 520
**Unit J-29 - Setup Github access** ... 521
Section 29.1 - Create a Github account ... 521
Section 29.2 - Become a member of the Duckietown organization ... 521
Section 29.3 - Add a public key to Github ... 521
**Unit J-30 - ROS installation and reference** ... 524
Section 30.1 - Install ROS ... 524
Section 30.2 - `rqt_console` ... 524
Section 30.3 - `roslaunch` ... 525
Section 30.4 - `rviz` ... 525
Section 30.5 - `rostopic` ... 525
Section 30.6 - `catkin_make` ... 525
Section 30.7 - `rosrun` ... 525
Section 30.8 - `rostest` ... 525
Section 30.9 - `rospack` ... 525
Section 30.10 - `rosparam` ... 525
Section 30.11 - `rosdep` ... 526
Section 30.12 - `roswtf` ... 526
Section 30.13 - `rosbag` ... 526
Section 30.14 - `roscore` ... 527
Section 30.15 - Troubleshooting ROS ... 527

Section 30.16 - Other materials about ROS. ... 527
**Unit J-31 - How to install PyTorch on the Duckiebot** ... 528
Section 31.1 - Step 1: install dependencies and clone repository ... 528
Section 31.2 - Step 2: Change swap size ... 529
Section 31.3 - Step 3: compile PyTorch ... 529
Section 31.4 - Step 4: try it out ... 530
Section 31.5 - (Step 5, optional: unswap the swap) ... 530
**Unit J-32 - How to install Caffe and Tensorflow on the Duckiebot** ... 532
Section 32.1 - Caffe ... 532
Section 32.2 - Tensorflow ... 533
**Unit J-33 - Movidius Neural Compute Stick Install** ... 538
Section 33.1 - Laptop Installation ... 538
Section 33.2 - Duckiebot Installation ... 538
**Unit J-34 - How To Use Neural Compute Stick** ... 540
Section 34.1 - Workflow ... 540
Section 34.2 - Benchmarking ... 540

**Part K - Software development guide** ... 541
**Unit K-1 - Python** ... 542
Section 1.1 - Background reading ... 542
Section 1.2 - Python virtual environments ... 542
Section 1.3 - Useful libraries ... 542
Section 1.4 - Context managers ... 542
Section 1.5 - Exception hierarchies ... 542
Section 1.6 - Object orientation - Abstract classes, class hierarchies ... 542
Section 1.7 - Downloading resources ... 542
Section 1.8 - IPython ... 543
Section 1.9 - Idioms ... 543
**Unit K-2 - Working with YAML files** ... 544
Section 2.1 - Pointers to documentation ... 544
Section 2.2 - Editing YAML files ... 544
Section 2.3 - Reading and writing YAML files in Python ... 544
Section 2.4 - Duckietown wrapping API ... 544
**Unit K-3 - Duckietown code conventions** ... 545
Section 3.1 - Python ... 545
Section 3.2 - Logging ... 546
Section 3.3 - Exceptions ... 546
Section 3.4 - Scripts ... 546
**Unit K-4 - Configuration** ... 547
Section 4.1 - Environment variables (updated Sept 12) ... 547
Section 4.2 - The “scuderia” (vehicle database) ... 547
Section 4.3 - The `machines` file ... 548
Section 4.4 - People database ... 548
Section 4.5 - Modes of operation ... 549
**Unit K-5 - Node configuration mechanisms** ... 550
**Unit K-6 - Minimal ROS node - `pkg_name`** ... 551
Section 6.1 - The files in the package ... 551
Section 6.2 - Writing a node: `talker.py` ... 552
Section 6.3 - The `Talker` class ... 554
Section 6.4 - Launch File ... 555
Section 6.5 - Testing the node ... 556
Section 6.6 - Adding a command line parameter ... 558
Section 6.7 - Documentation ... 559
Section 6.8 - Guidelines ... 559
**Unit K-7 - Makefile system** ... 560
Section 7.1 - User guide ... 560
Section 7.2 - Makefile organization ... 560
Section 7.3 - Makefile help ... 561
**Unit K-8 - Jupyter** ... 563
Section 8.1 - Installation ... 563

Section 8.2 - Configuration ... 563
Section 8.3 - Running it ... 563
Section 8.4 - Extra configuration for virtual machines ... 564
**Unit K-9 - ROS package verification** ... 565
Section 9.1 - Naming ... 565
Section 9.2 - `package.xml` ... 565
Section 9.3 - Messages ... 565
Section 9.4 - Readme file ... 565
Section 9.5 - Launch files ... 565
Section 9.6 - Test files ... 565
**Unit K-10 - Duckietown utility library** ... 566
Section 10.1 - Images ... 566
**Unit K-11 - Bug squashing guide** ... 568
Section 11.1 - Historical notes ... 568
Section 11.2 - The basic truths of bug squashing ... 568
Section 11.3 - What could it be? ... 569
Section 11.4 - How to find the bug by yourself ... 569
Section 11.5 - How to ask for help? ... 570
Section 11.6 - How to give help ... 570
**Unit K-12 - Creating unit tests with ROS** ... 572
**Unit K-13 - Continuous integration** ... 573
Section 13.1 - Never break the build ... 573
Section 13.2 - How to stay in the green ... 573
Section 13.3 - How to make changes to `master` : pull requests ... 574
**Unit K-14 - Road Release Process** ... 577
Section 14.1 - Merge `Master` into your branch ... 577
Section 14.2 - Unit Tests ... 577
Section 14.3 - Continuous Integration ... 577
Section 14.4 - Regression Tests ... 577
Section 14.5 - Simulation Tests ... 577
Section 14.6 - Hardware-in-the-loop (HWIL) ... 577
Section 14.7 - Road Test ... 578
Section 14.8 - Make a Pull Request ... 578

**Part L - Duckietown system** ... 579
**Unit L-1 - Teleoperation** ... 580
Section 1.1 - Implementation ... 580
Section 1.2 - Camera ... 580
Section 1.3 - Actuators ... 580
Section 1.4 - IMU ... 580
**Unit L-2 - Parallel autonomy** ... 581
**Unit L-3 - Lane control** ... 582
Section 3.1 - Implementation ... 582
**Unit L-4 - Indefinite navigation** ... 583
Section 4.1 - Implementation ... 583
**Unit L-5 - Planning** ... 584
Section 5.1 - Implementation ... 584
**Unit L-6 - Coordination** ... 585
Section 6.1 - Implementation ... 585
**Unit L-7 - Duckietown ROS Guidelines** ... 586
Section 7.1 - Node and Topics ... 586
Section 7.2 - Parameters ... 586
Section 7.3 - Launch file ... 586

**Part M - Fall 2017** ... 588
**Unit M-1 - The Fall 2017 Duckietown experience** ... 589
Section 1.1 - The rules of Duckietown ... 589
**Unit M-2 - First Steps in Duckietown** ... 590
Section 2.1 - Onboarding Procedure ... 590

Section 2.2 - (optional) Add Duckietown Engineering Linkedin profile ... 591
Section 2.3 - Laptops ... 591
Section 2.4 - Next steps for people in Zurich ... 592
Section 2.5 - Next steps for people in Chicago ... 592
**Unit M-3 - Logistics for Zürich branch** ... 594
Section 3.1 - HR ... 594
Section 3.2 - Website / class journal ... 594
Section 3.3 - Duckiebox ... 594
Section 3.4 - Duckietown room access ... 594
Section 3.5 - Extra spaces ... 594
**Unit M-4 - Logistics for Montréal branch** ... 596
Section 4.1 - Website ... 596
Section 4.2 - Class Schedule ... 596
Section 4.3 - Lab Access ... 596
Section 4.4 - The Local Staff ... 596
Section 4.5 - Storing Your Robot ... 596
**Unit M-5 - Logistics for Chicago branch** ... 597
**Unit M-6 - Logistics for NCTU branch** ... 599
**Unit M-7 - Git usage guide for Fall 2017** ... 601
Section 7.1 - Differences ... 601
Section 7.2 - Repositories ... 601
Section 7.3 - Git policy for homeworks (TTIC/UDEM) ... 602
Section 7.4 - Git policy for project development ... 605
**Unit M-8 - Organization** ... 606
Section 8.1 - The Activity Tracker ... 607
Section 8.2 - The Areas sheet ... 607
**Unit M-9 - Getting and giving help** ... 608
Section 9.1 - *Who* to ask for help ... 608
Section 9.2 - *How* to ask for help ... 608
**Unit M-10 - Slack Channels** ... 610
Section 10.1 - Channels ... 610
**Unit M-11 - Zürich branch diary** ... 611
Section 11.1 - Lectures and lab sessions ... 611
Section 11.2 - Wed Sep 20: Welcome to Duckietown! ... 611
Section 11.3 - Monday Sep 25: Introduction to autonomy ... 611
Section 11.4 - Monday Sep 25, late at night: Onboarding instructions ... 612
Section 11.5 - Wednesday Sep 27: Duckiebox distribution, and getting to know each other ... 612
Section 11.6 - Thursday Sep 28: Misc announcements ... 613
Section 11.7 - Sep 28: some announcements ... 614
Section 11.8 - Oct 01 (Mon): Announcement ... 614
Section 11.9 - Oct 02 (Mon): Networking, logical/physical architectures ... 614
Section 11.10 - Oct 04 (Wed): Modeling ... 615
Section 11.11 - Oct 09 (Mon): Autonomy architectures and version control ... 615
Section 11.12 - Oct 11 (Wed): Computer vision and odometry calibration ... 615
Section 11.13 - Oct 13 (Fri): new series of tasks out ... 616
Section 11.14 - Oct 16 (Mon): Line detection ... 616
Section 11.15 - Oct 18 (Wed): Feature extraction ... 616
Section 11.16 - Oct 20 (Fri): Lab session ... 616
Section 11.17 - Oct 23 (Mon) Filtering I ... 617
Section 11.18 - Oct 25 (Wed) Filtering II ... 617
Section 11.19 - Nov 1 (Wed) Control Systems ... 617
Section 11.20 - Nov 6 (Mon) Project Pitches ... 617
Section 11.21 - Nov 8 (Wed) Motion Planing ... 617
Section 11.22 - Nov 13 (Mon) Project Team Assignments ... 617
Section 11.23 - Nov 15 (Wed) Putting things together ... 618
Section 11.24 - Nov 20 (Mon) Testing Autonomous Vehicles{#Zurich-2017-11-20} ... 618
Section 11.25 - Nov 22 (Wed) Fleet Control ... 618
Section 11.26 - Nov 27 (Mon) Inermediate design Report ... 618
Section 11.27 - Nov 29 (Wed) Fleet Control ... 618
**Unit M-12 - Montréal branch diary** ... 619

Section 12.1 - Wed Sept 6 ... 619
Section 12.2 - Friday Sept 8 ... 619
Section 12.3 - Sun Sept 10 ... 619
Section 12.4 - Mon Sept 11 ... 619
Section 12.5 - Wed Sept 13 ... 619
Section 12.6 - Mon Sept 18 ... 620
Section 12.7 - Wed Sept 20 ... 620
Section 12.8 - Mon Sept 25 ... 620
Section 12.9 - Wed Sept 27 ... 621
Section 12.10 - Mon Oct 2 ... 621
Section 12.11 - Wed Oct 4 ... 621
Section 12.12 - Mon Oct 9 ... 622
Section 12.13 - Wed Oct 11 ... 622
Section 12.14 - Friday Oct 13 ... 622
Section 12.15 - Monday Oct 16 ... 622
Section 12.16 - Wednesday Oct. 18 ... 622
Section 12.17 - Friday Oct. 20 ... 622
Section 12.18 - Monday Oct. 23 ... 622
Section 12.19 - Wednesday Oct. 25 ... 623
Section 12.20 - Monday Oct. 30 ... 623
Section 12.21 - Wednesday Nov. 1 ... 623
Section 12.22 - Monday Nov. 6 ... 623
Section 12.23 - Wednesday Nov. 8 ... 623
Section 12.24 - Thursday Nov. 9 ... 623
Section 12.25 - Monday Nov. 13 ... 623
Section 12.26 - Wednesday Nov. 15 ... 623
Section 12.27 - Oct 30 (Mon) ... 624
Section 12.28 - Nov 01 (Wed) ... 624
Section 12.29 - Nov 06 (Mon) ... 624
Section 12.30 - Nov 08 (Wed) ... 624
Section 12.31 - Nov 13 (Mon) ... 624
Section 12.32 - Nov 15 (Wed) ... 624
Section 12.33 - Nov 20 (Mon) ... 624
Section 12.34 - Nov 22 (Wed) ... 624
Section 12.35 - Nov 27 (Mon) ... 624
Section 12.36 - Nov 29 (Wed) ... 624
Section 12.37 - Dec 04 (Mon) ... 624
Section 12.38 - Dec 06 (Wed) ... 625
Section 12.39 - Dec 11 (Mon) ... 625
Section 12.40 - (Template for every lecture) Date: Topic ... 625
Section 12.41 - (Template for every lecture) Date: Topic ... 625
**Unit M-13 - Chicago branch diary** ... 626
Section 13.1 - Checkoffs: ... 626
Section 13.2 - Problem Sets: ... 626
Section 13.3 - Monday September 25: Introduction to Duckietown ... 626
Section 13.4 - Tuesday September 26: Onboarding ... 627
Section 13.5 - Wednesday, September 27: Duckiebox Ceremony ... 627
Section 13.6 - Monday, October 2: Modern Robotic Systems ... 627
Section 13.7 - Wednesday, October 4: Modern Robotic Systems (Continued) ... 628
Section 13.8 - Monday, October 9: Modeling ... 628
Section 13.9 - Wednesday, October 11: Introduction to Computer Vision ... 629
Section 13.10 - Monday, October 16: Camera Calibration and Image Filtering ... 629
Section 13.11 - Wednesday, October 18: Edge Detection and Lane Detection ... 630
Section 13.12 - Monday, October 23: Feature Detection and Place Recognition ... 630
Section 13.13 - Wednesday, October 25: Filtering I ... 630
Section 13.14 - Monday, October 30: Filtering II ... 631
Section 13.15 - Wednesday, November 1: Introduction to SLAM ... 631
Section 13.16 - Monday, November 6: Introduction to Planning ... 631
Section 13.17 - Wednesday, November 8: Introduction to Planning (Continued) ... 632
Section 13.18 - Monday, November 13: Introduction to Control ... 632

Section 13.19 - Wednesday, November 15: Introduction to Control (Continued) ... 632
Section 13.20 - Monday, November 20: Testing for Autonomous Vehicles ... 632
**Unit M-14 - NCTU branch diary** ... 633
Section 14.1 - Checkoffs: ... 633
Section 14.2 - Course Material: ... 633
Section 14.3 - Thursday September 14: Introduction to Duckietown and Creative Software Project. 633
Section 14.4 - Thursday September 21: Project Ideas ... 633
Section 14.5 - Thursday September 28: Robotics System ... 634
Section 14.6 - Thursday October 5: OpenCV, Python and Jupyter Notebook ... 634
Section 14.7 - Thursday October 19: Camera and Wheel Calibration ... 634
Section 14.8 - (Template for every lecture) Date: Topic ... 635
**Unit M-15 - Slack Channels** ... 636
Section 15.1 - Channels ... 636
**Unit M-16 - Guide for TAs** ... 637
Section 16.1 - Dramatis personae ... 637
Section 16.2 - First steps ... 637
**Unit M-17 - Checkoff: Assembly and Configuration** ... 639
Section 17.1 - Pick up your Duckiebox ... 639
Section 17.2 - Soldering your boards ... 639
Section 17.3 - Assemble your Robot ... 639
Section 17.4 - Optional: Reproduce the SD Card Image ... 639
Section 17.5 - Setup your laptop ... 640
Section 17.6 - Make your robot move ... 640
Section 17.7 - Upload your video ... 640
**Unit M-18 - Checkoff: Take a Log** ... 641
Section 18.1 - Mount your USB drive ... 641
Section 18.2 - Take a Log ... 641
Section 18.3 - Verify your log ... 641
Section 18.4 - Upload the log ... 641
**Unit M-19 - Checkoff: Robot Calibration** ... 642
Section 19.1 - Pull and rebuild your Software repo on robot and laptop ... 642
Section 19.2 - Make a branch in the duckiefleet repo ... 642
Section 19.3 - Kinematic calibration ... 642
Section 19.4 - Camera calibration ... 642
Section 19.5 - Visually verify the calibration is good in Duckietown ... 643
Section 19.6 - Submit a PR ... 643
**Unit M-20 - Homework: Data Processing (UdeM)** ... 644
Section 20.1 - Follow the git policy for homeworks ... 644
Section 20.2 - Exercise: Basic image operations ... 644
Section 20.3 - Exercise: Log decimation ... 644
Section 20.4 - Exercise: Instagram filters ... 644
Section 20.5 - Exercise: Live Instagram ... 644
**Unit M-21 - Homework: Data Processing (TTIC)** ... 645
Section 21.1 - Follow the git policy for homeworks ... 645
Section 21.2 - Exercise: Basic image operations ... 645
Section 21.3 - Exercise: Log analysis ... 645
Section 21.4 - Exercise: Log decimation ... 645
Section 21.5 - Exercise: Video thumbnails ... 645
Section 21.6 - Exercise: Instagram filters ... 645
Section 21.7 - Exercise: Log Instagram ... 645
Section 21.8 - Exercise: Live Instagram ... 645
Section 21.9 - Exercise: Feedback ... 646
**Unit M-22 - Exercises: Data Processing (Zurich)** ... 647
Section 22.1 - Git setup ... 647
Section 22.2 - The exercises ... 647
Section 22.3 - Exercise: Basic image operations ... 647
**Unit M-23 - Homework: Augmented Reality** ... 649
Section 23.1 - Follow the git policy for homeworks ... 649
Section 23.2 - Exercise: Augmented Reality ... 649
Section 23.3 - Submission ... 649

Section 23.4 - Bonus: Defining `intersection_4way.yaml` ... 650
**Unit M-24 - Checkoff: Navigation** ... 651
Section 24.1 - Pull from master ... 651
Section 24.2 - Lane Following ... 651
Section 24.3 - Indefinite Navigation ... 652
**Unit M-25 - Homework: Lane Filtering** ... 653
Section 25.1 - Follow the git policy for homeworks ... 653
Section 25.2 - Pick your Poison ... 653
Section 25.3 - Setup instructions ... 653
Section 25.4 - Submission ... 654
**Unit M-26 - Guide for mentors** ... 655
**Unit M-27 - Project proposals** ... 656
**Unit M-28 - System Architecture** ... 657
Section 28.1 - Preliminaries ... 657
Section 28.2 - Mission 1 ... 657
Section 28.3 - Mission 2 ... 658
**Unit M-29 - Template of a project** ... 661
Section 29.1 - Preliminaries ... 661
Section 29.2 - Problem Statement ... 661
Section 29.3 - Relevant Resources ... 661
Section 29.4 - Deliverables (Goals) ... 661
Section 29.5 - Proposed Approach ... 662
Section 29.6 - Logging and Testing Procedure ... 662
Section 29.7 - Current status ... 662
Section 29.8 - Tasks ... 662
Section 29.9 - Timeline ... 662
Section 29.10 - Meetings notes ... 663
**Unit M-30 - The Map Description** ... 664

**Part N - Fall 2017 projects** ... 665
Section 0.1 - Instructions for using the template ... 665
Section 0.2 - Group names and ID numbers ... 665
**Unit N-1 - Group name: preliminary design document** ... 666
Section 1.1 - Part 1: Mission and scope ... 666
Section 1.2 - Part 2: Definition of the problem ... 666
Section 1.3 - Part 3: Preliminary design ... 668
Section 1.4 - Part 4: Project planning ... 669
**Unit N-2 - Group name: intermediate report** ... 670
Section 2.1 - Part 1: System interfaces ... 670
Section 2.2 - Part 2: Demo and evaluation plan ... 670
Section 2.3 - Part 3: Data collection, annotation, and analysis ... 671
**Unit N-3 - Group name: final report** ... 672
Section 3.1 - The final result ... 672
Section 3.2 - Mission and Scope ... 672
Section 3.3 - Definition of the problem ... 673
Section 3.4 - Contribution / Added functionality ... 673
Section 3.5 - Formal performance evaluation / Results ... 673
Section 3.6 - Future avenues of development ... 673
**Unit N-4 - The Heroes quests** ... 674
Section 4.1 - Motto ... 674
Section 4.2 - Overview ... 674
Section 4.3 - Quest 1 ... 674
Section 4.4 - Quest 1, Part 1: Mission and scope ... 675
Section 4.5 - Quest 1, Part 2: Definition of the problem ... 675
Section 4.6 - Quest 1, Part 3: Preliminary design ... 676
Section 4.7 - Quest 1, Part 4: Project planning ... 676
Section 4.8 - Quest 2 ... 677
Section 4.9 - Quest 2, Part 1: Mission and scope ... 677
Section 4.10 - Quest 2, Part 2: Definition of the problem ... 677
Section 4.11 - Quest 2, Part 3: Preliminary design ... 678

Section 4.12 - Quest 2, Part 4: Project planning ..... 678
**Unit N-5 - PDD - Smart City** ..... 680
Section 5.1 - Part 1: Mission and scope ..... 680
Section 5.2 - Part 2: Definition of the problem ..... 680
Section 5.3 - Part 3: Preliminary design ..... 685
Section 5.4 - Part 4: Project planning ..... 685
**Unit N-6 - PDD - System Identification** ..... 687
Section 6.1 - Part 1: Mission and scope ..... 687
Section 6.2 - Part 2: Definition of the problem ..... 687
Section 6.3 - Part 3: Preliminary design ..... 690
Section 6.4 - Part 4: Project planning ..... 690
**Unit N-7 - System Identification: Intermediate Report** ..... 692
Section 7.1 - Part 1: System interfaces ..... 692
Section 7.2 - Part 2: Demo and evaluation ..... 694
Section 7.3 - Plan for formal performance evaluation ..... 694
Section 7.4 - Part 3: Data collection, annotation, and analysis ..... 695
Section 7.5 - Analysis ..... 696
**Unit N-8 - The Controllers: preliminary report** ..... 698
Section 8.1 - Part 1: Mission and scope ..... 698
Section 8.2 - Part 2: Definition of the problem ..... 699
Section 8.3 - Part 3: Preliminary design ..... 701
Section 8.4 - Part 4: Project planning ..... 702
**Unit N-9 - The Controllers: Intermediate Report** ..... 704
Section 9.1 - Part 1: System interfaces ..... 704
Section 9.2 - Part 2: Demo and evaluation plan ..... 711
Section 9.3 - Part 3: Data collection, annotation, and analysis ..... 713
**Unit N-10 - The Controllers: final report** ..... 715
Section 10.1 - The final result ..... 715
Section 10.2 - Mission and Scope ..... 715
Section 10.3 - Definition of the problem ..... 718
Section 10.4 - Contribution / Added functionality ..... 721
Section 10.5 - Formal performance evaluation / Results ..... 726
Section 10.6 - Future avenues of development ..... 744
**Unit N-11 - PDD - Saviors** ..... 746
Section 11.1 - Part 1: Mission and scope ..... 746
Section 11.2 - Part 2: Definition of the problem ..... 746
Section 11.3 - Part 3: Preliminary design ..... 748
Section 11.4 - Part 4: Project planning ..... 749
**Unit N-12 - The Saviors: intermediate report** ..... 751
Section 12.1 - Part 1: System interfaces ..... 751
Section 12.2 - Part 2: Demo and evaluation plan ..... 753
Section 12.3 - Part 3: Data collection, annotation, and analysis ..... 754
**Unit N-13 - The Saviors: Final Report** ..... 756
Section 13.1 - Structure ..... 756
Section 13.2 - The Final Result ..... 756
Section 13.3 - Mission and Scope ..... 756
Section 13.4 - Definition of the Problem ..... 758
Section 13.5 - Contribution / Added Functionality ..... 761
Section 13.6 - Formal Performance Evaluation / Results ..... 775
Section 13.7 - Future Avenues of Development ..... 778
Section 13.8 - Theory Chapter ..... 779
**Unit N-14 - PDD - Navigators** ..... 791
Section 14.1 - Part 1: Mission and Scope ..... 791
Section 14.2 - Part 2: Definition of the Problem ..... 792
Section 14.3 - Part 3: Preliminary Design ..... 794
Section 14.4 - Part 4: Project Planning ..... 795
**Unit N-15 - The Navigators: intermediate report** ..... 797
Section 15.1 - Part 1: System interfaces ..... 797
Section 15.2 - Part 2: Demo and evaluation plan ..... 801
Section 15.3 - Part 3: Data collection, annotation, and analysis ..... 802

**Unit N-16 - PDD - Parking** ... 803
Section 16.1 - Part 1: Mission and scope ... 803
Section 16.2 - Part 2: Definition of the problem ... 803
Section 16.3 - Part 3: Preliminary design ... 804
Section 16.4 - Part 4: Project planning ... 806
**Unit N-17 - Parking: intermediate report** ... 808
Section 17.1 - Part 1: System interfaces ... 808
Section 17.2 - Part 2: Demo and evaluation plan ... 810
Section 17.3 - Part 3: Data collection, annotation, and analysis ... 810
**Unit N-18 - Explicit Coordination: preliminary report** ... 812
Section 18.1 - Part 1: Mission and scope ... 812
Section 18.2 - Part 2: Definition of the problem ... 813
Section 18.3 - Part 3: Preliminary design ... 817
Section 18.4 - Part 4: Project planning ... 818
**Unit N-19 - Explicit Coordination: intermediate Report** ... 821
Section 19.1 - Part 1: System interfaces ... 821
Section 19.2 - Part 2: Demo and evaluation plan ... 823
Section 19.3 - Part 3: Data collection, annotation, and analysis ... 826
**Unit N-20 - Explicit coordination: final report** ... 827
Section 20.1 - The final result ... 827
Section 20.2 - Mission and Scope ... 827
Section 20.3 - Definition of the problem ... 828
Section 20.4 - Contribution / Added functionality ... 829
Section 20.5 - Formal performance evaluation / Results ... 833
Section 20.6 - Future avenues of development ... 834
**Unit N-21 - Implicit Coordination: preliminary report** ... 835
Section 21.1 - Part 1: Mission and scope ... 835
Section 21.2 - Part 2: Definition of the problem ... 836
Section 21.3 - Part 3: Preliminary design ... 837
Section 21.4 - Part 4: Project planning ... 838
**Unit N-22 - Implicit Coordination: intermediate report** ... 841
Section 22.1 - Part 1: System interfaces ... 841
Section 22.2 - Part 2: Demo and evaluation plan ... 843
Section 22.3 - Part 3: Data collection, annotation, and analysis ... 844
**Unit N-23 - Implicit Coordination: final report** ... 846
Section 23.1 - The final result ... 846
Section 23.2 - Motivation, Mission and Scope ... 846
Section 23.3 - Opportunity and Existing solution ... 847
Section 23.4 - Definition of the problem ... 847
Section 23.5 - Contribution / Added functionality ... 847
Section 23.6 - Results and Performance Evaluation ... 850
Section 23.7 - Future Avenues ... 851
**Unit N-24 - PDD - Distributed Estimation** ... 852
Section 24.1 - Part 1: Mission and scope ... 852
Section 24.2 - Part 2: Definition of the problem ... 852
Section 24.3 - Part 3: Preliminary design ... 854
Section 24.4 - Part 4: Project planning ... 855
**Unit N-25 - Distributed Estimation: intermediate report** ... 858
Section 25.1 - Part 1: System interfaces ... 858
Section 25.2 - Part 2: Demo and evaluation plan ... 859
Section 25.3 - Part 3: Data collection, annotation, and analysis ... 860
**Unit N-26 - Fleet Messaging: final report** ... 861
Section 26.1 - The final result ... 861
Section 26.2 - Mission and Scope ... 861
Section 26.3 - Definition of the problem ... 862
Section 26.4 - Contribution / Added functionality ... 862
Section 26.5 - Formal performance evaluation / Results ... 863
Section 26.6 - Future avenues of development ... 864
**Unit N-27 - Demo instructions Fleet Communications** ... 867
Section 27.1 - Duckietown setup notes ... 867

Section 27.2 - Duckiebot setup notes ... 867
Section 27.3 - Pre-flight checklist ... 867
Section 27.5 - Demo instructions ... 867 + Ref. error
More than one element matching '#sub:demo-template-run-249-demoInstructions_fleetcommunication-3310ba4'. Section
27.5 - Demo instructions ... 867 + Ref. error
More than one element matching '#sub:demo-template-run-249-demoInstructions_fleetcommunication-3310ba4'.
Section 27.6 - Troubleshooting ... 869
Section 27.7 - Demo failure demonstration ... 869
**Unit N-28 - Single SLAM Project** ... 870
Section 28.1 - Part 1: Mission and scope ... 870
Section 28.2 - Part 2: Definition of the problem ... 870
Section 28.3 - Part 3: Preliminary design ... 871
Section 28.4 - Part 4: Project planning ... 871
**Unit N-29 - Fleet Planning: Preliminary Report** ... 873
Section 29.1 - Part 1: Mission and scope ... 873
Section 29.2 - Part 2: Definition of the problem ... 873
Section 29.3 - Part 3: Preliminary design ... 875
Section 29.4 - Part 4: Project planning ... 875
**Unit N-30 - Fleet Planning: Intermediate Report** ... 878
Section 30.1 - Part 1: System Interfaces ... 878
Section 30.2 - Part 2: Demo and evaluation plan ... 881
Section 30.3 - Part 3: Data collection, annotation, and analysis ... 885
**Unit N-31 - Fleet Planning: Final Report** ... 887
Section 31.1 - 3.1 The final result ... 887
Section 31.2 - 3.2 Mission and scope ... 887
Section 31.3 - 3.3 Definition of the problem ... 889
Section 31.4 - 3.4 Contribution / Added functionality ... 890
Section 31.5 - 3.5 Formal performance evaluation / Results ... 899
Section 31.6 - 3.6 Future avenues of development ... 901
**Unit N-32 - 4 Conclusion** ... 904
Section 32.1 - References ... 904
**Unit N-33 - Transferred Lane following** ... 906
Section 33.1 - Video of expected results ... 906
Section 33.2 - Duckietown setup notes ... 906
Section 33.3 - Duckiebot setup notes ... 906
Section 33.4 - Pre-flight checklist ... 906
Section 33.5 - Demo instructions ... 906
Section 33.6 - Train the network from scratch ... 907
Section 33.7 - Troubleshooting ... 907
**Unit N-34 - Transfer Learning in Robotics** ... 909
Section 34.1 - Transfer Learning Definition ... 909
Section 34.2 - Why is transfer learning important in autonomous driving (or duckietown) ... 909
Section 34.3 - Transfer Learning in duckietown ... 909
**Unit N-35 - PDD - Supervised-learning** ... 912
Section 35.1 - Part 1: Mission and scope ... 912
Section 35.2 - Part 2: Definition of the problem ... 912
Section 35.3 - Part 3: Preliminary design ... 914
Section 35.4 - Part 4: Project planning ... 915
**Unit N-36 - Supervised Learning: intermediate report** ... 917
Section 36.1 - Part 1: System interfaces ... 917
Section 36.2 - Part 2: Demo and evaluation plan ... 918
Section 36.3 - Part 3: Data collection, annotation, and analysis ... 919
**Unit N-37 - PDD Neural Slam** ... 921
Section 37.1 - Part 1: Mission and scope ... 921
Section 37.2 - Part 2: Definition of the problem ... 921
Section 37.3 - Part 3: Preliminary design ... 922
Section 37.4 - Part 4: Project planning ... 922
**Unit N-38 - PDD - Visual Odometry** ... 924
Section 38.1 - Part 1: Mission and scope ... 924

Section 38.2 - Part 2: Definition of the problem ... 924
Section 38.3 - Part 3: Preliminary design ... 925
Section 38.4 - Part 4: Project planning ... 926
**Unit N-39 - Visual Odometry Project** ... 927
Section 39.1 - Epipolar geometry and DIBR ... 927
Section 39.2 - Learning Depth Prediction ... 927
Section 39.3 - True Depth ... 929
Section 39.4 - Conclusions ... 929
Section 39.5 - References ... 930
**Unit N-40 - Deep Visual Odometry ROS Package** ... 931
Section 40.1 - devel-visual-odometry ... 931
**Unit N-41 - PDD - Anti-Instagram** ... 932
Section 41.1 - Part 1: Mission and scope ... 932
Section 41.2 - Part 2: Definition of the problem ... 933
Section 41.3 - Part 3: Preliminary design ... 935
Section 41.4 - Part 4: Project planning ... 936
**Unit N-42 - Transfer: preliminary design document** ... 939
Section 42.1 - Part 1: Mission and scope ... 939
Section 42.2 - Part 2: Definition of the problem ... 940
Section 42.3 - Part 3: Preliminary design ... 941
Section 42.4 - Part 4: Project planning ... 941

**Part O - Packages - Infrastructure** ... 943
**Unit O-1 - Package `duckieteam`** ... 944
Section 1.1 - Package information ... 944
Section 1.2 - `create-machines` ... 944
Section 1.3 - `create-roster` ... 944
**Unit O-2 - Package `duckietown`** ... 945
Section 2.1 - Package information ... 945
**Unit O-3 - Package `duckietown_msgs`** ... 946
Section 3.1 - Package information ... 946
**Unit O-4 - Package `easy_algo`** ... 947
Section 4.1 - Package information ... 947
Section 4.2 - Motivation ... 947
Section 4.3 - Usage ... 947
Section 4.4 - User interface ... 948
Section 4.5 - The developer's point of view ... 948
**Unit O-5 - Package `easy_logs`** ... 950
Section 5.1 - Package information ... 950
Section 5.2 - Overview ... 950
Section 5.3 - Command-line utilities `find`, `summary`, `details` ... 950
Section 5.4 - Selector language ... 952
Section 5.5 - Automatic log download using `download` ... 953
Section 5.6 - Browsing the cloud ... 953
Section 5.7 - Advanged log indexing and generation ... 954
Section 5.8 - How to set up the backend needed to use files from the cloud ... 955
**Unit O-6 - Package `easy_node`** ... 957
Section 6.1 - Package information ... 957
Section 6.2 - YAML file format ... 957
Section 6.3 - Using the `easy_node` API ... 959
Section 6.4 - Configuration using `easy_node`: the user's point of view ... 963
Section 6.5 - Visualizing the configuration database ... 965
Section 6.6 - Benchmarking ... 967
Section 6.7 - Automatic documentation generation ... 968
Section 6.8 - Parameters and services defined for all packages ... 969
Section 6.9 - Other ideas ... 969
**Unit O-7 - Package `easy_regression`** ... 970
Section 7.1 - Package information ... 970
Section 7.2 - Design goals ... 970
Section 7.3 - Formalization ... 970

Section 7.4 - Example of configuration file ... 972
Section 7.5 - Language for the conditions to check ... 972
**Unit O-8 - Package `what_the_duck`** ... 974
Section 8.1 - Package information ... 974

**Part P - Packages - Teleoperation** ... 975
**Unit P-1 - Package `adafruit_drivers`** ... 976
Section 1.1 - Package information ... 976
**Unit P-2 - Package `dagu_car`** ... 977
Section 2.1 - Package information ... 977
Section 2.2 - Node `forward_kinematics_node` ... 977
Section 2.3 - Node `inverse_kinematics_node` ... 977
Section 2.4 - Node `velocity_to_pose_node` ... 978
Section 2.5 - Node `car_cmd_switch_node` ... 978
Section 2.6 - Node `wheels_driver_node` ... 978
Section 2.7 - Node `wheels_trimmer_node` ... 979
**Unit P-3 - Package `joy_mapper`** ... 980
Section 3.1 - Package information ... 980
Section 3.2 - Testing ... 980
Section 3.3 - Dependencies ... 980
Section 3.4 - Node `joy_mapper_node2` ... 980
**Unit P-4 - Package `pi_camera`** ... 982
Section 4.1 - Package information ... 982
Section 4.2 - Node `camera_node_sequence` ... 982
Section 4.3 - Node `camera_node_continuous` ... 982
Section 4.4 - Node `decoder_node` ... 983
Section 4.5 - Node `img_process_node` ... 983
Section 4.6 - Node `cam_info_reader_node` ... 983

**Part Q - Packages - Lane control** ... 985
**Unit Q-1 - Package `anti_instagram`** ... 986
Section 1.1 - Package information ... 986
Section 1.2 - Unit tests integrated with `rostest` ... 986
Section 1.3 - Unit tests needed external files ... 986
Section 1.4 - Node `anti_instagram_node` ... 986
**Unit Q-2 - Package `complete_image_pipeline`** ... 988
Section 2.1 - Package information ... 988
Section 2.2 - Program `single_image_pipeline` ... 988
**Unit Q-3 - Basic Markduck guide** ... 990
Section 3.1 - Markdown ... 990
Section 3.2 - Variables in command lines and command output ... 990
Section 3.3 - Character escapes ... 990
Section 3.4 - Keyboard keys ... 990
Section 3.5 - Figures ... 991
Section 3.6 - Subfigures ... 992
Section 3.7 - Shortcut for tables ... 993
Section 3.8 - Linking to documentation ... 993
**Unit Q-4 - Basic Markduck guide** ... 996
Section 4.1 - Markdown ... 996
Section 4.2 - Variables in command lines and command output ... 996
Section 4.3 - Character escapes ... 996
Section 4.4 - Keyboard keys ... 996
Section 4.5 - Figures ... 997
Section 4.6 - Subfigures ... 998
Section 4.7 - Shortcut for tables ... 999
Section 4.8 - Linking to documentation ... 999
**Unit Q-5 - Package `ground_projection`** ... 1002
Section 5.1 - Package information ... 1002
Section 5.2 - Node `ground_projection_node` ... 1002

**Unit Q-6 - Package `lane_control`** ..... 1003
Section 6.1 - Package information ..... 1003
Section 6.2 - `lane_controller_node` ..... 1003
**Unit Q-7 - Package `lane_filter`** ..... 1005
Section 7.1 - Package information ..... 1005
Section 7.2 - `lane_filter_node` ..... 1005
**Unit Q-8 - Package `line_detector2`** ..... 1008
Section 8.1 - Package information ..... 1008
Section 8.2 - Testing the line detector using visual inspection ..... 1008
Section 8.3 - Quantitative tests ..... 1009
Section 8.4 - `line_detector_node2` ..... 1010
**Unit Q-9 - Package `line_detector`** ..... 1012
Section 9.1 - Package information ..... 1012

**Part R - Packages - Indefinite navigation** ..... 1013
**Unit R-1 - AprilTags library** ..... 1014
Section 1.1 - Package information ..... 1014
**Unit R-2 - Package `apriltags_ros`** ..... 1016
Section 2.1 - Package information ..... 1016
**Unit R-3 - Package `fsm`** ..... 1017
Section 3.1 - Package information ..... 1017
Section 3.2 - Node `fsm_node` ..... 1017
Section 3.3 - Node `logic_gate_node` ..... 1019
**Unit R-4 - Package `indefinite_navigation`** ..... 1021
Section 4.1 - Package information ..... 1021
**Unit R-5 - Package `intersection_control`** ..... 1022
Section 5.1 - Package information ..... 1022
**Unit R-6 - Package `navigation`** ..... 1023
Section 6.1 - Package information ..... 1023
**Unit R-7 - Package `stop_line_filter`** ..... 1024
Section 7.1 - Package information ..... 1024

**Part S - Packages - Localization and planning** ..... 1025
**Unit S-1 - Package `duckietown_description`** ..... 1026
Section 1.1 - Package information ..... 1026
**Unit S-2 - Package `localization`** ..... 1027
Section 2.1 - Package information ..... 1027

**Part T - Packages - Coordination** ..... 1028
**Unit T-1 - Package `led_detection`** ..... 1029
Section 1.1 - Package information ..... 1029
Section 1.2 - LED detector ..... 1029
Section 1.3 - Unit tests ..... 1029
**Unit T-2 - Package `led_emitter`** ..... 1031
Section 2.1 - Package information ..... 1031
Section 2.2 - In depth ..... 1031
**Unit T-3 - Package `led_interpreter`** ..... 1033
Section 3.1 - Package information ..... 1033
**Unit T-4 - Package `led_joy_mapper`** ..... 1034
Section 4.1 - Package information ..... 1034
**Unit T-5 - Package `rgb_led`** ..... 1035
Section 5.1 - Package information ..... 1035
Section 5.2 - Demos ..... 1035
**Unit T-6 - Package `traffic_light`** ..... 1036
Section 6.1 - Package information ..... 1036

**Part U - Packages - Additional functionality** ..... 1037
**Unit U-1 - Package `mdoap`** ..... 1038
Section 1.1 - Package information ..... 1038

**Unit U-2 - Package `parallel_autonomy`** ..... 1039
Section 2.1 - Package information ..... 1039
**Unit U-3 - Package `vehicle_detection`** ..... 1040
Section 3.1 - Package information ..... 1040

**Part V - Packages - Templates** ..... 1041
**Unit V-1 - Package `pkg_name`** ..... 1042
Section 1.1 - Package information ..... 1042
Section 1.2 - How to use this template ..... 1042
**Unit V-2 - Package `rostest_example`** ..... 1043
Section 2.1 - Package information ..... 1043

**Part W - Packages - Convenience** ..... 1044
**Unit W-1 - Package `duckie_rr_bridge`** ..... 1045
Section 1.1 - Package information ..... 1045
**Unit W-2 - Package `duckiebot_visualizer`** ..... 1046
Section 2.1 - Package information ..... 1046
Section 2.2 - Node `duckiebot_visualizer` ..... 1046
**Unit W-3 - Package `duckietown_demos`** ..... 1047
Section 3.1 - Package information ..... 1047
**Unit W-4 - Package `duckietown_unit_test`** ..... 1048
Section 4.1 - Package information ..... 1048
**Unit W-5 - Package `veh_coordinator`** ..... 1049
Section 5.1 - Package information ..... 1049

**Part X - Packages - Deep Learning** ..... 1050
**Unit X-1 - Last modified** ..... 1051

# Part A

# The Duckietown project

# Unit A-1
# What is Duckietown?

## 1.1. Goals and objectives

Duckietown is a robotics education and outreach effort.

The most tangible goal of the project is to provide a low-cost educational platform for learning about autonomous systems, consisting of lectures and other learning material, the Duckiebot autonomous robots, and the Duckietowns, which constitute the infrastructure in which the Duckiebots navigate.

We focus on the *learning experience* as a whole, by providing a set of modules, teaching plans, and other guides, as well as a curated role-play experience.

We have two targets:

1. For **instructors**, we want to create a "class-in-a-box" that allows people to offer a modern and engaging learning experience. Currently, this is feasible at the advanced undergraduate and graduate level, though in the future we would like to provide a platform that can be adapted to a range of different grade and experience levels.
2. For **self-guided learners**, we want to create a "self-learning experience" that allows students to go from having zero knowledge of robotics to a graduate-level understanding.

In addition, the Duckietown platform is also suitable for research.

## 1.2. Learn about the Duckietown educational experience

The video in Figure 1.1 is the "Duckumentary", a documentary about the first version of the class, during Spring 2016.

Figure 1.1. The Duckumentary, created by Chris Welch.

The video in Figure 1.2 is a documentary created by Red Hat on the current developments in self-driving cars.

Figure 1.2. The road to autonomy

If you'd like to know more about the educational experience, [1] present a more formal description of the course design for Duckietown: learning objectives, teaching methods, etc.

## 1.3. Learn about the platform

The video in Figure 1.3 shows some of the functionality of the platform.

If you would like to know more, the paper [2] describes the Duckiebot and its software. (With 30 authors, we made the record for a robotics conference!)

Figure 1.3. Duckietown functionality

# Unit A-2
# Duckietown history and future

## 2.1. The beginnings of Duckietown

The original Duckietown class was at MIT in 2016 (Figure 2.1).

Figure 2.1. Part of the first MIT class, during the final demo.

Duckietown was built by elves (Figure 2.2).

Figure 2.2. The elves of Duckietown

These are some advertisement videos we used.

Figure 2.3. The need for autonomy

Figure 2.4. Advertisement

Figure 2.5. Cool Duckietown by night

## 2.2. University-level classes in 2016

Later that year, the Duckietown platform was also used in these classes:

- National Chiao Tung University 2016, Taiwan - Prof. Nick Wang;
- Tsinghua University, People's Republic of China - Prof. (Samuel) Qing-Shan Jia's *Computer Networks with Applications* course;
- Rennselaer Polytechnic Institute 2016 - Prof. John Wen;

Figure 2.6. Duckietown at NCTU in 2016

## 2.3. University-level classes in 2017

In 2017, these four courses will be taught together, with the students interacting

among institutions:

- ETH Zürich 2017 - Prof. Emilio Frazzoli, Dr. Andrea Censi;
- University of Montreal, 2017 - Prof. Liam Paull;
- TTI/Chicago 2017 - Prof. Matthew Walter; and
- National Chiao Tung University, Taiwan - Prof. Nick Wang.

Furthermore, the Duckietown platform is used also in the following universities:

- Rennselaer Polytechnic Institute (Jeff Trinkle)
- National Chiao Tung University, Taiwan - Prof. Yon-Ping Chen's *Dynamic system simulation and implementation* course.
- Chosun University, Korea - Prof. Woosuk Sung's course;
- Petra Christian University, Indonesia - Prof. Resmana Lim's *Mobile Robot Design Course*
- National Tainan Normal University, Taiwan - Prof. Jen-Jee Chen's *Vehicle to Everything* (V2X) course; and
- Yuan Zhu University, Taiwan - Prof. Kan-Lin Hsiung's Control course.

## 2.4. Chile

TODO: to write

## 2.5. Duckietown High School

### 1) Introduction

DuckietownHS is inspired by the Duckietown project and targeted for high schools. The goal is to build and program duckiebots capable of moving autonomously on the streets of Duckietown. The technical objectives of DuckietownHS are simplified compared to those of the Duckietown project intended for universities so it is perfectly suited to the technical knowledge of the classes involved. The purpose is to create self-driving DuckiebotHS vehicles which can make choices and move autonomously on the streets of Duckietown, using sensors installed on the vehicles and special road signs positioned within Duckietown.

Once DuckiebotHS have been assembled and programmed to meet the specifications contained in this document and issued by the "customer" Perlatecnica, special missions and games will be offered for DuckiebotHS. The participants can also submit their own missions and games.

Just like the university project, DuckietownHS is an open source project, a role-playing game, a means to raise awareness on the subject and a learning experience for everyone involved. The project is promoted by the non-profit organization Perlatecnica based in Italy.

### 2) Purpose

The project has two main purposes:

- It is a course where students and teachers take part in a role play and they take the typical professional roles of an engineering company. They must design and implement a Duckietown responding to the specifications of the project, assemble

DuckiebotHS (DBHS), and develop the software that will run on them. The deliverables of the project will be tutorials, how-to, source code, documentation, binaries and images and them will be designed and manufactured according to the procedures of the DTE.

- In respect of that mentioned above, special missions and games for DBHS will be introduced by the "customer" Perlatecnica.

### 3) Perlatecnica's role

Perlatecnica assumes the role of the customer and commissions the Duckietown Engineering company to design and construct the Duckietown and DuckiebotHS. It will provide all necessary product requirements and will assume the responsibility to validate the compliancy of all deliverables to the required specifications.

### 4) The details of the project

The project consists in the design and realization of DuckiebotHS and DuckietownHS. They must have the same characteristics as the city of the University project as far as the size and color of the delimiting roadway bands is concerned but with a different type of management of the traffic lights system that regulates the passage of DuckiebotHS at intersections. The DuckietownHS (DTHS) and DuckiebotHS (DBHS) are defined in the documentation and there is little room for the DTE to make its own choices in terms of design. The reason for this is that the DBHS produced by the different DTE's need to be identical from a hardware point of view so that the software development makes the difference.

### 5) Where to start

The purchase of the necessary materials is the first step to take. For both DTHS and DBHS a list of these materials is provided with links to possible sellers. Even though Amazon is typically indicated as a seller this is nothing more than an indication to facilitate the purchase for those less experienced. It is left to the individual DTE to choose where to buy the required parts. It is allowed to buy and use parts that are not on the list but this is not recommended as they will make the Duckiebot unfit to enter in official competitions. When necessary an assembly tutorial will be provided together with the list of materials. Once the DTHS city and the DBHS robots have been assembled, the next step will be the development of the software for the running of both the city and the DuckiebotHS. The city and the Duckiebot run on a board based on a microcontroller STM32 from STMicroelectronics the Nucleo F401RE that will be programmed via the online development environment mbed. Perlatecnica will not release any of the official codes necessary for the navigation of the DuckiebotHS as these are owned by the DTE who developed them. The full standard document is available on the project official web site.

Each DTE may release the source code under a license Creative Commons CC BY-SA 4.0.

### 6) The first mission of the Duckiebot

Once you have completed the assembly of all the parts that make up the Duckietown and DuckiebotHS you should start programming the microcontroller so that the Duckiebot can move independently.

The basic mission of the DuckiebotHS is to move autonomously on the roads respecting the road signs and traffic lights, choosing a random journey and without crashing into other DuckiebotHS.

For the development of the code, there are no architectural constraints, but we recommend proceeding with order and to focus primarily on its major functions and not on a specific mission.

The main functions are those of perception and movement.

Moving around in DuckietownHS, the DuckiebotHS will have to drive on straight roads, make 90 degree curves while crossing an intersection but also make other unexpected curves. While doing all this the Duckiebot can be supported by a gyroscope that provides guidance to the orientation of the vehicle.

UNIT A-3

# Duckietown classes

Duckietown is an international effort. Many educational institutions have adopted the platform and used to teach robotics and related subjects. If you have used Duckietown in any of your course and cannot find a mention to it here, contact us.

**TODO:** add contact email

Here, we provide a chronologically ordered compendium of the Duckietown related learning experiences worldwide.

## 3.1. 2016

Duckietown was created in 2016.

### 1) Massachusetts Institute of Technology

Location: United States of America, Cambridge, Massachusetts

Course title: 2.166 Vehicle Autonomy

Instructors: *plenty*

Educational Level: Graduate

Time period:

Link:

Highlights: Role-playing experience (Duckietown Engineering Co.), public demo

Summary:

Figure 3.1. Duckietown at MIT in 2016

### 2) National Chiao Tung University

Location: Hsinchu, Taiwan
Course title: Robotic Vision
Instructors: Prof. Nick Wang
Educational Level: Graduate
Time period:
Link:
Summary:
Highlights:

### 3) Tsinghua University

Location:
Course title:
Instructors:
Educational Level:
Time period:
Link:
Summary:
Highlights:

### 4) Rennselaer Polytechnic Institute

Location:
Course title:
Instructors:
Educational Level:
Time period:
Link:
Summary:
Highlights:

## 3.2. 2017

### 1) ETH Zürich

Location: Zürich
Course title: Autonomous Mobility on Demand (AMOD): from car to fleet (151-0323-00L)
Instructors: E. Frazzoli, A. Censi
Educational Level: Graduate
Time period: Sept. - Dec. 2017

Link: Official Course Website

Summary: 46 students

Highlights:

### 2) Université de Montréal

Location: Montreal

Course title: IFT 6080 Sujets en exploitation des ordinateurs

Instructors: Liam Paull

Educational Level: Graduate

Time period: Sept. - Dec. 2017

Link: Official Course Website

Summary: 12 students admitted spanning across Université de Montréal, École Polytechnic, McGill University and Concordia University

Highlights:

### 3) Toyota Technological Institute at Chicago / University of Chicago

Location: Chicago

Course title: TTIC 31240 Self-driving Vehicles: Models and Algorithms for Autonomy

Instructors: Matthew R. Walter

Educational Level: Undergraduate/Graduate

Time period: Sept. - Dec. 2017

Link: Official Course Website

Summary: 12 students admitted from TTI-Chicago and the University of Chicago

Highlights:

### 4) National Chiao Tung University

Location: Hsinchu, Taiwan

Course title: Creative Software Project - Autonomous Vehicle

Instructors: Hsueh-Cheng 'Nick' Wang

Educational Level: Undergraduate

Time period: Sept. 2017 - Jan. 2018

Link: Official Course Website

Summary: 57 students (and 10 teaching assistants) admitted from National Chiao Tung University

Highlights: Marine Robot, Multi-Robot Patrolling, Robot Terrian Discrimination in Gazebo, and Depth Lane Following

# Unit A-4
# First steps

## 4.1. How to get started

If you are an instructor, please jump to Section 4.2 - Duckietown for instructors.

If you are a self-guided learner, please jump to Section 4.3 - Duckietown for self-guided learners.

If you are a company, and interested in working with Duckietown, please jump to Section 4.4 - Introduction for companies.

## 4.2. Duckietown for instructors

TODO: to write

## 4.3. Duckietown for self-guided learners

TODO: to write

## 4.4. Introduction for companies

TODO: to write

## 4.5. How to keep in touch

TODO: add link to Facebook

TODO: add link to Mailing list

TODO: add link to Slack?

## 4.6. How to contribute

TODO: If you want to contribute to the software...

TODO: If you want to contribute to the hardware...

TODO: If you want to contribute to the documentation...

TODO: If you want to contribute to the dissemination...

## 4.7. Frequently Asked Questions

## 1) General questions

*Q: What is Duckietown?*

Duckietown is a low-cost educational and research platform.

*Q: Is Duckietown free to use?*

Yes. All materials are released according to an open source license.

*Q: Is everything ready?*

Not quite! Please sign up to our mailing list to get notified when things are a bit more ready.

*Q: How can I start?*

See the section First Steps.

*Q: How can I help?*

If you would like to help actively, please email duckietown@mit.edu.

## 2) FAQ by students / independent learners

*Q: I want to build my own Duckiebot. How do I get started?*

**TODO:** to write

## 3) FAQ by instructors

*Q: How large a class can it be? I teach large classes.*

**TODO:** to write

*Q: What is the budget for the robot?*

**TODO:** to write

*Q: I want to teach a Duckietown class. How do I get started?*

Please get in touch with us at duckietown@mit.edu. We will be happy to get you started and sign you up to the Duckietown instructors mailing list.

*Q: Why the duckies?*

Compared to other educational robotics projects, the presence of the duckies is what makes this project stand out. Why the duckies?

We want to present robotics in an accessible and friendly way.

**TODO:** copy usual discussion from somewhere else.

**TODO:** add picture of kids with Duckiebots.

# Part B
# Duckumentation documentation

UNIT B-1

# Contributing to the documentation

## 1.1. Where the documentation is

All the documentation is in the repository `duckietown/duckuments`.

The documentation is written as a series of small files in Markdown format.

It is then processed by a series of scripts to create this output:

- a publication-quality PDF;
- an online HTML version, split in multiple pages;
- a one-page version.

## 1.2. Editing links

The simplest way to contribute to the documentation is to click any of the "✎" icons next to the headers.

They link to the "edit" page in Github. There, one can make and commit the edits in only a few seconds.

## 1.3. Installing the documentation system

In the following, we are going to assume that the documentation system is installed in `~/duckuments`. However, it can be installed anywhere.

We are also going to assume that you have setup a Github account with working public keys.

➡ Basic SSH config.

➡ Key pair creation.

➡ Adding public key on Github.

We are also going to assume that you have installed the `duckietown/software` in `~/duckietown`.

### 1) Dependencies (Ubuntu 16.04)

On Ubuntu 16.04, these are the dependencies to install:

```
$ sudo apt install libxml2-dev libxslt1-dev
$ sudo apt install libffi6 libffi-dev
$ sudo apt install python-dev python-numpy python-matplotlib
$ sudo apt install virtualenv
$ sudo apt install bibtex2html pdftk
$ sudo apt install imagemagick
```

### 2) Download the duckuments repo

Download the `duckietown/duckuments` repository in the `~/duckuments` directory:

```
$ git lfs clone --depth 100 git@github.com:duckietown/duckuments ~/duckuments
```

Here, note we are using `git lfs clone` – it's much faster, because it downloads the Git LFS files in parallel.

If it fails, it means that you do not have Git LFS installed. See Unit J-28 - Git LFS.

The command `--depth 100` tells it we don't care about the whole history.

### 3) Setup the virtual environment

Next, we will create a virtual environment using inside the `~/duckuments` directory. Make sure you are running Python 2.7.x. Python 3.x is not supported at the moment.

Change into that directory:

```
$ cd ~/duckuments
```

Create the virtual environment using `virtualenv`:

```
$ virtualenv --system-site-packages deploy
```

Other distributions: In other distributions you might need to use `venv` instead of `virtualenv`.

Activate the virtual environment:

```
$ source ~/duckuments/deploy/bin/activate
```

### 4) Setup the `mcdp` external repository

Make sure you are in the directory:

```
$ cd ~/duckuments
```

Clone the `mcdp` external repository, with the branch `duckuments`.

```
$ git clone -b duckuments git@github.com:AndreaCensi/mcdp
```

Install it and its dependencies:

```
$ cd ~/duckuments/mcdp
$ python setup.py develop
```

> **Note:** If you get a permission error here, it means you have not properly activated the virtual environment.

Other distributions: If you are not on Ubuntu 16, depending on your system, you might need to install these other dependencies:

```
$ pip install numpy matplotlib
```

You also should run:

```
$ pip install -U SystemCmd==2.0.0
```

## 1.4. Compiling the documentation (updated Sep 12)

**Check before you continue**

Make sure you have deployed and activated the virtual environment. You can check this by checking which `python` is active:

```
$ which python
/home/user/duckuments/deploy/bin/python
```

To compile the master versions of the docs, run:

```
$ make master-clean master
```

To see the result, open the file

```
./duckuments-dist/master/duckiebook/index.html
```

If you want to do incremental compilation, you can omit the `clean` and just use:

```
$ make master
```

This will be faster. However, sometimes it might get confused. At that point, do `make master-clean`.

### 1) Compiling the Fall 2017 version only (introduced Sep 12)

To compile the Fall 2017 versions of the docs, run:

```
$ make fall2017-clean fall2017
```

To see the result, open the file

```
./duckuments-dist/master/duckiebook/index.html
```

For incremental compilation, use:

```
$ make fall2017
```

2) Single-file compilation

There is also the option to compile one single file.

To do this, use:

```
$ ./compile-single path to .md file
```

This is the fastest way to see the results of the editing; however, there are limitations:

- no links to other sections will work.
- not all images might be found.

## 1.5. The workflow to edit documentation (updated Sep 12)

This is the basic workflow:

1. Create a branch called *yourname*`-branch` in the `duckuments` repository.
2. Edit the Markdown in the *yourname*`-branch` branch.
3. Run `make master` to make sure it compiles.
4. Commit the Markdown and push on the *yourname*`-branch` branch.
5. Create a pull request.
6. Tag the group `duckietown/gardeners`.

➡ Create a pull request from the command-line using hub.

## 1.6. Reporting problems

First, see the section Unit B-7 - Markduck troubleshooting for common problems and their resolution.

Please report problems with the duckuments using the duckuments issue tracker. If it is urgent, please tag people (Andrea); otherwise these are processed in batch mode every few days.

If you have a problem with a generated PDF, please attach the offending PDF.

If you say something like "This happens for Figure 3", then it is hard to know which figure you are referencing exactly, because numbering changes from commit to commit.

If you want to refer to specific parts of the text, please commit all your work on your branch, and obtain the name of the commit using the following commands:

```
$ git -C ~/duckuments rev-parse HEAD      # commit for duckuments
$ git -C ~/duckuments/mcdp rev-parse HEAD # commit for mcdp
```

UNIT B-2

# Basic Markduck guide

The Duckiebook is written in Markduck, a Markdown dialect.

It supports many features that make it possible to create publication-worthy materials.

## 2.1. Markdown

The Duckiebook is written in a Markdown dialect.

➡ A tutorial on Markdown.

## 2.2. Variables in command lines and command output

Use the syntax "`![name]`" for describing the variables in the code.

*example*

For example, to obtain:

```
$ ssh robot name.local
```

Use the following:

```
For example, to obtain:

    $ ssh ![robot name].local
```

Make sure to quote (with 4 spaces) all command lines. Otherwise, the dollar symbol confuses the LaTeX interpreter.

## 2.3. Character escapes

Use the string "`$`" to write the dollar symbol "`$`", otherwise it gets confused with LaTeX math materials. Also notice that you should probably use "USD" to refer to U.S. dollars.

Other symbols to escape are shown in Table 2.1.

TABLE 2.1. SYMBOLS TO ESCAPE

| | |
|---|---|
| use `$` | instead of `$` |
| use ``` | instead of `` ` `` |
| use `&#lt;` | instead of `<` |
| use `&#gt;` | instead of `>` |

## 2.4. Keyboard keys

Use the `kbd` element for keystrokes.

*example*

For example, to obtain:

Press a then Ctrl-C.

use the following:

```
Press <kbd>a</kbd> then <kbd>Ctrl</kbd>-<kbd>C</kbd>.
```

## 2.5. Figures

For any element, adding an attribute called `figure-id` with value `fig:`*figure ID* or `tab:`*table ID* will create a figure that wraps the element.

For example:

```
<div figure-id="fig:figure ID">
    figure content
</div>
```

It will create HMTL of the form:

```
<div id='fig:code-wrap' class='generated-figure-wrap'>
    <figure id='fig:figure ID' class='generated-figure'>
        <div>
            figure content
        </div>
    </figure>
</div>
```

To add a caption, add an attribute `figure-caption`:

```
<div figure-id="fig:figure ID" figure-caption="This is my caption">
    figure content
</div>
```

Alternatively, you can put anywhere an element `figcaption` with ID *figure id*`:caption`:

```
<element figure-id="fig:figure ID">
    figure content
</element>

<figcaption id='fig:figure ID:caption'>
    This the caption figure.
</figcaption>
```

To refer to the figure, use an empty link:

```
Please see [](#fig:figure ID).
```

The code will put a reference to “Figure XX”.

## 2.6. Subfigures

You can also create subfigures, using the following syntax.

```
<div figure-id="fig:big">
    <figcaption>Caption of big figure</figcaption>

    <div figure-id="subfig:first" figure-caption="Caption 1">
        <p style='width:5em;height:5em;background-color:#eef'>first subfig</p>
    </div>

    <div figure-id="subfig:second" figure-caption="Caption 2">
        <p style='width:5em;height:5em;background-color:#fee'>second subfig</p>
    </div>
</div>
```

This is the result:

first subfig

(a) Caption 1

second subfig

(b) Caption 2

Figure 2.1. Caption of big figure

By default, the subfigures are displayed one per line.

To make them flow horizontally, add `figure-class="flow-subfigures"` to the external figure `div`. Example:

first subfig

(a) Caption 1

second subfig

(b) Caption 2

Figure 2.2. Caption of big figure

## 2.7. Shortcut for tables

The shortcuts `col2`, `col3`, `col4`, `col5` are expanded in tables with 2, 3, 4 or 5 columns. The following code:

```
<col2 figure-id="tab:mytable" figure-caption="My table">
    <span>A</span>
    <span>B</span>
    <span>C</span>
    <span>D</span>
</col2>
```

gives the following result:

TABLE 2.2. MY TABLE

| | |
|---|---|
| A | B |
| C | D |

### 1) labels-row1 and labels-row1

Use the classes `labels-row1` and `labels-row1` to make pretty tables like the following.

`labels-row1`: the first row is the headers.

`labels-col1`: the first column is the headers.

TABLE 2.3. USING CLASS="LABELS-COL1"

| | | | |
|---|---|---|---|
| **header A** | B | C | 1 |
| **header D** | E | F | 2 |
| **header G** | H | I | 3 |

TABLE 2.4. USING CLASS="LABELS-ROW1"

| header A | header B | header C |
|---|---|---|
| D | E | F |
| G | H | I |
| 1 | 2 | 3 |

## 2.8. Linking to documentation

### 1) Establishing names of headers

You give IDs to headers using the format:

```
### header title {#topic ID}
```

For example, for this subsection, we have used:

```
### Establishing names of headers {#establishing}
```

With this, we have given this header the ID "`establishing`".

### 2) How to name IDs - and why it's not automated

Some time ago, if there was a section called

```
## My section
```

then it would be assigned the ID "my-section".

This behavior has been removed, for several reasons.

One is that if you don't see the ID then you will be tempted to just change the name:

```
## My better section
```

and silently the ID will be changed to "my-better-section" and all the previous links will be invalidated.

The current behavior is to generate an ugly link like "autoid-209u31j".

This will make it clear that you cannot link using the PURL if you don't assign an ID.

Also, I would like to clarify that all IDs are *global* (so it's easy to link stuff, without thinking about namespaces, etc.).

Therefore, please choose descriptive IDs, with at least two IDs.

E.g. if you make a section called

```
## Localization  {#localization}
```

that's certainly a no-no, because "localization" is too generic.

✔

```
## Localization {#intro-localization}
```

Also note that you don't *need* to add IDs to everything, only the things that people could link to. (e.g. not subsubsections)

### 3) Linking from the documentation to the documentation

You can use the syntax:

```
[](#topic ID)
```

to refer to the header.

You can also use some slightly more complex syntax that also allows to link to only the name, only the number or both (Table 2.5).

TABLE 2.5. SYNTAX FOR REFERRING TO SECTIONS.

```
See [](#establishing).
```

See Subsection 2.8.1 - Establishing names of headers

```
See <a class="only_name" href="#establishing"></a>.
```

See Establishing names of headers.

```
See <a class="only_number" href="#establishing"></a>.
```

See 2.8.1.

```
See <a class="number_name" href="#establishing"></a>.
```

See Subsection 2.8.1 - Establishing names of headers.

### 4) Linking to the documentation from outside the documentation

You are encouraged to put links to the documentation from the code or scripts.
To do so, use links of the form:

```
http://purl.org/dth/topic ID
```

Here "`dth`" stands for "Duckietown Help". This link will get redirected to the corresponding document on the website.
For example, you might have a script whose output is:

```
$ rosrun mypackage myscript
Error. I cannot find the scuderia file.
See: http://purl.org/dth/scuderia
```

When the user clicks on the link, they will be redirected to Section 4.2 - The "scuderia" (vehicle database).

UNIT B-3

# Special paragraphs and environments

## 3.1. Special paragraphs tags

The system supports parsing of some special paragraphs.

> **Note:** some of these might be redundant and will be eliminated. For now, I am documenting what is implemented.

### 1) Special paragraphs must be separated by a line

A special paragraph is marked by a special prefix. The list of special prefixes is given in the next section.

There must be an empty line before a special paragraph; this is because in Markdown a paragraph starts only after an empty line.

This is checked automatically, and the compilation will abort if the mistake is found.

For example, this is invalid:

```
See: this book
See: this other book
```

This is correct:

```
See: this book

See: this other book
```

Similarly, this is invalid:

```
Author: author
Maintainer: maintainer
```

and this is correct:

```
Author: author

Maintainer: maintainer
```

### 2) Todos, task markers

```
TODO: todo
```

**TODO:** todo

```
TOWRITE: towrite
```

**To write:** towrite

```
Task: task
```

**Task:** task

```
Assigned: assigned
```

> Assigned to: assigned

## 3) Notes and remarks

```
Remark: remark
```

> **Remark:** remark

```
Note: note
```

> **Note:** note

```
Warning: warning
```

**Warning:** warning

## 4) Troubleshooting

```
Symptom: symptom
```

> **Symptom:** symptom

```
Resolution: resolution
```

**Resolution:** resolution

## 5) Guidelines

```
Bad: bad
```

✖ bad

```
Better: better
```

✔ better

## 6) Questions and answers

```
Q: question
```

*Q: question*

```
A: answer
```

**Answer:** answer

## 7) Authors, maintainers, Point of Contact

```
Maintainer: maintainer
```

**Maintainer:** maintainer

```
Author: author
```

**Author:** author

```
Point of Contact: Point of Contact name
```

**Point of contact:** Point of Contact name

```
Slack channel: slack channel name
```

**Slack channel:** slack channel name

## 8) References

```
See: see
```

➡ see

```
Reference: reference
```

➡ reference

```
Requires: requires
```

**Requires:** requires

```
Results: results
```

**Results:** results

```
Next steps: next steps
```

**Next:** next steps

```
Recommended: recommended
```

**Recommended:** recommended

```
See also: see also
```

☀ see also

## 3.2. Other `div` environments

For these, note the rules:

- You must include `markdown="1"`.
- There must be an empty line after the first `div` and before the closing `/div`.

### 1) Example usage

```
<div class='example-usage' markdown="1">

This is how you can use `rosbag`:

    $ rosbag play log.bag

</div>
```

*example*

This is how you can use `rosbag`:

```
$ rosbag play log.bag
```

2) Check

```
<div class='check' markdown="1">

Check that you didn't forget anything.

</div>
```

**Check before you continue**
Check that you didn't forget anything.

3) Requirements

```
<div class='requirements' markdown="1">

List of requirements at the beginning of setup chapter.

</div>
```

KNOWLEDGE AND ACTIVITY GRAPH

List of requirements at the beginning of setup chapter.

## 3.3. Notes and questions

There are three environments: "comment", "question", "doubt", that result in boxes that can be expanded by the user.

These are the one-paragraph forms:

```
Comment: this is a comment on one paragraph.
```

+ comment
this is a comment on one paragraph.

```
Question: this is a question on one paragraph.
```

+ question
this is a question on one paragraph.

```
Doubt: I have my doubts on one paragraph.
```

+ doubt
I have my doubts on one paragraph.

These are the multiple-paragraphs forms:

```
<div class='comment' markdown='1'>
A comment...

A second paragraph...
</div>
```

+ comment

A comment...

A second paragraph...

```
<div class='question' markdown='1'>
A question...

A second paragraph...
</div>
```

+ question

A question...

A second paragraph...

```
<div class='doubt' markdown='1'>
A question...

Should it not be:

    $ alternative command

A second paragraph...
</div>
```

+ doubt

A question...

Should it not be:

```
$ alternative command
```

A second paragraph...

UNIT B-4

# Using LaTeX constructs in documentation

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Working knowledge of LaTeX.

## 4.1. Embedded LaTeX

You can use LaTeX math, environment, and references. For example, take a look at

$$x^2 = \int_0^t f(\tau)\ \text{d}\tau$$

or refer to Proposition 1 - Proposition example.

**Proposition 1.** (Proposition example) This is an example proposition: $2x = x + x$.

The above was written as in Listing 4.1.

```
You can use $\LaTeX$ math, environment, and references.
For example, take a look at

\[
    x^2 = \int_0^t f(\tau)\ \text{d}\tau
\]

or refer to [](#prop:example).

\begin{proposition}[Proposition example]\label{prop:example}
This is an example proposition: $2x = x + x$.
\end{proposition}
```

Listing 4.1. Use of LaTeX code.

For the LaTeX environments to work properly you *must* add a `\label` declaration inside. Moreover, the label must have a prefix that is adequate to the environment. For example, for a proposition, you must insert `\label{prop:`*name*`}` inside.

The following table shows the list of the LaTeX environments supported and the label prefix that they need.

TABLE 4.1. LATEX ENVIRONMENTS AND LABEL PREFIXES

| | |
|---|---|
| definition | def:*name* |
| proposition | prop:*name* |
| remark | rem:*name* |
| problem | prob:*name* |
| theorem | thm:*name* |
| lemma | lem:*name* |

Examples of all environments follow.

*example*

```
\begin{definition}   \label{def:lorem}
Lorem
\end{definition}
```

**Definition 1.** Lorem

```
\begin{proposition}   \label{prop:lorem}
Lorem
\end{proposition}
```

**Proposition 2.** Lorem

```
\begin{remark}   \label{rem:lorem}
Lorem
\end{remark}
```

**Remark 1.** Lorem

```
\begin{problem}   \label{prob:lorem}
Lorem
\end{problem}
```

**Problem 1.** Lorem

```
\begin{example}   \label{exa:lorem}
Lorem
\end{example}
```

**Example 1.** Lorem

```
\begin{theorem}   \label{thm:lorem}
Lorem
\end{theorem}
```

**Theorem 1.** Lorem

```
\begin{lemma}   \label{lem:lorem}
Lorem
\end{lemma}
```

**Lemma 1.** Lorem

```
I can also refer to all of them:
[](#def:lorem),
[](#prop:lorem),
[](#rem:lorem),
[](#prob:lorem),
[](#exa:lorem),
[](#thm:lorem),
[](#lem:lorem).
```

I can also refer to all of them: Definition 1, Proposition 2, Remark 1, Problem 1, Example 1, Theorem 1, Lemma 1.

## 4.2. LaTeX equations

We can refer to equations, such as (1):

$$2a = a + a \tag{1}$$

This uses `align` and contains (2) and (3).

$$\begin{aligned} a &= b && (2) \\ &= c && (3) \end{aligned}$$

```
We can refer to equations, such as \eqref{eq:one}:

\begin{equation}
    2a = a + a          \label{eq:one}
\end{equation}

This uses `align` and contains  \eqref{eq:two} and \eqref{eq:three}.

\begin{align}
    a &= b       \label{eq:two} \\
      &= c       \label{eq:three}
\end{align}
```

Note that referring to the equations is done using the syntax `\eqref{eq:name}`, rather than `[](#eq:name)`.

## 4.3. LaTeX symbols

The LaTeX symbols definitions are in a file called `docs/symbols.tex`.

Put all definitions there; if they are centralized it is easier to check that they are coherent.

## 4.4. Bibliography support

You need to have installed `bibtex2html`.

The system supports Bibtex files.

Place `*.bib` files anywhere in the directory.

Then you can refer to them using the syntax:

```
[](#bib:bibtex ID)
```

For example:

```
Please see [](#bib:siciliano07handbook).
```

Will result in:

> Please see [3].

## 4.5. Embedding Latex in Figures through SVG

KNOWLEDGE AND ACTIVITY GRAPH

> **Requires:** In order to compile the figures into PDFs you need to have Inkscape installed. Instructions to download and install Inkscape are here.

To embed latex in your figures, you can add it directly to a file and save it as *filename*`.svg` file and save anywhere in the `/docs` directory.

You can run:

```
$ make process-svg-figs
```

And the SVG file will be compiled into a PDF figure with the LaTeX commands properly interpreted.

You can then include the PDF file in a normal way (Section 2.5 - Figures) using *filename*`.pdf` as the filename in the `<img>` tag.

(a) Image saved as svg

(b) Image as PDF after processing

Figure 4.1. Embedding LaTeX in images

It can take a bit of work to get the positioning of the code to appear properly on the figure.

# UNIT B-5
# Advanced Markduck guide

## 5.1. Embedding videos

It is possible to embed Vimeo videos in the documentation.

> **Note:** Do not upload the videos to your personal Vimeo account; they must all be posted to the Duckietown Engineering account.

This is the syntax:

```
<dtvideo src="vimeo:vimeo ID"/>
```

*example*

For example, this code:

```
<div figure-id="fig:example-embed">
    <figcaption>Cool Duckietown by night</figcaption>
    <dtvideo src="vimeo:152825632"/>
</div>
```

produces this result:

Figure 5.1. Cool Duckietown by night

Depending on the output media, the result will change:

- On the online book, the result is that a player is embedded.
- On the e-book version, the result is that a thumbnail is produced, with a link to the video;
- On the dead-tree version, a thumbnail is produced with a QR code linking to the video (TODO).

## 5.2. `move-here` tag

If a file contains the tag `move-here`, the fragment pointed by the `src` attribute is moved at the place of the tag.

This is used for autogenerated documentation.

Syntax:

```
# Node `node`

<move-here src='#package-node-autogenerated'/>
```

## 5.3. Comments

You can insert comments using the HTML syntax for comments: any text between "`<!-`" and "`->`" is ignored.

```
# My section

<!-- this text is ignored -->

Let's start by...
```

## 5.4. Referring to Github files

You can refer to files in the repository by using:

```
See [this file](github:org=org,repo=repo,path=path,branch=branch).
```

The available keys are:

- `org` (required): organization name (e.g. `duckietown`);
- `repo` (required): repository name (e.g. `Software`);
- `path` (required): the filename. Can be just the file name or also include directories;
- `branch` (optional) the repository branch; defaults to `master`;

For example, you can refer to [the file `pkg_name/src/subscriber_node.py`]() by using the following syntax:

```
See [this file](github:org=duckietown,repo=Software,path=pkg_name/src/subscriber_node.py)
```

You can also refer to a particular line:

This is done using the following parameters:

- `from_text` (optional): reference the first line containing the text;
- `from_line` (optional): reference the line by number;

For example, you can refer to [the line containing "Initialize"]() of `pkg_name/src/subscriber_node.py` by using the following syntax:

```
For example, you can refer to [the line containing "Initialize"][link2]
of `pkg_name/src/subscriber_node.py` by using the following syntax:

[link2]: github:org=duckietown,repo=Software,path=pkg_name/src/
subscriber_node.py,from_text=Initialize
```

You can also reference a range of lines, using the parameters:

- `to_text` (optional): references the final line, by text;
- `to_line` (optional): references the final line, by number.

You cannot give `from_text` and `from_line` at the same time. You cannot give a `to_...` without the `from_...`.

For example, this link refers to a range of lines: click it to see how Github highlights the lines from "Initialize" to "spin".

This is the source of the previous paragraph:

```
For example, [this link refers to a range of lines][interval]: click it to see how Github
highlights the lines from "Initialize" to "spin".

[interval]: github:org=duckietown,repo=Software,path=pkg_name/src/
subscriber_node.py,from_text=Initialize,to_text=spin
```

## 5.5. Putting code from the repository in line

In addition to referencing the files, you can also copy the contents of a file inside the documentation.

This is done by using the tag `display-file`.

For example, you can put a copy of `pkg_name/src/subscriber_node.py` using:

```
<display-file src="
    github:org=duckietown,
    repo=Software,
    path=pkg_name/src/subscriber_node.py
"/>
```

and the result is the following automatically generated listing:

```python
#!/usr/bin/env python
import rospy

# Imports message type
from std_msgs.msg import String

# Define callback function
def callback(msg):
    s = "I heard: %s" % (msg.data)
    rospy.loginfo(s)

# Initialize the node with rospy
rospy.init_node('subscriber_node', anonymous=False)

# Create subscriber
subscriber = rospy.Subscriber("topic", String, callback)

# Runs continuously until interrupted
rospy.spin()
```

Listing 5.2. subscriber_node.py

If you use the `from_text` and `to_text` (or `from_line` and `to_line`), you can actually display part of a file. For example:

```
<display-file src="
    github:org=duckietown,
    repo=Software,
    path=pkg_name/src/subscriber_node.py,
    from_text=Initialize,
    to_text=spin
    "/>
```

creates the following automatically generated listing:

```python
# Initialize the node with rospy
rospy.init_node('subscriber_node', anonymous=False)

# Create subscriber
subscriber = rospy.Subscriber("topic", String, callback)

# Runs continuously until interrupted
rospy.spin()
```

Listing 5.3. subscriber_node.py

# Unit B-6
# *Compiling the PDF version

This part describes how to compile the PDF version.

> **Note:** The dependencies below are harder to install. If you don't manage to do it, then you only lose the ability to compile the PDF. You can do `make compile` to compile the HTML version, but you cannot do `make compile-pdf`.

## 6.1. Installing nodejs

Ensure the latest version (>6) of `nodejs` is installed.

Run:

```
$ nodejs --version
6.xx
```

If the version is 4 or less, remove `nodejs`:

```
$ sudo apt remove nodejs
```

Install `nodejs` using the instructions at this page.

Next, install the necessary Javascript libraries using `npm`:

```
$ cd $DUCKUMENTS
$ npm install MathJax-node jsdom@9.3 less
```

### 1) Troubleshooting nodejs installation problems

The only pain point in the installation procedure has been the installation of `nodejs` packages using `npm`. For some reason, they cannot be installed globally (`npm install -g`).

Do **not** use `sudo` for installation. It will cause problems.

If you use `sudo`, you probably have to delete a bunch of directories, such as: `~/duckuments/node_modules`, `~/.npm`, and `~/.node_modules`, if they exist.

## 6.2. Installing Prince

Install PrinceXML from this page.

## 6.3. Installing fonts

Copy the `~/duckuments/fonts` directory in `~/.fonts`:

```
$ mkdir -p ~/.fonts    # create if not exists
$ cp -R ~/duckuments/fonts ~/.fonts
```

and then rebuild the font cache using:

```
$ fc-cache -fv
```

## 6.4. Compiling the PDF

To compile the PDF, use:

```
$ make compile-pdf
```

This creates the file:

```
./duckuments-dist/master/duckiebook.pdf
```

# Unit B-7
# Markduck troubleshooting

## 7.1. Changes don't appear on the website

For these issues, see Unit B-8 - The Duckuments bot.

## 7.2. Troubleshooting errors in the compilation process

> **Symptom:** "Invalid XML"

**Resolution:** "Markdown" doesn't mean that you can put anything in a file. Except for the code blocks, it must be valid XML. For example, if you use "`>`" and "`<`" without quoting, it will likely cause a compile error.

> **Symptom:** "Tabs are evil"

**Resolution:** Do not use tab characters. The error message in this case is quite helpful in telling you exactly where the tabs are.

> **Symptom:** The error message contains `ValueError: Suspicious math fragment 'KEYMATHS000END-KEY'`

**Resolution:** You probably have forgotten to indent a command line by at least 4 spaces. The dollar in the command line is now being confused for a math formula.

## 7.3. Common mistakes with Markdown

Here are some common mistakes encountered.

### 1) Not properly starting a list

There must be an empty line before the list starts.

This is correct:

```
I want to learn:

- robotics
- computer vision
- underwater basket weaving
```

This is incorrect:

```
I want to learn:
- robotics
- computer vision
- underwater basket weaving
```

and it will be rendered as follows:

I want to learn: - robotics - computer vision - underwater basket weaving

# Unit B-8
# The Duckuments bot

**Note:** This is an advanced section mainly for Liam.

## 8.1. Documentation deployment

The book is published to a different repository called `duckuments-dist` and from there published as `book.duckietown.org`.

## 8.2. Understand what's going on

There is a bot, called `frankfurt.co-design.science`, which is an AWS machine (somewhere in Frankfurt).

Every minute, it tries to do the following:

1. It checks out the last version of `mcdp`;
2. It checks out the last version of `duckuments`;
3. It compiles the various versions;
4. It uploads the results to a repository called `duckuments-dist`.

This process takes 4 minutes for an incremental change, and about 10-15 minutes for a big change, such as a change in headers IDs, which implies re-checking all cross-references.

## 8.3. Logging

There are logs you can access to see what's going on.

The high-level compilation log tells you in what phase of the cycle the bot is. Scroll to the bottom.

Ideally what you want to see is something like the following:

```
Starting
Mon Sep 11 10:49:04 CEST 2017
  succeded html
  succeded fall 2017
  succeded upload
  succeded split
  succeded html upload
  succeded PDF
  succeded PDF upload
Mon Sep 11 10:54:21 CEST 2017
Done.
```

This shows that the compilation took 5 minutes.

Every two hours you will see something like this:

```
automatic-compile-cleanup killing everything
```

and the next iteration will take longer because it starts from scratch.

The last log is a live version of the compilation log. This might not be tremendously informative because it is very verbose.

## 8.4. Debugging Github Pages problems

Sometimes, it's Github Pages that lags behind.

To check this, the bot also makes available the compilation output as a website called `book2.duckietown.org`. You can take any URL starting with `book.duckietown.org`, put `book2`, and you will see what is on the server.

This can identify if the problem is Github.

# UNIT B-9
# Documentation style guide

This chapter describes the conventions for writing the technical documentation.

## 9.1. General guidelines for technical writing

The following holds for all technical writing.

- The documentation is written in correct English.
- Do not say "should" when you mean "must". "Must" and "should" have precise meanings and they are not interchangeable. These meanings are explained in this document.
- "Please" is unnecessary in technical documentation.
  - ✖ "Please remove the SD card."
  - ✔ "Remove the SD card".
- Do not use colloquialisms or abbreviations.
  - ✖ "The pwd is `ubuntu`."
  - ✔ "The password is `ubuntu`."
  - ✖ "To create a ROS pkg..."
  - ✔ "To create a ROS package..."
- Python is capitalized when used as a name.
  - ✖ "If you are using python..."
  - ✔ "If you are using Python..."
- Do not use emojis.
- Do not use ALL CAPS.
- Make infrequent use of **bold statements**.
- Do not use exclamation points.

## 9.2. Style guide for the Duckietown documentation

- The English version of the documentation is written in American English. Please note that your spell checker might be set to British English.

✖ behaviour

✔ behavior

✖ localisation

✔ localization

- It's ok to use "it's" instead of "it is", "can't" instead of "cannot", etc.
- All the filenames and commands must be enclosed in code blocks using Markdown backticks.
  - ✖ "Edit the ~/.ssh/config file using vi."
  - ✔ "Edit the `~/.ssh/config` file using `vi`."

- Ctrl-C, `ssh` etc. are not verbs.
  - ✖ "Ctrl-C from the command line".
  - ✔ "Use Ctrl-C from the command line".
- Subtle humor and puns about duckies are encouraged.

## 9.3. Writing command lines

Use either "`laptop`" or "`duckiebot`" (not capitalized, as a hostname) as the prefix for the command line.

For example, for a command that is supposed to run on the laptop, use:

```
laptop $ cd ~/duckietown
```

It will become:

```
$ cd ~/duckietown
```

For a command that must run on the Duckiebot, use:

```
duckiebot $ cd ~/duckietown
```

It will become:

```
$ cd ~/duckietown
```

If the command is supposed to be run on both, omit the hostname:

```
$ cd ~/duckietown
```

## 9.4. Frequently misspelled words

- "Duckiebot" is always capitalized.
- Use "Raspberry Pi", not "PI", "raspi", etc.
- These are other words frequently misspelled: 5 GHz WiFi

## 9.5. Other conventions

When the user must edit a file, just say: "edit `/this/file`".

Writing down the command line for editing, like the following:

```
$ vi /this/file
```

is too much detail.

(If people need to be told how to edit a file, Duckietown is too advanced for them.)

## 9.6. Troubleshooting sections

Write the documentation as if every step succeeds.

Then, at the end, make a “Troubleshooting” section.

Organize the troubleshooting section as a list of symptom/resolution.

The following is an example of a troubleshooting section.

### 1) Troubleshooting

**Symptom:** This strange thing happens.

**Resolution:** Maybe the camera is not inserted correctly. Remove and reconnect.

**Symptom:** This other strange thing happens.

**Resolution:** Maybe the plumbus is not working correctly. Try reformatting the plumbus.

# UNIT B-10
# Learning in Duckietown

Assigned to: Jacopo

**Note:** This chapter is a draft.

- ✖ We do not refer to teachers and students
- ✔ We refer to learners. Instructors are learners ahead in the learning curve in comparison to other learners
- ✖ We don't examinate
- ✔ We assess competences

## 10.1. The learning feedback loop

A relevant contribution of modern educational theory is recognizing the importance of feedback in the learning process ().

We love feedback, as it stands at the foundation of control systems theory.

Figure 10.1. The learning loop.

Here, we provide definitions of the key elements in a learning feedback loop, and analogies to their control systems counterparts.

### 1) Intended Learning Outcomes

**Definition 2.** (Intended Learning Outcome)
An intended learning outcome is a desired, *measurable*, output of the learning process.

Intended learning outcomes are:

- the starting point in the construction of a learning activity ([1]),
- more effective when formulated with active verbs,
- the equivalent of reference trajectories, or setpoints, in control systems. They represent the ideal output of the controlled system.

*example*

- ✖ Students will understand robotics
- ✔ Students each build a Duckiebot, implement software and produce demos

### 2) Learning Activities

**Definition 3.** (Learning Activities)
Methods chosen by the instructor to ease the learners achievement of the intended learning outcomes.

*Active learning* practices ([1]) have been shown to improve learning.

*example*

- ✖ Instructor explains at the blackboard
- ✔ Learners work in groups to achieve objectives

Learning activities are analogous to the the output of the controller (instructor), or input to the system (learners). They have to meet the requirements imposed by external constraints, not shown in

### 3) Assessment

**Definition 4.** (Assessment)
An assessment is a procedure to quantify the level of fulfillment of learning outcomes.

An assessment is analogous to a sensor in a control system loop. It measures the system's output.

*example*

Examples of assessments include: quizzes, colloquia, produced documentation, homework, etc.

## 10.2. Knowledge, Skills and Competences

Here, we provide definitions for knowledge, skills and competences, in addition to describing their relationships (Figure 10.2).

Figure 10.2. The relationship between Knowledge, Skills and Competences.

### 1) Knowledge

**Definition 5.** (Knowledge)
Theoretical facts and information aimed at enabling understanding, and generating or improving *skills*.

*example* Bayesian inference is handy piece of knowledge when doing robotics.

### 2) Practice

**Definition 6.** (Practice)
Practical procedures aimed at generating or improving *skills*, either directly or indirectly by improving knowledge.

*example* Exercises and proofs can be used to practice different skills.

### 3) Skills

**Definition 7.** (Skills)
A proficiency, facility or dexterity that enables carrying out a function. Skills stem from *knowledge*, *practice* and/or aptitude. Skills can be clustered in cognitive, technical and interpersonal, respectively relating to ideas, things and people.

*example*

Analyzing tradeoffs between performances and constraints is a critical cognitive skill for robotics.

Python language is a useful technical skill in robotics.

Public speaking is a valuable interpersonal skill useful beyond robotics.

In Duckietown we formalize didactic indivisible units, or *atoms*, aimed at improving skills through knowledge and practice. Knowledge atoms are listed in XXX. We define as practice atoms:

**TODO:** add general reference to all learning atmos, folder atoms_30_learning_material

**Definition 8.** (Exercise)
An exercise is a practice atom aimed at improving technical skills. Exercises are listed in XXX.

Exercises are targeted to different "things" to which technical skills are related. They may be mathematical exercises aimed at practicing a method, or they may be coding exercises aimed at practicing resolutions of hardware implementation challenges.

**Definition 9.** (Proof)
A proof is a practice atom aimed at improving cognitive skills.

*example* Deriving the Kalman filter equations helps practice the *idea* that there is no better approach to state estimation for linear time invariant systems, with "well behaved" measurement and process noises.

### 4) Competences

**Definition 10.** (Competences)
Set of skills and/or knowledge that leads to superior performance in carrying out a function. Competences must be *measurable*.

Competences are desirable intended learning outcomes, and typically address the *how* of the learning process.

*example*

Programming is a competence. It requires a skill, e.g., Python, and knowledge, e.g., Bayesian inference, to know what to code. Practice can help improve knowledge or hone skills.

UNIT B-11

# Knowledge graph

**Note:** This chapter describes something that is not implemented yet.

## 11.1. Formalization

### 1) Atoms

**Definition 11.** (Atom) An *atom* is a concrete resource (text, video) that is the smallest unit that is individually addressable. It is indivisible.

Each atom as a type, as follows:

```
text
  text/theory
  text/setup
  text/demo
  text/exercise
  text/reference
  text/instructor-guide
  text/quiz
video
  video/lecture
  video/instructable
  video/screencast
  video/demo
```

### 2) Semantic graph of atoms

Atoms form a directed graph, called "semantic graph".

Each node is an atom.

The graph has four different types of edges:

- "**Requires**" edges describe a strong dependency: "You need to have done this. Otherwise it will not work."
- "**Recommended**" edges describe a weaker dependency; it is not strictly necessary to have done that other thing, but it will significantly improve the result of this.
- "**Reference**" edges describe background information. "If you don't know / don't remember, you might want to see this"
- "**See also**" edges describe interesting materials for the interested reader. Completely optional; it will not impact the result of the current procedure.

### 3) Modules

A "module" is an abstraction from the point of view of the teacher.

**Definition 12.** (Module) A *module* is a directed graph, where the nodes are either atoms or other modules, and the edges can be of the four types described in Subsec-

tion 11.1.2 - Semantic graph of atoms.

Because modules can contain other modules, they allow to describe hierarchical contents. For example, a class module is a module that contains other modules; a "degree" is a module that contains "class" modules, etc.

Modules can overlap. For example, a "Basic Object Detection" and an "Advanced Object Detection" module might have a few atoms in common.

## 11.2. Atoms properties

Each atom has the following **properties**:

- An **ID** (alphanumeric + `-` and '_'). The ID is used for cross-referencing. It is the same in all languages.
- A **type**, as above.

There might be different **versions** of each atom. This is used primarily for dealing with translations of texts, different representations of the same image, Powerpoint vs Keynote, etc.

A version is a tuple of attributes.

The attributes are:

- **Language**: A language code, such as `en-US` (default), `zh-CN`, etc.
- **Mime type**: a MIME type.

Each atom version has:

- A human-readable **title**.
- A human-readable **summary** (1 short paragraph).

### 1) Status values (updated Sep 12)

Each document has a **status** value.

The allowed values are described in Table 11.1.

TABLE 11.1. STATUS CODES

| | |
|---|---|
| `draft` | We just started working on it, and it is not ready for public consumption. |
| `beta` | Early reviewers should look at it now. |
| `ready` | The document is ready for everybody. |
| `recently-updated` | The document has been recently updated (less than 1 week) |
| `to-update` | A new pass is needed on this document, because it is not up to date anymore. |
| `deprecated` | The document is ready for everybody. |

## 11.3. Markdown format for text-like atoms

For the text-like resources, they are described in Markdown files.

The name of the file does not matter.

All files are encoded in UTF-8.

Each file starts with a `H1` header. The contents is the title.

The header has the following attributes:

1. The ID. (`{#ID}`)
2. The status is given by an attribute `status`, which should be value of the values in Table 11.1.
3. (Optional) The language is given by an attribute `lang` (`{lang=en-US}`).
4. (Optional) The type is given by an attribute `type` (`{type=demo}`).

Here is an example of a header with all the attributes:

```
# Odometry calibration {#odometry-calibration lang=en-US type='text/theory' status=ready}

This first paragraph will be used as the ''summary'' for this text.
```

Listing 11.4. `calibration.en.md`

And this is how the Italian translation would look like:

```
# Calibrazione dell'odometria {#odometry-calibration lang=it type='text/theory'
status=draft}

Questo paragrafo sarà usato come un sommario del testo.
```

Listing 11.5. `calibration.it.md`

## 11.4. How to describe the semantic graphs of atoms

In the text, you describe the semantic graph using tags and IDs.

In Markdown, you can give IDs to sections using the syntax:

```
# Setup step 1  {#setup-step1}

This is the first setup step.
```

Then, when you write the second step, you can add a a semantic edge using the following.

```
# Setup step 2  {#setup-step2}

This is the second setup step.

Requires: You have completed the first step in [](#setup-step1).
```

The following table describes the syntax for the different types of semantic links:

TABLE 11.2. SEMANTIC LINKS

| | |
|---|---|
| Requires | `Requires: You need to have done [](#setup-step).` |
| Recommended | `Recommended: It is better if you have setup Wifi as in [](#setup-wifi).` |
| Reference | `Reference: For more information about rostopic, see [](#rostopic).` |
| See also | `See also: If you are interested in feature detection, you might want to learn about [SIFT](#SIFT).` |

## 11.5. How to describe modules

TODO: Define a micro-format for this.

# UNIT B-12
# Translations

**Note:** This part is not implemented yet.

## 12.1. File organization

Translations are organized file-by-file.

For every file *name*`.md`, name the translated file *name*.*language code*`.md`, where the language code is one of the standard codes, and put it in the same directory.

For example, these could be a set of files, including a Chinese (simplified), italian, and Spanish translation:

```
representations.md
representations.zh-CN.md
representations.it.md
representations.es.md
```

The reason is that in this way you can check automatically from Git whether `representations.zh-CN.md` is up to date or `representations.md` has been modified since.

## 12.2. Guidelines for English writers

Here are some considerations for the writers of the original version, to make the translators' job easier.

It is better to keep files smallish so that (1) the translation tasks can feel approachable by translators; (2) it is easier for the system to reason about the files.

Name all the headers with short, easy identifiers, and never change them.

## 12.3. File format

All files are assumed to be encoded in UTF-8.

The header IDs should not be translated and should remain exactly the same. This will allow keeping track of the different translations.

For example, if this is the original version:

```
# Robot uprising {#robot-uprising}

Hopefully it will never happen.
```

Then the translated version should be:

```
# La rivolta dei robot {#robot-uprising}

Speriamo che non succeda.
```

# Part C
# Operation manual - Duckiebot

In this section you will find information to obtain the necessary equipment for Duckietowns and different Duckiebot configurations.

# Unit C-1
# Duckiebot configurations

Knowledge and activity graph

**Requires:** nothing

**Results:** Knowledge of Duckiebot configuration naming conventions, their components and functionalities.

**Next:** After reviewing the configurations, you can proceed to purchasing the components, reading a description of the components, or assembling your chosen configuration.

We define different Duckiebot configurations depending on their time of use and hardware components. This is a good starting point if you are wondering what parts you should obtain to get started.

## 1.1. Duckiebot Configurations: Fall 2017

The configurations are defined with a root: `DB17-`, indicating the "bare bones" Duckiebot used in the Fall 2017 synchronized course, and an appendix `y` which can be the union (in any order) of any or all of the elements of the optional hardware set $\mathcal{O} = \{$`w`, `j`, `d`, `p`, `l`, `c`$\}$.

The elements of $\mathcal{O}$ are labels identifying optional hardware that aids in the development phase and enables the Duckiebot to talk to other Duckiebots. The labels stand for:

- `w`: 5 GHz wireless adapter to facilitate streaming of images;
- `j`: wireless joypad that facilitates manual remote control;
- `d`: USB drive for additional storage space;
- `c`: a different castor wheel to *replace* the preexisting omni-directional wheel;
- `l`: includes LEDs, LED hat, bumpers and the necessary mechanical bits to set the bumpers in place. Note that the installation of the bumpers induces the *replacement* of a few `DB17` components;

**Note:** During the Fall 2017 course, three Duckietown Engineering Co. branches (Zurich, Montreal, Chicago) are using these configuration naming conventions. Moreover, all institutions release hardware to their Engineers in training in two phases. We summarize the configuration releases below.

## 1.2. Configuration functionality

### 1) DB17

This is the minimal configuration for a Duckiebot. It is the configuration of choice for tight budgets or when operation of a single Duckiebot is more of interest than fleet behaviors.

- **Functions**: A `DB17` Duckiebot can navigate autonomously in a Duckietown, but cannot communicate with other Duckiebots.
- **Components**: A "bare-bones" `DB17` configuration includes:

TABLE 1.1. COMPONENTS OF THE DB17 CONFIGURATION

| | |
|---|---|
| Chassis | USD 20 |
| Camera with 160-FOV Fisheye Lens | USD 22 |
| Camera Mount | USD 8.50 |
| 300mm Camera Cable | USD 2 |
| Raspberry Pi 3 - Model B | USD 35 |
| Heat Sinks | USD 5 |
| Power supply for Raspberry Pi | USD 7.50 |
| 16 GB Class 10 MicroSD Card | USD 10 |
| Mirco SD card reader | USD 6 |
| DC Motor HAT | USD 22.50 |
| Spliced USB-A power cable | USD 0 |
| 2 Stacking Headers | USD 2.50/piece |
| Battery | USD 20 |
| 16 Nylon Standoffs (M2.5 12mm F 6mm M) | USD 0.05/piece |
| 4 Nylon Hex Nuts (M2.5) | USD 0.02/piece |
| 4 Nylon Screws (M2.5x10) | USD 0.05/piece |
| 2 Zip Ties (300x5mm) | USD 9 |
| Total cost for `DB17` configuration | USD 173.6 |

- **Description of components**: Unit C-2 - Acquiring the parts (`DB17-jwd`)
- **Assembly instructions**: Without videos, with videos

### 2) `DB17-w`

This configuration is the same as `DB17` with the *addition* of a 5 Ghz wireless adapter.

- **Functions**: This configuration has the same functionality of `DB17`. In addition, it equips the Duckiebot with a secondary, faster, Wi-Fi connection, ideal for image streaming.
- **Components**:

TABLE 1.2. COMPONENTS OF THE DB17-W CONFIGURATION

| | |
|---|---|
| `DB17` | USD 173.6 |
| Wireless Adapter (5 GHz) | USD 20 |
| Total cost for `DB17-w` configuration | USD 193.6 |

- **Description of components**: Unit C-2 - Acquiring the parts (`DB17-jwd`)
- **Assembly instructions**: Without videos, with videos

### 3) `DB17-j`

This configuration is the same as `DB17` with the *addition* of a 2.4 GHz wireless joypad.

- **Functions**: This configuration has the same functionality of `DB17`. In addition, it equips the Duckiebot with manual remote control capabilities. It is particularly useful for getting the Duckiebot our of tight spots or letting younger ones have a drive, in addition to providing handy shortcuts to different functions in development phase.
- **Components**:

TABLE 1.3. COMPONENTS OF THE DB17-J CONFIGURATION

| | |
|---|---|
| DB17 | USD 173.6 |
| Joypad | USD 10.50 |
| Total cost for DB17-j configuration | USD 184.1 |

- **Description of components**: Unit C-2 - Acquiring the parts (DB17-jwd)
- **Assembly instructions**: Without videos, with videos

### 4) DB17-d

This configuration is the same as DB17 with the *addition* of a USB flash hard drive.

- **Functions**: This configuration has the same functionality of DB17. In addition, it equips the Duckiebot with an external hard drive that is convenient for storing videos (logs) as it provides both extra capacity and faster data transfer rates than the microSD card in the Raspberry Pi. Moreover, it is easy to unplug it from the Duckiebot at the end of the day and bring it over to a computer for downloading and analyzing stored data.
- **Components**:

TABLE 1.4. COMPONENTS OF THE DB17-D CONFIGURATION

| | |
|---|---|
| DB17 | USD 173.6 |
| Tiny 32GB USB Flash Drive | USD 12.50 |
| Total cost for DB17-d configuration | USD 186.1 |

- **Description of components**: Unit C-2 - Acquiring the parts (DB17-jwd)
- **Assembly instructions**: Without videos, with videos

### 5) DB17-c

In this configuration, the DB17 omni-directional wheel is *replaced* with a caster wheel.

- **Functions**: The caster wheel upgrade provides a smoother ride.
- **Components**:

TABLE 1.5. COMPONENTS OF THE DB17-C CONFIGURATION

| | |
|---|---|
| DB17 | USD 173.6 |
| Caster (DB17-c) | USD 6.55/4 pieces |
| 4 Standoffs (M3 12mm F-F) | USD 0.63/piece |
| 8 Screws (M3x8mm) | USD 4.58/100 pieces |
| 8 Split washer lock | USD 1.59/100 pieces |
| Total cost for DB17-c configuration | USD 178.25 |

**TODO:** update links of mechanical bits from M3.5 to M3.

> **Note:** The omni-directional caster wheel is included in the chassis package, so replacing it does not reduce the DB17 cost.

- **Description of components**: Unit E-1 - Acquiring the parts (DB17-lc)
- **Assembly instructions**: Unit E-3 - Assembling the Duckiebot (DB17-lc)

### 6) DB17-l

In this configuration the Duckiebot in equipped with the necessary hardware for controlling and placing 5 RGB LEDs on the Duckiebot. Differently from previous configurations that add or replace a single component, DB17-l introduces several hardware components that are all necessary for a proper use of the LEDs.

It may be convenient at times to refer to hybrid configurations including any of the DB17-jwcd in conjunction with a *subset* of the DB17-l components. In order to disambiguate, let the partial upgrades be defined as:

- DB17-l1: *adds* a PWM hat to DB17, in addition to a short USB angled power cable and a M-M power wire;
- DB17-l2: *adds* a bumpers set to DB17, in addition to the mechanical bits to assemble it;
- DB17-l3: *adds* a LED hat and 5 RGB LEDs to DB17-l112, in addition to the F-F wires to connect the LEDs to the LED board.

> **Note:** introducing the PWM hat in DB17-l1 induces a *replacement* of the spliced cable powering solution for the DC motor hat. Details can be found in Unit E-3 - Assembling the Duckiebot (DB17-lc).

- **Functions**: DB17-l is the necessary configuration to enable communication between Duckiebots, hence fleet behaviors (e.g., negotiating the crossing of an intersection). Subset configurations are sometimes used in a standalone way for: (DB17-l1) avoid using a sliced power cable to power the DC motor hat in DB17, and (DB17-l2) for purely aesthetic reasons.
- **Components**:

TABLE 1.6. COMPONENTS OF THE DB17-L CONFIGURATION

| | |
|---|---|
| DB17 | USD 173.6 |
| PWM/Servo HAT (DB17-l1) | USD 17.50 |
| Power Cable (DB17-l1) | USD 7.80 |
| Male-Male Jumper Wire (150mm) (DB17-l1) | USD 1.95 |
| Bumper set (DB17-l2) | USD 7 (custom made) |
| 8 M3x10 pan head screws (DB17-l2) | USD 7 (custom made) |
| 8 M3 nuts (DB17-l2) | USD 7 (custom made) |
| Bumpers (DB17-l2) | USD 7 (custom made) |
| LEDs (DB17-l3) | USD 10 |
| LED HAT (DB17-l3) | USD 28.20 for 3 pieces |
| 20 Female-Female Jumper Wires (300mm) (DB17-l3) | USD 8 |
| 4 4 pin female header (DB17-l3) | USD 0.60/piece |
| 12 pin male header (DB17-l3) | USD 0.48/piece |
| 2 16 pin male header (DB17-l3) | USD 0.61/piece |
| 3 pin male header (DB17-l3) | USD 0.10/piece |
| 2 pin female shunt jumper (DB17-l3) | USD 2/piece |
| 40 pin female header (DB17-l3) | USD 1.50 |
| 5 200 Ohm resistors (DB17-l3) | USD 0.10/piece |
| 10 130 Ohm resistors (DB17-l3) | USD 0.10/piece |
| Total for DB17-l configuration | USD 305 |

- **Description of components**: Unit E-1 - Acquiring the parts (DB17-lc)
- **Assembly instructions**: Unit E-3 - Assembling the Duckiebot (DB17-lc)

## 1.3. Branch configuration releases: Fall 2017

All branches release their hardware in two phases, namely a and b.

### 1) Zurich

- First release (`DB17-Zurich-a`): is a `DB17-wjd`.
- Second release (`DB17-Zurich-b`): is a `DB17-wjdcl`.

### 2) Montreal

- First release (`DB17-Montreal-a`): is a hybrid `DB17-wjd` + PWM hat (or `DB17-wjdl1`).
- Second release (`DB17-Montreal-b`): is a `DB17-wjdl`.

> **Note:** The Montreal branch is not implementing the `DB17-c` configuration.

### 3) TTIC

- First release (`DB17-Chicago-a`): is a `DB17-wjd`.
- Second release (`DB17-Chicago-b`): is a `DB17-wjdl`.

> **Note:** The Chicago branch is not implementing the `DB17-c` configuration.

UNIT C-2

# Acquiring the parts (DB17-jwd)

The trip begins with acquiring the parts. Here, we provide a link to all bits and pieces that are needed to build a Duckiebot, along with their price tag. If you are wondering what is the difference between different Duckiebot configurations, read this.

In general, keep in mind that:

- The links might expire, or the prices might vary.
- Shipping times and fees vary, and are not included in the prices shown below.
- Substitutions are OK for the mechanical components, and not OK for all the electronics, unless you are OK in writing some software.
- Buying the parts for more than one Duckiebot makes each one cheaper than buying only one.
- For some components, the links we provide contain more bits than actually needed.

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Cost: USD 174 + Shipping Fees (minimal configuration `DB17`)

**Requires:** Time: 15 days (average shipping for cheapest choice of components)

**Results:** A kit of parts ready to be assembled in a `DB17` or `DB17-wjd` configuration.

**Next:** After receiving these components, you are ready to do some soldering before assembling your `DB17` or `DB17-wjd` Duckiebot.

## 2.1. Bill of materials

TABLE 2.1. BILL OF MATERIALS

| | |
|---|---|
| Chassis | USD 20 |
| Camera with 160-FOV Fisheye Lens | USD 22 |
| Camera Mount | USD 8.50 |
| 300mm Camera Cable | USD 2 |
| Raspberry Pi 3 - Model B | USD 35 |
| Heat Sinks | USD 5 |
| Power supply for Raspberry Pi | USD 7.50 |
| 16 GB Class 10 MicroSD Card | USD 10 |
| Mirco SD card reader | USD 6 |
| DC Motor HAT | USD 22.50 |
| 2 Stacking Headers | USD 2.50/piece |
| Battery | USD 20 |
| 16 Nylon Standoffs (M2.5 12mm F 6mm M) | USD 0.05/piece |
| 4 Nylon Hex Nuts (M2.5) | USD 0.02/piece |
| 4 Nylon Screws (M2.5x10) | USD 0.05/piece |
| 2 Zip Ties (300x5mm) | USD 9 |
| Wireless Adapter (5 GHz) (`DB17-w`) | USD 20 |
| Joypad (`DB17-j`) | USD 10.50 |
| Tiny 32GB USB Flash Drive (`DB17-d`) | USD 12.50 |
| Total for `DB17` configuration | USD 173.6 |
| Total for `DB17-w` configuration | USD 193.6 |
| Total for `DB17-j` configuration | USD 184.1 |
| Total for `DB17-d` configuration | USD 186.1 |
| Total for `DB17-wjd` configuration | USD 216.6 |

## 2.2. Chassis

We selected the Magician Chassis as the basic chassis for the robot (Figure 2.1).

We chose it because it has a double-decker configuration, and so we can put the battery in the lower part.

The chassis pack includes 2 DC motors and wheels as well as the structural part, in addition to a screwdriver and several necessary mechanical bits (standoffs, screws and nuts).

Figure 2.1. The Magician Chassis

## 2.3. Raspberry Pi 3 - Model B

The Raspberry Pi is the central computer of the Duckiebot. Duckiebots use Model B (Figure 2.2) ( A1.2GHz 64-bit quad-core ARMv8 CPU, 1GB RAM), a small but powerful computer.

Figure 2.2. The Raspberry Pi 3 Model B

### 1) Power Supply

We want a hard-wired power source (5VDC, 2.4A, Micro USB) to supply the Raspberry Pi (Figure 2.3) while not driving. This charger can double down as battery charger as well.

Figure 2.3. The Power Supply

**Note:** Students in the ETHZ-Fall 2017 course will receive a converter for US to CH plug.

### 2) Heat Sinks

The Raspberry Pi will heat up significantly during use. It is warmly recommended to add heat sinks, as in Figure 2.4. Since we will be stacking HATs on top of the Raspberry Pi with 15 mm standoffs, the maximum height of the heat sinks should be well below 15 mm. The chip dimensions are 15x15mm and 10x10mm.

Figure 2.4. The Heat Sinks

### 3) Class 10 MicroSD Card

The MicroSD card (Figure 2.5) is the hard disk of the Raspberry Pi. 16 GB of capacity are sufficient for the system image.

Figure 2.5. The MicroSD card

### 4) Mirco SD card reader

A microSD card reader (Figure 2.6) is useful to copy the system image to a Duckiebot

from a computer to the Raspberry Pi microSD card, when the computer does not have a native SD card slot.

Figure 2.6. The Mirco SD card reader

## 2.4. Camera

The Camera is the main sensor of the Duckiebot. All versions equip a 5 Mega Pixels 1080p camera with wide field of view (**160°**) fisheye lens (Figure 2.7).

Figure 2.7. The Camera with Fisheye Lens

### 1) Camera Mount

The camera mount (Figure 2.8) serves to keep the camera looking forward at the right angle to the road (looking slightly down). The front cover is not essential.

Figure 2.8. The Camera Mount

The assembled camera (without camera cable), is shown in (Figure 2.9).

Figure 2.9. The Camera on its mount

### 2) 300mm Camera Cable

A longer (300 mm) camera cable Figure 2.10 makes assembling the Duckiebot easier, allowing for more freedom in the relative positioning of camera and computational stack.

Figure 2.10. A 300 mm camera cable for the Raspberry Pi

## 2.5. DC Motor HAT

We use the DC Stepper motor HAT (Figure 2.11) to control the DC motors that drive

the wheels. This item will require soldering to be functional. This HAT has dedicate PWM and H-bridge for driving the motors.

Figure 2.11. The Stepper Motor HAT

### 1) Stacking Headers

We use a long 20x2 GPIO stacking header (Figure 2.12) to connect the Raspberry Pi with the DC Motor HAT. This item will require soldering to be functional.

Figure 2.12. The Stacking Headers

## 2.6. Battery

The battery (Figure 2.13) provides power to the Duckiebot.

We choose this battery because it has a good combination of size (to fit in the lower deck of the Magician Chassis), high output amperage (2.4A and 2.1A at 5V DC) over two USB outputs, a good capacity (10400 mAh) at an affordable price. The battery linked in the table above comes with two USB to microUSB cables.

Figure 2.13. The Battery

## 2.7. Standoffs, Nuts and Screws

We use non electrically conductive standoffs (M2.5 12mm F 6mm M), nuts (M2.5), and

screws (M2.5x10mm) to hold the Raspberry Pi to the chassis and the HATs stacked on top of the Raspberry Pi.

The Duckiebot requires 8 standoffs, 4 nuts and 4 screws.

Figure 2.14. Standoffs, Nuts and Screws

## 2.8. Zip Tie

Two 300x5mm zip ties are needed to keep the battery at the lower deck from moving around.

Figure 2.15. The zip ties

## 2.9. Configuration DB17-w

### 1) Wireless Adapter (5 GHz)

The Edimax AC1200 EW-7822ULC 5 GHz wireless adapter (Figure 2.16) boosts the connectivity of the Duckiebot, especially useful in busy Duckietowns (e.g., classroom). This additional network allows easy streaming of images.

Figure 2.16. The Edimax AC1200 EW-7822ULC wifi adapter

## 2.10. Configuration DB17-j

### 1) Joypad

The joypad is used to manually remote control the Duckiebot. Any 2.4 GHz wireless controller (with a *tiny* USB dongle) will do.

The model linked in the table (Figure 2.17) does not include batteries.

Figure 2.17. A Wireless Joypad

**Requires:** 2 AA 1.5V batteries (Figure 2.18).

Figure 2.18. A Wireless Joypad

## 2.11. Configuration DB17-d

### 1) Tiny 32GB USB Flash Drive

In configuration `DB17-d`, the Duckiebot is equipped with an "external" hard drive (Figure 2.19). This add-on is very convenient to store logs during experiments and later port them to a workstation for analysis. It provides storage capacity and faster data transfer than the MicroSD card.

Figure 2.19. The Tiny 32GB USB Flash Drive

# Unit C-3
# Soldering boards (DB17)

Assigned to: Shiying

Knowledge and activity graph

Requires: Parts: Duckiebot `DB17` parts. The acquisition process is explained in Unit C-2 - Acquiring the parts (`DB17-jwd`). The configurations are described in Unit C-1 - Duckiebot configurations. In particular:

- GPIO Stacking Header
- DC and Stepper Motor HAT for Raspberry Pi

Requires: Tools: Solderer

Requires: Experience: some novice-level experience with soldering.

Requires: Time: 30 minutes

Results: Soldered DC Motor HAT

Note: It is better to be safe than sorry. Soldering is a potentially hazardous activity. There is a fire hazard as well as the risk of inhaling toxic fumes. Stop a second and make sure you are addressing the safety standards for soldering when following these instructions. If you have never soldered before, seek advice.

## 3.1. General tips

Note: There is a general rule in soldering: solder the components according to their height, from lowest to highest.

In this instruction set we will assume you have soldered something before and are acquainted with the soldering fundamentals. If not, before proceeding, read this great tutorial on soldering:

➡ Alternative instructions: how to solder on Headers and Terminal Block

### 1) Preparing the components

Take the GPIO stacking header Figure 3.1 out of Duckiebox and sort the following components from DC motor HAT package:

- Adafruit DC/Stepper Motor HAT for Raspberry Pi
- 2-pin terminal block (2x), 3-pin terminal block (1x)

Figure 3.1. GPIO_Stacking_Header

Figure 3.2. DC/Stepper Motor HAT and solder components

## 2) Soldering instructions

1) Make a 5 pin terminal block by sliding the included 2 pin and 3 pin terminal blocks into each other Figure 3.3.

Figure 3.3. 5 pin terminal_block

2) Slide this 5 pin block through the holes just under “M1 GND M2” on the board. Solder it on (we only use two motors and do not need connect anything at the “M3 GND M4” location) (Figure 3.6);

3) Slide a 2 pin terminal block into the corner for power. Solder it on. (Figure 3.5);

4) Slide in the GPIO Stacking Header onto the 2x20 grid of holes on the edge opposite the terminal blocks and with vice versa direction (Figure 3.4). Solder it on.

> **Note:** stick the GPIO Stacking Header from bottom to top, different orientation than terminal blocks (from top to bottom).

Figure 3.4.

Figure 3.5. Side view of finished soldering DC/Stepper Motor HAT

Figure 3.6. upside view of finished soldering DC/Stepper Motor HAT

UNIT C-4

# Preparing the power cable (DB17)

In configuration `DB17` we will need a cable to power the DC motor HAT from the battery. The keen observer might have noticed that such a cable was not included in the `DB17` Duckiebot parts chapter. Here, we create this cable by splitting open any USB-A cable, identifying and stripping the power wires, and using them to power the DC motor HAT. If you are unsure about the definitions of the different Duckiebot configurations, read Unit C-1 - Duckiebot configurations.

It is important to note that these instructions are relevant only for assembling a `DB17-wjdc` configuration Duckiebot (or any subset of it). If you intend to build a `DB17-1` configuration Duckiebot, you can skip these instructions.

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** One male USB-A to anything cable.

**Requires:** A pair of scissors.

**Requires:** A multimeter (only if you are not purchasing the suggested components)

**Requires:** Time: 5 minutes

**Results:** One male USB-A to wires power cable

## 4.1. Video tutorial

The following video shows how to prepare the USB power cable for the configuration `DB17`.

Figure 4.1

## 4.2. Step-by-step guide

### 1) Step 1: Find a cable

To begin with, find a male USB-A to anything cable.

If you have purchased the suggested components listed in Unit C-2 - Acquiring the parts (`DB17-jwd`), you can use the longer USB cable contained inside the battery package (Figure 4.2), which will be used as an example in these instructions.

Figure 4.2. The two USB cables in the suggested battery pack.

Put the shorter cable back in the box, and open the longer cable (Figure 4.3)

Figure 4.3. Take the longer cable, and put the shorter on back in the box.

## 2) Step 2: Cut the cable

**Check before you continue**
Make sure the USB cable is *unplugged* from any power source before proceeding.

Take the scissors and cut it (Figure 4.4) at the desired length from the USB-A port.

Figure 4.4. Cut the USB cable using the scissors.

The cut will look like in Figure 4.5.

Figure 4.5. A cut USB cable.

### 3) Step 3: Strip the cable

Paying attention not to get hurt, strip the external white plastic. A way to do so without damaging the wires is shown in Figure 4.6.

Figure 4.6. Stripping the external layer of the USB cable.

After removing the external plastic, you will see four wires: black, green, white and red (Figure 4.7).

Figure 4.7. Under the hood of a USB-A cable.

Once the bottom part of the external cable is removed, you will have isolated the four wires (Figure 4.8).

Figure 4.8. The four wires inside a USB-A cable.

### 4) Step 4: Strip the wires

**Check before you continue**

Make sure the USB cable is *unplugged* from any power source before proceeding.

Once you have isolated the wires, strip them, and use the scissors to cut off the data wires (green and white, central positions) (Figure 4.9).

Figure 4.9. Strip the power wires and cut the data wires.

If you are not using the suggested cable, or want to verify which are the data and power wires, continue reading.

### 5) Step 5: Find the power wires

If you are using the USB-A cable from the suggested battery pack, black and red are the power wires and green and white are instead for data.

If you are using a different USB cable, or are curious to verify that black and red actually are the power cables, take a multimeter and continue reading.

Plug the USB port inside a power source, e.g., the Duckiebot's battery. You can use some scotch tape to keep the cable from moving while probing the different pairs of wires with a multimeter. The voltage across the pair of power cables will be roughly twice the voltage between a power and data cable. The pair of data cables will have no voltage differential across them. If you are using the suggested Duckiebot battery as power source, you will measure around 5V across the power cables (Figure 4.10).

Figure 4.10. Finding which two wires are for power.

## 6) Step 6: Test correct operation

You are now ready to secure the power wires to the DC motor HAT power pins. To do so though, you need to have soldered the boards first. If you have not done so yet, read Unit C-3 - Soldering boards (DB17).

If you have soldered the boards already, you may test correct functionality of the newly crafted cable. Connect the battery with the DC motor HAT by making sure you plug the black wire in the pin labeled with a minus: `-` and the red wire to the plus: `+` (Figure 4.11).

Figure 4.11. Connect the power wires to the DC motor HAT

# Unit C-5
# Assembling the Duckiebot (DB17-jwd)

**Point of contact:** Shiying Li

Once you have received the parts and soldered the necessary components, it is time to assemble them in a Duckiebot. Here, we provide the assembly instructions for configurations `DB17-wjd`.

Knowledge and activity graph

**Requires:** Duckiebot `DB17-wjd` parts. The acquisition process is explained in Unit C-2 - Acquiring the parts (DB17-jwd).

**Requires:** Having soldered the `DB17-wjd` parts. The soldering process is explained in Unit C-3 - Soldering boards (DB17).

**Requires:** Having prepared the power cable. The power cable preparation is explained in Unit C-4 - Preparing the power cable (DB17). Note: Not necessary if you intend to build a `DB17-l` configuration.

**Requires:** Having installed the image on the MicroSD card. The instructions on how to reproduce the Duckiebot system image are in Unit C-7 - Reproducing the image.

**Requires:** Time: about 40 minutes.

**Results:** An assembled Duckiebot in configuration `DB17-wjd`.

**Note:** The FAQ section at the bottom of this page may already answer some of you comments, questions or doubts.

**Note:** While assembling the Duckiebot, try to make as symmetric (along the longitudinal axis) as you can. It will help going forward.

## 5.1. Chassis

Open the Magician chassis package (Figure 5.1) and take out the following components:

- Chassis-bottom (1x), Chassis-up (1x);
- DC Motors (2x), motor holders (4x);
- Wheels (2x), steel omni-directional wheel (1x);
- All spacers and screws;
- Screwdriver.

Figure 5.1. Components in Duckiebot package.

**Note:** You won't need the battery holder and speed board holder (on the right side in Figure 5.1).

### 1) Bottom

Insert the motor holders on the chassis-bottom and put the motors as shown in the figure below (with the longest screws (M3x30) and M3 nuts).

Figure 5.2. Components for mounting the motor

Figure 5.3. The scratch of assembling the motor

Figure 5.4. Assembled motor

**Note:** Orient the motors so that their wires are inwards, i.e., towards the center of the chassis-bottom. The black wires should be closer to the chassis-bottom to make wiring easier down the line.

**Note:** if your Magician Chassis package has unsoldered motor wires, you will have to solder them first. Check these instructions. In this case, your wires will not have the male pin headers on one end. Do not worry, you can still plug them in the stepper motor hat power terminals.

**TODO:** make instructions for soldering motor wires

### 2) Wheels

Plug in the wheels to the motor as follows (no screws needed):

Figure 5.6. Wheel assembly schematics

Figure 5.7. Assembled wheels

Figure 5.5. Wheel assembly instructions

### 3) Omni-directional wheel

The Duckiebot is driven by controlling the wheels attached to the DC motors. Still, it requires a “passive” omni-directional wheel (the *caster* wheel) on the back.

The Magician chassis package contains a steel omni-directional wheel, and the related standoffs and screws to secure it to the chassis-bottom part.

Figure 5.8. The omni-directional wheel schematics

Figure 5.9. Assembled omni-directional wheel

### 4) Caster wheel

As alternative to omnidirection wheel, caster wheel has less friction. If you have purchased caster wheel, read this section.

To assemble the caster wheel, the following materials are needed:

- caster wheel (1x)
- Metal standoffs (M3x12mm F-F, 6mm diameter) (4x)
- Metal screws (M3x8mm) (8x)
- Split/Spring lock washers (M3) (8x)
- Flat lock washers (M3) (8x)

Figure 5.10. Component-List for assemblying the caster wheels

*Prepare the Screws with Washers:*

The lock washers belongs to screw-head side Figure 5.11, i.e. the split lock washer and the flat lock washers stays always near the screw head. The split lock washer stays near the screw head. First split lock washer, then flat lock washer.

Figure 5.11. Insert the locker washers into metal screws from left to right

Figure 5.12. The metal screws with the lock washers

*Assembly the metal standoffs on the caster wheels:*

Fasten the screws with washers on the caster wheels from the bottom up and screw the metal standoffs from top to bottom. The caster before mounting looks like in Figure 5.13.

Figure 5.13. The assembled caster before mounting it under the chassis-bottom

*Assembly the caster wheels under the chassis bottom:*

Assembly the prepared caster wheels in the front side of duckiebot under the chassis bottom. Fasten the screws with washers from top to bottom.

✔ In order to get all the screws properly into the metal standoffs, let all the screws stay loose within the right positions before all the screws are inserted into the standoffsFigure 5.15.

Figure 5.14. Assembly the caster wheels under chassis-bottom

Figure 5.15. Assembled caster wheels (sideview)

### 5) Mounting the standoffs

Put the car upright (omni wheel pointing towards the table) and arrange wires so that they go through the center rectangle. Put 4 spacers with 4 of M3x6 screws on exact position of each corner as below Figure 5.17.

Figure 5.16. Metal spacers and M3x6mm screws

Figure 5.17. The spacers on each corner of the chassis-bottom

The bottom part of the Duckiebot's chassis is now ready. The next step is to assemble the Raspberry Pi on chassis-top part.

## 5.2. Assembling the Raspberry Pi, camera, and HATs

### 1) Raspberry Pi

Before attaching anything to the Raspberry Pi you should add the heat sinks to it. There are 2 small sinks and a big one. The big one best fits onto the processor (the big "Broadcom"-labeled chip in the center of the top of the Raspberry Pi). One of the small ones can be attached to the small chip that is right next to the Broadcom chip. The third heat sink is optional and can be attached to the chip on the underside of the Raspberry Pi. Note that the chip on the underside is bigger than the heat sink. Just mount the heat sink in the center and make sure all of them are attached tightly.

When this is done fasten the nylon standoffs on the Raspberry Pi, and secure them on the top of the chassis-up part by tightening the nuts on the opposite side of the chassis-up.

Figure 5.18. Components for Raspberry Pi3

Figure 5.19. Heat sink on Raspberry Pi3

Figure 5.20. Nylon standoffs for Raspberry Pi3

Figure 5.21. Attach the nylon huts for the standoffs (bottom view)

Figure 5.22. Assembled Raspberry Pi3 (top view)

### 2) Micro SD card

**Requires:** Having the Duckiebot image copied in the micro SD card.

Take the micro SD card from the Duckiebox and insert its slot on the Raspberry Pi. The SD card slot is just under the display port, on the short side of the PI, on the flip side of where the header pins are.

Figure 5.23. The micro SD card and mirco SD card readers

Figure 5.24. Inserted SD card

### 3) Camera

> **Note:** If you have camera cables of different lengths available, keep in mind that both are going to work. We suggest to use the longer one, and wrap the extra length under the Raspberry Pi stack.

*The Raspberry Pi end:*

First, identify the camera cable port on the Pi (between HDMI and power ports) and remove the orange plastic protection (it will be there if the Pi is new) from it. Then,

grab the long camera cable (300 mm) and insert in the camera port. To do so, you will need to gently pull up on the black connector (it will slide up) to allow the cable to insert the port. Slide the connector back down to lock the cable in place, making sure it "clicks".

**TODO:** insert image with long cable

Figure 5.25. Camera port on the Raspberry Pi and camera cable

> **Note:** Make sure the camera cable is inserted in the right direction! The metal pins of the cable should be in contact with the metal terminals in the camera port of the PI.

Figure 5.26. Camera with long cable

*The camera end:*

If you have the long camera cable, the first thing to do is removing the shorter cable that comes with the camera package. Make sure to slide up the black connectors of the camera-camera cable port in order to unblock the cable.

Take the rear part of the camera mount and use it hold the camera in place. Note that the camera is just press-fitted to the camera mount, no screws/nuts are needed.

In case you have not purchased the long camera cable, do not worry! It is still very possible to get a working configuration, but you will have little wiggling space and assembly will be a little harder.

Place the camera on the mount and fasten the camera mount on the chassis-up using M3x10 flathead screws and M3 nuts from the Duckiebox.

Protip: make sure that the camera mount is: (a) geometrically centered on the chassis-up; (b) fastened as forward as it can go; (c) it is tightly fastened. We aim at having a standardized position for the camera and to minimize the wiggling during movement.

Figure 5.27. Raspberry Pi and camera with short cable

> **Note:** If you only have a short camera cable, make sure that the cable is oriented in this direction (text on cable towards the CPU). Otherwise you will have to disassemble the whole thing later. On the long cable the writing is on the other side.

### 4) Extending the intra-decks standoffs

In order to fit the battery, we will need to extend the Magician chassis standoffs with the provided nylon standoff spacers. Grab 4 of them, and secure them to one end of the long metal standoffs provided in the Magician chassis package.

Secure the extended standoff to the 4 corners of the chassis-bottom. The nylon standoffs should smoothly screw in the metal ones. If you feel resistance, don't force it or the nylon screw might break in the metal standoff. In that case, unscrew the nylon spacer and try again.

Figure 5.28. 4 nylon M3x5 extended standoffs and 4 M3x6 metal screws from Magician chassis package

### 5) Fasten the Battery with zip ties

Put the battery between the upper and lower decks of the chassis. It is strongly recommended to secure the battery from moving using zip ties.

Figure 5.29. Secure the battery to the chassis-top through the provided zipties. One can do the trick, two are better.

> **Note:** Figure 5.29 can be taken as an example of how to arrange the long camera cable as well.

### 6) Assemble chassis-bottom and chassis-up

Arrange the motor wires through the chassis-up, which will be connected to Stepper Motor HAT later.

Figure 5.30. The motor wires go through the center of chassis-up

Figure 5.31. Side view of metal screws and the extended standoffs

**Note:** Use the provided metal screws from chassis package for fastening the chassis-up above the nylon standoffs instead of the provided M3 nylon screws.

### 7) Place the DC Motor hat on top of the Raspberry Pi

Make sure the GPIO stacking header is carefully aligned with the underlying GPIO pins before applying pressure.

**Note:** In case with short camera cable, ensure that you doesn't break the cable while mounting the HAT on the Raspberry Pi. In case with long camera cable,

Figure 5.32. Assembled DC motor hat with short camera cable

**TODO:** insert pic with long camera cable

### 8) Connect the motor's wires to the terminal

We are using M1 and M2 terminals on the DC motor hat. The left (in robot frame) motor is connected to M1 and the right motor is connected to M2. If you have followed Part A correctly, the wiring order will look like as following pictures:

- Left Motor: Red
- Left Motor: Black
- Right Motor: Black
- Right Motor:Red

### 9) Connect the power cables

You are now ready to secure the prepared power wires in Unit C-4 - Preparing the power cable (DB17) to the DC motor HAT power pins.

Connect the **battery** (not the Raspberry Pi) with the DC motor HAT by making sure you plug the black wire in the pin labeled with a minus: `-` and the red wire to the plus: `+` (Figure 4.11).

Fix all the cables on the Duckiebot so that it can run on the way without barrier.

Figure 5.33. Insert the prepared power wire to DC motor HAT power pins.

> **Note:** If you have a `DB17-Montreal-a` or `DB17-Chicago-a` release, neglect this step and follow the pertinent instructions in Unit E-3 - Assembling the Duckiebot (DB17-1c) regarding the assembly of the PWM hat, its powering through the short angled USB cable, and the power transfer step using a M-M wire.

### 10) Joypad

With each joypad (Figure 5.34) comes a joypad dongle (Figure 5.35). Don't lose it!

Figure 5.34. All components in the Joypad package

Insert the joypad dongle into one of the USB port of the Raspberry Pi.

Figure 5.35. The dongle on the Raspberry Pi

Insert 2 AA batteries on the back side of the joypad Figure 5.36.

Figure 5.36. Joypad and 2 AA batteries

## 5.3. FAQ

*Q: If we have the bumpers, at what point should we add them?*

**Answer:** You shouldn't have the bumpers at this point. The function of bumpers is to keep the LEDs in place, i.e., they belong to `DB17-1` configuration. These instructions cover the `DB17-jwd` configurations. You will find the bumper assembly instructions in Unit E-3 - Assembling the Duckiebot (`DB17-1c`).

*Q: Yeah but I still have the bumpers and am reading this page. So?*

**Answer:** The bumpers can be added after the Duckiebot assembly is complete.

*Q: I found it hard to mount the camera (the holes weren't lining up).*

**Answer:** Sometimes in life you have to push a little to make things happen. (But don't push too much or things will break!)

*Q: The long camera cable is a bit annoying - I folded it and shoved it in between two hats.*

**Answer:** The shorter cable is even more annoying. We suggest wrapping the long camera cable between the chassis and the Raspberry Pi. With some strategic planning, you can use the zipties that keep the battery in place to hold the camera cable in place as well (see figure below-to add)

**TODO:** add pretty cable handling pic

***Q:*** *I found that the screwdriver that comes with the chassis kit is too fat to screw in the wires on the hat.*

**Answer:** It is possible you got one of the fatter screwdrivers. You will need to figure it out yourself (or ask a TA for help).

***Q:*** *I need something to cut the end of the zip tie with.*

**Answer:** Scissors typically work out for these kind of jobs (and no, they're not provided in a Fall 2017 Duckiebox).

UNIT C-6

# Assembling the Duckiebot (DB17-wjd TTIC)

**Point of contact:** Andrea F. Daniele

Once you have received the parts and soldered the necessary components, it is time to assemble them in a Duckiebot. Here, we provide the assembly instructions for the configuration `DB17-wjd` (TTIC only).

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Duckiebot `DB17-wjd` parts. The acquisition process is explained in Unit C-2 - Acquiring the parts (`DB17-jwd`).

**Requires:** Having soldered the `DB17-wjd` parts. The soldering process is explained in Unit C-3 - Soldering boards (`DB17`).

**Requires:** Having prepared the power cable. The power cable preparation is explained in Unit C-4 - Preparing the power cable (`DB17`). Note: Not necessary if you intend to build a `DB17-l` configuration.

**Requires:** Time: about 30 minutes.

**Results:** An assembled Duckiebot in configuration `DB17-wjd`.

**Note:** The FAQ section at the bottom of this page may already answer some of you comments, questions or doubts.

This section is comprised of 14 parts. Each part builds upon some of the previous parts, so make sure to follow them in the following order.

- Part I: Motors
- Part II: Wheels
- Part III: Omni-directional wheel
- Part IV: Chassis standoffs
- Part V: Camera kit
- Part VI: Heat sinks
- Part VII: Raspberry Pi 3
- Part VIII: Top plate
- Part IX: USB Power cable
- Part X: DC Stepper Motor HAT
- Part XI: Battery
- Part XII: Upgrade to `DB17-w`
- Part XIII: Upgrade to `DB17-j`
- Part XIV: Upgrade to `DB17-d`

## 6.1. Motors

Open the Magician Chassis package (Figure 5.1) and take out the following components:

- Chassis-bottom (1x)
- DC Motors (2x)
- Motor holders (4x)

- M3x30 screw (4x)
- M3 nuts (4x)

Figure 6.1 shows the components needed to complete this part of the tutorial.

Figure 6.1. Components needed to mount the motors.

## 1) Video tutorial

The following video shows how to attach the motors to the bottom plate of the chassis.

Figure 6.2

## 2) Step-by-step guide

*Step 1:*

Pass the motor holders through the openings in the bottom plate of the chassis as shown in Figure 6.3.

Figure 6.3. The sketch of how to mount the motor holders.

*Step 2:*

Put the motors between the holders as shown in Figure 6.4.

Figure 6.4. The sketch of how to mount the motors.

> **Note:** Orient the motors so that their wires are inwards (i.e., towards the center of the plate).

*Step 3:*

Use *4* M3x30 screws and *4* M3 nuts to secure the motors to the motor holders. Tighten the screws to secure the holders to the bottom plate of the chassis as shown in Figure 6.5.

Figure 6.5. The sketch of how to secure the motors to the bottom plate.

### 3) Check the outcome

Figure 6.6 shows how the motors should be attached to the bottom plate of the chassis.

Figure 6.6. The motors are attached to the bottom plate of the chassis.

## 6.2. Wheels

From the Magician Chassis package take the following components:

- Wheels (2x)

Figure 6.7 shows the components needed to complete this part of the tutorial.

Figure 6.7. The wheels.

### 1) Video tutorial

The following video shows how to attach the wheels to the motors.

Figure 6.8

### 2) Check the outcome

Figure 6.9 shows how the wheels should be attached to the motors.

Figure 6.9. The wheels are attached to the motors.

## 6.3. Omni-directional wheel

The Duckiebot is driven by controlling the wheels attached to the DC motors. Still, it requires a *passive* support on the back. In this configuration an omni-directional wheel is attached to the bottom plate of the chassis to provide such support.

From the Magician Chassis package take the following components:

- Steel omni-directional wheel (1x)
- Long metal spacers (2x)
- M3x6 screws (4x)

Figure 6.10 shows the components needed to complete this part of the tutorial.

Figure 6.10. The omni-directional wheel with *2* long spacers and *4* M3x6 screws.

### 1) Video tutorial

The following video shows how to attach the omni-directional wheel to the bottom plate of the chassis.

Figure 6.11

### 2) Step-by-step guide

*Step 1:*

Secure the long spacers to the plate using *2* M3x6 screws and the omni-directional wheel to the spacers using also *2* M3x6 screws as shown in Figure 6.12.

Figure 6.12. The sketch of how to mount the omni-directional wheel.

### 3) Check the outcome

Figure 6.13 shows how the omni-directional wheel should be attached to the plate.

Figure 6.13. The omni-directional wheel is attached to the plate.

## 6.4. Chassis standoffs

From the Magician Chassis package take the following components:

- Long metal spacers/standoffs (4x)
- M3x6 screws (4x)

From the Duckiebot kit take the following components:

- M3x5 nylon spacers/standoffs (4x)

Figure 6.14 shows the components needed to complete this part of the tutorial.

Figure 6.14. The standoffs to mount on the bottom plate.

### 1) Video tutorial

The following video shows how to attach the standoffs to the bottom plate of the chassis.

Figure 6.15

## 2) Step-by-step guide

*Step 1:*

Secure the long metal spacers to the bottom plate using *4* M3x6 screws as shown in Figure 6.16.

Figure 6.16. The sketch of how to mount the standoffs on the plate.

*Step 2:*

Attach the *4* nylon standoffs on top of the metal ones.

## 3) Check the outcome

Figure 6.17 shows how the standoffs should be attached to the plate.

Figure 6.17. The standoffs attached to the plate.

## 6.5. Camera kit

From the Magician Chassis package take the following components:

- M3x10 flathead screws (2x)
- M3 nuts (2x)

From the Duckiebot kit take the following components:

- Camera Module (1x)
- (Optional) 300mm Camera cable (1x)
- Camera mount (1x)

> Note: If you have camera cables of different lengths available, keep in mind that both are going to work. We suggest to use the longer one, and wrap the extra length under the Raspberry Pi stack.

Figure 6.18 shows the components needed to complete this part of the tutorial.

Figure 6.18. The parts needed to fix the camera on the top plate.

## 1) Video tutorial

The following video shows how to secure the camera to the top plate of the chassis.

Figure 6.19

## 2) Step-by-step guide

*Step 1 (Optional):*

If you do not have the 300mm Camera cable you can jump to *Step 3*.

If you do have the long camera cable, the first thing to do is removing the shorter cable that comes attached to the camera module. Make sure to slide up the black connectors of the camera port on the camera module in order to unblock the cable.

*Step 2:*

Connect the camera cable to the camera module as shown in Figure 6.20.

Figure 6.20. How to connect the camera cable to the camera module.

*Step 3:*

Attach the camera module to the camera mount as shown in Figure 6.21.

Figure 6.21. How to attach the camera to the camera mount.

> **Note:** The camera is just press-fitted to the camera mount, no screws/nuts are needed.

*Step 4:*

Secure the camera mount to the top plate by using the *2* M3x10 flathead screws and the nuts as shown in Figure 6.22.

Figure 6.22. How to attach the camera mount to the top plate.

### 3) Check the outcome

Figure 6.23 shows how the camera should be attached to the plate.

Figure 6.23. The camera attached to the plate.

## 6.6. Heat sinks

From the Duckiebot kit take the following components:

- Raspberry Pi 3 (1x)
- Heat sinks (2x)
- Camera mount (1x)

Figure 6.24 shows the components needed to complete this part of the tutorial.

Figure 6.24. The heat sinks and the Raspberry Pi 3.

### 1) Video tutorial

The following video shows how to install the heat sinks on the Raspberry Pi 3.

Figure 6.25

### 2) Step-by-step guide

*Step 1:*
Remove the protection layer from the heat sinks.

*Step 2:*
Install the big heat sink on the big “Broadcom”-labeled integrated circuit (IC).

*Step 3:*
Install the small heat sink on the small “SMSC”-labeled integrated circuit (IC).

### 3) Check the outcome

Figure 6.26 shows how the heat sinks should be installed on the Raspberry Pi 3.

Figure 6.26. The heat sinks installed on the Raspberry Pi 3.

## 6.7. Raspberry Pi 3

From the Magician Chassis package take the following components:

- Top plate (with camera attached) (1x)

From the Duckiebot kit take the following components:

- Raspberry Pi 3 (with heat sinks) (1x)
- M2.5x12 nylon spacers/standoffs (8x)
- M2.5 nylon hex nuts (4x)

Figure 6.27 shows the components needed to complete this part of the tutorial.

Figure 6.27. The parts needed to mount the Raspberry Pi 3 on the top plate.

### 1) Video tutorial

The following video shows how to mount the Raspberry Pi 3 on the top plate of the chassis.

Figure 6.28

### 2) Step-by-step guide

*Step 1:*

Mount *8* M2.5x12 nylon standoffs on the Raspberry Pi 3 as shown in Figure 6.29.

Figure 6.29. How to mount the nylon standoffs on the Raspberry Pi 3.

*Step 2:*

Use the M2.5 nylon hex nuts to secure the Raspberry Pi 3 to the top plate as shown in Figure 6.30.

Figure 6.30. How to mount the Raspberry Pi 3 on the top plate.

### 3) Check the outcome

Figure 6.31 shows how the Raspberry Pi 3 should be mounted on the top plate of the chassis.

Figure 6.31. The Raspberry Pi 3 mounted on the top plate.

## 6.8. Top plate

From the Magician Chassis package take the following components:

- Bottom plate (with motors, wheels and standoffs attached) (1x)
- Top plate (with camera and Raspberry Pi 3 attached) (1x)
- M3x6 screws (4x)

Figure 6.32 shows the components needed to complete this part of the tutorial.

Figure 6.32. The parts needed to secure the top plate to the bottom plate.

### 1) Video tutorial

The following video shows how to secure the top plate on top of the bottom plate.

Figure 6.33

## 2) Step-by-step guide

*Step 1:*

Pass the motor wires through the openings in the top plate.

*Step 2:*

Use *4* M3x6 screws to secure the top plate to the nylon standoffs (mounted on the bottom plate in Section 6.4 - Chassis standoffs) as shown in Figure 6.34.

Figure 6.34. How to secure the top plate to the bottom plate.

## 3) Check the outcome

Figure 6.35 shows how the top plate should be mounted on the bottom plate.

Figure 6.35. The chassis completed.

## 6.9. USB Power cable

The power cable preparation is explained in Unit C-4 - Preparing the power cable (DB17).

## 6.10. DC Stepper Motor HAT

From the Duckiebot kit take the following components:

- USB power cable (prepared in Unit C-4 - Preparing the power cable (DB17)) (1x)
- DC Stepper Motor HAT (1x)
- M2.5x10 Nylon screws (or M2.5x12 nylon standoffs) (4x)

Figure 6.36 shows the components needed to complete this part of the tutorial.

Figure 6.36. The parts needed to add the DC Stepper Motor HAT to the Duckiebot.

### 1) Video tutorial

The following video shows how to connect the DC Stepper Motor HAT to the Raspberry Pi 3.

Figure 6.37

### 2) Step-by-step guide

*Step 1:*

Connect the wires of the USB power cable to the terminal block on the DC Stepper Motor HAT labeled as "5-12V Motor Power" as shown in Figure 6.38. The black wire goes to the negative terminal block (labeled with a minus: `-`) and the red wire goes to the positive terminal block (labeled with a plus: `+`).

Figure 6.38. How to connect the USB power cable to the DC Stepper Motor HAT.

*Step 2:*

Pass the free end of the camera cable through the opening in the DC Stepper Motor HAT as shown in Figure 6.39.

Figure 6.39. How to pass the camera cable through the opening in the DC Stepper Motor HAT.

*Step 3:*

Connect the free end of the camera cable to the **CAMERA** port on the Raspberry Pi 3 as shown in Figure 6.40.

Figure 6.40. How to connect the camera cable to the CAMERA port on the Raspberry Pi 3.

To do so, you will need to gently pull up on the black connector (it will slide up) to allow the cable to insert the port. Slide the connector back down to lock the cable in place, making sure it "clicks".

> Note: Make sure the camera cable is inserted in the right direction! The metal pins of the cable must be in contact with the metal terminals in the camera port of the PI. Please be aware that different camera cables have the text on different sides and with different orientation, **do not** use it as a landmark.

*Step 4:*

Attach the DC Stepper Motor HAT to the GPIO header on the Raspberry Pi 3. Make

sure that the GPIO stacking header of the Motor HAT is carefully aligned with the underlying GPIO pins before applying pressure.

> **Note:** In case you are using a short camera cable, ensure that the camera cable does not stand between the GPIO pins and the the GPIO header socket before applying pressure.

*Step 5:*

Secure the DC Stepper Motor HAT using *4* M2.5x10 nylon screws.

> **Note:** If you are planning on upgrading your Duckiebot to the configuration `DB17-1`, you can use *4* M2.5x12 nylon standoffs instead.

*Step 6:*

Connect the motor wires to the terminal block on the DC Stepper Motor HAT as shown in Figure 6.41.

Figure 6.41. How to connect the motor wires to the terminal block on the DC Stepper Motor HAT.

While looking at the Duckiebot from the back, identify the wires for left and right motor. Connect the left motor wires to the terminals labeled as **M1** and the right motor wires to the terminals labeled as **M2**. This will ensure that the pre-existing software that we will later install on the Duckiebot will send the commands to the correct motors.

### 3) Check the outcome

Figure 6.42 shows how the DC Stepper Motor HAT should be connected to the Raspberry Pi 3.

Figure 6.42. The DC Stepper Motor HAT connected to the Raspberry Pi 3.

## 6.11. Battery

From the Duckiebot kit take the following components:

- Battery (1x)
- Zip tie (1x)
- Short micro USB cable (1x)

Figure 6.43 shows the components needed to complete this part of the tutorial.

Figure 6.43. The parts needed to add the battery to the Duckiebot.

### 1) Video tutorial

The following video shows how to add the battery to the Duckiebot and turn it on.

Figure 6.44

### 2) Step-by-step guide

*Step 1:*
Pass the zip tie through the opening in the top plate.

*Step 2:*
Slide the battery between the two plates. Make sure it is above the zip tie.

*Step 3:*
Push the free end of the zip tie through the opening in the top plate.

*Step 4:*
Tighten the zip tie to secure the battery.

*Step 5:*
Connect the short micro USB cable to the Raspberry Pi 3.

*Step 6:*
Connect the short micro USB cable to the battery.

*Step 7:*
Connect the USB power cable to the battery.

*Step 8:*
Make sure that the LEDs on the Raspberry Pi 3 and the DC Stepper Motor HAT are on.

### 3) Check the outcome

Figure 6.45 shows how the battery should be installed on the Duckiebot.

Figure 6.45. The configuration `DB17` completed.

## 6.12. Upgrade to DB17-w

This upgrade equips the Duckiebot with a secondary, faster, Wi-Fi connection, ideal for image streaming. The new configuration is called `DB17-w`.

Figure 6.46 shows the components needed to complete this upgrade.

Figure 6.46. The parts needed to upgrade the Duckiebot to the configuration DB17-w.

1) Instructions

- Insert the USB WiFi dongle into one of the USB ports of the Raspberry Pi.

Figure 6.47. Upgrade to DB17-w completed.

## 6.13. Upgrade to DB17-j

This upgrade equips the Duckiebot with manual remote control capabilities. It is particularly useful for getting the Duckiebot out of tight spots or letting younger ones have a drive, in addition to providing handy shortcuts to different functions in development phase. The new configuration is called `DB17-j`.

Figure 6.48 shows the components needed to complete this upgrade.

Figure 6.48. The parts needed to upgrade the Duckiebot to the configuration DB17-j.

**Note:** The joystick comes with a USB receiver (as shown in Figure 6.48).

### 1) Instructions

- Insert the USB receiver into one of the USB ports of the Raspberry Pi.
- Insert 2 AA batteries on the back side of the joystick.
- Turn on the joystick by pressing the `HOME` button. Make sure that the LED above the `SELECT` button is steady.

Figure 6.49. Upgrade to DB17-j completed.

**TODO:** explain how to test the joystick with `jstest`

## 6.14. Upgrade to DB17-d

This upgrade equips the Duckiebot with an external hard drive that is convenient for storing videos (logs) as it provides both extra capacity and faster data transfer rates than the microSD card in the Raspberry Pi 3. Moreover, it is easy to unplug it from the Duckiebot at the end of the day and bring it over to a computer for downloading and analyzing stored data. The new configuration is called `DB17-d`.

Figure 6.50 shows the components needed to complete this upgrade.

Figure 6.50. The parts needed to upgrade the Duckiebot to the configuration DB17-d.

### 1) Instructions

- Insert the USB drive into one of the USB ports of the Raspberry Pi.

Figure 6.51. Upgrade to DB17-d completed.

- Mount your USB drive as explained in Unit J-15 - Mounting USB drives.

## 6.15. FAQ

*Q: If we have the bumpers, at what point should we add them?*

**Answer:** You shouldn't have the bumpers at this point. The function of the bumpers is to keep the LEDs in place, i.e., they belong to `DB17-l` configuration. These instructions cover the `DB17-wjd` configurations. You will find the bumper assembly instructions in Unit E-3 - Assembling the Duckiebot (`DB17-lc`).

*Q: Yeah but I still have the bumpers and am reading this page. So?*

**Answer:** The bumpers can be added after the Duckiebot assembly is complete.

*Q: I found it hard to mount the camera (the holes weren't lining up).*

**Answer:** Sometimes in life you have to push a little to make things happen. (But don't push too much or things will break!)

*Q: The long camera cable is a bit annoying - I folded it and shoved it in between two hats.*

**Answer:** The shorter cable is even more annoying. We suggest wrapping the long camera cable between the chassis and the Raspberry Pi. With some strategic planning, you can use the zipties that keep the battery in place to hold the camera cable in place as well (see figure below-to add)

**TODO:** add pretty cable handling pic

*Q: I found that the screwdriver that comes with the chassis kit is too fat to screw in the wires on the hat.*

**Answer:** It is possible you got one of the fatter screwdrivers. You will need to figure it out yourself (or ask a TA for help).

*Q: I need something to cut the end of the zip tie with.*

**Answer:** Scissors typically work out for these kind of jobs (and no, they're not provided in a Fall 2017 Duckiebox).

# Unit C-7
# Reproducing the image

These are the instructions to reproduce the Ubuntu image that we use.

**Note:** Please note that the image is already available, so you don't need to do this.

However, this documentation is useful if you would like to port the software to a different distribution. Also, we periodically run through these instructions to make sure that they work.

**Note:** Just in case, let's re-state this: in Fall 2017, you don't necessarily need to do the following.

Knowledge and activity graph

**Requires:** Internet connection to download the packages.

**Requires:** A PC running any Linux with an SD card reader.

**Requires:** Time: about 4 hours (most of it spent waiting for things to download/compile).

**Results:** A baseline Ubuntu Mate 16.04.2 image with updated software.

## 7.1. Download and uncompress the Ubuntu Mate image

Download the image from the page

https://ubuntu-mate.org/download/

The file we are looking for is:

```
filename: ubuntu-mate-16.04.2-desktop-armhf-raspberry-pi.img.xz
    size: 1.2 GB
  SHA256: dc3afcad68a5de3ba683dc30d2093a3b5b3cd6b2c16c0b5de8d50fede78f75c2
```

After download, run the command `sha256sum` to make sure you have the right version:

```
$ sha256sum ubuntu-mate-16.04.2-desktop-armhf-raspberry-pi.img.xz
dc3afcad68a5de3ba683dc30d2093a3b5b3cd6b2c16c0b5de8d50fede78f75c2
```

If the string does not correspond exactly, your download was corrupted. Delete the file and try again.

Then decompress using the command `xz`:

```
$ xz -d ubuntu-mate-16.04.2-desktop-armhf-raspberry-pi.img.xz
```

## 7.2. Burn the image to an SD card

Next, burn the image on to the SD card.

➡ This procedure is explained in Section 17.2 - How to burn an image to an SD card.

#### 1) Verify that the SD card was created correctly

Remove the SD card and plug it in again in the laptop.
Ubuntu will mount two partitions, by the name of `PI_ROOT` and `PI_BOOT`.

#### 2) Installation

Boot the disk in the Raspberry Pi.
Choose the following options:

```
language: English
username: ubuntu
password: ubuntu
hostname: duckiebot
```

Choose the option to log in automatically.
Reboot.

#### 3) Update installed software

The WiFi was connected to airport network `duckietown` with password `quackquack`.
Afterwards I upgraded all the software preinstalled with these commands:

```
$ sudo apt update
$ sudo apt dist-upgrade
```

Expect `dist-upgrade` to take quite a long time (up to 2 hours).

## 7.3. Raspberry Pi Config

The Raspberry Pi is not accessible by SSH by default.
Run `raspi-config`:

```
$ sudo raspi-config
```

choose “3. Interfacing Options”, and enable SSH,
We need to enable the camera and the I2C bus.
choose “3. Interfacing Options”, and enable camera, and I2C.
Also disable the graphical boot
choose “2. Boot Options”, configure option for startup. –>B1. Console Text console

## 7.4. Install packages

Install these packages.

Etckeeper:

```
$ sudo apt install etckeeper
```

Editors / shells:

```
$ sudo apt install -y vim emacs byobu zsh
```

Git:

```
$ sudo apt install -y git git-extras
```

Other:

```
$ sudo apt install htop atop nethogs iftop
$ sudo apt install aptitude apt-file
```

Development:

```
$ sudo apt install -y build-essential libblas-dev liblapack-dev libatlas-base-dev gfortran
libyaml-cpp-dev raspberrypi-kernel-headers
```

Python:

```
$ sudo apt install -y python-dev ipython python-sklearn python-smbus
$ sudo apt install -y python-termcolor
$ sudo apt install python-frozendict
$ sudo apt install python-tables
$ pip install comptests
$ pip install procgraph
$ sudo pip install scipy --upgrade
$ sudo pip install ruamel.yaml --upgrade
```

**Note:** scipy –upgrade(0.19.1) took about an hour with ethernet connection.

I2C:

```
$ sudo apt install -y i2c-tools
```

## 7.5. Install Edimax driver

First, mark the kernel packages as not upgradeable:

```
$ sudo apt-mark hold raspberrypi-kernel raspberrypi-kernel-headers
raspberrypi-kernel set on hold.
raspberrypi-kernel-headers set on hold
```

Then, download and install the Edimax driver from this repository.

```
$ git clone git@github.com:duckietown/rtl8822bu.git
$ cd rtl8822bu
$ make
$ sudo make install
```

## 7.6. Install ROS

Install ROS.

➡ The procedure is given in Section 30.1 - Install ROS.

## 7.7. Wireless configuration (old version)

There are two files that are important to edit.
The file `/etc/network/interfaces` should look like this:

```
# interfaces(5) file used by ifup(8) and ifdown(8)
# Include files from /etc/network/interfaces.d:
#source-directory /etc/network/interfaces.d

auto wlan0

# The loopback network interface
auto lo
iface lo inet loopback

# Wireless network interface
allow-hotplug wlan0
iface wlan0 inet dhcp
wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf
iface default inet dhcp
```

The file `/etc/wpa_supplicant/wpa_supplicant.conf` should look like this:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1

network={
ssid="duckietown"
psk="quackquack"
proto=RSN
key_mgmt=WPA-PSK
pairwise=CCMP
auth_alg=OPEN
}
network={
   key_mgmt=NONE
}
```

## 7.8. Wireless configuration

The files that describe the network configuration are in the directory

```
/etc/NetworkManager/system-connections/
```

This is the contents of the connection file `duckietown`, which describes how to connect to the `duckietown` wireless network:

```
[connection]
id=duckietown
uuid=e9cef1bd-f6fb-4c5b-93cf-cca837ec35f2
type=wifi
permissions=
secondaries=
timestamp=1502254646

[wifi]
mac-address-blacklist=
mac-address-randomization=0
mode=infrastructure
ssid=duckietown

[wifi-security]
group=
key-mgmt=wpa-psk
pairwise=
proto=
psk=quackquack

[ipv4]
dns-search=
method=auto

[ipv6]
addr-gen-mode=stable-privacy
dns-search=
ip6-privacy=0
method=auto
```

This is the file

```
/etc/NetworkManager/system-connections/create-5ghz-network
```

Contents:

```
[connection]
id=create-5ghz-network
uuid=7331d1e7-2cdf-4047-b426-c170ecc16f51
type=wifi
# Put the Edimax interface name here:
interface-name=wlx74da38c9caa0 - to change
permissions=
secondaries=
timestamp=1502023843

[wifi]
band=a
# Put the Edimax MAC address here
mac-address=74:DA:38:C9:CA:A0 - to change
mac-address-blacklist=
mac-address-randomization=0
mode=ap
seen-bssids=
ssid=duckiebot-not-configured

[ipv4]
dns-search=
method=shared

[ipv6]
addr-gen-mode=stable-privacy
dns-search=
ip6-privacy=0
method=ignore
```

Note that there is an interface name and MAC address that need to be changed on each PI.

## 7.9. SSH server config

This enables the SSH server:

```
$ sudo systemctl enable ssh
```

## 7.10. Create swap Space

Do the following:

Create an empty file using the `dd` (device-to-device copy) command:

```
$ sudo dd if=/dev/zero of=/swap0 bs=1M count=512
```

This is for a 512 MB swap space.

Format the file for use as swap:

```
$ sudo mkswap /swap0
```

Add the swap file to the system configuration:

```
$ sudo vi /etc/fstab
```

Add this line to the bottom:

```
/swap0 swap swap
```

Activate the swap space:

```
$ sudo swapon -a
```

## 7.11. Passwordless sudo

First, make `vi` the default editor, using

```
$ sudo update-alternatives --config editor
```

and then choose `vim.basic`.
Then run:

```
$ sudo visudo
```

And then change this line:

```
%sudo   ALL=(ALL:ALL) ALL
```

into this line:

```
%sudo   ALL=(ALL:ALL) NOPASSWD:ALL
```

## 7.12. Clean up

You can use the command `dpigs` to find out which packages take lots of space.

```
$ sudo apt install wajig  debian-goodies
```

Either:

```
$ wajig large
$ dpigs -H -n 20
```

Stuff to remove:

```
$ sudo apt remove thunderbird
$ sudo apt remove libreoffice-\*
$ sudo apt remove openjdk-8-jre-headless
$ sudo apt remove fonts-noto-cjk
$ sudo apt remove brasero
```

At the end, remove extra dependencies:

```
$ sudo apt autoremove
```

And remove the `apt` cache using:

```
$ sudo apt clean
```

The total size should be around 6.6GB.

## 7.13. Ubuntu user configuration

### 1) Groups

You should make the `ubuntu` user belong to the `i2c` and `input` groups:

```
$ sudo adduser ubuntu i2c
$ sudo adduser ubuntu input
$ sudo adduser ubuntu video
```

> You may need to do the following (but might be done already through `raspi-config`): XXX

```
$ sudo udevadm trigger
```

### 2) Basic SSH config

Do the basic SSH config.

➡ The procedure is documented in Section 19.3 - Local configuration.

> **Note:** this is not in the aug10 image.

### 3) Passwordless SSH config

Add `.authorized_keys` so that we can all do passwordless SSH.
The key is at the URL

```
https://www.dropbox.com/s/pxyou3qy1p8m4d0/duckietown_key1.pub?dl=1
```

Download to `.ssh/authorized_keys`:

```
$ curl -o .ssh/authorized_keys URL above
```

#### 4) Shell prompt

Add the following lines to `~ubuntu/.bashrc`:

```
echo ""
echo "Welcome to a duckiebot!"
echo ""
echo "Reminders:"
echo ""
echo "1) Do not use the user 'ubuntu' for development - create your own user."
echo "2) Change the name of the robot from 'duckiebot' to something else."
echo ""

export EDITOR=vim
```

## 7.14. Check that all required packages were installed

At this point, before you copy/distribute the image, create a user, install the software, and make sure that `what-the-duck` does not complain about any missing package.
(Ignore `what-the-duck`'s errors about things that are not set up yet, like users.)

## 7.15. Creating the image

You may now want to create an image that you can share with your friends. They will think you are cool because they won't have to duplicate all of the work that you just did. Luckily this is easy. Just power down the duckiebot with:

```
$ sudo shutdown -h now
```

and put the SD card back in your laptop.

➡ The procedure of how to burn an image is explained in Section 17.2 - How to burn an image to an SD card; except you will invert the `if` and `of` destinations.

You may want to subsequently shrink the image, for example if your friends have smaller SD cards than you.

➡ The procedure of how to shrink an image is explained in Section 17.3 -

How to shrink an image.

## 7.16. TODO: Git LFS

Note: We should install Git LFS on the Raspberry Pi, but so far AC did not have any luck. See Section 28.1 - Generic installation instructions.

# Unit C-8
# Installing Ubuntu on laptops

Assigned to: Andrea

Before you prepare the Duckiebot, you need to have a laptop with Ubuntu installed.

Knowledge and activity graph

**Requires:** A laptop with free disk space.
**Requires:** Internet connection to download the Ubuntu image.
**Requires:** About 30 minutes.
**Results:** A laptop ready to be used for Duckietown.

## 8.1. Install Ubuntu

Install Ubuntu 16.04.3.

➡ For instructions, see for example this online tutorial.

**On the choice of username:** During the installation, create a user for yourself with a username different from `ubuntu`, which is the default. Otherwise, you may get confused later.

## 8.2. Install useful software

Use `etckeeper` to keep track of the configuration in `/etc`:

```
$ sudo apt install etckeeper
```

Install `ssh` to login remotely and the server:

```
$ sudo apt install ssh
```

Use `byobu`:

```
$ sudo apt install byobu
```

Use `vim`:

```
$ sudo apt install vim
```

Use `htop` to monitor CPU usage:

```
$ sudo apt install htop
```

Additional utilities for `git`:

```
$ sudo apt install git git-extras
```

Other utilities:

```
$ sudo apt install avahi-utils ecryptfs-utils
```

## 8.3. Install ROS

Install ROS on your laptop.

➡ The procedure is given in Section 30.1 - Install ROS.

## 8.4. Other suggested software

### 1) Redshift

This is Flux for Linux. It is an accessibility/lab safety issue: bright screens damage eyes and perturb sleep [6].

Install redshift and run it.

```
$ sudo apt install redshift-gtk
```

Set to "autostart" from the icon (on the panel - near wifi/lan).

## 8.5. Passwordless sudo

Set up passwordless `sudo`.

➡ This procedure is described in Section 7.11 - Passwordless sudo.

+ comment

Huh I don't know - this is great for usability, but horrible for security. If you step away from your laptop for a second and don't lock the screen, a nasty person could `sudo rm -rf /`. -FG

## 8.6. SSH and Git setup

### 1) Basic SSH config

Do the basic SSH config.

➡ The procedure is documented in Section 19.3 - Local configuration.

### 2) Create key pair for ***username***

Next, create a private/public key pair for the user; call it *username@robot name*.

➡ The procedure is documented in Section 19.5 - Creating an SSH key-pair.

### 3) Add ***username***'s public key to Github

Add the public key to your Github account.

➡ The procedure is documented in Section 29.3 - Add a public key to Github.

If the step is done correctly, this command should succeed:

```
$ ssh -T git@github.com
```

### 4) Local Git setup

Set up Git locally.

➡ The procedure is described in Section 27.3 - Setting up global configurations for Git.

## 8.7. Installation of the duckuments system

Optional but very encouraged: install the duckuments system. This will allow you to have a local copy of the documentation and easily submit questions and changes.

➡ The procedure is documented in Section 1.3 - Installing the documentation system.

# Unit C-9
# Duckiebot Initialization

Assigned to: Andrea

Knowledge and activity graph

Requires: An SD card of dimensions at least 16 GB.

Requires: A computer with an internet connection, an SD card reader, and 16 GB of free space.

Requires: An assembled Duckiebot in configuration `DB17`. This is the result of Unit C-5 - Assembling the Duckiebot (`DB17-jwd`).

Results: A Duckiebot that is configured correctly, that you can connect to with your laptop and hopefully also has internet access

## 9.1. Acquire and burn the image

On the laptop, download the compressed image at this URL:

https://www.dropbox.com/s/ckpqpp0cav3aucb/duckiebot-RPI3-AD-2017-09-12.img.xz?dl=1

The size is 1.7 GB.

You can use:

```
$ wget -O duckiebot-RPI3-AD-2017-09-12.img.xz URL above
```

+ comment

The original was:

```
$ curl -o duckiebot-RPI3-AD-2017-09-12.img.xz URL above
```

It looks like that `curl` cannot be used with Drobpox links because it does not follow redirects.

To make sure that the image is downloaded correctly, compute its hash using the program `sha256sum`:

```
$ sha256sum duckiebot-RPI3-AD-2017-09-12.img.xz
7136f9049b230de68e8b2d6df29ece844a3f830cc96014aaa92c6d3f247b6130
duckiebot-RPI3-AD-2017-09-12.img.xz
```

Compare the hash that you obtain with the hash above. If they are different, there was some problem in downloading the image.

Uncompress the file:

```
$ xz -d -k --verbose duckiebot-RPI3-AD-2017-09-12.img.xz
```

This will create a file of 11 GB in size.

Next, burn the image on disk.

> ➡ The procedure of how to burn an image is explained in Section 17.2 - How to burn an image to an SD card.

## 9.2. Turn on the Duckiebot

Put the SD Card in the Duckiebot.

Turn on the Duckiebot by connecting the power cable to the battery.

TODO: Add figure

+ comment

In general, for the battery: if it's off, a single click on the power button will turn the battery on. When it's on, a single click will show you the charge indicator (4 white lights = full), and holding the button for 3s will turn off the battery. Shutting down the Duckiebot is not recommended because it may cause corruption of the SD card.

## 9.3. Connect the Duckiebot to a network

You can login to the Duckiebot in two ways:

1. Through an Ethernet cable.
2. Through a `duckietown` WiFi network.

In the worst case, you can use an HDMI monitor and a USB keyboard.

### 1) Option 1: Ethernet cable

Connect the Duckiebot and your laptop to the same network switch.

Allow 30 s - 1 minute for the DHCP to work.

### 2) Option 2: Duckietown network

The Duckiebot connects automatically to a 2.4 GHz network called "`duckietown`" and password "`quackquack`".

Connect your laptop to the same wireless network.

## 9.4. Ping the Duckiebot

To test that the Duckiebot is connected, try to ping it.

The hostname of a freshly-installed duckiebot is `duckiebot-not-configured`:

```
$ ping duckiebot-not-configured.local
```

You should see output similar to the following:

```
PING duckiebot-not-configured.local (X.X.X.X): 56 data bytes
64 bytes from X.X.X.X: icmp_seq=0 ttl=64 time=2.164 ms
64 bytes from X.X.X.X: icmp_seq=1 ttl=64 time=2.303 ms
...
```

## 9.5. SSH to the Duckiebot

Next, try to log in using SSH, with account `ubuntu`:

```
$ ssh ubuntu@duckiebot-not-configured.local
```

The password is `ubuntu`.

By default, the robot boots into Byobu.

Please see Unit J-26 - Byobu for an introduction to Byobu.

+ doubt

Not sure it's a good idea to boot into Byobu. -??

## 9.6. Setup network

➡ Unit C-10 - Networking aka the hardest part

## 9.7. Update the system

Next, we need to update to bring the system up to date.

Use these commands

```
$ sudo apt update
$ sudo apt dist-upgrade
```

## 9.8. Give a name to the Duckiebot

It is now time to give a name to the Duckiebot.

These are the criteria:

- It should be a simple alphabetic string (no numbers or other characters like "`-`", "`_`", etc.) .
- It will always appear lowercase.
- It cannot be a generic name like "`duckiebot`", "`robot`" or similar.

From here on, we will refer to this string as "*robot name*". Every time you see *robot name*, you should substitute the name that you chose.

## 9.9. Change the hostname

We will put the robot name in configuration files.

**Note:** Files in `/etc` are only writable by `root`, so you need to use `sudo` to edit them. For example:

```
$ sudo vi filename
```

Edit the file

```
/etc/hostname
```

and put "*robot name*" instead of `duckiebot-not-configured`.
Also edit the file

```
/etc/hosts
```

and put "*robot name*" where `duckiebot-not-configured` appears.
The first two lines of `/etc/hosts` should be:

```
127.0.0.1   localhost
127.0.1.1   robot name
```

**Note:** there is a command `hostname` that promises to change the hostname. However, the change given by that command does not persist across reboots. You need to edit the files above for the changes to persist.

**Note:** Never add other hostnames in `/etc/hosts`. It is a tempting fix when DNS does not work, but it will cause other problems subsequently.

Then reboot the Raspberry Pi using the command

```
$ sudo reboot
```

After reboot, log in again, and run the command `hostname` to check that the change has persisted:

```
$ hostname
robot name
```

## 9.10. Expand your filesystem

If your SD card is larger than the image, you'll want to expand the filesystem on your robot so that you can use all of the space available. Achieve this with:

```
$ sudo raspi-config --expand-rootfs
```

and then reboot

```
$ sudo shutdown -r now
```

once rebooted you can test whether this was successful by doing

```
$ df -lh
```

the output should give you something like:

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/root        15G  6.3G  8.2G  44% /
devtmpfs        303M     0  303M   0% /dev
tmpfs           431M     0  431M   0% /dev/shm
tmpfs           431M   12M  420M   3% /run
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           431M     0  431M   0% /sys/fs/cgroup
/dev/mmcblk0p1   63M   21M   43M  34% /boot
tmpfs            87M     0   87M   0% /run/user/1000
```

You should see that the Size of your `/dev/root` Filesystem is “close” to the size of your SD card.

## 9.11. Create your user

You must not use the `ubuntu` user for development. Instead, you need to create a new user.

Choose a user name, which we will refer to as *username*.

To create a new user:

```
$ sudo useradd -m username
```

Make the user an administrator by adding it to the group `sudo`:

```
$ sudo adduser username sudo
```

Make the user a member of the groups `input`, `video`, and `i2c`

```
$ sudo adduser username input
$ sudo adduser username video
$ sudo adduser username i2c
```

Set the shell `bash`:

```
$ sudo chsh -s /bin/bash username
```

To set a password, use:

```
$ sudo passwd username
```

At this point, you should be able to login to the new user from the laptop using the password:

```
$ ssh username@robot name
```

Next, you should repeat some steps that we already described.

+ comment
What steps?? -LP

### 1) Basic SSH config

Do the basic SSH config.

➡ The procedure is documented in Section 19.3 - Local configuration.

### 2) Create key pair for *username*

Next, create a private/public key pair for the user; call it *username@robot name*.

➡ The procedure is documented in Section 19.5 - Creating an SSH key-pair.

### 3) Add SSH alias

Once you have your SSH key pair on both your laptop and your Duckiebot, as well as your new user- and hostname set up on your Duckiebot, then you should set up an SSH alias as described in Section 15.1 - SSH aliases. This allows your to log in for example with

```
$ ssh abc
```

instead of

```
$ ssh username@robot name
```

where you can chose *abc* to be any alias / shortcut.

### 4) Add *username*'s public key to Github

Add the public key to your Github account.

➡ The procedure is documented in Section 29.3 - Add a public key to Github.

If the step is done correctly, the following command should succeed and give you a welcome message:

```
$ ssh -T git@github.com
Hi username! You've successfully authenticated, but GitHub does not provide shell
access.
```

#### 5) Local Git configuration

➡ This procedure is in Section 27.3 - Setting up global configurations for Git.

#### 6) Set up the laptop-Duckiebot connection

Make sure that you can login passwordlessly to your user from the laptop.

➡ The procedure is explained in Section 19.6 - How to login without a password. In this case, we have: *local* = laptop, *local-user* = your local user on the laptop, *remote* = *robot name*, *remote-user* = *username*.

If the step is done correctly, you should be able to login from the laptop to the robot, without typing a password:

```
$ ssh username@robot name
```

#### 7) Some advice on the importance of passwordless access

In general, if you find yourself:

- typing an IP
- typing a password
- typing `ssh` more than once
- using a screen / USB keyboard

it means you should learn more about Linux and networks, and you are setting yourself up for failure.

Yes, you "can do without", but with an additional 30 seconds of your time. The 30 seconds you are not saving every time are the difference between being productive roboticists and going crazy.

Really, it is impossible to do robotics when you have to think about IPs and passwords...

## 9.12. Other customizations

If you know what you are doing, you are welcome to install and use additional shells, but please keep Bash as be the default shell. This is important for ROS installation.

For the record, our favorite shell is ZSH with `oh-my-zsh`.

## 9.13. Hardware check: camera

Check that the camera is connected using this command:

```
$ vcgencmd get_camera
supported=1 detected=1
```

If you see `detected=0`, it means that the hardware connection is not working.
You can test the camera right away using a command-line utility called `raspistill`.
Use the `raspistill` command to capture the file `out.jpg`:

```
$ raspistill -t 1 -o out.jpg
```

Then download `out.jpg` to your computer using `scp` for inspection.

➡ For instructions on how to use `scp`, see Subsection 21.1.1 - Download a file with SCP.

## 9.14. Final touches: duckie logo

In order to show that your Duckiebot is ready for the task of driving around happy little duckies, the robot has to fly the Duckietown flag. When you are still logged in to the Duckiebot you can download and install the banner like this:
Download the ANSI art file from Github:

```
$ wget --no-check-certificate -O duckie.art "https://raw.githubusercontent.com/
duckietown/Software/master/misc/duckie.art"
```

(optional) If you want, you can preview the logo by just outputting it onto the command line:

```
$ cat duckie.art
```

Next up create a new empty text file in your favorite editor and add the code for showing your duckie pride:
Let's say I use `nano`, I open a new file:

```
$ nano 20-duckie
```

And in there I add the following code (which by itself just prints the duckie logo):

```
#!/bin/sh
printf "\n$(cat /etc/update-motd.d/duckie.art)\n"
```

Then save and close the file. Finally you have to make this file executable...

```
$ chmod +x 20-duckie
```

...and copy both the duckie logo and the script into a specific directory `/etc/update-motd.d` to make it appear when you login via SSH. `motd` stands for "message of the day". This is a mechanism for system administrators to show users news and messages when they login. Every executable script in this directory which has a filename a la *NN-some name* will get exected when a user logs in, where *NN* is a two digit number that indicates the order.

```
sudo cp duckie.art /etc/update-motd.d
sudo cp 20-duckie /etc/update-motd.d
```

Finally log out of SSH via `exit` and log back in to see duckie goodness.

### 1) Troubleshooting

**Symptom:** `detected=0`

**Resolution:** If you see `detected=0`, it is likely that the camera is not connected correctly.

If you see an error that starts like this:

```
mmal: Cannot read camera info, keeping the defaults for OV5647
...
mmal: Camera is not detected. Please check carefully the camera module is installed
correctly.
```

then, just like it says: "Please check carefully the camera module is installed correctly.".

**Symptom:** random `wget`, `curl`, `git`, and `apt` calls fail with SSL errors.

**Resolution:** That's probably actually an issue with your system time. Type the command `timedatectl` into a terminal, hit enter and see if the time is off. If it is, you might want to follow the intructions from this article, or entirely uninstall your NTP service and manually grab the time on reboot. It's a bit dirty, but works surprisingly well.

**Symptom:** Cannot find `/etc` folder for configuring the Wi-Fi. I only see `Desktop`, `Downloads` when starting up the Duckiebot.

**Resolution:** If a directory name starts with `/`, it's not supposed to be in the home directory, but rather at the root of the filesystem. You are currently in `/home/ubuntu`. Type `ls /` to see the folders at the root, including `/etc.

# Unit C-10
# Networking aka the hardest part

Knowledge and activity graph

**Requires:** A Duckiebot in configuration `DB17-C0+w`

**Requires:** Either a router that you have control over that has internet access, or your credentials for connecting to an existing wireless network

**Requires:** Patience (channel your inner Yoda)

**Results:** A Duckiebot that you can connect to and that is connected to the internet

**Note:** this page is primarily for folks operating with the "two-network" configuration, `C0+w`. For a one adapter setup you will can skip directly to Section 10.2 - Setting up wireless network configuration, but you will have to connect to a network that you can ssh through.

The basic idea is that we are going to use the "Edimax" thumbdrive adapter to create a dedicated wireless network that you can always connect to with your laptop. Then we are going to use the built-in Broadcom chip on the Pi to connect to the internet, and then the network will be bridged.

## 10.1. (For `DB17-w`) Configure the robot-generated network

This part should work every time with very low uncertainty.

The Duckiebot in configuration `C0+w` can create a WiFi network.

It is a 5 GHz network; this means that you need to have a 5 GHz WiFi adapter in your laptop.

First, make sure that the Edimax is correctly installed. Using `iwconfig`, you should see four interfaces:

```
$ iwconfig
wlxAABBCCDDEEFFGG  unassociated  Nickname:"rtl8822bu"

...

lo        no wireless extensions.

enxb827eb1f81a4  no wireless extensions.

wlan1     IEEE 802.11bgn  ESSID:"duckietown"

...
```

Make note of the name `wlxAABBCCDDEEFFGG`.

Look up the MAC address using the command:

```
$ ifconfig wlxAABBCCDDEEFFGG
wlxAABBCCDDEEFFGG Link encap:Ethernet  HWaddr AA:BB:CC:DD:EE:FF:GG
```

Then, edit the connection file

```
/etc/NetworkManager/system-connections/create-5ghz-network
```

Make the following changes:

- Where it says `interface-name=...`, put "`wlxAABBCCDDEEFFGG`".
- Where it says `mac-address=...`, put "`AA:BB:CC:DD:EE:FF:GG`".
- Where it says `ssid=duckiebot-not-configured`, put "`ssid=robot name`".

![Newly upgraded]

To ensure nobody piggybacks on our connection, which poses a security risk especially in a public environment, we will protect access to the 5 GHz WiFi through a password. To set a password you will need to log in the Duckiebot with the default "ubuntu" username and password and change your system files. In the `/etc/NetworkManager/system-connections/create-5ghz-network`, add:

```
[wifi-security]
key-mgmt=wpa-psk
psk=YOUR_OWN_WIFI_PASSWORD_NO_QUOTATION_MAKRS_NEEDED
auth-alg=open
```

and then reboot.

At this point you should see a new network being created named "*robot name*", protected by the password you just set.

+ comment

Make sure the password contains min. 8 character or combined with numbers. If no networks shows up after the configuration and no feedback from system, please check the content of the file again. The program, which activates the Edimax wifi adapter is very sensitive to its content.

?? Adding a password to your 5GHz connection is a mandatory policy in the Zurich branch.

## 10.2. Setting up wireless network configuration

You are connected to the Duckiebot via WiFi, but the Duckiebot also needs to connect to the internet in order to get updates and install some software. This part is a little bit more of a "black art" since we cannot predict every possible network configurations. Below are some settings that have been verified to work in different situations:

### 1) Option 1: `duckietown` WiFi

Check with your phone or laptop if there is a WiFi in reach with the name of `duckietown`. If there is, you are all set. The defaut configuration for the Duckiebot is to have one

WiFi adapter connect to this network and the other broadcast the access point which you are currently connected to.

### 2) Option 2.a): `eduroam` WiFi (Non-UdeM/McGill instructions)

If there should be no `duckietown` network in reach then you have to manually add a network configuration file for the network that you'd like to connect to. Most universities around the world should have to `eduroam` network available. You can use it for connecting your Duckiebot.

Save the following block as new file in `/etc/NetworkManager/system-connections/eduroam`:

```
[connection]
id=eduroam
uuid=38ea363b-2db3-4849-a9a4-c2aa3236ae29
type=wifi
permissions=user:oem:;
secondaries=

[wifi]
mac-address=the MAC address of your internal wifi adapter, wlan0
mac-address-blacklist=
mac-address-randomization=0
mode=infrastructure
seen-bssids=
ssid=eduroam

[wifi-security]
auth-alg=open
group=
key-mgmt=wpa-eap
pairwise=
proto=

[802-1x]
altsubject-matches=
eap=ttls;
identity=your eduroam username@your eduroam domain
password=your eduroam password
phase2-altsubject-matches=
phase2-auth=pap

[ipv4]
dns-search=
method=auto

[ipv6]
addr-gen-mode=stable-privacy
dns-search=
method=auto
```

Set the permissions on the new file to 0600.

```
sudo chmod 0600 /etc/NetworkManager/system-connections/eduroam
```

### 3) Option 2.b): eduroam WiFi (UdeM/McGill instructions)

Save the following block as new file in `/etc/NetworkManager/system-connections/eduroam-`*USERNAME* : where USERNAME is the your logged-in username in the duckiebot.

```
[connection]
id=eduroam
uuid=38ea363b-2db3-4849-a9a4-c2aa3236ae29
type=wifi
permissions=user:USERNAME:;
secondaries=

[wifi]
mac-address=the MAC address of your internal wifi adapter, wlan0
mac-address-blacklist=
mac-address-randomization=0
mode=infrastructure
seen-bssids=
ssid=eduroam

[wifi-security]
auth-alg=open
group=
key-mgmt=wpa-eap
pairwise=
proto=

[802-1x]
altsubject-matches=
eap=peap;
identity=DGTIC UNIP
password=DGTIC PWD
phase2-altsubject-matches=
phase2-auth=mschapv2

[ipv4]
dns-search=
method=auto

[ipv6]
addr-gen-mode=stable-privacy
dns-search=
method=auto
```

Set the permissions on the new file to 0600.

```
sudo chmod 0600 /etc/NetworkManager/system-connections/eduroam-USERNAME
```

### 4) Option 3 (For Univeristé de Montréal students only): Use `UdeM avec cryptage`

**TODO:** someone replicate please - LP

> **Note:** your can use the `autoconnect-priority=XX` inside the `[connection]` block to establish a priority. If you want to connect to one network preferentially if two are available then give it a higher priority.

Save the following block as new file in `/etc/NetworkManager/system-connections/secure`:

```
[connection]
id=secure
uuid=e9cef1bd-f6fb-4c5b-93cf-cca837ec35f2
type=wifi
permissions=
secondaries=
timestamp=1502254646
autoconnect-priority=100

[wifi]
mac-address-blacklist=
mac-address-randomization=0
mode=infrastructure
ssid=UdeM avec cryptage
security=wifi-security

[wifi-security]
key-mgmt=wpa-eap

[802-1x]
eap=peap;
identity=DGTIC UNIP
phase2-auth=mschapv2
password=DGTIC PWD

[ipv4]
dns-search=
method=auto

[ipv6]
addr-gen-mode=stable-privacy
dns-search=
ip6-privacy=0
method=auto
```

Set the permissions on the new file to 0600.

```
sudo chmod 600 /etc/NetworkManager/system-connections/secure
```

### 5) Option 4: custom WiFi

First run the following to see what networks are available:

```
$ nmcli dev wifi list
```

You should see the network that you are trying to connect ( *SSID* ) to and you should know the password. To connect to it run:

```
$ sudo nmcli dev wifi con SSID password PASSWORD
```

### 6) Option 5: ETH Wifi

The following instructions will lead you to connect your PI to the “eth” wifi network.

First, run the following on duckiebot

```
$ iwconfig
...

lo        no wireless extensions.

enxbxxxxxxxxxxx  no wireless extensions.

...
```

Make note of the name `enxbxxxxxxxxxxx`. *xxxxxxxxxxx* should be a string that has 11 characters that is formed by numbers and lower case letters.

Second, edit the file `/etc/network/interfaces` which requires `sudo` so that it looks like the following, and make sure the `enxbxxxxxxxxxxx` matches.

Pay special attention on the line “pre-up wpa_supplicant -B -D wext -i wlan0 -c /etc/wpa_supplicant/wpa_supplicant.conf”.This is expected to be exactly one line instead of two but due to formatting issue it is shown as two lines.

Also, make sure every characters match exactly with the provided ones. TAs will not help you to do spelling error check.

```
# interfaces(5) file used by ifup(8) and ifdown(8) Include files from /etc/network/
interfaces.d:
source-directory /etc/network/interfaces.d

# The loopback network interface
auto lo
auto enxbxxxxxxxxxxxx

# the wired network setting
iface enxbxxxxxxxxxxxx inet dhcp

# the wireless network setting
auto wlan0
allow-hotplug wlan0
iface wlan0 inet dhcp
    pre-up wpa_supplicant -B -D wext -i wlan0 -c /etc/wpa_supplicant/wpa_supplicant.conf
    post-down killall -q wpa_supplicant
```

Third, edit the file `/etc/wpa_supplicant/wpa_supplicant.conf` which requires `sudo` so that it looks like the following, and make sure you substitute [identity] and [password] content with your eth account information:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1

network={
    ssid="eth"
    key_mgmt=WPA-EAP
    group=CCMP TKIP
    pairwise=CCMP TKIP
    eap=PEAP
    proto=RSN
    identity="your user name goes here"
    password="your password goes here"
    phase1="peaplabel=0"
    phase2="auth=MSCHAPV2"
    priority=1
}
```

Fourth, reboot your PI.

```
$ sudo reboot
```

Then everything shall be fine. The PI will connect to “eth” automatically everytime it starts.

Note that, if something went wrong, your Duckiebot tries to connect to the network for 5.5mins at startup while it’s blocking SSH connection to it completely (“Connection refused” error when connecting). If this is the case, please wait those 5.5mins

until your Duckiebot lets you connect again and recheck your settings.

**TODO:** Find a solution to this since it occurs very often

# Unit C-11
# Software setup and RC remote control

Knowledge and activity graph

**Requires:** Laptop configured, according to Unit C-8 - Installing Ubuntu on laptops.

**Requires:** You have configured the Duckiebot. The procedure is documented in Unit C-9 - Duckiebot Initialization.

**Requires:** You have created a Github account and configured public keys, both for the laptop and for the Duckiebot. The procedure is documented in Unit J-29 - Setup Github access.

**Results:** You can run the joystick demo.

## 11.1. Clone the Duckietown repository

Clone the repository in the directory `~/duckietown`:

```
$ git clone git@github.com:duckietown/Software.git ~/duckietown
```

For the above to succeed you should have a Github account already set up.

It should not ask for a password.

**Note:** you must not clone the repository using the URL starting with `https`. Later steps will fail.

### 1) Troubleshooting

**Symptom:** It asks for a password.

**Resolution:** You missed some of the steps described in Unit J-29 - Setup Github access.

**Symptom:** Other weird errors.

**Resolution:** Probably the time is not set up correctly. Use `ntpdate` as above:

```
$ sudo ntpdate -u us.pool.ntp.org
```

Or see the hints in the troubleshooting section on the previous page.

## 11.2. Update the system

The software used for the Duckiebots changes every day, this means that also the dependencies change. In order to check whether your system meets all the requirements for running the software and install all the missing packages (if any), we can run the following script:

```
$ cd ~/duckietown
$ /bin/bash ./dependencies_for_duckiebot.sh
```

This command will install only the packages that are not already installed in your system.

## 11.3. Set up the ROS environment on the Duckiebot

All the following commands should be run in the `~/duckietown` directory:

```
$ cd ~/duckietown
```

Now we are ready to make the workspace. First you need to source the baseline ROS environment:

```
$ source /opt/ros/kinetic/setup.bash
```

Then, build the workspace using:

```
$ catkin_make -C catkin_ws/
```

☀ For more information about `catkin_make`, see Section 30.6 - `catkin_make`.

**Note:** there is a known bug, for which it fails the first time on the Raspberry Pi. Try again; it will work.

+ comment
I got no error on first execution on the Raspberry Pi

## 11.4. Clone the duckiefleet repository

Clone the relevant `duckiefleet` repository into `~/duckiefleet`.

See Subsection 4.1.2 - Duckiefleet directory `DUCKIEFLEET_ROOT` to find the right `duckiefleet` repository.

In `~/.bashrc` set `DUCKIEFLEET_ROOT` to point to the directory:

```
export DUCKIEFLEET_ROOT=~/duckiefleet
```

Also, make sure that you execute `~/.bashrc` in the current shell by running the command:

```
source ~/.bashrc
```

## 11.5. Add your vehicle data to the robot database

Next, you need to add your robot to the vehicles database. This is not optional and required in order to launch any ROS scripts.

You have already a copy of the vehicles database in the folder `robots` of `DUCKIEFLEET_ROOT`.

Copy the file `emma.robot.yaml` to *robotname*`.robot.yaml`, where *robotname* is your robot's hostname. Then edit the copied file to represent your Duckiebot.

➡ For information about the format, see Section 4.2 - The "scuderia" (vehicle database).

Generate the machines file.

➡ The procedure is listed here: Section 4.3 - The `machines` file.

Finally, push your robot configuration to the duckiefleet repo.

## 11.6. Test that the joystick is detected

Plug the joystick receiver in one of the USB port on the Raspberry Pi.

To make sure that the joystick is detected, run:

```
$ ls /dev/input/
```

and check if there is a device called `js0` on the list.

**Check before you continue**

Make sure that your user is in the group `input` and `i2c`:

```
$ groups
username sudo input i2c
```

If `input` and `i2c` are not in the list, you missed a step. Ohi ohi! You are not following the instructions carefully!

➡ Consult again Section 9.11 - Create your user.

To test whether or not the joystick itself is working properly, run:

```
$ jstest /dev/input/js0
```

Move the joysticks and push the buttons. You should see the data displayed change according to your actions.

## 11.7. Run the joystick demo

SSH into the Raspberry Pi and run the following from the `duckietown` directory:

```
$ cd ~/duckietown
$ source environment.sh
```

The `environment.sh` setups the ROS environment at the terminal (so you can use commands like `rosrun` and `roslaunch`).

Now make sure the motor shield is connected.

Run the command:

```
$ roslaunch duckietown joystick.launch veh:=robot name
```

If there is no "red" output in the command line then pushing the left joystick knob controls throttle - right controls steering.

This is the expected result of the commands:

| | |
|---|---|
| left joystick up | forward |
| left joystick down | backward |
| right joystick left | turn left (positive yaw) |
| right joystick right | turn right (negative yaw) |

It is possible you will have to unplug and replug the joystick or just push lots of buttons on your joystick until it wakes up. Also make sure that the mode switch on the top of your joystick is set to "X", not "D".

XXX Is all of the above valid with the new joystick?

Close the program using Ctrl-C.

### 1) Troubleshooting

**Symptom:** The robot moves weirdly (e.g. forward instead of backward).

**Resolution:** The cables are not correctly inserted. Please refer to the assembly guide for pictures of the correct connections. Try swapping cables until you obtain the expected behavior.

**Resolution:** Check that the joystick has the switch set to the position "x".And the mode light should be off.

**Symptom:** The left joystick does not work.

**Resolution:** If the green light on the right to the "mode" button is on, click the "mode" button to turn the light off. The "mode" button toggles between left joystick or the cross on the left.

**Symptom:** The robot does not move at all.

**Resolution:** The cables are disconnected.

**Resolution:** The program assumes that the joystick is at `/dev/input/js0`. In doubt, see Section 11.6 - Test that the joystick is detected.

## 11.8. The proper shutdown procedure for the Raspberry Pi

Generally speaking, you can terminate any `roslaunch` command with Ctrl-C.

To completely shutdown the robot, issue the following command:

```
$ sudo shutdown -h now
```

Then wait 30 seconds.

Warning: If you disconnect the power before shutting down properly using `shutdown`, the system might get corrupted.

Then, disconnect the power cable, at the **battery end**.

As an alternative you can use the `poweroff` command:

```
$ sudo poweroff
```

Warning: If you disconnect frequently the cable at the Raspberry Pi's end, you might damage the port.

# Unit C-12
# Reading from the camera

Knowledge and activity graph

**Requires:** You have configured the Duckiebot. The procedure is documented in Unit C-9 - Duckiebot Initialization.

**Requires:** You know the basics of ROS (launch files, `roslaunch`, topics, `rostopic`).

**TODO:** put reference

**Results:** You know that the camera works under ROS.

## 12.1. Check the camera hardware

It might be useful to do a quick camera hardware check.

➡ The procedure is documented in Section 9.13 - Hardware check: camera.

## 12.2. Create two windows

On the laptop, create two Byobu windows.

➡ A quick reference about Byobu commands is in Unit J-26 - Byobu.

You will use the two windows as follows:

- In the first window, you will launch the nodes that control the camera.
- In the second window, you will launch programs to monitor the data flow.

**Note:** You could also use multiple *terminals* instead of one terminal with multiple Byobu windows. However, using Byobu is the best practice to learn.

## 12.3. First window: launch the camera nodes

In the first window, we will launch the nodes that control the camera. All the following commands should be run in the `~/duckietown` directory:

```
$ cd ~/duckietown
```

Activate ROS:

```
$ source environment.sh
```

Run the launch file called `camera.launch`:

```
$ roslaunch duckietown camera.launch veh:=robot name
```

At this point, you should see the red LED on the camera light up continuously.

In the terminal you should not see any red message, but only happy messages like the following:

```
...
[INFO] [1502539383.948237]: [/robot name/camera_node] Initialized.
[INFO] [1502539383.951123]: [/robot name/camera_node] Start capturing.
[INFO] [1502539384.040615]: [/robot name/camera_node] Published the first image.
```

☀ For more information about `roslaunch` and "launch files", see Section 30.3 - `roslaunch`.

## 12.4. Second window: view published topics

Switch to the second window. All the following commands should be run in the `~/duckietown` directory:

```
$ cd ~/duckietown
```

Activate the ROS environment:

```
$ source environment.sh
```

### 1) List topics

You can see a list of published topics with the command:

```
$ rostopic list
```

☀ For more information about `rostopic`, see Section 30.5 - `rostopic`.

You should see the following topics:

```
/robot name/camera_node/camera_info
/robot name/camera_node/image/compressed
/robot name/camera_node/image/raw
/rosout
/rosout_agg
```

### 2) Show topics frequency

You can use `rostopic hz` to see the statistics about the publishing frequency:

```
$ rostopic hz /robot name/camera_node/image/compressed
```

On a Raspberry Pi 3, you should see a number close to 30 Hz:

```
average rate: 30.016
    min: 0.026s max: 0.045s std dev: 0.00190s window: 841
```

### 3) Show topics data

You can view the messages in real time with the command `rostopic echo`:

```
$ rostopic echo /robot name/camera_node/image/compressed
```

You should see a large sequence of numbers being printed to your terminal.

That's the "image" — as seen by a machine.

If you are Neo, then this already makes sense. If you are not Neo, in Unit C-14 - RC+camera remotely, you will learn how to visualize the image stream on the laptop using `rviz`.

use Ctrl-C to stop `rostopic`.

**TODO:** Physically focus the camera.

# Unit C-13
# RC control launched remotely

Assigned to: Andrea

Knowledge and activity graph

**Requires:** You can run the joystick demo from the Raspberry Pi. The procedure is documented in Unit C-11 - Software setup and RC remote control.

**Results:** You can run the joystick demo from your laptop.

## 13.1. Two ways to launch a program

ROS nodes can be launched in two ways:

1. "local launch": log in to the Raspberry Pi using SSH and run the program from there.
2. "remote launch": run the program directly from a laptop.

Which is better when is a long discussion that will be done later. Here we set up the "remote launch".

TODO: draw diagrams

## 13.2. Download and setup `Software` repository on the laptop

As you did on the Duckiebot, you should clone the `Software` repository in the `~/duckietown` directory.

➡ The procedure is documented in Section 11.1 - Clone the Duckietown repository.

Then, you should build the repository.

➡ This procedure is documented in Section 11.3 - Set up the ROS environment on the Duckiebot.

## 13.3. Rebuild the machines files

In a previous step you have created a robot configuration file and pushed it to the duckiefleet repo. Now you have to pull duckiefleet on the laptop and rebuild the machines configuration file there.

➡ The procedure is documented in Section 11.5 - Add your vehicle data to the robot database.

## 13.4. Start the demo

Now you are ready to launch the joystick demo remotely.

**Check before you continue**

Make sure that you can login with SSH **without a password**. From the laptop, run:

```
$ ssh username@robot name.local
```

If this doesn't work, you missed some previous steps.

Run this *on the laptop*:

```
$ source environment.sh
$ roslaunch duckietown joystick.launch veh:=robot name
```

You should be able to drive the vehicle with joystick just like the last example. Note that remotely launching nodes from your laptop doesn't mean that the nodes are running on your laptop. They are still running on the Raspberry Pi in this case.

☀ For more information about `roslaunch`, see Section 30.3 - `roslaunch`.

## 13.5. Watch the program output using `rqt_console`

Also, you might have notice that the terminal where you launch the launch file is not printing all the printouts like the previous example. This is one of the limitation of remote launch.

Don't worry though, we can still see the printouts using `rqt_console`.

On the laptop, open a new terminal window, and run:

```
$ export ROS_MASTER_URI=http://robot name.local:11311/
$ rqt_console
```

You should see a nice interface listing all the printouts in real time, completed with filters that can help you find that message you are looking for in a sea of messages. If `rqt_console` does not show any message, check out the *Troubleshooting* section below.

You can use Ctrl-C at the terminal where `roslaunch` was executed to stop all the nodes launched by the launch file.

☀ For more information about `rqt_console`, see Section 30.2 - `rqt_console`.

## 13.6. Troubleshooting

**Symptom:** `rqt_console` does not show any message.

**Resolution:** Open `rqt_console`. Go to the Setup window (top-right corner). Change the "Rosout Topic" field from `/rosout_agg` to `/rosout`. Confirm.

**Symptom:** `roslaunch` fails with an error similar to the following:

```
remote[robot name.local-0]: failed to launch on robot name:

Unable to establish ssh connection to [username@robot name.local:22]:
Server u'robot name.local' not found in known_hosts.
```

**Resolution:** You have not followed the instructions that told you to add the `HostKeyAlgorithms` option. Delete `~/.ssh/known_hosts` and fix your configuration.

➡ The procedure is documented in Section 19.3 - Local configuration.

# Unit C-14
# RC+camera remotely

Assigned to: Andrea

Knowledge and activity graph

**Requires:** You can run the joystick demo remotely. The procedure is documented in Unit C-13 - RC control launched remotely.

**Requires:** You can read the camera data from ROS. The procedure is documented in Unit C-12 - Reading from the camera.

**Requires:** You know how to get around in Byobu. You can find the Byobu tutorial in Unit J-26 - Byobu.

**Results:** You can run the joystick demo from your laptop and see the camera image on the laptop.

## 14.1. Assumptions

We are assuming that the joystick demo in Unit C-13 - RC control launched remotely worked.

We are assuming that the procedure in Unit C-12 - Reading from the camera succeeded.

We also assume that you terminated all instances of `roslaunch` with Ctrl-C, so that currently there is nothing running in any window.

## 14.2. Terminal setup

On the laptop, this time create **four** Byobu windows.

➡ A quick reference about Byobu commands is in Unit J-26 - Byobu.

You will use the four windows as follows:

- In the first window, you will run the joystick demo, as before.
- In the second window, you will launch the nodes that control the camera.
- In the third window, you will launch programs to monitor the data flow.
- In the fourth window, you will use `rviz` to see the camera image.

TODO: Add figures

## 14.3. First window: launch the joystick demo

In the first window, launch the joystick remotely using the same procedure in Section 13.4 - Start the demo.

```
$ source environment.sh
$ roslaunch duckietown joystick.launch veh:=robot name
```

You should be able to drive the robot with the joystick at this point.

## 14.4. Second window: launch the camera nodes

In the second window, we will launch the nodes that control the camera.

The launch file is called `camera.launch`:

```
$ source environment.sh
$ roslaunch duckietown camera.launch veh:=robot name
```

You should see the red led on the camera light up.

+ comment

It is recommended to launch the joystick and the camera from onboard the robot after sshing in - LP

## 14.5. Third window: view data flow

Open a third terminal on the laptop.

You can see a list of topics currently on the `ROS_MASTER` with the commands:

```
$ source environment.sh
$ export ROS_MASTER_URI=http://robot name.local:11311/
$ rostopic list
```

You should see the following:

```
/diagnostics
/robot name/camera_node/camera_info
/robot name/camera_node/image/compressed
/robot name/camera_node/image/raw
/robot name/joy
/robot name/wheels_driver_node/wheels_cmd
/rosout
/rosout_agg
```

## 14.6. Fourth window: visualize the image using `rviz`

Launch `rviz` by using these commands:

```
$ source environment.sh
$ source set_ros_master.sh robot name
$ rviz
```

☀ For more information about `rviz`, see Section 30.4 - `rviz`.

In the `rviz` interface, click “Add” on the lower left, then the “By topic” tag, then select

the “Image” topic by the name

```
/robot name/camera_node/image/compressed
```

Then click “ok”. You should be able to see a live stream of the image from the camera.

## 14.7. Proper shutdown procedure

To stop the nodes: You can stop the node by pressing Ctrl-C on the terminal where `roslaunch` was executed. In this case, you can use Ctrl-C in the terminal where you launched the `camera.launch`.

You should see the red light on the camera turn off in a few seconds.

Note that the `joystick.launch` is still up and running, so you can still drive the vehicle with the joystick.

# Unit C-15
# Interlude: Ergonomics

Assigned to: Andrea

So far, we have been spelling out all commands for you, to make sure that you understand what is going on.

Now, we will tell you about some shortcuts that you can use to save some time.

Note: in the future you will have to debug problems, and these problems might be harder to understand if you rely blindly on the shortcuts.

Knowledge and activity graph

Requires: Time: 5 minutes.

Results: You will know about some useful shortcuts.

## 15.1. SSH aliases

Instead of using

```
$ ssh username@robot name.local
```

You can set up SSH so that you can use:

```
$ ssh my-robot
```

To do this, create a host section in `~/.ssh/config` on your laptop with the following contents:

```
Host my-robot
    User username
    Hostname robot name.local
```

Here, you can choose any other string in place of "`my-robot`".

Note that you **cannot** do

```
$ ping my-robot
```

You haven't created another hostname, just an alias for SSH.

However, you can use the alias with all the tools that rely on SSH, including `rsync` and `scp`.

## 15.2. set_ros_master.sh

Instead of using:

```
$ export ROS_MASTER_URI=http://robot name.local:11311/
```

You can use the "`set_ros_master.sh`" script in the repo:

```
$ source set_ros_master.sh robot name
```

Note that you need to use `source`; without that, it will not work.

# Unit C-16
# Wheel calibration

Assigned to: Andrea Daniele

Knowledge and activity graph

Requires: You can run the joystick demo remotely. The procedure is documented in Unit C-13 - RC control launched remotely.

Results: Calibrate the wheels of the Duckiebot such that it goes in a straight line when you command it to. Set the maximum speed of the Duckiebot.

## 16.1. Introduction

The motors used on the Duckiebots are called "Voltage-controlled motors". This means that the velocity of each motor is directly proportional to the voltage it is subject to. Even though we use the same model of motor for left and right wheel, they are not exactly the same. In particular, every motor responds to a given voltage signal in a slightly different way. Similarly, the wheels that we are using look "identical", but they might be slightly different.

If you drive the Duckiebot around using the joystick, you might notice that it doesn't really go in a straight line when you command it to. This is due to those small differences between the motors and the wheels explained above. Different motors can cause the left wheel and right wheel to travel at different speed even though the motors received the same command signal. Similarly, different wheels travel different distances even though the motors made the same rotation.

+ comment

It might be helpful to talk about the ROS Parameter Server here, or at least reference another page. -AD

## 16.2. What is the Calibration step?

We can counter this behavior by *calibrating* the wheels. A calibrated Duckiebot sends two different signals to left and right motor such that the robot moves in a straight line when you command it to.

The relationship between linear and angular velocity of the robot and the velocities of left and right motors are:

$$v_{\text{right}} = (g + r) \cdot (v + \frac{1}{2}\omega l)$$
$$v_{\text{left}} = (g - r) \cdot (v - \frac{1}{2}\omega l)$$

where $v_{\text{right}}$ and $V_{\text{left}}$ are the velocities of the two motors, $g$ is called *gain*, $r$ is called *trim*, $v$ and $\omega$ are the desired linear and the angular velocity of the robot, and $l$ is the distance between the two wheels. The gain parameter $g$ controls the maximum speed of

the robot. With $g > 1.0$, the vehicle goes faster given the same velocity command, and for $g < 1.0$ it goes slower. The trim parameter $r$ controls the balance between the two motors. With $r > 0$, the right wheel will turn slightly more than the left wheel given the same velocity command; with $r < 0$, the left wheel will turn slightly more the right wheel.

+ comment

It might be helpful to add the differential equations that relate velocities and voltages of the motors. -AD

## 16.3. Perform the Calibration

### 1) Calibrating the `trim` parameter

The trim parameter is set to **0.00** by default, under the assumption that both motors and wheels are perfectly identical. You can change the value of the trim parameter by running the command:

```
$ rosservice call /robot name/inverse_kinematics_node/set_trim -- trim value
```

To calibrate the trim parameter use the following steps:

*Step 1:*

Make sure that your Duckiebot is ON and connected to the network.

*Step 2:*

On your Duckiebot, launch the joystick process:

```
$ roslaunch duckietown joystick.launch veh:=robot name
```

*Step 3:*

Use some tape to create a straight line on the floor (Figure 16.1).

Figure 16.1. Straight line useful for wheel calibration

*Step 4:*

Place your Duckiebot on one end of the tape. Make sure that the Duckiebot is perfectly centered with respect to the line.

*Step 5:*

Command your Duckiebot to go straight for about 2 meters. Observe the Duckiebot from the point where it started moving and annotate on which side of the tape the Duckiebot drifted (Figure 16.2).

Figure 16.2. Left/Right drift

*Step 6:*

Measure the distance between the center of the tape and the center of the axle of the Duckiebot after it traveled for about 2 meters (Figure 16.3).

Make sure that the ruler is orthogonal to the tape.

Figure 16.3. Measure the amount of drift after 2 meters run

If the Duckiebot drifted by less than **10** centimeters you can stop calibrating the trim

parameter. A drift of **10** centimeters in a **2** meters run is good enough for Duckietown. If the Duckiebot drifted by more than **10** centimeters, continue with the next step.

*Step 7:*

If the Duckiebot drifted to the left side of the tape, decrease the value of ***r***, by running, for example:

```
$ rosservice call /robot name/inverse_kinematics_node/set_trim -- -0.1
```

In order to run this command you should create another ***ssh*** connection to the duckiebot as the joystick process should be still running.

*Step 8:*

If the Duckiebot drifted to the right side of the tape, increase the value of ***r***, by running, for example:

```
$ rosservice call /robot name/inverse_kinematics_node/set_trim -- 0.1
```

In order to run this command you should create another ***ssh*** connection to the duckiebot as the joystick process should be still running.

*Step 9:*

Repeat the steps 4-8.

### 2) Calibrating the `gain` parameter

The gain parameter is set to **1.00** by default. You can change its value by running the command:

```
$ rosservice call /robot name/inverse_kinematics_node/set_gain -- gain value
```

You won't really know if it's right until you verify it though! onto the next section

### 3) Verify your calibration

Construct a calibration station similar to the one in Figure 16.4:

Figure 16.4. Kinematic calibration verification setup

The following are the specs for this 3x1 mat "runway":

- Red line as close to the edge without crossing the interlocking bits
- Blue/Black line 8 cm from red line and parallel to it.
- White lines on the edge without intersecting the interlocking bits
- Yellow line in the middle of the white lines
- Blue/black start position is ~3-4 cm from the edge (not including the interlocking bits)

Place your robot as shown in Figure 16.4.

On your robot execute:

```
$ cd Duckietown root
$ make hw-test-kinematics
```

You should see your robot drive down the lane. If it is calibrated properly, you will see a message saying that it has `PASSED`, otherwise it is `FAILED` and you should adjust your gains based on what you observe and try again.

### 4) Store the calibration

When you are all done, save the parameters by running:

```
$ rosservice call /robot name/inverse_kinematics_node/save_calibration
```

The first time you save the parameters, this command will create the file

```
DUCKIEFLEET_ROOT/calibrations/kinematics/robot name.yaml
```

You can add and commit it to the repository. Then you should create a pull request in the duckiefleet repo

# Unit C-17
# Camera calibration

Knowledge and activity graph

**Requires:** You can see the camera image on the laptop. The procedure is documented in Unit C-14 - RC+camera remotely.

**Requires:** You have all the repositories (described in Unit M-7 - Git usage guide for Fall 2017) cloned properly and you have your environment variables set properly.

**Results:** Calibration for the robot camera.

## 17.1. Intrinsic calibration

### 1) Setup

Download and print a PDF of the calibration checkerboard (A4 intrinsic, A3 extrinsic, US Letter). Fix the checkerboard to a planar surface.

Figure 17.1

**Note:** the squares must have side equal to 0.031 m = 3.1 cm.

### 2) Calibration

Make sure your Duckiebot is on, and both your laptop and Duckiebot are connected to the duckietown network.

*Step 1:*

Open two terminals on the laptop.

*Step 2:*

In the first terminal, log in into your robot using SSH and launch the camera process:

```
$ cd duckietown root
$ source environment.sh
$ roslaunch duckietown camera.launch veh:=robot name raw:=true
```

*Step 3:*

In the second laptop terminal run the camera calibration:

```
$ cd duckietown root
$ source environment.sh
$ source set_ros_master.sh robot name
$ roslaunch duckietown intrinsic_calibration.launch veh:=robot name
```

You should see a display screen open on the laptop (Figure 17.2).

Figure 17.2

Position the checkerboard in front of the camera until you see colored lines overlaying the checkerboard. You will only see the colored lines if the entire checkerboard is within the field of view of the camera.

You should also see colored bars in the sidebar of the display window. These bars indicate the current range of the checkerboard in the camera's field of view:

- X bar: the observed horizontal range (left - right)
- Y bar: the observed vertical range (top - bottom)
- Size bar: the observed range in the checkerboard size (forward - backward from the camera direction)
- Skew bar: the relative tilt between the checkerboard and the camera direction

Also, make sure to focus the image by rotating the mechanical focus ring on the lens of the camera.

+ comment

Do not change the focus during or after the calibration, otherwise your calibration is no longer valid. I'd also suggest to not to use the lens cover anymore; removing the lens cover changes the focus. -MK

Now move the checkerboard right/left, up/down, and tilt the checkerboard through various angles of relative to the image plane. After each movement, make sure to

pause long enough for the checkerboard to become highlighted. Once you have collected enough data, all four indicator bars will turn green. Press the "CALIBRATE" button in the sidebar.

Calibration may take a few moments. Note that the screen may dim. Don't worry, the calibration is working.

Figure 17.3

### 3) Save the calibration results

If you are satisfied with the calibration, you can save the results by pressing the "COMMIT" button in the side bar. (You never need to click the "SAVE" button.)

Figure 17.4

This will automatically save the calibration results on your Duckiebot:

```
duckiefleet root/calibrations/camera_intrinsic/robot name.yaml
```

*Step 4:*

Now let's push the `robot name.yaml` file to the git repository. You can stop the `camera.launch` terminal with Ctrl-C or open a new terminal in Byobu with F2.

Update your local git repository:

```
$ cd duckiefleet root
$ git pull
$ git status
```

You should see that your new calibration file is uncommitted. You need to commit the file to your branch.

```
$ git checkout -b Github username-devel
$ git add calibrations/camera_intrinsic/robot name.yaml
$ git commit -m "add robot name intrinsic calibration file"
$ git push origin Github username-devel
```

Before moving on to the extrinsic calibration, make sure to kill all running processes by pressing Ctrl-C in each of the terminal windows.

## 17.2. Extrinsic calibration

### 1) Setup

Arrange the Duckiebot and checkerboard according to Figure 17.5. Note that the axis of the wheels should be aligned with the y-axis (Figure 17.5).

Figure 17.5

Figure 17.6 shows a view of the calibration checkerboard from the Duckiebot. To ensure proper calibration there should be no clutter in the background and two A4 papers should be aligned next to each other.

Figure 17.6

### 2) Calibration procedure

*Step 1:*

Log in into your robot using SSH and launch the camera:

```
$ cd duckietown root
$ source environment.sh
$ roslaunch duckietown camera.launch veh:=robot name raw:=true
```

*Step 2:*

Run the `ground_projection_node.py` node on your laptop:

```
$ cd duckietown root
$ source environment.sh
$ source set_ros_master.sh robot name
$ roslaunch ground_projection ground_projection.launch  veh:=robot name local:=true
```

*Step 3:*

Check that everything is working properly. In a new terminal (with environment sourced and ros_master set to point to your robot as above)

```
$ rostopic list
```

You should see new ros topics:

```
/robot name/camera_node/camera_info
/robot name/camera_node/framerate_high_switch
/robot name/camera_node/image/compressed
/robot name/camera_node/image/raw
/robot name/camera_node/raw_camera_info
```

The ground_projection node has two services. They are not used during operation. They just provide a command line interface to trigger the extrinsic calibration (and for debugging).

```
$ rosservice list
```

You should see something like this:

```
...
/robot name/ground_projection/estimate_homography
/robot name/ground_projection/get_ground_coordinate
...
```

If you want to check whether your camera output is similar to the one at the Figure 17.6 you can start `rqt_image_view`:

```
$ rosrun rqt_image_view rqt_image_view
```

In the `rqt_image_view` interface, click on the drop-down list and choose the image topic:

`/robot name/camera_node/image/compressed`

*Step 4:*
Now you can estimate the homography by executing the following command (in a new terminal):

```
$ rosservice call /robot name/ground_projection/estimate_homography
```

This will do the extrinsic calibration and automatically save the file to your laptop:

`duckiefleet root/calibrations/camera_extrinsic/robot name.yaml`

As before, add this file to your local Git repository on your laptop, push the changes to your branch and do a pull request to master. Finally, you will want to update the local repository on your Duckiebot.

# Unit C-18
# Updated camera calibration and validation

Here is an updated, more practical extrinsic calibration and validation procedure.

## 18.1. Check out the experimental branch

Check out the branch `andrea-better-camera-calib`.

## 18.2. Place the robot on the pattern

Arrange the Duckiebot and checkerboard according to Figure 18.1. Note that the axis of the wheels should be aligned with the y-axis (Figure 18.1).

Figure 18.1

Figure 18.2 shows a view of the calibration checkerboard from the Duckiebot. To ensure proper calibration there should be no clutter in the background and two A4 papers should be aligned next to each other.

Figure 18.2

## 18.3. Extrinsic calibration procedure

Run the following on the Duckiebot:

```
$ rosrun complete_image_pipeline calibrate_extrinsics
```

That's it!

No laptop is required.

You can also look at the output files produced, to make sure it looks reasonable. It should look like Figure 18.3.

Figure 18.3

Note the difference between the two types of rectification:

1. In `bgr_rectified` the rectified frame coordinates are chosen so that the frame is filled entirely. Note the image is stretched - the April tags are not square. This is the rectification used in the lane localization pipeline. It doesn't matter that the image is stretched, because the homography learned will account for that deformation.
2. In `rectified_full_ratio_auto` the image is not stretched. The camera matrix is preserved. This means that the aspect ratio is the same. In particular note the April tags are square. If you do something with April tags, you need this rectification.

## 18.4. Camera validation by simulation

You can run the following command to make sure that the camera calibration is reasonable:

```
$ rosrun complete_image_pipeline validate_calibration
```

What this does is simulating what the robot should see, if the models were correct (Figure 18.4).

Figure 18.4. Result of `validate_calibration`.

Then it also tries to localize on the simulated data (Figure 18.5). It usual achieves impressive calibration results!

Simulations are doomed to succeed.

Figure 18.5. Output of `validate_calibration`: localization in simulated environment.

## 18.5. Camera validation by running one-shot localization

Place the robot in a lane.

Run the following command:

```
$ rosrun complete_image_pipeline single_image_pipeline
```

What this does is taking one snapshot and performing localization on that single image. The output will be useful to check that everything is ok.

### 1) Example of correct results

Figure 18.6 is an example in which the calibration was correct, and the robot localizes perfectly.

Figure 18.6. Output when camera is properly calibrated.

### 2) Example of failure

This is an example in which the calibration is incorrect.

Look at the output in the bottom left: clearly the perspective is distorted, and there is no way for the robot to localize given the perspective points.

Figure 18.7. Output when camera not properly calibrated.

## 18.6. The importance of validation

Validation is useful because otherwise it is hard to detect wrong calibrations.

For example, in 2017, a bug in the calibration made about 5 percent of the calibrations useless (Figure 18.8), and people didn't notice for weeks (!).

Figure 18.8. In 2017, a bug in the calibration made about 5 percent of the calibrations useless.

# Unit C-19
# Taking and verifying a log

Knowledge and activity graph

**Requires:** Unit C-12 - Reading from the camera
**Requires:** Unit C-11 - Software setup and RC remote control
**Results:** A verified log.

## 19.1. Preparation

**Note:** it is recommended that you log to your USB and not to your SD card.

➡ To mount your USB see Unit J-15 - Mounting USB drives.

## 19.2. Run something on the Duckiebot

For example, if you want to drive the robot around and collect image data you could run:

```
$ make demo-joystick-camera
```

But anything could do.

## 19.3. View images on the laptop

Run on the laptop:

```
$ cd Duckietown root
$ source environment.sh
$ source set_ros_master.sh robot name
$ rqt_image_view
```

and verify that indeed your camera is streaming imagery.

## 19.4. Record the log

### 1) Option: Full Logging

To log everything that is being published, on the Duckiebot in a new terminal (See Unit J-26 - Byobu):

```
$ make log-full
```

where here we are assuming that you are logging to the USB and have followed Unit J-15 - Mounting USB drives.

#### 2) Option: Log Minimal

To log only the imagery, camera_info, the control commands and a few other essential things, on the Duckiebot in a new terminal (See Unit J-26 - Byobu):

```
$ make log-minimal
```

where here we are assuming that you are logging to the USB and have followed Unit J-15 - Mounting USB drives.

## 19.5. Verify a log

On the Duckiebot run:

```
$ rosbag info FULL_PATH_TO_BAG --freq
```

Then:

- verify that the "duration" of the log seems "reasonable" - it's about as long as you ran the log command for
- verify that the "size" of the log seems "reasonable" - the log size should grow at about 220MB/min
- verify in the output that your camera was publishing very close to **30.0Hz** and verify that you joysick was publishing at a rate between **3Hz** and **6Hz**.

**TODO:** More complex log verification methods.

# Unit C-20
# Troubleshooting

Knowledge and activity graph

**Requires:** The Raspberry Pi of the Duckiebot is connected to the battery.
**Requires:** The Stepper Motor HAT is connected to the battery.
**Requires:** You have a problem!

## 20.1. The Raspberry Pi does not turn ON

**Symptom:** The red LED on the Raspberry Pi is OFF

**Resolution:** Press the button on the side of the battery (Figure 20.1).

Figure 20.1. The power button on the RAVPower Battery.

## 20.2. I cannot access my Duckiebot via SSH

**Symptom:** When I run `ssh robot_name.local` I get the error `ssh: Could not resolve hostname robot_name.local`.

**Resolution:** Make sure that your Duckiebot is ON. Connect it to a monitor, a mouse and a keyboard. Run the command

```
$ sudo service avahi-daemon status
```

You should get something like the following

```
● avahi-daemon.service - Avahi mDNS/DNS-SD Stack
   Loaded: loaded (/lib/systemd/system/avahi-daemon.service; enabled; vendor preset:
enabled)
   Active: active (running) since Sun 2017-10-22 00:07:53 CEST; 1 day 3h ago
 Main PID: 699 (avahi-daemon)
   Status: "avahi-daemon 0.6.32-rc starting up."
   CGroup: /system.slice/avahi-daemon.service
           ├─699 avahi-daemon: running [robot_name_in_avahi.local
           └─727 avahi-daemon: chroot helpe
```

Avahi is the module that in Ubuntu implements the mDNS responder. The mDNS responder is responsible for advertising the hostname of the Duckiebot on the network so that everybody else within the same network can run the command `ping robot_name.local` and reach your Duckiebot. Focus on the line containing the

hostname published by the `avahi-daemon` on the network (i.e., the line that contains *robot_name_in_avahi*`.local`). If *robot_name_in_avahi* matches the *robot_name*, go to the next Resolution point. If *robot_name_in_avahi* has the form *robot_name*-`XX`, where `XX` can be any number, modify the file `/etc/avahi/avahi-daemon.conf` as shown below.

Identify the line

```
use-ipv6=yes
```

and change it to

```
use-ipv6=no
```

Identify the line

```
#publish-aaaa-on-ipv4=yes
```

and change it to

```
publish-aaaa-on-ipv4=no
```

Restart Avahi by running the command

```
$ sudo service avahi-daemon restart
```

## 20.3. The Duckiebot does not move

**Symptom:** I can SSH into my Duckiebot and run the joystick demo but the joystick does not move the wheels.

**Resolution:** Press the button on the side of the battery (Figure 20.1).

**Resolution:** Check that the red indicator on the joystick stopped blinking.

(a) Bad joystick status (b) Bad joystick status

Figure 20.2

**Symptom:** The joystick is connected (as shown in Figure 20.2b - Bad joystick status) but the Duckiebot still does not move.

**Resolution:** Make sure that the controller is connected to the Duckiebot and that the OS receives the data from it. Run

```
$ jstest /dev/input/js0
```

If you receive the error

```
jstest: No such file or directory
```

it means that the USB receiver is not connected to the Raspberry Pi or is broken. If the command above shows something like the following

```
Driver version is 2.1.0.
Joystick (ShanWan PC/PS3/Android) has 8 axes (X, Y, Z, Rz, Gas, Brake, Hat0X, Hat0Y)
and 15 buttons (BtnX, BtnY, BtnZ, BtnTL, BtnTR, BtnTL2, BtnTR2, BtnSelect, BtnStart,
BtnMode, BtnThumbL, BtnThumbR, ?, ?, ?).
Testing ... (interrupt to exit)
Axes:  0:     0  1:     0  2:     0  3:     0  4:-32767  5:-32767  6:     0  7:     0
Buttons:  0:off  1:off  2:off  3:off  4:off  5:off  6:off  7:off  8:off  9:off 10:off
11:off 12:off 13:off 14:off
```

it means that the USB receiver is connected to the Raspberry Pi. Leave the terminal above open and use the joystick to command the Duckiebot. If you observe that the numbers shown in the terminal change according to the commands sent through the joystick than the problem is in ROS. Make sure that the joystick demo is launched. Restart the Duckiebot if needed and try again.

If the numbers do not change while using the joystick then follow this guide at the next Resolution point.

**Resolution:** The controller might be connected to another Duckiebot nearby. Turn off the controller, go to a room with no other Duckiebots around and turn the controller back on. Retry.

# Part D

# Operation manual - Duckietown

# Unit D-1
# Duckietown parts

Duckietowns are the cities where Duckiebots drive. Here, we provide a link to all bits and pieces that are needed to build a Duckietown, along with their price tag. Note that while the topography of the map is highly customable, we recommend using the components listed below. Before purchasing components for a Duckietown, read Unit D-2 - Duckietown Appearance Specification to understand how Duckietowns are built.

In general, keep in mind that:

- The links might expire, or the prices might vary.
- Shipping times and fees vary, and are not included in the prices shown below.
- Substitutions are probably not OK, unless you are OK in writing some software.

Knowledge and activity graph

**Requires:** Cost (per $m^2$): USD ??? + Shipping Fees

**Requires:** Time: ??? days (average shipping time)

**Results:** A kit of parts ready to be assembled in a Duckietown.

**Next:** Assembling a Duckietown.

**TODO:** Figure out costs

## 1.1. Bill of materials

Table 1.1. Bill of materials for Duckietown

| | |
|---|---|
| Duckies | USD 17/100 pieces |
| Floor Mats | USD 37.5/6 pieces (24 sqft) |
| Duct tape - Red | USD 8.50/roll |
| Duct tape - White | USD 8.50/roll |
| Duct tape - Yellow | USD 8/roll |
| Traffic signs | USD 18.50/13 pieces |
| Total for **Duckietown**/$m^2$ | USD ?? |

**TODO:** Add suggestions for “small”, “medium”, “big” towns as a function of $m^2$ and supported bots

## 1.2. Duckies

Duckies (Figure 1.1) are essential yet non functional.

Figure 1.1. The Duckies

## 1.3. Floor Mats

The floor mats (Figure 1.2) are the ground on which the Duckiebots drive.

We choose these mats because they have desirable surface properties, are modular, and have the right size to be street segments. Each square is (~61x61cm) and can connect on every side of other squares. There are 6 mats in each package.

Figure 1.2. The Floor Mats

Each mat can be a segment of road: straight, a curve, or an intersection (3, or 4 way). To design your Duckietown, see Unit D-2 - Duckietown Appearance Specification.

## 1.4. Duck Tape

We use duck (duct) tape of different colors (Figure 1.3) for defining the roads and their signals. White indicates the road boundaries, yellow determines lane boundaries and red are stop signs.

The white and red tape we use are 2 inches wide, while the yellow one is 1 inch wide.

Figure 1.3. The Duck Tapes

To verify how much tape you need for each road segment type, see Unit D-2 - Duckietown Appearance Specification.

## 1.5. Traffic Signs

Traffic signs (Figure 1.4) inform Duckiebots on the map of Duckietown, allowing them to make driving decisions.

Figure 1.4. The Signs

Depending on the chosen road topograhy, the number of necessary road signal will vary. To design your Duckietown, see Unit D-2 - Duckietown Appearance Specification. You will also need to print out and the signs that you will need as described in Unit D-3 - Signage.

## 1.6. Traffic lights Parts

Traffic lights regulate intersections in Duckietown. Here, we provide a link to all bits and pieces that are needed to build a traffic light, along with their price tag. You will need one traffic per either three, or four way intersections. The components listed below meet the appearence specifications described in Unit D-2 - Duckietown Appearance Specification.

In general, keep in mind that:

- The links might expire, or the prices might vary.
- Shipping times and fees vary, and are not included in the prices shown below.
- Substitutions are probably OK, if you are willing to write some software.

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Cost: USD ??? + Shipping Fees

**Requires:** Time: ??? days (average shipping time)

**Results:** A kit of parts ready to be assembled in a traffic light.

**Next:** Assembling a traffic light.

**TODO:** Estimate time and costs

### 1) Bill of materials

TABLE 1.2. BILL OF MATERIALS FOR TRAFFIC LIGHT

| | |
|---|---|
| Raspberry Pi | USD ?? |
| 4 LEDs | USD ?? |
| Wires | USD ?? |
| Total for Traffic Light | USD ?? |

**TODO:** Complete table

### 2) Raspberry Pi

(Figure 1.5) are essential yet non functional.

Figure 1.5. The placeholder

# Unit D-2
# Duckietown Appearance Specification

This document describes the Duckietown specification. These are a set of rules for which a functional system has been verified.

Any Duckietown not adhering to the rules described here cannot call itself a "Duckietown", since it is not one.

Additionally, any Duckietown not adhering to these rules may cause the Duckiebots to fail in unexpected ways.

## 2.1. Version history

Note here the changes to the specification, so that we are able to keep in sync the different Duckietowns.

- Version 1.0 - used for MIT 2.166
- Version 2.0 - user for Fall 2017

## 2.2. Overview

Duckietown is built with two layers:

1. The first is the *floor layer*. The floor is built of interconnected exercise mats with tape on them.
2. The second layer is the *signals layer* and contains all the signs and other objects that sit on top of the mats.

> Note: the visual appearance of the area where the Duckietown is created is variable. If you discover that this appearance is causing negative performance, a "wall" of blank tiles constructed vertically can be used to reduce visual clutter.

## 2.3. Layer 1 - The Tile Layer

Each tile is a 2 ft x 2 ft square and is able to interlock with the others.

There are six types of tiles, as shown in Figure 2.1. Currently, the left turn and right turn tiles are symmetric: one is the 90 degree rotation of the other.

(a) DT17_tile_straight (b) DT17_tile_curve_left (c) DT17_tile_curve_right

(d) DT17_tile_three_way_center (e) DT17_tile_four_way_center (f) DT17_tile_empty

Figure 2.1. The principal tile types in Duckietown

Figure 2.2. A 3 by 3 loop (DT17_map_loop3)

Figure 2.3. Four way intersection usage

Figure 2.4. Three way intersection usage

## 1) Tapes

There are 3 colors of tapes: white, yellow, and red.

*White tape:*

**Proposition 3.** A Duckiebot never collides with Duckietown if it never crosses or touches a white tape strip.

Here are some facts about the white tapes:

- White tapes must be solid (not dashed)
- The width of the white tape is 2 inches (5.08 cm).
- The white tape is always placed on the right hand side of a lane. We assume that the Duckiebots drive on the right hand side of the road.

+ comment
this should be part of the “traffic rules” sections.

- For curved roads, the white lane marker is formed by five pieces of white tape, while the inner corner is formed by three pieces, placed according to the specifications in the image below, where the edge pieces are matched to adjacent straight or curved tiles (Figure 2.5).

Figure 2.5. The specification for a curved road tile

*Yellow tape:*

On a two-way road, the yellow tape should be dashed. Each piece should have a length of approximately **2 in** with a **1 in** gap separating each piece.

Yellow tapes on curves: see curved road image in white tape section, pieces at tile edges should be in center of lane, piece at the middle of the curve should be approximately 20.5 cm from middle of inner center white piece of tape, with approximated circular arc in between.

*Red tape:*

Red tapes MAY **only** appear on **intersection** tiles.

The red tape must be the full width of the duck tape roll and should cross the entire lane perpendicular to the lane.

The placement of red tape should always be **under** yellow and white tape.

A Duckiebot navigates Duckietown by a sequence of:

- Navigating one or more straight ties until a red tape appears,
- Wait for the coordination signal,
- Execute an intersection traversal,
- Relocalize in a StraightTile.

The guarantee is:

**Proposition .** If the Duckiebot stops before or ON the red strip, no collisions are possible.

### 2) Topological Constraints During Map Construction

Here are some topological rule constraints that must be met:

1. An intersection can NOT be adjacent to a curved road tile or another intersection tile.
2. Any two adjacent non-empty tiles must have a feasible path from one to the other **of length two**: if they are adjacent, they must be connected.

Some examples of **non-conforming** topologies are shown in Figure 2.6.

(a) Topology violates rule 2 since the bottom two curved tiles are adjacent but not connected

(b) Topology violates rule 1 since curved tiles are adjacent to intersection tiles

(c) Topology violates rule 2 since left-most tiles are adjacent but not connected

Figure 2.6. Some non-conforming Duckietown map topologies

### 3) Parking Lots

**Note:** An experimental new development.

A parking lot is a place for Duckiebots to go when they are tired and need a rest.

A parking lot introduces three additional tile types:

1. **Parking lot entry tile**: This is similar to a straight tile except with a red stop in the middle. The parking lot sign (Figure 2.7o - parking) will be visible from this stop line.
2. **Parking spot tiles**:
3. **Parking spot access tiles**:

**TODO:** the tape on the spot and spot access tiles is currently not yet specified.

The following are the rules for a conforming parking lot:

1. One "parking spot" has size one tile.
2. From each parking spot, there is a path to go to the parking lot entry tile that does not intersect any other parking spot. (i.e. when a Duckiebot is parked, nobody will disturb it).
3. From any position in any parking spot, a Duckiebot can see at least two orthogonal lines or an sign with an April tag.

TODO: this point needs further specification

#### 4) Launch Tiles

**Note:** Experimental

A "launch tile" is used to introduce a new Duckiebot into Duckietown in a controllable way. The launch file should be placed adjacent to a turn tile so that a Duckiebot may "merge" into Duckietown once the initialization procedure is complete.

TODO: Specification for tape on the launch tile

A "yield" sign should be visible from the launch tile.

## 2.4. Layer 2 - Signage and Lights

**IMPORTANT:** All signage should sit with base on the floor and stem coming through the connection between the tiles. Generally, it is advisable to adhere the sign to the floor with double-sided tape. **Under no circumstances should the white (any other tape) be obscured.**

## 2.5. Traffic Signs

**Requires:** To print and assemble the signs refer to Unit D-3 - Signage.

#### 1) Specs

Center of signs are 13 cm height with apriltags of 6.5 cm sq. and a white border pasted below them.

#### 2) Type

The allowable traffic signs are as in Figure 2.7.

(a) stop (b) yield (c) no-right-turn (d) no-left-turn

(e) do-not-enter (f) oneway-right (g) oneway-left (h) 4-way-intersect

(i) right-T-intersect (j) left-T-intersect (k) T-intersection (l) pedestrian

(m) t-light-ahead (n) duck-crossing (o) parking

Figure 2.7. Duckietown Traffic Signs

### 3) Placement

Signs may appear on the opposite side and at the corner of the adjacent tile from which they are viewed. In the absence of any signs, it is assumed that all network flows are allowed so a sign MUST be placed and visible whenever this is not the case.

Signs must only be placed on empty tiles, or next to one of the other tile types if on the border of a map. The sign placements for four different cases are shown in Figure 2.8. At intersections, from each stop line 2 signs should be clearly visible: 1) the intersection type (traffic light or stop sign) and 2) the intersection topology.

At present, 4-way intersections must be equipped with traffic lights for safe navigation.

(a) 4-way intersection (b) 3-way intersection

(c) straight road (d) curved road

Figure 2.8. Placement of Traffic Signs

On straight and curved roads, additional signs can be added as desired. Their placement is indicated in Figure 2.8c - straight road and Figure 2.8d - curved road. The signs should be placed at the border between two tiles and should face towards oncoming traffic as indicated.

In these figures the arrow is the direction of the sign.

## 2.6. Street Name Signs

### 1) Specs

- Font: arial.
- Color: Perhaps we could start with real-world settings: white as foreground and green as background.
- Border: currently no additional borders
- The rounded corners are modified into 90 degrees.
- Height: sign board height is 1.5 in. (**2.1 in**),
- Width: Currently 4.5 in for id 500-511. (**6.1 in +1.1 in "ST" or 5.5 in + 1.7 in "AVE"**)
- Alphabet = English upper case. Different writing systems may need different algorithms.
- Text direction: Horizontal for alphabetical languages

### 2) Placement

- **Similar to traffic light**: The street name should sit on a pole that is based at the corner of the tile outside of the allowable driving region. The bottom of the street name should be at a height of 7in, and allow a duckiebot to pass through. The street names should be visible from both sides of the road.

Every segment of road must have at least one road name sign.

Every turn tile should have a road name sign.

The placement of the road name signs is as indicated in Figure 2.9.

(a) Turn (b) Straight

Figure 2.9. Placement of Road Name Signs

Street name signs should never be perpendicular to the road - they are too big and obtrusive.

## 2.7. Traffic Lights

**Requires:** The assembly procedure for building the a traffic light is found in Unit D-5 - Traffic lights Assembly

### 1) Specs

**To write:** towrite

### 2) Placement

The lights must be at a height of exactly 20 cm above the center of the intersection tile.

The Raspberry Pi should sit on a pole that is based at the corner of the tile outside of the allowable driving region.

# Unit D-3
# Signage

Assigned to: Liam

Knowledge and activity graph

**Requires:** The raw materials as described in Unit D-1 - Duckietown parts

**Results:** A set of signs to be used for assembling your Duckietown.

## 3.1. Build a map

Before beginning with sign assembly you should design a map that adheres to the specification.

An example that was used for the 2017 version of the class is here: Unit M-30 - The Map Description

The full set of currently existing signs is available here: pdf docx

The set of tags used for the 2017 map are available here: pdf docx

## 3.2. Making New Signage

If you find that what is available in the database in insufficient for your needs, then you will need to add to the existing database.

To do so you will have to load the original AprilTags file available here: pdf ps

Which tag you should use depends on what type of sign you are trying add. The ranges of tags are specified in Table 3.1.

Table 3.1. April tag ID ranges

| Purpose | Size | Family | ID Range |
|---|---|---|---|
| Traffic signs | 6.5cm x 6.5cm | 36h11 | 1-199 |
| Traffic lights | 6.5cm x 6.5cm | 36h11 | 200-299 |
| Localization | 6.5cm x 6.5cm | 36h11 | 300-399 |
| Street Name Signs | 6.5cm x 6.5cm | 36h11 | 400-587 |

First, find the last sign of the type that you are trying to make in the signs and tags doc. You will use the next available ID after this one.

Construct the new sign by first copying and pasting an existing sign of similar type, and then replacing/adding the new AprilTag. To add the new april tag, use a screen capture mathod to crop precisely around the tag at the top and sides and include the sign id at the bottom. Then paste the tag into your word file under your desired and resize it exactly 6.5cm (2.56inches).

If you make a new road name sign, you may need to change the font size of the name so that it appears on one line (this is why we like people with names like “ROY” and “RUS”).

Important: You must also add your new sign to the April Tags DB in the software re-

po.

Add a new block like the ones that already exists or modify the one with the appropriate tag id:

```
- tag_id: NEW_TAG_ID
  tag_type: in {TrafficSign, Light, Localization, StreetName}
  street_name: either NEW_STREET_NAME or blank
  vehicle_name: currently not used
  traffic_sign_type: either TRAFFIC_SIGN_TYPE or blank
```

The value of *NEW_STREET_NAME* is up to you to decide (have fun with it!). The value of *TRAFFIC_SIGN_TYPE* should be one of the signs in Figure 2.7

When finished, regenerate the PDF version of the Word file, and commit everything to the repo (via a pull request of course).

> **Note:** It is also possible of course to start you own completely different signs and tags database, but make sure that you specify in the april_tags code which database to load from.

**TODO:** Update the way that the april tags code loads the database

## 3.3. Assembly

To assemble the signs, you should print out the pdf version of the signs and tags file on the thickest card stock available. Cut the signs out with a straight edge and a very sharp knife, leaving a small border of white around the sign. Then use double-sided tape or some other adhesive to affix the the paper sign to the wooden base.

## 3.4. Placement

For placement of signs see Subsection 2.5.3 - Placement.

# Unit D-4
# Duckietown Assembly

> **Assigned to:** Shiying

Follow the rules in the Unit D-2 - Duckietown Appearance Specification.

+ comment

do we need this page at all?

UNIT D-5

# Traffic lights Assembly

**Assigned to:** Marco Erni

# Unit D-6
# Semantics of LEDS

Assigned to: ???

headlights: white, constant

Assumption:

- **20 fps** to do LED detection
- 1s to decide
- 3 frequencies to detect

tail lights: red, **6 hz square wave**

traffic light “GO” = green, 1 hz **square wave**

traffic light “STOP” = red, 1.5 Hz **square wave**

duckie light on top, state 0 = off

duckie light on top, state 1 = blue, **3 Hz, square wave**

duckie light on top, state 2 = ?, **2.5 Hz square wave**

duckie light on top, state 3 = ?, **2 Hz square wave**

# Part E

# Duckiebot - `DB17-1c` configurations

This section contains the acquisition, assembly and setup instructions for the `DB17-1c` configurations. These instructions are separate from the rest as they approximately match the `b` releases of the Fall 2017 Duckietown Engineering Co. branches.

To understand how configurations and releases are defined, refer to: Unit C-1 - Duckiebot configurations.

# Unit E-1
# Acquiring the parts (DB17-lc)

Upgrading your DB17 (DB17-wjd) configuration to DB17-lc (DB17-wjdlc) starts here, with purchasing the necessary components. We provide a link to all bits and pieces that are needed to build a DB17-lc Duckiebot, along with their price tag. If you are wondering what is the difference between different Duckiebot configurations, read Unit C-1 - Duckiebot configurations.

In general, keep in mind that:

- The links might expire, or the prices might vary.
- Shipping times and fees vary, and are not included in the prices shown below.
- Buying the parts for more than one Duckiebot makes each one cheaper than buying only one.
- A few components in this configuration are custom designed, and might be trickier to obtain.

Knowledge and activity graph

**Requires:** A Duckiebot in DB17-wjd configuration.

**Requires:** Cost: USD 77 + Bumpers manufacturing solution

**Requires:** Time: 21 Days (LED board manufacturing and shipping time)

**Results:** A kit of parts ready to be assembled in a DB17-lc configuration Duckiebot.

**Next:** After receiving these components, you are ready to do some soldering before assembling your DB17-lc Duckiebot.

## 1.1. Bill of materials

TABLE 1.1. BILL OF MATERIALS

| | |
|---|---|
| LEDs (DB17-l) | USD 10 |
| LED HAT (DB17-l) | USD 28.20 for 3 pieces |
| Power Cable (DB17-l) | USD 7.80 |
| 20 Female-Female Jumper Wires (300mm) (DB17-l) | USD 8 |
| Male-Male Jumper Wire (150mm) (DB17-l) | USD 1.95 |
| PWM/Servo HAT (DB17-l) | USD 17.50 |
| Bumpers | TBD (custom made) |
| 40 pin female header (DB17-l) | USD 1.50 |
| 5 4 pin female header (DB17-l) | USD 0.60/piece |
| 2 16 pin male header (DB17-l) | USD 0.61/piece |
| 12 pin male header (DB17-l) | USD 0.48/piece |
| 3 pin male header (DB17-l) | USD 0.10/piece |
| 2 pin female shunt jumper (DB17-l) | USD 2/piece |
| 5 200 Ohm resistors (DB17-l) | USD 0.10/piece |
| 10 130 Ohm resistors (DB17-l) | USD 0.10/piece |
| Caster (DB17-c) | USD 6.55/4 pieces |
| 4 Standoffs (M3.5 12mm F-F) (DB17-c) | USD 0.63/piece |
| 8 Screws (M3.5x8mm) (DB17-c) | USD 4.58/100 pieces |
| 8 Split washer lock (DB17-c) | USD 1.59/100 pieces |
| Total for DB17-wjd configuration | USD 212 |
| Total for DB17-lc components | USD 77 + Bumpers |
| Total for DB17-wjdlc configuration | USD 299+Bumpers |

## 1.2. LEDs

The Duckiebot is equipped with 5 RGB LEDs (Figure 1.1). LEDs can be used to signal to other Duckiebots, or just make *fancy* patterns.

The pack of LEDs linked in the table above holds 10 LEDs, enough for two Duckiebots.

Figure 1.1. The RGB LEDs

### 1) LED HAT

The LED HAT (Figure 1.2) provides an interface for our RGB LEDs and the computational stack. This board is a daughterboard for the Adafruit 16-Channel PWM/Servo HAT, and enables connection with additional gadgets such as ADS1015 12 Bit 4 Channel ADC, Monochrome 128x32 I2C OLED graphic display, and Adafruit 9-DOF IMU Breakout - L3GD20H+LSM303. This item will require soldering.

This board is custom degined and can only be ordered in minimum runs of 3 pieces.

The price scales down quickly with quantity, and lead times may be significant, so it is better to buy these boards in bulk.

Figure 1.2. The LED HAT

## 2) PWM/Servo HAT

The PWM/Servo HAT (Figure 1.3) mates to the LED HAT and provides the signals to control the LEDs, without taking computational resources away from the Rasperry Pi itself. This item will require soldering.

Figure 1.3. The PWM-Servo HAT

## 3) Power Cable

To power the PWM/Servo HAT from the battery, we use a short (30cm) angled male USB-A to 5.5/2.1mm DC power jack cable (Figure 1.4).

Figure 1.4. The 30cm angled USB to 5.5/2.1mm power jack cable.

## 4) Male-Male Jumper Wires

The Duckiebot needs one male-male jumper wire (Figure 1.5) to power the DC Stepper Motor HAT from the PWM/Servo HAT.

Figure 1.5. Premier Male-Male Jumper Wires

### 5) Female-Female Jumper Wires

20 Female-Female Jumper Wires (Figure 1.6) are necessary to connect 5 LEDs to the LED HAT.

Figure 1.6. Premier Female-Female Jumper Wires

## 1.3. Bumpers

These bumpers are designed to keep the LEDs in place and are therefore used only in configuration `DB17-1`. They are custom designed parts, so they must be produced and cannot be bought. We used laser cutting facilities.

Figure 1.7. The Bumpers

## 1.4. Headers, resistors and jumper

Upgrading `DB17` to `DB17-1` requires several electrical bits: 5 of 4 pin female header, 2 of 16 pin male headers, 1 of 12 pin male header, 1 of 3 pin male header, 1 of 2 pin female shunt jumper, 5 of 200 Ohm resistors and finally 10 of 130 Ohm resistors.

These items require soldering.

Figure 1.8. The Headers

Figure 1.9. The Resistors

## 1.5. Caster (DB17-c)

The caster (Figure 1.10) is an `DB17-c` component that substitutes the steel omnidirectional wheel that comes in the Magician Chassis package. Although the caster is not essential, it provides smoother operations and overall enhanced Duckiebot performance.

Figure 1.10. The caster wheel

To assemble the caster at the right height we will need to purchase:

- 4 standoffs (M3 12mm F-F) (Figure 1.11a - Standoffs for caster wheel.),
- 8 screws (M3x8mm) (Figure 1.11b - Screws for caster wheel.), and
- 8 split lock washers (Figure 1.11c - Split lock washers for caster wheel.).

(a) Standoffs for caster wheel.

(b) Screws for caster wheel.

(c) Split lock washers for caster wheel.

Figure 1.11. Mechanical bits to assemble the caster wheel.

**TODO:** missing figures, update caster bits figures

# Unit E-2
# Soldering boards (DB17-1)

**Note:** General rule in soldering

- soldering the components according to the height of components - from lowest to highest

Assigned to: Shiying

Knowledge and activity graph

**Requires:** Duckiebot `DB17-1` parts. The acquisition process is explained in Unit E-1 - Acquiring the parts (`DB17-1c`). The configurations are described in Unit C-1 - Duckiebot configurations.

**Requires:** Time: 30 minutes

**Results:** A `DB17-1` Duckiebot

## 2.1. General rules

General rule in soldering:

- soldering the components according to the height of components - from lowest to highest

## 2.2. 16-channel PWM/Servo HAT

(alternative instructions: how to solder on the PWM/Servo HAT)

### 1) Prepare the components

Put the following components on the table according the Figure

- GPIO Stacking Header for A+/B+/Pi 2
- Adafruit Mini Kit of 16-Channel PWM / Servo HAT for Raspberry Pi
  - 3x4 headers (4x)
  - 2-pin terminal block
  - 16-Channel PWM / Servo HAT for Raspberry Pi (1x)

Figure 2.1.

### 2) Soldering instructions

1. Solder the 2 pin terminal block next to the power cable jack

2. Solder the four 3x4 headers onto the edge of the HAT, below the words “Servo/PWM Pi HAT!”
3. Solder the GPIO Stacking Header at the top of the board, where the 2x20 grid of holes is located.

## 2.3. LSD board

**TODO:** add LSD board image, top and bottom.

### 1) Prepare the components

Put the following components according the figure on the table:

- 1 x 40 pin female header
- 5 x 4 pin female header
- 2 x 16 pin male header
- 1 x 12 pin male header
- 1 x 3 pin male header
- 1 x 2 pin female shunt jumper
- 5 x 200 Ohm resistors

- 10 x 130 Ohm resistors
- 3 x 4 pin male header for servos

Figure 2.2. LSD HAT and all of needed components

### 2) Soldering instructions

1. Put the resistors on the top of the board according to silkscreen markings, solder it on from the bottom side.

Tips:

1. Solder all female headers to the bottom of the board. Alignment becomes easy if the female headers are plugged into the PWM heat, and the LSD board rests on top.
2. Solder all male headers to the top of the board. Male header positions are outlined on the silkscreen.

## 2.4. LED connection

Parts list:

- 4 x 6” female-female jumper cable

Instructions:

1. Connect LED accordingly to silkscreen indication on PRi 2 LSD board
2. silkscreen legend: Rx, Gx, Bx are red, green, and blue channels, accordingly, where x is the LED number; C is a common line (either common anode or common cathode)
3. For adafruit LEDs are common anode type. The longest pin is common anode. Single pin on the side of common is red channel. The two other pins are Green and

Blue channels, with the blue furthest from the common pin.

4. Both types of LEDs are supported. Use shunt jumper to select either common anode (CA) or common cathode (CC) on 3-pin male header. Note, however, that all LEDs on the board must be of the same type.

## 2.5. Putting everything together!

1. Stack the boards
   a. Screw the first eight standoffs into the Pi - provide hints on the location of standoffs and the suggested orientation of the boards w/r to the chassis
   b. connect the camera to the Pi [image showing the connector ?]
   c. Stack the DC/Stepper Motor HAT onto the Pi, aligning both sets of GPIO pins over each other and screw the standoffs to secure it. Try to not bend the camera connector too much during this step
   d. Stack the 16-channel PWM/Servo HAT onto the Pi, both sets of GPIO pins over each other and screw the standoffs to secure it
2. Slide the battery between the two chassis plates
3. Power the PWM/Servo HAT and Pi connecting them to the battery with the cables included in the duckiebox
4. Power the DC/Stepper motor from the PWM/Servo HAT using the male-to-male cable in the duckiebox, connect the positive
5. connect the Pi to the board
6. Finished!

# Unit E-3
# Assembling the Duckiebot (DB17-1c)

Assigned to: Shiying

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Duckiebot `DB17-1c` parts. The acquisition process is explained in Unit E-1 - Acquiring the parts (`DB17-1c`).

**Requires:** Soldering `DB17-1c` parts. The soldering process is explained in Unit E-2 - Soldering boards (`DB17-1`).

**Requires:** Having assembled the Duckiebot in configuration `DB17` (or any `DB17-wjd`). The assembly process is explained in Unit C-5 - Assembling the Duckiebot (`DB17-jwd`).

**Requires:** Time: about 30 minutes.

**Results:** An assembled Duckiebot in configuration `DB17-wjd1c`.

## 3.1. Assembly the Servo/PWM hat (DB17-11)

Recommend: If you have Bumpers or caster to assembly, it is recommend to have Bumpers and caster assembled before the PWM hat. The assembly process is explained in Unit E-4 - Bumper Assembly and [] (#caster_wheel_instruction).

### 1) Locate the components for Servo/PWM hat

- Soldered PWM hat (1x)
- Nylon Standoffs (M3.5 12mm F-F) (4x)
- Power cable: short angled male USB-A to 5.5/2.1mm DC power jack cable
- Male-Male Jumper Wires (1x)
- Screwdriver

Figure 3.1. Components-List for PWM HAT

### 2) Remove the hand-made USB power cable from DC Motor HAT

From now on, the DC Motor Hat will be powered by the PWM HAT via male -male jumper wire. Before that, the previous hand-made USB power cable needed to be removed. Insert the male-male jumper wire into `+` power terminal on the DC motor HAT (DC-end).

Figure 3.2. Insert the male-male wire into `+` terminal block on the DC motor HAT

### 3) Stack the PWM HAT above the DC motor HAT

Put a soldered Servo/PWM HAT board (in your Duckiebox) with 4 standoffs on the top of Stepper Motor HAT.

Insert the other end of male-male jumper wire into "+5"V power terminal on the PWM HAT (PWM-end). It leads the power to DC motor HAT.

Figure 3.3. Insert the PWM-end into +5V terminal on PWM HAT

### 4) Power Supply for PWM HAT

To power the PWM/Servo HAT from the battery, plugin a short (30cm) angled male USB-A to 5.5/2.1mm DC power jack cable into PWM HAT. The other end of the pow-

er cable will plugin to the battery when it is in use.

Figure 3.4. Plugin the short angled male DC power cable

## 3.2. Assembling the Bumper Set (DB17-12)

For instructions on how to assemble your bumpers set, refer to: Unit E-4 - Bumper Assembly.

## 3.3. Assembling the LED HAT and LEDs (DB17-13)

For instructions on how to assemble the LED HAT and related LEDs, refer to: Unit E-5 - DB17-1 setup.

**TODO:** finish above, estimate assembly time, add bumper assembly instructions, add LED positioning and wiring instructions, add castor wheel assembly instructions

# Unit E-4
# Bumper Assembly

Knowledge and activity graph

**Requires:** Duckiebot `DB17-1c` parts.

**Requires:** Having the Duckiebot with configuration `DB17-wjd` assembled. The assembly process is explained in Unit E-3 - Assembling the Duckiebot (`DB17-1c`).

**Requires:** Time: about 15 minutes.

**Results:** A Duckiebot with Bumpers (configuration DB17-l2)

## 4.1. Locate all required parts

The following should be included in your parts envelope (See image below for these components):

- 1x front bumper (Camera side)
- 1x rear bumper (the side of Caster/Omnidirectional wheel)
- 2x rear bumper brace (the side of Caster/Omnidirectional wheel)
- 8x M2.5x10 nylone or metall screws
- 8x M2.5 nuts

The following is not included in your parts envelope but will be needed for assembly:

- Small screwdriver

Figure 4.1. Components in Duckiebot package.

Figure 4.2. screws for fasten the bumpers

#### 1) Reminder: Use care when assembling!

When assembling, be sure to be gentle! Tighten screws just enough so that the parts will remain stationary. When inserting LEDs, gently work them into their holders. While the acrylic is relatively tough, it can be fractured with modest force. We don't have many replacements (at this moment) so we may not be able to replace a broken part.

## 4.2. Remove protective paper

Peel protective layer off of all parts on all sides.

Figure 4.3. Bumpers with its protective layer

## 4.3. Assembly Rear Spacers

#### 1) Disassembly the spacers between the both chassis

The backside of duckiebot before assemblying the bumpers looks as Figure 4.4:

Figure 4.4. The configuration before assemblying the bumpers

Now remove the spacers and the (short)metall screws from the standoffs (configuration 'DB17-wjd') and replace it with 4 M3x10 nylon screws for connecting the chassis and the bumper spacers.

Figure 4.5. The spacers configuration for the bumpers, Left: old configuration, Right: new configuration

M3x10 screws attaching bottom rear brace:

Figure 4.6. Attach the spacers with M3*10 nylone screws

Back View, fully assembled:

Figure 4.7. Back view of rear spacers

> **Note:** For the ETH students, the M3*10 nylon screws for attaching the rear spacers with chassis were already distributed during the duckiebot ceremony and not included in second distribution. Please reuse them!

### 2) Mount Rear Bumper

Carefully guide rear bumper on to rear bumper brace tabs. Ensure that the hole for

charging aligns with the charging port on your battery.

Figure 4.8. Front view of rear bumper

Locate 4 M2.5 nylon/metall nuts and 4 M2.5*10 nylon screws. Place a nut in the wide part of the t-slot and thread a screw into the nut as shown in the following pictures. Note Use care when assembling! If you are having trouble with the nuts falling out, take a small piece of transparent tape and place it over both sides of the t-slot with the nut inside. It won't look as nice but it will be much easier to assemble.

✔ Test the screws and the nuts once by screwing them together before you use it for the bumpers. It make the fllowing assembly process much easier.

> **Note:** For ETH 2017, the screws and nuts using for this step are M2.5*10 nylon screws (white) and M2.5 nylon or metall nuts from the envelope.

Figure 4.9. Hold the nuts with the fingers

Figure 4.10. Screw the screws into nuts while holding the nuts with the fingers

The completed rear bumper should look like this:

Figure 4.11. Completed rear bumpers

Congrats! your rear bumper assembly is complete!

## 4.4. Mount Front Bumper

The front side of duckiebot in 'DB17-wjd' should look like in Figure 4.12

Take 2x M2.5*10 nylon screws and 2x M2.5 nylon nuts and install them as shown in the following pictures. The first picture shows the correct holes to mount these screws (The correct position is the widest pair of 3mm holes beside the camera). The nuts should tightened on by a few threads (these are the two nuts that are not yet tightened at the top of the second picture):

Figure 4.12. Front side of configuration DB17-wjd, without bumpers

✔ Before tighten the front bumper, you should organise the wires of LEDs going through the right holes of chassis. Take a look in Figure 5.4 The center LED should be bent at a right angle in the direction that the wire is fed through the body.

Take the front bumper and carefully press the LEDs into the flexure holders. Take care that the wires are routed behind the front bumper. Also note that the front center LED wire should not be crushed between the bumper and the right spacer (you will likely fracture the bumper if you try to force it). The center LED should be bent at a right angle in the direction that the wire is fed through the body (see Figure 4.13).

Figure 4.13. Insert the LEDs before tighten the front bumper.

Position the bumper so that the nuts align with the t-slots. You may need to loosen or tigthen the screws to align the nuts. You may also need to angle the front bumper when inserting to get it past the camera screws.

Figure 4.14. Completed front bumpers

Gently tighten the nuts. The front bumper should now stay in position.

Congrats! your bumper assembly is complete!

# Unit E-5
# DB17-l setup

Assigned to: Shiying

Knowledge and activity graph

**Requires:** Duckiebot `DB17-lc` parts. The acquisition process is explained in Unit E-1 - Acquiring the parts (`DB17-lc`).

**Requires:** Soldering `DB17-lc` parts. The soldering process is explained in Unit E-2 - Soldering boards (`DB17-l`).

**Requires:** Having assembled PWM Hat on the Duckiebot with configuration `DB17-wjd`. The assembly process is explained in Unit E-3 - Assembling the Duckiebot (`DB17-lc`).

**Requires:** Time: about 15 minutes.

**Results:** A Duckiebot with LEDs attached (configuration DB17-l3)

## 5.1. Locate all required parts

To attach the LEDs on the duckiebots, the following components are needed:

- Soldered LSD board with Jumper
- LED lampes (5x)
- Female-female wires (20x)
- Nylon standoffs M2.5*12
- some Tape

Figure 5.1. Components-List for LED configuration

## 5.2. Connecting Wires to LEDs

## 1) LEDs

The LEDs are common anode type. The longest pin is called the common. The single pin on the side of common is red channel. The two other pins are Green and Blue channels, with the blue furthest from the common pin.

Figure 5.2. LED pins with its functions

Use the long wires with two female ends. Attach one to each of the pins on the LED.

To figure out the order to connect them to the LSD hat, use the legend on the silkscreen and the information above. i.e. RX - means the red pin, CX - means the common, GX means the green, and BX means the blue. The “X” varies in number from 1-5 depending on which LED is being connected as discussed in the next section.

Figure 5.3. Locate 5 groups of 4 pins with label RX, C, GX, BX) on the LSD board

✔ Use Tape to keep the LEDs stick with the wires.

## 5.3. Connecting LEDs to LSD Hat

Silkscreen legend: Rx, Gx, Bx are red, green, and blue channels, accordingly, where x is the LED number; C is a common line (either common anode or common cathode).

Define the following names for the lights:

- “top” = top light - the “top” light is now at the bottom
- fl = front left
- fr = front right
- br = back right
- bl = back left

The LEDs are wired according to Figure 5.4.

Figure 5.4

Mappings from the numbers on the LED hats to the positions shown (TOP is now the one in the middle at the front)

- FR -> 5
- BR -> 4
- TOP -> 3
- BL -> 2
- FL -> 1

## 5.4. Running the Wires Through the Chassis

It is advised that the LED cables are routed through the positions noted in the images below before installing the bumpers:

Front Left, Front Middle, and Front Right LED Wiring suggestion:

Figure 5.5

## 5.5. Final tweaks

Adjust the LED terminals (particularly in the front) so that they do not interfere with the wheels. This can be accomplished by bending them up, away from the treads.

# Part F
# Operation Manual - demos

This part describes a set of demos that demonstrate the functionality of the robots. Some of these demos require different hardware on the robot, or in Duckietown.

UNIT F-1

# Demo template

This is the template for the description of a demo.

First, we describe what is needed, including:

- Robot hardware
- Number of Duckiebots
- Robot setup steps
- Duckietown hardware

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Duckiebot in configuration ???

**Requires:** Camera calibration completed.

## 1.1. Video of expected results

First, we show a video of the expected behavior (if the demo is succesful).

## 1.2. Duckietown setup notes

Here, describe the assumptions about the Duckietown, including:

- Layout (tiles types)
- Instrastructure (traffic lights, wifi networks, ...) required
- Weather (lights, ...)

Do not write instructions here. The instructions should be somewhere in the part about Duckietowns. Here, merely point to them.

## 1.3. Duckiebot setup notes

Write here any special setup for the Duckiebot, if needed.

Do not write instructions here. The instructions should be somewhere in the appropriate setup part.

## 1.4. Pre-flight checklist

The pre-flight checklist describes the steps that are sufficient to ensure that the demo will be correct:

Check: operation 1 done

Check: operation 2 done

## 1.5. Demo instructions

Here, give step by step instructions to reproduce the demo.

Step 1: XXX

Step 2: XXX

## 1.6. Troubleshooting

Add here any troubleshooting / tips and tricks required.

## 1.7. Demo failure demonstration

Finally, put here a video of how the demo can fail, when the assumptions are not respected.

# Unit F-2
# Lane following

This is the description of lane following demo.

Knowledge and activity graph

**Requires:** Wheels calibration completed wheel calibration
**Requires:** Camera calibration completed Camera calibration
**Requires:** Joystick demo has been successfully launched Joystick demo
**Requires:** Duckiebot in configuration DB17-jwd
**Requires:** Calibrating the gain parameter to **0.6**.

## 2.1. Video of expected results

Video of demo lane following

## 2.2. Duckietown setup notes

Assumption about Duckietown:

- A duckietown with white and yellow lanes. No obstacles on the lane.
- Layout conform to Duckietown Appearance Specifications
- Required tiles types: straight tile, turn tile
- Additional tile types:3-way/4-way intersection
- Configurated Wifi network or Duckietown Wifi network

Environment of demo:

- Good lightning

## 2.3. Duckiebot setup notes

- Make sure the camera is heading ahead.
- Duckiebot in configuration DB17-jwd
- Calibrating the gain parameter to **0.6**.

## 2.4. Pre-flight checklist

Check: Turn on joystick. Check: Turn on battery of the duckiebot. Check: Duckiebot drives correctly with joystick.

## 2.5. Demo instructions

Step 1: On duckiebot, in /DUCKIERTOWN_ROOT/ directory, run command:

```
$ make demo-lane-following
```

Wait a while so that everything has been launched. Drive around with joystick to check if connection is working.

Step 2: Press X to recalibrate Anti-Instagram.

Step 3: Start the autonomous lane following by pressing the **R1** button on joystick. to start autonomous.

Step 4: The Duckiebot should drive autonomously in the lane. Intersections and red lines are neglected and the Duckiebot will drive across them like it is a **normal** lane. Enjoy the demo.

Step 5: Stop the autonomous driving by pressing **L1** button on the joystick and switch to joystick control.

## 2.6. Troubleshooting

### 1) The Duckiebot does not drive nicely across intersections

This is not a valid failure. The Duckiebot assumes only normal lanes during this demo. There is no module concerning the intersections. Therefore, weird behavior at intersections is expected and normal because there are no lines. The demo is only to demonstrate the lane following.

### 2) The Duckiebot does not drive nicely in the lane

Solution 1:

Step 1: Turn on line segments.

```
$ rosparam set /'robotname'/line_detector_node/verbose true
```

Step 2: Open rviz.

```
$ rviz
```

Step 3: Look at the .../image_with_lines image output. Apply the anti-instagram calibration by pushing the X button on the joystick. Check if you see enough segments. If not enough segments are visible, press X button on joystick for Anti-Instagram relaunch. Check if you see more segments and the color of the segments are according to the color of the lines in Duckietown

Solution 2: * Check the extrinsic and intrinsic calibration

### 3) Demo does not compile

Solution: * Run what-the-duck and follow instructions .

### 4) Duckiebot drives not with joystick

Solution:

- Turn joystick on and off multiple times.
- Check if battery is powered on.

#### 5) The Duckiebot cuts white line while driving on inner curves (avanced)

Solution (advanced):

Set alternative controller gains. While running the demo on the Duckiebot use the following to set the gains to the alternative values:

```
$ rosparam set /robot_name/lane_controller_node/k_d -45

$ rosparam set /robot_name/lane_controller_node/k_theta -11
```

Those changes are only active while running the demo and need to be repeated at every start of the demo if needed. If this improved the performance of your Duckiebot, you should think about permenantly change the default values in your catkin_ws.

## 2.7. Demo failure demonstration

If Anti-Instagram is badly calibrated, the duckiebot will not see enough line segments. This is especially a problem in the curve and the duckiebot will leave the lane. An example of this failure can be seen in this video for which we had a bad Anti-Instagram calibration. Hence, the Duckiebot sees not enough line segments and the lane following fails in the curve. To solve the problem Anti-Instagram needs to be relaunched. In the last part of the video the **X** button on the joystick is pressed and the Anti-Instagram node gets relaunched. We can see in RVIZ that the number of detected line segments gets increased drastically after the recalibration.

UNIT F-3

# Demo Saviors

This is the description of the saviors obstacle avoidance demo.

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Duckiebot in configuration DB17-wjd.
**Requires:** Camera calibration completed.
**Requires:** Wheel calibration completed.
**Requires:** Joystick demo successfully launched.

## 3.1. Video of expected results

As it can be inferred from the video, the duckiebot should stop if an obstacle (duckie or cone) is placed in front of the Duckiebot.

## 3.2. Duckietown setup notes

Duckietown built to specifications. No special requirements like april-tags, traffic lights or similar needed.

To demonstrate functionality, place obstacles (duckies S/M/L or cones) on driving lane. Best performance is achieved when obstacles are placed on the straights, not immediately after a curve.

## 3.3. Duckiebot setup notes

Currently the bot has to be on the devel-saviors-23feb branch on git. Furthermore the **additional package sk-image has to be installed:**

```
$ sudo apt-get install python-skimage
```

## 3.4. Pre-flight checklist

Check: Joystick is turned on

Check: Sufficient battery charge on duckiebot

## 3.5. Demo instructions

Step by step instructions to run demo:

**Step 1:** On the duckiebot, navigate to `DUCKIETOWN_ROOT` and run

```
$ source environment.sh

$ catkin_make -C catkin_ws/

$ make demo-lane-following
```

Wait for a couple of seconds until everything has been properly launched.

**Step 2:** In a second terminal on the duckiebot, run:

```
$ roslaunch obst_avoid obst_avoid_lane_follow_light.launch veh:=robot_name
```

This launches the obstacle avoidance node, wait again until it's properly started up.

**Step 3:** Press the X button on the joystick to generate an anti-instagram transformation.

Within about the next 10 seconds in the terminal of Step2 this **YELLOW** message should appear:

`!!!!!!!!!!!!!!!!!!!!TRAFO WAS COMPUTED SO WE ARE READY TO GO!!!!!!!!!!!!`

**Step 4:** To *(optionally)* visualise output of the nodes run the following commands on your notebook:

```
$ source set_ros_master.sh robot_name

$ roslaunch obst_avoid obst_avoid_visual.launch veh:=robot_name

$ rviz
```

Topics of interest are:

`/robot_name/obst_detect_visual/visualize_obstacles` (Markers which show obstacles, visualize via rviz!),

`/robot_name/obst_detect_visual/image/compressed` (Image with obstacle detection overlay, visualize via rqt!),

`/robot_name/obst_detect_visual/bb_linelist` (bounding box of obstacle detection, visualize via rqt),

`/robot_name/duckiebot_visualizer/segment_list_markers` (line segments).

**Step 5:** To drive press R1 to start lane following. Duckiebot stops if obstacle detected and in reach of the duckiebot. Removal of the obstacle should lead to continuation of lane following.

## 3.6. Troubleshooting

P: Objects aren't properly detected, random stops on track.

S: Make sure that anti instagram was run properly. Repeat Step 3 if needed.

P: Duckiebot crashes obstacles.

S: Might be due to processes not running fast enough. Check if CPU load is too high, reduce if needed.

## 3.7. Further Reading

More information and details about our software packages can be found in our README on Github or in our Final Report.

# Unit F-4
# Intersection Navigation

## Knowledge and activity graph

**Requires:** ???

UNIT F-5

# Indefinite Navigation

This is the description of the indefinite navigation demo.

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Wheels calibration completed.wheel calibration

**Requires:** Camera calibration completed.Camera calibration

**Requires:** Joystick demo has been successfully launched.Joystick demo

**Requires:** Fully set up Duckietown (including April tags for intersections)

**Requires:** A motor gain of approximately 0.65 (strong influence in open-loop intersections)

## 5.1. Video of expected results

Figure 5.1. Demo: indefinite navigation

**TODO:** add a different video with an up to specification Duckietown.

## 5.2. Duckietown setup notes

A Duckietown with white and yellow lanes. No obstacles on the lane. Red stop lines at intersections. If several Duckiebots are present while running this demo, LEDs need to be installed for explicit communication and coordination of Duckiebots at intersections.

## 5.3. Duckiebot setup notes

Make sure the camera is securely tightened and properly calibrated.

## 5.4. Pre-flight checklist

Check: Joystick is turned on.

Check: Sufficient battery charge of the Duckiebot.

Check: Gain is set to approximately 0.65.

## 5.5. Demo instructions

Follow these steps to run the indefinite navigation demo on your Duckiebot:

Step 1: On the Duckiebot, navigate to the `/DUCKIETOWN_ROOT/` directory, run the command:

```
$ make indefinite-navigation
```

Wait until everything has been launched. Press X to start anti-instagram. Pressing R1 will start the autonomous lane following and L1 can be used to revert back to manual control.

In the current open-loop intersection navigation, no Duckiebot will successfully run the demo the first time. Parameter tuning is a must. The only two parameters that should be adjusted are the gain and trim, previously defined during the wheel calibration procedure.

The parameter pair which makes your bot go straight will unlikely work for the lane following due to the current controller design. Start with your parameter pair obtained from wheel calibration. If your Duckiebot stays too long on a curve during crossing an intersection, decrease your gain in steps of 0.05. If the Duckiebot doesn't make the turn enough long, increase your gain in steps of 0.05.

Command to modify your gain (in this example to 0.65):

```
$ rosservice call /robot name/inverse_kinematics_node/set_gain -- 0.65
```

+ question

on laptop or bot?

Everything below is helpful for debugging if your robot does not follow the lane at all.

Step 2: Navigate to the Duckietown folder:

```
$ cd ~/duckietown
```

then source the environment:

```
$ source environment.sh
```

set the the ROS master to your vehicle:

```
$ source set_ros_master.sh robot name
```

and finally launch rviz:

```
$ rviz
```

In rviz, two markerarrays:

- /*robot name*/duckiebot_visualizer/segment_list_markers, and
- /*robot name*/lane_pose_visualizer_node/lane_pose_markers

can be visualized. The green arrow representing the pose estimate of the robot has to be in a reasonable direction.

Step 3: Always on the laptop, run:

```
$ rqt
```

In rqt, the images can be visualized are:

- /*robot name*/camera_node/image/compressed,
- /*robot name*/line_detector_node/image_with_lines,
- /*robot name*/lane_filter_node/belief_img.

## 5.6. Troubleshooting

**Maintainer:** Contact Julien Kindle(ETHZ) via Slack for further assistance.

# Unit F-6
# Coordination

This demo (instructions from MIT2016) allow multiple Duckiebots that stop at an intersection to coordinate with LED signals and clear the intersection. Two demos are available, one for the case with traffic lights and one without them.

Knowledge and activity graph

**Requires:** Duckiebot in configuration DB17-lc
**Requires:** Camera calibration and kinematic calibration completed.

## 6.1. Duckietown setup notes

The Duckietown used for these demos needs to have the following characteristics:

- Three or four way intersection tiles.
- The intersections must be provided with two signs that have to be clearly visible:

1) The intersection type (stop sign or traffic light),

2) Intersection topology. Traffic light if needed.

## 6.2. Duckiebot setup notes

No special setup is needed for the Duckiebot. If more Duckiebots are available, they should be used too since the demo is about coordination and LED emission and detection. One to four Duckiebots can be placed at an intersection.

## 6.3. Demo instructions

### 1) Openhouse-dp4 (traffic light coordination)

This demo is for the Traffic-light coordination. The intersection must therefore be provided with a traffic light.

Everything should be run from branch: phase-3-integration

- Step 1: Run the following commands:

Make sure you are in the Duckietown folder:

```
$ cd ~/duckietown
```

Activate ROS:

```
$ source environment.sh
```

On the traffic light run:

```
$ make traffic-light veh:=$traffic_light_name
```

On the Duckiebot run:

```
$ make openhouse-dp4
```

- Step 2: Wait for build to finish. Press 'X' to run anti-instagram. After you see the message 'transform has been published', place the Duckiebot on the lane and press 'R1'. To switch to manual mode press 'L1'

*Expected outcomes:*

- The Duckiebot should stop at traffic light intersections, and wait for the green light to flash (with the right frequency).
- When the green light comes on, it will take the Duckiebot a few seconds to react, before proceeding across the intersection.
- Currently the direction of turn is randomized, so the Duckiebot should choose a random turning direction.

### 2) Openhouse-dp5 (stop sign explicit coordination)

This demo is for the stop sign coordination. The intersection is without a traffic light

Everything should be run from branch: phase-3-integration

- Step 1: Run the following commands:

Make sure you are in the duckietown folder:

```
$ cd ~/duckietown
```

Activate ROS:

```
$ source environment.sh
```

On the Duckiebot run:

```
$ make openhouse-dp5
```

- Step 2: Wait for build to finish. Press 'X' to run anti-instagram. After you see the message 'transform has been published', place the Duckiebot on the lane and press 'R1'. To switch to manual mode press 'L1'

*Expected outcomes:*

- The Duckiebot should stop at the intersection and 'look around' for other Duckiebots at the intersection (on the right).
- After other robots at the intersection clears (i.e. finished crossing, others gave 'pass' signal), the Duckiebot should start flashing 'I am going' signal and crosses the intersection.

## 6.4. Troubleshooting

- When it shows "Event:intersection_go", the Duckiebot does not move. This prob-

lem is related to AprilTags.

Solution: go to config/baseline/pi_camera and change the framerate to 30 instead of 15.

- [Warning] Topics '/![robot name]/camera_node/image/rect' and '/![robot name]/camera_node/raw_camera_info' do not appear to be synchronized. Solution:

UNIT F-7

# Implicit Coordination

This demo allows multiple Duckiebots that stop at an intersection to coordinate themself without explicit communication and clear the intersection.

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Duckiebot in configuration DB-wj

**Requires:** Camera calibration completed. camera calibration

**Requires:** Wheel calibration completed. wheel calibration

**Requires:** Object Detection setup.

## 7.1. Video of expected results

First, we show a video of the expected behavior (if the demo is succesful).

## 7.2. Duckietown setup notes

The Duckietown used for this demo has to be equipped with...

- at least one intersection.
- april tags providing information about the intersection type.

## 7.3. Duckiebot setup notes

No special setup is needed for the Duckiebot.

## 7.4. Laptop setup notes

Tensorflow needs to be installed:

Installing Tensorflow on Ubuntu

Inference model needs to be added:

Move to correct folder:

```
$ cd ~/duckietown/catkin_ws/src/80-deep-learning/object_detection/models
```

Download model from here and place in the folder

## 7.5. Pre-flight checklist

Check: Joystick is turned on.

Check: Sufficient battery charge of the Duckiebot.

## 7.6. Demo instructions

Everything should be run from branch 'devel-implicit-coordination-intersection', don't forget to build again.

Step 1: Place the participating Duckiebots on the ways to the same intersection with enough distance to recognize stop lines.

Step 2: Run the following commands on the Duckiebots:

Make sure you are in the Duckietown folder:

```
$ cd ~/Duckietown
```

Run the demo:

```
$ make demo-implicit_coordination
```

Step 3: Run the following commands on the laptop:

Make sure you are in the Duckietown folder:

```
$ cd ~/Duckietown
```

Set up the environment:

```
$ source environment.sh
```

Set your duckiebot as the ROS master:

```
$ source set_ros_master.sh robot name
```

Run multivehicle detection:

```
$ roslaunch duckietown multivehicle_detection.launch veh:=robot name
```

Step 4: Press R1 on your connected gamepad. The duckiebots now perform lane following until they stop at the intersection, where they coordinate themself and clear the intersection.

## 7.7. Troubleshooting

## 7.8. Demo failure demonstration

coming soon

UNIT F-8

# Explicit Coordination

This demo allows different Duckiebots to coordinate at an intersection using LED-based communication. It handles three or four way intersections with or without a traffic light.

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Duckiebot in configuration DB17-lc

**Requires:** Camera calibration and kinematic calibration completed.

## 8.1. Duckietown setup notes

The Duckietown used for this demo needs to have the following characteristics:

- Three or four way intersection tiles.
- The intersections must be provided with two signs that have to be clearly visible:

1) The intersection type (stop sign or traffic light),

2) Intersection topology. Traffic light if needed.

## 8.2. Duckiebot setup notes

No special setup is needed for the Duckiebot. If more Duckiebots are available, they should be used too since the demo is about coordination and LED emission and detection. One to four Duckiebots can be placed at an intersection.

## 8.3. Demo instructions

### 1) If there is a traffic light:

To start the traffic light:

- Make sure the traffic light is connected to power
- From your laptop connect to wifi TrafficLight# (# corresponds to a decimal number), the password is 'Traff!cL!ght'

SSH into the raspberry pi:

```
$ ssh tlo@trafficlight#.local
```

The password is 'Traff!cL!ght'

- To start the demo on the traffic light:

Make sure you are on the branch new_light_traffic:

```
$ cd ~/duckietown

$ git fetch --all

$ git checkout new_light_traffic
```

Start the demo:

```
$ make traffic-light
```

2) On the Duckiebot:

Make sure you are in the Duckietown folder:

```
$ cd ~/duckietown
```

Checkout to branch devel-explicit-coord-jan15:

```
$ git fetch --all

$ git checkout devel-explicit-coord-jan15
```

Run the demo:

```
$ make coordination2017
```

Wait for build to finish. Place the Duckiebot on the lane and press 'R1'. To switch to manual mode press 'L1'

*Expected outcomes:*

- The Duckiebot should stop at traffic light intersections, and wait for the green light to flash.
- The Duckiebot should stop at red stop line and coordinate with other Duckiebots that are also stopped at the same intersection, if there are some, according to the communication protocol and wait until it has assessed that it can proceed to navigate the intersection.
- When the green light comes on, or the Duckiebot has decided that is time to go, it will start to clear the intersection.
- Currently the direction of turn is randomized, so the Duckiebot should choose a random turning direction.

## 8.4. Troubleshooting

- When it shows "Event:intersection_go", the Duckiebot does not move. This problem is related to AprilTags.

Solution: go to config/baseline/pi_camera and change the framerate to 30 instead of 15.

• [Warning] Topics '/![robot name]/camera_node/image/rect' and '/![robot name]/camera_node/raw_camera_info' do not appear to be synchronized. Solution:

# Unit F-9
# Parallel Autonomy

## Knowledge and activity graph

**Requires:** ???

# Unit F-10
# Follow Leader

This is the description of the follow the leader demo.

Knowledge and activity graph

**Requires:** Wheels calibration completed.wheel calibration
**Requires:** Camera calibration completed.Camera calibration
**Requires:** Lane following demo works.Lane following demo
**Requires:** Circle grid tags on Duckiebots.
**Requires:** More than one Duckiebot

## 10.1. Video of expected results

First, we show a video of the expected behavior (if the demo is successful).

## 10.2. Duckietown setup notes

A Duckietown with white and yellow lanes. No obstacles on the lane.

## 10.3. Duckiebot setup notes

Make sure the camera is heading ahead. Tighten the screws if necessary. Attach circle grid with tape at the rear. A pdf file with circle grids in the right size (0.0125m between dot centers) can be found in /DUCKIETOWN_ROOT/catkin_ws/src/50-misc-additional-functionality/vehicle_detection/CircleGrid.pdf.

Figure 10.1. Duckiebot with Circle Grid

## 10.4. Pre-flight checklist

Check: Joystick is turned on.

Check: Sufficient battery charge of the Duckiebot.

## 10.5. Demo instructions

Step 1: Pull the branch devel-implicit-coord-formationKeeping from the Duckietown repository and run catkin_make.

Place more than one Duckiebot on the outer lanes of Duckietown, such that the Duckiebots drive an circle around Duckietown with lane following. The distance between the Duckiebots should be more than 0.5m to start the demo successfully.

Run the following commands on different Duckiebots at the same time.

Step 1: On Duckiebot, in /DUCKIETOWN_ROOT/ directory, checkout the branch devel-implicit-coord-formationKeeping and run the following commands:

```
$ catkin_make -C catkin_ws/

$ make demo-vehicle_follow_leader
```

Wait a while so that everything has been launched. Press R2 to start autonomous lane following with vehicle avoidance. Press L1 to switch to joystick control or R1 to switch to lane following without vehicle avoidance. In autonomous lane following mode with vehicle avoidance the Duckiebot is keeping a distance to another Duckiebot on front of him.

Step 2: On laptop, make sure ros environment has been activated.

```
$ rqt_image_view
```

In rqt_image_view the images with circle grid are visualized: /(vehicle_name)/vehicle_detection_node/circlepattern_image.

If the circle grid is detected, the grid is drawn in the image.

## 10.6. Troubleshooting

Add here any troubleshooting / tips and tricks required.

## 10.7. Demo failure demonstration

Finally, put here a video of how the demo can fail, when the assumptions are not respected.

# Part G
# Preliminaries

TODO: to write

# Unit G-1
# Chapter template

Theory chapters benefit from a standardized exposition. Here, we define the template for these chapters. Rememeber to check Unit B-2 - Basic Markduck guide for a comprehensive and up-to-date list of Duckiebook supported features.

## 1.1. Example Title: PID control

Start with a brief introduction of the discussed topic, describing its place in the bigger picture, justifying the reading constraints/guidelines below. Write it as if the reader knew the relevant terminology. For example:

PID control is the simplest approach to making a system behave in a desired way rather than how it would naturally behave. It is simple because the measured output is directly feedbacked, as opposed to, e.g., the system's states. The control signal is obtained as a weighted sum of the tracking error (_P_roportional term), its integral over time (_I_ntegrative term) and its instantaneous derivative (_D_erivative term), from which the appellative of PID control. The tracking error is defined as the instantaneous difference between a reference and a measured system output.

Knowledge and activity graph

Knowledge necessary:

Required Reading: Insert here a list of topics and suggested resources related to *necessary* knowledge in order to understand the content presented. Example:

**Requires:** Terminology: autonomy overview

**Requires:** System Modeling: basic kinematics, basic dynamics, linear algebra, State space representations, Linear Time Invariant Systems

Knowledge and activity graph

Suggested Reading: Insert here a list of topics and suggested resources related to *recommended* knowledge in order to better understand the content presented. Example:

**Recommended:** Definitions of Stability, Performances and Robustness: [7], ...

**Recommended:** observability/detectability and controllability/reachability: [7]

**Recommended:** Discrete time PID: [7]

**Recommended:** Bode diagrams: [7]

**Recommended:** Nyquist plots: [7]

**Recommended:** [...]

## 1.2. Problem Definition

In this section we crisply define the problem object of this chapter. It serves as a very

brief recap of exactly what is needed from previous atoms as well. E.g.

Let:

$$\begin{aligned} \dot{x}_t &= Ax_t + Bu_t \\ y &= Cx_t + Du_t \end{aligned} \tag{1}$$

be the LTI model of the Duckiebot's plant, with $x \in \mathcal{X}$, $y \in \mathbb{R}^p$ and $u \in \mathbb{R}^m$. We recall (Duckiebot Modeling) that:

$$\begin{aligned} A &= \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \\ B &= [b_1 \ \dots \ b_m]^T \\ C &= [c_1 \ \dots \ c_p] \\ D &= 0. \end{aligned}$$

[...]

Remember you can use the `problem` environment of LaTeX to formally state a problem:

**Problem 2.** (PID) Given a system (1) and measurements of the output $\tilde{y}_t = y_t + n_t, n_t \sim \mathcal{N}(0, \sigma)$, find a set of PID coefficients that meet the specified requirements for: - stability, - performance, - robustness.

as shown in (Figure 1.1).

Figure 1.1. A classical block diagram for PID control. We like to use a lot of clear figures in the Duckiebook.

## 1.3. Introduced Notions

### 1) Section 1: title-1 (e.g.: Definitions)

**Definition 13.** (Reference signals) A reference signal $\tilde{y}_t \in \mathcal{L}_2(\mathcal{T})$ is ...

Definition 13 - Reference signals is very important.

> **Check before you continue**
> Insert 'random' checks to keep the reader's attention up:
> if you can't be woken up in the middle of the night and remember the definition

of $\mathcal{L}_2(\cdot)$, read: [7]

**Definition 14.** (Another definition) Lorem

#### 2) Section 2: title-2 (e.g.: Output feedback)

Now that we know what we're talking about, lets get in the meat of the problem. Here is what is happening:

$$\mathcal{L}orem$$

#### 3) Section 3: title-3 (e.g.: Tuning the controller)

Introduce the 'synthesis through attempts' methodology (a.k.a. tweak until death)

#### 4) Section 4: title-4 (e.g.: Performance Metrics)

How do we know if the PID controller designed above is doing well? We need to define some performance metrics first:

Overshoot, Module at resonance, Settling Time, Rising Time

[...]

*example*

This is a 'think about it' interrupt, used as attention grabber:

When a Duckiebot 'overshoots', it means that [...] and the following will happen [...].

#### 5) Section N: title-N (e.g.: Saving the world with PID)

And finally, this is how you save the world, in theory.

## 1.4. Examples

This section serves as a collection of theoretical and practical examples that can clarify part or all of the above.

#### 1) Theoretical Examples

More academic examples

*T-Example 1:*

Immagine a spring-mass-damper system...

*T-Example M:*

[...]

#### 2) Implementation Examples

More Duckiebot related examples

*I-Example 1:*

*I-Example M:*

[...]

## 1.5. Pointers to Exercises

Here we just add references to the suggested exercises, defined in the appropriate exercise chapters.

## 1.6. Conclusions

- What did we do? (recap)
- What did we find? (analysis)
- Why is it useful? (synthesis)
- Final Conclusions (what have we learned)

## 1.7. Next Steps

Strong of this new knowledge (what have we learned), we can now [...].

KNOWLEDGE AND ACTIVITY GRAPH

Further Reading: insert here reference resources for the interested reader:

- ☀ learn all there is to know about PID: [7]
- ☀ become a linear algebra master: Matrix cookbook

## 1.8. References

Do not include a reference chapter. References are automatically compiled to the Bibliography Section.

**Author:** Jacopo

**Maintainer:** Jacopo

**Point of contact:** Jacopo

# Unit G-2
# Symbols and conventions

Assigned to: Andrea

## 2.1. Conventions

You should not have to use presentation macros like `\mathcal`, `\boldsymbol`, etc.; rather, for each class of things that we have (set, matrices, random variables, etc.) we are going to define a LaTeX macro.

### 1) Sets

Use the macro `\aset{X}` to refer to the set $\mathcal{X}$.

### 2) Matrices

Use the macro `\amat{M}` to refer to the matrix $\mathbf{M}$.

### 3) Tuples

To indicate tuples, use the macro `\tup`, which produces $\langle a, b, c \rangle$.

### 4) Time series

If $x$ is a function of time, use $x_t$ rather than $x(t)$.

✖ Consider the function $x(t)$.

✔ Consider the function $x_t$.

To refer to the time variable, use `\Time`: $t \in \mathbb{T}$.

### 5) Random variables

To refer to a random variable, use the macro `\rv`. This is rendered using a **bold** symbol.

**Example .** $p(\boldsymbol{x} = x_0)$ is the probability that the random variable $\boldsymbol{x}$ has the value $x_0 \in \mathcal{X}$.

### 6) Well-known sets

Use `\reals` for the real numbers.

Use `\nats` for the natural numbers.

Use `\ints` for the integers numbers.

TABLE 2.1. BASIC SYMBOLS

| command | result | |
|---|---|---|
| `\aset{X}, \aset{Y}` | $\mathcal{X}, \mathcal{Y}$ | Symbols for sets |
| `\amat{M}, \amat{P}` | $\mathbf{M}, \mathbf{P}$ | Symbols for matrices |
| `\avec{u}, \avec{v}` | $\mathbf{u}, \mathbf{v}$ | Symbols for vectors |
| `\nats` | $\mathbb{N}$ | Natural numbers |
| `\ints` | $\mathbb{Z}$ | Integers |
| `\reals` | $\mathbb{R}$ | Real numbers |
| `\definedas` | $\triangleq$ | Defined as |
| `\tup{a,b,c}` | $\langle a, b, c \rangle$ | Tuples |
| `\Time` | $\mathbb{T}$ | Time axis |

## 2.2. Spaces

Here are some useful symbols to refer to geometric spaces.

TABLE 2.2. SPACES

| command | result | |
|---|---|---|
| `\SOthree` | $\mathrm{SO}(3)$ | Rotation matrices |
| `\SEthree` | $\mathrm{SE}(3)$ | Euclidean group |
| `\SEtwo` | $\mathrm{SE}(2)$ | Euclidean group |
| `\setwo` | $\mathfrak{se}(2)$ | Euclidean group algebra |

States and poses:

TABLE 2.3. POSES AND STATES

| command | result | |
|---|---|---|
| `\pose` | $\boldsymbol{q}_t \in \mathrm{SE}(2)$ | Pose of the robot in the plane |
| `\state_t \in \statesp` | $\boldsymbol{x}_t \in \mathcal{X}$ | System state (includes the pose, and everything else) |

# Unit G-3
# Sets

Assigned to: Dzenan

Knowledge and activity graph

Results: k:sets

## 3.1. Definition

**Definition 15.** (Set) A set $\mathcal{X} = \{x_1, x_2, \ldots\}$ is a well-defined collection of distinct *elements*, or *members* of the set, $x_i$, $i = 1, 2, \ldots$.

## 3.2. Maps

Knowledge and activity graph

Requires: k:sets
Results: k:maps

## 3.3. Definition

We define a *function* (or *map*) as a mapping between *sets*.

**Definition 16.** (Function) A function $f : \mathcal{X} \to \mathcal{Y}$ is a mapping between the sets $\mathcal{X}$ and $\mathcal{Y}$. For every input element $x \in \mathcal{X}$, the mapping will associate an output $y = f(x) \in \mathcal{Y}$ .

## 3.4. Properties of maps

Maps can be classified by the nature of the relationship between inputs and outputs in: *injective*, *surjective* or *bijective* add-ref.

### 1) Injective maps

TODO: to write

### 2) Surjective maps

TODO: to write

### 3) Bijective maps

TODO: to write

# Unit G-4
# Numbers

Knowledge and activity graph

**Requires:** k:sets
**Results:** k:naturals, k:integers, k:reals

## 4.1. Natural numbers

$\mathbb{N} = \{0, 1, 2, \cdots\}$

The natural numbers are the set positive numbers, including zero.

Given two natural their addition is always a natural number:

$$a + b = c \in \mathbb{N}, \forall a, b \in \mathbb{N}. \tag{1}$$

The same does not hold of the subtraction operation:

$$a - b = c \in \mathbb{N} \iff a \geq b.$$

For this reason set of integer numbers is defined.

## 4.2. Integers

$\mathbb{Z} = \{\cdots, -2, -1, 0, 1, 2, \cdots\}$

The integers are the set of positive and negative natural numbers, including the zero. By definition, the set of integers includes the naturals: $\mathbb{Z} \subset \mathbb{N}$.

The sum (i.e., addition and subtraction) of two integers is always an integer:

$$a + b = c \in \mathbb{Z}, \forall a, b \in \mathbb{N}$$
$$a - b = c \in \mathbb{Z}, \forall a, b \in \mathbb{N}.$$

The multiplication of two integers is always an integer, but the same does not apply for the division operation:

$$\frac{a}{b} = c \in \mathbb{Z} \iff a = kb, k \in \mathbb{Z}, b \neq 0.$$

For this reason the rational numbers are introduced.

## 4.3. Rationals

The set of rational numbers includes all fractions of integers: $\mathbb{Q} = \{c | \frac{a}{b} = c, a, b \in \mathbb{Z}, b \neq 0\}$.

The set of rational number is complete under sum and product (i.e., multiplication and division), but not under other operations such as the root. E.g., $\sqrt{2}$ cannot be expressed as a fraction of two integers. These numbers are not rational, and therefore

are defined as irrationals.

## 4.4. Irrationals

Irrational numbers are all those numbers that cannot be expressed as a fraction. Notable examples of irrational numbers include the aforementioned $\sqrt{2}$, but even pi ($\pi$) and the Euler number ($e$).

Irrational numbers are not typically referred to as as a set by themselves, rather, the union of the rational and irrational numbers defines the set of *reals*.

## 4.5. Reals

The real numbers ($\mathbb{R}$) are arguably the most used set of numbers, and are often considered the default set if no specification is provided.

The real numbers are defined as the union of rational and irrational numbers, and therefore by definition include the integers and the naturals.

The reals are still not complete under all "canonical" operations. In fact, there is no solution to the root (of even index) of a negative number.

For this reason, the complex numbers are introduced.

## 4.6. Complex

Complex numbers are defined as the sum of a real and an imaginary part:

$$z = a + ib, a, b \in \mathbb{R}, i = \sqrt{-1}$$

and can be represented on the plane of Gauss, a Cartesian plane featuring the real part of $z$, $Re(z) = a$, on the x-axis and the imaginary part, $Im(z) = b$, on the y-axis (Figure 4.1).

Figure 4.1. The Gaussian plane is used to represent complex numbers

Complex numbers introduce the concept of *phase* of a number, which is related to its "orientation", and are invaluable for describing many natural phenomena such as electricity and applications such as signal decoders.

For more information on the algebra and properties of natural numbers:

➡ Unit G-5 - Complex numbers.

UNIT G-5

# Complex numbers

Assigned to: Dzenan

## 5.1. Powers of $i$

The powers of $i$

$$
\begin{aligned}
i &= sqrt{-1} \\
i^2 &= -1 \\
i^3 &= i^2 \cdot i = -i \\
i &= i^2 \cdot i^2 = 1 \\
i^5 &= i^4 / cdot i = i \\
&\vdots
\end{aligned}
$$

## 5.2. Complex conjugate

# Unit G-6
# Linearity and Vectors

Assigned to: Jacopo

Linear algebra provides the set of mathematical tools to (a) study linear relationships and (b) describe linear spaces. It is a field of mathematics with important ramifications.

Linearity is an important concept because it is powerful in describing the input-output behavior of many natural phenomena (or *systems*). As a matter of fact, all those systems that cannot be modeled as linear, still can be approximated as linear to gain an intuition, and sometimes much more, of what is going on.

So, in a way or the other, linear algebra is a starting point for investigating the world around us, and Duckietown is no exception.

**Note:** This chapter is not intended to be a comprehensive compendium of linear algebra.

➡ this reference

✳ this other reference

**TODO:** add references throughout all chapter

Knowledge and activity graph

**Requires:** Real numbers are complex for you?: Number theory addref

**Requires:** ∀ is a typo for A and ∈ are Euros? Mathematical symbolic language.

## 6.1. Linearity

In this section we discuss vectors, matrices and linear spaces along with their properties.

Before introducing the these arguments, we need to formally define what we mean by linearity. The word *linear* comes from the latin *linearis*, which means *pertaining to or resembling a line*. You should recall that a line can be represented by an equation like $y = mx + q$, but here we intend linearity as a property of maps, so there is a little more to linearity than lines (although lines *are* linear maps indeed).

To avoid confusions, let us translate the concept of linearity in mathematical language.

**Definition 17.** (Linearity) A function $f : \mathcal{X} \to \mathcal{Y}$ is linear when, $\forall x_i \in \mathcal{X}$, $i = \{1, 2\}$, and $\forall a \in \mathbb{R}$:

$$f(ax_1) = af(x_1), \quad \text{and:} \tag{2}$$
$$f(x_1 + x_2) = f(x_1) + f(x_2) \tag{3}$$

Condition (2) is referred to as the property of *homogeneity* (of order 1), while condi-

tion (3) is referred to as *additivity*.

**Remark 2.** (Superposition Principle) Conditions (2) and (3) can be merged to express the same meaning through:

$$f(ax_1 + bx_2) = af(x_1) + bf(x_2), \forall x_i \in \mathcal{X}, i = \{1, 2\}, \forall a, b \in \mathbb{R}. \tag{4}$$

This equivalent condition (4) is instead referred to as *superposition principle*, which unveils the bottom line of the concept of linearity: adding up (equivalently, scaling up) inputs results in an added up (equivalently, scaled up) output.

## 6.2. Vectors

Let $n$ belong to the set of natural numbers $\mathbb{N}$, i.e., $n \in \mathbb{N}$, and let $a_i \in \mathbb{R}$, $i = \{1, \dots, n\}$ be real coefficients. While $\mathbb{R}$ is the set of real numbers, $\mathbb{R}^n$ is the set of all $n$-tuples of real numbers.

**Definition 18.** (Vector and components) An $n$-dimensional ***vector*** is an $n$-tuple:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \in \mathbb{R}^{n \times 1} \equiv \mathbb{R}^n, \tag{5}$$

of *components* $v_1, \dots, v_n \in \mathbb{R}$.

**Remark 3.** (Vector notation) A more general notation for tuples can be used when denoting vectors:

$$\mathbf{v} = \langle v_1, \cdots, v_n \rangle. \tag{6}$$

In these preliminaries, we will adopt the (5) "engineering" notation as it arguably simplifies remembering vector-matrix operations (Unit G-9 - Matrices and vectors).

You can imagine a vector Figure 6.1 as a "directional *number*", or an arrow that starts a certain point and goes in a certain direction (in $\mathbb{R}^n$). In this representation, the *number* is the length of the arrow, or the *magnitude* of the vector (sometimes referred to even as *modulus*), and it can be derived through the vector's components.

**Definition 19.** (Length of a vector) We define the length, or *modulus*, of a vector $\mathbf{v} \in \mathbb{R}^n$ as:

$$\|\mathbf{v}\| = \sqrt{v_1^2 + \cdots + v_n^2} \in \mathbb{R}. \tag{7}$$

**Remark 4.** (2-norm) Generally speaking, it is not always possible to define the length of a vector (addref). But when it is possible (e.g., in Hilbert spaces), and in Duckietown it always is, there are many ways to define it. The most common and intuitive definition is the *Euclidian-* or *2-norm*, which is defined above in (7).

We will discuss norms more in detail in Unit G-13 - Norms.

**Definition 20.** (Unit vector) A unit vector, or *versor*, is a vector $\mathbf{e}$ of of unit length:

$$\|\mathbf{e}\| = 1. \tag{8}$$

Unit vectors are used to define the directions of the components of a vector, allowing for an algebraic rather than vectorial representation. As we will see in Subsection 6.2.1 - Vector algebra, this will make the algebra of vectors more intuitive.

Figure 6.1. A vector, its components expressed as multiples of unit vectors.

*example*

Let $\mathbf{v} \in \mathbb{R}^3$ be a vector defined in the Cartesian space. Let, moreover, $(\mathbf{i}, \mathbf{j}, \mathbf{k})^T$ be the versor of the Cartesian axis, i.e.:

$$\begin{aligned} \mathbf{i} &= [1, 0, 0]^T; \\ \mathbf{j} &= [0, 1, 0]^T; \\ \mathbf{k} &= [0, 0, 1]^T. \end{aligned} \tag{9}$$

Then, a vector can be written equivalently in vector or algebraic form: $\mathbf{v} = [v_1, v_2, v_3]^T = v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}$. Unit vectors are sometimes explicitly denoted with a hat (^), e.g., $\hat{\mathbf{i}}, \hat{\mathbf{j}}, \hat{\mathbf{k}}$.

**Remark 5.** (Normalizing vectors) Every vector can be made into a unit vector, or *normalized*, by dividing each of its components by the vector's magnitude:

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|} = \left[\frac{v_1}{\|\mathbf{v}\|}, \frac{v_2}{\|\mathbf{v}\|}, \frac{v_3}{\|\mathbf{v}\|}\right]^T. \tag{10}$$

## 1) Vector algebra

We here define operations amongst two given vectors defined in the same space: $\mathbf{u} = [u_1, u_2, u_3]^T, \mathbf{v} = [v_1, v_2, v_3]^T \in \mathbb{R}^3$.

*Vectorial Sum:*

The sum of two vectors is a vector, and its components are the sum of the two vectors components.

**Definition 21.** (Vectorial sum)

$$\mathbf{u} + \mathbf{v} = [u_1 + v_1, u_2 + v_2, u_3 + v_3]^T. \tag{11}$$

**Remark 6.** (Sum) Mathematical operations come in pairs, which represent the same concept. A *sum* operation, sometimes more extensively referred to as the *algebric sum*, is the concept of summing, i.e., it includes both addition and subtraction. (A subtraction is nothing but an addition between positive and negative numbers.)

The parallelogram law helps visualize the results of the vectorial sum operation Fig-

ure 6.2.

Figure 6.2. The sum of two vectors can be visualized with the parallelogram law.

*Dot, or scalar, product:*

The dot, or scalar, product of two vectors ($\mathbf{u}$,$\mathbf{v} \in \mathbb{R}^3$) is a scalar ($a \in \mathbb{R}$) equal to the sum of the products of the components of the vectors. Equivalently, it can be expressed as the product of the magnitudes of the two vectors times the cosine of the angle between them, $\phi \in [0, 2\pi)$.

**Definition 22.** (Scalar product)

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + u_3 v_3 = \|u\| \|v\| \cos(\phi) \in \mathbb{R} \tag{12}$$

The dot product is a measure of the *projection* of vectors on one another (Figure 6.3).

**Note:** When the two vectors are perpendicular, or orthogonal, the dot product is zero ($\cos(\pi/2) = 0$). This fact is often used as a test for orthogonality. Orthogonality is an important concept for linear spaces, as the most "efficient" basis are orthogonal.

Figure 6.3. The scalar product between two vectors measures the projection of one on each other.

*Cross, or vector, product:*

While the dot product depends on the metric chosen in the space (the Euclidian

norm, in our case), the cross product even requires the definition of an orientation, or handedness.

**Proposition 4.** (Standard Basis) In the Euclidian space $\mathbb{R}^3$, $\hat{\mathbf{i}}, \hat{\mathbf{j}}, \hat{\mathbf{k}}$ are the unit vectors for the standard basis, which is right handed.

In a right handed reference system such as the standard basis, the right hand rule (Figure 6.4) is the handy-est way to identify the direction of the vector resulting from a cross product.

ProTip: There is a valid reason for which it is called the *right hand* rule. Don't use your left hand because you are holding a pen with the right one.

Figure 6.4. The right hand rule points in the direction of the resulting vector from a cross product.

The cross, or vector, product between two vectors ($\mathbf{u}$, $\mathbf{v} \in \mathbb{R}^3$) is a vector that is orthogonal to each of the two vectors, hence is normal, or perpendicular, to the plane containing them. Its magnitude is given by the product of their magnitude times the sine of the angle between them, and its direction is indicated by the normal unit vector ($\hat{\mathbf{n}} \in \mathbb{R}^3$), identified by the right hand rule.

**Definition 23.** (Vector product)

$$\mathbf{u} \times \mathbf{v} = [u_2 v_3 - u_3 v_2, u_3 v_1 - u_1 v_3, u_1 v_2 - u_2 v_1]^T = \|u\| \|v\| \sin(\phi) \hat{\mathbf{n}}. \tag{13}$$

**Remark 7.** (Geometric interpretation) A cross product encodes two pieces on information: a direction, which is *orthogonal* to the plane spanned by the two vectors, and a magnitude, which is equal to the area of the parallelogram having $\mathbf{u}$, and $\mathbf{v}$ as sides.

> Note: Keeping **(13)** and Remark 7 - Geometric interpretation in mind, it should be intuitive to understand that:

$$\begin{aligned} \mathbf{v} \times \mathbf{v} &= \mathbf{0}, \forall \mathbf{v} \in \mathbb{R}^n, \\ \mathbf{v} \times \mathbf{0} &= \mathbf{0}, \forall \mathbf{v} \in \mathbb{R}^n. \end{aligned} \tag{14}$$

**Note:** The zero vector ($\mathbf{0}$) is a vector with zero magnitude, not the same as the number zero ($0$).

**Note:** Each component of $\mathbf{w}$ is the difference of the products of the two *other* components of $\mathbf{u}$, and $\mathbf{v}$, in the order given by the chosen handedness of the basis. This combination resembles a *cross* (Figure 6.5), from which the name of *cross product*.

Figure 6.5. Each component of the resulting vector is the product of the alternated other components, forming a cross.

**Note:** The components of a cross product can be computed through the Sarrus rule (see Section 7.6 - Determinant).

As consequence of the vectorial product's definition and right handedness of the basis, the following hold true in the Cartesian space:

$$\begin{aligned} \hat{\mathbf{i}} \times \hat{\mathbf{j}} &= \hat{\mathbf{k}} \\ \hat{\mathbf{j}} \times \hat{\mathbf{k}} &= \hat{\mathbf{i}} \\ \hat{\mathbf{k}} \times \hat{\mathbf{i}} &= \hat{\mathbf{j}}. \end{aligned} \tag{15}$$

## 2) Properties of vectors

In this section we highlight the properties of vector operations, that derive from their definitions.

*Sum:*

The vector sum obejs the following:

- $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$,
- $(\mathbf{u} + \mathbf{v}) + \mathbf{w} = \mathbf{u} + (\mathbf{v} + \mathbf{w})$,
- $a(\mathbf{u} + \mathbf{v}) = a\mathbf{u} + a\mathbf{v}$,
- $(a + b)\mathbf{u} = a\mathbf{u} + b\mathbf{u}$,
- $\mathbf{u} + \mathbf{0} = \mathbf{u}$, therefore $\mathbf{u} + (-\mathbf{u}) = \mathbf{0}$.

*Dot product:*

Letting $\phi \in [0, 2\pi)$ be the angle between two vectors $\mathbf{u}, \mathbf{v}$, the dot product obejs the following:

- $\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\|\|\mathbf{v}\| \cos(\phi)$,
- $\mathbf{u} \cdot \mathbf{u} = \|\mathbf{u}\|^2$,
- $\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{u}$,
- $\mathbf{u} \cdot (\mathbf{v} + \mathbf{w}) = \mathbf{u} \cdot \mathbf{v} + \mathbf{u} \cdot \mathbf{w}$,
- $a(\mathbf{u} \cdot \mathbf{v}) = (a\mathbf{u}) \cdot \mathbf{v}$,
- $\mathbf{0} \cdot \mathbf{u} = 0$
- $\mathbf{u} \cdot \mathbf{v} = 0 \iff \mathbf{u} = \mathbf{0}, \mathbf{v} = \mathbf{0}$, or $\mathbf{u} \perp \mathbf{v}$.

*Cross product:*

Letting $\phi \in [0, 2\pi)$ be the angle between two vectors $\mathbf{u}, \mathbf{v}$, the cross product obejs the following:

- $\mathbf{u} \times \mathbf{v} = \|\mathbf{u}\|\|\mathbf{v}\| \sin(\phi)\hat{\mathbf{n}}$,
- $\mathbf{u} \times \mathbf{v} = -\mathbf{v} \times \mathbf{u}$,
- $(a\mathbf{u}) \times \mathbf{v} = \mathbf{u} \times (a\mathbf{v}) = a(\mathbf{u} \times \mathbf{v})$,
- $\mathbf{u} \times (\mathbf{v} + \mathbf{w}) = \mathbf{u} \times \mathbf{v} + \mathbf{u} \times \mathbf{w}$,
- $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) = (\mathbf{u} \times \mathbf{v}) \cdot \mathbf{w}$,
- $\mathbf{u} \times (\mathbf{v} + \mathbf{w}) = (\mathbf{w} \cdot \mathbf{u})\mathbf{v} - (\mathbf{v} \cdot \mathbf{u})\mathbf{w} \neq (\mathbf{u} \times \mathbf{v}) + \mathbf{w}$,
- $\mathbf{u} \times \mathbf{v} = 0 \iff \mathbf{u} = \mathbf{0}, \mathbf{v} = \mathbf{0}$, or $\mathbf{u} \parallel \mathbf{v}$.

## 6.3. Linear dependance

**Definition 24.** (Linear dependance) Two or more vectors $\{\mathbf{v_1}, \cdots, \mathbf{v_n}\}$ are *linearly dependant* if there exists a set of scalars $\{a_1, \cdots, a_k\}, k \leq n$, that are *not all zero*, such that:

$$a_1\mathbf{v_1} + \cdots + a_k\mathbf{v_k} = \mathbf{0}. \tag{1}$$

Note: When (1) is true, it is possible to write at least one vector as a linear combination of the others.

**Definition 25.** (Linear independance) Two or more vectors $\mathbf{v_1}, \cdots, \mathbf{v_n}$ are *linearly independant* if (1) can be satisfied only by $k = n$ and $a_i = 0, \forall i = 1, \cdots, n$.

## 6.4. Pointers to Exercises

Here we just add references to the suggested exercises, defined in the appropriate exercise chapters.

TODO: add exercises

## 6.5. Conclusions

In this section we have defined the fundamental concept of linearity and linear dependance. Moreover, we have introduced vectors, with their operations and algebraic properties.

Vectors and linearity are the base for understanding linear spaces, which are useful because they introduce some fundamental concepts related to the foundation of

*modeling* of natural phenomena. Modeling will be invaluable in understanding the behavior of systems, and a powerful tool to *predict* future behaviors of the system, and *control* them when needed.

KNOWLEDGE AND ACTIVITY GRAPH

- ☀ [15]
- ☀ Matrix cookbook

**Author:** Jacopo

**Maintainer:** Jacopo

**Point of contact:** Jacopo

# Unit G-7
# Matrices basics

Assigned to: Dzenan

Knowledge and activity graph

Requires: k:basic_math
Requires: k:linear_algebra
Results: k:matrices

A matrix:

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{1}$$

is a table ordered by ($m$) horizontal rows and ($n$) vertical columns. Its elements are typically denoted with lower case latin letters, with subscripts indicating their row and column respectively. For example, $a_{ij}$ is the element of $A$ at the $i$-th row and $j$-th column.

Figure 7.1. A matrix. This image is taken from [17]

Note: A vector is a matrix with one column.

## 7.1. Matrix dimensions

The number of rows and columns of a matrix are referred to as the matrix *dimensions*. $\mathbf{A} \in \mathbb{R}^{m \times n}$ has dimensions $m$ and $n$.

**Definition 26.** (Fat matrix) When $n > m$, i.e., the matrix has more columns than rows, $\mathbf{A}$ is called *fat* matrix.

**Definition 27.** (Tall matrix) When $n < m$, i.e., the matrix has more rows than columns, $\mathbf{A}$ is called *tall* matrix.

**Definition 28.** (Fat matrix) When $n = m$, $\mathbf{A}$ is called *square* matrix.

**Note:** Square matrices are particularly important.

## 7.2. Matrix diagonals

- Main diagonal
- Secondary diagonal

## 7.3. Diagonal matrix

**Definition 29.** (Diagonal matrix) A diagonal matrix has non zero elements only on its main diagonal.

$$\mathbf{A} = \begin{bmatrix} a_1 1 & 0 & \dots & 0 \\ 0 & a_2 2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & a_n n \end{bmatrix}$$

## 7.4. Identity matrix

**Definition 30.** (Identity matrix) An identity matrix is a diagonal square matrix with all elements equal to one.

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 1 \end{bmatrix}$$

## 7.5. Null matrix

**Definition 31.** (Null matrix) The null, or Zero, matrix is a matrix whos elements are all zeros.

$$\mathbf{0} = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix}$$

## 7.6. Determinant

- 2x2
- 3x3
- nxn

## 7.7. Rank of a matrix

## 7.8. Trace of a matrix

UNIT G-8

# Matrix inversions

Assigned to: Dzenan

## 8.1. Adjugate matrix

$\mathbf{A}\mathrm{adj}(\mathbf{A}) = \det(A)\mathbf{I}$

## 8.2. Matrix Inverse

Square matrix:

$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$

## 8.3. Nonsingularity of a matrix

Exercise: Calculate a (square) Matrix Inverse

Exercise: Inverting a well-conditioned matrix (practice)

Exercise: Inverting an ill-conditioned matrix (practice)

# UNIT G-9
# Matrices and vectors

Assigned to: Dzenan

- matrix-vector product

## 9.1. Matrix as representation of linear (vector) spaces

- linear system to matrix representation
- linearly dependent and independent spaces

### 1) Fundamental spaces

- Null space
- Range/image

### 2) Eigenvalues and Eigenvectors

- for square matrices
- for rectangular matrices (topic for advanced-linear-algebra?)
- condition number of a matrix (?)

UNIT G-10

# Matrix operations (basic)

**Assigned to:** Dzenan

- sum of matrices
- product of matrices
- matrix transpose – Symmetric matrix {#mat-sym}
- matrix concatenation

## 1) Properties

# UNIT G-11
# Matrix operations (complex)

Assigned to: Dzenan

- matrix scalar product
- matrix Hadamart product
- matrix power
- matrix exponential

## 1) Properties

UNIT G-12

# Matrix diagonalization

Assigned to: Dzenan

- singular value decomposition SVD (topic for advanced-linear-algebra?)

## 1) Preferred spaces (matrix diagonalization)

- show how to diagonalize matrices and why it is relevant (it will come in handy for state space representation chapter chapter)

## 12.1. Left and Right Inverse (topic for advanced-linear-algebra?)

- what if the matrix is not square? (topic for advanced-linear-algebra?)
- Moore-Penrose pseudo-inverse

*Example: Find eigenvalues and eigenvectors:*

*Example: Find range and null spaces of a matrix:*

# Unit G-13
# Norms

Assigned to: Dzenan

TODO: finish writing

Other metrics can be defined to measure the "length" of a vector. Here, we report some commonly used norms. For a more in depth discussion of what constitutes a norm, and their properties, see:

➡ [18].

## 1) $p$-norm

Let $p \geq 1 \in \mathbb{R}$. The $p$-norm is defined as:

**Definition 32.** ($p$-norm)

$$\|\mathbf{v}\|_p = \left(\sum_{i=1}^{n} |v_i|^p\right)^{\frac{1}{p}}. \tag{1}$$

The $p$-norm is a generalization of the $2$-norm ($p = 2$ in (???)) introduced above (Definition 19 - Length of a vector). The following $1$-norm and $\infty$-norm can as well be obtained from (1) with $p = 1$ and $p \to \infty$ respectively.

## 2) One norm

The $1$-norm is the sum of the absolute values of a vector's components. It is sometimes referred to as the *Taxicab norm*, or *Manhattan distance* as it well describes the distance a cab has to travel to get from a zero starting point to a final destination $v_i$ on a grid.

**Definition 33.** ($1$-norm) Given a vector $\mathbf{v} \in \mathbb{R}^n$, the $1$-norm is defined as:

$$\|\mathbf{v}\| = \sum_{i=1}^{n} |v_i|. \tag{2}$$

## 3) $\infty$-norm

The infinity norm measures the maximum component, in absolute value, of a vector.

**Definition 34.** ($\infty$-norm)

$$\|\mathbf{v}\| = \max(|v_1|, \cdots, |v_n|). \tag{3}$$

## 13.1. Definition

## 13.2. Properties

# UNIT G-14
# From ODEs to LTI systems

Assigned to: Miguel

## 14.1. LTI Systems

### 1) Properties of LTI systems

UNIT G-15

# Linearization

Assigned to: Miguel

## 15.1. Linearization of a nonlinear system

# UNIT G-16
# Probability basics

In this chapter we give a brief review of some basic probabilistic concepts. For a more in-depth treatment of the subject we refer the interested reader to a textbook such as [19].

## 16.1. Random Variables

The key underlying concept in probabilistic theory is that of an *event*, which is the output of a random trial. Examples of an event include the result of a coin flip turning up HEADS or the result of rolling a die turning up the number "4".

**Definition 35.** (Random Variable) A (either discrete or continuous) variable that can take on any value that corresponds to the feasible output of a random trial.

For example, we could model the event of flipping a fair coin with the random variable $X$. We write the probability that $X$ takes HEADS as $p(X = \text{HEADS})$. The set of all possible values for the variable $X$ is its *domain*, $\mathcal{X}$. In this case,

$$\mathcal{X} = \{\text{HEADS}, \text{TAILS}\}.$$

Since $X$ can only take one of two values, it is a *binary* random variable. In the case of a die roll,

$$\mathcal{X} = \{1, 2, 3, 4, 5, 6\},$$

and we refer to this as a *discrete* random variable. If the output is real value or a subset of the real numbers, e.g., $\mathcal{X} = \mathbb{R}$, then we refer to $X$ as a *continuous* random variable.

Consider once again the coin tossing event. If the coin is fair, the have $p(X = HEADS) = p(X = TAILS) = 0.5$. Here, the function $p(x)$ is called the *probability mass function* or pmf. The pmf is shown in Figure 16.1.

Figure 16.1. The pmf for a fair coin toss

Here are some very important properties of $p(x)$: - $0 \leq p(x) \leq (1)$ - $\sum_{x \in \mathcal{X}} = 1$

In the case of a continuous random variable, we will call this function $f(x)$ and call it a *probability density function*, or pdf.

In the case of continuous RVs, technically the $p(X = x)$ for any value $x$ is zero since $\mathcal{X}$ is infinite. To deal with this, we also define another important function, the *cumulative density function*, which is given by $F(x) \triangleq p(X \leq x)$, and now we can define $f(x) \triangleq \frac{d}{dx}F(x)$. A pdf and corresponding cdf are shown in Figure 16.2 (This happens to be a Gaussian distribution, defined more precisely in Subsection 16.1.8 - The Gaussian Distribution.).

Figure 16.2. The continuous pdf and cdf

### 1) Joint Probabilities

If we have two different RVs representing two different events $X$ and $Y$, then we represent the probability of two distinct events $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ both happening, which we will denote as following: $p(X = x \text{ AND } Y = y) = p(x, y)$. The function $p(x, y)$ is called *joint distribution*.

### 2) Conditional Probabilities

Again, considering that we have to RVs, $X$ and $Y$, imagine these two events are linked in some way. For example, $X$ is the numerical output of a die roll and $Y$ is the binary even-odd output of the same die roll. Clearly these two events are linked since they are both uniquely determined by the same underlying event (the rolling of the die). In this case, we say that the RVs are *dependent* on one another. In the event that we know one of events, this gives us some information about the other. We denote this using the following *conditional distribution* $p(X = x \text{ GIVEN } Y = y) \triangleq p(x|y)$.

> **Check before you continue**
>
> Write down the conditional pmf for the scenario just described assuming an oracle tells you that the die roll is even. In other words, what is p(x|\text{EVEN})?
>
> (Warning: if you think this is very easy that's good, but don't get over-confident.)

The joint and conditional distributions are related by the following (which could be

considered a definition of the joint distribution):

$$p(x, y) = p(x|y)p(y) \tag{1}$$

and similarly, the following could be considered a definition of the conditional distribution:

$$p(x|y) = \frac{p(x, y)}{p(y)} \text{ if } p(y) > 0 \tag{2}$$

In other words, the conditional and joint distributions are inextricably linked (you can't really talk about one without the other).

If two variables are *independent*, then the following relation holds: $p(x, y) = p(x)p(y)$ .

### 3) Bayes' Rule

Upon closer inspection of (1), we can see that the choice of which variable to condition upon is completely arbitrary. We can write:

$$p(y|x)p(x) = p(x, y) = p(x|y)p(y)$$

and then after rearranging things we arrive at one of the most important formulas for mobile robotics, Bayes' rule:

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} \tag{3}$$

Exactly why this formula is so important will be covered in more detail in later sections (TODO), but we will give an initial intuition here. TODO

Consider that the variable $X$ represents something that we are trying to estimate but cannot observe directly, and that the variable $Y$ represents a physical measurement that relates to $X$. We want to estimate the distribution over $X$ given the measurement $Y$, $p(x|y)$, which is called the *posterior* distribution. Bayes' rule lets us to do this. For every possible state, you take the probability that this measurement could have been generated, $p(y|x)$, which is called the *measurement likelihood*, you multiply it by the probability of that state being the true state, $p(x)$, which is called the *prior*, and you normalize over the probability of obtaining that measurement from any state, $p(y)$, which is called the *evidence*.

**Check before you continue**

From Wikipedia: Suppose a drug test has a 99% true positive rate and a 99% true negative rate, and that we know that exactly 0.5% of people are using the drug. Given that a person's test gives a positive result, what is the probability that this person is actually a user of the drug.

Answer: ≈ 33.2%. This answer should surprise you. It highlights the power of the *prior*.

### 4) Marginal Distribution

If we already have a joint distribution $p(x, y)$ and we wish to recover the single variable distribution $p(x)$, we must *marginalize* over the variable $Y$. The involves summing (for discrete RVs) or integrating (for continuous RVs) over all values of the variable we wish to marginalize:

$$p(x) = \sum_{y} p(x, y) f(x) \qquad = \int p(x, y) dy$$

This can be thought of as projecting a higher dimensional distribution onto a lower dimensional subspace. For example, consider Figure 16.3, which shows some data plotted on a 2D scatter plot, and then the marginal histogram plots along each dimension of the data.

Figure 16.3. A 2D joint data and 2 marginal 1D histogram plots

Marginalization is an important operation since it allows us to reduce the size of our state space in a principled way.

## 5) Conditional Independence

Two RVs, $X$ and $Y$ may be correlated, we may be able to encapsulate the dependence through a third random variable $Z$. Therefore, if we know $Z$

Figure 16.4. A graphical representation of the conditional independence of $X$ and $Y$ given $Z$

+ comment

Is there a discussion of graphical models anywhere? Doing a good job of sufficiently describing graphical models and the dependency relations that they express requires careful thought. Without it, we should refer readers to a graphical models text (e.g., Koller and Friedman, even if it is dense)

## 6) Moments

The $n$th moment of an RV, $X$, is given by $E[X^n]$ where $E[]$ is the expection operator with:

$$E[f(X)] = \sum_{\mathcal{X}} x\, f(x)$$

in the discrete case and

$$E[f(X)] = \int x\, f(x) dx$$

in the continuous case.

The 1st moment is the *mean*, $\mu_X = E[X]$.

The $n$th central moment of an RV, $X$ is given by $E[(X - \mu_X)^n]$. The second central moment is called the *covariance*, $\sigma_X^2 = E[(X - \mu_X)^2]$.

## 7) Entropy

**Definition 36.** The *entropy* of an RV is a scalar measure of the uncertainty about the value the RV.

A common measure of entropy is the *Shannon entropy*, whose value is given by

$$H(X) = -E[\log_2 p(x)] \tag{4}$$

This measure originates from communication theory and literally represents how many bits are required to transmit a distribution through a communication channel. For many more details related to information theory we recommend [20].

As an example, we can easily write out the Shannon entropy associated with a binary RV (e.g. flipping a coin) as a function of the probability that the coin turns up heads (call this $p$):

$$H(X) = -p \log_2 p - (1 - p) \log_2 (1 - p) \tag{5}$$

Figure 16.5. The Shannon entropy of a binary RV $X$

Notice that our highest entropy (uncertainty) about the outcome of the coin flip is when it is a fair coin (equal probability of heads and tails). The entropy decays to 0 as we approach $p = 0$ and $p = 1$ since in these two cases we have no uncertainty about the outcome of the flip. It should also be clear why the function is symmetrical around the $p = 0.5$ value.

## 8) The Gaussian Distribution

In mobile robotics we use the Gaussian, or normal, distribution a lot.

+ comment

The banana distribution is the official distribution in robotics! - AC

+ comment

The banana distribution is Gaussian! http://www.roboticsproceedings.org/rss08/p34.pdf - LP

The 1-D Gaussian distribution pdf is given by:

$$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2} \tag{6}$$

where $\mu$ is called the *mean* of the distribution, and $\sigma$ is called the *standard deviation*. A plot of the 1D Gaussian was previously shown in Figure 16.2.

We will rarely deal with the univariate case and much more often deal with the multi-variate Gaussian:

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2 * \pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp[-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu})] \tag{7}$$

The value from the exponent: $(\boldsymbol{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu})$ is sometimes written $||\boldsymbol{x} - \boldsymbol{\mu}||_{\boldsymbol{\Sigma}}$ and is referred to as the *Mahalanobis distance* or *energy norm*.

Mathematically, the Gaussian distribution has some nice properties as we will see. But is this the only reason to use this as a distribution. In other words, is the assumption of Gaussianicity a good one?

There are two very good reasons to think that the Gaussian distribution is the "right" one to use in a given situation.

1. The *central limit theorem* says that, in the limit, if we sum an increasing number of independent random variables, the distribution approaches Gaussian
2. It can be proven (TODO:ref) that the Gaussian distribution has the maximum entropy subject to a given value for the first and second moments. In other words, for a given mean and variance, it makes the *least* assumptions about the other moments.

Exercise: derive the formula for Gaussian entropy

UNIT G-17

# Kinematics

**Assigned to:** Harshit

Kinematics is the study of *position*, *velocity* and *acceleration* of geometric points.

**Note:** A point has no dimensions. For example, the center of mass of a body is a point.

# Unit G-18
# Dynamics

Assigned to: Harshit

UNIT G-19

# Coordinate systems

## 19.1. Motivation

In order to uniquely specify a position in some space, we use some numbers, *coordinates*, in a *coordinate system*. For example, in daily life, we would use the intersection of two streets, each with a unique name in the city, to specify the location of some cafe. If you find yourself in the ocean, you might communicate your location to someone by telling them the latitude and longitude readout on your GPS device. Generally speaking, a coordinate system provides us a way to denote *any* position in an *unambiguous* way. So you can communicate any position to another person, and by using the given coordinates, that person can arrive at the same exact position in space. Note, however, different coordinates can correspond to the same point.

## 19.2. Definitions

**Definition 37.** (Coordinate system) A coordinate system is a surjective function mapping from a tuple of reals to some space $S$, respecting the local topology. (The local topology, which is beyond the scope of this chapter, roughly means that "nearby" points have coordinates "close" together.)

## 19.3. Examples

Because the ability of *naming* or specifying a point in space is so fundamentally important, we often take that coordinates given by some coordinate system, often a Cartesian coordinate system, as the name of the point.

### 1) 1D

Consider the real number line $\mathbb{R}$ as the space $S$. We can name an arbitrary point $x \in \mathbb{R}$, which happens to be a real number, by itself. So to check with our definition, any two distinct points on the real line would have two distinct coordinates. Furthermore any point has a coordinate. Let's call this coordinate system $A$.

One should note that the coordinate system given above is not the only possible way to name the points on a real line. We can assign coordinate $-x$ to the point $x \in \mathbb{R}$ and obtain an equally valid coordinate system $B$. To be more specific, given a point $p$ with the coordinate $a$ in the first coordinate system $A$, we know that in the second coordinate system $B$, $p$ would have the coordinate $-a$. Therefore, there is a way to translate between coordinates in the two systems.

### 2) 2D plane

Consider the real plane $\mathbb{R}^2$ as the space $S$. This is an important space within robotics and beyond. It can be used to represent images, with each pixel having its own position, or the location of your Duckiebot in Duckietown.

*2D Cartesian coordinate systems:*

We can draw two perpendicular lines on the plane $S$. We call these two lines the $x$

-axis and the $y$-axis. We assign $(0,0)$ to the point of intersection, which we call *the origin*. Then we decide on the positive directions and units for each axis and the unit. We will use coordinates $(x,y)$ to specify a point located $x$-many units in the positive $x$-direction and $y$-many units in the positive $y$-direction away from the origin, respectively. If we draw the axis-parallel lines with *integral* $x$ coordinate and lines with integral $y$ coordinate, we obtain a visualization similar to that in Figure 19.1.

When representing the location of your Duckiebot in Duckietown, you might decide to choose a corner of the map as the origin and take east as the positive $x$-direction and north as the positive $y$-direction, and 1 meter as the unit length. In this case, a Duckiebot with location $(1,-2)$ would sit at 1 meter east and 2 meters south of the designated corner of the map.

Figure 19.1. A Cartesian coordinate system in the 2D plane

**Remark 8.** (Image space) It is customary to put the origin of an image at the top-left corner with the $x$-axis being horizontal and increasing to the right and the $y$-axis vertical and increasing downwards. In this way, the $x$ and $y$ coordinates index the column and row, respectively, of a particular pixel in the image. Such a convention is observed in `OpenCV` and other software libraries.

*Polar coordinate systems:*

An alternative coordinate system for the plane is the polar coordinate system, where we specify a point by its direction and distance from a fixed reference point. To set up a polar coordinate system, you first decide on the *pole*, the reference point, then the *polar axis*, the reference direction. We will call the distance from the pole, the *radial coordinate* or *radius*, commonly denoted by $r$ or $\rho$, and the angle from the polar axis, the *angular coordinate* or *polar angle*, commonly denoted by $\phi$, $\varphi$ or $\theta$. See an example in Figure 19.2.

Figure 19.2. A polar coordinate system with pole O and polar axis L.

Note that in a polar coordinate system, a point has many equally valid names.

> **Check before you continue**
>
> Exercise to readers: provide two such points and a few of their coordinates each.

Now, consider a Cartesian coordinate system $C$ whose origin is at the pole and its positive $x$-direction coincides with the polar axis. It is not hard to convert polar coordinates to Cartesian coordinates in $C$.

> **Check before you continue**
>
> Exercise to readers: consult Figure 19.3 and write out the conversion formulae.)

Figure 19.3. Converting from polar coordinates to Cartesian coordinates.

Given the many options, you might wonder which coordinate system to use in any given situation. The answer is a practical one. Choose the one that helps simplify the problem at hand. As a trivial example, consider the equation for a unit circle. In a Cartesian coordinate system, it would be $x^2 + y^2 = 1$ whereas in a polar coordinate system, it would be much simpler: $r = 1$.

**Check before you continue**

Exercise to readers: how about the equation for a straight line in polar coordinates?

### 3) 3D space

This is an important space since we live in a three-dimensional world. Since many of our robots operate in this same world, many robots similarly represent coordinates in three dimensions, including unmanned aerial vehicles (UAVs) and autonomous underwater vehicles (AUVs).

*3D Cartesian coordinate systems:*

Suppose the plane is the page or screen you are reading from, which is just a slice through the 3D space around it, we can extend a 2D Cartesian coordinate system on the plane to 3D by adding a $z$-axis that is perpendicular to the page, i.e., the $z$-, $x$-, and $y$-axis are mutually perpendicular. As done before, we need to choose a positive $z$-direction and there are two choices: coming out of the page or going into the page. They form the right-handed coordinate system or the left-handed coordinate system, respectively. We shall use right-handed coordinate systems unless otherwise noted. For more on handedness, see Wikipedia. Now the resulting coordinates become $(x, y, z)$, see Figure 19.4.

Figure 19.4. A 3D Cartesian coordinate system.

*Spherical coordinate systems:*

Similarly, we can extend the polar coordinate systems on the page or screen to 3D by defining a *zenith direction* (upwards) perpendicular to the plane, the *polar angle* to be the angle away from zenith, and the *azimuth angle* to be the orthogonal projection of a point's angle away from the polar axis on the plane. Together, a point has coordinates $(r, \theta, \phi)$ where $\theta$ denotes the polar angle and $\phi$, the azimuth angle. See Figure 19.5.

**Check before you continue**

Exercise to readers: how to convert spherical coordinates to 3D Cartesian coordinates? and back?

Figure 19.5. A spherical coordinate system.

## 19.4. Further reading

If you find this topic interesting, there are many more coordinate systems than the ones covered here, such as the:

- cylindrical coordinate system in 3D space and
- parabolic coordinate system in 2D space.

**Author:** Falcon Dai

**Maintainer:** Falcon Dai

# UNIT G-20
# Reference frames

**Check before you continue**

Required Reading: The following assumes a working familiarity with 2D and 3D Cartesian coordinate systems. If you are not familiar with Cartesian coordinate systems, please read the chapter on coordinate systems.

## 20.1. Motivation

Does Earth move around the sun or does the sun move around Earth? It turns out that this is an ill-posed question until we specify a *frame of reference* for measurement. For us, observers on Earth, it appears that the sun is moving. But for an observer on the moon, both Earth and the sun are moving. In general, motions are relative and we need a reference when measuring motion.

## 20.2. Definition

A *reference frame*, or just *frame*, is an abstract coordinate system and the set of physical reference points that uniquely specify the coordinate system (location of the origin and orientation). As a way of specifying the reference frame, we often say object $\boldsymbol{A}$ is moving *relative to* object $\boldsymbol{B}$, instead of specifying explicitly a reference frame $\boldsymbol{F_B}$ that is attached to object $\boldsymbol{B}$.

## 20.3. Example

Suppose there are two cars A and B both moving at 60 miles per hour eastward. When saying this, we implicitly assume the reference frame of the ground. In the ground's reference frame, the ground itself is at rest, car A and car B are both moving at 60 miles per hour eastward. In car A's reference frame, however, car B is at rest and the ground is *moving* 60 miles per hour westward!

## 20.4. Translating motions in one frame to another

To simplify the following discussion, we additionally assume that we choose reference frames such that their axes are parallel. In order to translate motions between reference frames, we assume two rules.

- Relative motions are equal in magnitude and opposite in direction. If frame $\boldsymbol{R}$ is moving relative to frame $\boldsymbol{S}$ at $\mathbf{u}$, then frame $\boldsymbol{S}$ is moving at $-\mathbf{u}$ relative to $\boldsymbol{R}$.
- Motions are additive. If a frame $\boldsymbol{T}$ is moving at $\mathbf{u}$ relative to frame $\boldsymbol{S}$, and frame $\boldsymbol{S}$ is moving at $\mathbf{v}$ relative to frame $\boldsymbol{R}$, then frame $\boldsymbol{T}$ is moving at $\mathbf{u}+\mathbf{v}$ relative to frame $\boldsymbol{R}$.

**Check before you continue**

Exercise to readers: derive these rules from kinematics.

As a corollary to the first rule, we immediately derive that frame $\boldsymbol{R}$ is at rest relative to itself, since $\mathbf{0}$ is the only vector is also its own opposite.

**Check before you continue**

Exercise to readers: translate car B's motion relative to the ground to relative to car A.

**Author:** Falcon Dai

**Maintainer:** Falcon Dai

# Unit G-21
# Time series

Knowledge and activity graph

**Requires:** Knowledge of k:tuple

**Results:** Knowledge of time series.

**Results:** Knowledge of time series operations, such as upsampling and down-sampling.

## 21.1. Definition of time series

**Definition 38.** A *time series* with domain $\mathcal{X}$ and time domain $\mathbb{T}$ is a sequence of tuples $\langle t_k, x_k \rangle \in \mathbb{T} \times \mathcal{X}$.

## 21.2. Operations on time series

### 1) Up-sampling

TODO: to write

### 2) Down-sampling

TODO: to write

## 21.3. Further reading

Super-dense time

UNIT G-22

# Transformations

**Check before you continue**

Required Reading: The following assumes a working familiarity with 2D and 3D Cartesian reference frames. If you are not familiar with Cartesian reference frames, please read the chapter on reference frames. Some familiarity with linear alegra is also helpful.

## 22.1. Introduction

*Transformations* are functions that map points to other points in space, i.e. $f : X \to X$. These maps are useful for describing motions over time. A particularly important class of transformations are *linear transformations*. These transformations can be represented by square matrices as they are *linear* and has the same domain and image.

## 22.2. Definitions and important examples

Please refer to the *Robotics Handbook* section 1.2, and in the context of this course, reader's goal is to attain a conceptual understanding, not necessarily knowing the exact formulae.

**TODO:** we need to provide links to the handbook. -AC

## 22.3. Applications

**TODO:** adapt the concepts in the context of Duckietown.

**Author:** Falcon Dai

**Maintainer:** Falcon Dai

# Unit G-23
# Types

Knowledge and activity graph

**Results:** Introduction to type theory.
**Requires:** Knowledge of set theory.

## 23.1. Types vs sets

Types are similar to sets in that they are collections that include some objects and exclude others. On a first glance, you can interpret $\mathbf{0} : \mathbf{nat}$ as $\mathbf{0} \in \mathbb{N}$, reading the colon as membership in sets without any ill effect. But types and sets are the basic concepts belonging to two different formal systems, *type theory* and *set theory*, respectively. It is not essential to appreciate the difference in the scope of this course but for the curious readers, this section on Wikipedia can serve as a brief summary.

## 23.2. Example: product types

Given two types $\boldsymbol{A}$ and $\boldsymbol{B}$, we can construct the type $\boldsymbol{A} \times \boldsymbol{B}$, which we call their *cartesian product*. (Compare this with the similar concept of *cartesian product of sets* which is defined in terms of its elements and not a primitive.)

## 23.3. In Duckietown

As we will be using ROS, which models a robotic system as a network of communicating programs. In order to understand each other, all the communicating programs talk to each other in well-defined *message types*. Message types in ROS are product types composed of primitive types and other message types.

> **Check before you continue**
> Please read section 2 on ROS/msg page and answer: what are some primitive types in ROS? what are the fields and their types in message type `Header`?

Additionally, ROS/common_msgs page provides a list of pre-defined message types commonly used in robotics, such as Image (note how `Header`, a non-primitive type, is included in the definition) and Pose2D. As you have likely guessed, an RGB camera publishes `Image` messages, and a routing planning program might subscribe to the duckiebot's current position in duckietown, as represented in a `Pose2D` message and calculates the appropriate wheel actions.

## 23.4. Historical notes

Historically, the flexibility of naive set theory allows for some paradoxical sets such as a set that contains all sets that does not contain itself. Does this set contains

itself? This is known as Russell's paradox which demonstrated that naive set theory is inconsistent. In response, Russell and colleagues developed type theory which demands all terms to be *typed*, i.e., to have a type, and used a hierarchy of types to avoid Russell's paradox. Later, a subclass of type theories known as intuitionistic type theories internalized many key ideas in constructive mathematics and became a foundation for programming languages where *computability* is a major concern.

On a side note, this is not to say sets cannot serve as a formal foundation of mathematics. Russell's paradox only shows that *naive* set theory is *inconsistent*. In fact, most working mathematicians today believe that the axiomatized Zermelo-Fraenkel set theory (together with the axiom of choice, usually abbreviated as ZFC) can serve as a "consistent" foundation of all mathematics.

## 23.5. Further reading

Type theory is a fascinating subject in itself and recently, Homotopy Type Theory (HoTT) captured a lot of research interest. For more on the subject consult HoTT website. The first chapter of the HoTT book also provides a reasonable introduction to type theory. For more practical applications of these abstract ideas, you may be intrigued by the field of formal verification, where software is verified by mathematical proofs against the formal specification, automatically.

**Author:** Falcon Dai

**Maintainer:** Falcon Dai

# Unit G-24
# Computer science concepts

This unit will contain brief explanations / pointers to basic concepts of computer science.

## 24.1. Real-time Operating system

TODO: to write

# Part H
# A course in autonomy

These are the theory units.

# Unit H-1
# Autonomous Vehicles

Assigned to: Liam

## 1.1. Autonomous Vehicles in the News

These days it is hard to separate the fact from the fiction when it comes to autonomous vehicles, particularly self-driving cars. Virtually every major car manufacturer has pledged to deploy some form of self-driving technology in the next five years. In addition, there are many startups and software companies which are also known to be developing self-driving car technology.

Here's a non-exhaustive list of some of companies that are actively developing autonomous cars:

- Waymo (Alphabet/Google group)
- Tesla Autopilot project
- Uber Advanced Technologies Group
- [Cruise Automation] (the recent developments)
- nuTonomy
- Toyota Research Institute (Broader than just autonomous cars)
- Aurora Innovation
- Zoox
- Audi
- Nissan's autonomous car
- Baidu
- Apple "Project Titan" (no official details released)
- Drive.ai

## 1.2. Levels of Autonomy

Before even discussing any detailed notion of autonomy, we have to specify exactly what we are talking about. In the United States, the governing body is the NHTSA, and they have recently (Oct 2016) redefined the so-called "levels of autonomy" for self-driving vehicles.

In broad terms, they are as follows

- **Level 0**: the human driver does everything;
- **Level 1**: an automated system on the vehicle can sometimes assist the human driver conduct some parts of the driving task;
- **Level 2**: an automated system on the vehicle can actually conduct some parts of the driving task, while the human continues to monitor the driving environment and performs the rest of the driving task;
- **Level 3**: an automated system can both actually conduct some parts of the driving task and monitor the driving environment in some instances, but the human driver must be ready to take back control when the automated system requests;
- **Level 4**: an automated system can conduct the driving task and monitor the dri-

ving environment, and the human need not take back control, but the automated system can operate only in certain environments and under certain conditions; and

- **Level 5**: the automated system can perform all driving tasks, under all conditions that a human driver could perform them.

# Unit H-2
# Autonomy overview

Assigned to: Liam

This unit introduces some basic concepts ubiquitous in autonomous vehicle navigation.

## 2.1. Basic Building Blocks of Autonomy

The minimal basic backbone processing pipeline for autonomous vehicle navigation is shown in Figure 2.1.

Figure 2.1. The basic building blocks of any autonomous vehicle

For an autonomous vehicle to function, it must achieve some level of performance for all of these components. The level of performance required depends on the *task* and the *required performance*. In the remainder of this section, we will discuss some of the most basic options. In the next section we will briefly introduce some of the more advanced options that are used in state-of-the-art autonomous vehicles.

### 1) Sensors

(a) Velodyne 3D Laser Scanner (b) Camera (c) Automotive Radar

(d) GPS Receiver (e) Inertial Measurement Unit

Figure 2.2. Some common sensors used for autonomous navigation

**Definition 39.** (Sensor) A *sensor* is a device that or mechanism that is capable of generating a measurement of some external phyiscal quantity

In general, sensors have two major types. *Passive* sensors generate measurements without affecting the environment that they are measuring. Examples include inertial sensors, odometers, GPS receivers, and cameras. *Active* sensors emit some form of energy into the environment in order to make a measurement. Examples of this type of sensor include Light Detection And Ranging (LiDAR), Radio Detection And Ranging (RaDAR), and Sound Navigation and Ranging (SoNAR). All of these sensors emit energy (from different spectra) into the environment and then detect some property of the energy that is reflected from the environment (e.g., the time of flight or the phase shift of the signal)

### 2) Raw Data Processing

The raw data that is input from a sensor needs to be processed in order to become useful and even understandandable to a human.

First, **calibration** is usually required to convert units, for example from a voltage to a physical quantity. As a simple example consider a thermometer, which measures temperature via an expanding liquid (usually mercury). The calibration is the known mapping from amount of expansion of liquid to temperature. In this case it is a linear mapping and is used to put the markings on the thermometer that make it useful as a sensor.

We will distiguish between two fundamentally types of calibrations.

**Definition 40.** (Intrinsic Calibration) An *intrinsic calibration* is required to determine sensor-specific paramaters that are internal to a specific sensor.

**Definition 41.** (Extrinsic Calibration) An *extrinsic calibration* is required to determine the external configuration of the sensor with respect to some reference frame.

**Check before you continue**

For more information about reference frames check out Unit G-20 - Reference frames

Calibration is very important consideration in robotics. In the field, the most advanced algorithms will fail if sensors are not properly calibrated.

Once we have properly calibrated data in some meaningful units, we often do some preprocessing to reduce the overall size of the data. This is true particularly for sensors that generate a lot of data, like cameras. Rather than deal with every pixel value generated by the camera, we will process an image to generate feature-points of interest. In "classical" computer vision many different feature descriptors have been proposed (Harris, BRIEF, BRISK, SURF, SIFT, etc), and more recently Convolutional Neural Networks (CNNs) are being used to learn these features.

The important property of these features is that they should be as easily to associate as possible across frames. In order to achieve this, the feature descriptors should be invariant to nuissance parameters.

Figure 2.3. Top: A raw image with feature points indicated. Bottom: Lines projected onto ground plane using extrinsic calibration and ground projections

### 3) State Estimation

Now that we have used our sensors to generate a set of meaningful measurements, we need to combine these measurements together to produce an estimate of the underlying hidden *state* of the robot and possibly to environment.

**Definition 42.** (State) The state $\boldsymbol{x}_t \in \mathcal{X}$ is a *sufficient statistic* of the environment, i.e. it contains all sufficient information required for the robot to carry out its task in that environment. This can (and usually does) include the *configuration* of the robot itself.

What variables are maintained in the statespace $\mathcal{X}$ depends on the problem at hand. For example we may just be interested in a single robot's configuration in the plane, in which case $\boldsymbol{x}_t \equiv \boldsymbol{q}_t$. However, in other cases, such as simultaneous localization and mapping, me may also be tracking the map in the state space.

According to Bayesian principles, any system parameters that are not fully known and deterministic should be maintained in the state space.

In general, we do not have direct access to values in $\boldsymbol{x}$, instead we rely on our (noisy) sensor measurements to tell us something about them, and then we *infer* the values.

Figure 2.4. Lane Following in Duckietown. *Top Right*: Raw image; *Bottom Right*: Line detections; *Top Left*: Line projections and estimate of robot position within the lane (green arrow); *Bottom Left*: Control signals sent to wheels.

The animation in Figure 2.4 shows the lane following procedure. The output of the state estimator produces the **green arrow** in the top left pane.

## 4) Navigation and Planning

Figure 2.5. An example of nested control loops

In general we decompose the task of controlling an autonomous vehicle into a series of **nested control loops**.

The loops are called nested since the output of the outer loop is used as the reference input to the inner loop. An example is shown in Figure 2.5.

> **Recommended:** If Figure 2.5 is **VERY** mysterious to you, then you may want to have a quick look in a basic feedback control textbook. For example [21] or [22].

In this case we show three loops. At the outer loop, some goal state is provided. The actual state of the robot is used as the feedback. The controller is the block labeled `Navigation and Motion Planning`. The job of this block is generate a **feasible path** from the current state to the goal state. This is executed in **configuration space** rather than the state space (although these two spaces may happen to be the same they are fundamentally conceptually different.

Figure 2.6. Navigation in Duckietown

### 5) Control

The next inner loop of the nested controller in Figure 2.5 is the `Vehicle Controller`, which takes as input the reference trajectory generated by the `Navigation and Motion Planning` block and the current configuration of the robot, and uses the error between the two to generate a control signal.

The most basic feedback control law (See Unit H-27 - Feedback control is called PID (for proportional, integral, derivative) which will be discussed in Unit H-28 - PID Control. For an excellent introduction to this control policy see Figure 2.7.

Figure 2.7. Controlling Self Driving Cars

We will also investigate some more advanced non-linear control policies such as Model Predictive Control, which is an optimization based technique.

### 6) Actuation

The very innermost control loop deals with actually tracking the correct voltage to be sent to the motors. This is generally executed as close to the hardware level as possible. For example we have a `Stepper Motor HAT` See the parts list.

### 7) Infrastructure and Prior Information

In general, we can make the autonomous navigation a simpler one by exploiting existing structure, infastructure, and contextual prior knowledge.

Infrastructure example: Maps or GPS satellites

Structure example: Known color and location of lane markings

Contextual prior knowledge example: Cars tend to follow the *Rules of the Road*

## 2.2. Advanced Building Blocks of Autonomy

The basic building blocks enable static navigation in Duckietown. However, many other components are necessary for more realistic scenarios.

### 1) Object Detection

Figure 2.8. Advanced Autonomy: Object Detection

One key requirement is the ability to detect objects in the world such as but not limited to: signs, other robots, people, etc.

### 2) SLAM

The simultaneous localization and mapping (SLAM) problem involves simultaneously estimating not only the robot state but also the **map** at the same time, and is a fundamental capability for mobile robotics. In autonomous driving, generally the most common application for SLAM is actual in the map-building task. Once a map is built then it can be pre-loaded and then used for pure localization. A demonstration of this in Duckietown is shown in Figure 2.9.

Figure 2.9. Localization in Duckietown

### 3) Other Advanced Topics

Other topics that will be covered include:

- Visual-inertial navigation (VINS)
- Fleet management and coordination
- Scene segmentation
- Deep perception
- Text recognition

UNIT H-3

# Modern Robotic Systems

**Assigned to:** Andrea Censi

KNOWLEDGE AND ACTIVITY GRAPH

**Results:** The many parts of a robotic system

**Results:** Modern robotics development

**Results:** Example of cloud learning, annotators in the loop scenario.

## 3.1. No robot is an island

A robot is only a small part of a modern robotic system.

We briefly discuss what we could consider the possible components of a robotic system.

Of course, **the robot**, which includes:

- The hardware: actuation, sensing, computation;
- The software;

**The other robots** with which the robot interacts. Perhaps they just need to avoid each other; perhaps they are collaborating.

The **other machines**. For example, a battery dock with that the robot can use to recharge

The **infrastructure**, including the network and off-board storage and computation resources.

The **people**, including:

- The supervisors;
- The safety operator;
- The customers;
- ...

## 3.2. Development of modern robotic systems

While robotic systems have existed for decades, modern autonomous robotic systems are developed in a different way than traditional robotics/automation projects.

The classical model of development consists of:

- design;
- product development;
- integration (by “system integrators”);
- installation;
- support.

These are instead the characteristics of modern robotics development model:

- continuous feedback from the users

- learning from data, acquired by the robot
- continuous integration
- incremental updates
- agile development

## 3.3. Example of a typical data processing pipeline

Variations on the following idea are what is typically implemented by autonomous vehicles developers for object detection.

The problem is to implement a machine-learning system that learns to perform an object detection task based on supervised learning.

Training and validation happen on large datasets that are continuously updated with new data coming from the cars, and new annotations coming from annotators.

### 1) The cloud as a component

We can consider "the cloud" as a component of a modern robotic system. The cloud can be modeled as a component that provides:

- Essentially "infinite" computation;
- Essentially "infinite" storage;
- Very large latency.

Because of the latency, it is not possible for real-time robotics applications to run completely on the cloud.

Therefore, the cloud is much more useful for tasks like the following:

- running simulations;
- training machine learning models;
- running regression tests.

As of 2017, the largest cloud-computing services are the ones offered by Amazon (AWS), Microsoft (Azure), Google (Google Cloud).

### 2) The annotators as components

For supervised learning tasks, one needs to have large annotations databases.

The idea of using human annotators as a "software service" was first deployed on a large scale by Amazon with the "Mechanical Turk" project.

Nowadays, there exist companies that are specialized in providing annotations for AI tasks.

Examples of annotation services are:

- MightyAI: "Training Data as a Service"
- Samasource

### 3) Putting it all together in feedback

Now that we have introduced the components, we will see how one can put everything together in a system (Figure 3.1).

Figure 3.1. Example of modern data-based pipeline

This system works as follows:

1. The robots collect data during normal operation.
2. The data becomes part of a large cloud-base storage.
3. The data is divided in training and validation data.
4. Continuously, a new model is trained based on the latest data.
5. The new candidate model is evaluated using regression tests; the goal is to outperform the previous model.
6. The new models are broadcast to the robots.
7. The regression tests also look for which new data is most useful to annotate, and this information is passed to the annotators, who create more annotations.
8. The regression tests also look for which new data would be interesting to collect, and this is done by a dedicated car.

## 3.4. Vignettes from an optimistic future

In the near future, it might be that the design of robotic systems might become even more complicated. For example, it might be that blockchain technologies will allow machines to trade between them.

*example*

A self-driving car realizes it is too dirty; by itself, it finds a robotic car-wash, and together they agree on the time and the price, and the car pays by itself using Bitcoin.

## 3.5. Take-away points

- The robot is but a small part of a robotic system.
- Development methods have changed recently: data is very important, as well as delocalized computation.

## 3.6. Further reading

- One of the cloud robotics papers XXX
- Mechanical Turk XXX
- The Robot Design Game
- A Blockchain tutorial XXX

UNIT H-4

# System architectures basics

Assigned to: Andrea Censi

KNOWLEDGE AND ACTIVITY GRAPH

Results: Physical and logical architectures.

Results: Deployment as mapping a physical architecture onto the logical.

Requires: Basic graph concepts.

Requires: Basic operating systems concepts.

## 4.1. Logical and physical architecture

When we design a robotic system, or any cyber-physical system, we distinguish between what we shall call "logical architecture" and the "physical architecture".

The logical architecture describes how the functionality is divided in abstract modules, and what these modules communicate.

The physical architecture describes how the modules are instantiated. For example, this includes the information of how many processors are used, and which processor runs which routine.

## 4.2. Logical architecture

The logical architecture is independent of the implementation (the hardware, the language.)

The logical architecture describes:

- The system decomposition in components
- The data flow: who tells whom what
- How knows what: other sources of information, such as priors.
- How information is represented

A logical architecture would also describe what are the representations used. This is explored more fully in the unit about representations.

## 4.3. Actor model

TODO: Brief discussion of actor model

## 4.4. Physical architecture

TODO: Processors

TODO: Buses / networks

**TODO:** Middleware

**TODO:** Orchestrators

**TODO:** In ROS that is the `roscore` program.

## 4.5. Mapping logical architecture onto physical

For deployment, one must choose how the logical architecture is mapped on the physical architecture.

This can be seen as a graph mapping problem.

One can define a computation graph as a graph where nodes are algorithms and edges are events.

A resource graph is a graph where nodes are processors and edges are communication channels.

Given a computation graph and a resource graph one must choose where to put each node in the computation graph in the resource graph.

> **Remark:** More formally, the assignment is called a graph homomorphism.

## 4.6. Example in Duckietown

You will see a concrete example of different ways to map software components on logical architectures in one of the first exercises.

Duckietown uses ROS. In ROS, the components are called *nodes*. In ROS, the granularity is at the level of hosts rather than processors. In regular vanilla Linux, the kernel decides which physical processor executes which process at any time.

In ROS, the assignment of nodes to processors happens using a *launch file*.

By modifying the launch file we can choose the layout of the computation.

Typically you will encounter three ways to deploy a graph:

1. **Running everything on the robot.** This is the regular "autonomous" mode.
2. **Running everything on the robot, but orchestrating from the laptop**. In this case, the `roscore` program runs on a laptop, and the other components on the robot.
3. **Running heavy computation on the laptop**. In this mode, the heavy computation processes run on the laptop, while the actuation and sensing drivers run on the robot.

## 4.7. Take-away points

- The logical architecture describes the system decomposition, independent of the implementation.
- The physical architecture describes how is the computation physically realized.
- There are multiple ways to map a *computation graph* onto a *resource graph*. This is something that is immediately useful to understand for rapid development.

## 4.8. Further reading

- E. A. Lee and S. A. Seshia, Embedded Systems – A Cyber-Physical Systems Ap-

proach, MIT Press, Second Edition, 2017.

# Unit H-5
# Autonomy architectures

Assigned to: Andrea Censi

## 5.1. The state of the art in autonomy architectures

Fundamentally, we do not know how to build good robotic architectures.?

Not a smooth evolution from previous practices

industrial robot -> self-driving car: a big gap?

## 5.2. Further reading

- Braitenberg vehicles - Online version in German, on Springer
- 4D RCS architecture
- How the body shapes the way we think.

## 5.3. Figures

Figure 5.1. Add caption here

Figure 5.2. Add caption here

Figure 5.3. Add caption here

Figure 5.4. Add caption here

Figure 5.5. Add caption here

Figure 5.6. Add caption here

Figure 5.7. Add caption here

Figure 5.8. Add caption here

# Unit H-6
# Representations

Assigned to: Matt

Knowledge and activity graph

Required Reading: The following assumes working knowledge of 2D and 3D Cartesian coordinate systems, reference frames, and coordinate transformations. If you are not familiar with these topics, please see the following preliminary chapters.

- Coordinate systems
- Reference frames
- Coordinate transformations.

Robots are embodied autonomous agents that interact with a physical environment.

As you read through this book, you will find that shared representations of the agent (i.e., the robot) and the environment in which it operates are fundamental to a robot's ability to sense, plan, and act—the capabilities that are key to making a robot a robot.

The *state* $\boldsymbol{x}_t \in \mathcal{X}$ is a representation that consists of a compilation of all knowledge about the robot and its environment that is *sufficient* both to perform a particular task as well as to predict the future. Of course, "predict the future" is vague, and we can not expect the state to include knowledge to predict *everything* about the future, but rather what is relevant in the context of the task.

**Definition 43.** The state $\boldsymbol{x}_t \in \mathcal{X}$ is a representation of the robot and the environment that is sufficient to predict the future in the context of the task being performed.

+ comment

I understand that explicitly referencing the task as relevant to the notion of state is odd, but I want to convey that the representation that we choose for the state does depend upon the task.

Now, let's be a bit more formal with regards to what we mean by a state being "sufficient" to predict the future. We are specifically referring to a state that exhibits the *Markov property*, i.e., that it provides a complete summary of the past (again, in the context of the task). Mathematically, we say that a state is *Markov* if the future state is independent of the past given the present state (technically, this corresponds to first-order Markov):

$$p(x_{t+1}|x_t, a_t, x_{t-1}, a_{t-1}, \ldots, x_0, a_0) = p(x_{t+1}|x_t, a_t)$$

A state exhibits the Markov property if and only if the above holds.

+ comment

I usually say that "state" is all that is needed to predict the future. The agent

only needs to keep track of a smaller thing than the state to act optimally. There isn't a good name to use; but it should be distinct from "state".

+ comment

I think that this oversimplifies things. What is it about the future that is being predicted? Certainly, the states used by autonomous systems aren't sufficient to predict everything about the future (e.g., a self-driving car can't predict whether a street light is going to come on, but that probably doesn't matter).

Knowledge about the robot and its environment is often extracted from the robot's multimodal sensor streams, such as wheel encoders and cameras. Consequently, one might choose to formulate the state as the collection of all of the measurements that the robot acquires over time. Indeed, the use of low-level sensor measurements as the representation of the state has a long history in robotics and artificial intelligence and has received renewed attention of-late, notably in the context of deep learning approaches to visuomotor policy learning.

However, while measurement history is a sufficient representation of the robot and its operating environment, it serves as a challenging definition of state.

First, raw measurements are redundant, both within a single observation and across successive measurements. For example, one doesn't need to reason over all pixels in an image, let alone all pixels within successive images to understand the location of a street sign.

Second, raw measurements contain a large amount of information that is not necessary for a given task. For example, the pixel intensities that capture the complex diffusion light due to clouds convey information that is not useful for self-driving cars. Requiring algorithms that deal with state to reason over these pixels would be unnecessarily burdensome.

Third, measurement history is very inefficient: its size grows linearly with time as the robot makes new observations, and it may be computationally intractable to access and process such a large amount of data.

This motivates the desire for a *minimal* representation that expresses knowledge that is both necessary and sufficient for the robot to perform a given task. More concretely, we will consider parameterized (symbolic) formulations of the state and will prefer representations that involve as few parameters as possible, subject to the state being Markov and the constraints imposed by the task.

Metric spaces (namely, Euclidean spaces) constitute the most commonly adopted state space in robotics, be it through the use of feature-based parameterizations of the state or gridmap representation. However, it is not uncommon to define the state of the robot and its environment in terms of a topological space or a hybrid metric-topological space.

Importantly, the exteroceptive and proprioceptive sensor measurements from which the robot's perception algorithms infer the state are noisy, the models that describe the motion of the robot and environment are error-prone, and many aspects of the state are not directly observable (e.g., your Duckiebot may not be able to "see" some of the other Duckiebots on the road due to occlusions or the cameras limited field-of-view). As a result, robots must reason over probabilistic models of the state, commonly referred to as the *belief*, in order to effectively mitigate this uncertainty.

## 6.1. Robot Representations

TODO: Discuss conventions

The state of the robot typically includes its *pose* $\boldsymbol{q}_t$, which specifies the position and orientation of a coordinate frame affixed to the robot relative to a fixed global coordinate frame commonly referred to as the "world frame". The pose then defines the rigid-body rigid-body transformation between the two reference frames.

For rigid-body robots that operate in a plane ($\mathbb{R}^2$), the pose $\boldsymbol{q} \in \mathbf{SE(2)}$ consists of the Cartesian coordinates $(\boldsymbol{x}, \boldsymbol{y})$ that specify the robot's position and the angle $\boldsymbol{\theta}$ that defines the robot's orientation (yaw). For robots in $\mathbb{R}^3$, including some ground platforms, aerial vehicles, and underwater vehicles, the pose $\boldsymbol{q} \in \mathbf{SE(3)}$ consists of three Cartesian coordinates $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ that encode the robot's position, and three Euler angles $\boldsymbol{\phi}, \boldsymbol{\theta}, \boldsymbol{\psi}$ that specify the robot's orientation.

Figure 6.1. The pose $\pose_t$ of a robot operating in a two-dimensional world consists of the robots $(x,y)$ position and orientation $\theta$ relative to a fixed reference frame.

In addition to the robot's pose, it is often useful to include its linear and angular velocities as elements of the state, resulting in an additional set of three or six parameters for robots that operate in two dimensions and three dimensions, respectively.

TODO: Discuss specific robot state representation for Duckietown.

## 6.2. Environment Representations

The two most commonly used metric representations of the robot's environment are occupancy grid models and feature-based representations.

Occupancy grid representations discretize the world into a set of grid cells. Associated with each cell are its Cartesian coordinates in a global reference frame, $(\boldsymbol{x}, \boldsymbol{y})$ in $\mathbb{R}^2$ and $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ in $\mathbb{R}^3$, and a binary label that indicates whether the cell is occupied, where a value of one denotes that the cell is occupied.

+ comment
pictures for these two? -AC

Feature-based models constitute the second commonly used environment representation. Feature-based representations model the environment as a collection of landmarks, and parameterize each in terms of the landmark's position ($(\boldsymbol{x}, \boldsymbol{y})$ or $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ in $\mathbb{R}^2$ and $\mathbb{R}^3$, respectively) and possibly its orientation.

Discuss:

- Difference between topological and metric environment representations;
- Details of topological representation;
- Common metric representations, notably feature-based maps and gridmaps;

### 1) Duckietown Environment Representation

**TODO:** Discuss specific environment representation for Duckietown.

**Author:** Matthew Walter

**Maintainer:** Matthew Walter

UNIT H-7

# Modern signal processing

Assigned to: Andrea Censi

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Understanding of time series. See: Unit G-21 - Time series.

Results: Understanding of the basic concepts of event-based processing and why it is different from periodic processing.

Results: Understanding of latency, frequency.

## 7.1. Event-based processing

The response to these challenges claims the reformulation of answers to fundamental questions referred to discrete-time systems: "when to sample," "when to update control action," or "when to transmit information." XXX

One of the generic answers to these questions is covered by adoption of the event-based paradigm that represents a model of calls for resources only if it is really necessary in system operation. In event-based systems, any activity is triggered as a response to events defined usually as a significant change of the state of relevance. Most "man-made" products, such as computer networks, complex software architectures, and production processes are natural, discrete event systems as all activity in these systems is driven by asynchronous occurrences of events. XXX

## 7.2. Periodic vs event-based processing

In standard signal processing, the data is assumed to arrive periodically with a certain fixed interval; in robotics, it is common to work with streams of data that arrive irregularly.

As per Definition 38, a time series is a sequence $\langle t_k, x_k \rangle \in \mathbb{T} \times \mathcal{X}$, where $\mathbb{T}$ is time and $\mathcal{X}$ is the domain in which the signals take values.

If the time series is periodic, it means that

$$t_k = t_0 + \Delta t_k.$$

Therefore, periodic processing is a special case of event-based processing.

## 7.3. Why working with events

How

### 1) Sensor-driven processing

The sensor tells the controller when there is interesting data to process.

#### 2) Task-driven processing

The controller tells the sensor when to send data.

### 7.4. Definition of message statistics

#### 1) Latency

TODO: to write

#### 2) Frequency

TODO: to write

#### 3) Jitter

TODO: to write

#### 4) Signal reliability

TODO: to write

Some events might be lost, because the network loses the packet.

### 7.5. On the independence of latency and frequency

Latency and frequency are to be considered two completely independent quantities. Here's

### 7.6. Design considerations and trade-offs

Should you structure your application with periodic processing, or event-based processing? Here's a few things to keep in mind.

#### 1) Need for real-time system

To have truly periodic processing, you need to have an operating system that is "real-time".

➡ Section 24.1 - Real-time Operating system

A real-time operating system will be able to schedule processing of data at given intervals.

#### 2) Periodic processing is easier to analyze

Periodic processing is easier to analyze from the theoretical point of view.

#### 3) Packet losses vs latency

There are situations where sometimes you prefer to lose packets.

Assumes: k:udp, k:tcp

4) Latency

➡ Latency numbers every programmer should know

## 7.7. Further reading

- A reference on all things event-based (control and signal processing) [23]
- A simple discussion of event-based control by K. J. Åström [24]

For a couple of different possible models for event-based processing:

- Synchronous Data Flow [25], in which each actor consumes a fixed amount of messages on each port.
- Delta Data Flow [26]

# Unit H-8
# Middleware

Assigned to: Andrea

Knowledge and activity graph

Results: Knowledge of middleware, marshalling, service discovery.

## 8.1. Why using middleware

- Reusable code
- Abstraction

Assumes: k:code-reusability, k:code-abstraction, k:code-encapsulation.

## 8.2. What middleware provides

### 1) Marshalling

Also known as “serialization”. In Python, this is called “pickling”.

### 2) Service discovery

Service discovery: do I know who to contact

Pinging `.local` addresses

### 3) Scheduling

Deciding when to do some processing

## 8.3. A few alternatives of robotics middleware

- ROS
- DDS
- LCM

## 8.4. Trade-offs and design considerations

### 1) Starting curve vs long-term productivity

TODO: to write

### 2) Performance vs convenience

TODO: to write

# Unit H-9
# Modularization and Contracts

Assigned to: Andrea Censi

## 9.1. Monolithic vs modular design

Why do we want to separate things into modules?

- Now we can divide the work among different people.

Now we need interfaces

## 9.2. Contracts

We call "contracts" the guarantees that modules give to each other.

We organize the discussion as follows:

- API: Types and signatures
- Timing: Latency, frequency, availability
- Resources: Computation and memory
- Semantics
- Quality: Reliability / probability

## 9.3. Level 0: API (Types and signatures)

TODO: to write

## 9.4. Level 1: Timing: latency, frequency, availability

TODO: to write

## 9.5. Level 2: Resources: Computation and memory

TODO: to write

## 9.6. Level 3: Semantics

TODO: to write

e.g. reference frames

## 9.7. Level 4: Quality

Reliability, probability

# Unit H-10
# Configuration management

Assigned to: Andrea

Knowledge and activity graph

Results: How to deal with configuration effectively.

## 10.1. Motivation

### 1) Aside: Python

We use Python examples but the general concepts are applicable to any programming languages. Some of the idioms here are good for dynamic object-oriented languages.

## 10.2. Stages of configuration management: from clueless to pro

We will look at the typical story of how one deals with parameters.

TODO: summary

## 10.3. The primordial stage (clueless developer)

At the beginning you have:

```
def f(x):
    return x * 10 + 1
```

Then it becomes:

```
def f(x):
    # return x * 10 + 1
    return x * 11 + 1
```

## 10.4. Recognition that parameters are important

The next step is making sure that the parameters are extracted from the code

After this step, the code looks much better:

```
def f(x):
    a = 10
    b = 1
    y = x * a + b
    return y
```

Why is it better?

- It separates the "business logic" from the parameters.
- It gives meaningful name to the parameters

### 1) Aside: legibility of code

Note that it is not true that having fewer lines of code means the code is better! Legibility first.

> Code is for humans to read, and only tangentially for machines to interpret. -???

➡ Python manifesto

## 10.5. Parameters in interfaces

The next step is recognizing the parameters in the interface of the function.

In this case, we might write something like this:

```
def f(x, a=10, b=1):
    y = x * a + b
    return y
```

The next step is encapsulation of the object.

**TODO:** parameters and defaults

## 10.6. Encapsulation of functionality objects

For example, consider an image resizing function:

```
def resize(y, w, h):
    y = cv2.imresize(x, (w, h))
    return y
```

Probably the better way is the following. We create an `ImageResizer` class that is parametrized.

```
class ImageResizer():

    def __init__(self, w, h):
        self.w, self.h = w, h

    def __call__(self, x):
        y = cv2.imresize(x, (self.w, self.h))
        return y
```

Now we can call this object as follows:

```
image_filter = ImageResizer(w=200,h=320)

x = ...
y = image_filter(x)
```

Notice that now we have abstracted the interface from the implementation: after the object is created, we can call it an "image filter", with the implication that it is "something that given an image into another".

TODO: For example, we can now write a generic `apply_filter` function:

### 1) In Duckietown

TODO: In Duckietown...

## 10.7. Convenient interfaces

Now, we still have the problem of how the user specified these parameters. We need some sort of glue that goes from the user interface, which might be the command line interface, a graphical interface, or configuration files.

It makes sense that this functionality is implemented consistently across the system.

## 10.8. Abstracting the configuration

The next step is understanding that a user never wants to deal with single parameters, but rather we need to give names to entire configurations.

For example, we wight want to say something like:

```
image_resizer_large:
    w: 640
    h: 480

image_resizer_small:
    w: 320
    h: 320
```

From the point of view of the user, it is easy to experiment.

From the point of view of the developer, it promotes an approach in which one writes more generic code.

If there

```
apply_filters:
    filter_names:
    - image_resizer_large
    - increase_contrast
    - crop_bottom
```

For example, suppose that there is a magic function

```
def instance(name):
    """ Returns an instantiation of the named object """
    ...
```

Then, one could implement a filter combinations like in the following code. The initialization parameter is the names of the filters, which are then instantiated and stored. The `__call__()` function calls all the filters in the that where defined

```
class ApplyFilters():
    def __init__(self, filter_names):
        self.filters = map(instance, filter_names)

    def __call__(self, image):
        for f in self.filters:
            image = f(image)
        return image
```

## 10.9. Duckietown solution

We now look at the Duckietown solution for the previous problems.

The code is in EasyAlgo.

## 10.10. Beyong: Customization, History tracking

# Unit H-11
# Duckiebot modeling

Obtaining a mathematical model of the Duckiebot is important in order to (a) understand its behavior and (b) design a controller to obtain desired behaviors and performances, robustly.

The Duckiebot uses an actuator (DC motor) for each wheel. By applying different torques to the wheels a Duckiebot can turn, go straight (same torque to both wheels) or stay still (no torque to both wheels). This driving configuration is referred to as *differential drive*.

In this section we will derive the model of a differential drive wheeled robot. The Duckiebot model will receive voltages as input (to the DC motors) and produce a configuration, or pose, as output. The pose describes unequivocally the position and orientation of the Duckiebot with respect to some Newtonian "world" frame.

Different methods can be followed to obtain the Duckiebot model, namely the Lagrangian or Newton-Euler, we choose to describe the latter as it arguably provides a clearer physical insight. Showing the equivalence of these formulations is an interesting exercises that the interested reader can take as a challenge. A useful resource for modeling of a Duckiebot may be found here [27].

Knowledge and activity graph

Requires:k:reference_frames (inertial, body), k:intro-transformations (Cartesian, polar)

**Requires:** k:intro-kinematics

**Requires:** k:intro-dynamics

Suggested: k:intro-ODEs-to-LTIsys

**Results:** k:diff-drive-robot-model

## 11.1. Preliminaries

TODO: relabel inertial frame -> local frame; $(\cdot)^I \rightarrow (\cdot)^L$

We first briefly recapitulate on the (reference frames)[#reference-frames] that we will use to model the Duckiebot, with the intent of introducing the notation used throughout this chapter. It is important to note that we restrict the current analysis to the plane, so all of the following in defined in $\mathbb{R}^2$.

To describe the behavior of a Duckiebot three reference frames will be used:

- A "*world*" frame: a right handed fixed reference system with origin in some arbitrary point $O$. We will indicate variables expressed in this frame with a superscript $W$, e.g., $X^W$, unless there is no risk of ambiguity, in which case no superscript will be used.
- An "*inertial*" frame: a fixed reference system parallel to the "world" frame, that spans the plane on which the Duckiebot moves. We will denote its axis as $\{X_I, Y_I\}$

, and it will have origin in point $A = (x_A, y_A)$, i.e., the midpoint of the robot's wheel axle. We will indicate variables expressed in this frame with a superscript $I$, e.g., $X^I$, unless there is no risk of ambiguity, in which case no superscript will be used.

- A *body* (or "robot") frame: a local reference frame fixed with respect to the robot, centered in $A$ as well. The $x$ axis points in the direction of the front of the robot, and the $y$ axis lies along the axis between the wheels, so to form a right handed reference system. We denote the robot body frame with $\{X_R, X_R\}$. The same superscript convention as above will be used. The wheels will have radius $R$.

> Note: The robot is assumed to be a rigid body, symmetric, and $X_r$ coincides with axis of symmetry. Moreover, the wheels are considered identical and at the same distance, $L$, from the axle midpoint $A$.

Moreover:

- The center of mass $C^W = (x_c, y_c)$ of the robot is on the $x_r$ axis, at a distance $c$ from $A$, i.e., ($C^R = (c, 0)$);
- $X^r$ forms an *orientation angle* $\theta$ with the local horizontal.

These notations are summarized in Figure 11.1.

Figure 11.1. Relevant notations for modeling a differential drive robot

### 1) Moving between frames

We briefly recapitulate on a few transformations that we will use throughout this chapter.

*Translations:*

Let $\mathbf{x^I} = [x^I, y^I]$ be a vector represented in the inertial frame and $\mathbf{X^I} = [x^I, y^I, 1]^T$ be its augmented version. It is possible to express such vector in the world frame through a translation:

$$\mathbf{X^W} = \mathbf{T}\mathbf{X^R}, \tag{11}$$

where the translation matrix $\mathbf{T}$ is defined as:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & x_A \\ 0 & 1 & y_A \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x^I \\ y^I \\ 1 \end{bmatrix}.$$

In the Euclidian space, each translation preserves distances (norms), i.e., is an isometry. So, for example, a velocity expressed in the inertial frame will have the same magnitude as that velocity expressed in the world frame.

*Rotations:*

Let $\mathbf{x^R} = [x^R, y^R]^T$ be a vector represented in the robot frame and $\mathbf{X^R} = [x^R, y^R, 1]^T$ its augmented version. It is possible to express such vector in the inertial frame through a rotation:

$$\mathbf{X^I} = \mathbf{R}(\theta)\mathbf{X^R}, \tag{12}$$

where $\mathbf{R}(\theta) \in SO(2)$ is an orthogonal rotation matrix:

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{13}$$

Note: The orthogonality condition implies that $\mathbf{R}^T(\theta)\mathbf{R}(\theta) = \mathbf{R}(\theta)\mathbf{R}^T(\theta) = \mathbf{I}$, hence:

$$\mathbf{R}^T(\theta) = \mathbf{R}^{-1}(\theta), \tag{1}$$

which is quite nice.

*Roto-translation:*

A corollary of (11) and (12) is that the translations and rotations can be combined in a single transformation, because $\mathbf{X^W} = \mathbf{T}\mathbf{X^I} = \mathbf{T}\mathbf{R}(\theta)\mathbf{X^R}$. The combined transformation matrix is given by:

$$\mathbf{T}\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & x_A \\ sin\theta & \cos\theta & y_A \\ 0 & 0 & 1 \end{bmatrix}. \tag{14}$$

## 11.2. Dynamics

While kinematics studies the properties of motions of geometric (i.e., massless) points, dynamical modeling takes into account the actual material distribution of the system. Once mass comes into play, motion is the result of the equilibrium of external forces and torques with inertial reactions. While different approaches can be used to derive these equations, namely the Lagrangian or Newtonian approaches (former based on energy considerations, latter on equilibrium of generalized forces), we choose to follow the Newtonian one here for it grants, arguably, a more explicit physical intuition of the problem. Obviously both methods lead to the same results when the same hypothesis are made.

### 1) Notations

For starters, recalling that $C^r = (c, 0)$ is the center of mass of the robot, we define the relevant notations:

TABLE 11.1. NOTATIONS FOR DYNAMIC MODELING OF A DIFFERENTIAL DRIVE ROBOT

| | |
|---|---|
| $(v_u, v_w)$ | Longitudinal and lateral velocities of $C$, robot frame |
| $(a_u, a_w)$ | Longitudinal and lateral accelerations of $C$, robot frame |
| $(F_{u_R}, F_{u_L})$ | Longitudinal forces exerted on the vehicle by the right and left wheels |
| $(F_{w_R}, F_{w_L})$ | Lateral forces exerted on the vehicle by the right and left wheels |
| $(\tau_R, \tau_L)$ | Torques acting on right and left wheel |
| $\theta$, $\omega = \dot{\theta}$ | Vehicle orientation and angular velocity |
| $M$ | Vehicle mass |
| $J$ | Vehicle yaw moment of inertia with respect to the center of mass $C$ |

Figure 11.2 summarizes these notations.

Before deriving the dynamic model of the robot, it is useful to recall some elements of polar coordinates kinematics.

### 2) Polar coordinates kinematics

Let $\mathbf{r}(t)$ identify a point in the space from the inertial frame at distance $r(t)$ from the $A$.

Warning: You might want to refresh Euler formula to convince yourself about the following.

$$\begin{aligned}
\mathbf{r}(t) &= r(t)e^{j\theta(t)} \\
\dot{\mathbf{r}}(t) &= v_u(t)e^{j\theta(t)} + v_w(t)e^{j(\theta(t)+\frac{\pi}{2})} \\
\ddot{\mathbf{r}}(t) &= a_u(t)e^{j\theta(t)} + a_w(t)e^{j(\theta(t)+\frac{\pi}{2})},
\end{aligned} \tag{15}$$

with:

$$\begin{aligned}
v_u(t) &= \dot{r}(t) \\
v_w(t) &= r(t)\dot{\theta}(t) \\
a_u(t) &= \dot{v}_u - v_w\dot{\theta}(t) = \ddot{r}(t) - r\dot{\theta}^2(t) \\
a_w(t) &= \dot{v}_w + v_u\dot{\theta}(t) = 2\dot{r}(t)\dot{\theta}(t) + r(t)\ddot{\theta}(t).
\end{aligned} \tag{16}$$

Keeping (15) and (16) in mind, it is useful to note (for later use) that, letting $\mathbf{r}(t)$ identify the position of the center of mass $C$ in the inertial frame:

$$\begin{cases} x_C^W(t) &= x_A(t) + r(t)\cos\theta(t) \\ y_C^W(t) &= y_A(t) + r(t)\sin\theta(t) \end{cases}, \tag{17}$$

and therefore:

$$\begin{cases} \dot{x}_A^I(t) &= x_C^W(t) - v_u(t)\cos\theta(t) + v_w(t)\sin\theta(t) \\ \dot{y}_A^I(t) &= y_C^W(t) - v_u(t)\sin\theta(t) - v_w(t)\cos\theta(t) \end{cases}. \tag{18}$$

### 3) Free body diagram

The next step, and definitely the most critical, is writing the free body diagram of the problem (Figure 11.2). In this analysis the only forces acting on the robot are those applied from the wheels to the vehicle's chassis. It is important to note that the third passive wheel (omnidirectional or caster) is not being taken into account.

Figure 11.2. Free body diagram of a differential drive robot

### 4) Equilibrium of forces and moments

We derive the dynamic model by imposing the simultaneous equilibrium of forces along the longitudinal and lateral directions in the robot frame with the respective inertial forces, and of the moments around the vertical axis (coming out of the.. screen) passing through the center of mass of the robotic vehicle.

$$\begin{aligned} Ma_u(t) &= F_{u_L} + F_{u_R} \\ Ma_w(t) &= F_{w_L} + F_{w_R} \\ \ddot{\theta}(t) &= \frac{L}{J}(F_{u_R} - F_{u_L}) + \frac{c}{J}(F_{w_R} + F_{w_L}). \end{aligned} \tag{19}$$

By substituting the (16) in (19), the equilibrium equations are expressed in terms of accelerations of the center of mass in the robot frame:

$$\dot{v}_u(t) = v_w(t)\dot{\theta}(t) + \frac{F_{u_L} + F_{u_R}}{M} \tag{20}$$

$$\dot{v}_w(t) = -v_u(t)\dot{\theta}(t) + \frac{F_{w_L} + F_{w_R}}{M} \tag{21}$$

$$\ddot{\theta}(t) = \frac{L}{J}(F_{u_R} - F_{u_L}) - \frac{c}{J}(F_{w_R} + F_{w_L}). \tag{22}$$

This is a general dynamic model (in the sense of no kinematic constraints) of a differential drive robot under the geometric assumptions listed above. It is noted that it is a coupled and nonlinear model, not exactly a best case scenario (we like linear

things, because there are plenty of tools to handle them). When using the general dynamic model above, it makes sense to associate the general kinematic model as well, given by:

$$\begin{aligned}\dot{x}_A(t) &= v_u(t)\cos\theta(t) - (v_w(t) - c\dot{\theta})\sin\theta(t) \\ \dot{y}_A(t) &= v_u(t)\sin\theta(t) + (v_w(t) - c\dot{\theta})\sin\theta(t).\end{aligned} \tag{23}$$

the above (23) can be obtained by recalling on one side that translations are isometric transformations, and on the other side that:

$$\begin{cases} x_A(t) &= x_C^W(t) - c\cos\theta(t) \\ y_A(t) &= y_C^W(t) - c\sin\theta(t) \end{cases} \tag{24}$$

$$\begin{cases} x_C^I(t) &= v_u(t)\cos\theta(t) - v_w(t)\sin\theta(t) \\ y_C^I(t) &= v_u(t)\sin\theta(t) + v_w(t)\cos\theta(t) \end{cases} \tag{25}$$

**Note:** Equation (23) can be rewritten as:

$$\mathbf{v}_\mathbf{A}^\mathbf{I} = \mathbf{R}(\theta)\mathbf{v}_\mathbf{A}^\mathbf{R} \tag{2}$$

, where:

$$\mathbf{v}_\mathbf{A}^\mathbf{R} = [v_u(t), v_w(t) - c\dot{\theta}(t)]^T. \tag{3}$$

In order to simplify the model, we proceed to impose some kinematic constraints.

## 11.3. Kinematics

In this section we derive the kinematic model of a differential drive mobile platform under the assumptions of (a) no lateral slipping and (b) pure rolling of the wheels. We refer to these two assumptions as kinematic constraints.

### 1) Differential drive robot kinematic constraints

The kinematic constraints are derived from two assumptions:

- *No lateral slipping motion*: the robot cannot move sideways, but only in the direction of motion, i.e., its lateral velocity in the robot frame is zero:

$$\dot{y}_A^r = 0. \tag{4}$$

By inverting (12), this constraint can be expressed through the inertial frame variables, yielding:

$$\dot{y}_A(t)\cos\theta(t) - \dot{x}_A(t)\sin\theta(t) = 0. \tag{5}$$

Imposing (5) on (18) results in:

$$v_w(t) = \dot{y}_C^I(t)\cos\theta(t) - \dot{x}_C^I(t)\sin\theta(t), \tag{6}$$

and by recalling that $C^R = (c, 0)$:

$$\dot{y}_C^I(t)\cos\theta(t) - \dot{x}_C^I(t)\sin\theta(t) = c\dot{\theta}(t). \tag{7}$$

Hence, we obtain the strongest expression of this constraint:

$$v_w(t) = c\dot{\theta}(t), \tag{8}$$

and therefore:

$$\dot{v}_w(t) = c\ddot{\theta}(t). \tag{9}$$

**Note:** a simpler way of deriving (9) is noticing, from (3), that $\dot{y}_A^R = v_w(t) - c\dot{\theta}(t)$.

- *Pure rolling*: the wheels never slips or skids (Figure 11.3). Recalling that $R$ is the radius of the wheels (identical) and letting $\dot{\varphi}_l, \dot{\varphi}_r$ be the angular velocities of the left and right wheels respectively, the velocity of the ground contact point P in the robot frame is given by:

Figure 11.3. Pure rolling kinematic constraint

$$\begin{cases} v_{P,r} &= R\dot{\varphi}_r \\ v_{P,l} &= R\dot{\varphi}_l \end{cases}. \tag{26}$$

Another notable consequence of this assumption is that, always in the robot frame, the full power of the motor can be translated into a propelling force for the vehicle in the longitudinal direction. Or, more simply, it allows to write:

$$F_{u,(\cdot)}(t)R = \tau_{(\cdot)}(t), \tag{10}$$

where $\tau_{(\cdot)}(t)$ is the torque exerted by each motor on its wheel $(\cdot) = l, r$.

## 11.4. Differential drive robot kinematic model

In a differential drive robot, controlling the wheels at different speeds generates a rolling motion of rate $\omega = \dot{\theta}$. In a rotating field there always is a fixed point, the *center of instantaneous curvature* (ICC), and all points at distance $d$ from it will have a velocity given by $\omega d$, and direction orthogonal to that of the line connecting the ICC and the wheels (i.e., the *axle*). Therefore, by looking at Figure 11.1, we can write:

$$\begin{cases} \dot{\theta}(d - L) &= v_l \\ \dot{\theta}(d + L) &= v_r \end{cases}, \tag{27}$$

from which:

$$\begin{cases} d &= L\frac{v_r+v_l}{v_r-v_l} \\ \dot{\theta} &= \frac{v_r-v_l}{2L} \end{cases}. \tag{28}$$

A few observations stem from (28):

- If $v_r = v_l$ the bot does not turn ($\dot{\theta} = 0$), hence the ICC is not defined;
- If $v_r = -v_l$, then the robot "turns on itself", i.e., $d = 0$ and $ICC \equiv A$;
- If $v_r = 0$ (or $v_l = 0$), the rotation happens around the right (left) wheel and $d = 2L$ ($d = L$).

**Note:** Moreover, a differential drive robot cannot move in the direction of the ICC, it is a singularity.

By recalling the *no lateral slipping motion* (4) hypothesis and the *pure rolling* constraint (26), and noticing that the translational velocity of $A$ in the robot frame is $v_A = \dot{\theta}d = (v_r + v_l)/2$ we can write:

$$\begin{cases} \dot{x}_A^R &= R(\dot{\varphi}_R + \dot{\varphi}_L)/2 \\ \dot{y}_A^R &= 0 \\ \dot{\theta} &= \omega = R(\dot{\varphi}_R - \dot{\varphi}_L)/(2L) \end{cases}, \tag{29}$$

which in more compact yields the *simplified forward kinematics* in the robot frame:

$$\begin{bmatrix} \dot{x}_A^R \\ \dot{y}_A^R \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{R}{2} & \frac{R}{2} \\ 0 & 0 \\ \frac{R}{2L} & -\frac{R}{2L} \end{bmatrix} \begin{bmatrix} \dot{\varphi}_R \\ \dot{\varphi}_L \end{bmatrix}. \tag{30}$$

Finally, by using (13), we can recast (30) in the inertial frame.

**Note:** The *simplified forward kinematics* model of a differential drive vehicle is given by:

$$\dot{\mathbf{q}}^I = \mathbf{R}(\theta) \begin{bmatrix} \dot{x}_A^r \\ \dot{y}_A^r \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{R}{2}\cos\theta & \frac{R}{2}\cos\theta \\ \frac{R}{2}\sin\theta & \frac{R}{2}\sin\theta \\ \frac{R}{2L} & -\frac{R}{2L} \end{bmatrix} \begin{bmatrix} \dot{\varphi}_R \\ \dot{\varphi}_L \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_A \\ \omega \end{bmatrix}. \tag{31}$$

## 11.5. Simplified dynamic model

By implementing the kinematic constraints formulations derived above, i.e., the no lateral slipping ((9)) and pure rolling ((10)) in the general dynamic model, it is straightforward to obtain:

$$\begin{aligned} \dot{v}_u(t) &= c\dot{\theta}^2(t) + \frac{1}{RM}(\tau_R(t) + \tau_L(t)) \\ \dot{v}_w(t) &= c\dot{\theta}(t) \\ \ddot{\theta} &= -\frac{Mc}{Mc^2 + J}\dot{\theta}(t)v_u(t) + \frac{L}{R(Mc^2 + J)}(\tau_R(t) - \tau_L(t)) \end{aligned} \tag{32}$$

## 11.6. DC motor dynamic model

The equations governing the behavior of a DC motor are driven by an input *armature voltage* $V(t)$:

$$
\begin{aligned}
V(t) &= Ri(t) + L\frac{di}{dt} + e(t) \\
e(t) &= K_b\dot{\varphi}(t) \\
\tau_m(t) &= K_t i(t) \\
\tau(t) &= N\tau_m(t),
\end{aligned}
$$

where $(K_b, K_t)$ are the back emf and torque constants respectively and $N$ is the gear ratio ($N = 1$ in the Duckiebot).

Figure 11.4 shows a diagram of a typical DC motor.

Figure 11.4. Diagram of a DC motor

Having a relation between the applied voltage and torque, in addition to the dynamic and kinematic models of a differential drive robot, allows us to determine all possible state variables of interest.

> Note: torque disturbances acting on the wheels, such as the effects of friction, can be modeled as additive terms (of sign opposite to $\tau$) in the DC motor equations.

## 11.7. Conclusions

In this chapter we derived a model for a differential drive robot. Although several simplifying assumption were made, e.g., rigid body motion, symmetry, pure rolling and no lateral slipping - still the model is nonlinear.

Regardless, we now have a sequence of descriptive tools that receive as input the voltage signal sent by the controller, and produce as output any of the state variables, e.g., the position, velocity and orientation of the robot with respect to a fixed inertial frame.

Several outstanding questions remain. For example, we need to determine what is the best representation for our robotic platform - polar coordinates, Cartesian with respect to an arbitrary reference point? Or maybe there is a better choice?

Finally, the above model assumes the knowledge of a number of constants that are characteristic of the robot's geometry, materials, and the DC motors. Without the knowledge of those constant the model could be completely off. Determination of these parameters in a measurement driven way, i.e., the "system identification" of the robot's plant, is subject of the *odometry* class.

# Unit H-12
# Odometry Calibration

Assigned to: Jacopo

# UNIT H-13

# Computer vision basics

> **Assigned to:** Matt

For the interested reader, there are several excellent books that discuss fundamental topics in computer vision. These include (in no particular order), Computer Vision: Algorithims and Applications (available online), Computer Vision: A Modern Approach and Multiple View Geometry in Computer Vision.

## 1) Computer vision: What and why?

**Topics**:

- **Objective**: Discover what is present in the world, where things are, and what actions are taking place from image sequences. In the context of robotics, this corresponds to learning a (probabilistic) world model that encodes the robot's environment, i.e., building a representation of the environment.
- History of computer vision
  - Summer Vision Project
  - OCR (license plates, signs, checks, mail, etc.)
  - Face detection
  - Structure from motion
  - Dense reconstruction
  - Augmented reality
  - Applications to self-driving cars and beyond

# Unit H-14
# Camera geometry

Assigned to: Matt

Note: We could break this up into separate sub-sections for (prospective) projection and lenses.

**Topics**:

- Pinhole cameras (tie into eclipse viewing, accidental cameras)
- Geometry of projection: equations of projection; vanishing points; weak perspective; orthographic projection;
- Review of 3D transformations (translation and rotation); homogeneous transformations
- Perspective projection in homogeneous coordinates

## 1) Lenses

- Lenses: Why?; first-order optics (thin lens); thick lens
- First-order optics (thin lens)

UNIT H-15

# Camera calibration

Assigned to: Matt

Note: The discussion of intrinsic and extrinsic models could be moved up to the geometry subsection

Topics

- Why calibration?
- Intrinsic parameters: idealized perspective projection; pixel scaling (square and non-square); offset; skew.
- Intrinsic model expressed in homogeneous coordinates
- Extrinsic parameters: translation and rotation of camera frame (non-homogeneous and homogeneous)
- Combined expression for intrinsic and extrinsic
- Calibration targets
- Calibration models

# Unit H-16
# Image filtering

Assigned to: Matt

**Background**: Discrete-time signal processing

Note: We currently don't have a preliminary section on signal processing and it's probably sufficient to point readers to a good reference (e.g., Oppenheim and Schafer).

**Topics**

Note: May want to stick to linear filtering

- Motivation (example)
- Convolution: definition and properties (including differentiation)
- Linear filters: examples (sharpening and smoothing)
- Gaussian filters
  - kernels
  - properties (convolution with a Gaussian, separability)

UNIT H-17

# Illumination invariance

Assigned to: Matt

# Unit H-18
# Line Detection

Assigned to: Matt

**Topics:**

- What is edge detection?
- Image derivatives
- Finite difference filters
- Image gradients
- Effects of noise and need for smoothing
- Derivative theorem of convolution
- Derivative of Gaussian filters
- Smoothing vs. derivative filters
- Line detection as an edge detection problem

UNIT H-19

# Feature extraction

Assigned to: Matt

UNIT H-20

# Place recognition

Assigned to: Matt

UNIT H-21

# Filtering 1

> **Assigned to:** Liam

**TODO:** Markov Property

**TODO:** Bayes Filter

**TODO:** Graphical representation

**TODO:** Kalman Filter

**TODO:** Extended Kalman Filter

# Unit H-22
# Filtering 2

> Assigned to: Liam

**TODO:** MAP estimation

**TODO:** Laplace Approximation

**TODO:** Cholesky Decomposition?

**TODO:** Incrementalization - Givens Rotations - iSAM

UNIT H-23

# Mission planning

Assigned to: ETH

# Unit H-24
# Planning in discrete domains

Assigned to: ETH

UNIT H-25

# Motion planning

Assigned to: ETH

# UNIT H-26

# RRT

Assigned to: ETH

UNIT H-27

# Feedback control

**Assigned to:** Jacopo

# Unit H-28
# PID Control

Assigned to: Jacopo

UNIT H-29

# MPC Control

**Assigned to:** Jacopo

# Unit H-30
# Object detection

**Assigned to:** Nick and David

UNIT H-31

# Object classification

**Assigned to:** Nick and David

# Unit H-32
# Object tracking

**Assigned to:** Nick and David

UNIT H-33

# Reacting to obstacles

**Assigned to:** Jacopo

# Unit H-34
# Semantic segmentation

Assigned to: Nick and David

UNIT H-35

# Text recognition

Assigned to: Nick

UNIT H-36

# SLAM - Problem formulation

Assigned to: Liam

UNIT H-37

# SLAM - Broad categories

**Assigned to:** Liam

## Unit H-38

# VINS

Assigned to: Liam

UNIT H-39

# Advanced place recognition

**Assigned to:** Liam

UNIT H-40

# Fleet level planning (placeholder)

Assigned to: ETH

UNIT H-41

# Fleet level planning (placeholder)

Assigned to: ETH

# Unit H-42
# Bibliography

[2] Liam Paull, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Daniel Hoehener, Shih-Yuan Liu, Michael Novitzky, Igor Franzoni Okuyama, Jason Pazis, Guy Rosman, Valerio Varricchio, Hsueh-Cheng Wang, Dmitry Yershov, Hang Zhao, Michael Benjamin, Christopher Carr, Maria Zuber, Sertac Karaman, Emilio Frazzoli, Domitilla Del Vecchio, Daniela Rus, Jonathan How, John Leonard, and Andrea Censi. **Duckietown: an open, inexpensive and flexible platform for autonomy education and research.** In *IEEE International Conference on Robotics and Automation (ICRA)*. Singapore, May 2017. pdf

[3] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[1] Jacopo Tani, Liam Paull, Maria Zuber, Daniela Rus, Jonathan How, John Leonard, and Andrea Censi. **Duckietown: an innovative way to teach autonomy.** In *EduRobotics 2016*. Athens, Greece, December 2016. pdf

[6] Tosini, G., Ferguson, I., Tsubota, K. *Effects of blue light on the circadian system and eye physiology*. Molecular Vision, 22, 61–72, 2016 (online).

[7] Albert Einstein. Zur Elektrodynamik bewegter Körper. (German) On the electrodynamics of moving bodies. *Annalen der Physik*, 322(10):891–921, 1905. DOI

[15] Ivan Savov. Linear algebra explained in four pages. https://minireference.com/static/tutorials/linear_algebra_in_4_pages.pdf, 2017. Online; accessed 23 August 2017.

[14] Kaare Brandt Petersen and Michael Syskind Pedersen. The matrix cookbook. .pdf

[17] Matrix (mathematics). Matrix (mathematics) — Wikipedia, the free encyclopedia, 2017. Online; accessed 2-September-2017. www:

[18] Peter D. Lax. *Functional Analysis*. Wiley Interscience, New York, NY, USA, 2002.

[19] Athanasios Papoulis and S. Unnikrishna Pillai. *Probability, Random Variables and Stochastic Processes*. McGraw Hill, fourth edition, 2002.

[20] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

[21] H. Choset, W. Burgard, S. Hutchinson, G. Kantor, L. E. Kavraki, K. Lynch, and S. Thrun. *Principles of robot motion: Theory, agorithms, and implementation*. MIT Press, June 2005.

[22] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, New York, NY, USA, 2006.

[23] M. Miskowicz. *Event-Based Control and Signal Processing*. Embedded Systems. CRC Press, 2015. http

[24] Karl J. Aström. *Event Based Control*, pages 127–147. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. DOI http

[25] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[26] Rajit Manohar and K Mani Chandy. Delta-dataflow networks for event stream processing. pages 1–6, June 2010.

[27] Romano M DeSantis. Modeling and path-tracking control of a mobile wheeled robot with a differ-

ential drive. *Robotica*, 13(4):401–410, 1995.

[28] Richard Szeliski. *Computer Vision: Algorithms an Applications*. Springer, 2010. http

[29] David Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentce Hall, Upper Saddle River, NJ, 2011.

[30] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, second edition, 2004.

[31] Censi Tani. Project pitches. Slides, 10 2017.

[32] Davide Scaramuzza. Design and realization of a stereoscopic vision system for robotics, with applications to tracking of moving objects and self-localization. Master thesis, department of electronic and information engineering, University of Perugia, Perugia, Italy, 2005. .pdf

[33] Gang Yi Jiang. Lane and obstacle detection based on fast inverse perspective mapping algorithm. In *IEEE Xplore, Conference: Systems, Man, and Cybernetics, 2000 IEEE International Conference on, Volume: 4*, Febuary 2000. http

[34] Alessandra Fascioli Massimo Bertozzi, Alberto Broggi. Stereo inverse perspective mapping: Theory and applications. *Image and Vision Computing 16 (1998)*, 1997. .pdf

[35] Kesheng Wu. Optimizing connected component labeling algorithms, 2005. http

[36] Richard Fitzpatrick. Moment of inertia tensor, 03 2011. .html

[37] Google offers raspberry pi owners this new ai vision kit to spot cats, people, emotions, 12 2017. http

[38] Davide Scaramuzza. Lecture: Vision algorithms for mobile robotics, 2017. .html

[40] Davide Scaramuzza. Towards robust and safe autonomous drones, September 2015. http

[41] Rasmussen. Presentation on theme: "computer vision : Cisc 4/689". http

[42] Inverse perspective mapping, September 2012. http

[44] VIKAS GUPTA. Color spaces in opencv (c++ / python), May 2017. http

[45] Convert from hsv to rgb color space, 2018. http

[47] Michel Alves. About perception and hue histograms in hsv space, July 2015. http

[52] CHENG. Slides from the lecture "computer graphics".

[48] Hsl and hsv, 12 2017. http

# Part I
# Exercises

These exercises can guide you from the status of a novice coder to experienced roboticist.

UNIT I-1

# Exercise: Basic image operations

> **Assigned to:** Andrea Daniele

## 1.1. Skills learned

- Accessing command line arguments.
- Reading and writing files.
- Working with pixel-based image representations.
- OpenCV and `duckietown_utils` APIs for reading/writing images.

## 1.2. Instructions

Create an implementation of the program `dt-image-flip0`, specified below.

If this exercise is too easy for you, skip to [Unit I-2 - Exercise: Basic image operations, adult version](#).

## 1.3. Specification of `dt-image-flip0`

The program `dt-image-flip0` takes as an argument a JPG file with extension `.jpg`:

```
$ dt-image-flip0 file.jpg
```

and creates a file called *file*`.flipped.jpg` that is flipped around the horizontal axis.

(a) The original picture. (b) The flipped output

Figure 1.1

Example input-output for the program `dt-image-flip`.

## 1.4. Useful APIs

### 1) Load image from file

The OpenCV library provides a utility function called `imread` that loads an image from a file.

### 2) Flip an image

+ comment

This is the kind of thing that they need to figure out how to do with pixels. -AC

### 3) Write an image to a file

The `duckietown_utils` package provides the utility function `write_image_as_jpg()` that writes an image to a JPEG file.

UNIT I-2

# Exercise: Basic image operations, adult version

**Assigned to:** Andrea Daniele

## 2.1. Skills learned

- Dealing with exceptions.
- Using exit conditions.
- Verification and unit tests.

## 2.2. Instructions

Implement the program `dt-image-flip` specified in the following section.

This time, we specify exactly what should happen for various anomalous conditions. This allows to do automated testing of the program.

## 2.3. `dt-image-flip` specification

The program `image-ops` expects exactly two arguments: a filename (a JPG file) and a directory name.

```
$ dt-image-flip file outdir
```

If the file does not exist, the script must exit with error code `2`.

If the file cannot be decoded, the script must exit with error code `3`.

If the file exists, then the script must create:

- *outdir*`/regular.jpg`: a copy of the initial file
- *outdir*`/flip.jpg`: the file, flipped vertically.
- *outdir*`/side-by-side.jpg`: the two files, side by side.

If any other error occurs, the script should exit with error code `99`.

(a) The original picture. (b) The output `flip.jpg` (c) The output `side-by-side.jpg`

Figure 2.1. Example input-output for the program `image-ops`.

## 2.4. Useful APIs

### 1) Images side-by-side

+ comment

Good explanation, but shouldn't it go in the previous exercise? -AC

An image loaded using the OpenCV function `imread` is stored in memory as a NumPy array. For example, the image shown above (Figure 2.1a - The original picture. ) will be represented in memory as a NumPy array with shape `(96, 96, 3)`. The first dimension indicates the number of pixels along the `Y-axis`, the second indicates the number of pixels along the `X-axis` and the third is known as *number of channels* (e.g., **B**lue, **G**reen, and **R**ed).

NumPy provides a utility function called `concatenate` that joins a sequence of arrays along a given axis.

## 2.5. Testing it works with `image-ops-tester`

We provide 4 scripts that can be used to make sure that you wrote a conforming `dt-image-flip`. The scripts are `image-ops-tester-good`, `image-ops-tester-bad1`, `image-ops-tester-bad2`, and `image-ops-tester-bad3`. You can find them in the directory `/exercises/dt-image-flip/image-ops-tester` in the `duckietown/duckuments` repository.

The script called `image-ops-tester-good` tests your program in a situation in which we expect it to work properly. The 3 "bad" test scripts (i.e., `image-ops-tester-bad1` through `image-ops-tester-bad3`) test your code in situations in which we expect your program to complain in the proper way.

Use them as follows:

```
$ image-ops-tester-scenario candidate-program
```

> **Note:** The tester scripts must be called from their own location. Make sure to change your working directory to `/exercises/dt-image-flip/image-ops-tester` before launching the tester scripts.

If the script cannot be found, `image-ops-tester-`*scenario* will return 1.

`image-ops-tester-`*scenario* will return 0 if the program exists and conforms to the specification (Section 2.3 - `dt-image-flip` specification).

If it can establish that the program is not good, it will return 11.

## Bottom line

Things that are not tested are broken.

UNIT I-3

# Exercise: Simple data analysis from a bag

> **Assigned to:** Andrea Daniele

## 3.1. Skills learned

- Reading Bag files.
- Statistics functions (mean, median) in Numpy.
- Use YAML format.

## 3.2. Instructions

Create an implementation of `dt-bag-analyze` according to the specification below.

## 3.3. Specification for `dt-bag-analyze`

Create a program that summarizes the statistics of data in a bag file.

```
$ dt-bag-analyze bag file
```

Compute, for each topic in the bag:

- The total number of messages.
- The minimum, maximum, average, and median interval between successive messages, represented in seconds.

Print out the statistics using the YAML format. Example output:

```
$ dt-bag-analyze bag file
"topic name":
    num_messages: value
    period:
        min: value
        max: value
        average: value
        median: value
```

## 3.4. Useful APIs

### 1) Read a ROS bag

A bag is a file format in ROS for storing ROS message data. The package `rosbag` defines the class `Bag` that provides all the methods needed to serialize messages to and from a single file on disk using the bag format.

### 2) Time in ROS

In ROS the time is stored as an object of type `rostime.Time`. An object `t`, instance of `rostime.Time`, represents a time instant as the number of seconds since epoch (stored in `t.secs`) and the number of nanoseconds since `t.secs` (stored in `t.nsecs`). The utility function `t.to_sec()` returns the time (in seconds) as a floating number.

## 3.5. Test that it works

Download the ROS bag `example_rosbag_H3.bag`. Run your program on it and compare the results:

```
$ dt-bag-analyze example_rosbag.bag
/tesla/camera_node/camera_info:
  num_messages: 653
  period:
    min: 0.01
    max: 0.05
    average: 0.03
    median: 0.03

/tesla/line_detector_node/segment_list:
  num_messages: 198
  period:
    min: 0.08
    max: 0.17
    average: 0.11
    median: 0.1

/tesla/wheels_driver_node/wheels_cmd:
  num_messages: 74
  period:
    min: 0.02
    max: 4.16
    average: 0.26
    median: 0.11
```

UNIT I-4

# Exercise: Bag in, bag out

> **Assigned to:** Andrea Daniele

## 4.1. Skills learned

- Processing the contents of a bag to produce another bag.

## 4.2. Instructions

Implement the program `dt-bag-decimate` as specified below.

## 4.3. Specification of `dt-bag-decimate`

The program `dt-bag-decimate` takes as argument a bag filename, an integer value greater than zero, and an output bag file:

```
$ dt-bag-decimate "input bag" n "output bag"
```

The output bag contains the same topics as the input bag, however, only 1 in `n` messages from each topic are written. (If `n` is 1, the output is the same as the input.)

## 4.4. Useful new APIs

### 1) Create a new Bag

In ROS, a new bag can be created by specifying the mode `w` (i.e., write) while instantiating the class `rosbag.Bag`.

For example:

```
from rosbag import Bag
new_bag = Bag('./output_bag.bag', mode='w')
```

Visit the documentation page for the class `rosbag.Bag` for further information.

### 2) Write message to a Bag

A ROS bag instantiated in *write* mode accepts messages through the function `write()`.

## 4.5. Check that it works

To check that the program works, you can compute the statistics of the data using the program `dt-bag-analyze` that you have created in Unit I-3 - Exercise: Simple data

analysis from a bag.

You should see that the statistics have changed.

UNIT I-5

# Exercise: Bag thumbnails

**Assigned to:** Andrea Daniele

## 5.1. Skills learned

- Reading images from images topic in a bag file.

## 5.2. Instructions

Write a program `dt-bag-thumbnails` as specified below.

## 5.3. Specification for `dt-bag-thumbnails`

The program `dt-bag-thumbnails` creates thumbnails for some image stream topic in a bag file.

The syntax is:

```
$ dt-bag-thumbnails bag topic output_dir
```

This should create the files:

```
output_dir/00000.jpg
output_dir/00001.jpg
output_dir/00002.jpg
output_dir/00003.jpg
output_dir/00004.jpg
...
```

where the progressive number is an incremental counter for the frames.

## 5.4. Test data

If you don't have a ROS bag to work on, you can download the test bag `example_ros-bag_H5.bag`. You should be able to get a total of 653 frames out of it.

## 5.5. Useful APIs

### 1) Read image from a topic

The `duckietown_utils` package provides the utility function `rgb_from_ros()` that processes a ROS message and returns the RGB image it contains (if any).

### 2) Color space conversion

In OpenCV, an image can be converted from one color space (e.g., BGR) to another supported color space (e.g., RGB). OpenCV provides a list of supported conversions. A `ColorConversionCode` defines a conversion between two different color spaces. An exhaustive list of color conversion codes can be found here. The conversion from a color space to another is done with the function `cv.cvtColor`.

UNIT I-6

# Exercise: Instagram filters

> **Assigned to:** Andrea Daniele

## 6.1. Skills learned

- Image pixel representation;
- Image manipulation;
- The idea that we can manipulate operations as objects, and refer to them (higher-order computation);
- The idea that we can compose operations, and sometimes the operations do commute, while sometimes they do not.

## 6.2. Instructions

Create `dt-instagram` as specified below.

## 6.3. Specification for `dt-instagram`

Write a program `dt-instagram` that applies a list of filters to an image.

The syntax to invoke the program is:

```
$ dt-instagram image in filters image out
```

where:

- *image in* is the input image;
- *filters* is a string, which is a colon-separated list of filters;
- *image out* is the output image.

The list of filters is given in Subsection 6.3.1 - List of filters.

For example, the result of the command:

```
$ dt-instagram image.jpg flip-horizontal:grayscale out.jpg
```

is that `out.jpg` contains the input image, flipped and than converted to grayscale.

Because these two commute, this command gives the same output:

```
$ dt-instagram image.jpg grayscale:flip-horizontal out.jpg
```

### 1) List of filters

Here is the list of possible values for the filters, and their effect:

- `flip-vertical`: flips the image vertically
- `flip-horizontal`: flips the image horizontally

- `grayscale`: Makes the image grayscale
- `sepia`: make the image sepia

## 6.4. Useful new APIs

### 1) User defined filters

In OpenCV it is possible to define custom filters and apply them to an image. A linear filter (e.g., sepia) is defined by a linear 9-dimensional kernel. The `sepia` filter is defined as:

$$K_{sepia} = \begin{bmatrix} 0.272 & 0.534 & 0.131 \\ 0.349 & 0.686 & 0.168 \\ 0.393 & 0.769 & 0.189 \end{bmatrix}$$

A linear kernel describing a color filter defines a linear transformation in the color space. A transformation can be applied to an image in OpenCV by using the function `transform()`.

UNIT I-7

# Exercise: Bag instagram

> **Assigned to:** Andrea Daniele

## 7.1. Instructions

Create `dt-bag-instagram` as specified below.

## 7.2. Specification for `dt-bag-instagram`

Write a program `dt-bag-instagram` that applies a filter to a stream of images stored in a ROS bag.

The syntax to invoke the program is:

```
$ dt-bag-instagram bag in topic filters bag out
```

where:

```
- `bag in` is the input bag;
- `topic` is a string containing the topic to process;
- `filters` is a string, which is a colon-separated list of filters;
- `bag out` is the output bag.
```

## 7.3. Test data

If you don't have a ROS bag to work on, you can download the test bag `example_rosbag_H5.bag`.

## 7.4. Useful new APIs

### 1) Compress an BGR image into a `sensor_msgs/CompressedImage` message

The `duckietown_utils` package provides the utility function `d8_compressed_image_from_cv_image()` that takes a BGR image, compresses it and wraps it into a `sensor_msgs/CompressedImage` ROS message.

## 7.5. Check that it works

Play your *bag out* ROS bag file and run the following command to make sure that your program is working.

```
$ rosrun image_view image_view image:=topic _image_transport:=compressed
```

UNIT I-8

# Exercise: Live Instagram

> **Assigned to:** Andrea Daniele

## 8.1. Skills learned

- Live image processing

## 8.2. Instructions

You may find useful: Unit K-6 - Minimal ROS node - `pkg_name`. That tutorial is about listening to text messages and writing back text messages. Here, we apply the same principle, but to images.

Create a ROS node that takes camera images and applies a given operation, as specified in the next section, and then publishes it.

## 8.3. Specification for the node `dt_live_instagram_`*robot name*`_node`

Create a ROS node `dt_live_instagram_`*robot name*`_node` that takes a parameter called `filter`, where the filter is something from the list Subsection 6.3.1 - List of filters.

You should launch your camera and joystick from '~/duckietown' with

```
$ make demo-joystick-camera
```

Then launch your node with

```
$ roslaunch dt_live_instagram_robot name dt_live_instagram_robot name_node.launch filter:=filter
```

This program should do the following:

- Subscribe to the camera images, by finding a topic that is called `...`/`compressed`. Call the name of the topic *topic* (i.e., *topic* = `...`).
- Publish to the topic *topic*/*filter*`/compressed` a stream of images (i.e., video frames) where the filter is applied to the images.

## 8.4. Check that it works

```
$ rqt_image_view
```

and look at *topic*/*filter*`/compressed`

# Unit I-9
# Exercise: Augmented Reality

Assigned to: Jonathan Michaux and Dzenan Lapandic

## 9.1. Skills learned

- Understanding of all the steps in the image pipeline.
- Writing markers on images to aid in debugging.

## 9.2. Introduction

During the lectures, we have explained one direction of the image pipeline:

```
image -> [feature extraction] -> 2D features -> [ground projection] -> 3D world coordinates
```

In this exercise, we are going to look at the pipeline in the opposite direction.

It is often said that:

> "The inverse of computer vision is computer graphics."

The inverse pipeline looks like this:

```
3D world coordinates -> [image projection] -> 2D features -> [rendering] -> image
```

## 9.3. Instructions

- Do intrinsics/extrinsics camera calibration of your robot as per the instructions.
- Write the ROS node specified below in Section 9.4 - Specification of `dt_augmented_reality`.

Then verify the results in the following 3 situations.

### 1) Situation 1: Calibration pattern

- Put the robot in the middle of the calibration pattern.
- Run the program `dt_augmented_reality` with map file `calibration_pattern.yaml`.

(Adjust the position until you get perfect match of reality and augmented reality.)

### 2) Situation 2: Lane

- Put the robot in the middle of a lane.
- Run the program `dt_augmented_reality` with map file `lane.yaml`.

(Adjust the position until you get a perfect match of reality and augmented reality.)

### 3) Situation 3: Intersection

- Put the robot at a stop line at a 4-way intersection in Duckietown.

- Run the program `dt_augmented_reality` with map file `intersection_4way.yaml`.

(Adjust the position until you get a perfect match of reality and augmented reality.)

#### 4) Submission

Submit the images according to location-specific instructions.

## 9.4. Specification of `dt_augmented_reality`

In this assignment you will be writing a ROS package to perform the augmented reality exercise. The program will be invoked with the following syntax:

```
$ roslaunch dt_augmented_reality-robot name augmenter.launch map_file:=map file
robot_name:=robot name local:=1
```

where *map file* is a YAML file containing the map (specified in Section 9.5 - Specification of the map).

If *robot name* is not given, it defaults to the hostname.

The program does the following:

1. It loads the intrinsic / extrinsic calibration parameters for the given robot.
2. It reads the map file.
3. It listens to the image topic `/`*robot name*`/camera_node/image/compressed`.
4. It reads each image, projects the map features onto the image, and then writes the resulting image to the topic /![robot name]/AR/![map file basename]/image/compressed

where *map file basename* is the basename of the file without the extension.

We provide you with ROS package template that contains the `AugmentedRealityNode`. By default, launching the `AugmentedRealityNode` should publish raw images from the camera on the new `/`*robot name*`/AR/`*map file basename*`/image/compressed` topic.

In order to complete this exercise, you will have to fill in the missing details of the `Augmenter` class by doing the following:

1. Implement a method called `process_image` that undistorts raw images.
2. Implement a method called `ground2pixel` that transforms points in ground coordinates (i.e. the robot reference frame) to pixels in the image.
3. Implement a method called `callback` that writes the augmented image to the appropriate topic.

## 9.5. Specification of the map

The map file contains a 3D polygon, defined as a list of points and a list of segments that join those points.

The format is similar to any data structure for 3D computer graphics, with a few changes:

1. Points are referred to by name.
2. It is possible to specify a reference frame for each point. (This will help make this into a general tool for debugging various types of problems).

Here is an example of the file contents, hopefully self-explanatory.

The following map file describes 3 points, and two lines.

```
points:
    # define three named points: center, left, right
    center: [axle, [0, 0, 0]] # [reference frame, coordinates]
    left: [axle, [0.5, 0.1, 0]]
    right: [axle, [0.5, -0.1, 0]]
segments:
- points: [center, left]
  color: [rgb, [1, 0, 0]]
- points: [center, right]
  color: [rgb, [1, 0, 0]]
```

### 1) Reference frame specification

The reference frames are defined as follows:

- `axle`: center of the axle; coordinates are 3D.
- `camera`: camera frame; coordinates are 3D.
- `image01`: a reference frame in which 0,0 is top left, and 1,1 is bottom right of the image; coordinates are 2D.

(Other image frames will be introduced later, such as the `world` and `tile` reference frame, which need the knowledge of the location of the robot.)

### 2) Color specification

RGB colors are written as:

```
[rgb, [R, G, B]]
```

where the RGB values are between 0 and 1.

Moreover, we support the following strings:

- `red` is equivalent to `[rgb, [1,0,0]]`
- `green` is equivalent to `[rgb, [0,1,0]]`
- `blue` is equivalent to `[rgb, [0,0,1]]`
- `yellow` is equivalent to `[rgb, [1,1,0]]`
- `magenta` is equivalent to `[rgb, [1,0,1]]`
- `cyan` is equivalent to `[rgb, [0,1,1]]`
- `white` is equivalent to `[rgb, [1,1,1]`
- `black` is equivalent to `[rgb, [0,0,0]]`

## 9.6. “Map” files

### 1) `hud.yaml`

This pattern serves as a simple test that we can draw lines in image coordinates:

```yaml
points:
    TL: [image01, [0, 0]]
    TR: [image01, [0, 1]]
    BR: [image01, [1, 1]]
    BL: [image01, [1, 0]]
segments:
- points: [TL, TR]
  color: red
- points: [TR, BR]
  color: green
- points: [BR, BL]
  color: blue
- points: [BL, TL]
  color: yellow
```

The expected result is to put a border around the image: red on the top, green on the right, blue on the bottom, yellow on the left.

### 2) `calibration_pattern.yaml`

This pattern is based off the checkerboard calibration target used in estimating the intrinsic and extrinsic camera parameters:

```yaml
points:
    TL: [axle, [0.315, 0.093, 0]]
    TR: [axle, [0.315, -0.093, 0]]
    BR: [axle, [0.191, -0.093, 0]]
    BL: [axle, [0.191, 0.093, 0]]
segments:
- points: [TL, TR]
  color: red
- points: [TR, BR]
  color: green
- points: [BR, BL]
  color: blue
- points: [BL, TL]
  color: yellow
```

The expected result is to put a border around the inside corners of the checkerboard: red on the top, green on the right, blue on the bottom, yellow on the left.

### 3) `lane.yaml`

We want something like this:

```
                 Ø
 |   |          | . |             |   |
 |   |          | . |             |   |
 |   |          | . |             |   |
 |   |          | . |             |   |
 |   |          | . |             |   |
 |   |          | . |             |   |
  WW      L       WY      L         WW
 1   2          3   4             5   6
```

Then we have:

```
points:
     p1: [axle, [Ø, Ø.2794, Ø]]
     q1: [axle, [D, Ø.2794, Ø]]
     p2: [axle, [Ø, Ø.2286, Ø]]
     q2: [axle, [D, Ø.2286, Ø]]
     p3: [axle, [Ø, Ø.Ø127, Ø]]
     q3: [axle, [D, Ø.Ø127, Ø]]
     p4: [axle, [Ø, -Ø.Ø127, Ø]]
     q4: [axle, [D, -Ø.Ø127, Ø]]
     p5: [axle, [Ø, -Ø.2286, Ø]]
     q5: [axle, [D, -Ø.2286, Ø]]
     p6: [axle, [Ø, -Ø.2794, Ø]]
     q6: [axle, [D, -Ø.2794, Ø]]
segments:
 - points: [p1, q1]
   color: white
 - points: [p2, q2]
   color: white
 - points: [p3, q3]
   color: yellow
 - points: [p4, q4]
   color: yellow
 - points: [p5, q5]
   color: white
 - points: [p6, q6]
   color: white
```

4) `intersection_4way.yaml`

```
points:
    NL1: [axle, [0.247, 0.295, 0]]
    NL2: [axle, [0.347, 0.301, 0]]
    NL3: [axle, [0.218, 0.256, 0]]
    NL4: [axle, [0.363, 0.251, 0]]
    NL5: [axle, [0.400, 0.287, 0]]
    NL6: [axle, [0.409, 0.513,   0]]
    NL7: [axle, [0.360, 0.314, 0]]
    NL8: [axle, [0.366, 0.456, 0]]
    NC1: [axle, [0.372, 0.007, 0]]
    NC2: [axle, [0.145, 0.008, 0]]
    NC3: [axle, [0.374, -0.0216, 0]]
    NC4: [axle, [0.146, -0.0180, 0]]
    NR1: [axle, [0.209, -0.234, 0]]
    NR2: [axle, [0.349, -0.237, 0]]
    NR3: [axle, [0.242, -0.276, 0]]
    NR4: [axle, [0.319, -0.274, 0]]
    NR5: [axle, [0.402, -0.283, 0]]
    NR6: [axle, [0.401, -0.479, 0]]
    NR7: [axle, [0.352,  -0.415, 0]]
    NR8: [axle, [0.352, -0.303, 0]]
    CL1: [axle, [0.586, 0.261, 0]]
    CL2: [axle, [0.595, 0.632, 0]]
    CL3: [axle, [0.618, 0.251, 0]]
    CL4: [axle, [0.637, 0.662, 0]]
    CR1: [axle, [0.565, -0.253, 0]]
    CR2: [axle, [0.567, -0.607, 0]]
    CR3: [axle, [0.610, -0.262, 0]]
    CR4: [axle, [0.611, -0.641, 0]]
    FL1: [axle, [0.781, 0.718, 0]]
    FL2: [axle, [0.763, 0.253, 0]]
    FL3: [axle, [0.863, 0.192, 0]]
    FL4: [axle, [1.185, 0.172, 0]]
    FL5: [axle, [0.842, 0.718, 0]]
    FL6: [axle, [0.875, 0.271,   0]]
    FL7: [axle, [0.879, 0.234, 0]]
    FL8: [axle, [1.180, 0.209, 0]]
    FC1: [axle, [0.823, 0.0162, 0]]
    FC2: [axle, [1.172, 0.00117, 0]]
    FC3: [axle, [0.845, -0.0100, 0]]
    FC4: [axle, [1.215, -0.0181, 0]]
    FR1: [axle, [0.764, -0.695, 0]]
    FR2: [axle, [0.768, -0.263, 0]]
    FR3: [axle, [0.810, -0.202, 0]]
    FR4: [axle, [1.203, -0.196, 0]]
    FR5: [axle, [0.795, -0.702, 0]]
    FR6: [axle, [0.803, -0.291, 0]]
    FR7: [axle, [0.832, -0.240, 0]]
    FR8: [axle, [1.210, -0.245, 0]]
segments:
- points: [NL1, NL2]
  color: white
- points: [NL3, NL4]
  color: white
```

## 9.7. Suggestions

Start by using the file `hud.yaml`. To visualize it, you do not need the calibration data. It will be helpful to make sure that you can do the easy parts of the exercise: loading the map, and drawing the lines.

## 9.8. Useful APIs

### 1) Loading a map file:

To load a map file, use the function `load_map` provided in `duckietown_utils`:

```
from duckietown_utils import load_map

map_data = load_map(map_filename)
```

(Note that `map` is a reserved symbol name in Python.)

### 2) Reading the calibration data for a robot

To load the *intrinsic* calibration parameters, use the function `load_camera_intrinsics` provided in `duckietown_utils`:

```
from duckietown_utils import load_camera_intrinsics

intrinsics = load_camera_intrinsics(robot_name)
```

To load the *extrinsic* calibration parameters (i.e. ground projection), use the function `load_homography` provided in `duckietown_utils`:

```
from duckietown_utils import load_homography

H = load_homography(robot_name)
```

### 3) Path name manipulation

From a file name like `"/path/to/map1.yaml"`, you can obtain the basename without extension `yaml` by using the function `get_base_name` provided in `duckietown_utils`:

```
from duckietown_utils import get_base_name

filename = "/path/to/map1.yaml"
map_name = get_base_name(filename) # = "map1"
```

### 4) Undistorting an image

To remove the distortion from an image, use the function `rectify` provided in `duckietown_utils`:

```
from duckietown_utils import rectify

rectified_image = rectify(image, intrinsics)
```

### 5) Drawing primitives

To draw the line segments specified in a map file, use the `render_segments` method defined in the `Augmenter` class:

```
class Augmenter():

    # ...

    def ground2pixel(self):
        '''Method that transforms ground points
        to pixel coordinates'''
        # Your code goes here.
        return "???"



image_with_drawn_segments = augmenter.render_segments(image)
```

In order for `render_segments` to draw segments on an image, you must first implement the method `ground2pixel`.

# Unit I-10
# Exercise: Lane Filtering - Particle Filter

Assigned to: Jonathan Michaux, Liam Paull, and Miguel de la Iglesia

## 10.1. Skills learned

- Understanding of basic filtering concepts
- Understanding of a particle filter

## 10.2. Introduction

During the lectures, we have discussed general filtering techniques, and specifically the **histogram filtering** approach that we are using to estimate our location within a lane in Duckietown.

This is a 2-dimensional filter over $d$ and $\theta$, the lateral displacement in the lane and the robot heading relative to the direction of the lane.

In this exercise, we will replace the histrogram filter with a particle filter.

## 10.3. Instructions

Create a ROS node and package that takes as input the list of line segments detected by the line detector, and outputs an estimate of the robot position within the lane to be used by the lane controller. You should be able to run:

```
$ source DUCKIETOWN_ROOT/environment.sh
$ source DUCKIETOWN_ROOT/set_vehicle.name.sh
$ roslaunch dt_filtering_ROBOT_NAME lane_following.launch
```

and then follow the instructions in Unit M-24 - Checkoff: Navigation for trying the lane following demo.

You should definitely look at the existing histogram filter for inspiration.

You may find this a useful resource to get started.

### 1) Workflow Tip

While you are working on your node and it is crashing, you need not kill and relaunch the entire stack (or even launch on your robot). You should build a workflow whereby you can quickly launch only the node you are developing from your laptop.

## 10.4. Submission

Submit the code using location-specific instructions

UNIT I-11

# Exercise: Lane Filtering - Extended Kalman Filter

**Assigned to:** Liam Paull

## 11.1. Skills learned

- Understanding of basic filtering concepts
- Understanding of an Extended Kalman Filter

## 11.2. Introduction

During the lectures, we have discussed general filtering techniques, and specifically the **histogram filtering** approach that we are using to estimate our location within a lane in Duckietown.

This is a 2-dimensional filter over $\boldsymbol{d}$ and $\boldsymbol{\theta}$, the lateral displacement in the lane and the robot heading relative to the direction of the lane.

In this exercise, we will replace the histrogram filter with an Extended Kalman Filter.

## 11.3. Instructions

Create a ROS node and package that takes as input the list of line segments detected by the line detector, and outputs an estimate of the robot position within the lane to be used by the lane controller. You should be able to run:

```
$ source DUCKIETOWN_ROOT/environment.sh
$ source DUCKIETOWN_ROOT/set_vehicle.name.sh
$ roslaunch dt_filtering_ROBOT_NAME lane_following.launch
```

and then follow the instructions in Unit M-24 - Checkoff: Navigation for trying the lane following demo.

You should definitely look at the existing histogram filter for inspiration.

You may find this a useful resource to get started.

### 1) Workflow Tip

While you are working on your node and it is crashing, you need not kill and relaunch the entire stack (or even launch on your robot). You should build a workflow whereby you can quickly launch only the node you are developing from your laptop.

## 11.4. Submission

Submit the code using location-specific instructions

UNIT I-12

# Exercise: Git and conventions

**TODO:** move here the exercises we had last year about branching.

UNIT I-13

# Exercise: Use our API for arguments

## 13.1. Skills learned

- Learn about the command-line API that we have in Duckietown,

## 13.2. Instructions

We have a useful API that makes it easy to create programs with command line arguments.

TODO: to write

## 13.3. Useful new API learned

- Our API for command line arguments.

UNIT I-14

# Exercise: Bouncing ball

## 14.1. Skills learned

- Show how to visualize data on a bag.
- Programmatic generation of images.
- Timestamps generation.

## 14.2. Instructions

Create a program

```
$ bag-bounce --fps fps --speed speed
```

that shows a bouncing ball on the screen, as if it were a billiard

## Useful new API learned

- Our API for command line arguments.

UNIT I-15

# Exercise: Visualizing data on image

## 15.1. Skills learned

- Show how to superimpose data on an image.

## 15.2. Instruction

Write an implementation of `bag-mark-spots`.

## 15.3. Specification for `bag-mark-spots`

Create a program that for each image, finds the pixels that are closest to a certain color, and creates as the output a big red, yellow white spot on them.

```
$ bag-mark-spots --input bag in --mark "[[255,0,0],255,0,0]," --size 5 --output [bag out]
```

UNIT I-16

# Exercise: Make that into a node

## 16.1. Learned skills

- Abstracting code in interfaces that can be reused.
- Launch files.

## 16.2. Instructions

Abstract the analysis above in a way that the same analysis code can be run equally from a bag or on the laptop.

Make a ROS node and two launch files:

- One runs everything on the Duckiebot, and the output is visualized on the laptop.
- One runs the processing on the laptop.

UNIT I-17

# Exercise: Instagram with EasyAlgo interface

## 17.1. Learned skills

- Use of our Duckietown API for abstracting algorithms.

## 17.2. Instructions

**TODO:** Do the above using our API for filters.

We define the interface `InstagramFilter` and the EasyAlgo configuration files.

## 17.3. See documentation

**TODO:** pointer to EasyAlgo

## 17.4. Use in your code

Write a node using the EasyNode framework that decides which filters to run given the configuration.

**TODO:** Also introduce the DUCKIETOWN_CONFIG_SEQUENCE.

UNIT I-18

# Milestone: Illumination invariance (anti-instagram)

**TODO:** Make them run our code, and also visualize what's going on

UNIT I-19

# Exercise: Launch files basics

## 19.1. Learned skills

- Launch files

UNIT I-20

# Exercise: Unit tests

## 20.1. Learned skills

- Write unit tests that can be integrated in our framework.

## 20.2. Unit tests with nosetests

- Unit tests

## 20.3. Unite tests with ROS tests

- Integration with ROS tests

## 20.4. Unite tests with comptests

UNIT I-21

# Exercise: Parameters (standard ROS api)

## 21.1. Learned skills

- Reading parameters
- Dynamic modification of parameters

UNIT I-22

# Exercise: Parameters (our API)

## 22.1. Learned skills

- Use Duckietown API

UNIT I-23

# Exercise: Place recognition abstraction

## 23.1. Learned skills

- ...

## 23.2. Instructions

We use the following interface:

```
class FeatureDescription():

    def feature(self, image):
        ''''' returns a ''feature description'' '''''

    def feature_compare(feature1, feature2):
        pass
```

We also provide a basic skeleton.

## 23.3. Try a basic feature:

Simplest feature: average color.

```
class AverageColor(FeatureDescription):

    def feature(self, image):
        return np.mean(image)

    def feature_mismatch(f1, f2)
        return np.abs(f1-f2)
```

Compute the similarity matrix for the

```
$ similarity --feature average_color --input input.bag  --output dirname
```

## 23.4. Bottom line

TODO: to write

UNIT I-24

# Exercise: Parallel processing

## 24.1. Learned skills

- Do things faster in parallel.

## 24.2. Instructions

We introduce the support for parallel processing that we have in the APIs.

UNIT I-25

# Exercise: Adding new test to integration tests

TODO: to write

## 25.1. Learned skills

- Do things faster in parallel.

## 25.2. Instructions

TODO: to write

## 25.3. Bottom line

TODO: to write

## 25.4. Milestone: Lane following

TODO: to write

UNIT I-26

# Exercise: localization

TODO: to write

UNIT I-27

# Exercise: Ducks in a row

TODO: to write

UNIT I-28

# Exercise: Comparison of PID

TODO: to write

UNIT I-29

# Exercise: RRT

TODO: to write

# UNIT I-30
# Exercise: Who watches the watchmen? (optional)

## 30.1. Skills learned

- Awareness that this is a losing game.

## 30.2. Instructions

A good exercise is writing `image-ops-tester` yourself.

However, we already gave you a copy of `image-ops-tester`, which you used in the previous step, so there is not much of a challenge. So, let's go one level up, and consider...

## 30.3. `image-ops-tester-tester` specification

Write a program `image-ops-tester-tester` that tests whether a program conforms to the specification of a `image-ops-tester` given in Section 2.5 - Testing it works with `image-ops-tester`.

The `image-ops-tester-tester` program is called as follows:

```
$ image-ops-tester-tester candidate-image-ops-tester
```

This must return:

- 0 if the candidate conforms to the specification;
- 1 if it doesn't;
- another error code if other errors arise.

## 30.4. Testing it works with `image-ops-tester-tester-tester`

We provide you with a helpful program called `image-ops-tester-tester-tester` that makes sure that a candidate script conforms to the specification of an `image-ops-tester-tester`.

Use it as follows:

```
$ image-ops-tester-tester-tester candidate image-ops-tester-tester
```

This should return 0 if everything is ok, or different than 0 otherwise.

## Bottom line

Even if things are tested, you can never be sure that the tests themselves work.

# Part J
# Software reference

This part describes things that you should know about UNIX/Linux environments.

Documentation writers: please make sure that every command used has a section in these chapters.

# Unit J-1
# Ubuntu packaging with APT

## 1.1. `apt install`

TODO: to write

## 1.2. `apt update`

TODO: to write

### 1) `apt dist-upgrade`

TODO: hold back packages

## 1.3. `apt-key`

TODO: to write

## 1.4. `apt-mark`

TODO: to write

## 1.5. `apt-get`

TODO: to write

## 1.6. `add-apt-repository`

TODO: to write

## 1.7. `wajig`

TODO: to write

## 1.8. `dpigs`

TODO: to write

UNIT J-2

# GNU/Linux general notions

**Assigned to:** Andrea

## 2.1. Background reading

- UNIX
- Linux
- free software; open source software.

# Unit J-3
# Every day Linux

## 3.1. man

This is an interface to the on-line reference manuals. Whenever you meet some unfamiliar commands, try use `man certain_command` before Googling. You will find it extremely clear, useful and self-contained.

## 3.2. cd

Go to the directory you want. If you just use:

```
$ cd
```

then you will go to your home directory, i.e., ~

## 3.3. sudo

Whenever you want to modify system files, you will need `sudo`. Commonly touched system files including `/etc`, `/opt` and so on. Since most of you have followed the guideline to use passwordless sudo, I would recommend that make sure what you are doing with sudo before you execute the command, otherwise you may need to reinstall the system.

## 3.4. ls

List all the files and documents in the current directory. From `~/.bashrc`, we know some commonly used alias. See more by `man ls`.

– `la` for `ls -a` which will list out all files and documents including the hidden ones(whose name starts with a dot).

– `ll` for `ls -l` which will display Unix file types, permissions, number of hard links, owner, group, size, last-modified date and filename.

## 3.5. cp

`cp fileA directoryB` will copy the file A to directory B. See more by executing `man cp`.

## 3.6. mkdir

Make new directory. See more by `man mkdir`.

## 3.7. touch

Update the access and modification times of the input file to current time. See more by `man touch`.

### 3.8. reboot

This command must be executed as root. `sudo` required. This will reboot your laptop or Raspberry Pi. See more by `man reboot`.

### 3.9. shutdown

This command requires `sudo`. You can set a countdown to shutdown you machine. More by `man shutdown`.

### 3.10. rm

Remove certain file. `rm -r` will remove files. More in `man rm`.

# Unit J-4
# Users

## 4.1. `passwd`

Update password of the current user. Old password needed.

UNIT J-5

# UNIX tools

## 5.1. `cat`

Cat some file will return you the content. More in `man cat`.

## 5.2. `tee`

Read from standard input and write to standard output and files. More on `man tee`.

## 5.3. `truncate`

TODO: to write

UNIT J-6

# Linux disks and files

## 6.1. `fdisk`

TODO: to write

## 6.2. `mount`

TODO: to write

## 6.3. `umount`

TODO: to write

## 6.4. `losetup`

TODO: to write

## 6.5. `gparted`

TODO: to write

## 6.6. `dd`

TODO: to write

## 6.7. `sync`

TODO: to write

## 6.8. `df`

TODO: to write

## 6.9. How to make a partition

TODO: to write

UNIT J-7

# Other administration commands

## 7.1. `visudo`

TODO: to write

## 7.2. `update-alternatives`

TODO: to write

## 7.3. `udevadm`

TODO: to write

## 7.4. `systemctl`

TODO: to write

# Unit J-8
# Make

## 8.1. `make`

TODO: to write

UNIT J-9

# Python-related tools

## 9.1. `virtualenv`

TODO: to write

## 9.2. `pip`

TODO: to write

UNIT J-10

# Raspberry-PI commands

## 10.1. raspi-config

TODO: to write

## 10.2. vcgencmd

TODO: to write

## 10.3. raspistill

TODO: to write

## 10.4. jstest

TODO: to write

## 10.5. swapon

TODO: to write

## 10.6. mkswap

TODO: to write

UNIT J-11

# Users and permissions

## 11.1. `chmod`

TODO: to write

## 11.2. `groups`

TODO: to write

## 11.3. `adduser`

TODO: to write

## 11.4. `useradd`

TODO: to write

# Unit J-12
# Downloading

## 12.1. `curl`

TODO: to write

## 12.2. `wget`

TODO: to write

## 12.3. `sha256sum`

TODO: to write

## 12.4. `xz`

TODO: to write

UNIT J-13

# Shells and environments

## 13.1. `source`

You can only do `source file_name` if the file can be executed by bash.

## 13.2. `which`

Tell you the /bin/ directory of your command. This is useful to distinguish which python you are using if you have virtualenv.

## 13.3. `export`

TODO: to write

# Unit J-14
# Other misc commands

## 14.1. pgrep

TODO: to write

## 14.2. npm

TODO: to write

## 14.3. nodejs

TODO: to write

## 14.4. ntpdate

TODO: to write

## 14.5. chsh

TODO: to write

## 14.6. echo

TODO: to write

## 14.7. sh

TODO: to write

## 14.8. fc-cache

TODO: to write

UNIT J-15

# Mounting USB drives

First **plug in the USB drive** nothing will work if you don't do that first. Now ssh into your robot. On the command line type:

```
$ lsusb
```

you should see your Sandisk USB drive as an entry. Congrats, you correctly plugged it in

```
$ lsblk
```

Under name you should see `sda1`, with size about 28.7GB and nothing under the `MOUNTPOINT` column (if you see something under `MOUNTPOINT` congrats you are done.

Next make a directory to mount to:

```
$ sudo mkdir /media/logs
```

Next mount the drive

```
$ sudo mount -t vfat /dev/sda1 /media/logs -o umask=000
```

Test by running `lsblk` again and you should now see `/media/logs` under `MOUNTPOINT`

## 15.1. Unmounting a USB drive

```
$ sudo umount /media/logs
```

# Unit J-16
# Linux resources usage

## 16.1. Measuring CPU usage using `htop`

You can use `htop` to monitor CPU usage.

```
$ sudo apt install htop
```

TODO: to write

## 16.2. Measuring I/O usage using `iotop`

Install using:

```
$ sudo apt install iotop
```

TODO: to write

## 16.3. How fast is the SD card?

➡ Section 17.1 - Testing SD Card and disk speed.

UNIT J-17

# SD Cards tools

## 17.1. Testing SD Card and disk speed

Test SD Card (or any disk) speed using the following commands, which write to a file called *filename*.

```
$ dd if=/dev/zero of=filename bs=500K count=1024
$ sync
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
$ dd if=filename of=/dev/null bs=500K count=1024
$ rm filename
```

Note the `sync` and the `echo` command are very important.

Example results:

```
524288000 bytes (524 MB, 500 MiB) copied, 30.2087 s, 17.4 MB/s
524288000 bytes (524 MB, 500 MiB) copied, 23.3568 s, 22.4 MB/s
```

That is write 17.4 MB/s, read 22 MB/s.

## 17.2. How to burn an image to an SD card

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** A blank SD card.
**Requires:** An image file to burn.
**Requires:** An Ubuntu computer with an SD reader.
**Results:** A burned image.

### 1) Finding your device name for the SD card

First, find out what is the device name for the SD card.

Insert the SD Card in the slot.

Run the command:

```
$ sudo fdisk -l
```

Find your device name, by looking at the sizes.

For example, the output might contain:

```
Disk /dev/mmcblk0: 14.9 GiB, 15931539456 bytes, 31116288 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

In this case, the device is `/dev/mmcblk0`. That will be the *device* in the next commands.

You may see `/dev/mmcblk0pX` or a couple of similar entries for each partition on the card, where `X` is the partition number. If you don't see anything like that, take out the SD card and run the command again and see what disappeared.

### 2) Unmount partitions

Before proceeding, unmount all partitions.

Warning: Before unmounting partitions, make sure you found the correct device in the previous step. In particular, `/dev/sda/` is the hard drive of your computer, and if you unmount its partitions, you can erase important data, including important files and the operating system you are not currently using.

Run `df -h`. If there are partitions like `/dev/mmcblk0pn`, then unmount each of them. For example:

```
$ sudo umount /dev/mmcblk0p1
$ sudo umount /dev/mmcblk0p2
```

### 3) Burn the image

Now that you know that the device is *device*, you can burn the image to disk.

Let the image file be *image file*.

Burn the image using the command `dd`:

```
$ sudo dd of=device if=image file status=progress bs=4M
```

Note: Use the name of the device, without partitions. i.e., `/dev/mmcblk0`, not `/dev/mmcblk0pX`.

Note: dd comand with status=progress parameter only work for dd –version 8.24 ubuntu16.04.2

## 17.3. How to shrink an image

KNOWLEDGE AND ACTIVITY GRAPH

Requires: An image file to burn.

Requires: An Ubuntu computer.

Results: A shrunk image.

Note: Majority of content taken from here

We are going to use the tool `gparted` so make sure it's installed

```
$ sudo apt install gparted
```

Let the image file be *image file* and its name be *imagename*. Run the command:

```
$ sudo fdisk -l image file
```

It should give you something like:

```
Device                        Boot  Start      End  Sectors  Size Id Type
duckiebot-RPI3-LP-aug15.img1         2048   131071   129024   63M  c W95 FAT32 (LBA)
duckiebot-RPI3-LP-aug15.img2       131072 21219327 21088256 10.1G 83 Linux
```

Take note of the start of the Linux partition (in our case 131072), let's call it *start*. Now we are going to mount the Linux partition from the image:

```
$ sudo losetup /dev/loop0 imagename.img -o $((start*512))
```

and then run `gparted`:

```
$ sudo gparted /dev/loop0
```

In `gparted` click on the partition and click "Resize" under the "Partition" menu. Resize drag the arrow or enter a size that is equal to the minimum size plus 20MB

**Note:** This didn't work well for me - I had to add much more than 20MB for it to work.

Click the "Apply" check mark. *Before* closing the final screen click through the arrows in the dialogue box to find a line such a "`resize2fs -p /dev/loop0 1410048K`". Take note of the new size of your partition. Let's call it *new size*.

Now remove the loopback on the second partition and setup a loopback on the whole image and run `fdisk`:

```
$ sudo losetup -d /dev/loop0
$ sudo losetup /dev/loop0 image file
$ sudo fdisk /dev/loop0

Command (m for help): enter d
Partition number (1,2, default 2): enter 2
Command (m for help): enter n
Partition type
p   primary (1 primary, 0 extended, 3 free)
e   extended (container for logical partitions)
Select (default p): enter p
Partition number (2-4, default 2): enter 2
First sector (131072-62521343, default 131072): start
Last sector, +sectors or +size{K,M,G,T,P} (131072-62521343, default 62521343): +new sizeK
```

**Note:** on the last line include the `+` and the `K` as part of the size.

```
Created a new partition 2 of type 'Linux' and of size 10.1 GiB.

Command (m for help): enter w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Re-reading the partition table failed.: Invalid argument

The kernel still uses the old table. The new table will be used at the next reboot or after
you run partprobe(8) or kpartx(8).
```

Disregard the final error.

You partition has now been resized and the partition table has been updated. Now we will remove the loopback and then truncate the end of the image file:

```
$ fdisk -l /dev/loop0
```

```
Device       Boot  Start      End  Sectors  Size Id Type
/dev/loop0p1        2048   131071   129024   63M  c W95 FAT32 (LBA)
/dev/loop0p2      131072 21219327 21088256 10.1G 83 Linux
```

Note down the end of the second partition (in this case 21219327). Call this *end*.

```
$ sudo losetup -d /dev/loop0
$ sudo truncate -s $(((end+1)*512)) image file
```

You now have a shrunken image file.

It might be useful to compress it, before distribution:

```
$ xz image file
```

UNIT J-18

# Networking tools

> **Assigned to:** Andrea

KNOWLEDGE AND ACTIVITY GRAPH

Preliminary reading:

- Basics of networking, including
  - what are IP addresses
  - what are subnets
  - how DNS works
  - how `.local` names work
  - ...

➡ XXX (ref to find).

TODO: to write

Make sure that you know:

## 18.1. `hostname`

TODO: to write

## 18.2. Visualizing information about the network

### 1) `ping`: are you there?

TODO: to write

### 2) `ifconfig`

TODO: to write

```
$ ifconfig
```

# Unit J-19
# Accessing computers using SSH

> Assigned to: Andrea

## 19.1. Background reading

TODO: to write

- Encryption
- Public key authentication

## 19.2. Installation of SSH

This installs the client:

```
$ sudo apt install ssh
```

This installs the server:

TODO: to write

This enables the server:

TODO: to write

## 19.3. Local configuration

The SSH configuration as a client is in the file

`~/.ssh/config`

Create the directory with the right permissions:

```
$ mkdir ~/.ssh
$ chmod 0700 ~/.ssh
```

Edit the file:

`~/.ssh/config`

(We suggest you use VIM to edit files; see a tutorial here.)

+ comment
laptop or duckiebot? - LP

Then add the following lines:

```
HostKeyAlgorithms ssh-rsa
```

The reason is that Paramiko, used by `roslaunch`, does not support the ECSDA keys.

## 19.4. How to login with SSH and a password

To log in to a remote computer *remote* with user *remote-user*, use:

```
$ ssh remote-user@remote
```

### 1) Troubleshooting

**Symptom**: "Offending key error".

If you get something like this:

```
Warning: the ECDSA host key for ... differs from the key for the IP address '... '

Offending key for IP in /home/user/.ssh/known_hosts:line
```

then remove line *line* in `~/.ssh/known_hosts`.

## 19.5. Creating an SSH keypair

This is a step that you will repeat twice: once on the Duckiebot, and once on your laptop.

The program will prompt you for the filename on which to save the file.

Use the convention

```
/home/username/.ssh/username@host name
/home/username/.ssh/username@host name.pub
```

where:

- *username* is the current user name that you are using (`ubuntu` or your chosen one);
- *host name* is the name of the host (the Duckiebot or laptop);

An SSH key can be generated with the command:

```
$ ssh-keygen -h
```

The session output will look something like this:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/username/.ssh/id_rsa):
```

At this point, tell it to choose this file:

```
/home/username/.ssh/username@host name
```

Then:

```
Enter passphrase (empty for no passphrase):
```

Press enter; you want an empty passphrase.

```
Enter same passphrase again:
```

Press enter.

```
Your identification has been saved in /home/username/.ssh/username@host name
Your public key has been saved in /home/username/.ssh/username@host name.pub
The key fingerprint is:
XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX  username@host name
The key's randomart image is:
+--[ RSA 2048]----+
|            .    |
|       o   o  .  |
|      o = o  . o |
|       B . .  * o|
|        S o    O |
|         o o  . E|
|          o o  o |
|           o  +  |
|            .. . |
+-----------------+
```

Note that the program created two files.
The file that contains the private key is

```
/home/username/.ssh/username@host name
```

The file that contains the public key has extension `.pub`:

```
/home/username/.ssh/username@host name.pub
```

Next, tell SSH that you want to use this key.
Make sure that the file `~/.ssh/config` exists:

```
$ touch ~/.ssh/config
```

Add a line containing

```
IdentityFile ~/.ssh/PRIVATE_KEY_FILE
```

(using the filename for the private key).

+ comment
make sure to include the full path to the file, not just the filename.

Check that the config file is correct:

```
$ cat ~/.ssh/config
...
IdentityFile ~/.ssh/PRIVATE_KEY_FILE
...
```

To copy the generated SSH key to the clipboard xclip can be used (Installation of xclip if necessary).

```
$ sudo apt-get install xclip
$ xclip -sel clip < ~/.ssh/username@host name.pub
```

## 19.6. How to login without a password

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** You have two computers, called "*local*" and "*remote*", with users "*local-user*" and "*remote-user*". Here, we assume that *local* and *remote* are complete hostnames (such as `duckiebot.local`.).

**Requires:** The two computers are on the same network and they can ping each other.

**Requires:** You have created a keypair for *local-user* on *local*. This procedure is described in Section 19.5 - Creating an SSH keypair.

**Results:** From the *local* computer, *local-user* will be able to log in to *remote* computer without a password.

First, connect the two computers to the same network, and make sure that you can ping *remote* from *local*:

```
local $ ping remote.local
```

Do not continue if you cannot do this successfully.

If you have created a keypair for *local-user*, you will have a public key in this file on the *local* computer:

```
/home/local-user/.ssh/local-user@local.pub
```

This file is in the form:

```
ssh-rsa long list of letters and numbers local-user@local
```

You will have to copy the contents of this file on the *remote* computer, to tell it that this key is authorized.

On the *local* computer, run the command:

```
local $ ssh-copy-id remote-user@remote
```

now you should be able to login to the remote without a password:

```
local $ ssh remote-user@remote
```

This should succeed, and you should not be asked for a password.

## 19.7. Fixing SSH Permissions

Sometimes, SSH does not work because you have the wrong permissions on some files. In doubt, these lines fix the permissions for your `.ssh` directory.

```
$ chmod 0700 ~/.ssh
$ chmod 0700 ~/.ssh/*
```

## 19.8. ssh-keygen

TODO: to write

UNIT J-20

# Wireless networking in Linux

## 20.1. iwconfig

TODO: to write

## 20.2. iwlist

### 1) Getting a list of WiFi networks

What wireless networks do I have around?

```
$ sudo iwlist interface scan | grep SSID
```

### 2) Do I have 5 GHz?

Does the interface support 5 GHz channels?

```
$ sudo iwlist interface freq
```

Example output:

```
wlx74da38c9caa0  20 channels in total; available frequencies :
  Channel 01 : 2.412 GHz
  Channel 02 : 2.417 GHz
  Channel 03 : 2.422 GHz
  Channel 04 : 2.427 GHz
  Channel 05 : 2.432 GHz
  Channel 06 : 2.437 GHz
  Channel 07 : 2.442 GHz
  Channel 08 : 2.447 GHz
  Channel 09 : 2.452 GHz
  Channel 10 : 2.457 GHz
  Channel 11 : 2.462 GHz
  Channel 36 : 5.18 GHz
  Channel 40 : 5.2 GHz
  Channel 44 : 5.22 GHz
  Channel 48 : 5.24 GHz
  Channel 149 : 5.745 GHz
  Channel 153 : 5.765 GHz
  Channel 157 : 5.785 GHz
  Channel 161 : 5.805 GHz
  Channel 165 : 5.825 GHz
  Current Frequency:2.437 GHz (Channel 6)
```

Note that in this example only *some* 5Ghz channels are supported (36, 40, 44, 48, 149, 153, 157, 161, 165); for example, channel 38, 42, 50 are not supported. This means that you need to set up the router not to use those channels.

UNIT J-21

# Moving files between computers

## 21.1. scp

TODO: to write

### 1) Download a file with SCP

Use this command:

```
$ scp hostname:/path/to/out.jpg .
```

to download `out.jpg` to your current directory.

## 21.2. rsync

TODO: to write

# Unit J-22
# VIM

> Assigned to: Andrea

To do quick changes to files, especially when logged remotely, we suggest you use the VI editor, or more precisely, VIM ("VI iMproved").

## 22.1. External documentation

➡ A VIM tutorial.

## 22.2. Installation

Install like this:

```
$ sudo apt install vim
```

## 22.3. vi

TODO: to write

## 22.4. Suggested configuration

Suggested `~/.vimrc`:

```
syntax on
set number
filetype plugin indent on
highlight Comment ctermfg=Gray
autocmd FileType python set complete isk+=.,(
```

## 22.5. Visual mode

TODO: to write

## 22.6. Indenting using VIM

Use the [>] command to indent.

To indent 5 lines, use [5] [>] [>].

To mark a block of lines and indent it, use [V].

For example, use [V] [J] [J] [>] to indent 3 lines.

Use [<] to dedent.

# Unit J-23
# Atom

## 23.1. Install Atom

Following the instructions here:

```
$ sudo add-apt-repository ppa:webupd8team/atom
$ sudo apt update
$ sudo apt install atom
```

## 23.2. Using Atom to code remotely

With Atom, you are able to remotely code on files located on your Duckiebot with a GUI. The benefit of using Atom is that you are able to install extensions such as an IDE for Python, a Markdown previewer, or just use custom themes to avoid coding in the terminal.

Follow these instructions:

Install remote-atom

```
$ sudo apm install remote-atom
```

Now, we need to edit our SSH config so that any data send to the port 52698, which is the port remote-atom is using, is forwarded via SSH to our local machine. Edit the file "~/.ssh/config". There, you add "RemoteForward 52698 127.0.0.1:52698" to your host. The resulting host will look similar to

```
Host lex
    User julien
    Hostname lex.local
    RemoteForward 52698 127.0.0.1:52698
```

Now, we need to connect to our duckiebot via SSH and install rmate (and simultaniously rename it to ratom)

```
$ sudo wget -O /usr/local/bin/ratom https://raw.github.com/aurora/rmate/master/rmate
$ sudo chmod +x /usr/local/bin/ratom
```

Now, you just need to launch Atom on your local machine, go to Packages->Remote Atom->Start Server.

You can now edit a file in a terminal connected to your duckiebot via SSH by typing

```
$ sudo ratom filename
```

And atom will automatically open the file on your local machine. In the settings of remote-atom, you can also set the package to start the server automatically on launching atom on your local machine.

# Unit J-24
# Liclipse

## 24.1. Why using IDEs

To be productive in coding, you need to have a proper IDE.

Look at how quickly you can create Python files by using an IDE (Figure 24.1). In this case, the editor is importing the required symbols. Think about how long would it take to do it without an IDE.

Figure 24.1. Eclipse editor capabilities

## 24.2. Other alternatives

In addition to LiClipse, the one that is suggested for this class, there exist:

- PyCharm
- Eclipse

## 24.3. Installing LiClipse

Follow the instructions at this page.

At the moment of writing, these are:

```
$ wget http://www.mediafire.com/file/rwc4bk3nthtxcvv/liclipse_4.1.1_linux.gtk.x86_64.tar.gz
$ tar xvzf liclipse_4.1.1_linux.gtk.x86_64.tar.gz
$ sudo ln -s `pwd`/liclipse/LiClipse /usr/bin/liclipse
```

Now you can run it using `liclipse`:

```
$ liclipse
```

When it runs for the first time, choose “use this as default” and click “launch”.

Choose “Import-> General -> Existing project into workspace”. Select the folder `~/duckietown`.

+ comment

Only Import -> General -> Projects from Folder or Archive, selecting `~/duckuments` worked for me. JT

+ comment
This is not about the duckuments, it's for `duckietown` - AC

If it asks about interpreters, select "auto config".

When it shows "uncheck settings that should not be changed", just click OK.

## 24.4. Set shortcuts for LiClipse

Go to "window/preferences/General/Keys".

Find "print", and unbind it.

Find "Quick switch editor", and set it to Ctrl-P. (This is now the same as Atom.)

Find "Previous tab" and assign Ctrl-Shift-[ (This is now the same as Atom.)

Find "Next tab" and assign Ctrl-Shift-]. (This is now the same as Atom.)

Find "Show in (PyDev package explorer)" and assign Ctrl-Shift-M.

## 24.5. Shortcuts for LiClipse

Make sure that you can do the following tasks:

- Use the global browser: Press Cmd-Shift-T, type "what". It should autocomplete to `what_the_duck`. Press enter; it should jump to the file.
- Switch among open editors with Ctrl-P.
- Switch between tabs with Ctrl-Shift-], -[.
- See the current file in the directory, using Cmd-Shift-M.

TABLE 24.1. LICLIPSE COMMANDS

| On Linux | On Mac | |
|---|---|---|
| Ctrl-Shift-T | Cmd-Shift-T | Globals browser |
| Ctrl-P | Cmd-P | Quick editor switch |
| Ctrl-Shift-[ | Cmd-Shift-[ | Previous tab (needs to be configured) |
| Ctrl-Shift-] | Cmd-Shift-] | Next tab (needs to be configured) |
| Ctrl-Shift-M | Cmd-Shift-M | Show in (PyDev package explorer) |
| Ctrl-1 | Cmd-1 | Find symbol |

## 24.6. Other configuration for Liclipse

From the "Preferences" section, it's suggested to:

- Get rid of the minimap on the right.
- Get rid of spellchecking.

Then, there is the issue of "code completion". This is a love-it-or-hate-it issue. The choice is yours.

# Unit J-25
# Slack

## 25.1. Installing the Slack app on Linux

TODO: to write

## 25.2. Disabling Slack email notifications

Most importantly, please take the time to disable email notification for Slack.

The point of Slack is that you don't get email. You can work on Duckietown only when you have the time to do so.

TODO: to write how

## 25.3. Disabling Slack pop-up notification on the desktop

Also remove pop up notifications from the app. It should be a discrete notification that says "hey, when you have some time, look at Twist", not "HEY HEY HEY PAY ATTENTION TO ME NOW".

TODO: to write procedure

UNIT J-26

# Byobu

**Assigned to:** Andrea

You need to learn to use Byobu. It will save you much time later.
(Alternatives such as GNU Screen are fine as well.)

## 26.1. Advantages of using Byobu

**TODO:** To write

## 26.2. Installation

On Ubuntu, install using:

```
$ sudo apt install byobu
```

## 26.3. Documentation

☀ See the screencast on the website http://byobu.co/.

## 26.4. Quick command reference

You can change the escape sequence from Ctrl-A to something else by using the configuration tool that appears when you type F9.

Commands to use windows:

TABLE 26.1. WINDOWS

| | Using function keys | Using escape sequences |
|---|---|---|
| Create new window | F2 | Ctrl-A then C |
| Previous window | F3 | |
| Next window | F4 | |
| Switch to window | | Ctrl-A then a number |
| Close window | Ctrl-F6 | |
| Rename window | | Ctrl-A then , |

Commands to use panes (windows split in two or more):

TABLE 26.2. COMMANDS FOR PANES

| | Using function keys | Using escape sequences |
|---|---|---|
| Split horizontally | Shift-F2 | Ctrl-A then \| |
| Split vertically | Ctrl-F2 | Ctrl-A then % |
| Switch focus among panes | Ctrl-↑↓←→ | Ctrl-A then one of ↑↓←→ |
| Break pane | | Ctrl-A then ! |

Other commands:

TABLE 26.3. OTHER

| | Using function keys | Using escape sequences |
|---|---|---|
| Help | | Ctrl-A then ? |
| Detach | | Ctrl-A then D |

## 26.5. Commands on OS X

Scroll up and down using fn option ↑ and fn option ↓.

Highlight using alt

UNIT J-27

# Source code control with Git

> **Assigned to:** Andrea

## 27.1. Background reading

- Good book
- Git Flow

## 27.2. Installation

The basic Git program is installed using

```
$ sudo apt install git
```

Additional utilities for `git` are installed using:

```
$ sudo apt install git-extras
```

This include the `git-ignore` utility.

## 27.3. Setting up global configurations for Git

This should be done twice, once on the laptop, and later, on the robot.
These options tell Git who you are:

```
$ git config --global user.email "email"
$ git config --global user.name  "full name"
```

Also do this, and it doesn't matter if you don't know what it is:

```
$ git config --global push.default simple
```

## 27.4. Git tips

### 1) Delete branches

Delete local:

```
$ git branch -d branch-name
```

Delete remote:

```
$ git push origin --delete branch-name
```

Propagate on other machines by doing:

```
$ git fetch --all --prune
```

### 2) Shallow clone

You can clone without history with the command:

```
$ git clone --depth 1 repository URL
```

## 27.5. Git troubleshooting

### 1) Problem 1: https instead of ssh:

The symptom is:

```
$ git push
Username for 'https://github.com':
```

Diagnosis: the `remote` is not correct.
If you do `git remote` you get entries with `https:` :

```
$ git remote -v
origin  https://github.com/duckietown/Software.git (fetch)
origin  https://github.com/duckietown/Software.git (push)
```

Expectation:

```
$ git remote -v
origin  git@github.com:duckietown/Software.git (fetch)
origin  git@github.com:duckietown/Software.git (push)
```

Solution:

```
$ git remote remove origin
$ git remote add origin git@github.com:duckietown/Software.git
```

### 2) Problem 1: `git` push complains about upstream

The symptom is:

```
fatal: The current branch branch name has no upstream branch.
```

Solution:

```
$ git push --set-upstream origin branch name
```

## 27.6. `git`

TODO: to write

## 27.7. `hub`

### 1) Installation

Install `hub` using the instructions.

### 2) Creating pull requests

You can create a pull request using:

```
$ hub pull-request -m "description"
```

# Unit J-28
# Git LFS

This describes Git LFS.

## 28.1. Generic installation instructions

See instructions at:

https://git-lfs.github.com/

## 28.2. Ubuntu 16 installation (laptop)

Following these instructions, run the following:

```
$ sudo add-apt-repository ppa:git-core/ppa
$ curl -s https://packagecloud.io/install/repositories/github/git-lfs/script.deb.sh | sudo bash
$ sudo apt update
$ sudo apt install git-lfs
```

## 28.3. Raspberry Pi 3

> **Note:** unresolved issues.

The instructions above do not work.

Following this, the error that appears is that golang on the Pi is 1.6 instead it should be 1.7.

## 28.4. Troubleshooting

> **Symptom:** The binary files are not downloaded. In their place, there are short "pointer" files.

If you have installed LFS after pulling the repository and you see only the pointer files, do:

```
$ git lfs pull --all
```

UNIT J-29

# Setup Github access

**Assigned to:** Andrea

This chapter describes how to create a Github account and setup SSH on the robot and on the laptop.

## 29.1. Create a Github account

Our example account is the following:

```
Github name: greta-p
E-mail: greta-p@duckietown.com
```

Create a Github account (Figure 29.1).

Figure 29.1

Go to your inbox and verify the email.

## 29.2. Become a member of the Duckietown organization

Give the administrators your account name. They will invite you.

Accept the invitation to join the organization that you will find in your email.

## 29.3. Add a public key to Github

You will do this procedure twice: once for the public key created on the laptop, and later with the public key created on the robot.

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** A public/private keypair already created and configured. This procedure is explained in Section 19.5 - Creating an SSH keypair.

**Results:** You can access Github using the key provided.

+ comment

Since I am not as familiar with Linux an VIM it would have been great to say how we can get access to the public key: sudo vim /home/username/.ssh/username@host name.pub and than copy it and add it on github SL

Go to settings (Figure 29.2).

Figure 29.2

Add the public key that you created:

Figure 29.3

Figure 29.4

Figure 29.5

To check that all of this works, use the command

```
$ ssh -T git@github.com
```

The command tries to connect to Github using the private keys that you specified. This is the expected output:

```
Warning: Permanently added the RSA host key for IP address 'ip address' to the list of known
hosts.
Hi username! You've successfully authenticated, but GitHub does not provide shell access.
```

If you don't see the greeting, stop.

Repeat what you just did for the Duckiebot on the laptop as well, making sure to

change the name of the file containing the private key.

# Unit J-30
# ROS installation and reference

> Assigned to: Liam

## 30.1. Install ROS

This part installs ROS. You will run this twice, once on the laptop, once on the robot.
The first commands are copied from [this page](#).
Tell Ubuntu where to find ROS:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Tell Ubuntu that you trust the ROS people (they are nice folks):

```
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116
```

Fetch the ROS repo:

```
$ sudo apt update
```

Now install the mega-package `ros-kinetic-desktop-full`.

```
$ sudo apt install ros-kinetic-desktop-full
```

There's more to install:

```
$ sudo apt install ros-kinetic-{tf-conversions,cv-bridge,image-transport,camera-info-manager,theora-image-transport,joy,image-proc,c
```

> Note: Do not install packages by the name of `ros-X`, only those by the name of `ros-kinetic-X`. The packages `ros-X` are from another version of ROS.

> XXX: not done in aug20 image:

Initialize ROS:

```
$ sudo rosdep init
$ rosdep update
```

## 30.2. `rqt_console`

TODO: to write

## 30.3. roslaunch

TODO: to write

## 30.4. rviz

TODO: to write

## 30.5. rostopic

TODO: to write

### 1) rostopic hz

TODO: to write

### 2) rostopic echo

TODO: to write

## 30.6. catkin_make

TODO: to write

## 30.7. rosrun

TODO: to write

## 30.8. rostest

TODO: to write

## 30.9. rospack

TODO: to write

## 30.10. rosparam

TODO: to write

## 30.11. rosdep

TODO: to write

## 30.12. roswtf

TODO: to write

## 30.13. rosbag

A bag is a file format in ROS for storing ROS message data. Bags, so named because of their .bag extension, have an important role in ROS. Bags are typically created by a tool like `rosbag`, which subscribe to one or more ROS topics, and store the serialized message data in a file as it is received. These bag files can also be played back in ROS to the same topics they were recorded from, or even remapped to new topics.

### 1) rosbag record

The command `rosbag record` records a bag file with the contents of specified topics.

### 2) rosbag info

The command `rosbag info` summarizes the contents of a bag file.

### 3) rosbag play

The command `rosbag play` plays back the contents of one or more bag files.

### 4) rosbag check

The command `rosbag check` determines whether a bag is playable in the current system, or if it can be migrated.

### 5) rosbag fix

The command `rosbag fix` repairs the messages in a bag file so that it can be played in the current system.

### 6) rosbag filter

The command `rosbag filter` converts a bag file using Python expressions.

### 7) rosbag compress

The command `rosbag compress` compresses one or more bag files.

### 8) rosbag decompress

The command `rosbag decompress` decompresses one or more bag files.

### 9) rosbag reindex

The command `rosbag reindex` re-indexes one or more broken bag files.

## 30.14. `roscore`

TODO: to write

## 30.15. Troubleshooting ROS

**Symptom:** *computer* `is not in your SSH known_hosts file`

See this thread. Remove the `known_hosts` file and make sure you have followed the instructions in Section 19.3 - Local configuration.

## 30.16. Other materials about ROS.

- A gentle introduction to ROS

# Unit J-31
# How to install PyTorch on the Duckiebot

PyTorch is a Python deep learning library that's currently gaining a lot of traction, because it's a lot easier to debug and prototype (compared to TensorFlow / Theano).

To install PyTorch on the Duckietbot you have to compile it from source, because there is no pro-compiled binary for ARMv7 / ARMhf available. This guide will walk you through the required steps.

## 31.1. Step 1: install dependencies and clone repository

First you need to install some additional packages. You might already have installed. If you do, that's not a problem.

```
sudo apt-get install libopenblas-dev cython libatlas-dev m4 libblas-dev
```

In your current shell add two flags for the compiler

```
export NO_CUDA=1 # this will disable CUDA components of PyTorch, because the little
RaspberriPi doesn't have a GPU that supports CUDA
export NO_DISTRIBUTED=1 # no idea what this does, but it fixed a compilation bug for me
```

Then `cd` into a directory of your choice, like `cd ~/Downloads` or something like that and clone the PyTorch library.

```
git clone --recursive https://github.com/pytorch/pytorch
```

There was recently a bug in the ARM-relevant code that should now be fixed in the main Github branch, but just to make sure you have the most recent code:...

Change into the directory that you just cloned, and further into the following direcotries:

```
cd pytorch/torch/lib/ATen/
```

...and check that the file `Scalar.h` has the following code on line 16:

```
  Scalar() : Scalar(int64_t(0))
```

If the line instead reads the following, please manually change the code to the above line:

```
  Scalar() : Scalar(0L)
```

EDIT: As of 22nd January, 2018: the file Scalar.h is in pytorch/aten/src/ATen/Scalar.h; Line 24 In case, this changes in the future, use the command to find random phrases in the files:

```
grep -rn . -e 'Scalar() : Scalar(int64_t(0))'
```

## 31.2. Step 2: Change swap size

When I was compiling the library I ran out of SWAP space (which is 500MB by default). I was successful in compiling it with 2GB of SWAP space. Here is how you can increase the SWAP (only for compilation - later we will switch back to 500MB).

Create the swap file of 2GB

```
sudo dd if=/dev/zero of=/swap1 bs=1M count=2048
```

Make this empty file into a swap-compatible file

```
sudo mkswap /swap1
```

Then disable the old swap space and enable the new one

```
sudo nano /etc/fstab
```

This above command will open a text editor on your `/etc/fstab` file. The file should have this as the last line: `/swap0 swap swap`. In this line, please change the `/swap0` to `/swap1`. Then save the file with CTRL+o and ENTER. Close the editor with CTRL+x.

Now your system knows about the new swap space, and it will change it upon reboot, but if you want to use it right now, without reboot, you can manually turn off and empty the old swap space and enable the new one:

```
sudo swapoff /swap0
sudo swapon /swap1
```

## 31.3. Step 3: compile PyTorch

`cd` into the main directory, that you clones PyTorch into, in my case `cd ~/Downloads/pytorch` and start the compilation process:

```
python setup.py build
```

This shouldn't create any errors but it took me about an hour. If it does throw some exceptions, please let me know.

When it's done, you can install the pytorch package system-wide with

```
sudo python setup.py install
```

For some reason on my machine this caused recompilation of a few packages. So this might again take some time (but should be significantly less).

## 31.4. Step 4: try it out

If all of the above went through without any issues, congratulations. :) You should now have a working PyTorch installation. You can try it out like this.

First you need to change out of the installation directory (**this is important - otherwise you get a really weird error**):

```
cd ~
```

Then run Python:

```
python
```

And on the Python interpreter try this:

```
import torch
a = torch.FloatTensor((2,2))
a.add_(3)
print (a)
```

...this should print something like this:

```
3  3
3  3
[torch.FloatTensor of size 2x2]
```

## 31.5. (Step 5, optional: unswap the swap)

Now if you like having 2GB of SWAP space (additional RAM basically, but a lot slower than your built-in RAM), then you are done. The downside is that you might run out of space later on. If you want to revert back to your old 500MB swap file then do the following:

Open the `/etc/fstab` file in the editor:

```
sudo nano /etc/fstab
```

TODO

please change the `/swap0` to `/swap1`. Then save the file with CTRL+o and ENTER. Close the editor with CTRL+x.

# Unit J-32
# How to install Caffe and Tensorflow on the Duckiebot

Caffe and TensorFlow are popular deep learning libraries, and are supported by the Intel Neural Computing Stick (NCS).

## 32.1. Caffe

### 1) Step 1: install dependencies and clone repository

Install some of the dependencies first. The last command “sudo pip install .....” will cause some time.

```
sudo apt-get install -y gfortran cython
sudo apt-get install -y libprotobuf-dev libleveldb-dev libsnappy-dev libopencv-dev
libhdf5-serial-dev protobuf-compiler git
sudo apt-get install --no-install-recommends libboost-all-dev
sudo apt-get install -y python-dev libgflags-dev libgoogle-glog-dev liblmdb-dev
libatlas-base-dev python-skimage
sudo pip install pyzmq jsonschema pillow numpy scipy ipython jupyter pyyaml
```

Then, you need to clone the repo of caffe

```
cd
git clone https://github.com/BVLC/caffe
```

### 2) Step 2: compile Caffe

Before compile Caffe, you have to modify Makefile.config

```
cd caffe
cp Makefile.config.example Makefile.config
sudo vim Makefile.config
```

Then, change four lines from

```
’#’CPU_ONLY := 1
/usr/lib/python2.7/dist-packages/numpy/core/include
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include
LIBRARY_DIRS := $(PYTHON_LIB) /usr/local/lib /usr/lib
```

to

```
CPU_ONLY := 1
/usr/local/lib/python2.7/dist-packages/numpy/core/include
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include /usr/include/hdf5/serial/
LIBRARY_DIRS := $(PYTHON_LIB) /usr/local/lib /usr/lib /usr/lib/arm-linux-gnueabihf/hdf5/
serial/
```

Next, you can start to compile caffe

```
make all
make test
make runtest
make pycaffe
```

If you did't get any error above, congratulation on your success. Finally, please export pythonpath

```
sudo vim ~/.bashrc
export PYTHONPATH=/home/''$USER''/caffe/python:$PYTHONPATH
```

### 3) Step 3: try it out

Now, we can confirm whether the installation is successful. Download AlexNet and run caffe time

```
cd ~/caffe/
python scripts/download_model_binary.py models/bvlc_alexnet
./build/tools/caffe time -model models/bvlc_alexnet/deploy.prototxt -weights  models/
bvlc_alexnet/bvlc_alexnet.caffemodel   -iterations 10
```

And you can see the benchmark of AlexNet on Pi3 caffe.

## 32.2. Tensorflow

### 1) Step 1: install dependencies and clone repository

First, update apt-get:

```
$ sudo apt-get update
```

For Bazel:

```
$ sudo apt-get install pkg-config zip g++ zlib1g-dev unzip
```

For Tensorflow: (NCSDK only support python 3+. I didn't use mvNC on rpi3, so here I choose python 2.7)

(For Python 2.7)

```
$ sudo apt-get install python-pip python-numpy swig python-dev
$ sudo pip install wheel
```

(For Python 3.3+)

```
$ sudo apt-get install python3-pip python3-numpy swig python3-dev
$ sudo pip3 install wheel
```

To be able to take advantage of certain optimization flags:

```
$ sudo apt-get install gcc-4.8 g++-4.8
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 100
$ sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8 100
```

Make a directory that hold all the thing you need

```
$ mkdir tf
```

## 2) Step 2: build Bazel

Download and extract bazel (here I choose 0.7.0):

```
$ cd ~/tf
$ wget https://github.com/bazelbuild/bazel/releases/download/0.7.0/bazel-0.7.0-dist.zip
$ unzip -d bazel bazel-0.7.0-dist.zip
```

Modify some file:

```
$ cd bazel
$ sudo chmod u+w ./* -R

$ nano scripts/bootstrap/compile.sh
```

To line 117, add “-J-Xmx500M”:

```
run "${JAVAC}" -classpath "${classpath}" -sourcepath "${sourcepath}" \
      -d "${output}/classes" -source "$JAVA_VERSION" -target "$JAVA_VERSION" \
      -encoding UTF-8 "@${paramfile}" -J-Xmx500M
```

Figure 32.1

```
$ nano tools/cpp/cc_configure.bzl
```

Place the line return “arm” around line 133 (beginning of the _get_cpu_value function):

```
...
"""Compute the cpu_value based on the OS name."""
return "arm"
...
```

Figure 32.2

Build Bazel (it will take a while, about 1 hour):

```
$ ./compile.sh
```

When the build finishes:

```
$ sudo cp output/bazel /usr/local/bin/bazel
```

Run bazel check if it's working:

```
$ bazel
```

```
Usage: bazel <command> <options> ...

Available commands:
  analyze-profile     Analyzes build profile data.
  build               Builds the specified targets.
  canonicalize-flags  Canonicalizes a list of bazel options.
  clean               Removes output files and optionally stops the server.
  dump                Dumps the internal state of the bazel server process.
  fetch               Fetches external repositories that are prerequisites to the targets.
  help                Prints help for commands, or the index.
  info                Displays runtime info about the bazel server.
  mobile-install      Installs targets to mobile devices.
  query               Executes a dependency graph query.
  run                 Runs the specified target.
  shutdown            Stops the bazel server.
  test                Builds and runs the specified test targets.
  version             Prints version information for bazel.

Getting more help:
  bazel help <command>
                    Prints help and options for <command>.
  bazel help startup_options
                    Options for the JVM hosting bazel.
  bazel help target-syntax
                    Explains the syntax for specifying targets.
  bazel help info-keys
                    Displays a list of keys used by the info command.
```

Figure 32.3

### 3) Step 3: compile Tensorflow

Clone tensorflow repo (here I choose 1.4.0):

```
$ cd ~/tf
$ git clone -b r1.4 https://github.com/tensorflow/tensorflow.git
$ cd tensorflow
```

(Incredibly important) Changes references of 64-bit program implementations (which we don't have access to) to 32-bit implementations.

```
$ grep -Rl 'lib64' | xargs sed -i 's/lib64/lib/g'
```

Modify the file platform.h:

```
$ sed -i "s|#define IS_MOBILE_PLATFORM|//#define IS_MOBILE_PLATFORM|g" tensorflow/core/
platform/platform.h
```

Configure the build: (important) if you want to build for Python 3, specify /usr/bin/python3 for Python's location and /usr/local/lib/python3.x/dist-packages for the Python library path.

```
$ ./configure
Please specify the location of python. [Default is /usr/bin/python]: /usr/bin/python
Please specify optimization flags to use during compilation when bazel option "--config=opt"
Do you wish to use jemalloc as the malloc implementation? [Y/n] Y
Do you wish to build TensorFlow with Google Cloud Platform support? [y/N] N
Do you wish to build TensorFlow with Hadoop File System support? [y/N] N
Do you wish to build TensorFlow with the XLA just-in-time compiler (experimental)? [y/N] N
Please input the desired Python library path to use. Default is [/usr/local/lib/python2.7/di
Do you wish to build TensorFlow with OpenCL support? [y/N] N
Do you wish to build TensorFlow with CUDA support? [y/N] N
```

Figure 32.4

Build the Tensorflow (this will take a LOOOONG time, about 7 hrs ):

```
$ bazel build -c opt --copt="-mfpu=neon-vfpv4" --copt="-funsafe-math-optimizations"
--copt="-ftree-vectorize" --copt="-fomit-frame-pointer" --local_resources 1024,1.0,1.0
--verbose_failures tensorflow/tools/pip_package:build_pip_package
```

After finished compiling, install python wheel:

```
$ bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg
$ sudo pip install /tmp/tensorflow_pkg/tensorflow-1.4.0-cp27-none-linux_armv7l.whl
```

Check version:

```
$ python -c 'import tensorflow as tf; print(tf.__version__)'
```

And you're done! You deserve a break.

### 4) Step 3: try it out

Suppose you already have inception-v3 model (with inception-v3.meta and inception-v3.ckpt)

Create a testing python file

```
$ vim test.py
```

Write the following code:

```python
import tensorflow as tf
import numpy as np
import cv2
import sys
import time

def run(input_image):
    tf.reset_default_graph()
    with tf.Session() as sess:
        saver = tf.train.import_meta_graph('./output/inception-v3.meta')
        saver.restore(sess, 'inception_v3.ckpt')

        softmax_tensor = sess.graph.get_tensor_by_name('Softmax:0')
        feed_dict = {'input:0': input_image}
        classification = sess.run(softmax_tensor, {'input:0': input_image})    #first run fo warm-up

        start_time = time.time()
        classification = sess.run(softmax_tensor, {'input:0': input_image})
        print 'predict label:', np.argmax(classification[0])
        print 'predict time:', time.time() - start_time, 's'

if __name__=="__main__":
    args = sys.argv
    if len(args) != 2:
        print 'Usage: python %s filename'%args[0]
        quit()
    image_data =tf.gfile.FastGFile(args[1], 'rb').read()
    image = cv2.imread(args[1])
    image = cv2.resize(image, (299,299))
    image = np.array(image)/255.0
    image = np.asarray(image).reshape((1, 299, 299, 3))
    run(image)
```

Figure 32.5

Save, and excute it

```
$ python test.py cat.jpg
```

Then it will show the predict label and predict time.

# Unit J-33
# Movidius Neural Compute Stick Install

## 33.1. Laptop Installation

install based on ncsdk website

```
git clone http://github.com/Movidius/ncsdk
cd ~/ncsdk
make install
make examples
```

test installation

```
cd ~/ncsdk/examples/app/hello_ncs_py/
make run
```

## 33.2. Duckiebot Installation

you only need to install the NCSDK but there is also the option of installing Caffe and/or Tensorflow as well, in order to perhaps speed up the development cycle. I would recommend against it, as it can be a bigger problem than it solves.

### 1) Barebones Install (recommended)

you don't need tensorflow, caffe, or any tools in order to run the compiled networks and not installing them will save you a lot of hassle

on duckiebot:

```
git clone http://github.com/Movidius/ncsdk
cd ~/ncsdk/api/src
make
sudo make install
```

### 2) Caffe/Tensorflow Install

> Note: if you want to be able to compile your models on the duckiebot itself, install tensorflow or caffe beforehand and remember to install for python 3 (`pip3`)

follow directions here

make sure caffe and tensorflow as installed

```
python3 -c 'import tensorflow as tf; import caffe'
```

install sdk:

```
git clone http://github.com/Movidius/ncsdk
cd ~/ncsdk
make install
make examples
```

# Unit J-34
# How To Use Neural Compute Stick

## 34.1. Workflow

create and train model in tensorflow or caffe (brief note on configuration )
save tensorflow model as a `.meta` (or caffe model in `.prototxt`)

```
saver = tf.train.Saver()
...
saver.save(sess, 'model')
```

compile the model into NC format (documentation here)

```
mvNCCompile model.meta -o model.graph
```

move model onto duckiebot

```
scp model.meta user@robot name:~/path_to_networks/
```

run the compiled model

```
with open(path_to_networks + model.meta, mode='rb') as f:
    graphfile = f.read()
graph = device.AllocateGraph(graphfile)
graph.LoadTensor(input_image.astype(numpy.float16), 'user object')
output, userobj = graph.GetResult()
```

## 34.2. Benchmarking

get benchmarking (frames per second) from their app zoo

```
git clone https://github.com/movidius/ncappzoo
cd ncappzoo/apps/benchmarkncs
./mobilenets_benchmark.sh | grep FPSk
```

# Part K
# Software development guide

This part is about how to develop software for the Duckiebot.

# Unit K-1
# Python

## 1.1. Background reading

- Python
- Python tutorial

## 1.2. Python virtual environments

Install using:

```
$ sudo apt install virtualenv
```

## 1.3. Useful libraries

```
matplotlib
seaborne
numpy
panda
scipy
opencv
...
```

## 1.4. Context managers

TODO: to write

## 1.5. Exception hierarchies

TODO: to write

TODO: how to catch and re-raise

## 1.6. Object orientation - Abstract classes, class hierarchies

TODO: to write

## 1.7. Downloading resources

Use this recipe if you need to download things:

```
from duckietown_utils.download import download_if_not_exist
url = 'https://www.dropbox.com/s/bzezpw8ivlfu4b0/frame0002.jpg?dl=0'
f = 'local.jpg'
download_if_not_exist(url, f)
```

(Do not commit JPGs and other binary data to the `Software` repository.)

**TODO:** actually use `urls.yaml`

## 1.8. IPython

How to enter IPython:

```
from IPython import embed()

a = 10
embed() # enters interactive mode
```

## 1.9. Idioms

`segment_list = copy.deepcopy(segment_list)`

➡ add_duckietown_header

➡

# Unit K-2
# Working with YAML files

YAML files are useful for writing configuration files, and are used a lot in Duckietown.

## 2.1. Pointers to documentation

TODO: to write

## 2.2. Editing YAML files

TODO: to write

## 2.3. Reading and writing YAML files in Python

TODO: the yaml library

TODO: the ruamel.yaml library

## 2.4. Duckietown wrapping API

TODO: to write

UNIT K-3

# Duckietown code conventions

## 3.1. Python

### 1) Tabs

Never use tabs in Python files.

The tab characters are evil in Python code. Please be *very* careful in changing them.

Do *not* use a tool to do it (e.g. "Convert tabs to spaces"); it will get it wrong.

✔ checked by `what-the-duck`.

### 2) Spaces

Indentation is 4 spaces.

### 3) Line lengths

Lines should be below 85 characters.

✔ `what-the-duck` report those above 100 characters.

This is just a symptom of a bigger problem.

The problem here is that you do not do how to program well, therefore you create programs with longer lines.

Do not go and try to shorten the lines; the line length is just the symptom. Rather, ask somebody to take a look at the code and tell you how to make it better.

### 4) The encoding line

All files must have an encoding declared, and this encoding must be `utf-8`:

```
# -*- coding: utf-8 -*-
```

### 5) Sha-bang lines

Executable files start with:

```
#!/usr/bin/env python
```

### 6) Comments

Comments refer to the next line.

Comments, bad:

```
from std_msgs.msg import String # This is my long comment
```

Comments, better:

```
# This is my long comment
from std_msgs.msg import String
```

## 3.2. Logging

For logging, import this logger:

```
from duckietown_utils import logger
```

## 3.3. Exceptions

```
DTConfigException

raise_wrapped
compact = True
```

## 3.4. Scripts

```
def summary():
    fs = get_all_configuration_files()


if __name__ == '__main__':
    wrap_script_entry_point(summary)
```

### 1) Imports

Do not do a star import, like the following:

```
from rostest_example.Quacker import *
```

# Unit K-4
# Configuration

This chapter explains what are the assumptions about the configuration.

While the "Setup" parts are "imperative" (do this, do that); this is the "declarative" part, which explains what are the properties of a correct configuration (but it does not explain how to get there).

The tool `what-the-duck` (Subsection 8.1.3 - The `what-the-duck` program) checks some of these conditions. If you make a change from the existing conditions, make sure that it gets implemented in `what-the-duck` by filing an issue.

## 4.1. Environment variables (updated Sept 12)

You need to have set up the variables in Table 4.1.

> **Note:** The way to set these up is to add them in the file `~/.bashrc` (`export` *var*='*value*'). Do not modify the `environment.sh` script.

Table 4.1. Environment variables used by the software

| variable | reasonable value | contains |
|---|---|---|
| `DUCKIETOWN_ROOT` | `~/duckietown` | `Software` repository |
| `DUCKIEFLEET_ROOT` | `~/duckiefleet` | Where to look for class-specific information (people DB, robots DB). |
| `DUCKIETOWN_DATA` | `~/duckietown-data` | The place where to look for logs. |
| `DUCKIETOWN_TMP` | | If set, directory to use for temporary files. If not set, we use the default ( `/tmp`). |
| `DUCKIETOWN_CONFIG_SEQUENCE` | `defaults:baseline:vehicle:user` | The configuration sequence for EasyNode |

### 1) Duckietown root directory `DUCKIETOWN_ROOT`

TODO: to write

### 2) Duckiefleet directory `DUCKIEFLEET_ROOT`

For Fall 2017, this is the the repository `duckiefleet`.

For self-guided learners, this is an arbitrary repository to create.

## 4.2. The "scuderia" (vehicle database)

The system needs to know certain details about the robots, such as their host names, the name of the users, etc.

This data is contained in the `${DUCKIEFLEET_ROOT}/robots/{your_branch}` directory, in files with the pattern *robot name*`.robot.yaml`.

The file must contain YAML entries of the type:

```yaml
owner: ID of owner
username: username on the machine
hostname: host name
description: generic description
log:
    date: comment
    date: comment
```

A minimal example is in Listing 4.6.

```yaml
owner: censi
hostname: emma
username: andrea
description: Andrea's car.
log:
    2017-08-01: >
        Switched RPI2 with RPI3.
    2017-08-20: >
        There is something wrong with the PWM hat and the LEDs.
```

Listing 4.6. Minimal scuderia file `emma.robot.yaml`

Explanations of the fields:

- *hostname*: the host name. This is normally the same as the robot name.
- *username*: the name of the Linux user on the robot, from which to run programs.
- *owner*: the owner's globally-unique Duckietown ID.

## 4.3. The `machines` file

Make sure you already set up ROS (Section 11.3 - Set up the ROS environment on the Duckiebot).

Activate ROS:

```
$ cd ~/duckietown
$ source environment.sh
```

The `machines` file is created from the scuderia data using this command:

```
$ rosrun duckieteam create-machines
```

## 4.4. People database

> Assigned to: Andrea

TODO: Describe the people database.

### 1) The globally-unique Duckietown ID

This is a globally-unique ID for people in the Duckietown project.

It is equal to the Slack username.

+ comment

There is no Slack username anymore, so we should change this to some other convention. -AC

## 4.5. Modes of operation

There are 3 modes of operation:

1. `MODE-normal`: Everything runs on the robot.
2. `MODE-offload`: Drivers run on the robot, but heavy computation runs on the laptop.
3. `MODE-bag`: The data is provided from a bag file, and computation runs on the laptop.

TABLE 4.2. OPERATION MODES

| mode name | who is the ROS master | where data comes from | where heavy computation happen |
|---|---|---|---|
| `MODE-normal` | `duckiebot` | Drivers on Duckiebot | `duckiebot` |
| `MODE-offload` | `duckiebot` | Drivers on Duckiebot | `laptop` |
| `MODE-bag` | `laptop` | Bag file | `laptop` |

UNIT K-5

# Node configuration mechanisms

TODO: Where the config files are, how they are used.

UNIT K-6

# Minimal ROS node - pkg_name

> **Assigned to:** Andrea

This document outline the process of writing a ROS package and nodes in Python.

To follow along, it is recommend that you duplicate the `pkg_name` folder and edit the content of the files to make your own package.

## 6.1. The files in the package

### 1) CMakeLists.txt

We start with the file `CMakeLists.txt`.

Every ROS package needs a file `CMakeLists.txt`, even if you are just using Python code in your package.

> ☀ documentation about CMakeLists.txt XXX

For a Python package, you only have to pay attention to the following parts.

The line:

```
project(pkg_name)
```

defines the name of the project.

The `find_package` lines:

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  duckietown_msgs # Every duckietown packages must use this.
  std_msgs
)
```

You will have to specify the packages on which your package is dependent.

In Duckietown, most packages depend on `duckietown_msgs` to make use of the customized messages.

The line:

```
catkin_python_setup()
```

tells `catkin` to setup Python-related stuff for this package.

☀ ROS documentation about `setup.py`

### 2) package.xml

The file package.xml defines the meta data of the package.

Catkin makes use of it to flush out the dependency tree and figures out the order of compiling.

Pay attention to the following parts.

`<name>` defines the name of the package. It has to match the project name in `CMakeLists.txt`.

`<description>` describes the package concisely.

`<maintainer>` provides information of the maintainer.

`<build_depend>` and `<run_depend>`. The catkin packages this package depends on. This usually match the `find_package` in `CMakeLists.txt`.

### 3) setup.py

The file setup.py configures the Python modules in this package.

The part to pay attention to is

```
setup_args = generate_distutils_setup(
    packages=['pkg_name'],
    package_dir={'': 'include'},
)
```

The `packages` parameter is set to a list of strings of the name of the folders inside the `include` folder.

The convention is to set the folder name the same as the package name. Here it's the `include/pkg_name` folder.

You should put ROS-independent and/or reusable module (for other packages) in the `include/pkg_name` folder.

Python files in this folder (for example, the `util.py`) will be available to scripts in the `catkin` workspace (this package and other packages too).

To use these modules from other packages, use:

```
from pkg_name.util import *
```

## 6.2. Writing a node: `talker.py`

Let's look at src/talker.py as an example.

ROS nodes are put under the `src` folder and they have to be made executable to function properly.

➡ You use `chmod` for this; see Section 11.1 - chmod.

### 1) Header

Header:

```python
#!/usr/bin/env python
import rospy
# Imports module. Not limited to modules in this package.
from pkg_name.util import HelloGoodbye
# Imports msg
from std_msgs.msg import String
```

The first line, `#!/usr/bin/env python`, specifies that the script is written in Python.

Every ROS node in Python must start with this line.

The line `import rospy` imports the `rospy` module necessary for all ROS nodes in Python.

The line `from pkg_name.util import HelloGoodbye` imports the class `HelloGoodbye` defined in the file pkg_name/util.py.

Note that you can also include modules provided by other packages, if you specify the dependency in `CMakeLists.txt` and `package.xml`.

The line `from std_msgs.msg import String` imports the `String` message defined in the `std_msgs` package.

Note that you can use `rosmsg show std_msgs/String` in a terminal to lookup the definition of `String.msg`.

## 2) Main

This is the main file:

```python
if __name__ == '__main__':
    # Initialize the node with rospy
    rospy.init_node('talker', anonymous=False)

    # Create the NodeName object
    node = Talker()

    # Setup proper shutdown behavior
    rospy.on_shutdown(node.on_shutdown)

    # Keep it spinning to keep the node alive
    rospy.spin()
```

The line `rospy.init_node('talker', anonymous=False)` initializes a node named `talker`.

Note that this name can be overwritten by a launch file. The launch file can also push this node down namespaces. If the `anonymous` argument is set to `True` then a random string of numbers will be append to the name of the node. Usually we don't use anonymous nodes.

The line `node = Talker()` creates an instance of the Talker object. More details in the next section.

The line `rospy.on_shutdown(node.on_shutdown)` ensures that the `node.on_shutdown` will be called when the node is shutdown.

The line `rospy.spin()` blocks to keep the script alive. This makes sure the node stays

alive and all the publication/subscriptions work correctly.

## 6.3. The Talker class

We now discuss the `Talker` class in [talker.py](talker.py).

### 1) Constructor

In the constructor, we have:

```
self.node_name = rospy.get_name()
```

saves the name of the node.

This allows to include the name of the node in printouts to make them more informative. For example:

```
rospy.loginfo("[%s] Initializing." % (self.node_name))
```

The line:

```
self.pub_topic_a = rospy.Publisher("~topic_a", String, queue_size=1)
```

defines a publisher which publishes a `String` message to the topic `~topic_a`. Note that the `~` in the name of topic under the namespace of the node. More specifically, this will actually publish to `talker/topic_a` instead of just `topic_a`. The `queue_size` is usually set to 1 on all publishers.

➡ For more details see [rospy overview: publisher and subscribers](rospy overview: publisher and subscribers).

The line:

```
self.sub_topic_b = rospy.Subscriber("~topic_b", String, self.cbTopic)
```

defines a subscriber which expects a `String` message and subscribes to `~topic_b`. The message will be handled by the `self.cbTopic` callback function. Note that similar to the publisher, the `~` in the topic name puts the topic under the namespace of the node. In this case the subscriber actually subscribes to the topic `talker/topic_b`.

It is strongly encouraged that a node always publishes and subscribes to topics under their `node_name` namespace. In other words, always put a `~` in front of the topic names when you defines a publisher or a subscriber. They can be easily remapped in a launch file. This makes the node more modular and minimizes the possibility of confusion and naming conflicts. See [the launch file section](the launch file section) for how remapping works.

The line

```
self.pub_timestep = self.setupParameter("~pub_timestep", 1.0)
```

Sets the value of `self.pub_timestep` to the value of the parameter `~pub_timestep`. If the parameter doesn't exist (not set in the launch file), then set it to the default value `1.0`. The `setupParameter` function also writes the final value to the parameter server. This means that you can `rosparam list` in a terminal to check the actual values of parameters being set.

The line:

```
self.timer = rospy.Timer(rospy.Duration.from_sec(self.pub_timestep), self.cbTimer)
```

defines a timer that calls the `self.cbTimer` function every `self.pub_timestep` seconds.

### 2) Timer callback

Contents:

```
def cbTimer(self,event):
    singer = HelloGoodbye()
    # Simulate hearing something
    msg = String()
    msg.data = singer.sing("duckietown")
    self.pub_topic_name.publish(msg)
```

Everyt ime the timer ticks, a message is generated and published.

### 3) Subscriber callback

Contents:

```
def cbTopic(self,msg):
    rospy.loginfo("[%s] %s" %(self.node_name,msg.data))
```

Every time a message is published to `~topic_b`, the `cbTopic` function is called. It simply prints the message using `rospy.loginfo`.

## 6.4. Launch File

You should always write a launch file to launch a node. It also serves as a documentation on the I/O of the node.

Let's take a look at `launch/test.launch`.

```
<launch>
    <node name="talker" pkg="pkg_name" type="talker.py" output="screen">

        <param name="~pub_timestep" value="0.5"/>

        <remap from="~topic_b" to="~topic_a"/>
    </node>
</launch>
```

For the `<node>`, the `name` specify the name of the node, which overwrites `rospy.init_node()` in the `__main__` of `talker.py`. The `pkg` and `type` specify the package and the script of the node, in this case it's `talker.py`.

Don't forget the `.py` in the end (and remember to make the file executable through `chmod`).

The `output="screen"` direct all the `rospy.loginfo` to the screen, without this you won't see any printouts (useful when you want to suppress a node that's too talkative.)

The `<param>` can be used to set the parameters. Here we set the `~pub_timestep` to `0.5`. Note that in this case this sets the value of `talker/pub_timestep` to `0.5`.

The `<remap>` is used to remap the topic names. In this case we are replacing `~topic_b` with `~topic_a` so that the subscriber of the node actually listens to its own publisher. Replace the line with

```
<remap from="~topic_b" to="talker/topic_a"/>
```

will have the same effect. This is redundant in this case but very useful when you want to subscribe to a topic published by another node.

## 6.5. Testing the node

First of all, you have to `catkin_make` the package even if it only uses Python. `catkin` makes sure that the modules in the include folder and the messages are available to the whole workspace. You can do so by

```
$ cd ${DUCKIETOWN_ROOT}/catkin_ws
$ catkin_make
```

Ask ROS to re-index the packages so that you can auto-complete most things.

```
$ rospack profile
```

Now you can launch the node by the launch file.

```
$ roslaunch pkg_name test.launch
```

You should see something like this in the terminal:

```
... logging to /home/username/.ros/log/d4db7c80-b272-11e5-8800-5c514fb7f0ed/
roslaunch-robot name-15961.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is 1GB.

started roslaunch server http://robot name.local:33925/

SUMMARY
========

PARAMETERS
 * /rosdistro: $ROS_DISTRO
 * /rosversion: 1.11.16
 * /talker/pub_timestep: 0.5

NODES
  /
    talker (pkg_name/talker.py)

auto-starting new master
process[master]: started with pid [15973]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to d4db7c80-b272-11e5-8800-5c514fb7f0ed
process[rosout-1]: started with pid [15986]
started core service [/rosout]
process[talker-2]: started with pid [15993]
[INFO] [WallTime: 1451864197.775356] [/talker] Initialzing.
[INFO] [WallTime: 1451864197.780158] [/talker] ~pub_timestep = 0.5
[INFO] [WallTime: 1451864197.780616] [/talker] Initialzed.
[INFO] [WallTime: 1451864198.281477] [/talker] Goodbye, duckietown.
[INFO] [WallTime: 1451864198.781445] [/talker] Hello, duckietown.
[INFO] [WallTime: 1451864199.281871] [/talker] Goodbye, duckietown.
[INFO] [WallTime: 1451864199.781486] [/talker] Hello, duckietown.
[INFO] [WallTime: 1451864200.281545] [/talker] Goodbye, duckietown.
[INFO] [WallTime: 1451864200.781453] [/talker] Goodbye, duckietown.
```

Open another terminal and run:

```
$ rostopic list
```

You should see

```
/rosout
/rosout_agg
/talker/topic_a
```

In the same terminal, run:

```
$ rosparam list
```

You should see the list of parameters, including `/talker/pub_timestep`.
You can see the parameters and the values of the `talker` node with

```
$ rosparam get /talker
```

## 6.6. Adding a command line parameter

You can register a parameter in the launch file such that it is added to the ROS parameter dictionary. This allows you to call `rospy.get_param()` on you parameter from `talker.py`.
Edit your launch file to look like this:

```
<launch>
    <arg name="pub_timestep" default="0.5" />
    <node name="talker" pkg="pkg_name" type="talker.py" output="screen">

        <param name="~pub_timestep" value="$(arg pub_timestep)"/>

        <remap from="~topic_b" to="~topic_a"/>
    </node>
</launch>
```

Previously, you should have had the line `<param name="~pub_timestep" value="0.5" />` inside of the node tags. This sets a parameter of value `0.5` to be called `/talker/pub_timestep`. (Remeber that the tilde prefixes the variable with the current namespace). By adding the line `<arg name="pub_timestep" default="1" />`, we are telling the program to look for a parameter on the command line called `pub_timestep`, and that if it doesn't find one, to use the value one. Then, `value=$(arg pub_timestep)` retrieves the value set in the previous line.

Within `talker.py`, we can get the value of the inputted parameter. You should already have the line:

```
self.pub_timestep = self.setupParameter("~pub_timestep", 1.0)
```

This calles the talker's `setupParameter` method, which contains the line:

```
value = rospy.get_param(param_name,default_value)
```

Where `~pub_timestep` is passed in as `parapm_name` and `1.0` is passed in as the default value. Now that we have edited the launch file to accept a command line argument, `value` should be the value which is given on the command line, rather than `0.5`.

You can test that this works by calling `roslaunch` with the added parameter:

```
$ roslaunch pkg_name test.launch pub_timestep:=3
```

This should cause the time between messages to become three seconds.

The functions `rosparam list` and `rosparam info [param]` are useful in debugging issues with registering a parameter.

## 6.7. Documentation

You should document the parameters and the publish/subscribe topic names of each node in your package. The user should not have to look at the source code to figure out how to use the nodes.

## 6.8. Guidelines

- Make sure to put all topics (publish or subscribe) and parameters under the namespace of the node with `~`. This makes sure that the IO of the node is crystal clear.
- Always include the name of the node in the printouts.
- Always provide a launch file that includes all the parameters (using `<param>`) and topics (using `<remap>`) with each node.

# Unit K-7
# Makefile system

Assigned to: Andrea

We use Makefiles to describe frequently-used commands.

## 7.1. User guide

The command

```
$ make all
```

displays the help for each command. This help is also included here as Section 7.3 - Makefile help.

## 7.2. Makefile organization

There is one Makefile at the root of the `Software` repository, which includes other makefiles in the directory `Makefiles/`:

```
Makefile
Makefiles/
    Makefile.build.mk
    Makefile.demos.mk
    Makefile.docker.mk
    Makefile.generate.mk
    Makefile.hw_test.mk
    Makefile.maintenance.mk
    Makefile.openhouse.mk
    Makefile.stats.mk
    Makefile.test.mk
```

Each child Makefile is called

`Makefile.`*section*`.mk`

and it should contain only targets of the form *section*-*name*.

For example, `Makefile.stats.mk` contains the targets "`stats`, `stats-easy_node`, `stats-easy_logs`".

The target called *section* should provide an help for the section.

For example, when you run `make build`, you see:

```
### Building commands

Commands to build the software.

- `make build-machines`       :  Builds the machines file.
- `make build-machines-clean` :  Removes the machines file.
- `make build-clean`          :  Clean everything.
```

The output should be valid Markdown, so that it can be included in this documentation.

## 7.3. Makefile help

### 1) Statistics

These provide statistics about the data and the configuration.

- `make stats-easy_node`: Prints summary of declared nots using the EasyNode frameworks.
- `make stats-easy_logs`: Prints summary of available logs.
- `make stats-easy_algo`: Prints summary of available algorithms.

### 2) Testing

These commands run the unit tests.

- `make test-all`: Run all the tests.
- `make test-circle`: The tests to run in continuous integration. .
- `make test-catkin_tests`: Run the ROS tests.
- `make test-anti_instagram`: Run the `anti_instagram` tests.
- `make test-comptests`: Run the `comptests` tests.
- `make test-comptests-clean`: Run the `comptests` tests.
- `make test-comptests-collect-junit`: Collects the JUnit results.
- `make test-download-logs`: Downloads the logs needed for the tests.

### 3) Building commands

Commands to build the software.

- `make build-catkin` : Runs `catkin_make`.
- `make build-catkin-parallel` : Runs `catkin_make`, with 4 threads.
- `make build-catkin-parallel-max` : Runs `catkin_make`, with many threads.
- `make build-machines` : Builds the machines file.
- `make build-machines-clean` : Removes the machines file.
- `make build-clean` : Clean everything.

### 4) Docker commands

For using Docker images

- `make docker-build`: Creates the image.

- `make docker-upload`: Uploads the image.
- `make docker-clean`: Removes all local images.

### 5) Automated files generation

Generation of documentation

- `make generate-all`: Generates everything.
- `make generate-help`: Generates help.
- `make generate-easy_node`: Generates the easy node documentation.
- `make generate-easy_node-clean`: Cleans the generated files.

### 6) Demos

These are simple demos

**TODO:** to write

### 7) Hardware tests

To perform hardware tests:

- `make hw-test-camera` : Testing Camera HW by taking a picture (smile!).
- `make hw-test-kinematics : Testing kinematics calibration
- `make hw-test-turn-right`: Calibration right turn
- `make hw-test-turn-left`: Calibrating left turn
- `make hw-test-turn-forward`: Calibrating forward turn

### 8) Maintenance

A couple of utilities for robot maintenance.

- `make maintenance-fix-time`: Fixes the time.
- `make maintenance-clean-pyc`: Removes pyc files.

### 9) Open house demos

These were the open house demos.

**TODO:** to write

# Unit K-8
# Jupyter

## 8.1. Installation

Pull the branch 1710-place-recognition

```
$ cd duckietown
$ git pull
```

Source environment:

```
$ source environment.sh
```

Remove previous installations:

```
$ sudo apt remove ipython ipython-notebook
```

Then run:

```
$ pip install --user jupyter IPython==5.0
```

Check the versions are correct:

```
$ which ipython

/home/andrea/.local/bin/ipython
```

Check the version is correct:

```
$ ipython --version

5.0.0
```

## 8.2. Configuration

Set a password:

```
$ jupyter notebook password
```

## 8.3. Running it

```
$ jupyter notebook --notebook-dir=$DUCKIETOWN_ROOT/catkin_ws/src/75-notebooks
```

## 8.4. Extra configuration for virtual machines

Create a configuration file:

```
$ jupyter notebook --generate-config
```

Edit the file `~/.jupyter/jupyter_notebook_config.py`.
Uncomment and change the line:

```
#c.NotebookApp.ip = 'localhost'
```

Into:

```
c.NotebookApp.ip = '*'
```

UNIT K-9

# ROS package verification

Assigned to: Andrea

This chapter describes formally what makes a conforming ROS package in the Duckietown software architecture.

## 9.1. Naming

□ For exercises packages, the name of the package must be `package_`*handle*.

## 9.2. `package.xml`

□ There is a `package.xml` file.

✔ Checked by `what-the-duck`.

## 9.3. Messages

□ The messages are called ....

## 9.4. Readme file

□ There is a `README.md` file

✔ Checked by `what-the-duck`.

## 9.5. Launch files

□ there is the first launch file

## 9.6. Test files

TODO: to write

# Unit K-10
# Duckietown utility library

## 10.1. Images

This sections contains the documentation about the utility functions used for image processing available in the `duckietown_utils` Python package.

### 1) Function `write_image_as_jpg`

**Description**: Takes an BGR image and writes it as a JPEG file.

+ comment
Are we sure that the encoding is right? -AC

**Prototype:**

```
write_image_as_jpg( image, filename )
```

**Defined in:** `image_writing.py`.

**Arguments:**

| Name | Type | Description |
|---|---|---|
| image | numpy.ndarray | The BGR image to save as JPEG file. |
| filename | str | The path of the JPEG file. |

**Returns:** `None`.

### 2) Function `rgb_from_ros`

**Description**: Takes a ROS message containing an image and returns its RGB representation.

**Prototype:**

```
rgb_from_ros( msg )
```

**Defined in:** `image_conversions.py`.

**Arguments:**

| Name | Type | Description |
|---|---|---|
| msg | sensor_msgs.Image or sensor_msgs.CompressedImage | Message containing the image to extract. |

**Returns:** `numpy.ndarray` :: RGB representation of the image contained in the ROS message `msg`.

### 3) Function `d8_compressed_image_from_cv_image`

**Description**: Takes a OpenCV image (BGR format), compresses it and wraps it into a ROS message of type `sensor_msgs.CompressedImage`.

**Prototype:**

```
d8_compressed_image_from_cv_image( image_cv )
```

**Defined in:** `image_jpg_create.py`.

**Arguments:**

| Name | Type | Description |
|---|---|---|
| image_cv | numpy.ndarray | BGR representation of the image to compress. |

**Returns:** `sensor_msgs.CompressedImage` :: A ROS message containing a compressed version of the input `image_cv`.

# Unit K-11
# Bug squashing guide

This unit describes how to debug your programs.

Do read this accurately top-to-bottom. If you think this is too long and too verbose to read and you are in a hurry anyway: that is probably the attitude that introduced the bug.

## 11.1. Historical notes

First, count your blessings. You are lucky to live in the present. Once, there were actual bugs in your computer (Figure 11.1).

Figure 11.1. "First actual case of bug being found." Read the story here.

## 11.2. The basic truths of bug squashing

### 1) Truth: it is most likely something simple

The first truth is the following:

> It is always something simple.

People tend to make up complicated stories in their head about what is happening. One reason they do that is because when you are frustrated, it is better to imagine to battle against an imaginary dragon, rather than a little invisible Leprechaun who is playing tricks on you.

Especially in an easy environment like Linux/ROS/Python with coarse process-level parallelization, there is really little space for weird bugs to creep in. If you were using parallel C++ code, you would see lots of heisenbugs). Here, the reason is always something simple.

### 2) Truth: the fault is likely yours

The second truth is the following:

> While there are bugs in the system, it is more likely there is a bug in your code or in your environment.

## 11.3. What could it be?

### 1) 20%: Environment errors

Any problem that has to do with libraries not importing, commands not existing, or similar, are because the environment is not set up correctly. Biggest culprit: forgetting "source environment.sh" before doing anything, or rushing through the setup steps ignoring the things that failed.

### 2) 10%: Permission errors

Permission errors are most likely because people randomly used "sudo", thus creating root-owned files where they shouldn't be.

### 3) 9%: Bugs with the Duckietown software

Please report these, so that we can fix them.

### 4) 1%: Bug with ROS or other system library

Please report these, so that we can find workaround.

### 5) 10%: Problems with configuration files

Make sure that you have pulled `duckiefleet`, and pushed your changes.

Finally, given the questions we had so far, I can give you the prior distribution of mistakes:

### 6) 50%: Programming mistakes

Of these, 80% is something that would be obvious by looking at the stack trace and your code and could be easily fixed.

## 11.4. How to find the bug by yourself

### 1) Step 0: Is it late? Go to bed.

If it is later than 10pm, just go to bed, and look at it tomorrow.

After 10pm, bugs are introduced, rather than removed.

### 2) Step 1: Are you in a hurry? Do it another time.

Bug squashing requires a clear mind.

If you are in a hurry, it's better you do this another time; otherwise, you will not find the bug and you will only grow more frustrated.

### 3) Step 2: Make sure your environment is sane using `what-the-duck`

Finding a bug is a process of elimination of causes one-by-one, until you find the real culprit. Most of the problems come ultimately from the fact that your environment is not set up correctly.

We have a diagnostics program called `what-the-duck` that checks many things about the

environment.

➡ Subsection 8.1.3 - The what-the-duck program

So, first of all, run `what-the-duck`. Then, fix the errors that `what-the-duck` shows you.

This is the proper way to run `what-the-duck`:

```
$ cd ~/duckietown
$ source environment.sh
$ git checkout master
$ git pull
$ ./dependencies_for_duckiebot.sh # if you are on a Duckiebot
$ ./dependencies_for_laptop.sh  # if you are on a laptop
$ ./what-the-duck
```

**Note:** you have to do all the steps in the precise order.

The tool also produces an HTML report about your system which you should attach to any request for help.

## 11.5. How to ask for help?

I notice many people just writing: "I get this error: ... How can I fix it?". This is not the best way to get help. If you don't include the code and stack trace, it's hard to impossible to help you.

The best way to get help is the following:

Gold standard: Provide exact instructions on how to reproduce the error ("Check out this branch; run this command; I expect this; instead I get that"). This makes it easy for an instructor or TA to debug your problem in 30 seconds, give you the fix, and probably fix it for everybody else if it is a common problem.

Silver standard: Copy the relevant code to a Gist (gist.github.com) including the error stack trace. Because we have no way to reproduce the error, this starts a conversation which is basically guesswork. So you get half answers after a few hours.

## 11.6. How to give help

### 1) Step 1: Ask for the output of `what-the-duck`

If there are errors reported, the students should fix those before worrying about their current problem. Maybe you or they don't see the connection, but the connection might be there.

Also, in general, errors in the environment *will* cause other problems later on.

### 2) Step 2: Consider whether there are enough details to provide an informed answer

The worst thing you can do is guess work – this causes confusion.

I encourage the TAs to *not* answer any nontrivial question that is not at least at the silver standard. It is a waste of resources, it will likely not help, and it actually contributes to the confusion, with people starting to try random things until something works without understanding why things work, and ultimately creating a culture of

superstitions.

## Unit K-12

# Creating unit tests with ROS

```
catkin_make -C catkin_ws/ --pkg easy_logs
```

UNIT K-13

# Continuous integration

These are the conventions for the Duckietown repositories.

## 13.1. Never break the build

The `Software` and the `duckuments` repository use “continuous integration”.

This means that there are well-defined tests that must pass at all times.

For the `Software` repository, the tests involve building the repository and running unit tests.

For the `duckuments` repository, the tests involve trying to build the documentation using `make compile`.

If the tests do not pass, then we say that we have “broken the build”.

We also say that a branch is “green” if the tests pass, or “red” otherwise.

If you use the Chrome extension Pointless, you will see a green dot in different places on Github to signify the status of the build (Figure 13.1).

Figure 13.1. The green dot is good.

## 13.2. How to stay in the green

The system enforces the constraint that the branch `master` is always green, by preventing changes to the branches that make the tests fail.

We use a service called CircleCI. This service continuously looks at our repositories. Whenever there is a change, it downloads the repositories and runs the tests.

(It was a lot of fun to set up all of this, but fortunately you do not need to know how it is done.)

At this page you can see the summary of the tests. (You need to be logged in with your Github account and click “authorize Github”).

Figure 13.2. The CircleCi service dashboard, available at this page.

## 13.3. How to make changes to `master`: pull requests

It is not possible to push on to the master branch directly.

> ➡ See the Github documentation about pull requests to learn about the general concept.

The workflow is as follows.

(1) You make a private branch, say *your name*`-devel`.

(2) You work on your branch.

(3) You push often to your branch. Every time you push, CircleCI will run the tests and let you know if the tests are passing.

(4) When the tests pass, you create a "pull request". You can do this by going to the Github page for your branch and click on the button "compare and pull request" (Figure 13.3).

Figure 13.3. Compare and pull request button

(5) You now have an opportunity to summarize all the changes you did so far (Figure 13.4). Then click "create pull request".

Figure 13.4. Preparing the pull request

(6) Now the pull request has been created. Other people can see and comment on it. However, it has not been merged yet.

At this point, it might be that it says "Some checks haven't completed yet" (Figure 13.5). Click "details" to see what's going on, or just wait.

Figure 13.5. Wait for the checks to finish

When it's done, you will see either a success message (Figure 13.6) or a failure message

(Figure 13.7).

Figure 13.6. The tests are done

Figure 13.7. The tests have failed

(7) At this point, you can click "squash and merge" to merge the changes into master (Figure 13.8).

Figure 13.8. The final step

## 1) Troubleshooting

If you see a message like "merge failed" (Figure 13.9), it probably means that somebody pushed into master; merge master into your branch and continue the process.

Figure 13.9. Merge failed

UNIT K-14

# Road Release Process

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** You have implemented a new feature or improved an existing feature in your branch `devel-`*project_name*

**Requires:** A robot that is configured and able to follow lanes in Duckietown according to instructions in Unit M-24 - Checkoff: Navigation

**Results:** Your branch gets merged into `master` and doesn't break anything

This page is about what to do once you have developed and tested your new or improved feature and you want to contribute it back to the `master` branch.

**Note:** If your branch is not merged into `master` and passes all the tests it will be lost forever.

## 14.1. Merge `Master` into your branch

```
$ git merge origin master
```

## 14.2. Unit Tests

**TODO:** You should write some?

## 14.3. Continuous Integration

**TODO:** How to test that your branch is passing the tests? Look here ?

## 14.4. Regression Tests

If you have developed a perception module you should define a set of regression tests that ensure that it is working properly.

TODO:

## 14.5. Simulation Tests

**TODO:** Once the simulator is up and running it should become part of the testing infrastructure.

## 14.6. Hardware-in-the-loop (HWIL)

If you have been doing your development on your laptop or some other computer, the time is now to put the code on the RasPI and test.

## 14.7. Road Test

Verify that the previous functionality of the robot is preserved. For now repeat the instructions in Unit M-24 - Checkoff: Navigation, and ensure that the basic lane following and indefinite navigation abilities are preserved.

## 14.8. Make a Pull Request

Once all of the tests are passed and you are sufficiently convinced that your code is not going to break the system, you should make a Pull Request by going here. For the base branch put `master` and for the compare branch put your branch.

# Part L
# Duckietown system

This part describes the Duckietown algorithms and system architecture.

We do not go in the software details. The implementation details have been already talked about at length in Part K - Software development guide.

We do give links to the ROS packages implementing the functionality.

# Unit L-1
# Teleoperation

TODO: add video here

## 1.1. Implementation

Drivers:

- Unit P-1 - Package `adafruit_drivers`
- Unit P-4 - Package `pi_camera`

Operator interface:

- Unit P-3 - Package `joy_mapper`

## 1.2. Camera

TODO: to write

## 1.3. Actuators

TODO: to write

## 1.4. IMU

TODO: to write

UNIT L-2

# Parallel autonomy

TODO: to write

# Unit L-3
# Lane control

TODO: video here

## 3.1. Implementation

Perception:

- Unit Q-1 - Package `anti_instagram`
- Unit Q-5 - Package `ground_projection`
- Unit Q-9 - Package `line_detector`, Unit Q-8 - Package `line_detector2`
- Unit Q-7 - Package `lane_filter`

Control:

- Unit Q-6 - Package `lane_control`
- Unit P-2 - Package `dagu_car`

# UNIT L-4
# Indefinite navigation

TODO: add video here

## 4.1. Implementation

The packages involved in this functionality are:

- Unit R-2 - Package `apriltags_ros`
- Unit R-3 - Package `fsm`
- Unit R-4 - Package `indefinite_navigation`
- Unit R-5 - Package `intersection_control`
- Unit R-6 - Package `navigation`

> **Note:** we don't discuss the details of the packages here; we just give pointers to them.

# UNIT L-5
# Planning

TODO: add video here

## 5.1. Implementation

The packages involved in this functionality are:

- Unit S-2 - Package `localization`
- Unit S-1 - Package `duckietown_description`

> **Note:** we don't discuss the details of the packages here; we just give pointers to them.

# Unit L-6
# Coordination

TODO: add video here

## 6.1. Implementation

- Unit T-1 - Package `led_detection`
- Unit T-2 - Package `led_emitter`
- Unit T-3 - Package `led_interpreter`
- Unit T-4 - Package `led_joy_mapper`
- Unit T-6 - Package `traffic_light`
- Unit T-5 - Package `rgb_led`

# Unit L-7
# Duckietown ROS Guidelines

## 7.1. Node and Topics

In the source code, a node must only publish/subscribe to private topics.

In `rospy`, this means that the topic argument of `rospy.Publisher` and `rospy.Subscriber` should always have a leading `~`. ex: `~wheels_cmd`, `~mode`.

In `roscpp`, this means that the node handle should always be initialized as a private node handle by supplying with a ''~'' agrument at initialization. Note that the leading "~" must then be obmitted in the topic names of. ex:

```
ros::NodeHandle nh_(''~'');
sub_lineseglist_ = nh_.subscribe(''lineseglist_in'', 1, &GroundProjection::lineseglist_cb,
this);
pub_lineseglist_ = nh_.advertise<duckietown_msgs::SegmentList> (''lineseglist_out'', 1);
```

## 7.2. Parameters

All the parameters of a node must be private parameters to that node.

All the nodes must write the value of the parameters being used to the parameter server at initialization. This ensures transparence of the parameters. Note that the `get_param(name,default_value)` does not write the default value to the parameter server automatically.

The default parameter of `pkg_name/node_name` should be put in `~/duckietown/catkin_ws/src/duckietown/config/baseline/pkg_name/node_name/dafault.yaml`. The elemental launch file of this node should load the parameter using `<rosparam>`.

**Note:** The above is deprecated. The configuration is handled differently.

## 7.3. Launch file

Each node must have a launch file with the same name in the `launch` folder of the package. ex: `joy_mapper.py` must have a `joy_mapper.launch`. These are referred to as the elemental launch files.

Each elemental launch file must only launch one node.

The elemental launch file should put the node under the correct namespace through the `veh` arg, load the correct configuration and parameter file throught `config` and `param_file_name` args respectively. `veh` must not have a default value. This is to ensure the user to always provide the `veh` arg. `config` must be default to `baseline` and `param_file_name` must be default to `default`.

When a node can be run on the vehicle or on a laptop, the elemental launch file should provide a `local` arg. When set to true, the node must be launch on the launching machine, when set to false, the node must be launch on a vehicle throught the `mahcine` attribute.

A node should always be launched by calling its corresponding launch file instead of using `rosrun`. This ensures that the node is put under the correct namespace and all the necessary parameters are provided.

Do not use `<remap>` in the elemental launch files.

Do not use `<param>` in the elemental launch files.

# Part M
# Fall 2017

Welcome to the Fall 2017 Duckietown experience.

# Unit M-1
# The Fall 2017 Duckietown experience

This is the first time that a class is taught jointly across 3 continents!

There are 4 universities involved in the joint teaching for the term:

- ETH Zürich (ETHZ), with instructors Emilio Frazzoli, Andrea Censi, Jacopo Tani.
- University of Montreal (UdeM), with instructor Liam Paull.
- TTI-Chicago (TTIC), with instructor Matthew Walter.
- National Chiao Tung University (NCTU), with instructor Nick Wang.

This part of the Duckiebook describes all the information that is needed by the students of the four institutions.

At ETHZ, UdeM, TTIC, the class will be more-or-less synchronized. The materials are the same; there is some slight variation in the ordering.

Moreover, there will be some common groups for the projects.

The NCTU class is undergraduate level. Students will learn slightly simplified materials. They will not collaborate directly with the other classes.

## 1.1. The rules of Duckietown

**The first rule of Duckietown**

The first rule of Duckietown is: you don't talk about Duckietown, *using email*.

Instead, we use a communication platform called Slack.

There is one exception: inquiries about "meta" level issues, such as course enrollment and other official bureaucratic issues can be communicated via email.

**The second rule of Duckietown**

The second rule of Duckietown is: be kind and respectful, and have fun.

**The third rule of Duckietown**

The third rule of Duckietown is: read the instructions carefully.

Do not blindly copy and paste.

Only run a command if you know what it does.

# Unit M-2
# First Steps in Duckietown

## 2.1. Onboarding Procedure

Welcome aboard! We are so happy you are joining us at Duckietown!

This is your onboarding procedure. Please read all the steps and then complete all the steps.

If you do not follow the steps in order, you will suffer from unnecessary confusion.

### 1) Github sign up

If you don't already have a Github account, sign up now.

➡ Github signup page

Please use your full name when it asks you. Ideally, the username should be something like `FirstLast` or something that resembles your name.

When you sign up, use your university email. This allows to claim an educational discount that will be useful later.

### 2) Questionnaire

Taiwan For NCTU Students, complete this form:

NCTU Student Questionnaire

**NCTU: Please complete by 10/30**

For ETHZ, UdeM and TTIC fill in this Preliminary Student Questionnaire.

**Zurich: Please fill in questionnaire by Tuesday, September 26, 15:00 (extended from original deadline of 12:00).**

**Point of contact:** if you have problems with this step, please contact Jacopo Tani <tanij@ethz.ch>.

### 3) Accept invite to Github organization Duckietown

After we receive the questionnaire, we will invite you to the Duckietown organization. You need to accept the invite; until you do, you are not part of the Duckietown organization and can't access our repositories.

The invite should be waiting for you at this page.

### 4) Accept the invite to Slack

After we receive the questionnaire, we will invite you to Slack.

The primary mode of online confabulation between staff and students is Slack, a team communication forum that allows the community to collaborate in making Duckietown awesome.

(Emails are otherwise forbidden, unless they relate to a private, university-based administrative concern.)

We will send you an invite to Slack. Check your inbox.

If after 24 hours from sending the questionnaire you haven't received the invite, contact HR representative Kirsten Bowser <akbowser@gmail.com>.

**What is Slack?** More details about Slack are available here. In particular, remember to disable email notifications.

**Slack username**. When you accept your Slack invite, please identify yourself with first and last names followed by a "-" and your institution.

*example* Andrea Censi - Zurich

**Slack picture**. Please add a picture (relatively professional, with duckie accessories encouraged).

**Slack channels**. A brief synopsis of all the help-related Slack channels is here: Unit M-10 - Slack Channels.

Check out all the channels in Slack, and add yourself to those that pertain or interest you. Be sure to introduce yourself in the General channel.

## 2.2. (optional) Add Duckietown Engineering Linkedin profile

This is an optional step.

If you wish to connect with the Duckietown alumni network, on LinkedIn you can join the company "Duckietown Engineering", with the title "Vehicle Autonomy Engineer in training". Please keep updated your Linkedin profile with any promotions you might receive in the future.

## 2.3. Laptops

If you do not have access to a laptop that meets the following requirements, please post a note in the channel `#help-laptops`.

You need a laptop with these specifications:

- Linux Ubuntu 16.04 installed natively (dual boot), not in a virtual machine. See Subsection 2.3.1 - Can I use a virtual machine instead of dual booting? below for a discussion of the virtual machine option.
- A WiFi interface that supports 5 GHz wireless networks. If you have a 2.4 GHz WiFi, you will not be able to comfortably stream images from the robot; moreover, you will need to adapt certain instructions.
- Minimum 50 GB of free disk space in addition to the OS. Ideally you have 200 GB+. This is for storing and processing logs.

There are no requirements of having a particularly good GPU, or a particularly good CPU. You will be developing code that runs on a Raspberry PI. Any laptop bought in the last 3 years should be powerful enough. However, having a good CPU / lots of RAM makes it faster to run regression tests.

### 1) Can I use a virtual machine instead of dual booting?

Running things in a virtual machine is possible, but **not supported**.

This means that while there is a way to make it work (in fact, Andrea develops in a VMWare virtual machine on OS X), we cannot guarantee that the instructions will work on a virtual machine, and, most importantly, the TAs will *not* help you debug those problems.

The issues that you will encounter are of two types.

- There are performance issues. For example, 3D acceleration might not work in the virtual machine.
- Most importantly, there are network configuration issues. These come up late in the class, when you start connecting the laptop to the Duckiebot. At that point, ROS makes certain assumptions about subnets, that might not be satisfied by your virtual machine configuration. At that point, you need to be relatively skilled to fix it.

So, the required skill here is not "being able to install Ubuntu on a virtual machine", but rather "Being able to debug network problems involving multiple real/virtual networks and multiple real/virtual adapters".

Here's a quiz: do these commands look familiar to you?

```
$ route add default gw 192.168.1.254 eth0
$ iptables -A FORWARD -o eth1 -j ACCEPT
```

If so, then things will probably work ok for you.

Otherwise, we strongly suggest that you use dual booting instead of a virtual machine.

## 2.4. Next steps for people in Zurich

### 1) Get acquainted with class journal and class logistics

At this point, you should be all set up, able to access our Github repositories, and, most important of all, able to ask for help on Slack.

You can now get acquainted to the class journal, to know the next steps.

➡ Unit M-11 - Zürich branch diary

Also, in this page, we will collect the logistics information (lab times, etc.).

➡ Unit M-3 - Logistics for Zürich branch

### 2) Make sure you can edit the Duckuments

To receive your Duckiebox on Wednesday Sep 27, you need to prove to be able to edit the Duckuments successfully.

➡ See the instructions in this section.

If you can't come on Wednesday, please contact one of the TAs.

## 2.5. Next steps for people in Chicago

TODO: to write

# Unit M-3
# Logistics for Zürich branch

This section describes information specific to Zürich.

## 1) The local staff

These are the local TAs:

- Shiying Li (shili@student.ethz.ch)
- Ercan Selçuk (ercans@student.ethz.ch)
- Miguel de la Iglesia Valls (dmiguel@student.ethz.ch)
- Harshit Khurana (hkhurana@student.ethz.ch)
- Dzenan Lapandic (ldzenan@student.ethz.ch)
- Marco Erni (merni@ethz.ch)

Please contact them on Slack, rather than email.

Also feel free to contact the TAs in Montreal and Chicago.

## 3.1. HR

Feel free to contact Ms. Kirsten Bowser (akbowser@gmail.com) if you have problems regarding accounts, permissions, etc.

## 3.2. Website / class journal

During the term, we are not going to update the website.

Rather, all important information, such as deadlines, is in the class journal.

➡ Unit M-11 - Zürich branch diary

## 3.3. Duckiebox

The point of contact for Duckiebox distribution is Shiying Li.

## 3.4. Duckietown room access

The local Duckietown room is ML J 44.2.

**TODO:** write opening hours and rules

+ comment
double check the room number -AC

## 3.5. Extra spaces

There will be extra lab space available.

Space-time coordinates TBD.

# Unit M-4
# Logistics for Montréal branch

This unit contains all necessary info specific for students at Univeristé de Montréal.

## 4.1. Website

This is the official course website. It contains links to the the syllabus and description and other important info.

## 4.2. Class Schedule

The authoritative class schedule will be tracked in Unit M-12 - Montréal branch diary. This will contain all lecture material, homeworks, checkoffs, and labs.

## 4.3. Lab Access

The lab room for the class is 2333 in Pavillion André-Aisenstadt. The code for the door is XXX. Please do not distribute the code for the door, we are trying to limit access to this room as much as possible.

## 4.4. The Local Staff

The TA for the class is Florian Golemo. All communications with the course staff should happen through Slack.

The instructor is Prof. Liam Paull, whose office is 2347 Pavillion André-Aisenstadt.

## 4.5. Storing Your Robot

It is preferable that you keep your robot for the semester. However, if you do not have a secure location where you can store it, we can store it for you in Room XXX in Pavillion André-Aisenstadt. However, you will have to ask Prof. Liam Paull to access or store your robot there each time since we cannot give out access to this space to the students in the class.

# Unit M-5
# Logistics for Chicago branch

> Assigned to: Matt

This section describes information specific to TTIC and UChicago students.

## 1) Website

The course website provides a copy of the syllabus, grading information, and details on learning objectives.

## 2) Class Schedule

Classes take place on Mondays and Wednesdays from 9am-11am in TTIC Room 530. In practice, each class will be divided into an initial lecture period (approximately one hour), followed by a lab session.

The class schedule is maintained as part of the TTIC Class Diary, which includes details on lecture topics, links to slides, etc.

## 3) Course Grading

The following is taken from the course syllabus:

The class will assess your grasp of the material through a combination of problem sets, exams, and a final project. The contribution of each to your overall grade is as follows:

- 20%: Problem sets
- 10%: Checkoffs
- 20%: Participation
- 50%: Final project (includes report and presentation). The projects will be group-based, but we will assess the contribution of each student individually.

See the course syllabus for more information on how the participation and final project grades are determined.

## 4) Policy on Late Assignments and Collaboration

The following is taken from the course syllabus:

Late problem sets will be penalized 10% for each day that they are late. Those submitted more than three days beyond their due date will receive no credit.

Each student has a budget of three days that they can use to avoid late penalties. It is up to the student to decide when/how they use these days (i.e., all at once or individually). Students must identify whether and how many days they use when they submit an assignment.

It is not acceptable to use code or solutions from outside class (including those found online), unless the resources are specifically suggested as part of the problem set.

You are encouraged to collaborate through study groups and to discuss problem sets and the project in person and over Slack. However, you must acknowledge who you worked with on each problem set. You must write up and implement your own solutions and are not allowed to duplicate efforts. The correct approach is to discuss solu-

tion strategies, credit your collaborator, and write your solutions individually. Solutions that are too similar will be penalized.

## 5) Lab Access

Duckietown labs will take place at TTIC in the robotics lab on the 4th floor.

**Note**: TTIC and U. Chicago students in Matthew Walter's research group use the lab as their exclusive research and office space. It also houses several robots and hardware to support them. Please respect the space when you use it: try not to distract lab members while they are working and please don't touch the robots, sensors, or tools.

## 6) The Local LAs

Duckietown is a collaborative effort involving close interaction among students, TAs, mentors, and faculty across several institutions. The local learning assistants (LAs) at TTIC are:

- Andrea F. Daniele (afdaniele@ttic.edu)
- Falcon Dai (dai@ttic.edu)
- Jon Michaux (jmichaux@ttic.edu)

# Unit M-6
# Logistics for NCTU branch

> **Assigned to:** Nick and Eric (Nick's student)

This section describes information specific to NCTU students.

## 1) Website

The Duckietown Taiwan Branch Website provides some details about Duckietown Branch in NCTU-Taiwan and results of previous class in NCTU.

## 2) Class Schedule

Classes take place on Thursday from 1:20pm~4:20om in NCTU Engineering Building 5 Room 635. Each class will be devided into two sessoins. In the first session, Professor Wang will give lessons on foundamental theory and inspire students to come up more creatitive but useful ideas on final projects. In the second sessoin, TAs will give pratical lab on how to use Duckietown platform as their project platform and use ROS as their middleware toward a fantastic work.

The class schedule is maintained as part of the NCTU Class Diary, which includes details on lecture topics, links to slides, etc.

## 3) Course Grading

The following is taken from the course syllabus:

This course aims at developing software projects usable in real-world, and focuses on "learning by doing," "team work," and "research/startup oriented.". The contribution of each to your overall grade is as follows:

- Class Participation, In Class Quiz, Problem Sets (10%)
- Midterm Presentation (30%)
- Final Presentation (30%)
- Project Report and Demo Video (30%)

See the course syllabus for more information on course object and grading policy.

## 4) Policy on Late Assignments and Collaboration

The following is taken from the course syllabus:

Late problem sets will be penalized 10% for each day that they are late. Those submitted more than three days beyond their due date will receive no credit.

Each student has a budget of three days that they can use to avoid late penalties. It is up to the student to decide when/how they use these days (i.e., all at once or individually). Students must identify whether and how many days they use when they submit an assignment.

It is not acceptable to use code or solutions from outside class (including those found online), unless the resources are specifically suggested as part of the problem set.

You are encouraged to collaborate through study groups and to discuss problem sets and the project in person and over Slack. However, you must acknowledge who you

worked with on each problem set. You must write up and implement your own solutions and are not allowed to duplicate efforts. The correct approach is to discuss solution strategies, credit your collaborator, and write your solutions individually. Solutions that are too similar will be penalized.

### 5) Lab Access

Duckietown labs will take place at NCTU in the same place with the lecture.

**Note**: The course focus on "learning by doing" which means that the lab session of each class is expectially important.

### 6) The Local LAs

Duckietown is a collaborative effort involving close interaction among students, TAs, mentors, and faculty across several institutions. The local learning assistants (LAs) at NCTU are:

- Yu-Chieh 'Tony' Hsiao (tonycar12002@gmail.com)
- Pin-Wei 'David' Chen (ccpwearth@gmail.com)
- Chen-Lung 'Eric' Lu (eric565648@gmail.com)
- Yung-Shan 'Michael' Su (michael1994822@gmail.com)
- Chen-Hao 'Peter' Hung (losttime1001@gmail.com)
- Hong-Ming 'Peter' Huang (peterx7803@gmail.com)
- Tzu-Kuan 'Brian' Chuang (fire594594594@gmail.com)

UNIT M-7

# Git usage guide for Fall 2017

## 7.1. Differences

There are some difference among the branches. These will be marked using the following graphical notation.

?? This is only for Zurich.

?? This is only for Montreal.

?? This is only for Chicago.

Taiwan This is only for Taiwan.

## 7.2. Repositories

These are the repositories we use.

### 1) Software

The Software repository is the main repository that contains the software.
The URL to clone is:

```
git@github.com:duckietown/Software.git
```

In the documentation, this is referred to as `DUCKIETOWN_ROOT`.
During the first part of the class, you will only read from this repository.

### 2) Duckiefleet

The duckiefleet repository contains the data specific to this instance of the class.
The URL to clone is:

```
git@github.com:duckietown/duckiefleet.git
```

In the documentation, the location of this repo is referred to as `DUCKIEFLEET_ROOT`.
You will be asked to write to this repository, to update the robot DB and the people DB, and for doing exercises.

### 3) exercises-fall2017

For homework submissions, we will use the following URL:

```
git@github.com:duckietown/exercises-fall2017.git
```

As explained below, it is important that this repo is kept separate so that students can privately work on their exercises at schools where the homeworks are counted for grades.

#### 4) Duckuments

The Duckuments repository is the one that contains this documentation.

The URL to clone is:

```
git@github.com:duckietown/duckuments.git
```

Everybody is encouraged to edit this documentation!

In particular, feel free to insert comments.

#### 5) Lectures

The lectures repository contains the lecture slides.

The URL to clone is:

```
git@github.com:duckietown/lectures.git
```

Students are welcome to use this repository to get the slides, however, please note that this is a space full of drafts.

#### 6) Exercises

The exercises repository contains the solution to exercises.

The URL to clone is:

```
git@github.com:duckietown/XX-exercises.git
```

Only TAs have read and write permissions to this repository.

## 7.3. Git policy for homeworks (TTIC/UDEM)

?? This does not apply to Zurich.

Homeworks will require you to write and submit coding exercises. They will be submitted using git. Since we have a university plagiarism policy (UdeM's, TTIC/UChicago) we have to protect students work before the deadline of the homeworks. For this reason we will follow these steps for homework submission:

1. Go here and file a request at the bottom "Request a Discount" then enter your institution email and other info.
2. Go to exercises-fall2017
3. Click "Fork" button in the top right
4. Choose your account if there are multiple options
5. Click on the Settings tab of your repostory, not your account

6. Under "Collaborators and Teams" in the left, click the "X" in the right for the section for "Fall 2017 Vehicle Autonomy Engineers in training". You will get a popup asking you to confirm. Confirm.

If you have not yet cloned the duckietown repo do it now:

```
$ git clone git@github.com:duckietown/exercises-fall2017.git
```

Now you need to point the remote of your `exercises-fall2017` to your new local private repo. To do, from inside your already previously cloned `exercises-fall2017` repo do:

```
$ git remote set-url origin git@github.com:GIT_USERNAME/exercises-fall2017.git
```

Let's also add an `upstream` remote that points back to the original duckietown repo:

```
$ git remote add upstream git@github.com:duckietown/exercises-fall2017.git
```

If you type

```
$ git remote -v
```

You should now see:

```
origin  git@github.com:GIT_USERNAME/exercises-fall2017.git (fetch)
origin  git@github.com:GIT_USERNAME/exercises-fall2017.git (push)
upstream  git@github.com:duckietown/exercises-fall2017.git (fetch)
upstream  git@github.com:duckietown/exercises-fall2017.git (push)
```

Now the next time you push (without specifying a remote) you will push to your local private repo.

### 1) Duckiefleet file structure

You should put your homework files in folder at:

*DUCKIEFLEET_HOMEWORK_ROOT*/homeworks/*XX_homework_name*/*YOUR_ROBOT_NAME*

Some homeworks might not require ROS, they should go in a subfolder called `scripts`. ROS homeworks should go in packages which are generated using the process described here: Unit K-6 - Minimal ROS node - `pkg_name`. For an example see *DUCKIEFLEET_HOMEWORK_ROOT*/homeworks/01_data_processing/shamrock.

> **Note:** To make your ROS packages findable by ROS you should add a symlink from your *DUCKIEFLEET_HOMEWORK_ROOT* to `duckietown/catkin_ws/src`.

### 2) To submit your homework

When you are ready to submit your homework, you should do **create a release** and

**tag the Fall 2017 instructors/TAs group** to let us know that your work is complete. This can be done through the command line or through the github web interface:

Command line:

```
$ git tag XX_homework_name -m"@duckietown/fall-2017-instructors-and-tas homework complete"
$ git push origin --tags
```

Through Github:

1. Click on the "Releases" tab.
2. Click "Create a new Release".
3. Add a version (e.g. 1.0).
4. Release title put *XX_homework_name*.
5. In the text box put "@duckietown/fall-2017-instructors-and-tas homework complete".
6. Click "Publish release".

You may make as many releases as you like before the deadline.

### 3) Merging things back

Once all deadlines have passed for all institutions, we can merge all the homework. We will ask to create a "Pull Request" from your private repo.

1. In your private `exercises-fall2017` repo, click the "New pull request button".
2. Click "Create pull request" green button
3. The 4 drop down menus at the top should be left to right: (`base fork: duckietown/exercises-fall2017`, `base: master`, `head fork:` *YOUR_GIT_NAME*`/exercises-fall2017`, `compare:` *YOUR_BRANCH*)
4. Leave a comment if you like and click "Create pull request" green button below.
5. At some point a TA or instructor will either merge or leave you a comment.

### 4) For U de M students who have already submitted homework to the previous duckiefleet-2017 repo

?? These instructions assume that you are ok with losing the commit history from the first homework. If not, things get a little more complicated

Fork and clone the new "homework" repository using the process above. Followed by:

```
$ git clone git@github.com:GIT_USERNAME/exercises-fall2017.git
```

Copy over your homework files from the `duckiefleet-fall2017` repo into the `exercises-fall2017` repo.

`git rm` your folder from `duckiefleet-fall2017` and commit and push.

`git add` your folder to `exercises-fall2017` and commit and push.

Clone the new duckiefleet repo

```
$ git clone git@github.com:duckietown/duckiefleet.git
```

Update the symlink you created in your duckietown repo

```
$ ln -sf EXERCISES_FALL2017/homeworks $DUCKIETOWN_ROOT/catkin_ws/
src/name-of-the-symlink
```

## 7.4. Git policy for project development

Different than the homeworks, development for the projects will take place in the `Software` repo since plagiarism is not an issue here. The process is:

1. Create a branch from master
2. Develop code in that branch (note you may want to branch your branches. A good idea would be to have your own "`master`", e.g. "`your_project-master`" and then do pull requests/merges into that branch as things start to work)
3. At the end of the project submit a pull request to master to merge your code. It may or may not get merged depending on many factors.

# Unit M-8
# Organization

The following roster shows the teaching staff.

Andrea Censi

Liam Paull

Jacopo Tani

First Last

Emilio Frazzoli

First Last

First Last

First Last

First Last

First Last

First Last

First Last

First Last

First Last

First Last

Greta Paull

Kirsten Bowser

Staff: To add yourself to the roster, or to change your picture, add a YAML file and a jpg file to the `duckiefleet-fall2017` repository. in the `people/staff` directory.

## 8.1. The Activity Tracker

Link to Activity Tracker

The sheet called "Activity Tracker" describes specific tasks that you must do in a certain sequence. Tasks include things like "assemble your robot" or "sign up on Github".

The difference between the Areas sheet and the Task sheet is that the Task sheet contains tasks that you have to do once; instead, the Areas sheet contains ongoing activities.

In this sheet, each task is a row, and each person is a column. There is one column for each person in the class, including instructors, TAs, mentors, and students.

You have two options:

- Only use the sheet as a reference;
- Use the sheet actively to track your progress. To do this, send a message to Kirsten with your gmail address, and add yourself.

Each task in the first column is linked to the documentation that describes how to perform the task.

The colored boxes have the following meaning:

- Grey: not ready. This means the task is not ready for you to start yet.
- Red: not started. The person has not started the task.
- Blue: in progress. The person is doing the task.
- Yellow: blocked. The person is blocked.
- Green: done. The person is done with the task.
- n/a: the task is not applicable to the person. (Certain tasks are staff-only.)

## 8.2. The Areas sheet

Please familiarize yourself with this spreadsheet and bookmark it in your browser.

The sheet called "Areas" describes the points of contact for each part of this experience. These are the people that can offer support. In particular, note that we list two points of contact: one for America, and one for Europe. Moreover, there is a link to a Slack channel, which is the place where to ask for help. (We'll get you started on Slack in just a minute.)

# UNIT M-9
# Getting and giving help

## 9.1. *Who* to ask for help

### 1) Primary points of contact

The organization chart (Section 8.2 - The Areas sheet) lists the primary contact for each area.

### 2) Point of contacts for specific documents

Certain documents have specific points of contacts, listed at the top. These override the listing in the organization chart.

## 9.2. *How* to ask for help

The ways that we will support each other will depend on the type of situation. Here we will enumerate the different cases. Try to figure out which case is the most appropriate and act accordingly. These are ordered roughly in order of increasing severity.

### 1) Case: You find a mistake in the documentation

Action: Please fix it.

The goal for the instructions is that anybody is able to follow them. Last year, we managed to have two 15-year-old students reproduce the Duckiebot from instructions.

> ➡ How to edit the documentation is explained in Part B - Duckumentation documentation. In particular, the notation on how to insert a comment is explained in Section 3.3 - Notes and questions.

Note that because we use Git, we can always keep track of changes, and there is no risk of causing damage.

If you encounter typos, feel free to edit them directly.

Feel free to add additional explanations.

One thing that is very delicate is dealing with mistakes in the instructions.

A few times the following happened: there is a sequence of commands `cmd1;cmd2;cmd3` and `cmd2` has a mistake, and `cmd2b` is the right one, so that the sequence of commands is `cmd1;cmd2b;cmd3`. In those situations we first just corrected the command `cmd2`.

However, that created a problem: now half of the students had used `cmd1;cmd2;cmd3` and half of the students had used `cmd1;cmd2b;cmd3`: the states had diverged. Now chaos might arise, because there is the possibility of "forks".

Therefore, if a mistaken instruction is found, rather than just fixing the mistake, please add an addendum at the end of the section.

For example: "Note that instruction `cmd2` is wrong; it should be `cmd2b`. To fix this, please enter then command `cmd4`".

Later, when everybody has gone through the instructions, the mistake is fixed and the addendum is deleted.

### 2) Case: You find the instructions unclear and you need clarification

Action: Ask for clarification on the appropriate Slack channel. For a list of slack channels that could be helpful see Unit M-10 - Slack Channels. Once the ambiguity is clarified to your satisfaction, either you or the appropriate staff member should update the documentation if appropriate. For instructions on this see Part B - Duckumentation documentation.

### 3) Case: You understand the instructions but you are blocked for some reason

Action: This is more serious than the previous. Open an issue on the `duckiefleet-fall2017` github page. Once the issue is resolved, either you or the appropriate staff member should update the documentation if appropriate. For instructions on this see Part B - Duckumentation documentation.

### 4) Case: You are having a technical issue related to building the documentation

Action: Open an issue on the `duckuments` github page and provide all necessary information to reproduce it.

### 5) Case: You have found a well-defined defect in the software.

Action: open an issue on the `Software` repository github page and provide all necessary information for reproducing the bug.

# Unit M-10
# Slack Channels

This page describes all of the helpful Slack channels and their purposes so that you can figure out where to get help.

## 10.1. Channels

You can also easily join the ones that you are interested in by clicking the links in this message.

Table 10.1. Duckietown Slack Channels

| Channel | Purpose |
|---|---|
| `help-accounts` | Info about necessary accounts, such as Slack, Github, etc. |
| `help-assembly` | Help putting your robot together |
| `help-camera-calib` | Help doing the intrinsic and extrinsic calibration of your camera |
| `help-duckuments` | Help compiling the online documentation |
| `help-git` | Help with git |
| `help-infrastructure` | Help with software infrastructure, such as Makefiles, unit tests, continuous integration, etc. |
| `help-laptops` | Help getting your laptop setup with Ubuntu 16.04 |
| `help-parts` | Help getting the parts for the robot or replacement parts if you broke something |
| `help-robot-setup` | Help getting the robot setup to do basic things like be driven with a joystick |
| `help-ros` | Help with the Robot Operating System (ROS) |
| `help-wheel-calib` | Help doing your odometry calibration |

+ comment

Note that we can link directly to the channels. (See list in the org sheet.) -AC

UNIT M-11

# Zürich branch diary

## 11.1. Lectures and lab sessions

Lectures: Mon 13-15 HG F 26.5 Wed 10-12 HG E 22

Lab session: Fri 15-19 ML J 37.1

Duckielab: ML J 44.2

## 11.2. Wed Sep 20: Welcome to Duckietown!

This was an introduction meeting.

### 1) Material presented in class

These are the slides we showed:

- PDF
- Keynote (huge)

### 2) Feedback form

Please help us making the experience better by providing feedback (can be anonymous)

### 3) Pointers to reading materials

Read about Duckietown's history and watch the Duckumentary.

➡ Part A - The Duckietown project

Start learning about Git and Github.

➡ Unit J-27 - Source code control with Git

Montreal, Chicago? What's happening?

➡ Unit M-1 - The Fall 2017 Duckietown experience

## 11.3. Monday Sep 25: Introduction to autonomy

### 1) Material presented in class

These are the slides we presented:

a - Logistics: Keynote, PDF.

a - Autonomous Vehicles: Keynote, PDF.

c - Autonomous Mobility on Demand: Keynote, PDF.

d - Plan for the next months: Keynote, PDF.

### 2) Feedback form

Please help us making the experience better by providing feedback (can be anonymous)

### 3) Questions and answers

*Q: MyStudies is not updated; I am still on the waiting list. What should I do?*

**Answer:** Nothing. Don't worry, if you have received the onboarding email, you are in the class, even if you still appear in the waiting list. We will figure this out with the department.

*Q: What version of Linux do I need to install?*

**Answer:** 16.04.* (16.04.03 is the latest at time of this writing)

*Q: Do I need to install OpenCV, ROS, etc.?*

**Answer:** Not necessary. We will provide instructions for those steps.

*Q: My laptop is not ready. I'm having problems installing Linux on a partition.*

**Answer:** Don't worry, take a Duckie, and, take a breath. We have time to fix every issue. Start by asking for help in the #help-laptops channel in Slack. We will address the outstanding issues in the next classes.

*Q: How much space do I need on my Linux partition?*

**Answer:** At least 50 GB; 200 GB are recommended for easy processing of data (logs) later in the course. If you have less space (say ~100GB), it might be wise to acquire an external hard drive to use as storage.

*Q: Are there going to be Linux training sessions?*

**Answer:** Maybe. We didn't plan for it, but it seems that there is a need. Subject to figuring out the logistics, we might organize an extra "lab" session or produce a support video.

## 11.4. Monday Sep 25, late at night: Onboarding instructions

At some late hour of the night, we sent out the onboarding instructions.

➡ Section 2.1 - Onboarding Procedure

**Please complete the onboarding questionnaire by Tuesday, September 26, 15:00.**

## 11.5. Wednesday Sep 27: Duckiebox distribution, and getting to know each other

Today we distribute the Duckieboxes and we name the robots. In other words, we perform the *Duckiebox ceremony*.

- getting to know each other;
- naming the robots;
- distribute the Duckieboxes.

> Note: If you cannot make it to this class for the Duckiebox distribution, please inform the TA, to schedule a different time.

### 1) Preparation, step 1: choose a name for your robot

Before arriving to class, you must think of a name for your robot.

Here are the constraints:

- The name must work as a hostname. It needs to start with a letter, contains only letters and numbers, and no spaces or punctuation.
- It should be short, easy to type. (You'll type it a lot.)
- It cannot be your own name.
- It cannot be a generic name like "robot", "duckiebot", "car". It cannot contain brand names.

#### 2) Preparation, step 2: prepare a brief elevator pitch

As members of the same community, it is important to get to know a little about each other, so to know who to rely on in times of need.

During the Duckiebox distribution ceremony, you will be asked to walk up to the board, write your name on it, and introduce yourself. Keep it very brief (30 seconds), and tell us:

- what is your professional background and expertise / favorite subject;
- what is the name of your robot;
- why did you choose to name your robot in that way.

You will then receive a Duckiebox from our senior staff, a simple gesture but of sempiternal glory, for which you have now become a member of the Duckietown community. This important moment will be remembered through a photograph. (If in the future you become a famous roboticist, we want to claim it's all our merit.)

Finally, you will bring the Duckiebox to our junior staff, who will apply labels with your name and the name of the robot. They will also give you labels with your robot name for future application on your Duckiebot.

#### 3) Feedback form

Please help us making the experience better by providing feedback (can be anonymous)

#### 4) Material presented in class

- Duckiebot parts: PowerPoint presentation, PDF.

## 11.6. Thursday Sep 28: Misc announcements

We created the channel `#ethz-chitchat` for questions and other random things, so that we can leave this channel `#ethz-announcements` only for announcements.

We sent the final list to the Department; so hopefully in a couple of days the situation on MyStudies is regularized.

The "lab" time on Friday consists in an empty room for you to use as you wish, for example to assembe the robots together. In particular, it's on the same floor of the Duckietown room and the IDSC lab.

The instructions for assembling the Duckiebots are here. Note that you don't have to do the parts that we did for you: buying the parts, soldering the boards, and reproducing the image.

Expected progress: We are happy if we see everybody reaching up to Unit C-14 - RC+camera remotely by Monday October 9. You are encouraged to start very early; it's likely that you will not receive much help on Sunday October 8...

## 11.7. Sep 28: some announcements

A couple of announcements:

1. We created `#ethz-chitchat` for questions and other random things, so that we can leave this channel #ethz-announcements only for announcements.
2. MyStudies should be updated with everybody's names.
3. The "lab" time tomorrow consists in an empty room for you to use as you wish, for example to assemble the robots together. In particular, it's on the same floor of the Duckietown room and the IDSC lab.
4. The instructions for assembling the Duckiebots are here. Note that we did for you step I-2 (buying the parts) and I-3 (soldering the boards); and I-6 is optional.
5. We are happy if we see everybody reaching I-13 by the Monday after next. I encourage you to start sooner than later.
6. I see only 30 people in this channel instead of 42. Please tell your friends that now all the action is on Slack.

## 11.8. Oct 01 (Mon): Announcement

It looks like that the current documentation is misleading in a couple of points. This is partially due to the fact that there is some divergence between Chicago, Montreal and Zurich regarding (1) the parts given out and (2) the setup environment (which networks are available). We did the simple changes (e.g. removing the infamous part 6), but we need some more time to review the other issues. At this point, the best course of action is that you enjoy your weekend without working on Duckietown, while we spend the weekend fixing the documentation.

## 11.9. Oct 02 (Mon): Networking, logical/physical architectures

### 1) Materials presented in class

a - Logistics and other information: Keynote, PDF.

b - Networking Keynote, PDF.

c - System architecture basics Keynote, PDF.

DUCKIETOWN WEEKLY NEWS Oct. 2, 2017

ZÜRICH BRANCH OPENS

Duckies rejoice, hopeful of swift construction of AMOD service.

Who are the men without the duckie?

Speculations run wild regarding the identity and motives of three men who were in the picture without a duckie on their head.

Instruction confusion leads to assembling halt

Following the sudden realization that Montréal, Chicago, and Zürich have 3 different robot configurations, assembly and setup has been stopped while the instructions are being updated. Officials are hopeful that there will be a quick convergence of all branches to same configuration.

Rumors of Chicago and Montréal branches opening

No photographic evidence to be found.

Duckietown construction started

Construction suddenly halted when crew realized mats were upside down.

Duckieboxes distributed

Commentators say boxes are observed to be similar to chunks of Swiss cheese. Holes planned in future upgrade.

Popular vote dictates networking lecture

17 2

Non-anonymity of poll said to bias the results. Anonymous polls planned in the future.

Duckietown critiqued over use of non-free software

Free software advocate Richard Stallman critiques Duckietown over the choice of Github, due to the website's use of non-free software.

Figure 11.1

### 2) Feedback form

Please help us making the experience better by providing feedback (can be anonymous)

## 11.10. Oct 04 (Wed): Modeling

### 1) Materials presented in class

- Modeling of a differential drive vehicle: PowerPoint presentation, PDF.

### 2) Feedback form

Please help us making the experience better by providing feedback (can be anonymous)

## 11.11. Oct 09 (Mon): Autonomy architectures and version control

### 1) Materials presented in class

a - Autonomy Architectures Basics: Keynote, PDF.

b - Version control with Git: Keynote, PDF.

## 11.12. Oct 11 (Wed): Computer vision and odometry calibration

### 1) Materials presented in class

a - Computer Vision Basics: PDF, PowerPoint presentation.

b - Odometry Calibration: PDF, PowerPoint presentation.

## 11.13. Oct 13 (Fri): new series of tasks out

### 1) Taking a video of the joystick control

Please take a video of the robot as it drives with joystick control, as described in Section 17.7 - Upload your video and upload it according to the instructions.

Example of a great video, but with a nonconforming Duckiebot.

### 2) Camera calibration

Go forth and calibrate the camera! And get help in `#help-camera-calib`.

### 3) Wheel calibration

This is not ready yet! will be ready in a day or so.

### 4) Taking a log check off

Follow the instructions here to learn how to take a log.

### 5) Data processing exercises

See the list of exercises here.

Get help in `#ex-data-processing`.

TABLE 11.1. CURRENT DEADLINES FOR ZURICH

| | |
|---|---|
| Robot assembly | **overdue** |
| Robot/laptop configuration | **overdue** |
| Subsection 11.13.1 - Taking a video of the joystick control | Monday Oct 16 |
| Subsection 11.13.2 - Camera calibration | Friday Oct 20 |
| Subsection 11.13.3 - Wheel calibration | not ready yet |
| Subsection 11.13.4 - Taking a log check off | Wed Oct 18 |
| Subsection 11.13.5 - Data processing exercises | Monday Oct 23 |

## 11.14. Oct 16 (Mon): Line detection

### 1) Materials presented in class

a - Line detection: Keynote, PDF.

b - Logistics: Keynote, PDF.

## 11.15. Oct 18 (Wed): Feature extraction

### 1) Materials presented in class

- Feature extraction: Keynote, PDF.

## 11.16. Oct 20 (Fri): Lab session

1) Materials presented in class

- ROS Main concepts: PowerPoint presentation, PDF.

## 11.17. Oct 23 (Mon) Filtering I

1) Materials presented in class

- Lectures on filtering (Filtering I): PowerPoint presentation, PDF.

## 11.18. Oct 25 (Wed) Filtering II

a - Lectures (Particle Filter) PowerPoint presentation, PDF.

b - Lectures (Lane Filter) PowerPoint presentation, PDF.

## 11.19. Nov 1 (Wed) Control Systems

a - Lectures (Control Systems Module I) PowerPoint presentation, PDF.

b - Lectures (Control Systems Module II) PowerPoint presentation, PDF.

Points to be noted - Running what-the-duck on laptop and Duckiebot is mandatory. It helps save time in debugging errors and also is a standard way to ask for help from the staff. Keep repeating it periodically so as to keep the data up-to date - For the people lacking calibrated wheels, this serves as a reminder to calibrate the wheels and keep their duckiebot up-to date - It is advised to fill the lecture feedback form (Feedback form), so as to increase the effectiveness of the lectures - Always check the consistency of the camera calibration checkerboard before camera calibration (one has to check for the correct square size and correct distance between world and checkerboard reference)

## 11.20. Nov 6 (Mon) Project Pitches

Lecture Project Pitches PDF.

## 11.21. Nov 8 (Wed) Motion Planing

Lecture Motion Planing PDF.

A few references for planning of Andrea Censi:

- The Book on planning is the one by Lavalle, available for free here: http://planning.cs.uiuc.edu/.
- The mentioned robot BB-8: http://starwars.wikia.com/wiki/BB-8.
- The mendioned movie scene Donnie Darko: https://www.youtube.com/watch?v=rPeGaos7DB4.

## 11.22. Nov 13 (Mon) Project Team Assignments

- First Lecture: Project Team Assignments PDF.

- Second Lecture: First meeting of the Controllers group –> Filling out the Preliminary Design Document

## 11.23. Nov 15 (Wed) Putting things together

- First Lecture: Putting things together PDF.
- Second Lecture: Second meeting of the Controllers group –> Filling out the Preliminary Design Document

## 11.24. Nov 20 (Mon) Testing Autonomous Vehicles{#Zurich-2017-11-20}

- First Lecture: Testing Autonomous Vehicles PDF.

## 11.25. Nov 22 (Wed) Fleet Control

- Lecture: Fleet Control in Autonomous Mobility on Demand PDF.

## 11.26. Nov 27 (Mon) Inermediate design Report

- First Lecture: The intermediate Design report is introduced heretemplate-int-report. It is due on Monday 4th of December.
- Second Lecture was left for project discussion and interaction.

## 11.27. Nov 29 (Wed) Fleet Control

- First Lecture: Clausdio finished Fleet Control in Autonomous Mobility on Demand PDF.
- Second Lecture Julian Presented the state of the art in data driven vs Model driven robotics. PDF

# Unit M-12
# Montréal branch diary

## 12.1. Wed Sept 6

Class (11:30)

Slides:

- Duckietown History Future (keynote) (pdf)
- Duckietown Intro (keynote) (pdf)
- Autonomy Overview (keynote) (pdf)
- Autonomous Vehicles (keynote) (pdf)

Book materials:

- Part A - The Duckietown project
- Unit H-1 - Autonomous Vehicles
- Unit H-2 - Autonomy overview

## 12.2. Friday Sept 8

Acceptance emails sent

## 12.3. Sun Sept 10

- Onboarding email sent to accepted students

## 12.4. Mon Sept 11

Class (10:30)

> **Note:** This class we are meeting in rm. 2333 Pavillion André Aisenstadt.

- Logistics
- The Duckiebook
- Slack (Unit M-10 - Slack Channels)
- Git repos (Unit M-7 - Git usage guide for Fall 2017)
- How to get and give help (Unit M-9 - Getting and giving help)
- Grading scheme
- Student/Staff Introductions
- Duckiebox distribution.
- Go through the Duckiebox parts
- Unit M-17 - Checkoff: Assembly and Configuration initiated.

Deadline: **Mon Sept. 25**

## 12.5. Wed Sept 13

Class canceled. Continue working on Unit M-17 - Checkoff: Assembly and Configuration.

## 12.6. Mon Sept 18

Class (10:30 - 11:30)

- General discussion (how are things going? Sorry I was away last week.. anyone need anything?)
- Intro to robotics - Modern robotic systems
- The robot as a system - System architectures
- Decomposing the robotics sytems into smaller pieces - autonomy architectures
- Agreeing on the language that the different pieces "talk" - representations
- Background on basic probability theory?

Slides:

- Modern Robotic Systems (keynote) (pdf)
- System Architecture (keynote) (pdf)
- Autonomy Architectures (keynote) (pdf)
- Representations for Robotics (keynote) (pdf)

Book Materials:

- Modern Robotic Systems
- System Architecture Basics
- Autonomy Architectures
- Representations
- Probability Basics

Lab (11:30 - 12:30)

- Liam will be in 2333 setting up network and building Duckietown.

## 12.7. Wed Sept 20

Class (11:30 - 12:30)

- Robotics middlewares - what are they and basic concepts
- Introduction to the Robot Operating System (ROS)

Slides:

- Software Architectures (keynote) (pdf)
- Introduction to ROS (keynote) (pdf)

Book Materials:

- Introduction to ROS
- Middlewares

Lab (12:30 - 1:30)

Homeworks and Checkoffs:

- Unit M-17 - Checkoff: Assembly and Configuration deadline extended to **Monday Sept 25**. Submit by clicking here.

## 12.8. Mon Sept 25

Class (10:30-11:30)

- Microelectronics (pptx) (pdf)
- Modern Signal Processing (keynote) (pdf)
- Intro to Networking (keynote) (pdf)

Book Materials (still rough drafts, will be completed soon):

Lab (11:30 - 12:30)

- Liam will be in 2333
- Any final help needed for Unit M-17 - Checkoff: Assembly and Configuration
- Finalize Duckietown map setup

Homeworks and Checkoffs:

- Unit M-17 - Checkoff: Assembly and Configuration due by 11pm.
- New checkoff: "Taking a log" will be linked later today (due Monday Oct 2)
- New homework: "Data processing" will be linked later today (due Monday Oct 2)

## 12.9. Wed Sept 27

Class (11:30 - 12:30) in Z-305

- USB drive distribution
- New checkoff announced: Unit M-18 - Checkoff: Take a Log
- New homework announced: Unit M-20 - Homework: Data Processing (UdeM)
- Let's review the git policy for homeworks: Section 7.3 - Git policy for homeworks (TTIC/UDEM)
- Announcement regarding activity spreadsheet which is now embedded here: Section 8.1 - The Activity Tracker. Feel free to just look or send Kirsten your gmail on Slack and she will give you access to it.

Slides:

- Duckiebot Modeling (pptx) (pdf)

Book Material:

- Unit G-19 - Coordinate systems
- Unit G-20 - Reference frames
- Unit H-11 - Duckiebot modeling

## 12.10. Mon Oct 2

Class (10:30 - 11:30) in Z-210

Slides:

- Computer vision basics: (keynote) (pdf)

Book Material:

- Unit H-13 - Computer vision basics
- Unit H-14 - Camera geometry
- Unit H-15 - Camera calibration
- Unit H-16 - Image filtering

Lab (11:30 - 12:30) in AA 2333

## 12.11. Wed Oct 4

Class (11:30 - 12:30) in Z-305

Slides:

- Computer Vision - Lane and Line Detection (keynote) (pdf)

Book Material:

- Unit H-16 - Image filtering
- Unit H-18 - Line Detection

Lab (12:30 - 1:30) in AA 2333

> **Note:** Checkoff and Homework due at 11pm

## 12.12. Mon Oct 9

Holiday no class!

## 12.13. Wed Oct 11

Class 11:30 in Z-305

- Computer Vision - Feature descriptors (keynote) (pdf)

Book Material:

Lab 12:30 - 1:30 in AA2333

## 12.14. Friday Oct 13

New Checkoff Initiated: Unit M-19 - Checkoff: Robot Calibration Deadline is **Monday Oct. 23**. Deliverables are: - Screenshot of your robot passing the kinematic odometry test - PR to duckiefleet repo with your 3 robot calibrations (kinematics, camera intrinsics, camera extrinsics)

## 12.15. Monday Oct 16

Class 10:30-11:30 Z-205

- Reminder about checkoff.
- Intro to filtering (pptx) (pdf)

## 12.16. Wednesday Oct. 18

Class **12:30-1:30** Z-310

Guest Lecture from Prof. James Forbes from McGill on Extended Kalman filter

slides: (pdf)

## 12.17. Friday Oct. 20

Homework Unit M-23 - Homework: Augmented Reality announced. Deadline is **Friday Oct. 27** at 11pm

## 12.18. Monday Oct. 23

No class (Reading Week)

### 12.19. Wednesday Oct. 25

No class (Reading Week)

### 12.20. Monday Oct. 30

Class 10:30-11:30 Z-2015

Start of Introduction to SLAM

Slides: (pdf) (keynote)

### 12.21. Wednesday Nov. 1

Class 11:30-12:30

End of Introduction to SLAM

Slides: (pdf) (keynote)

### 12.22. Monday Nov. 6

Class 10:30-12:30

Project Pitches. Link to slides

### 12.23. Wednesday Nov. 8

Class 11:30-12:30

- Motion Planning. (pdf), (pptx)

Checkoff Navigation initiated. Deadline Nov 15 @ 11pm.

### 12.24. Thursday Nov. 9

- Filtering Homework initiated. Deadline Nov 17 @ 11pm.
- Project groups announced

### 12.25. Monday Nov. 13

Class 10:30-11:30

Control.

Module 1: (pptx)

Module 2: (pptx)

Module 3: (pptx)

### 12.26. Wednesday Nov. 15

Class 11:30-11:30

David Vazquez guest lecture

### 12.27. Oct 30 (Mon)

Assigned to: XXX

### 12.28. Nov 01 (Wed)

Assigned to: XXX

### 12.29. Nov 06 (Mon)

Assigned to: XXX

### 12.30. Nov 08 (Wed)

Assigned to: XXX

### 12.31. Nov 13 (Mon)

Assigned to: XXX

### 12.32. Nov 15 (Wed)

Assigned to: XXX

### 12.33. Nov 20 (Mon)

Assigned to: XXX

### 12.34. Nov 22 (Wed)

Assigned to: XXX

### 12.35. Nov 27 (Mon)

Assigned to: XXX

### 12.36. Nov 29 (Wed)

Assigned to: XXX

### 12.37. Dec 04 (Mon)

Assigned to: XXX

## 12.38. Dec 06 (Wed)

Assigned to: XXX

## 12.39. Dec 11 (Mon)

Assigned to: XXX

## 12.40. (Template for every lecture) Date: Topic

Assigned to: Name of TA

### 1) Preparation

Things that the students should do before class.

### 2) Material presented in class

Link to PDF and Keynote/Powerpoint materials.

### 3) Pointers to reading materials

Links to the units mentioned in the slides, and additional materials.

### 4) Questions and answers

Write here the FAQs that students have following the lecture or instructions.

## 12.41. (Template for every lecture) Date: Topic

Assigned to: Name of TA

### 1) Preparation

Things that the students should do before class.

### 2) Material presented in class

Link to PDF and Keynote/Powerpoint materials.

### 3) Pointers to reading materials

Links to the units mentioned in the slides, and additional materials.

### 4) Questions and answers

Write here the FAQs that students have following the lecture or instructions.

# UNIT M-13
# Chicago branch diary

Classes take place on Mondays and Wednesdays from 9am-11am in TTIC Room 530.

## 13.1. Checkoffs:

The following is a list of the current checkoffs, along with the date and time when they are due. Most submissions involve uploading videos to via Dropbox File Request using the link provided in the individual checkoff.

- **Friday October 6, 5pm CT**: Unit M-17 - Checkoff: Assembly and Configuration
- **Wednesday October 18, 11:59pm CT**: Unit M-18 - Checkoff: Take a Log
- **Sunday October 22, 11:59pm CT**: Unit M-19 - Checkoff: Robot Calibration
- **Wednesday November 15, 11:59pm CT**: Unit M-24 - Checkoff: Navigation

## 13.2. Problem Sets:

The following is a list of the current problem sets, along with the date and time when they are due.

➡ Section 7.3 - Git policy for homeworks (TTIC/UDEM)

for instructions on how the homework should be submitted.

> Note: Please keep track of how much time you spend on each problem set. We will ask you for this estimate along with other feedback at the end of each problem set.

- **Friday October 13, 11:59pm CT**: Unit M-21 - Homework: Data Processing (TTIC)
- **Friday October 27, 11:59pm CT**: Unit M-23 - Homework: Augmented Reality
- **Friday November 17, 11:59pm CT**: Unit M-25 - Homework: Lane Filtering

## 13.3. Monday September 25: Introduction to Duckietown

### 1) Lecture Content

- Duckietown Course Intro (Keynote, PDF)

### 2) Reading Material

- Part A - The Duckietown project
- Unit H-2 - Autonomy overview

### 3) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

## 13.4. Tuesday September 26: Onboarding

Tonight, we sent out the onboarding instructions.

➡ Section 2.1 - Onboarding Procedure

**Please complete the onboarding questionnaire by Thursday, September 27, 5:00pm CT**

## 13.5. Wednesday, September 27: Duckiebox Ceremony

Welcome to Duckietown! This lecture constitutes the *Duckiebox ceremony* during which we will distribute your Duckieboxes and ask you to name your yet-to-be built Duckiebots! We will also discuss logistics related to the course, but we admit that that isn't as exciting.

### 1) Lecture Content

- The Duckiebook
- Slack (Unit M-10 - Slack Channels)
- Git repos (Unit M-7 - Git usage guide for Fall 2017)
- How to get and give help (Unit M-9 - Getting and giving help)
- Unit M-17 - Checkoff: Assembly and Configuration initiated
- Getting to know one another
- Naming your robots
- Distribute the Duckieboxes!

As you name your Duckiebots, please consider the following constraints:

- The name must work as a hostname. It needs to start with a letter, contains only letters and numbers, and no spaces or punctuation.
- It should be short, easy to type. (You'll type it a lot.)
- It cannot be your own name.
- It cannot be a generic name like "robot", "duckiebot", "car". It cannot contain brand names.

### 2) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

## 13.6. Monday, October 2: Modern Robotic Systems

### 1) Lecture Content

- Logistics (Keynote, PDF)
- Autonomous Vehicles (Keynote, PDF)
- Modern Robotic Systems (Keynote, PDF)
- System Architecture Basics (Keynote, PDF)

### 2) Reading Material

- Unit H-1 - Autonomous Vehicles
- Unit H-2 - Autonomy overview
- Unit H-3 - Modern Robotic Systems

- Unit H-4 - System architectures basics

#### 3) Assignments

- Unit M-17 - Checkoff: Assembly and Configuration is due by Friday 5pm CT. Note: The page provides the URL where you should upload your video(s).

#### 4) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

### 13.7. Wednesday, October 4: Modern Robotic Systems (Continued)

**Note**: Today's lecture will take place in Room 501 due to the TTIC Board Meeting

#### 1) Lecture Content

- Logistics (Keynote, PDF)
- Representations (Keynote, PDF)
- Software Architectures (Keynote, PDF)
- Networking (Keynote, PDF)

#### 2) Reading Material

- Unit H-6 - Representations
- Unit J-18 - Networking tools
- Unit H-4 - System architectures basics

#### 3) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

### 13.8. Monday, October 9: Modeling

#### 1) Preparation

**Important**: Before starting these tutorials, make sure that you completed the following:

➡ Unit C-8 - Installing Ubuntu on laptops

Before coming to class, please read through the following tutorials:

- https://tinyurl.com/ROS101-Intro
- http://wiki.ros.org/ROS/Tutorials#Beginner_Level Complete all the tutorials in Section 1.1 Beginner Level. There is no need to install ROS as the tutorial instructs, because you already installed as part of the setup process. Sections 11 and 14 of this tutorial will ask you to implement simple ROS nodes using C++. You can skip them since Sections 12 and 15 are basically the same but using Python.

#### 2) Lecture Content

- Modeling (Powerpoint, PDF)

#### 3) Reading Material

- Unit H-11 - Duckiebot modeling

#### 4) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

### 13.9. Wednesday, October 11: Introduction to Computer Vision

#### 1) Lecture Content

- Introduction to Computer Vision (Keynote, PDF)
- Camera Models (Keynote, PDF)

#### 2) Reading Material

- Unit H-13 - Computer vision basics
- Unit H-14 - Camera geometry
- Richard Szeliski, *Computer Vision: Algorithms and Applications*, Chapters 1 and 2 (available online)
- David A. Forsyth and Jean Ponce, *Computer Vision: A Modern Approach*, Chapters 1 and 2

#### 3) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

### 13.10. Monday, October 16: Camera Calibration and Image Filtering

> **Note:** The second checkoff is due by 11:59pm CT Wednesday.

#### 1) Lecture Content

- Logistics (Keynote, PDF)
- Calibration (Keynote, PDF)
- Image Filtering (Keynote, PDF)

#### 2) Reading Material

- Unit H-13 - Computer vision basics
- Unit H-15 - Camera calibration
- Unit H-16 - Image filtering
- Richard Szeliski, *Computer Vision: Algorithms and Applications*, Chapters 3 and 6 (available online)
- David A. Forsyth and Jean Ponce, *Computer Vision: A Modern Approach*, Chapters 5.3 and 7

#### 3) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

## 13.11. Wednesday, October 18: Edge Detection and Lane Detection

> **Note:** The third checkoff is due by 11:59pm CT Sunday. Note that camera calibration is necessary for the next problem set, which will be posted soon.

### 1) Lecture Content

- Logistics (Keynote, PDF)
- Image Filtering (Review) (Keynote, PDF)
- Image Gradients (Keynote, PDF)
- Edge Detection (Keynote, PDF)
- Line Detection (Keynote, PDF)
- Lane Detection (Keynote, PDF)

### 2) Reading Material

- Unit H-18 - Line Detection
- Richard Szeliski, *Computer Vision: Algorithms and Applications*, Chapters 3 and 4 (available online)
- David A. Forsyth and Jean Ponce, *Computer Vision: A Modern Approach*, Chapters 7 and 8

### 3) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

## 13.12. Monday, October 23: Feature Detection and Place Recognition

### 1) Lecture Content

- Logistics (Keynote, PDF)
- Robust Fitting (Keynote, PDF)
- Feature Detection (Keynote, PDF)
- Place Recognition (Keynote, PDF)

### 2) Reading Material

- Richard Szeliski, *Computer Vision: Algorithms and Applications*, Chapters 4 and 6.1.3 (available online)

### 3) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

## 13.13. Wednesday, October 25: Filtering I

### 1) Lecture Content

- Place Recognition (continued) (Keynote, PDF)
- Introduction to Filtering (Powerpoint, PDF)

### 2) Reading Material

- Sebastian Thrun, Wolfram Burgard, and Dieter Fox, *Probabilistic Robotics*, Chapters 1 and 2

#### 3) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

### 13.14. Monday, October 30: Filtering II

#### 1) Lecture Content

- Nonparametric Filtering (Powerpoint, PDF)

#### 2) Reading Material

- Sebastian Thrun, Wolfram Burgard, and Dieter Fox, *Probabilistic Robotics*, Chapter 4

#### 3) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

### 13.15. Wednesday, November 1: Introduction to SLAM

#### 1) Lecture Content

- SLAM Intro (Keynote, PDF)

#### 2) Reading Material

- Sebastian Thrun, Wolfram Burgard, and Dieter Fox, *Probabilistic Robotics*, Chapters 9, 10, and 13

#### 3) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

### 13.16. Monday, November 6: Introduction to Planning

#### 1) Lecture Content

- Planning Intro (Keynote, PDF)
- Project Pitches (Google Slides)

#### 2) Reading Material

- Steven M. LaValle, *Planning Algorithms*, Chapters 3, 4, and 6. Available online

#### 3) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

## 13.17. Wednesday, November 8: Introduction to Planning (Continued)

### 1) Lecture Content

- Planning Intro (Keynote, PDF)

### 2) Reading Material

- Steven M. LaValle, *Planning Algorithms*, Chapter 5. Available online

### 3) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

## 13.18. Monday, November 13: Introduction to Control

### 1) Lecture Content

- Control Intro (Powerpoint, PDF)

### 2) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

## 13.19. Wednesday, November 15: Introduction to Control (Continued)

### 1) Lecture Content

- Controls (Powerpoint, PDF)
- Controls for Duckietown (Powerpoint, PDF)

### 2) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

## 13.20. Monday, November 20: Testing for Autonomous Vehicles

### 1) Lecture Content

- Testing for Autonomous Vehicles (Keynote, PDF)

### 2) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

UNIT M-14

# NCTU branch diary

Classes take place on Thursday from 1:20pm-4:20pm in NCTU Engineering Building 5 Room 635.

## 14.1. Checkoffs:

The following is a material of the current preview lecture, along with the date and time when they are due. Please make sure that you preview the materials every week before the class for insuring a good learning quality.

- **Thursday October 19, 1:20pm**: Week5 Material
- **Thursday October 26, 1:20pm**: Week6 Material

## 14.2. Course Material:

The following is a list of the course material. Please preview the course every week and better if you try the lab yourself.

➡ Course Material

- **Thursday October 12, 1:20pm** : Week5 Material

## 14.3. Thursday September 14: Introduction to Duckietown and Creative Software Project

### 1) Lecture Content

- Duckietown Course Intro (Week1 Material)

## 14.4. Thursday September 21: Project Ideas

How do you choose a good project idea? How about "writing" a good project idea?

➡ Week2 Material

### 1) Lecture Content

➡ Week2 Lecture

### 2) Weekly Lab

This is the first lab of the semester. We are going to teach you "git" which is essential when becoming a professional programmer and cooperating with professional team.

➡ Week2 Lab

## 14.5. Thursday September 28: Robotics System

What is a robotics system? We'll introduce the concept of robotics system and some well known software architecture and middleware. Also, robotic operation system will be introduced in this class.

See" Week3 Material

### 1) Lecture Content

➡ Week3 Lecture

### 2) Weekly Lab

This week we'll continue on the topic of git. Also, a new exciting chapter has been opened. We are going to prepare our own Duckiebot! Come and join us!.

➡ Week3 Lab

## 14.6. Thursday October 5: OpenCV, Python and Jupyter Notebook

Today we're going deeply into Duckiebot's "mind". What is the algorithmn behind lane follwing? What is the secret that the duckies are so smart to drive? Here comes the answer.

➡ Week4 Material

### 1) Lecture Content

➡ Week4 Lecture

### 2) Weekly Lab

Jupyter notebook is a very useful and conveniet tool while dealing with python language. We will teach you how to use it. A part of lane following algorithmn will be taught this week which is about line detector.

➡ Week4 Lab

## 14.7. Thursday October 19: Camera and Wheel Calibration

This week we are going to do the camera and wheel calibration. We will teach the student the theory of camera calibration, including extrinsic and intrinsic.

➡ Week6 Material

### 1) Lecture Content

➡ Week6 Lecture

#### 2) Weekly Lab

We will let them control the duckiebots by joystick to finish the wheel calibration. Also, we will give them chessboard for camera calibration.

➡ Week6 Lab

### 14.8. (Template for every lecture) Date: Topic

What is this lecture all about?

#### 1) Preparation

Things that the students should do before class.

#### 2) Lecture Content

Link to PDF and Keynote/Powerpoint materials.

#### 3) Feedback Form

Please help us making the experience better by providing feedback (can be anonymous)

#### 4) Reading Material

Links to the units mentioned in the slides, and additional materials.

#### 5) Questions and Answers

FAQs that students have following the lecture or instructions.

# Unit M-15
# Slack Channels

This page describes all of the helpful Slack channels and their purposes so that you can figure out where to get help.

## 15.1. Channels

Table 15.1. Duckietown Slack Channels

| Channel | Purpose |
|---|---|
| `help-accounts` | Info about necessary accounts, such as Slack, Github, etc |
| `help-assembly` | Help putting your robot together |
| `help-camera-calib` | Help doing the intrinsic and extrinsic calibration of your camera |
| `help-duckuments` | Help compiling the online documentation |
| `help-git` | Help with git |
| `help-infrastructure` | Help with software infrastructure, such as Makefiles, unit tests, continuous integration, etc. |
| `help-laptops` | Help getting your laptop setup with Ubuntu 16.04 |
| `help-parts` | Help getting the parts for the robot or replacement parts if you broke something |
| `help-robot-setup` | Help getting the robot setup to do basic things like be driven with a joystick |
| `help-ros` | Help with the Robot Operating System (ROS) |
| `help-wheel-calib` | Help doing your odometry calibration |

+ comment

Note that you can link directly to the channel. (See list in the org sheet.) -AC

UNIT M-16

# Guide for TAs

## 16.1. Dramatis personae

These are the TAs.

At ETHZ:

- Shiying Li (shili@student.ethz.ch)
- Ercan Selçuk (ercans@student.ethz.ch)
- Miguel de la Iglesia Valls (dmiguel@student.ethz.ch)
- Khurana Harshit (hkhurana@student.ethz.ch)
- Lapandic Dzenan (ldzenan@student.ethz.ch)
- Marco Erni (merni@ethz.ch)

At TTIC:

- Andrea F. Daniele (afdaniele@ttic.edu)
- Falcon Dai (dai@ttic.edu)
- Jon Michaux (jmichaux@ttic.edu)

At Montreal:

- Florian Golemo (fgolemo@gmail.com)

## 16.2. First steps

Here are the first steps for the TAs.

Note that many of these are not sequential and can be done in parallel.

### 1) Learn about Duckietown

Read about Duckietown's history; watch the Duckumentary.

➡ Part A - The Duckietown project

### 2) Online accounts

You have to set up:

- A personal Github account
- A Twist account
- A Slack account
- A Google Docs account (Gmail address)

Send an email to Kirsten Bowser (akbowser@gmail.com), with your GMail address and your Github account. She will give you further instructions.

**Point of contact:** Kirsten Bowser

### 3) Install Ubuntu

Install Ubuntu 16.04 on your laptop, and then install ROS, Atom, Liclipse, etc.

➡ Unit C-8 - Installing Ubuntu on laptops

### 4) Duckuments

Install the Duckuments system, so you can edit these instructions.

➡ Part B - Duckumentation documentation.

**Point of contact:** Andrea

### 5) Learn about Git and Github

Start learning about Git and Github. You don't have to read the entirety of the following references now, but keep them "on your desk" for later reference.

➡ Good book

➡ Git Flow

**Point of contact:** Liam?

### 6) Continuous integration

Understand the continuous integration system.

➡ Documentation on continuous integration.

**Point of contact:** Andrea

### 7) Duckiebot building

Build your Duckiebot according to the instructions.

➡ Part C - Operation manual - Duckiebot

**Point of contact:** Shiying (ETH)

**Point of contact:** ??? (UdeM)

**Point of contact:** ??? (TTIC)

As you read the instructions, keep open the Duckuments source, and note any discrepancies. You must note any unexpected thing that is not predicted from the instruction. If you don't understand anything, please note it.

The idea is that dozens of other people will have to do the same after you, so improving the documentation is the best use of your time, and it is much more efficient than answering the same question dozens of times.

### 8) Other documentation outside of the Duckuments

We have the following four documents outside of the duckuments:

1. Organization chart: This is where we assign areas of responsibility.
2. Lecture schedule
3. Checkoff spreadsheet
4. The big TODO list: Where we keep track of things to do.

UNIT M-17

# Checkoff: Assembly and Configuration

The first job is to get your Duckiebot put together and up and running.

## 17.1. Pick up your Duckiebox

**Slack channel:** #help-parts

There is a checklist inside. You should go through the box and ensure that the parts that are supposed to be in it actually are inside.

If you are missing something, contact your local responsible and ask for help on Slack in the appropriate channel.

These sections describe the parts that are in your box.

- ➡ Unit C-2 - Acquiring the parts (DB17-jwd)
- ➡ Unit E-1 - Acquiring the parts (DB17-1c)

?? You don't need to buy anything - you have all the parts that you need.

## 17.2. Soldering your boards

Depending on how kind your instructors/TAs are, you may have to solder your boards.

- ➡ Unit C-3 - Soldering boards (DB17)
- ➡ Unit E-2 - Soldering boards (DB17-1)

?? You don't need to solder anything.

## 17.3. Assemble your Robot

**Slack channel:** #help-assembly

You are ready to put things together now.

- ➡ Unit C-5 - Assembling the Duckiebot (DB17-jwd)
- ➡ Unit E-4 - Bumper Assembly
- ➡ Unit E-3 - Assembling the Duckiebot (DB17-1c)

## 17.4. Optional: Reproduce the SD Card Image

If you are very inexperienced with Linux/Unix/networking etc, then you may find it a valuable experience to reproduce the SD card image to "see how the sausage is made".

- ➡ Unit C-7 - Reproducing the image

??

You probably don't want to see how the sausage is made.

## 17.5. Setup your laptop

**Slack channel:** #help-laptops

The only officially supported OS is Ubuntu 16.04. If you are not running this OS it is recommended that you make a small partition on your hard drive and install the OS.

Related parts of the book are:

- Section 6.9 - How to make a partition if you want to make a partition
- Unit C-8 - Installing Ubuntu on laptops

## 17.6. Make your robot move

**Slack channel:** #help-robot-setup

Now you need to clone the software repo and run things to make your robot move.

First initialize the robot:

- Unit C-9 - Duckiebot Initialization

Then get it to move!

- Unit C-11 - Software setup and RC remote control

## 17.7. Upload your video

You should record a video demonstrating that your Duckiebot is up and running. Brownie points for creative videos. Please upload your videos via the following URL:

?? Chicago: upload your video

?? Zurich: upload your video

# Unit M-18
# Checkoff: Take a Log

Knowledge and activity graph

**Requires:** Unit M-17 - Checkoff: Assembly and Configuration
**Results:** A verified log in rosbag format uploaded to Dropbox.

**Slack channel:** #help-logging

?? Montreal deadline: Oct 4, 11pm

?? Zurich deadline: Oct 20, 17:00

## 18.1. Mount your USB drive

We will log to the USB drive that you were given.

➡ Unit J-15 - Mounting USB drives

## 18.2. Take a Log

Take a 5 min log as you drive in Duckietown.

?? For Montreal this is rm. 2333.

?? For Zurich this is ML J44. Ask the TA when it is available.

?? For Chicago, we are still building the town, so feel free to do this at home or in the lab.

➡ Unit C-19 - Taking and verifying a log for detailed instructions.

## 18.3. Verify your log

➡ Section 19.5 - Verify a log for detailed instructions.

## 18.4. Upload the log

?? Upload the log here

?? Upload the log here

?? Upload the log here

# Unit M-19
# Checkoff: Robot Calibration

Knowledge and activity graph

**Requires:** Unit M-17 - Checkoff: Assembly and Configuration

**Requires:** That you have correctly cloned and followed the git procedure outline in Unit M-7 - Git usage guide for Fall 2017

**Requires:** That you have correctly setup your environment variables according to Section 4.1 - Environment variables (updated Sept 12)

**Results:** You robot calibrations (wheels and camera (x2)) are merged to git through a PR

Slack channels: #help-wheel-calib, #help-camera-calib

## 19.1. Pull and rebuild your Software repo on robot and laptop

Some of the services have changed and this requires a rebuild.

On both laptop and robot do:

```
$ cd Duckietown root
$ source environment.sh
$ make build-catkin-clean
$ make build-catkin-parallel
```

## 19.2. Make a branch in the duckiefleet repo

?? Remember that the git policy has changed a bit. You are probably best to re-clone the duckiefleet repo. For details see Unit M-7 - Git usage guide for Fall 2017 and particularly the section `For U de M students who have already submitted homework to the previus duckiefleet-2017 repo`

Don't forget that master is now protected in duckiefleet. So make a new branch right away and call it *GIT_USERNAME*`-devel`

## 19.3. Kinematic calibration

Follow the procedure in Unit C-16 - Wheel calibration. Once you have successfully passed the automated test, take a screen shot and post it to the slack channel #checkoffs and we will all congratulate you.

## 19.4. Camera calibration

Follow the procedure in Unit C-17 - Camera calibration to do you intrinsic **and** extrinsic calibrations.

## 19.5. Visually verify the calibration is good in Duckietown

Take your robot to Duckietown. Put it in a lane.

On your robot execute

```
$ make demo-lane-following
```

On your laptop do (after setting ros master to your robot):

```
$ rqt_image_view
```

on your joystick you need to hit the top-right button (TODO: add picture). On the command line you should see the output `state_verbose = True`

in the drop down menu select *robot name*`/line_detector_node/image_with_lines`

on the display you should see all the color-coded line detections

now open the Rviz visualizer on your laptop (after setting ros master to your robot):

```
$ rviz
```

- click the `Add` button in the bottom left.
- then click the `By Topic` tab
- then click the triangle next to `/segment_list_markers` underneath `/duckiebot_visualizer`
- then double click on `MarkerArray`

On the display you should see the ground projected lines. Do they make sense? If not your calibration is wrong.

TODO: add a picture of what they should look like

## 19.6. Submit a PR

Don't forget at the end to submit a PR back to duckiefleet repo

# Unit M-20
# Homework: Data Processing (UdeM)

Knowledge and activity graph

**Requires:** Unit M-18 - Checkoff: Take a Log
**Requires:** Unit J-30 - ROS installation and reference
**Results:** Ability to perform basic operations on images
**Results:** Build your first ROS package and node
**Results:** Ability to process imagery live

**Slack channel:** #ex-data_processing
Montreal deadline: Oct 4, 11:00pm

## 20.1. Follow the git policy for homeworks

➡ Section 7.3 - Git policy for homeworks (TTIC/UDEM)

for instructions on how the homework should be submitted.

## 20.2. Exercise: Basic image operations

Complete Unit I-2 - Exercise: Basic image operations, adult version

## 20.3. Exercise: Log decimation

Complete Unit I-4 - Exercise: Bag in, bag out

## 20.4. Exercise: Instagram filters

Complete Unit I-6 - Exercise: Instagram filters

## 20.5. Exercise: Live Instagram

Complete Unit I-8 - Exercise: Live Instagram

Call your package `dt-instagram-live-`*robot name* and call your node `dt-instagram-live-`*robot name*

When you are done, take a 5min log (See Unit C-19 - Taking and verifying a log)in Duckietown (2333 in Montreal) and upload here

UNIT M-21

# Homework: Data Processing (TTIC)

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Unit J-30 - ROS installation and reference
**Results:** Ability to perform basic operations on images
**Results:** Build your first ROS package and node
**Results:** Ability to process imagery live

**Slack channel:** #help-data-processing
TTIC deadline: Friday, October 13 11:59pm CT

## 21.1. Follow the git policy for homeworks

➡ Section 7.3 - Git policy for homeworks (TTIC/UDEM)

for instructions on how the homework should be submitted.

## 21.2. Exercise: Basic image operations

Complete Unit I-2 - Exercise: Basic image operations, adult version

## 21.3. Exercise: Log analysis

Complete Unit I-3 - Exercise: Simple data analysis from a bag

## 21.4. Exercise: Log decimation

Complete Unit I-4 - Exercise: Bag in, bag out

## 21.5. Exercise: Video thumbnails

Complete Unit I-5 - Exercise: Bag thumbnails

## 21.6. Exercise: Instagram filters

Complete Unit I-6 - Exercise: Instagram filters

## 21.7. Exercise: Log Instagram

Complete Unit I-7 - Exercise: Bag instagram

## 21.8. Exercise: Live Instagram

Complete Unit I-8 - Exercise: Live Instagram

Call your package `dt-instagram-live_`*robot name* and call your node `dt-instagram-live_`*robot name*.

## 21.9. Exercise: Feedback

Complete the exercise feedback form. You will receive points towards this exercise if you complete the form.

UNIT M-22

# Exercises: Data Processing (Zurich)

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Unit J-30 - ROS installation and reference
**Results:** Ability to perform basic operations on images
**Results:** Build your first ROS package and node
**Results:** Ability to process imagery live

Slack channel to get help: `#ex-data-processing`

## 22.1. Git setup

First, make sure you are in the Github Zurich team.

If your name is not here, contact Kirsten, and stop. You will not be able to do the next step.

Please clone this repository:

```
git@github.com:AndreaCensi/exercises-fall2017.git
```

using

```
$ git clone git@github.com:AndreaCensi/exercises-fall2017.git
```

This repository is writable by all Zurich people, but not readable by Chicago and Montreal. Because they grade the homework, we need to keep it secret.

We invite everybody to just push their exercises to this repository. (This is also compulsory to get help from TAs, so that the TAs can give comments that are useful for everybody.)

## 22.2. The exercises

We have created a series of exercises that are supposed to help somebody who doesn't know how to program in Python/Linux to get to a decent level.

We suggest the following:

1. First, read through the exercises and note the skills that are learned for each one.
2. Look at the last two: "Log Instagram" and "Live Instagram". Do you think you can do them? If so, just do those two and feel free to skip the rest.
3. Otherwise, you have a lot to catch up. No problem. Take your time. Start with the basic exercise. TAs are here to help.

## 22.3. Exercise: Basic image operations

➡ Unit I-2 - Exercise: Basic image operations, adult version

## 1) Exercise: Log analysis

➡ Unit I-3 - Exercise: Simple data analysis from a bag

## 2) Exercise: Log decimation

➡ Unit I-4 - Exercise: Bag in, bag out

## 3) Exercise: Video thumbnails

➡ Unit I-5 - Exercise: Bag thumbnails

## 4) Exercise: Instagram filters

➡ Unit I-6 - Exercise: Instagram filters

## 5) Exercise: Log Instagram [recommended for everybody]

➡ Unit I-7 - Exercise: Bag instagram

## 6) Exercise: Live Instagram [recommended for everybody]

➡ Unit I-8 - Exercise: Live Instagram

Call your package `dt-instagram-live-`*robot name* and call your node `dt-instagram-live-`*robot name*.

## 7) Exercise: Feedback form

**TODO:** provide link to exercise feedback form.

UNIT M-23

# Homework: Augmented Reality

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Unit J-30 - ROS installation and reference
**Requires:** Unit C-17 - Camera calibration
**Results:** Ability to project fake things from an image back into the world

**Slack channel:** #ex-augmented_reality

?? Montreal deadline: Oct 27, 11:00pm

?? Chicago deadline: Oct 27, 11:59pm

?? Zurich deadline: Oct ???, 11:59pm

## 23.1. Follow the git policy for homeworks

Please follow the instructions on how the homework should be submitted.

➡ Section 7.3 - Git policy for homeworks (TTIC/UDEM)

## 23.2. Exercise: Augmented Reality

Complete Unit I-9 - Exercise: Augmented Reality.

??

```
Please hold on for Github instructions.
```

Note that you should do a pull in `Software` to get all the goodies and utils described in the exercise, and `exercises-fall2017` repos (in `exercises-fall2017` repo this means pulling from the `duckietown` remote:

```
$ git pull upstream master
```

if you have followed the instructions properly.

In the `exercises-fall2017` repository, you will find a template that you can use to make your own package. Basically everywhere you see `littleredcorvette` you would need to replace it with your *robot_name*.

## 23.3. Submission

?? Please upload the images requested in the homework to your git repository. When complete, please tag a release from your repo.

?? Please upload the images requested in the homework to your git repository. When complete, please tag a release from your repo.

### 23.4. Bonus: Defining `intersection_4way.yaml`

The first student to do it (from any institution) gets notoriety and a bonus.

UNIT M-24

# Checkoff: Navigation

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Unit M-19 - Checkoff: Robot Calibration
**Requires:** Unit M-18 - Checkoff: Take a Log
**Results:** 2 logs of your robot autonomously navigating Duckietown

?? Montreal Deadline: Nov 15, 11pm

?? Chicago Deadline: Nov 15, 11pm

**Slack channel:** #help-navigation

## 24.1. Pull from master

As always - it's a good idea to pull from `master` often.

## 24.2. Lane Following

Place your robot on the Duckietown map somewhere on the "outer loop" (right hand lane so that it will follow the exterior of the map).

Launch the robot with the command from *DUCKIETOWN_ROOT*:

```
$ make demo-lane-following
```

Open a terminal on your laptop and set the ros master to your robot.

Toggle the `VERBOSE` flag by writing:

```
$ rosparam set /robot_name/line_detector_node/verbose true
```

Then open `rqt_image_view`. Look at the `.../image_with_lines` image output. Apply the **anti-instagram callibration** by pushing the `Y` button on the joystick (TODO: is it the same for the new joysticks?). You should see your image get corrected and the line detections become more correct. If nothing happens and your robot output complains of bad health, move the robot a little bit and try again.

You may also be interested to look at the `../belief_img` to see the output of the histogram filter. It should be quite stable if your robot is not moving. You can move the robot around to see how the posterior is updating.

If everything is looking good then push the `START` button on the joystick and your robot should start to drive.

The robot operation should look like this

Follow the instructions here to take a **minimal** log of at least 5 mins of uninterrupted

robot autonomous function.

?? Upload here

?? Upload here

1) Bonus

The student who uploads the longest log of uninterupted robot autonomous lane following from any institution will get a great bonus.

## 24.3. Indefinite Navigation

Follow the exact same procedure above but instead of running the lane following demo run the "indefinite navigation" demo:

```
$ make indefinite-navigation
```

Your robot will now stop at the stop lines and then make a random turn through the intersection. If it is crashing a lot you may need to turn the trajectories it takes through the intersection. To do so you may need to edit the file here:

```
turn_left: #time, velocity, angular vel
  - [0.8, 0.43, 0]
  - [1.8, 0.43, 2.896]
  - [0.8, 0.43, 0.0]
turn_right:
  - [0.6, 0.43, 0]
  - [1.2, 0.3, -4.506]
  - [1.0, 0.43, 0.0]
turn_forward:
  - [0.8, 0.43, 0.4]
  - [1.0, 0.43, 0.0]
  - [1.0, 0.43, 0.0]
```

to make it more reliably traverse the intersections.

Follow the instructions here to take a **minimal**. You may use the `BACK` button to stop it from crashing and then return it to autonomous mode with the `START` button.

?? Upload here

?? Upload here

1) Bonus

The student who uploads the longest log of uninterupted robot autonomous indefinite navigation from any institution will get a great bonus.

UNIT M-25

# Homework: Lane Filtering

?? Montreal deadline: Nov 17, 11:00pm

?? Chicago deadline: Nov 17, 11:59pm

**Slack channel:** #ex-filtering

## 25.1. Follow the git policy for homeworks

Please follow the instructions on how the homework should be submitted.

➡ Section 7.3 - Git policy for homeworks (TTIC/UDEM)

## 25.2. Pick your Poison

This homework is about filtering. Either replace the existing histogram lane filter with either an Extended Kalman Filter **or** a Particle Filter. If you do both you will get a bonus.

## 25.3. Setup instructions

Pull from `master` in the `Software` repo

Pull from the Duckietown (`upstream`) remote in your `exercises-fall2017` repo.

We are providing a script to change all the instances of the default robot (in this case `shamrock`) with *YOUR_ROBOT_NAME* to save you time. To run navigate to the `homeworks/03_filtering` directory and run:

```
$ ./change_robot_name_everywhere.sh YOUR_ROBOT_NAME
```

(you're welcome...)

In the

`homeworks/03_filtering/YOUR_ROBOT_NAME/dt_filtering_YOUR_ROBOT_NAME`

folder, the files you need to worry about are the following:

1) `default.yaml`: this contains the parameters that will be loaded. Here's what it currently looks like:

```
# default parameters for lane_filter/lane_filter_node
# change to your robot name below
filter:
- dt_filtering_shamrock.LaneFilterPF
  - configuration:
      example1: 0.2
      # fill in params here

#uncomment below and comment above if you are doing EKF
#  - dt_filtering_shamrock.LaneFilterEKF
#  - configuration:
      example2: 0.3
      #fill in other params here
```

This parameter file tells your node to automatically load the right filter. If you are working on particle filter you can leave it the way it is and just add your parameters that you need under `configuration`. If you are working on EKF, comment or delete the lines for the PF and uncomment the lines for the EKF and then add your params as needed.

The other file you need to concern yourself with is in

```
include/dt_filtering_YOUR_ROBOT_NAME
```

You will need to fill in the functions that are setup for you.

## 25.4. Submission

As normal, tag the TAs and instructors in a release from your repo when you are ready for your work to be evaluated.

UNIT M-26

# Guide for mentors

**Assigned to:** Liam?

# Unit M-27
# Project proposals

TODO: to write

UNIT M-28

# System Architecture

## 28.1. Preliminaries

**Name of Project:** System Architecture

**Slack channel:** `#devel-heroes`

**Software development branch:** `devel-sonja` (Software repo) or `sonja-branch` (Duckuments repo)

### 1) Missions

The system architect project can be split into two missions:

1. Ensure that the development of the system goes smoothly (wooden spoon)
2. Develop a framework/tool to formally describe (and later optimize) the system (bronze, silver and gold)

## 28.2. Mission 1

Ensure that the development and integration of the projects into the system goes smoothly and that the resulting system makes sense, and is useful for future duckierations (duckie + generations).

### 1) Problem Statement

Ensure that all teams know what their goal is and how it fits into the bigger picture

### 2) Relevant Resources

- The functional diagram of the system
- Duckuments
- Other teams' preliminary project reports

### 3) Deliverables (Goals)

The deliverables for Mission 1 will include the following:

- Functional diagram of the system
- Documentation of system architecture

Mission 1 is the "wooden spoon" level of the project.

### 4) Proposed Approach

- Become one with the goals of Duckietown In order to make Duckietown a better place, one has to keep in mind what "better" means in Duckie terms.
- Be familiar with the current system architecture and track changes This can include having to update the functional diagram, for instance.
- Keep in close contact with teams This will be done by attending the meetings of some of the other teams (especially early meetings). Some teams' meetings have been prioritized since many parts of the system are dependant on their work,

namely:
  - Anti-instagram
  - Controllers
  - Navigators
  - Explicit coordination All teams will designate a contact person who can contact me whenever they change their project boundaries or have doubts/ need advice on their project's boundaries/negotiating with other
- Offer nudges in a different direction if needed
- Acting as middleman/helper to facilitate negotiation of contracts between groups
- Monitor status of projects to find possible problems

#### 5) Logging and Testing Procedure

...

#### 6) Current status

Familiarisation with the current system status is under way.

Functional diagram has been updated to include multi-robot SLAM as alternative to single-robot SLAM to creating map.

#### 7) Tasks

- Familiarisation with existing system architecture
- Going to group meetings
- Identifying potential problems

#### 8) Timeline

...

#### 9) Meetings notes

...

## 28.3. Mission 2

Where there is a system, there is a want (nay, need) for optimisation. Describing a system's performance and resource requirements in a quantifiable way is a critical part of being able to benchmark modules and optimise the system.

Mission 2 is to formalise the description of the system characteristics, so that eventually the system performance can be optimised for some given resources.

#### 1) Problem Statement

Find a way to describe all the module requirements, specifications, etc in a formal, quantifiable language.

Find a way to calculate the requirements and specifications of a whole system or subsystem, based on the requirements and specifications of the individual modules of the system.

Find a way to calculate the optimal system configuration, based on the desired re-

quirements and specifications of the system.

### 2) Relevant Resources

- How the current system's characteristics are defined
- Which values/parameters are needed
- Possibly research on system description?
- Possibly graph theory?

### 3) Deliverables (Goals)

The different levels of Mission 2 are defined as follows:

- Bronze standard:
  - Formal, qualitative language to describe constraints/requirements between modules
- Silver standard:
  - each module has table of performance. Qualitative. Can compare and give yes/no queries. With given configuration x, is the cost/requirements possible with available resources? f(x) smaller equal to Rmax?
- Gold standard:
  - optimization is possible to find best implementation, given available resources. Given f(x) and Rmax, find optimal configuration x

The deliverables will then include:

- Documentation on the result of the project
- A description of the current system's characteristics (bronze)
- A program/tool that can give a qualitative answer (yes/no) to the question: Are these resources sufficient for this system configuration? (silver)
- A program/tool that will give an optimised system configuration, based on the given available resources (gold)

### 4) Proposed Approach

- Research on the topic of formal description of a system
- Find/develop a suitable language to describe module characteristics
- Require groups to compile a description of their respective modules' characteristics
- Find/develop functions to do mathematics on the language description of modules

### 5) Logging and Testing Procedure

...

### 6) Current status

Research is being done to identify some research areas that may be relevant and tools that may be helpful, in order to decide on an approach.

### 7) Tasks

- Research into existing methods of system description
- Graph based databases?

- Perhaps graph theory can be useful later if the (suspiciously graph-looking) system can be described suitably.

## 8) Timeline

...

## 9) Meetings notes

...

UNIT M-29

# Template of a project

Make a copy of this document before editing.

## 29.1. Preliminaries

Name of Project:

Team:

- Person 1 (UdeM) (Mentor)
- Person 2 (TTIC) (TA)
- ...

**Slack channel:** #XXX

Software development branch: XXX-devel

## 29.2. Problem Statement

Summarize the mission for the team - What is the need that is being addressed? Do not focus on technical specifics yet.

## 29.3. Relevant Resources

List papers, open source code, pages in the Duckiebook, lecture slides, etc, that could be relevant in your quest.

## 29.4. Deliverables (Goals)

Anything that is going to be an output. These should be quantified in terms of functionalities and performance metrics where appropriate.

Example 1: A Duckiebot detection system (functionality) with minimum precision of 0.8, a minimum recall of 0.5, a maximum latency of 50ms with maximum CPU consumption of 80% of one core (performance).

Example 2: At least 20 hours of logs Duckiebots annotated

- Bronze standard:
- Functionality:
- Performance:
- Silver standard:
- Functionality:
- Performance:
- Gold standard:
- Functionality:
- Performance:

Part of the deliverables should be:

1. A new or improved functionality (demonstrated live and with a video) with well-

documented code,
2. A approx 15-20min presentation about the functionality (with a slide deck + 1 slide overview poster to be shown at public demo),
3. A technical description of the underlying method in the form of a page in the Duckiebook,
4. Instructions for reproducing the functionality in the form of a page in the duckiebook.

## 29.5. Proposed Approach

After analysis of the resources and precise understanding of the problem you trying to solve, make a plan for how you will solve the problem. It is possible that at the start you could explore several seemingly promising avenues. However, you should converge on Bronze standard before moving to Silver standard etc.

- Bronze standard:
- Silver standard:
- Gold standard:

## 29.6. Logging and Testing Procedure

A detailed description of the logs and procedure you will use to verify that the system is working the way you say it is working. In most cases this should include a **regression** test so that when someone changes something else, we can make sure that your thing still works as well as it used to.

## 29.7. Current status

Write here the current status. What works now, as opposed to what the goal is. The difference between these two is the work to be done.

> Note: it is better to have something that does not work, and a good description of what should work and why it doesn't work, than to have something that kinda works, but nobody knows what the thing is supposed to do.

### 1) Functionality

Nothing implemented.

### 2) Performance

Infinitely slow.

## 29.8. Tasks

So and so should do such and such

## 29.9. Timeline

- This should be done by Nov 15
- That shoudl be done by Nov 16

## 29.10. Meetings notes

- Link 1
- Link 2

# Unit M-30
# The Map Description

The map to be used in the Fall 2017 class is shown in Figure 30.1.

Figure 30.1. The map to be used in the Fall 2017 class

The editable keynote file is in this directory of the duckuments repo. The ids on the signs correspond to the Apriltag IDs. For more details see Unit D-3 - Signage.

# Part N
# Fall 2017 projects

Welcome to the Fall 2017 projects.

## 0.1. Instructions for using the template

1. Make a copy of the template `10_templates` folder and paste it inside `/atoms_85_fall2017_projects`.
2. Rename the folder to the next available integer followed by the short group name. E.g.: `10_templates` becomes `11_first_group_name` for the first group, then `12_second_group_name` for the second, and so forth.
3. Edit the `10-preliminary-design-document-template` file name by substituting `template` with `group-name`
4. Open the preliminary design document and personalize the template to your group.

> **Note:** All groups have got their unique ID number and folders are renamed according to the following table. You are allowed and encouraged to use short names. Please merge from the master. New pull requests conflicting to this table will be rejected.

## 0.2. Group names and ID numbers

| ID | Group name | Short name |
|---|---|---|
| 11 | The Heroes | heroes |
| 12 | The Architects | smart-city |
| 13 | The Identifiers | sysid |
| 14 | The Controllers | controllers |
| 15 | The Saviors | saviors |
| 16 | The Navigators | navigators |
| 17 | Parking | parking |
| 18 | The Coordinators | explicit-coord |
| 19 | Formations and implicit coordination | implicit-coord |
| 20 | Distributed estimation | distributed-est |
| 21 | Fleet-level planning | fleet-planning |
| 22 | Transfer-learning | transfer-learning |
| 23 | Supervised learning | super-learning |
| 24 | Neural-slam | neural-slam |
| 25 | Visual-odometry | visual-odometry |
| 26 | Single-slam | single-slam |
| 27 | Anti-instagram | anti-instagram |

UNIT N-1

# Group name: preliminary design document

## 1.1. Part 1: Mission and scope

### 1) Mission statement

Instructions: What is the overarching mission of this team? You should write in one sentence.

Instructions: What is the need that is being addressed? Do not focus on technical specifics yet.

### 2) Motto

Instructions: Your rallying cry into battle. Traditionally, Duckietown uses Latin mottos. If you don't speak Latin, please contact Jacopo Tani to have your motto translated into latin.

QUIDQUID LATINE DICTUM SIT, ALTUM VIDETUR

(Anything that is said in Latin sounds important)

### 3) Project scope

Instructions: Are you going to rewrite Duckietown from scratch? Probably not. You need to decide what are the boundaries in which you want to move.

*What is in scope:*

Instructions: What do you consider in scope? (e.g. having a different calibration pattern)

*What is out of scope:*

Instructions: What do you consider out of scope? (e.g. hardware modifications)

*Stakeholders:*

Instructions: What other pieces of Duckietown interact with your piece?

Instructions: List here the teams.

## 1.2. Part 2: Definition of the problem

### 1) Problem statement

Time to define the particular problem that you choose to solve.

Suppose that we need to free our prince/princess from a dragon. So the mission is clear:

Mission = we must recover the prince/princess.

Now, are we going to battle the dragon, or use diplomacy?

If the first, then the problem statement becomes:

Problem statement = We need to slain a dragon.

Otherwise:

Problem statement = We need to convince the dragon to give us the prince/princess.

Suppose we choose to slain the dragon.

### 2) Assumptions

At this point, you might need to make some assumptions before proceeding.

- Does the dragon breath fire?
- What color is the dragon? Does the color matter?
- How big is this dragon, exactly?

### 3) Approach

All right. We are going to kill the dragon. How? Are we going to battle the dragon? Are we trying to poison him? Are we going to hire an army of mercenaries to kill the dragon for us?

### 4) Functionality-resources trade-offs

The space of possible implementations / battle plans is infinite. We need to understand what will be the trade-offs.

### 5) Functionality provided

How do you measure the functionality (what this module provides)? What are the "metrics"?

*example* numbers of dragons killed per hour

Note that this is already tricky. In fact, the above is not a good metric. Maybe we kill the dragon with an explosion, and also the prince/princess is killed. A better one might be:

*example* numbers of royals freed per hour

*example* probability of freeing a royal per attempt

It works better if you can choose the quantities so that functionality is something that you maximize to maximize. (so that you can "maximize performance", and "minimize resources").

#### 6) Resources required / dependencies / costs

How do you measure the resources (what this module requires)?

*example* numbers of knights to hire

*example* total salary for the mercenaries.

*example* liters of poison to buy.

*example* average duration of the battle.

It works better if you think of these resources as something to minimize.

#### 7) Performance measurement

How would you measure the performance/resources above? If you don't know how to measure it, it is not a good quantity to choose.

*example* we dress up Brian as a Dragon and see how long it takes to kill him.

### 1.3. Part 3: Preliminary design

#### 1) Modules

Can we decompose the problem?

Can you break up the solution in modules?

Note here we talk about logical modules, not the physical architecture (ROS nodes).

#### 2) Interfaces

For each module, what is the input, and what is the output?

How is the data represented?

Note we are not talking about ROS messages vs services vs UDP vs TCP etc.

#### 3) Preliminary plan of deliverables

What needs to be designed?

What needs to be implemented?

What already exists and needs to be revised?

#### 4) Specifications

Do you need to revise the Duckietown specification?

#### 5) Software modules

Here, be specific about the software:is it a ROS node, a Python library, a cloud service, a batch script?

#### 6) Infrastructure modules

Some of the modules have been designated as infrastructure

### 1.4. Part 4: Project planning

Now, make a plan for the next phase.

#### 1) Data collection

What data do you need to collect?

#### 2) Data annotation

Do you have data that needs to be annotated? What would the annotations be?

*Relevant Duckietown resources to investigate:*

List here Duckietown packages, slides, previous projects that are relevant to your quest

*Other relevant resources to investigate:*

List papers, open source code, software libraries, that could be relevant in your quest.

#### 3) Risk analysis

What could go wrong?

How to mitigate the risks?

# Unit N-2
# Group name: intermediate report

*It's time to commit on what you are building, and to make sure that it fits with everything else.*

This consists of 3 parts:

- Part 1: System interfaces: Does your piece fit with everything else? You will have to convince both system architect and software architect and they must sign-off on this.
- Part 2: Demo and evaluation plan: Do you have a credible plan for evaluating what you are building? You will have to convince the VPs of Safety and they must sign-off on this.
- Part 3: Data collection, annotation, and analysis: Do you have a credible plan for collecting, annotating and analyzing the data? You will have to convince the data czars and they must sign-off on this.

Table 2.1. Intermediate Report Supervisors

| | |
|---|---|
| System Architects | Sonja Brits, Andrea Censi |
| Software Architects | Breandan Considine, Liam Paull |
| Vice President of Safety | Miguel de la Iglesia, Jacopo Tani |
| Data Czars | Manfred Diaz, Jonathan Aresenault |

## 2.1. Part 1: System interfaces

Please note that for this part it is necessary for the system architect and software architect to check off before you submit it. Also note that they are busy people, so it's up to you to coordinate to make sure you get this part right and in time.

### 1) Logical architecture

- Please describe in detail what the desired functionality will be. What will happen when we click "start"?
- Please describe for each quantity, what are reasonable target values. (The system architect will verify that these need to be coherent with others.)
- Please describe any assumption you might have about the other modules, that must be verified for you to provide the functionality above.

### 2) Software architecture

- Please describe the list of nodes that you are developing or modifying.
- For each node, list the published and subscribed topics.
- For each subscribed topic, describe the assumption about the latency introduced by the previous modules.
- For each published topic, describe the maximum latency that you will introduce.

## 2.2. Part 2: Demo and evaluation plan

*Please note that for this part it is necessary for the VPs for Safety to check off before you submit it. Also note that they are busy people, so it's up to you to coordinate to make sure you get this part right and in time.*

#### 1) Demo plan

The demo is a short activity that is used to show the desired functionality, and in particular the difference between how it worked before (or not worked) and how it works now after you have done your development.

It should take a few minutes maximum for setup and running the demo.

- How do you envision the demo?
- What hardware components do you need?

#### 2) Plan for formal performance evaluation

- How do you envision the performance evaluation? Is it experiments? Log analysis?

In contrast with the demo, the formal performance evaluation can take more than a few minutes.

Ideally it should be possible to do this without human intervention, or with minimal human intervention, for both running the demo and checking the results.

### 2.3. Part 3: Data collection, annotation, and analysis

*Please note that for this part it is necessary for the Data Czars to check off before you submit it. Also note that they are busy people, so it's up to you to coordinate to make sure you get this part right and in time.*

#### 1) Collection

- How much data do you need?
- How are the logs to be taken? (Manually, autonomously, etc.)

Describe any other special arrangements.

- Do you need extra help in collecting the data from the other teams?

#### 2) Annotation

- Do you need to annotate the data?
- At this point, you should have you tried using thehive.ai to do it. Did you?
- Are you sure they can do the annotations that you want?

#### 3) Analysis

- Do you need to write some software to analyze the annotations?
- Are you planning for it?

UNIT N-3

# Group name: final report

*The objective of this report is to bring justice to your hard work during the semester and make so that future generations of Duckietown students may take full advantage of it. Some of the sections of this report are repetitions from the preliminary and intermediate design document (PDD, IDD respectively).*

## 3.1. The final result

*Let's start from a teaser.*

- Post a video of your best results (e.g., your demo video)

Add as a caption: see the operation manual to reproduce these results.

## 3.2. Mission and Scope

*Now tell your story:*

Define what is your mission here.

### 1) Motivation

*Now step back and tell us how you got to that mission.*

- What are we talking about? [Brief introduction / problem in general terms]
- Why is it important? [Relevance]

### 2) Existing solution

- Was there a baseline implementation in Duckietown which you improved upon, or did you implemented from scratch? Describe the "prior work"

### 3) Opportunity

- What didn't work out with the existing solution? Why did it need improvement?

Examples: - there wasn't a previous implementation - the previous performance, evaluated according to some specific metrics, was not satisfactory - it was not robust / reliable - somebody told me to do so (/s)

- How did you go about improving the existing solution / approaching the problem? [contribution]
- We used method / algorithm xyz to fix the gap in knowledge (don't go in the details here)
- Make sure to reference papers you used / took inspiration from

### 4) Preliminaries (optional)

- Is there some particular theorem / "mathy" thing you require your readers to know before delving in the actual problem? Add links here.

Definition of link: - could be the reference to a paper / textbook (check here how to add citations) - (bonus points) it is best if it is a link to Duckiebook chapter (in the dedicated "Preliminaries" section)

## 3.3. Definition of the problem

*Up to now it was all fun and giggles. This is the most important part of your report: a crisp mathematical definition of the problem you tackled. You can use part of the preliminary design document to fill this section.*

Make sure you include your: - final objective / goal - assumptions made (including contracts with "neighbors") - quantitative performance metrics to judge the achievement of the goal

## 3.4. Contribution / Added functionality

Describe here, in technical detail, what you have done. Make sure you include: - a theoretical description of the algorithm(s) you implemented - logical architecture (refer to IDD template for description) - software architecture (refer to IDD template for description) - details on the actual implementation where relevant (how does the implementation differ from the theory?) - any infrastructure you had to develop in order to implement your algorithm - If you have collected a number of logs, add link to where you stored them

*Feel free to create subsections when useful to ease the flow*

## 3.5. Formal performance evaluation / Results

*Be rigorous!*

- For each of the tasks you defined in you problem formulation, provide quantitative results (i.e., the evaluation of the previously introduced performance metrics)
- Compare your results to the success targets. Explain successes or failures.
- Compare your results to the "state of the art" / previous implementation where relevant. Explain failure / success.
- Include an explanation / discussion of the results. Where things (as / better than / worst than) you expected? What were the biggest challenges?

## 3.6. Future avenues of development

*Is there something you think still needs to be done or could be improved? List it here, and be specific!*

# Unit N-4
# The Heroes quests

The “Heroes” team is a special task force with the responsibility to make sure that “everything works” and create a smooth experience for the rest of the teams, in terms of developing own projects, integration with other teams and documentation. Apart from that, each of the heroes will also have their own individual quest...

## 4.1. Motto

E PLURIBUS UNUM

(From many, unity)

## 4.2. Overview

### 1) Responsibility

The system architect is ultimately responsible:

- The logical architecture
  - Decomposition in modules
  - Who knows what
  - Who says what to whom
- Definition of performance metrics
- Definition of contracts (bounds on metrics)

### 2) Philosophy

- Look over Duckie teams and remind them of the greater goals of Duckietown.
- Improve Duckietown by using a higher dimensional view.

### 3) Quests

The system architect project can be split into two quests:

1. Ensure that the development of the system goes smoothly (wooden spoon)
2. Develop a framework/tool to formally describe (and later optimize) the system (bronze, silver and gold)

The two quests and their respective descriptions will be explained separately in this document.

## 4.3. Quest 1

With many teams working on many different parts of the system, chaos is inevitable (without divine intervention). Quest 1 is to minimise the chaos by acting as system-level wactchdog; spotting and addressing interface, contract and dependency issues between the teams.

## 4.4. Quest 1, Part 1: Mission and scope

### 1) Mission statement

Ensure that the development and integration of the projects into the system goes smoothly and that the resulting system makes sense, and is useful for future duckierations (duckie + generations).

### 2) Project scope

*What is in scope:*

- Documentation regarding system architecture (including ownership of functional diagram of system)
- Assisting in contract negotiation
- Giving advice to teams regarding functional layout and interfaces

*What is out of scope:*

- Lower level architecture such as message formats, coding conventions etc (see Software Architect project)
- Documentation framework itself (see Knowledge Czarina project)

*Stakeholders:*

- The Duckietown masters
- All other project teams that are part of the system

## 4.5. Quest 1, Part 2: Definition of the problem

### 1) Problem statement

Ensure that all teams know what their goal is and how it fits into the bigger picture.

### 2) Assumptions

The current system makes at least kind-of sense. The current system will be used as a base, onto which improvements or functionality will be added by the projects.

### 3) Approach

- Become one with the goals of Duckietown
  - In order to make Duckietown a better place, one has to keep in mind what "better" means in Duckie terms.
- Be familiar with the current system architecture and track changes
  - This can include having to update the functional diagram, for instance.
  - This also means identifying which teams affect which modules in the diagram
- Keep in close contact with teams

  - All teams will designate a contact person who can contact me whenever they change their project boundaries or have doubts/ need advice on their project's boundaries/negotiating with other
  - The meetings of some of the other teams will be attended(especially early meetings). Some teams' meetings have been prioritized since many parts of the system are dependant on their work, namely:
    - Anti-instagram
    - Controllers
    - Navigators
    - Explicit coordination
- Offer nudges in a different direction if needed
- Acting as middleman/helper to facilitate negotiation of contracts between groups
- Monitor status of projects to find possible problems

#### 4) Functionality-resources trade-offs

*Functionality provided:*
- System integration of project modules

*Resources required / dependencies / costs:*
- Biggest resource: Time
- Finding out how to maximise usefulness while being efficient with time

*Performance measurement:*
- Approval of Duckietown masters
- Number of miscommunications about contracts between teams (measured in what-the-ducks per second)
- How many things didn't go wrong
  - Some jobs are of the type where no one notices you until something doesn't work. You should be the silent angel fixing all the problems that no one even noticed existed.

## 4.6. Quest 1, Part 3: Preliminary design

#### 1) Preliminary plan of deliverables

- Functional diagram of the system
  - The system functional diagram will be the main tool to visualise the system decomposition, and show the relationships between the different teams.
- Documentation of system architecture

## 4.7. Quest 1, Part 4: Project planning

#### 1) First steps for next phase

- Familiarisation with existing system architecture
- Going to group meetings
- Identifying potential problems

*Relevant Duckietown resources to investigate:*

- The functional diagram of the system
- Other teams' preliminary design reports

#### 2) Risk analysis

**Challenges:**

- Maintaining the balance between project level scope and Duckietown level scope. For instance, teams are focused on completing their project, and might forget the greater vision of Duckietown. This might mean having to convince teams to do slightly more work, for it to be more useful to Duckietown. After all, what's the point of doing a project if it does not contribute to Duckietown?
- Balancing priorities of quest 1 and 2. Quest 1 is crucial, and takes priority over quest 2. Therefore it will be challenging to find time (main resource) to work on quest 2.

## 4.8. Quest 2

Where there is a system, there is a want (nay, need) for optimisation. Describing a system's performance and resource requirements in a quantifiable way is a critical part of being able to benchmark modules and optimise the system.

The different levels of quest 2 are defined as follows:

- **Bronze standard:**
  - Define a formal, qualitative language to describe constraints/requirements between modules.
- **Silver standard:**
  - Each module has table of performance. A tool is developed that can give a qualitative answer to the question: With given configuration x, is the cost/requirements possible with available resources? f(x) smaller equal to Rmax?
- **Gold standard:**
  - Optimization of the system is possible to find the best implementation (most functionality and performance), given the available resources. Given f(x) and Rmax, find the/an optimal configuration x.

## 4.9. Quest 2, Part 1: Mission and scope

#### 1) Mission statement

Formalise the description of the system characteristics, so that eventually the system performance can be optimised for some given resources.

*Stakeholders:*

- The Duckietown masters, for they have started me on this quest.
- The future users of Duckietowns, as the Duckietown experience can be optimised per user, given their available resources.

## 4.10. Quest 2, Part 2: Definition of the problem

#### 1) Problem statement

- Find a way to describe all the module requirements, specifications, etc in a formal, quantifiable language.
- Find a way to calculate the requirements and specifications of a whole system or subsystem, based on the requirements and specifications of the individual modules of the system.
- Find a way to calculate the optimal system configuration, based on the desired requirements and specifications of the system.

#### 2) Approach

- Research on the topic of formal description of a system
- Find/develop a suitable language to describe module characteristics
- Require groups to compile a description of their respective modules' characteristics
- Find/develop functions to do mathematics on the language description of modules

#### 3) Functionality-resources trade-offs

*Functionality provided:*
- Being able to quantifiably describe the system
- Being able to compare and benchmark different implementations of parts of the system

*Resources required / dependencies / costs:*
- Time is the main resource that needs to be divided between tasks.
- The current system's characteristics will need to be measured/evaluated somehow to create a database of all the modules' specifications.

*Performance measurement:*
- How effectively is the system described, and can the tool provide the answers we seek.
- Which level of funcitonality was provided? (bronze, silver, gold)

## 4.11. Quest 2, Part 3: Preliminary design

#### 1) Preliminary plan of deliverables

- Documentation on the result of the project
- A description of the current system's characteristics (bronze)
- A program/tool that can give a qualitative answer (yes/no) to the question: Are these resources sufficient for this system configuration? (silver)
- A program/tool that will give an optimised system configuration, based on the given available resources (gold)

## 4.12. Quest 2, Part 4: Project planning

#### 1) First steps for next phase

- Look into yaml description of modules
- Research into existing methods of system description

- Graph based databases?
- Perhaps graph theory can be useful later if the (suspiciously graph-looking) system can be described suitably.

*Data collection:*

- Module descriptions can be collected from the respective groups (by asking nicely)

*Relevant Duckietown resources to investigate:*

- `devel-heroes-formal-description` branch on Software repository
- How the current system's characteristics are defined
- What are the dependencies/interfaces between modules
- Which projects affect which modules
- Which values/parameters are needed

## 2) Risk analysis

- Quest 1 takes priority over quest 2, since it is more crucial to the functioning of the system. This means that quest 2 may suffer if quest 1 takes more time than expected.
- Quest 2 has a research/experimental aspect, which makes it both interesting and challenging.
- There is a chance that it might not be solved, as it is not a trivial problem.

# Unit N-5
# PDD - Smart City

## 5.1. Part 1: Mission and scope

### 1) Mission statement

Make Duckietown a smarter city.

### 2) Motto

OMNES VIAE ANATUM URBEM DUCUNT

(All roads lead to Duckietown)

### 3) Project scope

*What is in scope:*

- Manufacturing process of tiles
  - Consider different ways to implement the lines on the road
  - Spray tiles for lines instead of tape
- Power Grid
  - Add power to each tile
  - Establish power grid layout
  - Establish power grid design and implementation

*What is out of scope:*

- Communication protocol (what to do with the data)
- Appearance specifications redefinition (cit. “we need an Italian architect for this”)
- Traffic controller HUB
- City wide power hub

*Stakeholders:*

| Team | Reference Person |
|---|---|
| Intersection Navigation | Nicolas Lanzetti (ETHZ) |
| Parking | Samuel Nyffenegger (ETHZ) |
| Traffic controller HUB | no teams actively working on this project in Fall-2017 |
| System Architect | Sonja (ETHZ) |
| Software Architect | Breandean (UdM) |
| Knowledge | Tzarina |

## 5.2. Part 2: Definition of the problem

### 1) Problem statement

We have to design a traffic lights system that integrates seamlessly and efficiently with the tiles currently used to build Duckietown. The development of a traffic lights system has to be considered as part of a bigger plan aimed to make Duckietown a smart city. A smart Duckietown has the capability of delivering wireless connectivity everywhere (Duckietown Wireless Network - DWN) in the town and power to each tile. A tile that can provide power is called a hot tile. The power grid that provides power to all the tiles is called Duckietown Power Grid (DPG). A simple use case for this infrastructure would then be the traffic lights system. Traffic lights at each intersection are powered and controlled by a Raspberry Pi with a Duckiebot-like LED Hat and 3 (or 4) LEDs. A Raspberry Pi responsible for the traffic lights at an intersection draws power from a hot tile and connects to the DWN.

## 2) Terminology

- DWN: Duckietown Wireless Network
- DPG: Duckietown Power Grid
- PD: Powered Device

## 3) Assumptions

- 2 traffic light per current city design (1 at 4-way the other at 3-way intersection)
- Appearance specifications are given and must be respected
- There exists a Duckietown Wireless Network (DWN) that any wifi enabled device placed within the town can connect to (e.g. traffic lights).
- There is access to power source close to Duckietown that can power the power grid

## 4) Approach

To create a smarter Duckietown and provide data and power to the tiles, we will wireless networks (e.g., WiFi, Bluetooth, etc.) for data communication, and we will implement a power grid to provide power to the various devices and PDs throughout Duckietown. Since we are using common implementation of wireless networks, the rest of this design document will focus on the specifications of the power grid.

**Power Grid Implementation Ideas:**

- Idea 1: Attach a 2-row breadboard along the edges of each tile, between the white tape and the teeth of the tile. PDs are connected to the power grid simply by inserting the two wires (+ and -) into the relative holes.
  - Problems: The primary problem under this approach is that breadboards are rated for only ~1amp, which is not nearly enough power needed for the power grid (for example, a single Raspberry Pi can use more power than that. This problem essentially eliminates the feasibility of this idea for the DPG.
- Idea 2: Attach a plastic rail to the edge of each tile, between the white tape and the teeth of the tile. The rail would carry two conductive strips (copper strips), one on each side (see image below).
  - Prototype: The image below shows a possible design of the plastic rail along with a compatible plug. The black part of the 3D model above constitutes the rail (sectional view) while the white part is the plug. The system is designed so that the plug, once pressed onto the rail, remains attached. The white box on the plug would contain one of the step-down converters (http://a.co/fAIAhuw)

described above. This would solve the problem of having a weak 5V power grid by running 24V through the grid and stepping it down to 5V only when, and exactly where, we need it. There would be limit neither to the number of plugs nor to the position where we can attach them (even better than a breadboard in this sense). We can then design simple connectors for straight and curved tiles to make everything modular. Since the most common PD in Duckietown is a Raspberry Pi, we can design a USB plug (shown below) to make things even easier.

- Enhancement 1: We can modify the plug by adding an extrusion to one side and carving its negative into the rail. This would prevent us from attaching the plug in the wrong direction, thus violating the positive/negative polarity of the conductors.
- Enhancement 2: Since the plastic material used by 3D printers is usually inflexible we can change the plug such that the plastic does not follow the design of the rail (i.e., it would look like a U flipped upside-down) and have a curved copper strip that stretches when the plug is pushed onto the rail and loosens when the plug sits completely on the rail. Basically, it follows the same concept used in the classic cigarette lighter plug present in a vehicle.
- Enhancement 3: We can use plastic T-slotted extrusion elements and design a plug that works with them. Problem: The primary problem with this approach is its difficult, especially given the project's short timeframe. However, we could focus on designing and building a prototype that works that could then be mass produced and implemented for the whole Duckietown sometime in the future.

- Idea 3: Have the connectors between tiles also serve as the output location for power to the tile. Use audio cable or RCA cable for the power rails and connect them at the corners of the tile using a 3 way connector, such as those shown below. This approach solves the issue regarding gendering the connectors and providing power nodes to the city. Cheap and easy to mass produce.
    - Problem: This may require modification of the tiles, such as removing one of the interlocking teeth to allow for the connector.

## 5) Functionality provided

The actual voltage and amperage available at each tile/power terminal will depend on the power grid approach we choose. Regardless of the implementation, the primary functionality provided by the power grid is access to power for at each tile in the Duckietown.

## 6) Resources required / dependencies / costs

The resources for this project are the parts to build the traffic lights and the power grid. Since, the specific parts and associated costs for the power grid are highly dependent on the implementation approach we decide on, we are unable to obtain specific details at this time. However, for all of the approaches, we will need enough parts to build a power grid that provides power for all of the tiles in the Duckietown.

#### 7) Performance measurement

Power Grid:

- Maximum number of powered devices per tile
- Ease of assembly/disassembly
- Ease of manufacturing
- How robust is the power grid under normal usage

System:

- Maximum image frame-rate traffic lights can sustain over the utilities network (network bandwidth)

## 5.3. Part 3: Preliminary design

#### 1) Modules

- Laying the power cables
- Connecting the power cables
- Output power for tile

#### 2) Interfaces

Input: 12/24 V, Output: 12/24 V between tiles, 5 V on tile

#### 3) Preliminary plan of deliverables

Power grid and integration into the individual tiles must be designed and implemented. While the traffic lights exist, there needs to be a revised method of providing power.

#### 4) Specifications

May have to modify Duckietown tiles.

#### 5) Software modules

None, this is a hardware project.

#### 6) Infrastructure modules

All modules are infrastructure.

## 5.4. Part 4: Project planning

#### 1) First Steps for the next phase

Decide connector option and wire routing.

#### 2) Data collection

Stability of power grid. How many traffic lights can be supported per voltage source.

### 3) Data annotation

None.

### 4) Relevant Duckietown resources to investigate

Specification of traffic light.

### 5) Other relevant resources to investigate

None.

### 6) Risk analysis

What could go wrong?

- Wire gauge too low to accommodate power load, causing shorts and possibly melting tiles or starting small fires.
- Live wires are exposed and come into human contact.

How to mitigate the risks?

- Appropriately fuse the tiles and use appropriate wires for power load.
- Insulate everything well and keep open contacts small and covered.

UNIT N-6

# PDD - System Identification

## 6.1. Part 1: Mission and scope

### 1) Mission statement

Estimate better models to make localization and control more efficient and robust to different configurations of the robot.

### 2) Motto

NOSCE TE IPSUM

(Know thyself)

### 3) Project scope

*What is in scope:*

- hardware specifications for calibration
- choose which model to "identify"
- Identify kinematic parameters (mapping of commands to actuators)
- Include model of the caster wheel

*What is out of scope:*

- Additional onboard or external sensors
- Measuring the latency of the system
- Camera calibration
- State estimation using motion blur

*Stakeholders:*

- Control
- localization

## 6.2. Part 2: Definition of the problem

### 1) Problem statement

Every Duckiebot is different in configuration.

> Mission = we need to make control robust to different configuration
>
> Problem statement = we need to identify kinematic model to make control robust enough

### 2) Assumptions

- Robot will move only in horizontal plane

- No lateral slipping of robot
- The body fixed longitudinal velocity and angular velocity will be provided as well as the timestamp of each measurement

### 3) Approach

- Simplified kinematic model will be used to estimate parameters for each duckiebot :
  - Semi axis length l
  - Mapping : voltage → velocity

*Kinematic model:*

We make use of the no lateral slipping motion hypothesis and the pure rolling constrain as shown in the #duckiebot-modeling, to write the following equation:

$$\begin{cases} \dot{x}_A^R &= R(\dot{\varphi}_R + \dot{\varphi}_L)/2 \\ \dot{y}_A^R &= 0 \\ \dot{\theta} &= \omega = R(\dot{\varphi}_R - \dot{\varphi}_L)/(2L) \end{cases}, \tag{1}$$

Further we make the assumption that for steady state that there is a linear relationship between the input voltage and the velociy of the wheel:

$$\begin{aligned} v_r &= R\dot{\varphi}_l = c_r V_r \\ v_r &= R\dot{\varphi}_l = c_l V_l \end{aligned} \tag{2}$$

This lets us rewrite equation (1):

$$\begin{cases} \dot{x}_A^R &= (c_r V_r + c_l V_l)/2 \\ \dot{y}_A^R &= 0 \\ \dot{\theta} &= \omega = (c_r V_r + c_l V_l)/(2L) \end{cases}, \tag{3}$$

Using the assumption that we can measure $v_A$ we can determine $c_r$ by setting the voltage $V_l = 0$. The same procedure can be done to get $c_l$.

Using the assumption that we can measure $\dot{\theta}$ we will then get the semiaxis length $L$.

Figure 6.1. Relevant notations for modeling a differential drive robot

### 4) Functionality provided

- A model with calibrated parameters for each duckiebot
- A calibration protocol that creates a map of:
  - input voltage → output longitudinal and angular velocity
- Semi axis length

### 5) Resources required / dependencies / costs

- Good state estimation, independent of a model
- potential approaches for state estimate
  - lane filter
  - april tags
  - camera calibration sheet
- Accurate line-detection
- Accurate april tags

### 6) Performance measurement

Run lane follower with old version and new version with kinematic model. Drive on the track for one minute and count the number of times the bot touches the side or center line.

**Metrics**

- Robustness to different duckies
  - Control can handle different duckiebot configurations based on our models-
- Robustness to different wheels
  - Omnidirectional wheel, caster wheel
- Robustness to initial pose

  - Run lane following using 5 different initial poses
- Repeatability
  - Run lane following 5 times and compare

We will use the performance measurement setup of the devel-control group

## 6.3. Part 3: Preliminary design

### 1) Modules and interfaces

Parameter estimation

- Input :
  - State estimation
  - Specific voltage to each wheel
- Output :
  - semi axis length and wheel radii

Mapping voltage → velocity

- Input :
  - Specific velocity to each wheel
- Output :
  - Voltage

### 2) Specifications

Duckiebots with different hardware configurations for testing

### 3) Software modules

- Parameter estimation:
  - runs calibration protocol
- Velocity translation: (Node)
  - get velocity as input and translate it to voltage as output

### 4) Infrastructure modules

None

## 6.4. Part 4: Project planning

| Date | Task Name | Target Deliverables |
|---|---|---|
| 17/11/17 | Kick-Off | Preliminary Design Document |
| 24/11/17 | Play around | Identify current problems |
| 01/12/17 | First estimation | find paramers of robot |
| 08/12/17 | Validation | Performance measure |
| 15/12/17 | Caster wheel | Performance measure of new implementation |
| 22/12/17 | Buffer | |
| 29/12/17 | Documentation | Duckuments |
| 05/01/18 | | End of Project |

### 1) Data collection

What data do you need to collect?

## 2) Data annotation

Performances of the current implementation

*Relevant Duckietown resources to investigate:*

- Current State Estimation
- Calibration files

*Other relevant resources to investigate:*

Handbook of robotics

the above contains a number of interesting sections of relevance to the work of this group:

- exact modeling of caster wheel and the kinematic constraints it introduces (pg. 395)
- different system identification procedures: parametric or nonparametric (Chapter 14); in particular, a note on Observability (pg. 337)
- we want to maximize performance of control + localization. Control uses unicycle model in Frenet frame (pg. 803 of handbook of robotics)
- We need to identify wheel radii (r_1, r_2: assume equal at start = r), semi-axle length L, and motors steady state parameters (mapping between voltage and angular rate, i.e. mapping between voltage and velocity once (a) wheel radius is known and no slipping hypothesis is made).
- Adaptive control (pg. 147): another approach is implementing an adaptive controller. It is meant to work with plant uncertainty.

Caster wheel literature

## 3) Risk analysis

What could go wrong?

- It could happen that we identify a model which is not useful for control.
- Perfect model will be useless if control is not improved

Mitigation strategy:

- Early testing with control group

# UNIT N-7
# System Identification: Intermediate Report

## 7.1. Part 1: System interfaces

### 1) Logical architecture

**Desired functionality**

The desired functionality is a node that takes the desired linear and angular velocity as input and maps it to the input voltages for the motors.

**Approach**

In a first step we measure manually the longitudinal and angular velocity for given input voltage. The aim is to use these measurements to find the $c_r$, $c_l$ and $L$ (or $c$, trim, $L$) of our model that was introduced in the preliminary design document.

$$\begin{cases} \dot{x}_A^R & = (c_r V_r + c_l V_l)/2 \\ \dot{y}_A^R & = 0 \\ \dot{\theta} & = \omega = (c_r V_r - c_l V_l)/(2L) \end{cases}, \tag{1}$$

This "linear" velocity to voltage function can be used for testing by the controller group. If we manage the first step, we will move on to a second step. Here we will aim to get a "non-linear" velocity to voltage map. There will be a calibration procedure that creates a custom velocity to voltage map for each Duckiebot that should be independent for different hardware configurations. The current idea for the calibration procedure is to drive the Duckiebot with different voltage commands to the motor while a checkerboard is in its viewfield. The logs can then be used to extract a pose estimation of the Duckiebot compared to the checkerboard. The recorded voltage and the pose data will then be fitted in a nonlinear manner, maybe using the acado optimization toolbox.

**Assumptions**

***Ground truth estimation***

- Repeatability (currently investigating)
- Can get ground truth also while driving (currently investigating)
- Minimum distance from which each square is visible is sufficient to do calibration trajectories with the Duckiebot (currently investigating)
- The calibration procedure gives very accurate ($\leq 5mm$) pose estimates to do a decent mapping

***Performance***

- Our assumption is that once calibrated the Duckiebot will be able to repeat the same behavior, and the kinematics do not change.

### 2) Software architecture

**Calibration procedure Node**

To achieve the desired functionality, we will create a mapping from velocity states to input motor voltages that will be used by the control. The procedure that will be fol-

lowed is summarized as follows:

- Drive the Duckiebot with different open loop voltage commands to the motor on a 2x2 tile while a checkerboard is in its view field. The voltage commands should be chosen to allow sufficient exploration of the velocity states to input voltages mapping. Take a log of the motion for offline processing in the next step.
- Process the logs of the images to extract a pose estimation of the Duckiebot using the checkerboard as a reference.
- Fit The recorded voltage and the pose data in a nonlinear manner potentially using the Acado optimization toolbox.

*Input*

- Intrinsic camera calibration
- Extrinsic camera calibration
- Rosbag of raw images and control commands of the Duckiebot driving in front of the checkerboard, the rosbag is recorded by running the calibration node

*Output*

- Control commands during calibration procedure (topic: car_cmd, same as control)
- calibration.yaml file containing velocity to voltage map

*Assumed Latency*

Offline. The calibration procedure is done on the computer

*Position estimate*

In order to to identify a system model, we need the best possible state estimation. This shall be achieved by calculating the camera extrinsics from the checkerboard for each frame. The picture below shows our first experiments with the setup.

Figure 7.1. Setup for state estimation

**Inverse_kinematics_node** We will edit the existing inverse_kinematics_node. It will get the desired velocity as input and find the corresponding motor speeds using the parameters from the calibration.yaml file generated by the Calibration procedure. For desired velocities that exceed the system's capabilities, the maximum possible velocity will be returned.

*Subscribed message*

car_cmd

***Published message***

wheels_cmd

***Assumed Latency***

Negligible, will run a very lightweight callback directly once it receives a car_cmd message

## 7.2. Part 2: Demo and evaluation

### 1) Demo plan

The main goal is to demonstrate improved calibration. For this purpose, we will first run the lane following module with a Duckiebot that has the default velocity to voltage mapping on the same test track, see picture. Afterwards, we will run the improved calibration procedure that will create the custom velocity to voltage mapping. We will then run the lane following module again and see how lane following task is improved. As the actual calibration procedure will take sometime, we will do it beforehand.

Figure 7.2. Duckietown test track

## 7.3. Plan for formal performance evaluation

We will run all of the tests 3x times uncalibrated, and 3x calibrated for the following Duckiebot configuration:

- Normal Duckiebot
- Duckiebot with different right and left wheel diameters and compare the improvements

**Offset in straight line** :

We will let the Duckiebot drive straight in open loop and measure its offset after X tiles of straight lane in Duckietown. The performance metric will be the absolute position offset of the expected to the actual terminal position after the run, measured with a ruler.

**Circle test** :

We will will drive the Duckiebot with a constant velocity $v_a$ and constant angular velocity $\dot{\omega}$ in open loop on a Duckietown corner tile. We will compare the actual path with the desired path. This is done both clock and counterclockwise. The performance metric will be the absolute position offset of the expected to the actual terminal position after the run, measured with a ruler.

**Integration test** :

We want to test how the improved calibration affects the line following mode. Compare different behaviors in line-following mode.

**Material needed to do calibration**

- 4 Black tiles
- Normal Camera Calibration Checkerboard that can be attached vertically

**Material needed to do performance test**

- Duckietown with straights and corners
- Ruler to measure offset of terminal position

## 7.4. Part 3: Data collection, annotation, and analysis

### 1) Collection

- How much data do you need?

At least one log file (rosbag) of the Duckiebot being driven with various voltage inputs and has a camera calibration checkerboard in view at all times. The checkerboard will be used to estimate position from images. We'll write the code for pose estimation (eg. assemble existing libraries).

- How are the logs to be taken? (Manually, autonomously, etc.)

Initially logs will be taken manually, but later will be taken automatically by a calibration node. The Duckiebot, should be on a Duckietown tile and have the checkerboard in view.

- Do you need extra help in collecting the data from the other teams?

We do not need data from other teams and therefore do not need help.

### 2) Annotation

- Do you need to annotate the data?

No, we need to extract pose from images, which is done by geometry from a checker-

board, not annotation

- At this point, you should have you tried using thehive.ai to do it. Did you?

No, because we do not need any annotations.

## 7.5. Analysis

- Do you need to write some software to analyze the annotations?

We don't need data annotation since we can do all the benchmarking by our own. However we are writing the software to estimate the model based on data collection.

We already did a basic analysis of the system by running control commands and measuring the distance or angle by hand. The duration and magnitude of the control commands were extracted from a rosbag. This lets us generate a map from control to velocities, thus basically we have a first guess of the model parameters (average C, C/2L). The results can be found in the two plots below. The estimated speed of the Duckiebot in lane following mode is 0.27 m/s and the yaw rate at maximum control input 3.7 rad/s. The estimate control gains are 0.67 m/s ($\pm 15\%$) per $v_{cmd}$ and 0.45 rad/s ($\pm 30\%$) per $\omega_{cmd}$.

Figure 7.3. Estimated linear velocity

Figure 7.4. Estimated yaw rate

# Unit N-8
# The Controllers: preliminary report

## 8.1. Part 1: Mission and scope

### 1) Mission statement

Make lane following more robust to model assumptions and Duckietown geometric specification violations and provide control for a different reference control.

### 2) Motto

IMPERIUM ET POTESTAS EST

(With control comes power)

### 3) Project scope

*What is in scope:*

- Control Duckiebot on straight lane segments and curved lane segments.
- Robustness to geometry (width of lane, width of lines)
- Detection and stopping at red (stop) lines
- Providing control for a given reference **d** for avoidance and intersections (but for intersections, we additionally need the pose estimation and a curvature from the navigators team)

*What is out of scope:*

- Pose estimation and curvature on Intersections (plus navigation / coordination)
- Model of Duckiebot and uncertainty quantification of parameters (System Identification)
- Object avoidance involving going to the left lane
- Extraction and classification of edges from images (anti-instagram)
- Any hardware design
- Controller for Custom maneuvers (e.g. Parking, Special intersection control)
- Robustness to non existing line

*Stakeholders:*

**System Architect**

She helps us to interact with other groups. We talk with her if we change our project.

**Software Architect**

They give us Software guidelines to follow. They give a message definition.

**Knowledge Tzarina** Duckiebook

**Anti-Instagram** They provide classified edges (differentiation of centerline, outer lines and stop lines)

Direction of the edges (background to line vs. line to background)

**Intersection Coordination (Navigators)** They tell where to stop at red line. We give a message once stopped. They give pose estimation and curvature (constant) to navigate on intersection. We provide controller for straight line or standard curves.

**Parking** They tell where to stop at red line. We give a message once stopped

**System Identification** They provide model of Duckiebot.

**Obstacle Avoidance Pipelines (Saviors)** They provide reference **d**.

**SLAM** They might want to know some information from our pose estimation (e.g. lane width or theta).

## 8.2. Part 2: Definition of the problem

### 1) Problem statement

We must keep the Duckiebots at a given distance d from center of the lane, on straight and curved roads, under bounded variations of the city geometric specifications.

Geometric specifications:

- Nominal lane width
- Tape width (white, yellow, red)
- Spacing between yellow dashed line
- Curvature of the curves

Given at every time step a reference **d** (the reference $\boldsymbol{\theta}$ gets choosen in a way to match the reference d):

$$q_r(t) = [x_r(t), y_r(t), \theta_r(t)]^t \rightarrow [d_r(t), \theta_r(t)]$$

The following image shows the definition of the parameters $\boldsymbol{d}$ and $\boldsymbol{\theta}$.

($\boldsymbol{d}$: distance perpendicular to the lane. 0 is defined in the center of the lane (see definition in duckumentation))

($\boldsymbol{\theta}$: angle between the lane and the robot body frame. )

and given a **model of the system**,

define a control action:

- the heading and velocity of the center between the wheels of robot, leading to a sequence of motor commands

at every time step, such that the pose (estimate of the pose) converges to the target.

**Performance:**

- Steady state within a tile
- Never leaves the lane
- Small steady state error

**Robustness to slight changes in:**

- Model parameters
- Width of the lane
- Width of the lines
- Curvature of the road

## 2) Assumptions

- There is a reason for the caster wheel
- A system model of the Duckiebot is provided
- The system model parameters for every duckiebot are given
- A set of classified lane edges are given with a certain frequency, latency, resolution and a maximum number of false positives and maximum number of misclassifications.
- Anti-instagram people do low level vision -> extract lines in image space
- Only small deviation from the specified geometric values
- Surface properties of Duckietown tiles are similar in each Duckietown
- Bounded initial pose, when driving straight no wheel of the Duckiebot should touch any line within 25cm

## 3) Approach

- Benchmark actual system → identify bottlenecks (in estimation and control)
- Identify bottlenecks by modifying different parts of the system and adding them each at a time and have a look at how much is the improvement → Make new branches for each of those modifications
- We want to improve the pose estimation by applying a particle filter. To measure the impact of the improved pose estimation, we will design a test procedure.
- One possible test procedure: Set a calibrated duckiebot to many known points on straight lanes and curved lanes. Save the information of these actual poses (measured by hand) together with the images taken by the Duckiebot's camera in the respective poses. On this data, different anti-instagram methods and different pose estimations can be run and evaluated directly, without any physical duckiebot nor Duckietown.
- We want to improve the parameters of the current controller by tuning it experimentally.
- We want to increase the frequency of the controller update.
- We want to handle actuator saturation, if we adding an I part to the controller.

## 4) Functionality provided

Drive on straight lane and curves without large deviations from the center of the lane.

#### 5) Resources required / dependencies / costs

Hardware resources:

- Tapes for lanes
- Tiles to make different straight lanes and curved lanes
- Timer
- Functional Duckiebot

Dependencies: see assumptions

We assume to have image space line segments extracted and classified from images.

- frequency
- latency
- accuracy (resolution)
- maximum false positives
- maximum misclassification error (confusion matrix)

#### 6) Performance measurement

Drive on the track for one minute and count the number of times the bot touches the side or center line. Repeat this 5 times.

**Metrics** *Error from the reference distance d when driving straight. - Mean and variance of 5 experiments* Estimate of lane width. - Estimate lane width and compare to measurement *Estimate road curvature. - Estimate curvature and compare to measured radius of curve* Speed - is a control variable. *Robustness to initial pose. - Run lane following using 5 different initial poses* Transient error after curved section (e.g. dies in one tile length). - 5 experiments of measuring the error d when driving straight after a curved segment → Did the transient error die? *Robustness to the curvature. - Run curve following on 5 lanes made of different combinations of curve tiles (left-left-right, left-right-left, ... )* Robustness to lane specifications - Run lane following on 5 lanes with different lane width when driving straight

## 8.3. Part 3: Preliminary design

#### 1) Modules

Estimation of Position:

- Input: segments detected by Anti-Instagram-Filter
- Output:
- Distance from center of lane,
- Heading angle,
- Curve or straight lane
- Curvature

Controller:

- Input: state (Distance from center of lane, heading angle, other) by an Estimator
- Output: Control Output to motors

### 2) Interfaces

- Anti-instagram: Labelled (centerline, outer line, stop line) edges with color and direction in image plane. The messages is defined already.
- Odometry calibration: get the model parameters. Yaml file
- Obstacle avoidance: get reference distance d from center of lane.. Zero assumed otherwise.
- Estimator: state vector at regular intervals.
- Control: Motor voltages at regular intervals

### 3) Specifications

We do not need to change the Duckietown specifications.

### 4) Software modules

- Estimator: NODE. There is a markovian approach. A particle filter should be implemented
- Controller: NODE. there is a P controller. A feedforward should be implemented. Pure-pursuit, or simple FeedForward.
- Outlier rejection: NODE (or part of estimator). there is nothing. A system has to be designed to detect edges that clearly don't belong to the line.
- Automated testing: NODE. There is nothing. A system should be implemented to test estimation from recorded data. It should be easy to update this data.

## 8.4. Part 4: Project planning

### 1) Timeline

| Date | Task Name | Target Deliverables |
|---|---|---|
| 17/11/17 | Kick-Off | Preliminary Design Document |
| 24/11/17 | Get familiar with state of the art | Benchmark state of the art, Identifiy bottlenecks |
| 01/12/17 | Theoretical derivation of Controller and Estimator | |
| 08/12/17 | Implementation of Controller and Estimator | |
| 15/12/17 | Benchmark new implementation | Performance measure of new implementation |
| 22/12/17 | Buffer | |
| 29/12/17 | Documentation | Duckuments |
| 05/01/18 | | End of Project |

### 2) Data collection

Take rosbag logs every time.

Rosbag:

- Image
- Edges from Anti-Instagram
- Motor control values

### 3) Data annotation

Curvature of road.

*Relevant Duckietown resources to investigate:*

Vision odometry Lane detection Anti instagram

*Other relevant resources to investigate:*

Particle Filter coded in python and useful intro to the subject

### 4) Risk analysis

| Risk | Likelihood | Impact | Risk response Strategy | Actions required |
|---|---|---|---|---|
| Cannot estimate curvature | 2 | 5 | mitigate | Start early with testing thresholds for curvature identification |
| Cannot define distance to curve | 2 | 4 | mitigate | Try various methods to identify the distance to curve |
| Duckiebot leaves the lane after curve (current situation) | 2 | 5 | mitigate | |
| Cannot handle the inputs given by other teams | 4 | 4 | mitigate | Get more information from Sonja, Talk to other teams, Clear comments in the code for easier problem detection |

# Unit N-9
# The Controllers: Intermediate Report

Table 9.1. Intermediate Report Supervisors

| | |
|---|---|
| System Architects | Sonja Brits, Andrea Censi |
| Software Architects | Breandan Considine, Liam Paull |
| Vice President of Safety | Miguel de la Iglesia, Jacopo Tani |
| Data Czars | Manfred Diaz, Jonathan Aresenault |

Table 9.2. Conventions used in the following document

| Variable | Description |
|---|---|
| $d_{ref}$ | Reference distance from center of lane |
| $d_{act}$ | Actual distance from center of lane |
| $d_{est}$ | Estimated distance from center of lane |
| $\theta_{act}$ | Actual angle between robot and center of lane |
| $\theta_{est}$ | Estimated angle between robot and center of lane |
| $c_{act}$ | Actual curvature of lane |
| $c_{est}$ | Estimated curvature of lane |
| $c_{ref}$ | Reference curvature of the path to follow |
| $v_{ref}$ | Reference velocity |

## 9.1. Part 1: System interfaces

### 1) Logical architecture

**Desired functionality**

We assume that the Duckiebot is placed on the right lane of the road within a defined boundary (as described in our preliminary design document) for the initial pose. By

starting the lane following module it should begin to follow the lane, whether it is straight or curved, until we stop the lane following module. The module consists of two parts, a pose-estimator part and a lane-controller part. We will briefly describe these two modules:

Figure 9.1. Pose of Duckiebot in a curve element.

- **Pose-estimator**: Estimates distance $d_{est}$ from the center of the lane and the angle $\theta_{est}$ with respect to the center of the lane as near as possible to the actual values $d_{act}$ and $\theta_{act}$. In a curve, $\theta_{act}$ is the angle between the centerline of the Duckiebot and the tangent to the centerline of the lane at the corresponding position (where the origin of the robot frame (center of the wheel axis) is closest to the centerline of the lane). Additionally, the estimator estimates the curvature $c_{est}$ of the lane. Further if possible, the estimator will approximate the lane width and the width of the side lines to be robust with respect to the geometric specifications of Duckietown. (The curvature $c_{est}$ was not estimated in the previous estimator.)
- **Lane-controller**: Given the pose ($d_{est}$, $\theta_{est}$) and curvature ($c_{est}$) estimation and a reference $d_{ref}$, the lane-controller will control the Duckiebot along this reference. The controller only accepts $d_{ref}$ which allow the Duckiebot to stay in the right lane. In other cases, the Duckiebot will be stopped to avoid accidents and a corresponding flag flag_obstacle_emergency_stop is set. The lane-controller further strictly limits the velocity of the Duckiebot, for values see next section. The lane-controller takes the velocity at all time from implicit coordination except when another team demands a lower value. In case no velocity is received, we set a default constant velocity by ourselves.

The following diagram shows the input the controller node needs to control for other teams.

Figure 9.2. Diagram showing values needed by the controller if used by other teams.

Special events:

- **Detection of red line**: After the stop line has been detected by the stop_line_filter_node, it sends an at_stop_line message (flag_at_stop_line), the controller will continually slow down the velocity of the Duckiebot and stop between 16 to 10 cm from the center of the red line and with an angle $\theta_{act}$ of +-10° (requirements given by Explicit Coordination Team). Furthermore the Duckiebot will stop in the center of the lane within a range of +- 5 cm.

Figure 9.3. Pose and distance range in front of red line.

- **Intersection**: When the Parking team set the flag_at_intersection *true* (because the Duckiebot is at a stop line and there is no april tag for a parking lot), we will stop the pose_estimator of the lane-following module and listen to data provided by the Navigators. It consists of the curvature $c_{ref}$ the Duckiebot needs to follow, as well as a reference $d_{ref}$ (which should be zero in case the Duckiebot needs to drive on the path with given curvature), the estimates of our distance $d_{est}$ and angle $\theta_{est}$ with respect to the path and a desired velocity $v_ref$. Everything on the intersection except of using our standard lane following controller is out of our scope. The pose_estimator of the lane-following module will start again after the intersection, triggered when the flag flag_at_intersection is turned *false*.
- **Obstacle avoidance**: Once flag_obstacle_detected is set *true* by the Savior Team, they will continuously send us references $d_{ref}$ to lead us around the obstacle, as well as a desired velocity $v_ref$. For better control performance, the velocity can be set lower than the usual velocity. The Saviors will start to send references when the Duckiebot still has a distance to the obstacle of at least 20-30cm to make sure the controller is able to react enough in advance. Out of scope is the controlled obstacle avoidance involving leaving the right lane. This case is determined by a stop flag (flag_obstacle_emergency_stop) sent from the Saviors module and the Duckiebot will stop. Since there is also a stop flag received from the stop_line_filter_node, we will introduce priorities for the several flags received to decide how the Duckiebot should be-

have.

- **Parking**: At a stop line, if a parking-lot april tag is detected (by the Parking team), the parking flag flag_at_parking_lot will be set *true* (otherwise the flag_at_parking_lot would be set *false* and the flag_at_intersection would be set *true*). After this the Parking team will take over responsibility. They will first calculate a path using RRT (Rapidly-Exploring Random Tree). In case, they set the flag_parking_stop to *false*, the lane controller will take over the control along this precalculated path. For this, the parking team needs to send the curvature $c_{ref}$ the Duckiebot needs to follow, as well as a reference $d_{ref}$ (which should be zero in case the Duckiebot needs to drive on the path with given curvature), the estimates of our distance $d_{est}$ and angle $\theta_{est}$ with respect to the path and a desired velocity $v_{ref}$. In case we need to stop, the Parking team will set the flag_parking_stop *true* again. Driving backwards is not in our scope. After leaving the parking lot, we will take over estimation and control again, unless directly after the parking lot is an intersection, which would be handled by the Navigators (after Explicit and/or Implicit Coordination).
- **Implicit Coordination**: If Implicit Coordination is running, they send to us the desired reference velocity $v_{ref}$ at all time and set the flag_implicit_coordination.
- **Fleet-level Planning**: As part of simulating pick-up / drop-off of customers, the Fleet-level Planning team will want to stop the Duckiebot at a certain distance from the center of the lane $d_{ref}$. Therefore they give us the desired $d_{ref}$ and a declining reference velocity $v_{ref}$ until the desired full stop.

**Target values**

The Duckiebot should run at a reduced velocity of 0.2 m/s for optimal controllability. The reason for the limited velocity is the low image update frequency which limits our pose estimation update, hence a lower velocity enhances the performance of our lane-follower module. Since not every Duckiebot has the same gain set, we will pass the desired velocity to team System Identification. Their module will convert the demanded velocity to the according input voltages for the motors.

Our goal is to control the deviation from the middle of the lane $d_{act}$ smaller than +- 2cm. The estimator should estimate our pose with $d_{est}$ and $\theta_{est}$ with an accuracy of +- 1cm. Further, the Duckiebot will stop in the center of the lane within a range of +- 2 cm, although a larger range of +- 5 cm is given by the explicit coordination team.

Whenever we detect a red line, we will slow down the Duckiebot and stop between 16 to 10 cm from the center of the red line and with an angle $\theta_{act}$ of +-10°, see caption 4.

In case the flag_obstacle_detected is activated by the Saviors, they will provide us with the input described above to avoid the obstacle without leaving the lane. Otherwise they activate the flag_obstacle_emergency_stop and we will need to stop the Duckiebot within 5 cm from the position at which the flag_obstacle_emergency_stop is received (requirement by Saviors).

**Assumptions**

We assume the following modules will behave in the described manner:

- **Savior**: They will detect obstacles on the road and are able to generate $d_{ref}$ that allows to avoid the object without leaving the lane or touching any object (safety for the Duckies!). They will set the flag_obstacle_detected 20-30cm in front of the obstacle, so we have enough time to avoid the obstacle. They are able to decide if avoidance of an obstacle is feasible or they have to set the flag_obstacle_emergency_stop.
- **Anti-instagram**: They can compensate the colors for different ambient light con-

ditions and allow for good edge detection robust to changing light conditions. In future, they could optionally help by introducing an area of interest of the image to process. Irrelevant image data could be filtered out to speed up the image processing pipeline. If Anti-Instagram could publish the area of interest as a node, we could use it in the estimation part to remove all visual clutter in the line segments.

- **Line detection**: Relevant line segments of the side, center and stop line are detected without introducing an exceeding amount of false detections and passed on in a SegmentList, classified including direction (from background do line or vice versa).
- **Navigators**: They are able to generate a path along the intersection and deliver accurate pose estimates which allow for proper working of the lane controller.
- **Parking**: They are able to generate a path from the parking entrance to a free parking space and deliver accurate pose estimates which allow for proper working of the lane controller. They take care of any backward driving without using the lane controller.
- **Implicit Coordination**: They are able to generate a reference velocity that will not result in a crash with another Duckiebot. They will provide enough distance to the Duckiebot in front, to allow the pose_estimator of the lane-following module and the stop_line_filter_node to work properly.
- **Fleet-level Planning**: They provide a profile of reference distance from the center of the lane $d_{ref}$ and velocity $v_{ref}$, such that they stop at their desired location to pick-up / drop-off their customer.

### 2) Software architecture

*Lane Filter Node*

The Lane Filter Node will, in addition to the existing fields, also estimate the curvature.

In the following table, published topics are listed:

TABLE 9.3. PUBLISHED TOPICS BY LANE FILTER NODE

| Topic | Max Latency |
|---|---|
| lane_pose | 15 ms |
| belief_img | 2 ms |
| entropy | negligible |
| in_lane | negligible |
| switch | negligible |

In the following table, subscribed topics are listed:

TABLE 9.4. PUBLISHED TOPICS BY LANE FILTER NODE

| Topic | Max Latency |
|---|---|
| segment_list | 25 ms |
| velocity | egligible |
| car_cmd (not yet) | negligible |

Total latency from image taken, processed through anti-instagram, up until setting the motor control command is on average 140 ms.

*Lane Controller Node*

In the following table, published topics are listed:

TABLE 9.5. PUBLISHED TOPICS BY LANE CONTROLLER NODE

| Topic | Type | Max Latency |
|---|---|---|
| car_cmd | duckietown_msgs/Twist2DStamped | negligible |

In the following table, subscribed topics are listed:

TABLE 9.6. SUBSCRIBED TOPICS BY LANE CONTROLLER NODE

| Topic | Type | Max Latency |
|---|---|---|
| lane_pose | duckietown_msgs/LanePose | 15 ms |
| lane_pose_intersection_navigation | duckietown_msgs/ControlMessage_ | 20 ms |
| lane_pose_obstacle_avoidance | duckietown_msgs/ControlMessage_ | 20 ms |
| lane_pose_parking | duckietown_msgs/ControlMessage_ | 25 ms |
| stop_line_reading | duckietown_msgs/StopLineReading | negligible |
| implicit_coordination_velocity | duckietown_msgs/ControlVelocity | negligible |
| Flags defined in table below | BoolStamped | negligible |

**Flags received by other nodes**

These following flags are received from other modules. While one of these flags is *true*, the Duckiebot will behave according to the descriptions in the system architecture section.

TABLE 9.7. FLAGS RECEIVED BY OTHER MODULES

| | |
|---|---|
| flag_at_stop_line | *True* when the distance to stop line is below a predefined distance. |
| flag_stop_line_deteced | *True* when number of detected red segments are above a threshold |
| flag_at_intersection | *True* when at intersection. This flag is passed to us by the Parking team. |
| flag_obstacle_detected | *True* when obstacle is in the lane. This flag is passed to us by the Saviors. |
| flag_obstacle_emergency_stop | *True* when it is not possible to avoid the obstacle without leaving the right lane. |
| flag_at_parking_lot | *True* when stopping at an intersection and the april tag for the parking lot is detected by the Parking team. |
| flag_parking_stop | Per default = *true*. If *false*, the lane-follower will move along the given trajectory on the parking lot. |
| flag_implicit_coordination | *True* when implicit coordination is running, in this case we listen to the velocity published by them. |

**Structure of the received messages with type *duckietown_msgs/LanePose***

The following table defines the structure of the pose messages the Lane Controller

Node receives.

TABLE 9.8. STRUCTURE OF POSE MESSAGE TO BE USED.

| Field | Abstract Data Type | SI Units | Description |
|---|---|---|---|
| header | Header | - | Header |
| d | float32 | $m$ | Estimated lateral offset |
| sigma_d | float32 | $m$ | Variance of lateral offset |
| phi | float32 | $rad$ | Estimated Heading error |
| sigma_phi | float32 | $rad$ | Variance of heading error |
| c | float32 | $1/m$ | Reference curvature |
| status | int32 | - | Status of Duckiebot 0 if normal, 1 if error is encountered |
| in_lane | bool | - | In lane status |

**Structure of the received messages with type *duckietown_msgs/ControlMessage***

The following table defines the structure of the control messages the Lane Controller Node receives from all the teams who want to send commands to our controller.

TABLE 9.9. STRUCTURE OF CONTROL MESSAGE TO BE USED.

| Field | Abstract Data Type | SI Units | Description |
|---|---|---|---|
| header | Header | - | Header |
| d_est | float32 | $m$ | Estimated lateral offset |
| d_ref | float32 | $m$ | Reference lateral offset |
| phi_est | float32 | $rad$ | Estimated Heading error |
| phi_ref | float32 | $rad$ | Reference heading |
| c_ref | float32 | $1/m$ | Reference curvature |
| v_ref | float32 | $m/s$ | Reference Velocity |

**Structure of the received messages with type _duckietown_msgs/ControlVelocity**

The following table defines the structure of the control messages that mostly the implicit coordination group will send us to control the velocity.

TABLE 9.10. STRUCTURE OF VELOCITY MESSAGE TO BE USED.

| Field | Abstract Data Type | SI Units | Description |
|---|---|---|---|
| header | Header | - | Header |
| v_ref | float32 | $m/s$ | Reference Velocity |

**Information to be provided from each team**

The following table defines the information we need from each team that uses the lane controller.

TABLE 9.11. INFORMATION NEEDED FROM EACH TEAM.

| Team | Information |
|---|---|
| Saviors | $d_{ref}$, $v_{ref}$ |
| Navigators | $d_{est}$, $d_{ref}$, $\theta_{est}$, $c_{ref}$, $v_{ref}$ |
| Parking team | $d_{est}$, $d_{ref}$, $\theta_{est}$, $c_{ref}$, $v_{ref}$ |
| Implicit Coordination | $v_{ref}$ |
| Fleet-level Planning | $d_{ref}$, $v_{ref}$ |

*Stop Line Filter Node*

In the following table, published topics are listed:

TABLE 9.12. PUBLISHED TOPICS BY STOP LINE FILTER NODE

| Topic | Max Latency |
|---|---|
| stop_line_reading | negligible |
| flag_at_stop_line | negligible |

In the following table, subscribed topics are listed:

TABLE 9.13. PUBLISHED TOPICS BY STOP LINE FILTER NODE

| Topic | Max Latency |
|---|---|
| segment_list | 25 ms |
| lane_pose | 15 ms |

## 9.2. Part 2: Demo and evaluation plan

### 1) Demo plan

The main goal is to demonstrate the improved curve driving. For this purpose, we will run two Duckiebots with different versions of Estimator and Controller running on the same test track, see picture. With the old lane following module, the Duckiebot corrected its position when it was not parallel to the white lines and the correction caused an overshoot. The new module allows the Duckiebot to detect an upcoming curve early and the controller will be adjusted to the curve. Our module also improves the execution of other tasks such as stopping at an intersection or in front of a Duckie. Further, the Duckiebot will drive with an offset of at most 2 cm. While running in an endless loop, we can also show that the steady state error has been minimized.

Figure 9.4. Possible map for lane following demo.

*What hardware components do you need?*
For our demo we want to build a small Duckietown test track, see picture. Thus we need about 21 tiles, DUCKIEtape and the usual Duckietown decoration.

## 2) Plan for formal performance evaluation

We will run all of the tests **5 times**.

- **Stopping in front of red line**: in the demo mode, we will let the Duckiebot drive to a red line and measure the distance between the center of the stop line to the wheel axis of the Duckiebot after it stopped.
- **Pose Estimation**: We will manually drive the Duckiebot along a pre-taped route in the Duckietown, of which the **d** is equal to zero, and collect multiple sets of data with the old pose estimator running and then with the new pose estimator running. From the bag data, we can analyse the estimation deviation from both estimators.
- **Offset minimization in straight lanes**: In the demo mode, we will let the Duckiebot drive down a straight lane. At the end of the straight lane, we will fix two laser pointers pointing to a wall and count how many times we can't see the light point on the wall while driving. In the case, a light dot disappears, the Duckiebot has left the target range. The distance between the laser pointers will be the width of a Duckiebot plus 4 cm.
- **Performance of the controller on curvy roads**: For curvy roads we will check the visual performance of the line following by counting how many times the Duckiebot touches a line on the S-curve section of the Zurich Duckietown. Additionally we want to compare the control motor commands in the curve section with the commands of the old controller and verify their smoothness.
- **Performance of the controller on lanes with dynamic width**: If we altered the controller to be more robust for non nominal appearance, we eventually check if the Duckiebot is robust to changes in lane specifications, such as narrower lanes or

different width of lane tapes. We will let the Duckiebot drive on modified tiles and check the performance of estimation and lane following.

## 9.3. Part 3: Data collection, annotation, and analysis

### 1) Collection

*How much data do you need?*
For every future step we need fixed logs and logs from driving. Baseline has been set and logs have been taken with the current implementation of the code to evaluate current performance.

*How are the logs to be taken? (Manually, autonomously, etc.)*

- **Manually**: Static logs with different values for $d_{act}$ and $\theta_{act}$ have been taken. These can be used for a unit test of the estimator.

| *Bot* | *Institution* | *Timestamp* | *Lane position* | *d: distance [cm]* | *phi: angle [deg]* |
|---|---|---|---|---|---|
| **yaf** | **ETHZ** | **2017-11-24-17-36-01** | **straight** | **0** | **0** |
| yaf | ETHZ | 2017-11-24-17-45-12 | straight | 1 | 0 |
| yaf | ETHZ | 2017-11-24-17-40-10 | straight | 5 | 0 |
| yaf | ETHZ | 2017-11-24-17-40-48 | straight | 10 | 0 |
| yaf | ETHZ | 2017-11-24-17-46-22 | straight | -1 | 0 |
| yaf | ETHZ | 2017-11-24-17-41-43 | straight | -5 | 0 |
| yaf | ETHZ | 2017-11-24-17-42-23 | straight | -10 | 0 |
| yaf | ETHZ | 2017-11-24-17-58-56 | straight | 0 | 5 |
| yaf | ETHZ | 2017-11-24-18-00-42 | straight | 0 | 10 |
| yaf | ETHZ | 2017-11-24-18-02-23 | straight | 0 | 30 |
| yaf | ETHZ | 2017-11-24-18-12-22 | straight | 0 | 60 |
| yaf | ETHZ | 2017-11-24-17-53-30 | straight | 0 | -5 |
| yaf | ETHZ | 2017-11-24-17-54-40 | straight | 0 | -10 |
| yaf | ETHZ | 2017-11-24-17-56-28 | straight | 0 | -30 |
| yaf | ETHZ | 2017-11-24-18-06-10 | straight | 5 | 10 |
| yaf | ETHZ | 2017-11-24-18-05-08 | straight | 5 | -10 |
| yaf | ETHZ | 2017-11-24-18-08-27 | straight | -5 | 10 |
| yaf | ETHZ | 2017-11-24-18-07-11 | straight | -5 | -10 |
| **yaf** | **ETHZ** | **2017-11-24-18-14-36** | **curve** | **0** | **0** |
| yaf | ETHZ | 2017-11-24-18-16-35 | curve | 1 | 0 |
| yaf | ETHZ | 2017-11-24-18-17-27 | curve | 5 | 0 |
| yaf | ETHZ | 2017-11-24-18-18-17 | curve | 10 | 0 |
| yaf | ETHZ | 2017-11-24-18-22-35 | curve | -1 | 0 |
| yaf | ETHZ | 2017-11-24-18-19-15 | curve | -5 | 0 |
| yaf | ETHZ | 2017-11-24-18-20-42 | curve | -10 | 0 |
| yaf | ETHZ | 2017-11-24-18-28-33 | curve | 0 | 5 |
| yaf | ETHZ | 2017-11-24-18-29-16 | curve | 0 | 10 |
| yaf | ETHZ | 2017-11-24-18-30-43 | curve | 0 | 30 |
| yaf | ETHZ | 2017-11-24-18-24-31 | curve | 0 | -5 |
| yaf | ETHZ | 2017-11-24-18-25-24 | curve | 0 | -10 |
| yaf | ETHZ | 2017-11-24-18-27-27 | curve | 0 | -30 |

Figure 9.5. Table of static logs taken to evaluate the estimator.

- **Autonomous**: Logs should also be taken during lane-following-demo to evaluate the estimator and control performance (see performance evaluation).

*Do you need extra help in collecting the data from the other teams?*
We do not need data from other teams and therefore do not need help.

### 2) Annotation

*Do you need to annotate the data?*

No, because we will receive the needed edges from the Anti-Instagram group.

*At this point, you should have tried using thehive.ai to do it. Did you?*
In autonomous driving thehive.ai is mostly used to annotate images in order to detect and recognize obstacles and for semantic segmentation. As our project does not rely on these information, we do not need it.

*Are you sure they can do the annotations that you want?*
Probably they could, but so do we with our estimator. There are no obstacles or Duckies to annotate.

### 3) Analysis

We don't need data annotation since we can do all the benchmarking by our own. We are not involved in any obstacle detection so we do not need any obstacles annotated.

*Do you need to write some software to analyze the annotations?*
No, because we do not use the annotations of thehive.ai.

*Are you planning for it?*
No

UNIT N-10

# The Controllers: final report

## 10.1. The final result

Figure 10.1. The Controllers Demo Video

See the operation manual to reproduce these results.

## 10.2. Mission and Scope

IMPERIUM ET POTESTAS EST

(With control comes power)

Our **Mission** was to make lane following more robust to model assumptions and Duckietown geometric specification violations and provide control for a different reference.

### 1) Motivation

In Duckietown, Duckiebots are cruising on the streets and also Duckies are sitting on the sidewalk waiting for a Duckiebot to pick them up. To ensure a baseline safety of the Duckiebots and the Duckies, we have to make sure the Duckiebots are able to follow the lane (or a path on intersections and in parking lots) and stop in front of red lines. For instance, the Duckiebot is driving on the right lane. It should never cross the centerline to avoid any collisions with an oncoming Duckiebot.

The overall goal of our project is to stay in the lane while driving and stopping in front of a red line. Due to the tight time plan, we focused on improving the existing code and benchmarking the tasks. In order to let the Duckiebot drive to a given point, the robot has to know where it is in the lane, calculate the error and define a control action to reach the target. To retrieve the location and orientation information, a pose estimator is implemented. The estimator receives line segments from the image pipeline with information about line tape colour (white, yellow, red) (Figure 10.2) and whether the segment is on the left or right edge of the line tape. Using those information, we determine if the Duckiebot is inside or outside

the lane, how far it is from the middle of the lane and at what angle it stands. The relative location to the middle of the lane and the orientation of the Duckiebot are passed on to the controller. In order to minimize the error, the controller calculates the desired velocity and heading of the Duckiebot using the inputs and controller parameters. The importance of our project in the framework "Duckietown" was obvious, as it contains the fundamental functionality of autonomous driving. Furthermore, many other projects relied on our project's functionality such as obstacle avoidance, intersection navigation or parking of a Duckiebot. We had to ensure that our part is robust and reliable.

Figure 10.2. Image with Line Segments, $d_{err}$ and $\phi_{err}$ displayed.

## 2) Existing solution

Figure 10.3. Pose of Duckiebot in a curve element.

From last year's project, the baseline implementation of a pose estimator and a controller were provided to us for further improvement. The prior pose estimator was designed to deliver the pose for a Duckiebot on straight lanes only. If the Duckiebot was in or before a curve and in the middle of the lane, the estimated pose showed an offset **d**, see definition of **d** in figure below. The existing controller

worked reasonably on straight lines. Although, due to the inputs from the pose estimator to the controller, the Duckiebot overshot in the curves and crossed the left/right line during or after the curve.

Figure 10.4. Old vs. new controller

### 3) Opportunity

In the **previous** implementation, the lane following was not guaranteed on curved lane segments, because the Duckiebot often left the lane while driving in the curve or after the curve. Although the Duckiebot sometimes returned correctly to the right lane after leaving it and continued following the lane, robust lane following was not provided. On straight lanes, the Duckiebot frequently drove with a large static offset from the center of the lane. The previously implemented pose estimator and controller left room for improvement.

Further, the previous lane controller was not benchmarked for robustness nor for performance, therefore we defined various tests to benchmark the previous controller and our updated solution. During the project, we continuously tested our code with the entire lane following pipeline for best practice and compared our implemented solution to the existing one to record the improvement.

Our **Scope** was first of all to enable controlled autonomous driving of the Duckiebot on straight lane segments and curved lane segments which are in compliance with the geometry defined in Duckietown Appearance Specifications. Further, we wanted to enhance the robustness to arbitrary geometry of lane width or curvature of the lane to ensure the autonomous driving of the Duckiebot in an individual Duckietown setup. We also tackled the detection and stopping at red (stop) lines. With the previous implementation, the Duckiebot stopped rather at random points in front of the red line. We wanted to improve the implementation, to ensure a stop in the middle of the lane, in a predefined range and at a straight angle to the red line. As the Duckietown framework is a complex system involving various functionalities such as obstacle avoidance and intersection navigation, our lane following pipeline provides the basic function for those functionalities and it has to be able to interact with the modules of other teams. Hence, it was also our duty to design an interface which can receive and apply information from other modules. For example, our controller can take reference **d** from obstacle avoidance, intersection crossing and parking. For intersection navigation and parking, our controller needs additionally the pose estimation and a curvature from the navigators and the parking team respectively.

Out of scope was:

- Pose estimation and curvature on Intersections (plus navigation / coordination)

- Model of Duckiebot and uncertainty quantification of parameters (System Identification)
- Object avoidance involving going to the left lane
- Extraction and classification of edges from images (anti-instagram)
- Any hardware design
- Controller for Custom maneuvers (e.g. Parking, Special intersection control)
- Robustness to non existing line

4) Preliminaries (optional)

## 10.3. Definition of the problem

Our final objective is to keep the Duckiebots at a given distance **d** from the center of the lane, on straight and curved roads, under bounded variations of the city geometric specifications.

The project was on the bottom line, taking the line segments which gave information about the line colour and the segment positions to estimate the Duckiebot's pose and return a command for the motors to steer the robot to the center of the lane. After roughly analysing the existing solution, we divided the work load into two topics **pose estimation** and **controller** to enable parallel dealing with the problems in the short period of time.

In our Preliminary Design Document and Intermediate Report, we have listed all variables and their definitions, as well as all system interfaces with other groups and assumptions we made. Due to limitation of time and different priorities of other teams, some integrations with other teams are not yet activated but they are already prepared in our code (some of it commented out).

1) Pose Estimation

Starting with the image taken by a monocular camera, we assume that Anti-Instagram + Ref. error

I do not know the link that is indicated by the link '#anti-instagram-final-report'.

compensates for color changes from different ambient light conditions and the detected segments of the line edges are always in the corresponding colour (yellow, white, red) to the tape and point to the correct direction (to the Duckiebot = left edge, away from the Duckiebot = right edge), see (Figure 10.2). The detected line segments are passed on in a list to the 'Lane Filter', the Duckiebot estimates the distance $d_{est}$ from center of the lane and heading $\theta_{est}$ with respect to the center of the lane.

To improve curve following, our main goal was to decrease the overshoot after a turn. A main reason for the overshoot was that the previous implementation of the pose estimator was designed for straight lanes only. We planned to predict if a curve is upcoming and in which direction the turn will be. This curvature estimation could then be used as an input for a feedforward part of the controller. This would help for a smoother transition in curves.

We planned on testing two different ideas for curvature estimation. One based on the distribution of segments to different domains with different ranges. The second one based on the Discrete Fourier Transform (DFT).

Our general goal was to improve the accuracy of estimated pose in regard to ref-

erence pose and increase the computational efficiency. Also robustness regarding slight changes in width of the lane, width of the lines and curvature should be achieved.

*How the pose estimation works:*

Figure 10.5. Vote generation on straight lane from one line segment.

The lane filter gets a list of detected segments by the line filter with their colors. One segment is described by two vectors pointing from the center of the Duckiebot to the start and endpoint of the segment as shown in Figure 10.5 where one segment is described by the vectors $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$. From these two points we can calculate the vector $\boldsymbol{t}$ which is tangential to the segment and the vector $\boldsymbol{n}$ which is perpendicular to the segment. The vector $\boldsymbol{t}$ can be calculated using

$$\boldsymbol{t} = \frac{\boldsymbol{p_2} - \boldsymbol{p_1}}{||\boldsymbol{p_2} - \boldsymbol{p_1}||}$$

and $\boldsymbol{n}$ is the unit vector perpendicular to $\boldsymbol{t}$. In case the segment was perfectly detected, the distance of the center of the Duckiebot to the white line is then the scalar product of $\boldsymbol{p_1}$ with $\boldsymbol{n}$. By using the width of the lane we can calculate the distance from the center of the lane.

To get the angle $\phi$ we use the fact that the tangential vector $\boldsymbol{t}$ is scaled to length one. From the geometry in Figure 10.6 we see that $\phi$ can be calculated as

$$\phi = \arcsin\left(t_2\right)$$

Figure 10.6. Geometry for getting the angle $\phi$.

This gives one vote consisting of the two coordinates shown in Figure 10.5 where $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ are representing the same coordinate. For every detected segment one of

those votes can be calculated. The one pose having the most votes will be selected as our estimated pose.

*Vote generation on a curve:*

Figure 10.7. Vote generation on curved lane from one line segment.

On a curved lane the generation of votes differs from the one on a straight lane. We derived the vote generation on a curved lane, but did not implement it since a working curvature estimation is needed for this. Again we generate a vote for one given segment with the two endpoints $p_1$ and $p_2$ and additionally we need to know the radius $r$ of this segment. The geometry is shown in Figure 10.7. We again calculate the tangential vector $t$ and normal vector $n$ as shown before. We get the center point $p$ of the segment using

$$p = \frac{p_1 + p_2}{2}$$

Now we can get the two lengths $a$ and $b$ by taking the scalar product of $p$ with $n$ and $p$ with $t$ respectively. Using this we can find the length of $r_1$ by

$$||r_1|| = \sqrt{(||r|| - a)^2 + (b)^2}$$

From this we can get the length of $d$ using

$$d = ||r_1|| - \left(r - \frac{lanewidth}{2}\right)$$

In the end, $\phi$ can be obtained using $d$ and the $x$ coordinate vector of the Duckiebot coordinate system. This gives us the **pose** of the Duckiebot in a curve.

### 2) Controller

In the existing implementation, the controller has taken the output of the estimator as input and calculated the motor command, velocity **v** and angular velocity $\omega$ , with help of hardcoded parameters. Running the current lane following demo, we determined the weaknesses of the controller's performance. Since the existing controller only had a proportional part (P-part) and the $\theta$ acted similarly to a derivative part, (D-part), we decided to implement following parts and benchmark its

performance.

- Integrator for both $\boldsymbol{\theta}$ and $\mathbf{d}$
  - First approach: No saturation for integrator
    - Problem: Very strong oscillation
  - Second approach: Saturation for integrator
    - Performance improved a lot. Oscillation is reduced
  - Third approach: Saturation of integrator and reset of integrator whenever zero error is reached
    - Performance improved again.
  - Fourth approach: Added true integration with correct timestep instead of simply summing up the error and neglecting the timestep
    - This is the correct approach since the time step needs to be taken into account.
- Feedforward for driving on a curved lane
  - We take the current curvature as an Input for the feedforward part
    - The feedforward part is very much dependent on the correct curvature estimation. If curvature was estimated correctly, feedforward worked well.

In some cases, the controller won't need the real time pose input of the pose estimator but a given path from other teams for example during intersection navigation. For this task, we had to define a more general 'Lane Pose' message, communicate and coordinate the integration with other teams and subscribe to the topic of their nodes (see Subscribed topics by Lane Controller Node). To decide about the input source, we have to subscribe to flags passed by the other teams and create a prioritization logic.

Regarding performance, the Duckiebot should have a small steady state error within a tile and never leave the lane given the Duckietown appearance specifications.

## 10.4. Contribution / Added functionality

### 1) Curvature Estimation

*Curvature estimation using multiple domains:*

The idea of our curvature estimation approach is to split the domain in front of the Duckiebot into multiple range areas. A version using three areas is shown in Figure 10.8.

Figure 10.8. Curvature estimation using the segments in different domains.

In each of the range areas the road can roughly be assumed as a straight lane. But for every area further away from the bot this straight lane fit is tilted more towards the left. For each area, only those segments with center point inside the area are considered. Using these segments for each area, we run the standard estimation and thereafter for each area we get a **d** and a $\boldsymbol{\theta}$ value. Now we can compare those results with the **d** and $\boldsymbol{\theta}$ value from the estimation of the position.

The expected results are shown in Figure 10.9 where the left most points in each graph represent the actual position estimation and the further three point represent the estimations of the three different range areas. As a leftover of the existing code where this was already the case, $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ are still used as synonyms within the code. Due to limited time, it did not make it to our highest priority at any time within our project, to merge those names. This should be done in future work.

Figure 10.9. Expected results of d and $\phi$ values for straight lane, left curve and right curve.

Unfortunately, the measurements for the higher ranges are very noisy as there are only a few line segments detected at further distance and therefore the signal to noise ratio is very bad. To get rid of outliers, we decided to add a median filter over values of the ranges. Additionally we saved this median **d** and $\boldsymbol{\phi}$ values over time for the last five time steps and again took the median value of it. Then we checked

if it is above or below the value of the closest range.

Another possibility would be to fit a line through the data points and decide on the lane type based on the slope of the line. Nevertheless, we decided to use the before mentioned method using the median values because we wanted to keep the **computational complexity** as low as possible.

By testing we found good results for the cutoffs shown in figure where $d_{median}$ and $\phi_{median}$ represent the median over 5 time steps of the values resulting from the range area and $d_{est}$ and $\phi_{est}$ represent the actual pose estimate.

TABLE 10.1. CUT OFFS FOR THE DECISION ON THE CURVATURE TYPE.

| | $d_{median} - d_{est}$ | $\phi_{median} - \phi_{est}$ |
|---|---|---|
| left curve | $< -0.3$ | $> 0.05$ |
| right curve | $> 0.2$ | $< -0.02$ |

*Curvature estimation using Discrete Fourier Transform:*

By generating a discrete binary image from the segments projected to the ground and applying the discrete fourier transform to this image, the curvature of the road in front of the Duckiebot can be detected. Fourier transforms of such binary images are shown in Figure 10.10. By using the correct fourier features, straight lanes, right and left curves could be detected due to to their differing fourier transform.

This method has been implemented successfully but the problem was first of all choosing the right resolution for the segment images and additionally, the method introduced a delay of about 0.2 seconds. Since we want to avoid decreasing the lane following performance of the Duckiebot, we decided to dismiss this method.

Figure 10.10. Discrete Fourier Transform (DFT) of a street image in ground frame (credits jukindle).

## 2) Controller

In the controller, a feedforward part was added to figuratively speaking straighten the lane and ease the work of the controller. Therefore, the feedforward part takes the reference curvature $c_{ref}$ and reference velocity $v_{ref}$ as inputs and returns the needed yaw rate $\omega$, which is then added to the output of the controller. The block diagram of the control loop is shown in Figure 10.11. Since the kinematic calibration was not yet yielding the demanded values of $v_{ref}$ and $\omega$ in $[m/s]$ and $[\text{rad}/s]$, correction factors **velocity_to_m_per_s** and **omega_to_rad_per_s** were introduced.

With the new kinematic calibration, those correction factors need to be adjusted or ideally become obsolete and need to be deleted in future work.

Figure 10.11. Block diagram including feedforward part (FF)

The feedforward part also enables the controller to work for other applications than just lane following. It can follow a path, for example on intersections and parking lots, if the information (including localization) is given in the format described in our Intermediate Report. The respective code is written but some of it is commented out respectively not activated yet due to the limited time of the project and the coordination between teams.

From the coordinates as shown in Figure 10.5, we get

$$\begin{bmatrix} \dot{d} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v \times \sin\theta \\ \omega \end{bmatrix}$$

Through linearization, assuming $\theta$ to stay small and with $u = \omega$, this becomes

$$\begin{bmatrix} \dot{d} \\ \dot{\theta} \end{bmatrix} = \begin{pmatrix} 0 & v \\ 0 & 0 \end{pmatrix} \times \begin{bmatrix} d \\ \theta \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \times u$$

with

$$x = \begin{bmatrix} d \\ \theta \end{bmatrix}$$

To reduce static offset, integral parts were implemented for both $d$ and $\theta$. This was achieved by augmenting the system to $\hat{x}$, as shown below.

$$\hat{x} = \begin{bmatrix} x \\ e_I \end{bmatrix}$$

$$\begin{bmatrix} \dot{x} \\ \dot{e_I} \end{bmatrix} = \begin{pmatrix} 0 & v & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix} \times \begin{bmatrix} x \\ e_I \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \times u + \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \times x_{ref}$$

With

$$\begin{bmatrix} e \\ e_I \end{bmatrix} = \begin{bmatrix} x_{ref} - x \\ \int (x_{ref} - x) dt \end{bmatrix}$$

$$x_{ref} = \begin{bmatrix} d_{ref} \\ \theta_{ref} \end{bmatrix}$$

In order to omit oscillation and guarantee the stability of the system, the poles were placed on the negative real axis. With that we found the initial values for $k_p$ and $k_I$ in

$$u = -\begin{bmatrix} k_p & k_I \end{bmatrix} \times \begin{bmatrix} e \\ e_I \end{bmatrix}$$

through

$$\begin{bmatrix} \dot{d} \\ \dot{\theta} \end{bmatrix} = \begin{pmatrix} 0 & v \\ -k_p & -k_I \end{pmatrix} \times \begin{bmatrix} d \\ \theta \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \times x_{ref}$$

The controllability matrix shows that the integrator of $\theta$ is not controllable, since it has rank 3 instead of 4:

$$C = \begin{bmatrix} B & AB & A^2B & A^3B \end{bmatrix} = \begin{pmatrix} 0 & v & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -v & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix}$$

To prevent the integral parts from diverging, an **Anti Reset Windup** was implemented. Therefore, whenever actuator limits were reached, the integral steps at the corresponding time step were not added to the integrator. The actuator limits were reached when the motors were sent lower values than would be necessary to reach the controller outputs, because of certain limitations within the software. The limitations include for example a limitation on the turn radius of the Duckiebot, because it should not be able to turn on the spot but to move more similarly to common passenger cars.

In curves, the integrator values accumulate rapidly and lead to an overshoot after the curve. A possible approach would be to turn off the integrator in curves, but in consequence the curvature estimation would need to be used and in addition need to be robust. Or if the feedforward part could be fully used (while also needing a robust and low-latency curvature estimation), the problem might be diminished. In the current state, the integrator is reset to zero whenever it is at or crosses the zone of zero error. In addition the integrator was also reset to zero, whenever the velocity sent to the motors was zero.

Since the integral part of theta is not controllable, it was set to zero. The resulting parameter, the proportional gains of both $d$ and $\theta$ plus the integrator gain of $d$, were tuned. First with pole placement initial values were approximated, as described above. For the final tuning, each parameter was varied until the unstable state and the approximate boundary to the stable state were found, while all the other parameters were kept in a stable state. This was repeated multiple times with ever more aggressive controller behavior until an optimum was found. The controller is optimized to run with a gain (of the kinematic calibration) of 0.6.

### 3) Benchmark

To benchmark the state zero at the beginning of the project and our final implementation and to compare them, we implemented a benchmark package. This package contains the benchmark code used for the Controllers project. It basically

takes one or more rosbags in a specific folder and evaluates the run of the corresponding Duckiebot for $d_{ref}$ and $\phi_{ref}$ and plots them into a diagram.

Additionally if the rosbag does not contain any pose information, it takes the pictures and calculates the transformation and line segments itself. It also plots the values onto the pictures, so those pictures can be combined to a video.

*Output:*

The output diagram should look like the one shown in Figure 10.12.

Figure 10.12. Output diagram of benchmark code.

Output data should look like the one shown in Figure 10.13.

```
Benchmark results for ducktaped_ETHZ_2018-01-12-14-48-09
        dist min:       -0.145
        dist max:       0.165
        dist mean:      -0.014
        dist med:       -0.005
        dist var:       0.0014
        dist std:       0.0381

        phi min:        -1.45
        phi max:        1.25
        phi mean:       0.117
        phi med:        0.05
        phi var:        0.1516
        phi std:        0.3894

Image segments statistics:      Average amount of segments:     33.946
        Median amount of segments:      34.0
        Average processing time per frame (on duckiebot):       0.073

Rosbag processing time: 3.983850
        Average processing time per frame: 0.002665
        Image preparer used on computer: prep_200_70
```

Figure 10.13. Output data of benchmark code.

An example of a processed frame is shown in Figure 10.2. To run the benchmark code see the **README** file in duckietown/Software/catkin_ws/src/10-lane-control/benchmark.

### 4) Logs

We took a huge amount of logs to benchmark the performance of the controller and estimator. These logs are available here. Our Duckiebots were a313, yaf, fobot, ducktaped and tori.

## 10.5. Formal performance evaluation / Results

We evaluated the improvement of the performance with help of several tests. The evaluation procedure are defined in our Intermediate Report. The main benchmark feature was the average deviation from tracking reference during a run (distance to middle lane) and the standard deviation of the same value. We also benchmarked the deviation from the heading angle as well but since the bot is mainly controlled according to the deviation of the tracking distance, it was the main feature to lead our development. Benchmarking in general occurred by letting the Duckiebot run a specific experiment and recording a rosbag. We wrote a distinct offline benchmarking application mentioned above, that analyzes the rosbag containing the recorded values and creates plots with the extracted information about tracking distance and heading angle over the run.

Furthermore, we assessed the performance of the Duckiebots in the following dimensions:

- **Estimator**:
  - Static lane pose estimation benchmark
  - Static curve pose estimation benchmark
  - Image resolution benchmark
  - Segment interpolation benchmark
  - Curvature estimation benchmark
- **Controller**:
  - Stop at red line benchmark
  - Controller benchmark
  - Non-conforming curve benchmark

### 1) Performance Evaluation of Estimator

*Static lane pose estimation benchmark:*

In the static lane pose estimation, we put the Duckiebot on predefined poses and checked how well the pose estimator performs. In this section, the Duckiebot was placed on a straight lane segment with different measured distances from the middle of the lane and different measured heading angles. The results can be seen in the following graphs:

Figure 10.14. Old Lane Estimator Performance of estimating $d$ on straight lane.

As one can see from Figure 10.14, the estimated distance from the middle of the

lane and the actual value correspond very good in most experiments. There is only one case where the actual deviation was $-1cm$ and measured was $4cm$. Note that the histogram resolution used to determine the pose is $1cm$.

Figure 10.15. Old Lane Estimator Performance of estimating $\phi$ on straight lane.

Figure 10.15 shows a similar picture for the heading angle estimation from the segments. Deviation from actual values vary from $1$ to $12°$, whereas the Duckiebot performed better when being rotated to the left. Note that the histogram resolution to determine the heading angle is $3°$ or $0.15rad$.

Figure 10.16. Differences of $d_{actual}$ and $d_{measured}$ and $\phi_{actual}$ and $\phi_{measured}$ respectively.

If we look at the overall deviations in all experiments shown in Figure 10.16, we can see that the pose estimator performs fairly well, and it is possible to control on the deviation of the distance. The heading angle shows more error. The average deviation from the actual tracking distance in all experiments accounts to $1.6cm$ and the average deviation from the actual heading angle in all experiments is $7.2°$.

*Static curve pose estimation benchmark:*

In the static curve pose estimation, we put the Duckiebot on predefined poses and checked how well the pose estimator performs. In this section the Duckiebot was

placed on left curve with different distances from the middle of the lane and different heading angles. The results can be seen in the following graphs:

Figure 10.17. Old Lane Estimator Performance of estimating $d$ on curved lane.

As one can see from Figure 10.17, the estimated distance from the middle lane and the actual value correspond partially to the actual values. Especially for the distance of $\mathbf{10}\boldsymbol{cm}$ to the right of the middle of the lane in a left curve the estimator has problems to detect the correct deviation. This is due to the low number of segments and the fact that the pose estimator is actually only constructed to estimate the pose on a straight lane. Also, there is quite some noise which leads to wrong interpretation of the distance, even when the Duckiebot is perfectly situated in the middle of the lane. For some experiments there is no pose estimation due to too much noise in the segment list. Note that the histogram resolution used to determine the pose is $\mathbf{1}\boldsymbol{cm}$.

Figure 10.18. Old Lane Estimator Performance of estimating $\phi$ on a curved lane.

Whereas the estimator is still able to estimate $\boldsymbol{d}$ quite well on a left curve, for the heading angle most of the values are completely off as can be seen in Figure 10.18. This means the heading angle prediction is not reliable on curved lanes. Note that the histogram resolution to determine the heading angle is $\mathbf{3}^\circ$ or $\mathbf{0.15}\boldsymbol{rad}$.

Figure 10.19. Differences of $d_{actual}$ and $d_{measured}$ and $\phi_{actual}$ and $\phi_{measured}$ respectively on a curved lane.

If we look at the overall deviations in all experiments shown in Figure 10.19, we can see that the pose estimator performs ok in the determination of the distance from the middle of the lane in a curved section. The values from the heading angle are unlikely correct and therefore should not be used as control input. The average deviation from the actual tracking distance in all experiments accounts to $\mathbf{1.8cm}$ and the average deviation from the actual heading angle in all experiments is $\mathbf{21.6°}$ .

*Image resolution benchmark:*

Since the image resolution has an impact on the number of segments being visible to the Duckiebot and the image processing latency time, we benchmarked the impact on the entire lane following performance. We tested different image resolutions, top cut off amounts and changed the histogram size to evaluate how it influences the control performance.

Figure 10.20. Measured $d_{mean}$ and $\phi_{mean}$ values for different image resolutions for Duckiebot 'yaf'.

As one can see in Figure 10.20, the performance of the Duckiebot measured as the mean deviation from the reference trajectory (which is usually $\mathbf{0cm}$) is get-

ting worse the higher the resolution. There are outliers though, since the highest resolution being tested shows better performance than the resolution just one step smaller. The best performance is achieved with slightly higher resolution at 150x200 pixels. To validate these results, we tested it on another Duckiebot as well.

Figure 10.21. Measured $d_{mean}$ and $\phi_{mean}$ values for different image resolutions for Duckiebot 'a313'.

We see that the results shown in Figure 10.20 and Figure 10.21 are not congruent. We think that this has to do with the fact that each Duckiebot is slightly different and also has different latencies.

The worse performance for higher resolutions can be explained with the change in processing time of the images. Although there are more line segments, which means more precise information about our pose, the processing time increases, and thus this adds latency and affects the whole system performance. The Duckiebot reacts slower to offsets of its pose. Figure 10.22 shows the segment processing time and number of segments for different image resolutions.

Figure 10.22. Segment processing time and median of number of segments for different image resolutions.

Increasing the top cutoff value means, that from the input image more of the top

part is cut away to reduce visual clutter from the image background. At the same time this also decrease the number of pixels being processed and thus lowers the mean latency as well.

Figure 10.23. Performance of 'a313' for different top cutoffs in pixels.

We run a benchmark to evaluate the influence of the top cutoff on the performance. The test was performed with an image resolution of 120x160 pixels. The results are shown in Figure 10.23. 40 pixels is the standard top cutoff values. This means the upper 40 pixels are cut away from each image. While increasing the top cutoff amount, the $d_{mean}$ decreases slightly while $\phi_{mean}$ increases slightly. We don't see big changes in performance until the top cutoff gets quite big. At this point the Duckiebot does not see enough to control according to the actual pose situation.

Figure 10.24. Segment process time and number of segments for different top cutoffs in pixels.

As we can see in Figure 10.24, the segment process time and therefore the latency decreases proportionally to the number of segments. This graph also explains the reduced performance in Figure 10.23 since with under 25 segments it is hard to get an accurate pose estimation. In this case a higher top cutoff lowers the performance and at the same time the latency. So, we might see an increase in performance if we combine higher top cutoff with higher resolution, since there the increased latency was an issue.

Figure 10.25. Performance for different resolutions of $d$ in histogram in $cm$.

We also tested the influence of the histogram size for the generation of the votes. The results are shown in Figure 10.25. Making the vote histogram cell size smaller increases the accuracy of the pose estimation. At the same time more segments are needed to get a precise estimate and reduce the influence of noise. We see that the performance is going down for a higher histogram resolution. At the same time Figure 10.26 shows that the segment processing time stays more or less constant for different histogram resolutions. This actually shows, that the decrease in performance results from the missing of a distinct pose for higher resolutions.

Figure 10.26. Segment processing time and number of segments for different resolutions of $d$ in histogram in $cm$.

j

TABLE 10.2. COMBINING IMAGE RESOLUTION AND TOP CUTOFF.

| Resolution | Top Cutoff | $t_{latency}[s]$ | $d_{mean}[cm]$ | $\phi_{mean}[rad]$ | $n_{segments}[-]$ |
|---|---|---|---|---|---|
| 120x160 | 40 | **0.019** | **−1.802** | **−0.025** | **39** |
| 150x200 | 75 | **0.012** | **−0.011** | **−0.004** | **23** |

As we can see from Table 10.2, the configuration with resolution 150x200 and top cutoff 75 can improve the lane control performance compared to the standard configuration with resolution 120x160 and top cutoff 40 without changing the lane controller itself or the pose estimator. Note that all the results from this section

have been tested with the improved lane controller.

*Segment interpolation benchmark:*

Another approach to improve the pose estimator is to increase the amount of line segments without increasing the image resolution. Here we take each line segment and divide it into smaller pieces of which each has a vote on the belief image. Good line segments cast more votes to the same pose estimate, while bad segments (e.g. which are further away or outliers) have less weight on casting wrong results. Think of it as a filter to improve quality of the lane pose estimate.

Figure 10.27. Performance for different segment interpolations.

As we can see in Figure 10.27, we tested up to interpolating a line segment 5 times. There aren't any significant changes to the lane following performance except for one outlier while interpolating with 3 line segments. If we look closer, we can see that the actual performance gets worse the more we interpolate due to processing speed of the raspberry pi.

Figure 10.28. Standard deviation of $d$ and mean latency for different segment interpolations.

Figure 10.28 shows the standard deviation of $\boldsymbol{d}$ ($\boldsymbol{d_{stdev}}$) and how it increases the more we interpolate due to higher latency. This behavior is observable on straight lanes where the Duckiebot oscillates around the reference trajectory. We can see

this in Figure 10.29 for a run with 5 interpolated segments per detected segments. From 50 to 70 sec we can observe the oscillations on a straight lane due to high latencies.

Figure 10.29. Benchmark graph for a run with 'a313' and 5 interpolated segments per detected segment.

*Curvature estimation benchmark:*

In this section, we want to evaluate the curvature estimation performance. What the curvature estimator basically does is dividing the input image into several circular sections with equi-radial distance to the Duckiebot. From each section it derives the pose and evaluates, how it changes in these sections. This will tell us, how the road in front of the Duckiebot looks like. Then again, this feature has an impact on the lane following performance of the Duckiebot since the processing power of the raspberry pi is limited and any added latency will slow the bot down.

Figure 10.30. Performance for different numbers of belief images from 1 to 7 where the first image is for the actual pose estimation and the further ones are for the curvature estimation (curvature resolution).

In Figure 10.30 on the horizontal we can see the number of belief images being evaluated (curvature estimation resolution). The higher the number, the better we can forecast the type of the road (left curve, right curve, straight lane). With a number of 1, there is no curvature estimation (basically the old pose estimator). We can

see that the performance compared to the old pose estimator is much better. This is because the reference run with 1 belief image has been recorded before and the calibration may have changed. Anyway, we can see a decrease in lane following performance, the higher the amount of belief images or image sections are created. This is due to higher cpu cost and increased latency. From tests we can see that a number of 4 belief images is sufficient to tell in most cases, at what kind of road type we are looking at.

Figure 10.31. Standard deviation of $d$ and $\phi$ for different numbers of belief images.

A look at Figure 10.31 showing the standard deviation tells us, that performance decreases with higher numbers of belief images (curvature estimation resolution).

Figure 10.32. Segment processing time and number of segments for different numbers of belief images.

Same as in other sections, the main performance is heavily depending on the overall latency of the code being executed on the Duckiebot. The latency of segment processing is shown in Figure 10.32

We improved the code on curvature estimation and retook all tests to better compare how the Duckiebot behaves. In the following we will see similar graphs with 1 belief image on the old pose estimator, 4 and 7 belief images on the new curva-

ture estimator and again 1, 4 and 7 belief images on the improved estimator.

Figure 10.33. Performance of 1, 4 and 7 belief images for old curvature estimation on the left 1, 4 and 7 belief images for improved curvature estimation on the right.

It is observable in Figure 10.33 that the improved curvature estimation performs slightly better in all three cases. In Figure 10.34 we see that the latency for the improved curvature estimator is lower and therefore the case with just one belief image (meaning the curvature estimation is turned off) performs especially well.

Figure 10.34. Segment processing time and number of segments of 1, 4 and 7 belief images for old curvature estimation on the left 1, 4 and 7 belief images for improved curvature estimation on the right.

## 2) Performance Evaluation of Controller

*Stopping in front of red stop line:*

We evaluated if the Duckiebot is able to stop in front of the red stop line within the defined specifications. In order to test the stopping behavior, we tested the old controller and the new controller and measured the pose in front of the stop line. The results in Table 10.3 show that we are able to improve the stopping in front of the red line. The performance shows to be in the bound of the target values. The

target stopping distance to the center of the red line should be 16 to 10 cm and the final heading angle should be in the range of $\phi = -10°$ to $\phi = 10°$.

TABLE 10.3. STOPPING AT STOP LINE EVALUATION.

| | $d_{mean}$ | $\phi_{mean}$ | Mean stopping distance to center of red line |
|---|---|---|---|
| Old Controller | $5.6cm$ | $5°$ | $17.4cm$ |
| New Controller | $-0.6cm$ | $3.6°$ | $8.2cm$ |

*Controller benchmark:*

The performance of the controller has been benchmarked under varying configurations, i.e. with the old baseline controller, the improved controller with the implemented Integrator and finally the same improved controller with addition of a correction for the static offset. The results of this benchmarking are shown in Table 10.4. Notably the controller did not use the improved estimator for this benchmark, rather the baseline estimator was used. The desired state throughout the benchmark is $d = 0.0$ and $\phi = 0.0$.

TABLE 10.4. RESULTS FOR CONTROLLER EVALUATION OF OLD CONTROLLER, NEW INTEGRAL PART AND OFFSET CORRECTION.

| | $d_{mean}[cm]$ | $d_{std}[cm]$ | $\phi_{mean}[rad]$ | $\phi_{std}[rad]$ |
|---|---|---|---|---|
| Old Controller | 3.16 | 0.45 | -0.40 | 0.1 |
| New Integral Part | -2.08 | 0.08 | -0.11 | 0.07 |
| Offset Correction | -0.45 | 0.16 | -0.03 | 0.20 |

As observable in Table 10.4, the lane following performance increased drastically after improving the controller. First, by implementing the Integrator into the controller, the performance improved in terms of a lower static offset as well as a lower mean heading angle. Additionally, the standard deviation of both d and $\phi$ was lowered considerably. This means that the Duckiebot stayed much closer to the desired position in the center of the lane, even after a long time. Therefore, the performance improved greatly with help of the Integrator alone already.

Further, by correcting the remaining static offset, the static offset was cancelled out completely and the median heading angle was lowered as well. This is a very important result, as the static offset represented a vital problem.

In addition to the quantitative benchmarking above, the performance was evaluated qualitatively as well by observing the driving Duckiebot. From those observations, the performance improvements in terms of a cancelled static offset as well as a much lower median heading angle were very clearly noticed as well. By directly comparing the performance of the old and new controller qualitatively, the improvement of the controller is very clearly visible. With the new controller the Duckiebot never touches the middle and outer lines, drives very robust, there is no static offset and no overshoot after the curves is observed.

*Non-conforming curve benchmark:*

Figure 10.35. Non conforming big curve.

We benchmarked the controller not only for the straight lanes and curves which are conforming with the Duckietown specification, rather the new improved controller was also tested on lanes with non-conforming geometries such as a very large and wide curve as shown in Figure 10.35 and a very narrow and harsh S-curve as shown in Figure 10.4. This benchmark was conducted in order to test the robustness of the controller to varying lane geometries. This is a very relevant test, as the geometry of the duckietown can not always be guaranteed. In addition, a controller which works good for a wide range of geometries would be desired. The results of those tests with non conforming curve geometries can be found in Figure 10.36.

Figure 10.36. Results from benchmark on non-conforming curves.

As can be seen from the results in Figure 10.36 for both tested non conforming curves the performance improved considerably by introducing the new controller. Both the mean distance to the center of the lane $\mathbf{d}$ and the mean heading angle $\phi$ have been improved. In addition, the standard deviation of both of those metrics were reduced as well. Those results show that the performance of the new controller was improved with respect to non conforming curve geometries. Since only two non conforming geometries have been tested, this test represents far from all non conforming geometries. In future benchmarks with respect to geometry robustness, this fact should be considered and more non conforming geometries should be tested. Nevertheless, the performance on those two tested non conforming curves are very promising and point to a strong robustness with respect to altering geometries.

*Performance of the controller on curvy road:*

Figure 10.37. Benchmark of Duckiebot on curvy roads with the baseline controller from last year.

Benchmark of Duckiebot with baseline controller from last year's implementation. Most notably, the median lateral position of the Duckiebot $d_{est_median}$ is higher compared to the new implementation of the controller.

Figure 10.38. Benchmark of Duckiebot on curvy roads with the new improved controller.

Benchmark of Duckiebot with new controller implementation. Most notably, the median lateral position of the Duckiebot $d_{est_median}$ is lower compared to the old implementation of the controller.

*Performance of the controller on non conforming lanes:*

We also made some test to show that our controller is able to cope with situations that are not conforming with the Duckietown specifications. Figure 10.39 shows a run with thicker white and yellow lines then specified and Figure 10.40 shows a run with some white and yellow lines missing. In both videos the new controller is still able to follow the lane as expected.

Figure 10.39. Lane following with thicker lines

Figure 10.40. Lane following with partial lines missing

### 3) Failed Implementations

**Estimator**

Although the 7 ranges estimation provided low mean deviation from the actual position and provided good prediction of the upcoming curve as well as its curvature direction. The 7 ranges estimation failed in the implementation of the lane following demo due to high computation requirement and the caused time latency.

**Controller**

Feedforward during lane following: As the feedforward part during lane following depends entirely on the estimation of the curve, this part failed due to bad estimation of the curves in certain situations as well as the increased latency due to the curvature estimation. Whenever a curve is not correctly detected or not precisely at the beginning of the curve, the feedforward part introduces additional instability. This is especially a problem in the notorious S curves. Therefore, the implementation of the feedforward works good if a precises estimation of the curve is available that works without introducing high latencies. Although, such a precise curvature estimation with low latency is not available at the moment. Hence, the feedforward part during lane following is not robust enough for the current curvature estimation. Nevertheless, the feedforward part is useful for other nodes to interact with the controller. In certain situations other nodes are able to use the feedforward part in order to follow paths (navigators on intersections and parking team on parking lots).

### 4) Challenges

- Limited computational power of Raspberry Pi.
  - Estimation of curvature introduced high latencies.
  - By increasing the resolution of the picture we would get more segments and this would make both a better pose and curvature estimation possible. Nevertheless, the latency is also increased significantly.
- Duckiebot with different wheels (slippery and non-slippery wheels)
- Some Duckiebots prove to have higher latency when running the same code. This increased latency is a problem.
- Lightning has a very big influence on the performance
  - Depending on the light condition of duckietown the number of detected segments as well as the correctness of the color is varying. Especially reflection on yellow tape makes it appear white. To tackle this issue, a polarisation filter was found to have a positive influence. This might need to be considered in future hardware updates.
  - Anti instagram might not be as good for every light condition.

*Effect of anti-instagram on segment detection in curves:*

In case Anti-Instagram is badly calibrated, the Duckiebot will not see enough line segments. This is especially a problem in the curve and the Duckiebot could leave the lane. An example of this failure can be seen in Figure 10.41 for which we had a bad Anti-Instagram calibration. Hence, the Duckiebot sees not enough line segments and the lane following fails in the curve. To solve the problem Anti-Instagram needs to be relaunched. In the last part of the video above the **X** button on the joystick is pressed and the Anti-Instagram node gets relaunched. We can see in the last part of the video RVIZ that the number of detected line segments gets increased dramatically after the recalibration.

Figure 10.41. Anit Instagram failure on curve

#### 5) Conclusion

Even though a slightly higher image resolution with higher top cutoff can improve the lane following performance slightly we sticked with the original resolution of 120x160 pixels with 40 pixels top cutoff because also other teams depend on the image resolution. We saw that our curvature estimation was able to detect the standard curves of Duckietown in many cases, at the same time it introduced a high latency which again lowered the performance. Therefore we decided to set the curvature resolution to 0 by default, which means that no curvature estimation is done. The code nevertheless is still in the lane filter to give a basis for further improvements.

Regarding the controller the test showed that the added integral part in the PID controller and the tuning of the control parameters gave a huge improvement of the lane following performance. The integrated feedforward part can not be used during the lane following, because it is depending on the curvature estimation. The feedforward part can be used by other teams (e.g. on intersections and parking lots) to integrate our controller.

The improved controller gives a clear improvement over the baseline controller as can be seen in Figure 10.1 and Figure 10.4

## 10.6. Future avenues of development

As there is always more to do and the performance for both the controller and the estimator can still be further enhanced we list in this section some suggestions for next steps to take.

#### 1) Estimator

To make curvature estimation applicable it has to be made more robust and at the same time more computationally efficient adding less delay to the system. In its current state the added delay is too high and the performance with curvature estimation switched on decreases.

#### 2) Controller

- Integrate the inputs of other teams, see.
- After doing the new kinematic calibration provided by the System Identification group:
  - The controller parameters should be adjusted according to the output of the

calibration.
  - The correction factors **velocity_to_m_per_s** and **omega_to_rad_per_s** need to be adjusted or ideally become obsolete and thus need to be deleted.
- To reduce impact of time delays, e.g. a Smith Predictor could be implemented.
- For the activation of the remaining interfaces (e.g. intersection navigation and parking), the respective commented out sections of the final code needs to be activated and the integration needs to be completed in collaboration with the other teams.

### 3) General

- Anti-Instagram should be enhanced, in order to identify more line segments and perceive the correct color.
- Adding a polarization filter to reduce impact of reflections on color perception.
- New edge detection with higher accuracy.
- Replacing the Raspberry Pi with something more computationally powerful to ensure low latency and enable a more complex pose estimation.

# Unit N-11
# PDD - Saviors

## 11.1. Part 1: Mission and scope

### 1) Mission statement

Detect obstacles, plan a route and drive around them.

### 2) Motto

URBEM ANATUM TUTIOS FACENDA

(Duckietown is to be made safer)

### 3) Project scope

**What is in scope**

Detecting cones and duckies of different sizes (obstacles) and plan a reasonable path or stop to avoid hitting them.

Stage 1: 1 obstacle and simply stop no crossing of lines (2 cases: drive by or stop)

Stage 2: 1 obstacle, drive by without crossing of line

Stage 3: 1 obstacle, potentially cross the line

Stage 3: Multiple obstacles, crossing line if needed

**What is out of scope**

No obstacles in crossings

Obstacles on the middle line

Complicated situations with oncoming traffic

**Stakeholders**

*Controllers (Lane following, adaptive curvature control)*

They ensure following our desired trajectory and we can tell them to stop or reduce the speed. They provide the heading and position relative to track (for path planning)

*Vehicle detection* - tbd

*Potentionally Anti-Instagram* They provide classified edges to limit the area where we have to find obstacles.

*Organizational Help* - System Architect - Software Architect - Duckiebook

## 11.2. Part 2: Definition of the problem

## 1) Problem statement

Reliably detect and avoid obstacles, plan a meaningful path around them or simply stop if nothing else is possible.

Robustness to changes in:

- Obstacle size
- Obstacle color (but only slight changes in yellow/orange)
- Illumination

## 2) Assumptions

- Obstacles are only yellow duckies (different sizes) and orange cones.
- No duckies on the middle line.
- No obstacles on intersections.
- Heading and position relative to track given.
- Control responsible for following trajectory.
- Possibility to influence vehicle speed (slow down, stop).
- Calibrated camera

## 3) Approach

Stage 0: Collect enough data and annotate them

Stage 1: Develop a first obstacle detection algorithm

Stage 2: Agree on final internal and external interface

Stage 3: from now on, obstacle detection and trajectory planning can be developed in parallel

Stage 4: handle the case(s) involving: 1 obstacle, no crossing of lines (2 cases: drive by or stop) → simple logical conditions

Stage 5: handle the case(s) involving: 1 obstacle, crossing line if needed (1 case: should always be possible to drive by) → Either with grid map or obstacle coordinates

Stage 6: handle the case(s) involving: Multiple obstacles, crossing line if needed

Stage 7: verify the whole system

## 4) Functionality provided

- Detect Obstacles
- Plan path around them or decide to stop

## 5) Resources required / dependencies / costs

- Calibrated camera.
- Position estimate and position uncertainty.
- Execution of our desired control commands
- Enough computing power

## 6) Performance measurement

- Avoid/hit-ratio in Stages 4-6 (see Approach)
- Percentage of correctly classified obstacles on our picture dataset

- Both of the measures above in case of changing light conditions

### 7) Functionality-resources trade-offs

- Robust obstacle detection (many filters,...) vs. computational efficiency
- Maximizing speed (e.g. controlles might want to do that) vs. motion blur

## 11.3. Part 3: Preliminary design

### 1) Modules

- **Obstacle Detection in 2D space**
- **Reconstruct 3D obstacle coordinates and radius**
- **Path planning/ Decision making**

### 2) Interfaces

**Detection 2D space**

- *Input:*
- Camera image
- Current position and orientation
- Lane coordinates
- Camera intrinsics
- (Curvature of upcoming track)
- *Output:*
- 2D obstacle coordinates

**Reconstruction of 3D obstacle coordinates and radius**

- *Input:*
- 2D obstacle coordinates
- Extrinsics
- *Output:*
- 3D obstacle coordinates and radius

**Avoid obstacle**

- *Input:*
- 3D obstacle coordinates
- Obstacle size
- Lane information
- *Output:*
- Trajectory
- Control command

### 3) Specifications

No need to revise duckietown specifications

### 4) Software modules

- Detection and Projection Node
- Path Planning Node

## 11.4. Part 4: Project planning

### 1) Timeline

| Date | Task Name | Target Deliverables |
|---|---|---|
| 15/11/17 | First Meeting | Preliminary Design Document |
| 17/11/17 | Record first bags | Pictures and Raw bag data |
| 22/11/17 | Exchange of ideas | Basic concept |
| 29/11/17 | Knowing Interfaces and State of the Art | Fine concept |
| 06/12/17 | | First implementation |
| ... | Testing | Optimized Code |
| ... | Documentation | Duckuments |
| 21/12/17 | | End of Project |

### 2) Data collection

Images of duckies on the road.

Video of a duckiebot in duckietown with recordings of the different stages.

To log:

- Distance to middle
- Theta
- Images
- Velocity

### 3) Data annotation

Label obstacles

*Relevant Duckietown resources to investigate:*

Image processing

feature extraction

MIT2016 object detection

Lane detection

Anti instagram

*Other relevant resources to investigate:*

OpenCV (filtering, color and edge detection)

### 4) Risk analysis

Interfaces (control approach of trajectory)

Computation power

### 5) Risk analysis

| Risk | Likelihood (1-10) | Impact | Actions required |
|---|---|---|---|
| Non robust State Estimation | tbd | very high | Communication/Collaboration with controlling subteam |
| Failure of following our desired control commands | tbd | very high | Communication/Collaboration with controlling subteam |

| Risk | Likelihood (1-10) | Impact | Actions required |
|---|---|---|---|
| Lack in computation power | 5 | high | early testing of whole system on duckiebot |
| Failure in duckie detection | 4 | extremely high | thorough testing on bags |
| Erroneously detecting the middle lane as duckie | 7 | middle | more sophisticated detection algorithm |

# Unit N-12
# The Saviors: intermediate report

## 12.1. Part 1: System interfaces

### 1) Logical architecture

**Description of the desired functionality:**

Detect duckies and cones during lane following. When the ducky/cone is not in the middle of the lane we try to avoid them, if avoidance is not possible, we simply stop. Our obstacle avoidance capability is limited due to the fact that the controllers cannot make our Duckiebot cross the lane. In a first step, if we are in an intersection, our node will stop performing its tasks. However, if we progress really fast we will also have a look at the case of obstacles in intersections.

**What will happen when we click "start"?**

If you click start our nodes are launched and we will start to detect duckies.

***In general our obstacle_detection_node is always active. The "flow" is as follows:***

**1.** Our "obstacle_detection"-node is activated and runs continuously. This nodes creates an instance of the class "detector" at the very beginning.

**2.** We have a an incoming filtered image stream from the "anti_instagram"-node to our node with the frequency, the "anti_instagram"-node is publishing. Our "obstacle_detection"-node regularly (i.e. with at least 2 Hz) calls the class functions of the "detector" instance.

**3.** For each frame the detector was called, the class function decides whether obstacles (duckies or traffic cones) are within the range of vision or not. Afterwards the "obstacle_detection"-node publishes an array containing all coordinates of the obstacles which is empty if there are none. The published topic is called "obstacle_coordinates".

**4.** The "obstacle_avoidance"-node takes this array as input and analyses it regarding which actions should be performed. The following scenarios are possible:

**4a)** The array is empty, therefore no action is performed and no flag will be published.

**4b)** At least one obstacle (e.g. a Duckie) is within the range of ¼ to ¾ of the lane (i.e. in the middle of our lane) such that it cannot be avoided without going to the opposite lane or driving outside of the street. In this case the "obstacle_avoidance"-node will set the "emergency_brake_flag" to true by publishing the topic "emergency_brake_flag". This will indicate to "The Controllers" to immediately stop the Duckiebot.

**4c)** If an obstacle is detected and it lies within the left or right quarter of our lane it can be avoided. In this case we will set the "obstacle_flag" to true by publishing this topic and also provide the controllers an input d (again by publishing a topic) to drive on the right side or the left side of the lane respectively.

**5.** If we passed the obstacle or if the obstacle disappeared we reset the "emergency_brake_flag" or the "obstacle_flag" and the controllers can take over again.

**6.** Additionally we are listening to the "lane_following_flag" which indicates, if true,

that the lane following is active. This tells us to perform our tasks. Otherwise our module should be inactive as our commands wouldn't have any effects anyway. In this way we can save computing resources and avoid misunderstandings.

7. Additionally we implemented an additional file "obstacle_detection_visualizer" which can be used for visualization of the detected obstacles. It is thought to run on an laptop (which is connected to the same rosmaster) because it is only used for the evaluation of the quality of the detection as well as the 2D-3D projection. In principle it can run on the raspberry pi as well. It subscribes to "obstacle_coordinates" and visualizes the obstacles in the 2D image by encircling the obstacles with a green rectangle (and publishing this modified image) as well as plotting them in the 3D coordinate frame (e.g. visualizable in RVIZ) as marker which shows the position as well as the size of the detected obstacles. With this additional software the optimization and the debugging are simplified significantly.

**Expected target values for the following quantities:**

- Detection Distance: 30-40cm
- Certainty of Detections: 90 percent
- Max. Amount of Detections in one frame: 3
- Max. Ratio of False Positives: 20 percent
- Obstacles avoided (without crash) to Obstacle detected ratio: 80 percent
- Object detection frame rate: at least 2 Hz

**Assumptions on other modules:**

- Control reaches desired d at most 10cm after request. Our request is "continuous".
- Steady state control error smaller than 2cm
- d (perpendicular position to lane direction) Position estimate accuracy smaller than 2cm
- when setting the emergency_flag, we stop within the next 5 cm

## 2) Software architecture

**List of nodes which are to be developped:**

- obstacle_detection
- obstacle_detection_visualizer
- obstacle_avoidance

**Published and subscribed topics for each node, including an estimate of the introduced latency for the topics being published and an assumption on the latency for all subscribed topics:**

***obstacle_detection***

*published topics:*

- obstacle_coordinates (max 0.5s)

*subscribed topics:*

- cameraImage (no latency)

***obstacle_detection_visualizer***

*published topics:*

- obst_detect/image/compressed (max 0.5s)
- obst_detect/visual/visualize_obstacle (max 0.5s)

*subscribed topics:*

- obstacle_coordinates (max 0.5s)
- cameraImage (no latency)

***obstacle_avoidance***

*published topics:*

- desired d (computing time)
- obstacle_avoidance_active_flag (max. 0.3s)
- obstacle_emergency_stop_flag (max. 0.3s)

*subscribed topics:*

- obstacle_coordinates (max 0.5s)
- state_estimation (current d, velocity. probably theta) (0.2s?)

## 12.2. Part 2: Demo and evaluation plan

### 1) Demo plan

**How do you envision the demo?**

Duckiebot driving around Duckietown with duckies in the street (on straights, (in curves), not in intersections)

- First the standard code will be running, which will result in Duckiebots crashing into duckies on the street.
- Afterwards we will launch the Software from last year which will clearly show some strong losses in the quality of the obstacle detection.
- With our code running, the Duckiebot will avoid duckies by driving around them or stopping. If stopped we will remove the duckie which will make the bot drive again.

**What hardware components do you need?**

- Enough duckies and traffic cones for being able to perform the evaluation. Traffic cones are already ordered and will be shipped on Monday 11th of December at the latest.
- The minimum city size to properly perform the demo is in our opinion an equivalent size as the one in the ML building.

### 2) Plan for formal performance evaluation

**How do you envision the performance evaluation? Is it experiments? Log analysis?**

**1. Performance evaluation of the Saviors only:**

*This first evaluation will be a general evaluation based on logs. We basically want to check if obstacles are detected correctly, and if the avoidance reacted as expected to prevent crashes.*

**1a)** Detailed Evaluation Parameters:

- compare the labelled data (= groundtruth) to the results when putting the same image into our pipeline
- checking the acutal frequency of our node
- test duckie recognition at different orientations and evaluate the maximum angle possible in which we still detect duckies
- evaluate the precision when having duckies only on our lane and on both, our lane and the opposite lane!!!

- evaluate the number of false positives in the 3 different situations: on straights, in "normal" curves, in the S-curve in duckietown
- measure the accuracy in centimeters of the real position of a duckie and the position we estimate!!!

**2. Performance evaluation for the Saviors / Controllers / State estimation:**

*In a first step we will evaluate our trajectory generation and control commandy by our obstacle_avoidance node in designed situations. This means that we assume having a certain state, position of an obstace and then we verify that we plan a feasible path around the obstacle without crossing the lane or to make the decision to stop*

**2a)** Detailed evaluation Parameters:

- percentage of correct decisions

*The second step will be with the real state estimation, lane following and obstacle detection. This evaluation will be based on experiments with our duckiebot driving around town with obstacles in the lane.*

**2b)** Detailed evaluation Parameters:

1. percentage of correct decisions
2. percentage of correct executions when having made a correct decision, meaning really driving around the duckie without hitting it or sending the stop command early enough to not crash into duckies

**2c)** the 2 measures above will be evaluated in different situations:

1. only place duckies on straights
2. only place duckies in turns
3. only place duckies in the S-Turns
4. no limit on placing of the duckies

**2d)** All of the above 4 situation will be also evaluated with:

- Variation 1: only duckies in our lane
- Variation 2: also duckies on the opposite lane possible

In our case, only the performance evaluation on bags can be designed for minimal human intervention whereas all the other test have to be done in presence of a human who is able to stop the system when a potential crash occurs.

## 12.3. Part 3: Data collection, annotation, and analysis

### 1) Collection

**How much data do you need?**

We already recorded 13 rosbags and are testing and improving the current obstacle detection. For the first step we think that this should work out.

**How are the logs to be taken? (Manually, autonomously, etc.)**

We took them manually but tried to "simulate" some non optimal controller behaviour, also because the autonomous mode did not work. For the future we might consider taking autonomous logs but only if we don't crash into obstacles

**Do you need extra help in collecting the data from the other teams?**

No

## 2) Annotation

**Do you need to annotate the data?**

Yes!

**At this point, you should have you tried using thehive.ai to do it. Did you? Are you sure they can do the annotations that you want?**

Yes we already tried it and it worked quite well. We use this platform to label duckies for us so that it will be possible to measure the performance of our duckie detection automatically on logs. Otherwise a human would need to label all of the pictures which would take a lot of time.

## 3) Analysis

**Do you need to write some software to analyze the annotations?**

We plan to write some software which applies our algorithm on every image of the log and which will mark the duckies using a box around them. Afterwards it will use the annotated data to read out where they have drawn the box and calculate the distance as well as visualize these results and differences. We then try to calculate the percentage of reasonable results and the percentage of outliers.

**Are you planning for it?**

Yes.

# Unit N-13
# The Saviors: Final Report

This is the final report of the fall 2017 Saviors group from ETH Zurich, namely Fabio Meier (fmeier@ethz.ch), Julian Nubert (nubertj@ethz.ch), Fabrice Oehler (foehler@ethz.ch) and Niklas Funk (nfunk@ethz.ch). We enjoyed contributing to this great project and in case there are any open questions left after having read this report, do not hesitate to contact us.

## 13.1. Structure

The Final Result
Mission and Scope
Definition of the Problem
Contribution / Added Functionality
Formal Performance Evaluation / Results
Future Avenues of Development
Theory Chapter

## 13.2. The Final Result

The Saviors Teaser:

**Note:** See the operation manual to reproduce these results.

The code description can be found here in the Readme.

## 13.3. Mission and Scope

***"URBEM ANATUM TUTIOS FACIENDA (EST) - MAKE DUCKIETOWN A SAFER PLACE"***

The goal of Duckietown is to provide a relatively simple platform to explore, tackle and solve many problems linked to autonomous driving. "Duckietown" is simple in the basics, but an infinitely expandable environment. From controlling single driving Duckiebots until complete fleet management, every scenario is possible and can be put into practice. Due to the previous classes and also the great work of many volunteers, many software packages were already developed and provided a

solid basis. But something was still missing.

### 1) Motivation

So far, none of the mentioned modules was capable of reliably detecting obstacles and reacting to them in real time. We were not the only ones who saw a problem in the situation at that time: *"Ensuring safety is a paramount concern for the citizens of Duckietown. The city has therefore commissioned the implementation of protocols for guaranteed obstacle detection and avoidance."*[31]. Therefore the foundation of our complete module lies in the disposal of this shortcoming. Finding a good solution for this safety related and very important topic helped us to stay motivated every day we were trying to improve our solution.

The goal of our module is to detect obstacles and react accordingly. Due to the limited amount of time, we focused the scope of our module to two points:

**1.** In terms of detection, on the one hand we focused to reliably detect yellow duckies and therefore to saving the little duckies that want to cross the road. On the other hand we had to detect orange cones to not crash into any construction site in Duckietown.

**2.** In terms of reacting to the detected obstacles we were mainly restricted by the constraint given by the controllers of our Duckiebots, who do not allow us to cross the middle of the road. This eliminated the need of also having to implement a Duckiebot detection algorithm. So we focused on writing software which tries to avoid obstacles within our own lane if it is possible (e.g. for avoiding cones on the side of the lane) and to stop otherwise.

Besides aforementioned restrictions and simplifications we faced the general problem of detecting obstacles given images from a monocular RGB camera mounted at the front of our Duckiebot and reacting to them properly without crashing or erroneously stopping the Duckiebot. Both processes above have to be implemented and have to run on the Raspberry Pis in real time. Due to the strong hardware limitations, we decided to not use any learning algorithms for the obstacle detection part. As it later transpired, a working "hard coded" software needs thorough analysis and understanding of the given problem. However, in the future, considering additional hardware like e.g. Tung, "Google offers Raspberry Pi owners this new AI vision kit" (2017), this decision might have to be reevaluated.

In practice a well working obstacle detection is one of the most important parts of an autonomous system to improve the reliability of the outcome even in unexpected situations. Therefore the relevance of an obstacle detection in a framework like "Duckietown" is very important. Especially because the aim of "Duckietown" is to simulate the real world as realistic as possible and also in other topics such as fleet planning, a system with obstacle detection behaves completely different than a system without.

### 2) Existing Solution

There was a previous implementation from the MIT classes in 2016. Of course we had a look into the old software and found out that one step of them was quite similar to ours: They based their obstacle detection on the colors of the obstacles. Therefore they also did their processing in the HSV color space as we did. Further information on why filtering colors in the HSV space is advantageous can be found in the Theory Chapter.

Nevertheless, we implemted our solution from scratch and didn't base ours on any further concepts found in their software. That is why you won't find any further similarites between the two implementations. The reasons for implementing our own code from scratch can be found in the next section Opprtunity. In short, last year's solution considered the image given the original camera's perspective and tried to classify the objects based on their contour. We are using a very different approach concerning those two crucial parts as you can see in the Contribution section.

#### 3) Opportunity

From the beginning it was quite clear that the old software was not working reliable enough. The information we have been given was that it was far off detecting all obstacles and that there were quite a few false positives: It detected yellow line segments in the middle of the road as obstacles (color and size are quite similar to the ones of typical duckies) which led to a stopping of the car. Furthermore, extracting the contour of every potential obstacle is highly computationally expensive. As mentioned, we had a look into the software and tried to understand it as well as possible but because it was not documented at all we couldn't go much into detail. On top of that, from the very beginning we had a completely different idea of how we wanted to tackle these challenges.

We also tried to start their software but we couldn't make it run after a significant amount of time. The readme file didn't contain any information and the rest of the software was not documented as well. This also reinforced us in our decision to write our own implementation from scratch.

#### 4) Preliminaries

Since our task was to reliably detect obstacles using a monocular camera only, we mainly dealt with processing the camera image, extracting the needed information, visualizing the results and to act accordingly in the real world.

For understanding our approach we tried to explain and summarize the needed concepts in the theory chapter (see section Theory Chapter). There you will find all the references to the relevant sources.

## 13.4. Definition of the Problem

In this chapter we try to explain our problem in a more scientific way and want to show all needed steps to fullfill the superordinate functionality of **"avoiding obstacles"**.

The only input is a RGB colored image, taken by a monocular camera (only one camera). The input image could look as Figure 13.1.

Figure 13.1. Sample image including some obstacles

With this information given, we want to find out whether an obstacle is in our way or not. If so, we want to either stop or adapt the trajectory to pass without crashing into the obstacle. This information is then forwarded as an output to the controllers who will process out commands and try to act accordingly.

Therefore one of the first very important decisions was to separate the *detection* and *reaction* parts of our **saviors pipeline**. This decision allowed us to divide our work efficiently, to start right away and is supposed to ensure a wide range of flexibility in the future by making it possible to easily replace, optimize or work on one of the parts separately (either the obstacle avoidance strategies or obstacle detection algorithms). Of course it also includes having to define a clear, reasonable interface in between the two modules, which will later be explained in detail.

You can have a look in our Preliminary Design Document and Intermediate Report to see how we defined the following topics in the beginning: The problem statement, our final objective, the underlying assumptions we lean on and the performance measurement to quantitatively check the perfomance of our algorithms. For the most part, it worked out to adhere to this document but for sake of completeness we will shortly repeat them again in the following for each of the two submodules.

### 1) Part 1: Computer Vision - Description

In principle we wanted to use the camera image only to ***reach the following***:

1. **Detect** the obstacles in the camera image
2. **Viusalize** them in the camera image for tuning parameters and optimizing the code
3. Give the **3D coordinates** of every detected obstacle in the real world
4. Give the **size** of every detected obstacle in the form of a radius around the 3D coordinate
5. Label each obstacle if it's **inside or outside the lane boundaries** (e.g. for the purpose of not stopping in a curve)
6. **Visualize** them as markers in the 3D world (rviz)

Since every algorithm has its limitations, we made the following ***assumptions***:

- Obstacles are only yellow duckies and orange cones
- Calibrated camera including intrinsics and extrinsics

Those assumptions changed slightly since the *Preliminary Design Document* because we are now also able to detect duckies on the middle line and in intersections.

It was our aim to reach the maximum within these specified limits. Therefore our goal was not only the detection and visualization in general but we also wanted to reach a ***maximum in robustness*** with respect to changes in:

- Obstacle size
- Obstacle color (within orange, and yellow to detect different traffic cones and duckies)
- Illumination

For evaluating the ***performance***, we used the following metrics, evaluated under different light conditions and different velocities (static and in motion):

- Percentage of correctly classified obstacles on our picture datasets
- Percentage of false positives
- Percentage of missed obstacles

An evaluation of our goals and the reached performance can be found in the Performance Evaluation section.

Our ***approach*** is simply based on analysing incoming pictures for obstacles and trying to track them to make the algorithm more robust against outliers. Since we only rely on the monocular camera, we do not have any direct depth information given. In theory, it would be possible to estimate the depth of each pixel through some monocular visual odometry algorithm considering multiple consecutive images. However this would be extremely computationally expensive. The large amount of motion blur in our setup, a missing IMU (for estimating the absolute scale) further argue against such an approach. In our approach we use the extrinsic calibration to estimate the position of the given obstacles. The intuition behind that is that it is possible to assume that all pixels seen from the camera belong to the ground plane (except for obstacles which stand out of it) and that the Duckikebot's relative position to this ground plane stays constant. Therefore you can assign a real world 3D coordinate to every pixel seen with the camera. For more details refer to the section below.

The final output is supposed to look as Figure 13.2.

Figure 13.2. Final output image including visualization of detected obstacles

### 2) Part 2: Avoidance in Real World - Description

With the from Part 1 given 3D position, size and the labelling whether the object is inside the lane boundaries or not, we wanted to reach the final objectives:

1. Plan path around obstacle if possile (we have to stay within our lane)
2. If this is not possible, simply stop

The assumptions for correctly reacting to the previously detected obstacles are:

- Heading and position relative to track given
- "The Controllers" are responsible for following our trajectory
- Possibility to influence vehicle speed (slow down, stop)

As we now know, the first assumption is normally not fulfilled. We describe in the functionality section why this comes out to be a problem.

For measuring the performance we used:

- Avoid/hit ratio
- Also performed during changing light conditions

## 13.5. Contribution / Added Functionality

### 1) Software Architecture

In general we have four interfaces which had to be created throughout the implementation of our software:

**1.** At first, we need to recieve an incoming picture which we want to analyse. As our chosen approach includes filtering for specific colors, we are obviously dependent on the lighting conditions. In a first stage of our project, we nevertheless simply subscribed to the raw camera image because of the considerable expense of integrating the *Anti Instagram Color Transformation* and since the *Anti Instagram* team also first had to further develop their algorithms. During our tests we quickly recognized that our color filtering based approach would always have some troubles if we don't compensate for the lighting change. Therefore, in the second part of the project we closely collaborated with the *Anti Instagram team* and are now subscribing to a color corrected image provided by them. Currently, to keep computational power on our Raspberry Pi low, the corrected image is published at 4Hz only and the color transformation needs at most 0.2 seconds.

**2.** The second part of our System Integration is the internal interface between the object detection and avoidance part. The interface is defined as a *PoseArray* which has the same timestamp as the picture from which the obstacles have been extracted. This Array, as the name already describes, is made up of single poses. The meaning of those are the following:

The position *x and y* describe the real world position of the obstacle which is in our case the center front coordinate of the obstacle. Since we assume planarity, the *z coordinate* of the position is not needed. That is why we are using this *z coordinate* to describe the radius of the obstacle.

Furthermore a negative *z coordinate* shows that there is a white line in between us and the obstacle which indicates that it is not dangerous to us since we assume to always having to stay in the lane boundaries. Therefore this information allows us to not stop if there is an obstacle behind a turn.

As for the scope of our project, the orientation of the obstacles is not really important, we use the *remaining four elements* of the Pose Message to pass the pixel coordinates of the bounding box of the obstacle seen in the bird view. This is not needed for our "Reaction" module but allows us to implement an efficient way of visualisation which will be later described in detail. Furthermore, we expect our obstacle detection module to add an additional delay of about max. 0.3s.

**3.** The third part is the interface between our obstacle avoidance node and the

*Controllers*. The obstacle avoidance node generates an obstacle avoidance *pose array* and obstacle avoidance *active flag*.

The obstacle avoidance pose array is the main interface between the Saviors and the group doing lane control. We use the pose array to transmit *d_ref* (target distance to middle of the lane) and *v_ref* (target robot speed). The *d_ref* is our main control output which enables us to position the robot inside the lane and therefore to avoid objects which are placed close the laneline on the track. Furthermore *v_ref* is used to stop the robot when there is an unavoidable object by setting the target speed to zero.

The *flag* is used to communicate to the lane control nodes when the obstacle avoidance is activated which then triggers *d_ref* and *v_ref* tracking.

**4.** The fourth part is an optional interface between the Duckiebot and the user's personal Laptop. Especially for the needs of debugging and infering what is going on, we decided to implement a visualisation node which can visualize on the one hand the input image including bounding boxes around all the objects which were classified as obstacles and furthermore this node can output the obstacles as markers which can be displayed in rviz.

In the following (Figure 13.3) you find a graph which summarises our software packages and gives a brief overview.

Figure 13.3. Module overview 'The Saviors'

## 2) Part 1: Computer Vision - Functionality

Let's again have a look on the usual incoming camera picture in Figure 13.1.

In the very beginning of the project, like the previous implementation in 2016, we tried to do the detection in the normal camera image but we tried to optimize for more efficient and general obstacle descriptors. Due to the specifications of a normal camera, lines which are parallel in the real world are in general not parallel any longer and so the size and shape of the obstacles are disturbed (elements of the same size appear also larger in the front than in the back). This made it very difficult to reliably differentiate between yellow ducks and line segments. We tried several different approaches to overcome this problem, namely:

- Patch matching of duckies viewed from different directions
- Patch matching with some kind of an ellipse (because line segments are supposed to be square)
- Measuring the maximal diameter
- Comparing the height and the width of the objects
- Taking the pixel volume of the duckies

Unfortunately none of the described approaches provided a sufficient perfor-

mance. Also a combination of them didn't make the desired impact. All metrices which are somehow associated with the size of the object just won't work because duckies further away from the duckiebot are simply a lot smaller than the one very close to the Duckiebot. All metrices associated with the "squareness" of the lines were strongly disturbed by the ocurring motion blur. This makes finding a general criterion very difficult and made us think about changing the approach entirely.

Therefore we developed and came up with the following new approach!

***Theoretical Description:***

In our setup, through the extrinsic camera calibration, we are given a mapping from each pixel in the camera frame to a corresponding real world coordinate. It is important to mention that this transformation assumes all seen pixels in the camera frame to lie in one plane which is in our case the ground plane/street. Our approach exactly exploits this fact by transforming the given camera image into a new, bird's view perspective which basically shows one and the same scene from above. Therefore the information provided by the extrinsic calibration is essential for our algorithm to work properly. In Figure 13.4 you can see the newly warped image seen from the *bird's view perspective*. This is one of the most important steps in our algorithm.

Figure 13.4. Image now seen from the bird's view perspective

This approach has already been shown by Prof. Davide Scaramuzza (UZH) and some other papers and is referred as **Inverse Perspective Mapping Algorithm**. (see: [32],[33],[34])

What stands out, is that the lines which are parallel in the real world are also parallel in this view. Generally in this "bird's" view, all objects which really belong to the ground plane are represented by their real shape (e.g. the line segments are exact rectangles) while all the objects which are not on the ground plane (namely our obstacles) are heavily disturbed in this top view. This top view is roughly keeping the size of the elements on the ground whereas the obstacles are displayed a lot larger.

*The theory behind the calculations and why the objects are so heavily distorted can be found in the Theory Chapter.*

Either way, we take advantage of this property. Given this bird's view perspective, we still have to extract the obstacles from it. To achieve this extraction, we first filter out everything except for orange and yellow elements, since we assumed that we only want to detect yellow duckies and orange cones. To simplify this step significantly, we transform the obtained color corrected images (provided by the Anti Instagram module) to the ***HSV color space***. We use this HSV color space and not the RGB space because it is much easier to account for slightly different illuminations - which of course still exist since the performance of the color correction is logically not perfect - in the HSV room compared to RGB. For the theory behind the HSV space, please refer to our appropriate Theory Chapter.

After this first color filtering process, there are only objects remaining which have approximately the colors of the expected obstacles. For the purpose of filtering out the real obstacles from the bunch of all the remaining objects which passed the color filter, we decided to do the following: We segment the image of the remaining objects, i.e. all connected pixels in the filtered image are getting the same label such that you can later analyse the objects one by one. Each number then represents an obstacle. For the process of segmentation, we used the following algorithm. (see [35])

Given the isolated objects, the task remains to finally decide which objects are considered obstacles and which not. In a first stage, there is a filter criterion based on a rotation invariant feature, namely the two eigenvalues of the inertia_tensor of the segmented region when rotating around its center of mass. (see [36])

In a second stage, we apply a tracking algorithm to reject the remaining outliers and decrease the likelihood for misclassifications. The tracker especially aims for objects which passed the first stage's criterion by a small margin.

For further informations and details about how we perform the needed operations, please refer to the next chapter.

The final output of the detection module is the one we showed in Figure 13.2.

***Actual Implementation:***

Now we want to go more into detail how we implemented the described steps.

In the beginning we again start from the picture you can see in Figure 13.1. In our case this is now the corrected image coming out form the *image_transformer_node* and was implemented by the *anti instagram* group. We then perform the follwing steps:

**1.** In a first step we crop this picture to make our algorithm a little bit more efficient and due to our limited velocities, it makes no sense to detect obstacles which are not needed to be taken into consideration by our obstacle avoidance module. However, we do not simply crop the picture by a fixed amount of pixels, but we use the extrinsic calibration to neglect all the pixels which are farther away than a user defined threshold, which is at the moment at 1.7 meters. So the amount of pixels which are neglected are different for every Duckiebot and depend on the extrinsic calibration. The resulting image can be seen in Figure 13.5. The calculations to find out where You have to cut the image are quite simple (note that it still bargains for homogeneous coordinates):

$$p_{camera} = H^{-1} P_{world}$$

Figure 13.5. Cropped image

**2.** Directly detecting the obstacles from this cropped input image failed for us due to the reasons descibed above. That is why the second step is to perform the trans-

formation to the bird's view perspective. For transforming the image, we first use the corners of the cropped image and transorm it to the real world. Then we scale the real world coordinates to pixel coordinates, so that it will have a width of 640 pixels afterwards. For warping all of the remaining pixels with low artifacts we use the function *cv2.getPerspectiveTransform()*. The obtained image can be seen in Figure 13.4.

**3.** Then we transform the given RGB picture into the HSV colorspace and apply the yellow and orange filter. While a HSV image is hardly readable for humans, it is way better to filter for specific colors. The obtained pictures can be seen in Figure 13.6 and Figure 13.7. The color filter operation is performed by the cv2 function *cv2.inRange(im_test, self.lower_yellow, self.upper_yellow)* where lower_yellow and upper_yellow are the thresholds for yellow in the HSV color space.

Figure 13.6. Yellow filtered image

Figure 13.7. Orange filtered image

**4.** Now there is the task of segmenting/isolating the objects which remained after the color filtering process. At the beginning of the project we therefore implemented our own segmentation algorithm which was however more inefficient and led to an overall computational load of 200% CPU usage and a maximum frequency of our whole module of about 0.5 Hz only. By using the scikit-image module which provides a very efficient label function, the computational efficiency could be shrunk considerably to about 70% CPU usage and allows the whole module to run at up to 3 Hz. It is important to remember that in our implementation the segmentation process is the one which consumes the most power. The output after the segmentation is the one in Figure 13.8, where the different colors represent the different segmented objects.

Figure 13.8. Segmented image

**5.** After the segmentation, we analyse each of the objects separately. At first there is a general filter which ensures that we are neglecting all the objects which contain less than a user influenced threshold of pixels. Since as mentioned above, the homographies of all the users are different, the exact amount of pixels, an object is

required to have, is again scaled by the individual homography. This is followed by a more in detail analysis which is color dependent. On the one hand there is the challenge to detect the orange cones reliably. Speaking about cones, the only other object that might be erroneously detected as orange are the stop lines. Of course, in general the goal should be to very reliably detect orange but as the light is about to change during the drive, we prepared to also detect the stop lines and being able to cope with them when they are erroneously detected. The other general challenge was that all objects that we have to detect can appear in all different orientations. Simply inferring the height and width of the segmented box, as we did it in the beginning, is obviously not a very good measure (e.g. in Figure 13.9 in the lower left the segmented box is square while the cone itself is not quadratic at all).

Figure 13.9. Bird's view with displayed obstacle boxes

That is why it is best to use a rotation invariant feature to classify the segmented object. In our final implementation we came up with using the two eigenvalues of the inertia tensor, which are obviously rotation invariant (when being ordered by their size). Being more specific about the detection of cones, when extracting the cone from Figure 13.8 it is looking like in Figure 13.10, while an erroneous detection of a stop line is looking like in Figure 13.11.

Figure 13.10. Segmented cone

Figure 13.11. Segmented stop line

Our filter criteria is now the ratio between the eigenvalues of the inertia tensor. This ratio is always by a factor of about 100 greater in case the object is a cone, compared to when we erroneously segment a red stop line. This criteria is very stable that is why there is no additional filtering needed to detect the cones.

If the segmented object is yellowish, things get a little more tricky as there are always many yellow objects in the picture, namely the middle lines. Line elements can be again observed under every possible orientation. Therefore the eigenvalues of the inertia tensor, which are as mentioned above rotation invariant, are again the way to go. In Figure 13.12 you can see a segmented line element and in Figure 13.13 again a segmented duckie.

Figure 13.12. Segmented middle line

Figure 13.13. Segmented duckie

As the labelled axis already reveal, they are of a different scale, but as we also got very small duckies, we had to choose a very small threshold. To detect the yellow duckies, the initial condition is that the first eigenvalue has to be greater than 20. This criteria alone however includes to sometimes erroneously detecting the lines as obstacles, that is why we implemented an additional tracking algorithm which works as follows: If an object's first eigenvalue is greater than 100 pixels and it is

detected twice - meaning in two consecutive images there is a object detected at roughly the same place - it is labelled as an obstacle. However, if an object is smaller or changed the size by more than 50% in the consecutive frames, then a more restrictive criteria is enforced. This more restrictive criterion states that we must have tracked this object for at least for 3 consecutive frames before being labelled as an obstacle. This criteria is working pretty well and a more thorough evaluation will be provided in the next section. In general those criteria help that the obstacles can be detected in any orientation. The only danger to the yellow detecting algorithm is motion blur, namely when the single lines are not separated but connected together by "blur".

**6.** After analysing each of the potential obstacle objects, we decide whether it is an obstacle or not. If so, we continue to steps **7.** and **8.**.

**7.** Afterwards, we calculate the position and radius of all of the obstacles. After segmenting the object we calculate the 4 corners (which are connected in Figure 13.14 to form the green rectangle). We defined the obstacle's position as the midpoint of the lower line (this point surely lies on the ground plane). For the radius, we use the distance in the real world between this point and the lower right corner. This turned out to be a good approximation of the radius. For an illustration you can have a look at Figure 13.14.

Figure 13.14. Position and radius of the obstacle

**8.** Towards the end of the project we came up with one additional last step based on the idea that only obstacles inside the white lane boundaries are of interest to us. That is why for each obstacle, we look whether there is something white in between us and the obstacle. In Figure 13.15 you can see an example situation where the obstacle inside the lane is marked as dangerous (red) while the other one is marked as not of interest to us since it is outside the lane boundary (green). In Figure 13.16 you see the search lines (yellow) along which we search for white elements.

Figure 13.15. Classification if objects are dangerous or not

Figure 13.16. Search lines to infer if something white is in between

**9.** As the last step of the detection pipeline we return a list of all obstacles including all the information via the *Posearray*.

### 3) Part 2: Avoidance in Real World - Functionality

The Avoidance deals with drawing the right conclusions from the received data and forwarding it.

***Theoretical Description:***

With the separation of the detection, an important part of the avoidance node is the interaction with the other work packages. We determined the need of getting information about the remaining Duckietown besides the detected obstacles. The obstacles need to be in relation to the track, in order to assess whether we have to stop, can drive around obstacles or if it is even already out of track. Due to other teams already working on the orientation within Duckietown, we deemed it best to not implement any further detections (lines, intersections etc.) in our visual perception pipeline. This saves similar algorithms being run twice on the processor. We decided to acquire the values of our current pose relative to the side lane, which is determined by the *devel-linedetection* group.

The idea was to make the system highly flexible. The option to adapt to following situations was deemed desirable:

- Multiple obstacles. Different path planning in case of a possible avoidance might be required.
- Adapted behavior if the robot is at intersections.
- Collision avoidance dependent on the fleet status within the Duckietown. Meaning if a Duckiebot drives alone in a town it should have the option to avoid a collision by driving onto the opposite lane.

Obstacles sideways of the robot were expected to appear as the Duckietowns tend to be flooded by duckies. Those detections on the side as well as far away false positive detections should not make the robot stop. To prevent that, we intended on implementing a parametrized bounding box ahead of the robot. Only obstacles within that box would be considered. Depending on the certainty of the detections as well as the yaw-velocities the parametrization would be tuned.

The interface getting our computed desired values to impact the actual Duckiebot is handled by *devel-controllers*. We agreed on the usage of their custom message format, in which we send desired values for the lateral lane position and the longitudinal velocity. Our intention was to account for the delay of the physical system

in the avoider node. Thus our planned trajectory will reach the offset earlier than the ideal-case trajectory would have to.

Due to above mentioned interfaces and multiple levels of goals we were aiming for an architecture which allows **gradual commissioning**. The intent was to be able to go from basic to more advanced for us as well as for groups in upcoming years. Those should be able to extend our framework and not have to rebuild it.

The logic shown in Figure 13.17 displays one of the first stages in the commissioning. Key is the reaction to the number of detected obstacles. Later stages will not trigger an emergency stop in case of multiple obstacle detections within the bounding box.

Figure 13.17. Logic of one of the First Stages in Commissioning

Our biggest concern were the added inaccuracies until the planning of the trajectory. Those include:

- Inaccuracy of the currently determined pose
- Inaccuracy of the obstacle detection
- Inaccuracy of the effectively driven path aka. controller performance

To us the determination of the pose was expected to be the most critical. Our preliminary results of the obstacle detection seemed reasonably accurate. The controller could be tweaked that the robot would rather drive out of the track than into the obstacle. Though an inaccurate estimation of the pose would just widen the duckie artificially.

*Devel-controllers* did not plan on being able to intentionally leave the lane. Meaning the space left to avoid an obstacle on the side of the lane is tight making above uncertainties more severe.

We evaluated the option to keep track of our position inside the map. Given a decent accuracy of said position we'd be able to create a map of the detected obsta-

cles. Afterwards - especially given multiple detections (also outside of the bounding box) - we could achieve a further estimation of our pose relative to the obstacles. This essentially would mean creating a *SLAM-algorithm* with obstacles as landmarks. We declared as out of scope given the size of our team as well as the computational constraints. The goal was to make use of a stable, continuous detection and in each frame react on it.

***Actual Implementation:***

**Interfaces**

One important part of the Software is the handling of the interfaces, mainly to *devel_controllers*. For further informations on this you can refer to the Software Architecture Chapter.

**Reaction**

The obstacle avoidance part of the problem is handled by an additional node, called the *obstacle_avoidance_node*. The node uses two main inputs which are the obstacle pose and the lane pose. The obstacle pose is an input coming from the obstacle detection node, which contains an array of all the obstacles currently detected. Each array element consists of an x and y coordinate of an obstacle in the robot frame (with the camera as origin) and the radius of the detected object. By setting the radius to a negative value, the detection node indicates that this obstacle is outside the lane and should not be considered for avoidance. The lane pose is coming from the line detection node and contains among other unused channels the current estimated distance to the middle of the lane (d) as well as the current heading of the robot $\boldsymbol{\theta}$. Figure 13.18 introduces the orientations and definitions of the different inputs which are processed in the obstacle avoidance node.

Figure 13.18. Variable Definitions seen from the Top

Using the obstacle pose array we determine how many obstacles need to be considered for avoidance. If the detected obstacle is outside the lane and therefore

marked with a negative radius by the obstacle detection node we can ignore it. Furthermore, we use the before mentioned bounding box with tunable size which assures that only objects in a certain range from the robot are considered. As soon as an object within limits is inside of the bounding box, the obstacle_avoidance_active flag is set to true and the algorithm already introduced in Figure 13.17 is executed.

**Case 1: Obstacle Avoidance**

If there is only one obstacle in range and inside the bounding box, the obstacle avoidance code in the avoider function is executed. First step of the avoider function is to transform the transmitted obstacle coordinates from the robot frame to a frame which is fixed to the middle of the lane using the estimated measurements of $\boldsymbol{\theta}$ and d. Doing this transformation allows us to calculate the distance of the object from the middle line. If the remaining space (in the lane (subtracted by a safety avoidance margin) is large enough for the robot to drive through we proceed with the obstacle avoidance, if not we switch to case 2 and stop the vehicle. Please refer to Figure 13.19

Figure 13.19. Geometry of described Scene

If the transformation shows that an avoidance is possible we calculate the d_ref we need to achieve to avoid the obstacle. This is sent to the lane control node and then processed as new target distance to the middle of the lane. The lane control node uses this target and starts to correct the Duckiebot's position in the lane. With each new obstacle pose being generated this target is adapted so that the Duckiebot eventually reaches target position. The slow transition movement allows us to avoid the obstacle even when it is not visible anymore shortly before the robot is at the same level as the obstacle.

At the current stage, the obstacle avoidance is not working due to very high inaccuracies in the estimation of $\boldsymbol{\theta}$. The value shows inaccuracies with an amplitude of 10°, which leads to wrong calculations of the transformation and therefore to misjudgement of the d_ref. The high amplitude of these imprecisions could be transformed to a uncertainty factor of around 3 which means that each object is around

3 times its actual size which means that even a small obstacle on the side of the lane would not allow a safe avoidance to take place. For this stage to work, the estimation of $\boldsymbol{\theta}$ would need significant improvement.

**Case 2: Emergency Stop**

Conditions for triggering an emergency stop:

- More than one obstacle in range
- Avoidance not possible because the obstacle is in the middle of the lane
- **Currently every obstacle detection in the bounding box triggers an emergency stop due to the above reasons**

If one of the above scenarios occurs, an avoidance is not possible and the robot needs to be stopped. By setting the target speed to zero, the lane controller node stops the Duckiebot. As soon as the situation is resolved by removing the obstacle which triggered the emergency stop, the robot can proceed with the lane following.

These tasks are then repeated at the frame rate of the obstacle detection array being sent.

### 4) Required Infrastructure - Visualizer

Especially when dealing with a vision based obstacle detection algorithm it is very hard to infer what is going on. One has to also keep the visual outputs low, to consume as less computing power as possible, especially on the Raspberry Pi. This is why we decided to not implement one single *obstacle detection node*, but effectively two of them, together with some scripts which should help to tune the parameters offline and to infer the number of false positives, etc.. The node which is designed to be run on the Raspberry Pi is our normal obstacle_detection_node. This should in general be run such that there is no visual output at all but that simply the PoseArray of obstacles is published through this node.

The other node, namely the *obstacle_detection_visual_node* is designed to be run on your own laptop which is basically visualising the information given by the posearray. There are two visualisations available. On the one hand there is a marker visualisation in rviz which shows the position and size of the obstacles. In here all the dangerous obstacles which must be considered are shown in red, whereas the non critical (which we think that they are outside the lane boundaries) are marked in green. On the other hand there is also a visualisation available which shows the camera image together with bounding boxes around the detected obstacles. Nevertheless, this online visualisation is still dependent on the connectivity and you can only hardly "freeze" single situations where our algorithm failed. That is why we also included some helpful scripts into our package. One script allows to thoroughly input many pictures and outputs them labelled together with the bounding boxes, while another one outputs all the intermediate steps of our filtering process which allows to fastly adapt e.g. the color thresholds which is in our opinion still the major reason for failure. More information on our created scripts can be found in our Readme on GitHub.

### 5) Recorded Logs

For being able to thorougly evaluate and tune our algorithms, we recorded various bags, which we uploaded to the Duckietown logs database.

## 13.6. Formal Performance Evaluation / Results

### 1) Evaluation of the Interface and Computational Load

In general as we are dealing with many color filters a reasonable color corrected image is the key to the good functioning of our whole module, but turned out to be the greatest challenge when it comes down to computational efficiency and performance. As described above we are really dependent on a color corrected image by the *Anti Instagram* module. Throughout the whole project we planned to use their *continuous anti-instagram node* which is supposed to compute a color transformation in fixed intervals of time. However, when it came down we acutally had to change this for the follwing reason: The continuous anti-instagram node, running at an update interval of 10 seconds, consumes a considerable amount of computing power, namely 80%. In addition to that, the image transformer node which is in fact transforming the whole image and currently running at 4 Hz needs another 74% of one kernel. If you now run those two algorithms combined with the lane-following demo which makes the vehicle move and combined with our own code which needs an additional 75% of computing power, our safety critical module could only run at 1.5Hz and resulted in poor behaviour.

Even if you increase the time interval in which the continuous anti-instagram node computes a new transformation there was no real improvement. That is why in our final setup we let the anti-instagram node once compute a reasonable transformation and then keep this one for the entire drive. Through this measure we were able to safe the 80% share entirely and this allowed our overall node to be run at about 3 Hz with introducing an additional maximal delay of about 0,3 seconds. Nevertheless we want to point out that all the infrastructure for using the continuous anti instagram node in the future is provided in our package.

To sum up, the interface between our node and the Anti Instagram node was for sure developed very well and the collabroation was very good but when it came to getting the code to work, we had to take one step back to achieve good performance. That is why it might be reasonable to put effort into this interface in the future, to being able to more efficiently transform an entire image and to reduce the computational power consumed by the node which continuously computes a new transformation.

### 2) Evaluation of the Obstacle Detection

In general, since our obstacle classification algorithm is based on the rotational invariant feature of the eigenvalues of the inertia tensor it is completely invariant to the current orientation of the duckiebot and its position with respect to the lanes.

To rigorously evaluate our detection algorithm, we started off with evaluating **static scenes**, meaning the Duckiebot is standing still and not moving at all. Our algorithm performed extremely well in those static situations. You can place an arbitrary amount of obstacles, where the orientation of the respective obstacles does not matter at all, in front of the Duckiebot. In those situations and also combining them with changing the relative orientation of the Duckiebot itself, we achieved a false positive percentage of **below 1%** and we labelled all of the obstacles with respect to the lane boundaries correctly. The only static setup which is sometimes problematic is when we place the smallest duckies very close in front of our vehicle (below 4 centimeters), without approaching them. Then we sometimes cannot

detect them. However this problem is mostly avoided during the dynamic driving, since we anyways want to stop earlier than 4 centimeters in front of potential obstacles. We are very happy with this static behaviour as in the worst case, if during the dynamic drive something goes wrong, you can still simply stop and rely upon the fact that the static performance is very good before continuing your drive. In the log chapter it is possible to find the corresponding logs.

This in return also implies that most of the misclassification errors during our **dynamic drive** are due to the effect of motion blur, assuming a stable color transformation provided by the anti instagram module. E.g. in Figure 13.20 two line segments in the background "blurred" together for two consecutive frames resulting in being labelled as an obstacle.

Figure 13.20. Obstacle Detector Error due to motion blur

Speaking more about of numbers, we took 2 duckiebots at a gain of around 0.6 and performed two drives at different days, so also at different lights and the results are the following: Evaluating each picture which will be given to the algorithm, we found out that on average, we detect **97%** of all the yellow duckies in each picture. In terms of cones we detect about **96%** of all cones in the evaluated frames. We consider these to be very good results as we have a very low rate of false positives (**below 3%**).

| Date | #correctly detected duckies | #correctly detected cones | #missed ducks | #missed cones | #false positive | #false position |
|---|---|---|---|---|---|---|
| 19/12/2017 | 423 | 192 | 14 | 8 | 9 | 45 |
| Robot:Arki | | | 3,2% | 4% | 1,4% | 7,2% |
| Duration:82s | | | | | | |

| Date | #correctly detected duckies | #correctly detected cones | #missed ducks | #missed cones | #false positive | #false position |
|---|---|---|---|---|---|---|
| 21/12/2017<br>Robot:Dori<br>Duration:100s | 387 | 103 | 10<br>2,5% | 5<br>4,4% | 15<br>3% | 28<br>5,7% |

When it comes to evaluating the performance of our obstacle classification with respect to classifying them as dangerous or not dangerous our performance is not as good as the detection itself, but we did also not put the same effort into it. As you can see in the table above, we have an error rate of **above 5%** when it comes to determining whether the obstalce's position is inside or outside the lane boundaries (this is denoted as false position in the table above). We are especially encountering problems when there is direct illumination on the yellow lines which are very reflective and therefore appear whitish. Figure 13.21 shows such a situation where the current implementation of our obstacle classification algorithm fails.

Figure 13.21. Obstacle Detector Classification Error

### 3) Evaluation of the Obstacle Avoidance

Since at the current state we stop for every obstacle which is inside the lane and inside the bounding box, the avoidance process is very stable since it does not have to generate avoidance trajectories. The final performance on the avoidance is mainly relying on the placement of the obstacles:

**1. Obstacle placement on a straight:** If the obstacle is placed on a straight with a sufficient distance from the corner the emergency stop works nearly every time if

the obstacle is detected correctly.

**2. Obstacle in a corner:** Due to the currently missing information of the curvature of the current tile the bounding box is always rectangular in front of the robot. This leads to problems if an obstacle is placed in a corner because it might enter the bounding box very late (if at all). Since the detection very close to the robot is not possible, this can lead to crashes.

**3. Obstacles on intersection:** These were not yet considered in our scope but still work if the detection is correct. It then behaves similar to case 1.

Furthermore there a few cases which can lead to problems independent of the obstacle placement: **1. Controller oscillations:** If the lane controller sees a lot of lag due to high computing loads or similar its control sometimes start to oscillate. These oscillations lead to a lot of motion blur which can induce problems in the detection and shorten the available reaction time to trigger an emergency stop.

**2. Controller offsets:** The current size of the bounding box assumes that the robot is driving in the middle of the lane. If the robot is driving with an offset to the middle of the lane it can happen that obstacles at the side of the lane aren't detected. This however rarely leads to crashes because then the robot is just avoiding the obstacle instead of stopping for it.

While testing our algorithms we saw successfull emergency stops in 10/10 cases for obstacles on a straight and in 3/10 cases for obstacles placed in corners assuming that the controller was acting normally. It is to be noted that the focus was lying on the reliable detections on the straights, which we intended to show on the demo day.

## 13.7. Future Avenues of Development

As already described above in the eval interface section, we think that there is still room for **improving the interface** between our code and the *Anti Instagram* module in terms of making the *continouus anti instagram node* as well as the *image transformer node* more computationally efficient. Another interesting thought which might be taken into consideration concerning this interface is the follwoing: As long as the main part of the anti instagram's color correction is linear (which was in most of our cases sufficient), it might be reasonable to just adapt the filter values than to subscribe to a fully transformed image. This effort could save a whole publisher and subscriber and it is obvious that it is by far more efficient to transform a few filter values once than to transform every pixel of every incoming picture. Towards the end of our project we invested some time in trying to get this approach to work but as time was not enough we could not make it. We especially struggled to transform the orange filter values, while it worked for the yellow ones (BRANCH: devel-saviors-ai-tryout2). We think that if in the future one will stick to the current hardware this might be a very interesting approach, also for other software components such as the lane detection or any other picture related algorithms which are based on the concept of filtering colors.

Another idea of our team would be to **exploit the transformation to the bird's view also for other modules**. We think that this approach might be of interest e.g. for extracting the curvature of the road or performing the lane detection from the rather more undistorted top view.

Another area of improvement would be to further develop our provided scripts to

being able to **automatically evaluate** the performance of our entire pipeline. As you can see in our code description in github there is a complete set of scripts available which makes it easily possible to transform a bag of raw camera images to a set of pictures on which we applied our obstacle detector, including the color correction part of Anti Instagram. The only missing step left is an automatic detection whether the drawn box is correct and in fact around an object which is considered to be an obstacle or not.

Furthermore to achieve more general performance propably even adaptions in the hardware might be considered (see [37]) to tune the obstacle detection algorithm and especially its generality. We think that setting up a **neural network** might make it possible to release the restrictions on the color of the obstacles.

In terms of avoidance there would be **possibilities to handle the high inacurracies of the pose estimation** by relying on the lane controller to not leave the lane and just use a kind of closed loop control to avoid the obstacle (use the new position of the detected obstacle in each frame to securely avoid the duckie). Applying filters to the signals, especially the heading estimation, could further improve the behaviour. This problem was detected late in the development and could not be tested due to time constraints. Going further, having both the line and obstacle detection in the same algorithm would allow the direct information on how far away obstacles are from the track. We expect that this would increase the accuracy compared to computing each individually and bringing it together.

The infrastructure is in place to include new scenarios like obstacles on intersection or multiple detected obstacles inside the bounding box. If multiple obstacles are in proximity, a more sophisticated trajectory generation could be put in place to avoid these obstacles in a safe and optimal way.

Furthermore the **avoidance in corners** could be easily significantly improved if the line detection would estimate the curvature of the current tile which would enable adaptions to the bounding box oncorner tiles. If the pose estimation is significantly improved one could also implement an adaptive bounding box which takes exactly the form of the lane in front of the robot (see Figure 13.22)

Figure 13.22. Adaptive bounding box

## 13.8. Theory Chapter

### 1) Introduction to Computer Vision

In general a camera is consisting of a converging lens and an image plane (Figure

13.23). In the following theory chapter, we will assume that the picture in the image plane is already undistorted, meaning we preprocessed it and eliminated the lens distortion.

Figure 13.23. Simplyfied camera model [38]

It is quite easy to infer from Figure 13.23 that for a real world point to be in focus, it has to hold, that both of the "rays" (see Figure 13.23) intersect in one point in the image plane, namely in point B. Mathematically written this means:

$$1: \frac{B}{A} = \frac{e}{z}$$

$$2: \frac{B}{A} = \frac{e-f}{f}$$

$$\Leftrightarrow \frac{e}{f} - 1 = \frac{e}{z} \tag{1}$$

This last equation (1) can be approximated since usually $z \gg f$ such that we effectively arrive at the pin-hole approximation with: $e \approx f$ (see Figure 13.24)

Figure 13.24. Pinhole camera approximation [38]

For the pixel coordinate on the image plane it holds:

$\frac{h'}{h} = \frac{f}{z} \Leftrightarrow h' = \frac{f}{z} * h.$

In a more general case, when you consider a 3 dimensional setup and think of a 2 dimensional image plane you have to add another dimension and it follows that a real world point being at $\vec{P_W} = \begin{pmatrix} X_W \\ Y_W \\ Z_W \end{pmatrix}$ will therefore be projected to the pixels in the image plane:

$x_{pix} = \alpha * \frac{f}{Z_W} * X_W + x_{offset}$ and $y_{pix} = \beta * \frac{f}{Z_W} * Y_W + y_{offset}$

where $\alpha$ and $\beta$ are scaling parameters and $x_{offset}$ and $y_{offset}$ are constants which

can be always added. Those equations are usually rewritten in homogeneous coordinates such that we have only linear operations left as:

$$\lambda * \begin{pmatrix} x_{pix} \\ y_{pix} \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha * f & 0 & x_{offset} \\ 0 & \beta * f & y_{offset} \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} X_W \\ Y_W \\ Z_W \end{pmatrix}$$

$$\Leftrightarrow \lambda * \overrightarrow{P_{pix}} = H * \overrightarrow{P_W} \tag{2}$$

**Note:** In general this Matrix H is what we get out of the intrinsic calibration procedure and it might happen, that if the World frame and Camera frame are not completely aligned that then the (1,2) and (2,1) entry of the H Matrix are not exactly zero.

This equation **(2)** and especially Figure 13.24 clearly show that since in every situation you only know H as well as $x_{pix}$ and $y_{pix}$ of the respective objects on the image plane, there is no way to determine the real position of the object, since everything can only be determined up to a scale ($\lambda$). Frankly speaking you only know the direction in which the object has to be but nothing more, which makes it a very difficult task to infer potential obstacles given the picture of a monocular camera only. This scale ambiguity is illustrated in Figure 13.25.

Figure 13.25. Scale ambiguity [40]

To conclude, given a picture from a monocular camera only you have no idea at which position the house really is, so without exploiting any further knowledge it is extremely difficult to reliably detect obstacles which is also the main reason why the old approach did not really work. On top of that come other artifacts such as that the same object will appear larger if it is closer to your camera and vice versa, and lines which are parallel in the real world will in general not be parallel in your camera image.

**Note:** The intuition, why we humans can infer the real scale of objects is that if you add a second camera, know the relative Transformation between the two cameras and see the same object in both images, then you can easily triangulate the full position of the object, since it is at the place where the two "rays" intersect! (see Figure 13.26).

Figure 13.26. Triangulation to obtain absolute scale [41]

### 2) Inverse Perspective Mapping / Bird's View Perspective

The first chapter above introduced the rough theory which is needed for understanding the follwing parts. The important additional information that we exploited heavily in our approach is that in our special case we know the coordinate $Z_W$. The reason therefore lies within the fact that unlike in another more general usecase of a mono camera, we know that our camera will always be at height $h$ with repsect to the street plane and that the angle $\theta_0$ also always stays constant. (Figure 13.27)

Figure 13.27. Illustration of our fixed camera position [42]

This information is used in the actual extrinsic calibration such that in Duckietown, due to the assumption that everything we see should in general be on the road, we can determine the full real world coordinates of every pixel, since we know the coordinate $Z_W$ which uniquely defines the absolute scale and can therefore uniquely determine $\lambda$ and H! Intuitively this comes from the fact that we can just intersect the known ray direction (see Figure 13.24) with the known "gound plane".

This makes it possible to project every pixel back into the "road plane" by computing for each available pixel:

$$\vec{P_W} = H^{-1} * \lambda * P_{pix}$$

**This "projection back onto the road plane" is called inverse perspective mapping!**

If you now visualize this “back” projection, you basically get the bird’s view since you can now map back every pixel in the image plane to a unique place on the road plane.

The only trick of this easy maths is that we exploited the knowledge that everything we see in the image plane is in fact on the road and has one and the same z-coordinate. You can see that the original input image Figure 13.28 is nicely transformed into the view from above where every texture and shape is nicely reconstructed if this assumption is valid Figure 13.29. You can especially see that all the yellow line segments in the middle of the road roughly have the same size in this bird’s view Figure 13.29 which is very different if you compare it to the original image Figure 13.28.

Figure 13.28. Normal incoming image without any obstacle

Figure 13.29. Incoming image without obstacle reconstructed in bird's view

The crucial part is now what happens in this bird’s view perspective, if the camera sees an object which is not entirely part of the ground plane, but stands out. These are basically obstacles we want to detect. If we still transform the whole image to the bird’s view, these obstacles which stand out of the image plane get heavily disturbed. Lets explain this by having a look at Figure 13.30.

Figure 13.30. Illustration why obstacle standing out of ground plane is heavily disturbed in bird's view, modified: [42]

The upper picture in Figure 13.30 depicts the real world situation, where the cone is standing out ot the image plane and therefore the tip is obviously not at the same height as the ground plane. However, as we still have this assumption and as stated above intuitively intersect the ray with the ground plane, the cone gets heavily disturbed and will look like the lower picture in Figure 13.30 after performing the inverse perspective mapping. From this follows that if there are any objects which DO stand out of the image plane then in the inverse perspective you basically see their shape being projected onto the ground plane. This behaviour can be easily exploited since all of these objects are heavily disturbed, drastically increase in size and can therefore be easily separated from the other objects which belong to the ground plane.

Let's have one final look at an example in Duckietown. In Figure 13.31 you see an incoming picture seen from the normal camera perspective, including obstacles. If you now perform the inverse perspective mapping, the picture looks like Figure 13.32 and as you can easily see, all the obstacles, namely the two yellow duckies and the orange cone which stand out of the ground plane are heavily disturbed and therefore it is quite easy to detect them as real obstacles.

Figure 13.31. Normal situation with obstacles in Duckietown seen from Duckiebot perspective

Figure 13.32. Same situation seen from bird's perspective

### 3) HSV Color Space

***Introduction and Motivation:***

The "typical" color model is called the RGB color model. It simply uses three numbers for the amount of the colors *red*, *blue* and *green*. It is an additive color system, so we can simply add two colors to produce a third one. Mathematically written it looks as follows and shows the way of how we deal with producing new colors:

$$\begin{pmatrix} r_{res} \\ g_{res} \\ b_{res} \end{pmatrix} = \begin{pmatrix} r_1 \\ g_1 \\ b_1 \end{pmatrix} + \begin{pmatrix} r_2 \\ g_2 \\ b_2 \end{pmatrix}$$

If the resulting color is white, the two colors *1* and *2* are called to be complementary (e.g. this is the case for blue and yellow).

This color system is very intuitive and is oriented on how the human vision perceives the different colors.

The *HSV* color space is an alternative representation of the RGB color model. On this occasion *HSV* is an acronym for *Hue*, *Saturation* and *Value*. It is not so easy summable as the RGB model and it is also hardly readable for humans. So the big question is: **Why should we transform our colors to the HSV space? Does it derive a benefit?**

The answer is yes. It is hardly readable for humans but it is way better to filter for specific colors. If we look at the definition openCV gives for the RGB space, the higher complexity for some tasks becomes obvious:

*In the RGB color space all "the three channels are effectively correlated by the amount of light hitting the surface", so the color and light properties are simply*

*not separated.* (see: [44])

Expressed in a more simpler way: In the RGB space the colors also influence the brightness and the brightness influences the colors. However, in the HSV space, there is only one channel - the *H* channel - to describe the color. The *S* channel represents the saturation and *H* the intensity. This is the reason why it is super useful for specific color filtering tasks.

The HSV color space is therefore often used by people who try to select specific colors. It corresponds better to how we experience color. As we let the ***H (Hue)*** channel go from 0 to 1, the colors vary from red through yellow, green, cyan, blue, magenta and back to red. So we have red values at 0 as well as at 1. As we vary the ***S (saturation)*** from 0 to 1 the colors simply vary from unsaturated (more grey like) to fully saturated (no white component at all). Increasing the ***V (value)*** the colors just become brighter. This color space is illustrated in Figure 13.33. (see: [45])

Figure 13.33. Illustration of the HSV Color Space [45]

Most systems use the so called RGB additive primary colors. The resulting mixtures can be very diverse. The variety of colors, called the *gamut*, can therefore be very large. Anyway, the relationship between the constitutent amounts of red, green, and blue lights is unintuitive.

***Derivation:***

The *HSV* model can be derived using geometric strategies. The RGB color space is simply a cube where the addition of the three color components (with a scale form 0 to 1) is displayed. You can see this on the left of Figure 13.34.

Figure 13.34. Comparison between the two colors spaces [47]

You can now simply take this cube and tilt it on its corner. We do it this way so that black rests at the orgin whereas white is the highest point directly above it along the vertical axis. Afterwards you can just measure the *hue* of the colors by their angle around the vertical axis (red is denoted as 0°). Going from the middle to the outer parts from 0 (where the grey like parts are) to 1 determines the *saturation*. This is illustrated in Figure 13.35.

Figure 13.35. 'Cutting the cube' [48]

The definitions of *hue* and *chroma* (proportion of the distance from the origin to the edge of the hexagon) amount to a geometric warping of hexagons into circles (for more informations see: [48]). Each side of the hexagon is mapped linearly onto a 60° arc of the circle. This is visualized in Figure 13.36.

Figure 13.36. Warping hexagons to circles [48]

For the *value* or lightness there are several possibilities to define an appropriate dimension for the color space. The simplest one is just the average of the three components, which is nothing else then the vertical height of a point in our tilted cubic. For this case we have:

$$I = 1/3 * (R + G + B)$$

For another definition the *value* is defined as the largest component of a color. This places all three primaries and also all of the “secondary colors” (cyan, magenta, yellow) into a plane with white. This forms a hexagonal pyramid out of the RGB cube. This is called the HSV “hexcone” model and is the common one. We get:

$$V = max(R, G, B)$$

(see: ([48]))

*In Practice:*

**1.** Form a hexagon by projecting the RGB unit cube along its pincipal diagonal onto a plane.

Figure 13.37. First layer of the cube (left) and flat hexagon (right) [52]

**2.** Repeat projection with smaller RGB cube (subtract 1/255 in length of every cube) to obtain smaller projected hexagon. Like this a *HSV hexcone* is formed by stacking up the 256 hexagons in decreasing order of size.

Figure 13.38. Stacking hexagons together [52]

Then the value is again defined as:

$$V = max(R, G, B)$$

**3.** Smooth edges of hexagon to circles (see previous chapter).

*Application:*

One nice example of the application of the HSV color space can be seen in Figure 13.39.

Figure 13.39. Image on the left is original. Image on the right was simply produced by rotating the H of each color by -30° while keeping S and V constant [48]

It just shows how simple color manipulation can be performed in a very intuitive way. We can turn many different applications to good account using this approach. As you have seen, color filtering also simply becomes a threshold query.

# Unit N-14

# PDD - Navigators

## 14.1. Part 1: Mission and Scope

### 1) Mission Statement

The objective of this project is to implement a method that allows Duckiebots to reliably navigate any kind of intersection they may encounter when driving through Duckietown.

### 2) Motto

TRANSIBITIS

(You shall pass)

### 3) Project Scope

*What is in scope:*

- Navigating three- and four-way intersection of predetermined shape.
- (Absolute or relative) localization within the intersection.
- Computing a path or trajectory that guides the Duckiebots across the intersection to the desired lane.
- Computing control inputs to follow path / track trajectory.
- Limiting travel time across intersection.
- Detecting when the Duckiebot successfully navigated across an intersection and finds itself in a regular lane.
- Proposing hardware modifications to the intersection (e.g. traffic lights, additional markers,...).

*What is out of scope:*

- Understanding that the Duckiebot is at an intersection.
- Deciding where to go at an intersection.
- Coordinating with other Duckiebots at an intersection (e.g. who drives first, ...).
- Object detection and collision avoidance.
- Global localization.

*Stakeholders:*

- Smart Cities
- The Controllers
- Anti-Instagram
- The Identifiers

Figure 14.1. Project boundary

## 14.2. Part 2: Definition of the Problem

### 1) Problem Statement

We seek to find a method that allows a Duckiebot to safely navigate an intersection. In particular, we attempt to solve the following problem: Given that the Duckiebot finds itself at rest at an intersection, 1) devise a method such that the Duckiebot can localize itself at the intersection, 2) compute a path or trajectory that guides the Duckiebot to the desired exit while respecting the Duckiebots system dynamics and control input limitations, 3) find control inputs to track the path or trajectory, and lastly 4) detect when the maneuver is finished and the Duckiebot finds itself in a regular lane.

### 2) Assumptions

- Size, shape of intersections is given and fixed.
- Color and size of lane markings are given and fixed.
- The type of intersection (e.g. three-way or four-way intersection) as well as the desired exit (e.g. left turn, right turn or straight) are provided.
- There are fiducial markers (e.g. April tags, stop signs, ...) placed at the intersection at predetermined positions.
- The Duckiebots' are initially at rest and their pose is within a certain range with respect to the intersection (distance to center of road, distance to stop line, orientation within the lane).
- The intersection is free of obstacles.
- Good light conditions (e.g. no illumination problems,...)

### 3) Approach

The task of navigating an intersection can be roughly split into two tasks: 1) localization and 2) path/trajectory planning and control. The latter problem is assumed that be relatively easy compared to the first. Paths or trajectories can, for example, be found using simple motion primitives such as splines and it is then straightforward to track these using control techniques such as proportional-derivative-integral controllers. Hence, we only list different approaches to solve the localization problem when navigating intersections in the following.

**Open-Loop Maneuver:** Given that the Duckiebot finds itself at an intersection, it

simply executes a predetermined trajectory (from the nominal starting position to the desired end position) to cross the intersection. Once the regular lane detection is able to estimate the Duckiebots pose again, the control is handed back to the lane following controller. *Extension:* Model inaccuracies that are likely to affect the performance of the open-loop maneuvers could be compensated for by using iterative learning based on the state estimate of the lane detection method at the end of the open loop maneuver. If the lane detection algorithm finds that the duckiebot is far off of the nominal path, the nominal path could be adjusted iteratively such that systematic errors are suppressed (Note: This could to some extent interfere with the System Identification Project).

**April-Tags:** At each intersection in Duckietown, April-tags tell the Duckiebots at which intersection they are and what kind of intersection they need to cross (e.g. three-way intersection). The April-tags are placed at predetermined position at the intersection and are standardized. Given the (absolute) size of an april tag and the Duckiebot's camera parameters, one can compute the Duckiebot's position relative to the april tag and use this information to track a trajectory (again, relative to an april tag). *Extension:* The environment in Duckietown is very structured and various different objects, e.g. traffic lights or street signs, are placed at predetermined locations. To increase robustness of this method or to handle the case when the April tags are out of sight of the Duckiebots (likely towards the end of crossing an intersection), these objects could also be used for localization.

**Line Localization:** The disadvantage of using fiducial markers for localization is that it is not applicable to the real world where there are no fiducial markers. However, also there also exist in the real world standardized marks, the line marks on the street. One possible approach could thus be the detection of all the line marks visible at intersection (e.g. stop marks of current lane but also of the different exits) and use this for localization. Since the appearance of intersections are standardized in Duckietown, the homography that best explains the detected line marks could be determined in order to localize the Duckiebot.

**Visual Odometry:** The most generic solution would be the use of visual odometry for the localization of the Duckiebot at the intersection, i.e. matching of distinct features between subsequent camera frames and using these point matches to compute the relative camera pose between the two frames. Visual odometry in in general computationally expensive. However, the Duckiebot's dynamic could be used to reduce the computational burden (e.g. 1-point RANSAC). Due to a lack of sensors on the Duckiebot, visual odometry is unable to determine the scale of the scene. To solve this, the information about e.g. the size of April-Tags, line width, etc. can be used.

The above methods can also be combined, e.g. using visual odometry and if available April-tags.

### 4) Functionality provided

- Localization within an intersection.
- Path/trajectory planning to navigate intersection.
- Possibly control strategy to track above.

### 5) Resources required / dependencies / costs

The proposed method(s) can be evaluated using the following measures (in order

of decreasing importance):

- Success rate, i.e. the percentage of trials for which the Duckiebot ends up in the desired lane and successfully hands over the control to the lane following controller.
- Accuracy and precision of final state, i.e. how close is the Duckiebot's state relative to the desired final state (e.g. distance relative to the center of the lane, orientation within lane, velocity) and how repeatable is this.
- Duration, i.e. the average time required for the Duckiebot to cross an intersection and if possible an upper limit (worst-case) on the time required.
- Robustness to changes in the size of the intersection (e.g. lane width, size of tiles, ...), i.e. how do the above performance measures change with respect to changes of the size of intersection.
- Path following or trajectory tracking error, using for example the maximum absolute distance error from the desired path.

#### 6) Performance measurement

- The success rate can evaluated by simply performing the intersection navigation tasks N times and counting the number of successful trials.
- The accuracy and precision of final state can be evaluated using the already existing lane detection method.
- The average duration can simply be computed from N experiments. A (probabilistic) upper bound to navigating an intersection can be found likewise.
- Either experiments on the real system with slightly different intersections or a sensitivity analysis (analytically or numerically) with respect to parameters defining the size of the intersection can be carried to evaluate the system's robustness.
- An absolute position system (e.g. an overhead motion capture system) can be used to evaluate the Duckiebot's trajectory tracking errors.

## 14.3. Part 3: Preliminary Design

#### 1) Modules

- Initial position understanding
- Information processing (in which kind of intersection we are, where we want to go, ..)
- Trajectory generation
- Control loop
- Conclusion

#### 2) Interfaces

The inputs and outputs are also shown on a very high level in the Stakeholders Graph.

#### 3) Software modules

- Intersection Localization: ROS node
- Path Planner / Trajectory Generator: Python library
- Path Following / Trajectory Tracking Controller: ROS node.

#### 4) Infrastructure modules

None (assuming that no hardware changes are required).

## 14.4. Part 4: Project Planning

#### 1) Data collection

Recordings of the camera feed of Duckiebots navigating intersections.

#### 2) Data annotation

No

*Relevant Duckietown resources to investigate:*

- Current (open-loop) solution
- April tag detector
- Control strategy
- Lane Detector
- Anti-Instagram
- State Machine (switch between lane-following and navigating intersection)
- Visual Odometry
- Duckiebot system dynamics and control input constraints

*Other relevant resources to investigate:*

- April Tags: https://april.eecs.umich.edu/software/apriltag.html
- Aruco Markers: https://sourceforge.net/projects/aruco/files/
- Visual Odometry: D. Scaramuzza, F. Fraundorfer, "Visual odometry [tutorial]", IEEE Robotics & Automation Magazine, 2011.

D. Scaramuzza, F. Fraundorfer, R. Siegwart, "Real-time monocular visual odometry for on-road vehicles with 1-point RANSAC", IEEE International Conference on Robotics and Automation, 2009.

- Localization using Line Detection: J. Barandiaran, D.Borro, "Edge-Based Markerless 3D Tracking of Rigid Objects", IEEE Conference on Artificial Reality and Telexistence, 2007.

#### 3) Risk Analysis

- Not enough distinct features for visual odometry.
- Similarly, not enough lane markings.
- Not sufficient computational power on board the Duckiebot.

*Mitigation strategies:*

Mainly the mentioned risks will be related to a closed-loop implementation, which is a basic goal of the project. However they should not affect the development of an improved open-loop solution, ex. using April Tags detection, so this may be a starting point to improve the current state and to start the development of the closed-loop solution.

For the application of a Visual Odometry algorithm, if features in the Duckietown will be not enough, we may plan to add/use additional hardware, which will be possibly developed in accord with the Smart CIties group.

Moreover, the paper “Real-time monocular visual odometry for on-road vehicles with 1-point RANSAC” may be useful for addressing issues about computational constraints thanks to the implementation of the 1-point RANSAC algorithm.

UNIT N-15

# The Navigators: intermediate report

## 15.1. Part 1: System interfaces

### 1) Logical architecture

Figure 15.1. Simple step diagram

The intersection navigation is started as soon as the Duckiebot is told that it is in front of an intersection. The following functions are then executed (in chronological order):

- The Duckiebot localizes itself with respect to the intersection.
- The Duckiebot waits until it receives a message on the topic “turn_type” which exit of the intersection it should take.
- A path is planned that guides the Duckiebot from its current location to the desired intersection exit.
- A path following controller steers the Duckiebot to its final location, while the Duckiebot continuously localizes itself and feeds the estimated pose (i.e. the distance from the desired path and the relative orientation error) to the lane following controller to account for, for example, disturbances or modelling errors.
- The Duckiebot detects when it traversed the intersection, i.e. when it finds itself again in a regular lane, and hands control back to the lane following controller by publishing on the topic “intersection_done”.

It is assumed that

- the Duckiebot stops between 0.10m and 0.16m in front of the center of the red

stop line, i.e. $d_x \in [0.1m, 0.16m]$, has an error of no more than 0.03m with respect to the center of its lane, i.e. $d_y \in [-0.03m, 0.03m]$, and that the orientation error is smaller than 0.17rad, i.e. $\theta \in [-0.17rad, 0.17rad]$ (see Fig. 1.2 for details, all values are with respect to the origin of the Duckiebot's axle-fixed coordinate frame).

- a lane following controller exists that takes as inputs the distance from desired path $d$ and and the orientation error with respect to the path tangent $\theta$ (see Fig. 1.3 for details).

Figure 15.2. Pose of the duckiebot in front of an intersection

Figure 15.3. Pose of the duckiebot relative to the desired path

## 2) Software architecture

We will develop two nodes; *"intersection_navigation"* and *"intersection_localization"*. In the following, their functionality and interfaces will be described in more detail.

**"intersection_navigation"-node**

The *"intersection_navigation"*-node is responsible for the high level logic of navigating the Duckiebot across an intersection, planning paths from the Duckiebot's initial position to the final position, estimating the Duckiebot's pose and communicating with the lane following controller. It subscribes to the following topics:

- "mode": Used to detect when Duckiebot is at an intersection or when the intersection control is active, respectively. No assumptions about the latency of this topic will be made. As soon as the mode is switched to "INTERSECTION_CONTROL", the *"intersection_navigation"*-node will take over.
- "turn_type": Tells the Duckiebot the type of turn it should take (e.g. left, right, straight, random). No assumption about the latency of this topic will be made. As soon as message is received, the maneuver will be executed.
- "intersection_pose_meas_inertial": Measured pose of the *"intersection_localization"*-node with respect to an inertial frame $\mathcal{I}$ (see Fig. 1.4). This message is used to estimate the pose of the Duckiebot at the intersection, which is then used by the controller to follow the desired pose. It is assumed that this message will have quite some delay (several 10ms), but the delay will be compensated by a state estimator using the timestamp of the message (i.e. camera frame) and using the past commands sent to the vehicle.
- "~image/compressed": Upon receiving such a message, the Duckiebot's pose at the time the image was taken will be estimated and sent to the *"intersection_localization"*-node to initialize the localization problem.
- "~car_cmd": The command published by the "lane_controller"-node used to track a desired path. These commands are stored in a queue inand will be used to compensate for delays and predict the Duckiebot's pose. It is assumed this topic has no delay, i.e. that commands are immediately executed.

The "intersection_navigation"-node publishes on the following topics:

- "intersection_done": A message on this topic will be broadcasted as soon as the Duckiebot finished traversing the intersection and is used to handback the control.
- "intersection_pose": Pose of the Duckiebot with respect to the desired path (see Fig. 3). This topic is basically identical to the "~lane_pose"-topic from the lane filter and will be used by the *"lane_controller"*-node in case "mode" is "INTERSECTION_CONTROL".

The "intersection_navigation"-node will be estimated to introduce no more than 500ns of delay in regular operation (it only does some logic) and hence an equal delay will be introduced to all the published topics. Initially, after the "intersection_localization"-node is initialized (see below), a path that guides the Duckiebot across the intersection will be planned. This task can be computationally expensive since it needs to be guaranteed that the path is feasible (e.g. not leaving the intersection, curvature constraints, ...) and may take up to 500ms. However, since the Duckiebot is at rest at the intersection, this will not cause any issues.

**"intersection_localization"-node**

The *"intersection_localization"*-node is responsible for localizing the Duckiebot at an intersection. For this purpose, it subscribes to the following topics:

- "mode": This topic is used to detect when the node should start localizing itself at an intersection. No assumption about its latency is made, since it is irrelevant for the node. As soon as the mode is switched to "INTERSECTION_CONTROL", the *"intersection_localization"*-node will start to estimate the Duckiebot's pose relative to the

intersection.

- "~image/compressed": The compressed camera image is used to localize the Duckiebot within the intersection. No assumption about the latency of this topic is made. In order to compensate for the expected latency, the timestamp of the camera frame will be also be used to indicate the time for which the pose is estimated.
- "intersection_pose_pred_inertial": The predicted pose of the Duckiebot at the time when the camera image was taken. This information will be used to initialize the localization problem this node solves. Since the Duckiebot's pose is predicted for the time the camera image was taken, delays are irrelevant.

The *"intersection_localization"*-node publishes the following topic:

- "intersection_pose_meas_inertial": This is the measured pose of the Duckiebot at the intersection with respect to an inertial frame $\mathcal{I}$ based on the received camera image. The measured pose will be timestamped with the timestamp of the camera image such that the *"intersection_navigation"*-node can compensate for the latency.

It is estimated that it will take approximately 15ms to estimate the Duckiebot's pose once the camera image is received, hence about 15ms of delay can be expected on the published topics. However, the delay will be compensated for by the *"intersection_navigation"*-node.

Figure 15.4. Pose of the duckiebot inside a four way intersection

## 15.2. Part 2: Demo and evaluation plan

### 1) Demo plan

The proposed closed-loop intersection navigation method will be demonstrated by placing the Duckiebot at an intersection and commanding it via joystick to traverse the intersection to a desired exit (the arrows on left hand side of the joystick will be used to enter the desired exit). The Duckiebot must be placed in a lane in front of the red stop line before executing the demo. The relative position within the lane and its orientation can vary (the initial pose must satisfy the assumption of Part 1). The demo takes as input arguments the desired exit and the navigation method, i.e. the proposed closed-loop navigation or the already existing open-loop navigation. Both arguments are optional.

In addition to the specific intersection navigation demo, the proposed closed-loop intersection navigation will be embedded in the indefinite navigation demo. The indefinite navigation demo can be run with two Duckiebots, one using the proposed closed-loop intersection navigation method and one using the existing open-loop intersection navigation method. In order to be able to distinguish the two Duckiebots, we suggest driving around with a blindfolded duck for the closed-loop method and with a duck that has its eyes wide open for the open-loop method, respectively. However, this demo will require more time to run and differences between the open-loop and closed-loop navigation (e.g. their sensitivity to the initial position at the intersection) will be harder to see.

Both demos can be run in a any Duckietown that contains at least one intersection, hence the hardware for a regular Duckietown is required (http://purl.org/dth/fall2017-map).

### 2) Plan for formal performance evaluation

The performance of the intersection navigation is evaluated experimentally using the following measures (in order of decreasing importance):

- Success rate, i.e. the percentage of trials for which the Duckiebot ends up in the desired lane and successfully hands over the control to the lane following controller. A trials is considered to be successful if the Duckiebot is completely inside the desired lane without touching any lane markings. The success rate is evaluated by simply performing the intersection navigation task N times and counting the number of successful trials.
- Accuracy and precision of final state, i.e. how close is the Duckiebot's state relative to the desired final state (e.g. distance relative to the center of the lane, orientation within lane, velocity) and how repeatable is this. The accuracy and precision of final state is estimated using the already existing lane detection method, and is measured for different initial conditions.
- Accuracy and precision of estimated pose during traversing the intersection. The estimated pose will be logged and an external motion capture system, which directly outputs the pose of the Duckiebot, will be used as ground-truth.
- Duration, i.e. the average time required for the Duckiebot to cross an intersection and if possible an upper limit (worst-case) on the time required. The average duration is computed by running a series of N experiments. A (probabilistic) upper bound to navigating an intersection is found likewise.

## 15.3. Part 3: Data collection, annotation, and analysis

### 1) Collection

The use of the provided platform for data collection, annotation and analysis is not needed since we are using logs and recordings of the camera feed of Duckiebots navigating intersections.

### 2) Annotation

No data will be annotated.

### 3) Analysis

No dataneeds to be analyzed.

UNIT N-16

# PDD - Parking

## 16.1. Part 1: Mission and scope

### 1) Mission statement

Implement parking feature and design specifications (feature and physical)

### 2) Project scope

Implement parking feature and design specifications (feature and physical)

*What is in scope:*

- forward parking
- bot localization with april tags
- open spot localization
- path generation (coming in and out of parking space)
- how to drive backwards
- parking lot full signal
- physical parking lot design specification

*What is out of scope:*

- Develop new algorithms to filter new lane line colors
- Fleet level coordination
- Handle multiple parking events at once

*Stakeholders:*

- Single SLAM or distributed-est
- Controls
- Smart City
- Anti Instagram

## 16.2. Part 2: Definition of the problem

### 1) Problem statement

We need to park N Duckiebots in a designated area in which they are able enter and exit in an efficient manner.

### 2) Assumptions

- Four tile structure with defined inlet, outlet and color scheme
- Known design specification of parking lot
- Assume when leaving parking space, path is free of other Duckiebots
- When entering lot and searching for parking space, parking lot is in static state

### 3) Approach

- Enter parking lot in designated inlet lane
- Localize based on april tags within field of view with known locations

- Control with feedback along predetermined path
- Detect parking space status (full/free) of each parking space in sequential manner
- Locate a free parking space
- Paths generated for maneuvering into parking space
- High fidelity control into parking space
- Signal generated to signify parking space is full
- When we want to leave a space, generate path out of parking space
- High fidelity control out of parking space (with caster wheel dynamics taken into account for feedback control)
- Control with feedback along predetermined path
- Exit parking lot in designated outlet lane

#### 4) Functionality provided

- Probability of a successful parking maneuver per parking maneuver attempt
- Number of Duckiebots within the parking lot boundary per hour

#### 5) Resources required / dependencies / costs

- Size of parking space
- Resources required to develop Duckiebot trajectory
- Number of april tags and infrastructure to support april tags

#### 6) Performance measurement

- Starting at parking lot entrance, measure the number of parking maneuvers completed within boundaries of designated parking spot (over N attempts)
- Starting in designated parking space, measure the number of Duckiebots able to arrive at the exit of the parking lot (over N attempts)
- Average time (for N vehicles) to enter and exit parking lot

## 16.3. Part 3: Preliminary design

#### 1) Modules

*Perception:*
- Lane filtering
- April tag detection
- “Fleet communication”: detecting

*Localization and parking map generation:*
- Ego localization
- Localization other Duckiebots
- Parking map design

*Planning:*
- Parking space allocation
- Path planning

*Control:*
- “Fleet communication”: publishing

#### 2) Interfaces

*Perception:*

**Lane filtering**

- Used for pose estimation at entrance and exit of parking lot and maybe at parking space
- Input: camera image
- Output: location lanes

**April tags detection and triangulation**

- Use for pose estimation while driving on the parking lot when no lanes can be identified, every parking space has its own april tag, relative Duckiebot-tag pose is extracted using computer vision
- Input: camera image
- Output: location of april tag, relative position Duckiebot-tag

**"Fleet communication": detecting**

- Blinking LEDs are used for communication: while parking signal who is driving (one at the time), while parked signal which parking lot is taken (Duckiebot on parking space 2, blink led in parking space 2 specific frequency)
- Input: camera image
- Output: other occupied signals (other means blinking signals is not from own Duckiebot)

*Localization and parking map generation:*

**Localization other Duckiebots**

- Determines the pose of other Duckiebots using specific blinking LEDs
- Input: other occupied signal(s)
- Output: pose other Duckiebot(s)

**Parking map design**

- Static map (physical known map with defined parking spaces and areas to move to them, without Duckiebots) is merged with pose of other Duckiebots to generate a {occupied, free} map of the parking lot
- Input: static map (has to be defined offline), pose other Duckiebots
- Output: parking map

**Ego localization**

- State estimation of position (x, y) and heading (theta) of own Duckiebot using lanes (at entrance/exit of parking lot) and april tags
- Input: parking map, relative position Duckiebot-tag(s), localization april tag(s), location lane(s)
- Output: pose Duckiebot

*Path planning:*

**Parking space allocation**

- Allocates a parking space to the Duckiebot given the parking map and the authority to move, executed once per Duckiebot
- Input: parking map, pose Duckiebot
- Output: pose parking space (x, y, theta)

**Path planning**

- The actual path planning module
- Input: pose parking space, pose Duckiebot, parking map
- Output: reference path or reference trajectory

*Control:*

- High fidelity control algorithm to drive Duckiebot on reference trajectory/path to allocated parking space, flag when parked
- Input: reference path, pose Duckiebot, (maybe parking map → constrained control)

- Output: motor voltage, parking status = {going to parking space, parked, want to leave, exiting parking space}

*"Fleet communication" publishing:*
- Flag status using specific frequency on LEDs (or color for human eye)
- Input: parking status
- Output: own_occupied_signal

### 3) Preliminary plan of deliverables

- Need: infrastructure, localization algorithm using april tags (maybe fusion with lane detection), high fidelity control algorithm, map generation algorithm, localization (ego and other Duckiebots), parking space allocation, path planning algorithm
- Exists: Lane detection, color filters, lane control, LED communication, april tag detection, control algorithm (maybe has to be improved),

### 4) Specifications

Yes, we need to add parking lot specifications.

### 5) Software modules

A collection of ROS nodes.

### 6) Infrastructure modules

Yes, we will include infrastructure modules to specify parking lot specifications.

## 16.4. Part 4: Project planning

### 1) Data collection

- April tag localization data
- April tag distance data (detection in a range of ~10 cm until ~1 m away from sign)
- Duckiebot to Duckiebot communication using flashing LEDs

### 2) Data annotation

No

*Relevant Duckietown resources to investigate:*
- April tag detection and localization (what is done already?)
- Control algorithm with good enough precision
- Transforming pose to configuration space
- Path planning algorithm (RRT*)
- Driving backwards (together with the control guys) while updating the pose of the Duckiebot

*Other relevant resources to investigate:*
- Transforming pose to configuration space
- Path planning algorithm (RRT*)

### 3) Risk analysis

- Localization fails while driving backwards

- Traffic jam at entrance of parking lot
- Fleet communication fails: incoming Duckiebot does not see currently parking Duckiebot, two Duckiebots leave at the same time
- Detection of april tags and extracting pose of the robot
- Map generation is wrong if Duckiebot is not parked to specification
- Control: level of precision adequate for parking
- Exit parking maneuver conflicts: who can drive first (Duckiebot which is exiting does probably not see anything)

# Unit N-17
# Parking: intermediate report

## 17.1. Part 1: System interfaces

### 1) Logical architecture

Description of the functionality. What happens when we click start?

- As soon as you arrive to the parking lot and see the corresponding entrance april tag, the Duckiebot switches from normal driving mode to parking mode. Parking mode is only allowed in the parking lot. If the bot exits the parking lot, it sees another april tag and it switches from parking mode back to normal driving mode (starting at a four way intersection).
- Inside the parking lot the robot estimates his pose (x, y and theta) using a bunch of april tags. There is one (maybe also two) april tag per parking space and some additional (entrance, exit, etc.) To do so, a new state estimation algorithm has to be implemented using the library 'AprilTags C++'. It estimates the relative position of the robot with respect to the april tag. The location of the april tag is encoded in the QR code. As soon as you see one (better two) tags, the pose can be calculated. We assume that we always see at least one tag.
- Given a prior information about the parking lot (where are the parking spaces, where can the robot drive etc) and real time vision information the robot chooses a parking space. At first we assume that the parking lot is empty or that other Duckiebots are static (do not move) and this is encoded in the parking map (places where the robot is not allowed to drive).
- We use RRT* to generate a path given the pose of the robot, the pose of the assigned parking space and the parking map. To do so we use the 'open motion planing library (OMPL)'.
- We control the robot to the optimal path using a sufficient controller using visual feedback.
- For driving to the exit, we generate a path and control our robot to this path which includes driving backwards to leave the parking space and turn to get to the exit in a forward motion.

Target values:

- accuracy: the error is a combination of localization accuracy and the offset due to the maximum allowable controller error. To park two Duckiebots next to each other within the space boundaries, the path planning accuracy has to be less (or equal) than 5 cm (which is the distance from the robot edge to the parking lane)
- the point of the robot which is the furthest away from the parking mid line should be less than half of the parking space width while the heading of the robot must be less than a constant (20 degrees) relative to the parking space boundary lines.

Assumptions about other modules: - we assume that the robot finds itself at the entrance of the parking lot whenever it wants to get a parking space

- once in the parking lot: parking is decoupled from everything else

## 2) Software architecture

`rosnode list`:

- someone
  - publishes: driving_mode
- /vehicle/parking_perception_localization
  - subscribes: driving_mode, camera_image,
  - publishes: parking_mode, space_status, pose_duckiebot, ,
- /vehicle/parking_path_planning
  - subscribes: parking_mode, pose_duckiebot, space_status
  - publishes: reference_for_control, (path)
- /vehicle/parking_control
- we copy this node from 'the controllers'
- subscribes: reference_for_control
- publishes: motor_voltage
- /vehicle/parking_LED
- subscribes: parking_mode, space_status
- publishes: -

`rostopic list`: - /vehicle/driving_mode - values = {driving, parking} - frequency: ~ 1 Hz

- /vehicle/parking_mode
  - values = {parking, staying, leaving, observing}
  - frequency: ~ 1 Hz
- /vehicle/space_status
  - 1xN array, N = number of parking space
  - values = {taken, free, not_detectable, my_parking_space}
  - frequency: ~ 1 Hz
- /vehicle/pose_duckiebot
  - x,y,theta
  - frequency: inherit from camera_image (~30 Hz)
- /vehicle/path
  - x,y,theta array
  - frequency: very low - only updated once (if my_parking_space = {1:3}) or twice (for my_parking_space = {4:6}, first path is to go to the middle and observe which parking spaces are free, second path is to go go the associated parking space)
  - computation time ~ 10 s
- /vehicle/reference_for_control
  - d (orthogonal distance to path), c (curvature), phi (differential heading path and Duckiebot)
  - frequency: first step (path generation) uses a lot of time ~ 10 s, afterwards fast (~ 30 Hz)
- /vehicle/motor_voltage
  - two values for the two motors
  - frequency: fast (~ 30 Hz)

Introduced latency to other modules:

- we need some additional lines for the april tag detection in order to switch the driving_mode to parking (this needs a new publisher in this node) –> latency

should be negligible

- otherwise we do not introduce delay to other modules since parking is decoupled

## 17.2. Part 2: Demo and evaluation plan

### 1) Demo plan

The parking feature including the design specification is new and will be implemented from scratch. Therefore, the desired functionality cannot be compared again a past version. The demo will be split in multiple parts.

In a first demo just a single Duckiebot will enter the parking lot and take the first parking space, knowing beforehand that this space will be free. The Duckiebot is supposed to park within the marked parking space and be able to leave the parking space after a given terminal command.

In a second demo, the Duckiebot will park in space 5 or 6. These are the spaces which are not visible from the beginning. Its need a two stage path panning. 1) Driving from the entrance to the middle of the parking lot and observe if parking spaces 4 to 6 are free. 2) Driving from the middle of the parking lot to space 5 or 6. The demo is completed when the duckie successfully drives to the exit after a given terminal command.

In a later step multiple Duckiebots (one by one) will enter the parking lot. The Duckiebot will first search for an available parking space (out of max. six), undertake a parking maneuver like in the first demo. The Duckiebot will signal the other Duckiebots that the parking space is taken by outputting a sequence with the LEDs at the back. After a given time each robot (one by one) will leave the parking lot again.

Needed hardware components:

- parking lot setup including April tags
- lanes marking the parking spaces
- Duckiebots (obstacles at parking spaces)

### 2) Plan for formal performance evaluation

The performance evaluation will be experimental. We will repeat the demo multiple times and measure the probability of a successful parking maneuver.

## 17.3. Part 3: Data collection, annotation, and analysis

### 1) Collection

We will need april tag data to understand at what distances and angles we can localize the Duckiebot from.

We will collect data at a distance up to 1.2m at increments of 10cm. We will collect data at an angle up to 40 degrees at increments of 10 degrees. We will do an “angle sweep” at each distance interval

Logs are taken manually

# Unit N-18
# Explicit Coordination: preliminary report

## 18.1. Part 1: Mission and scope

### 1) Mission statement

Coordinate intersection navigation safely through explicit communication.

### 2) Motto

IN HOC SIGNO VINCES

(with this sign, you will win)

### 3) Project scope

Employ LEDs based communication to efficiently manage traffic at intersections. First, LEDs should be used to reliably communicate Duckiebots positions and/or intentions. Then, optimal control theory or game theory might be considered to safely clear the intersection in reasonable time. By safely, we mean that no collisions occur (100% success). This requires robustness in message interpretation.

We aim to solve the problem of clearing the intersection in max. 60 seconds (meaning that even in the case of four Duckiebots participating we can solve the problem below this time). A decentralized solution should be implemented.

*What is in scope:*

The following aims are identified:

- List the messages that need to be exchanged between Duckiebots;
- Encode the messages in LED language;
- Produce the LED signal;
- Detect the LED signal;
- Decode the LED signal;
- Act safely upon the received information.

*What is out of scope:*

The following aims are beyond the scope of the project:

- Changing the communication protocol, i.e., we will stick on using LEDs.

*Stakeholders:*

The following pieces of Duckietown will be interacting with the project:

- Fleet planning;
- Implicit-navigation (determinate where the Duckiebot in the row has to stop);
- Anti-instagram filtering;
- Integration heros;

- Map designers;
- Traffic navigation;
- Controllers.

## 18.2. Part 2: Definition of the problem

### 1) Mission

Duckiebots must be able to cross an intersection in the defined reasonable time and without collisions.

### 2) Problem statement

The following problems are to be tackled:

- LED-based communication;
- Coordination at intersections.

### 3) Assumptions

The following assumptions are made for the LEDs communication:

- The Duckiebot is of type DB17-l.
- One to four Duckiebots are at the intersection with a certain position and orientation with respect to the stop line (projection of Duckiebot on 2D lane):
  - Min. 0 cm behind red line;
  - Max. 6 cm behind red line;
  - Max. +/- 2 cm from center of the line;
  - +/- 10° of rotation.
- Duckiebots are able to see the vehicles in front and on the right with respect to their position.
- LEDs work properly and emit the signals with the right color and frequency, and we can also detect the LEDs as the correct state that they represent, and attribute those LEDs to the correct Duckiebot that is displaying the LEDs.
- The Duckiebots do not move while "waiting" in the intersection, but they can move on the spot to look around to see the left.
- Camera works properly (frequency 30Hz, resolution of 64x48).

The following assumptions are made for the coordination:

- Signals are correctly recognized and associated to the corresponding messages.
- The intersection is among one of the standard intersections of Duckietown. That is, weird intersections will be ignored (at least in a first approach to the problem).
- The intersection type and the presence of a traffic lights are known.
- Intersection navigation is guaranteed to be working safely.
- One Duckiebot navigates the intersection at the time.

### 4) Approach

Firstly, the existing code and see its limits should be tested. Benchmark tests have to be designed in order to carefully measure the performances of the implementation. Then, the literature should check to see what has already been implemented.

The problem can be splitted in LED-based communication and intersection co-

ordination. We further distinguish between intersections with traffic lights and intersections without traffic lights (recall that it is assumed that the Duckiebots know the intersections they entering). In all cases we aim for a decentralized solution.

Possible approaches for LEDs communication:

- Last year's algorithm;
- Alternative approach 0.

Possible approaches for intersection coordination without traffic lights:

- Last year's approach;
- Alternative approach 1;
- Alternative approach 2;
- Alternative approach 3.

Possible approach for intersection coordination with traffic lights:

- Detect the signal of the traffic light and behave accordingly.

***Approaches for communication :***

*Last year's approach*: Communication works through LEDs blinking at different frequencies. Colors are just for human understanding and are operational meaningless. See last year's documentation for further details.

*Alternative approach 0*: Communication works through LEDs blinking at different frequencies, colors, and patterns.

***Approaches for coordination without traffic lights :***

*Last year's approach*: See last year's documentation.

*Alternative approach 1*:

Assumption: every Duckiebot does not see on the left. Here, Duckiebots do not need to turn in place to see the Duckiebot on the left

There are two messages:

- One signal to say if the Duckiebot can see someone on its right (e.g., a red LED blinking, signal A);
- One signal to say whether the Duckiebot is at intersection or is about to go (e.g., same LED blinking at different frequencies, signal B).

The coordination plan works as follows:

1. If Duckiebot:
   a. does not see signals from the right and
   b. it does see signal A from the opposite side of intersection. Then it will enter the intersection. That could happen with 1, 2, 3 Duckiebot at intersection.
2. If 4 Duckiebots are at intersection, that means, every Duckiebot is emitting signals A and B, each Duckiebot turns off with a certain probability (say 25%), so that one is going to see nobody on the right and will navigate the intersection.
3. If one Duckiebot does not see any kind of signal (A or B) it will proceed to navigate the intersection.
4. One case still need to be analyzed, i.e., the one in which we have 2 Duckiebots one opposite to the other. In this case:
   a. The Duckiebot does not see signals from the right and
   b. Does not see signal A from the opposite Duckiebot (but it does see signal B).

Then each Duckiebot turns off with some probability (say 50%) so that it will see

no more signals and then navigate the intersection, as explained in case 3.

*Alternative approach 2*: There are four messages:

- No color: waiting to enter the red queue;
- Red: waiting to enter the negotiation;
- Yellow: about to enter the intersection but still looking around, can go back to green;
- Green: negotiating.

In the following, let you be a Duckiebot.

1. You currently have no color. You stop at the intersection, turn left 20° in order to see all existing bots. You have no lights on at the moment. Possible scenarios:
   - If there are cars with yellow or green colors, you turn red. You will start negotiation after current negotiation ends and Duckiebots involved in the current negotiation pass the intersection.
   - If there is no green or yellow lights, but there is at least one Duckiebot with red color, you wait until you see a green or yellow color. You turn red. You go to 2 a).
   - If no lights are present, you turn yellow and wait for t seconds. If after t seconds there are still no lights on or all lights are red, you enter the intersection.
   - If there is at least one vehicle with yellow lights, you turn green (you need to negotiate with yellow), go to 2 b).

2. 1. You are currently red. Wait until yellows and/or greens are gone (assuming we also know their spots: opposite, left, right), check if there are any other reds: * If no reds are present, turn yellow and enter the intersection (you can execute immediately because you were red in the beginning of this step so if there are no reds when you checked, Duckiebots should be waiting for you o turn yellow or green so that they can turn red, therefore there cannot be a synchronization problem here). * If reds, turn green and go to 2 b) (if some reds turned green too quickly so that one red was unable to catch the red color, it still knew the spots of greens/yellow that existed in the beginning of 2 a), so if you see greens in the new spots can, you can still turn green and go to 2b). 2. You are currently green. Wait for random (0.2, 1) seconds: * If there exists a yellow negotiator wait for t seconds and repeat 2 b). * Else (if all negotiators show green), turn yellow, wait for t seconds. * If still all negotiators show green, enter the intersection. * If there is a yellow negotiator wait for t seconds and repeat 2 b).

(after the above algorithm, we could also implement the below to increase throughout)

Back lights: If you are entering the intersection, turn back lights to green. Else (negotiating, or waiting), turn off back lights.

Duckiebots behind a Duckiebot that stopped: If see a green back light, get ready to run, turn front lights to yellow and immediately enter the intersection after the car in front of you. If no green light on the back of the Duckiebot in front of you, stop at the red line as usual and start executing the intersection algorithm.

Improvements: We could possibly give frequency to back lights to let more than 2 Duckiebots enter the intersection, such that the first Duckiebot that is about to enter the intersection has constant green lights on the back, 2nd one behind it has green back lights with frequency x, and 3rd behind 2nd one knows it is the 3rd since it sees green light with frequency so it also enters the intersection immediately (and has lights off on the back so that 4th has to stop at the red line and exe-

cute the default intersection algorithm).

Therefore, this alternative needs 3 different signals, such as green, red, yellow and it will also use the no color case as a 4th signal.

*Alternative approach 3*:

This approach consists of is an exponential backoff model (assuming that two other visible cars is a low enough number to have it run quickly enough). Here:

- No turning is needed.
- If you see a Duckiebot at the right, give it "right of way".

The intersection policy works as follows:

START:

- go to CHECKING (blue)

CHECKING:

- if a bot is "out of place" (in the intersection)
  - go to WAIT (red)
  - go to CHECKING
- if no Duckiebots in CHECKING,
  - GO (green)
- if Duckiebot at right or front is CHECKING
  - go to WAIT (red)
  - go to CHECKING

WAIT:

- exponential backoff

## 5) Functionality-resources trade-offs

*Functionality provided:*

Metrics for LEDs communication:

- Maximize percentage of success in detecting a LED light or LED blinking in a picture taken from the Duckiebot camera.
- Maximize percentage of failed attempt of communication in a Duckietown intersection.
- Minimize time needed in order to detect signals.

Metrics for coordination:

- Maximize the times the intersection is cleared safely, i.e., without crashing.
- Minimize the time needed to clear the intersection for each Duckiebot and for the fleet.
- Maximize the number of successful intersections cleared safely below a threshold time (which may depend on the intersection itself and on the number of Duckiebot at the intersection).

*Resources required / dependencies / costs:*

The following resources are needed in order to test the behaviour:

- Four Duckiebots are needed to test the coordination.
- A Duckietown intersection.
- Traffic lights.

*Performance measurement:*

Performance measurements for LEDs communication:

- We put one Duckiebot in a lane at the intersection, one Duckiebot on the opposite lane across the intersection, and then one on the right. First the opposite and then the right Duckiebot will emit signals, the observer will receive them. We will see what happens if we let LED-detector-node run. We should be able to detect in which regions are LEDs blinking (See f23-presentation, Minute 11:20 in Google Drive).
- We put four Duckiebots at an intersection (with and without traffic light) and let them communicate. This allows to test whether the communication was successful or failed and the time needed to perform it.

Performance measurements for coordination:

- First, test the algorithm used to clear the intersection with computer simulations to see if there are any theoretical problems with the algorithm itself.
- After having an algorithm that computes a good "plan" to clear the intersection, we run it x times and see how many times the intersection is cleared safely.
- We measure how many seconds it takes to clear the intersection.

## 18.3. Part 3: Preliminary design

*Modules:*

Following subprojects are detected:

1. LED communication
   a. Duckiebot A emits a signal encoded in LED;
   b. Duckiebot B detects the signal from Duckiebot A;
   c. Duckiebot B interpret signal from Duckiebot A.
2. Coordination at intersection
   a. Each single Duckiebot computes "the plan" to clear the intersection;
   b. Each Duckiebot leave the intersection according to "the plan".

*Interfaces:*

1. For the LED communication:
   a. Led_emitter: Input for the emission is the state of Duckiebot A (waiting, entering, or navigating the intersection). The output is the corresponding LED signal.
   b. Led_detection: Input is the image of the camera. The output is the frequency/color/etc of the detected LED(s).
   c. Led_interpreter: Input is the frequency/color/etc of the detected LED. The output is the corresponding message.
2. For the coordination: The inputs are the type of intersection (with or without traffic light and number of streets) as well as the interpretation of the signals emitted by the other uckiebots. The output is decision on when to go, taken accordingly to the coordination policy.

### 1) Preliminary plan of deliverables

*Specifications:*

The Duckietown specification do not need to be revisited.

*Software modules:*

The software will be organized as follows:

- One ROS node for the emission.
- One ROS node for the detection.
- One ROS node for the interpretation.
- One ROS node for the coordination.

The existing code has already a similar structure, meaning that part of the code might be reused.

*Infrastructure modules:*

No infrastructure modules are needed.

## 18.4. Part 4: Project planning

### 1) First steps for the next phase

We are going to implement and test the alternative coordination algorithms along with the last year's algorithm and check safety and performance metrics.

TABLE 18.1. TIMETABLE

| Week | Task | Expected Outcome |
|---|---|---|
| 12.11.17-19.11.17 | Kick-off Hardware up to date Specifications for other groups First tests | Preliminary design |
| 20.11.17-26.11.17 | Test existing code Implementing new algorithms | Benchmark results First implementation |
| 27.11.17-03.12.17 | Implementing new algorithms continues Benchmark the new algorithms | Performance of the new algorithms- Deadline for Chicago |
| 04.12.17-10.12.17 | Try more sophisticated implementations such as algorithms that include communication with the bots behind, through back lights | Implementation |
| 11.12.17-17.12.17 | Benchmark new algorithms | Benchmark results |
| 18.12.17-24.12.17 | TBD | TBD |
| 25.12.17-31.12.17 | TBD | TBD |
| 01.01.18-07.01.18 | TBD | End of the project |

*Data collection:*

- Test what is seen at different initial orientations. This will be employed as specifications for the controller group and for the smart city group.
- Run experiments, take pictures, and screenshots for both LEDs detection and and intersection coordination.

*Data annotation:*

No data needs to be annotated.

*Relevant Duckietown resources to investigate:*

These are some of the important features used in the past course:

- Produce LED signal (protocol for emitting signal) - led_emitter.
- Detect LED blinking signal from camera - led_detection.
- Interpreting signaling data includes - led_interpreter.
  a. Determining what kind of signal we have (is it a Duckiebot? A traffic light? Something unwanted?).
  b. knowing how many other cars at intersection (A, B, C) and if there is a traffic light up (present/ absent).
- Coordination policy and ros node from last year.
- MIT 2016 presentation.

*Other relevant resources to investigate:*

Literature and codes from the class of 2016 at MIT. Additionally, research papers in computer vision (How to detect LEDs? How to use the camera? Etc.) and coordination (how to optimally coordinate a fleet? Etc.) might be of interest.

## 2) Risk analysis

The following risks for LEDs communication are identified:

- Detecting colors. A possible solution would be to test colors and frequency to see which one is more robust.
- Limited visibility. Slightly turning in place (and then coming back to the initial position) might alleviate this problem.

The following risks for coordination are identified:

- Synchronization problems when Duckiebots arrive at the intersection at different times and when they detect signals from other Duckiebots while Duckiebots are changing signals.

We can summarize the risks in the following table.

| Risk | Potential error | Consequence | Likelihood | Impact | Risk Priority Number (0-100) | Actio qui |
|---|---|---|---|---|---|---|
| Detecting color | Camera does not recognize colors | Signal cannot be encoded in color | 7 | 9 | 63 | Alter interp stra |
| Visibility | Limited visibility from DB camera | Limited exchange of messages | 10 | 8 | 80 | Stra that not inform from D |
| Synchronization before messages | Problems when bots arrive at the intersection at different times | Wrong communication | 3 | 10 | 30 | Take acco whil mu stra Safety ced nee |
| Synchronization during messages | Problems when bots detect signals from other bots while bots are changing signals. Problems if bot detects signals while other bots are changing signals | Wrong communication | 3 | 10 | 30 | Safety cedu this |

# Unit N-19

# Explicit Coordination: intermediate Report

## 19.1. Part 1: System interfaces

### 1) Logical architecture

Our job starts when Duckiebots are stationary at the red-line of the intersection (this is communicated to us via controllers/ parking). By clicking “start” the LED-coordination-node tells the LED-emitter-node to turn the LEDs white for all Duckiebots. Afterwards, the LED-detector-node checks for each Duckiebot if other LEDs are seen and tells it to the LED-coordination-node. Note that here there is, at least in a first approach to the problem, no turning, i.e., LEDs of Duckiebots on the left are not identified. The LED-coordination-node estimates the coordination move (either “hold on” or “go”) for each Duckiebot. The final output is a signal, named move_intersection, that will be used by the Navigators to start the procedure to navigate the intersection. Thereafter, we are not going to intervene until the Duckiebot finds itself at another intersection. Should the explicit coordination fail (for instance, because of Duckiebots not equipped with LEDs), the task of coordinating the intersection is given to the implicit coordination.

Our LED-detection, LED-emission and LED-coordination nodes affect only the Duckiebots behavior at intersection. Surely, our LED-signal could be seen from other Duckiebots in Duckietown but, at least for now, no group (except for the fleet planning group, see below) needs LEDs-based communication in other situations. A LED-signal will be used by fleet-planning to indicate the status of each vehicle (free, occupied, waiting, etc.). The Fleet planning will be using one LED for implementing this functionality (back-right one) while the other LEDs remain available for coordination purposes.

The following assumptions are made about other modules:

1. When the Duckiebot is made to stop at the red line by the Controllers at an intersection a flag “at_intersection” will be set and that is when the coordination will start. Most likely this flag will be sent out by the Parking group after it has been verified that after the intersection there is no parking.
2. Controllers guarantee that the Duckiebots will stop at the red line within the agreed tolerances (i.e., Min. 10cm behind center of red line; Max. 16cm behind center of red line; +/- 10° of rotation ; +/- 5 cm offset from center of the driving lane.).
3. Fleet planning and neural-SLAM are the ones responsible to give information about where the Duckiebots should go at the intersection (information that will not be used in any case for determining how the intersection will be cleared).
4. Navigators will take over once the Duckiebot has received the order that it can proceed to navigate the intersection (a signal “go”), moment from which our team, explicit-coordination, will no longer intervene.
5. If the Fleet planning and neural-SLAM decision is not available, the Navigators are responsible to generate a random choice for the direction that each Duckiebot will have to follow in the intersection navigation, once again, the direction that the Duckiebot will take is not of interest for the coordination part that is performed regardless of this information.

6. Explicit coordination and implicit coordination will never run at the same time on a Duckiebot.

## 2) Software architecture

Nodes:

1. LED_coordination:
   - Input: From Parking group “you are at an intersection” (additionally there is a parameter that indicates whether intersections are cleared with explicit or implicit coordination)
   - Output: Duckiebot move (“go”/ ”not go”)
   - Subscribed topic:
     - flag_at_intersection from Parking group, bool message: true/ false
   - Published topic: move_intersection
     - string message: go/ no_go
2. LED_emitter:
   - Input: Communication is needed
   - Output: LED turn on or stay off
   - Subscribed topic:
     - LED_switch from LED-coordination, string message: on/ off
   - Published topics: None

3. 1. LED_detection: Depending on the algorithm implemented: * Input: camera_image (possibly after anti-instagram) and message indicating whether detection is needed * Output: LED detected/ LED not detected * Subscribed topic: * LED_to_detect from LED_coordination, string message: yes/ no * camera_image from anti-instragram, CompressedImage * Published topic: * string message: LED_detected/ no_LED_detected

```
2. LED_detection: second option:
    * Input: camera_image (possibly after anti-instagram)
    * Output: LED detected/LED not detected with position and/or color and/or frequency
    * Subscribed topic:
        * LED_to_detect from LED_coordination, string message: yes/ no
        * camera_image from anti-instragram, CompressedImage
    * Published topic:
        * string message: LED_detected/ no_LED_detected with position and/or color and/or
frequency
```

A diagram of our nodes is shown below.

Figure 19.1. Nodes

We subscribe to the following topics:

- Corrected image with maximum assumed latency 1s;
- Flag at intersection with maximum assumed latency 1s.

The following topics are published:

- Flag go/no_go with maximum latency 60s (this is the time needed to make sure that the intersection can be navigated safely).

## 19.2. Part 2: Demo and evaluation plan

### 1) Demo plan

Our demo will be conceptually similar to the MIT2016 "openhouse-dp5", available from last year Unit F-6 - Coordination. The Duckiebots that are navigating in Duckietown, will stop at the red line and LED-communication and coordination will be performed leading to the eventual clearing of the intersection.

From testing last year's code we realized that the coordination does not seem to work with the mentioned demo. Duckiebots stop at the red line but they do not communicate so that they never leave the intersection or decide to go independently of the presence and decision of the other Duckiebots. Although we investigated the problem by looking at separate nodes, no solution has been found yet.

We aim to have a working demo that will show an effective clearing of an intersection with a variable number of Duckiebots (1 to 4) regardless of the type of intersection (3-way or 4-way, with or without traffic lights). The intersection should be cleared in a reasonable amount of time (less than 1 min) and be robust to different initial conditions (within the specified tolerances on the pose of the robots). The setup will be easy and quick: with a small Duckietown as shown in the figure below, up to four Duckiebots will be put on the road and the demo will be started from a laptop

with no further interventions required.

The required hardware will therefore be: A four way intersection tile (see image below, center), four three-way intersections tiles, twelve tiles with straight lines,four tiles with curved lines and four empty tiles. In total, twentyeight tiles, red, yellow and white tape as indicated in the figure below. Apriltags and all other required signals at the intersection will also be needed (standard type of Duckietown intersection) as well as a traffic light to illustrate the behaviour in a traffic light type of intersection.

Figure 19.2. Duckietown

## 2) Plan for formal performance evaluation

Performance will be evaluated with 3 tests:

TABLE 19.1. PERFORMANCE EVALUATION

| What is evaluated | How | Required | Collected quantities | Performance measure(s) |
|---|---|---|---|---|
| Coordination implementation performance | Run the demo | One to four Duckiebots at an intersection, every case has to be analyzed (three or four way intersection, with or without traffic light) | Time required to clear the intersection and binary variable that tells whether the intersection was effectively cleared without problems (collisions, lack of decision making, wrong detection of signals,etc.) or not. | Mean clearing time or success rate - Both for each case separately and for all combined. |
| LED emitter and LED detector | Run the specific nodes | Two to three Duckiebots in an intersection configuration (all relative positions have to be analyzed) | Number of unsuccessful LED emissions (the output has not the desired colour and/or frequency) and number of unsuccessful LED detections: the output of the LED detector does not contain all the signals it should have detected or it contains more than the ones effectively present (false positives) | Success rate - For both LED emission and detection. |
| LED detector | Run the specific node | Two to three Duckiebots in an intersection configuration (all relative positions have to be | Time required to perform the LED detection | Mean detection time |

## 19.3. Part 3: Data collection, annotation, and analysis

### 1) Collection

No data is needed to develop the algorithm. Data might be needed to test the implementation of the detection.

### 2) Annotation

No data need to be annotated.

### 3) Analysis

As no data annotation is needed, no software will be developed.

UNIT N-20

# Explicit coordination: final report

## 20.1. The final result

Video of the final result:

Figure 20.1. Explicit coordination's video

To reproduce the results please see the operation manual .

## 20.2. Mission and Scope

Our mission is to coordinate the intersection navigation safely and cleverly through explicit communication.

### 1) Motivation

Duckietowns are complex systems where the traffic situations of a real city should be emulated. These towns contain three- and four-way intersections: the Duckiebots should be able to navigate them without crashing into each other and this requires a clever coordination scheme. Intersections represent a key element of a smooth city navigation.

There are two ways of coordinating Duckiebots: - Using a traffic light, - Using a communication protocol between the vehicles.

Hence, we aim to have both a centralised and a decentralised solution as well as an integration of the two. While the centralised solution boils down to understand the signal emitted by a referee (i.e., a traffic light), the decentralised coordination scheme should allow the Duckiebots to operate on their own, i.e., to communicate between each other and to take decisions without any external help.

### 2) Existing solution

A prior implementation for intersection coordination was already available from the 2016's MIT class. The principle was simple: When a Duckiebot comes at an intersection and stops at the red line. Then, In the case with a traffic light, the Duckiebot detects the frequency at which the traffic light is blinking and acts based on the road rules. In the case without traffic light, the Duckiebot detects the frequency at which the other bots are blinking and adjusts its emitted frequency depending on its state. From an implementation perspective, the distinction was made a

priori, i.e., the Duckiebots were not making use of the information coming from the Apriltags detection and therefore not were able to navigate systems with both types of intersection.

From the description above, we can distinguish two modules:

- LED emission and detection,
- Coordination based on the detected signals.

For the emission, three signals can be produced: each signal is encoded with a specific color and frequency. While Duckiebots are designed to recognise only frequencies, color are used to allow humans to easily understand the signals emitted by the Duckiebots. The signals represent the states: negotiation (and in which phase of the negotiation it is) or navigation of the intersection.

For the detection, the position and frequency of blinking LEDs are registered.

For the coordination, with the assumption that each vehicle can only see other vehicles on its right but not on its left, the Duckiebot yields its position if the only visible car is on the right, otherwise the Duckiebot waits (light green or red) or crosses (yellow light).

### 3) Opportunity

The existing solution had essentially two drawbacks:

- The overall success rate was about 50%: The algorithm for LED-detection, and/or coordination failed, leading to a potential crash of the Duckiebots.
- In case of success, the LED-detection and/or coordination algorithms required an average of four minutes to clear an intersection with four Duckiebots. This results in an extremely slow process, which would block a city with dozens of Duckiebots.

Although the solution was problematic, it still gave us some important intuitions on how to solve the problem.

First of all, using a LED-communication protocol is a brilliant idea to let the Duckiebots communicate with each other. Since the previous communication algorithm had the only disadvantage of being slow (also because of several bugs), we started by re-thinking the coordination algorithm. The existing implementation for the coordination was rather complex and articulated, resulting in confused strategies which led to the failure rate of 50%. In order to develop a simpler and lighter algorithm we took inspiration from an existing media access control protocol (MAC): the so called Carrier Sense Multiple Access (CSMA, https://en.wikipedia.org/wiki/Carrier-sense_multiple_access). This algorithm gave us the basic idea behind our strategy and allowed us to have a lighter protocol.

In the second place, we re-designed the LED-detection/-interpreter to be more efficient and robust based on the detection of blobs and not only on the detection of frequencies.

Lastly, we wanted to have a demo that could deal with both intersections, i.e. with and without a traffic light, as opposed to the two available demos from MIT 2016's class.

## 20.3. Definition of the problem

### 1) Final objective

Given a Duckietown, Duckiebots must be able to cross an intersection efficiently. An intersection is said to be cleared efficiently if and only if:

- The time needed to cross the intersection is below 1 minute.
- Only one Duckiebot at a time crosses the intersection.
- The Duckiebots do not crash during the navigation.

### 2) Assumptions

*Functional assumptions:*

The following assumptions are made:

- The Duckiebot is of type DB17-l, i.e. has LEDs mounted on it.
- Camera works properly (frequency 30Hz, resolution 640x480)
- LEDs work properly and emit the signals with the correct color and frequency.
- Duckiebots are able to see the vehicles in front and on the right with respect to their position: one cannot assume that the left visual is clear.
- The Duckiebots do not move while "waiting" at the intersection.
- One Duckiebot navigates the intersection at the time.
- The intersection is among one of the standard intersections of Duckietown, detected through april tags.

*Assumptions on other groups:*

The following assumptions are made:

- The Controllers: one to four Duckiebots are at the intersection with a certain position and orientation with respect to the stop line. Responsible for this assumption are The Controllers, which should guide the Duckiebot at intersection and make it stops with following pose:
  - Min. 0 cm behind the stop red line;
  - Max. 6 cm behind the stop red line;
  - Max. +/- (left/right deviations) 2 cm from the center of the line; +/- 10° of rotation with respect to the perpendicular line of the red line.
- Navigators: as soon as coordination is decided, Duckiebots have to navigate through the intersection. This task is accomplished by The Navigators.
- Implicit coordination: it is assumed that explicit and implicit coordination are never running at the same time.

### 3) Performance metrics

The metrics that are going to be used to judge the achievement of the goal are two:

1. Mean clearing time
2. Success rate

Our goal is to work out a procedure so that the Duckiebots cross an intersection efficiently, therefore the performance metrics follow naturally. The clearing time should not exceed one minute and the success rate should be higher than 70% in all intersection configurations (1, 2, 3 vehicles, with or without a traffic light).

## 20.4. Contribution / Added functionality

### 1) LEDs Detection

LEDs are modeled as blobs in an image: A Blob is a group of connected pixels in an image that share some common properties, which, in the case of LEDs, is the intensity of the pixel. Among the many ways to detect blobs, the one that turned out to be more robust for our purpose was the following algorithm (https://www.learnopencv.com/blob-detection-using-opencv-python-c/): - Thresholding: Convert the source images to several binary images by thresholding the source image with thresholds. - Grouping: In each binary image, connected white pixels are grouped together. Let's call these binary blobs. - Merging: The centers of the binary blobs in the binary images are computed, and blobs located closer than a threshold are merged. - Center and Radius Calculation: The centers and radii of the new merged blobs are computed and returned. - Filtering: The blobs are filtered by size and shape.

To increase the robustness of the detection, a sequence of $N$ images in analyzed. Each blob $b$ is characterized with a position vector $x_b \in \mathbb{R}^2$ and a signal $y_b \in \{0,1\}^N$ of dimension, indicating whether the blob was detected in the image (1) or not (0). The blobs found are collected in a set called $B$. The algorithm works as follows: - Initialization: Initialize the set of blobs $B$ with the empty set. - Recursion: For image $i$ and blob $j$ (with position $x_{ij}$): + If $\|x_{ij} - x_b\| > \mathbf{TOL}$ for all $b \in B$ the blob is added to $B$ with $x = x_{ij}$ and $y = e_i$, where $e_i$ is a vector of dimension $N$ whose $i$ entry is 1 and all other entries are 0. + If $\|x_{ij} - x_b\| \leq \mathbf{TOL}$ for some $b \in B$. Then, we "merge" blob $j$ with the blob it was closest to, i.e., we set the $i$-th entry of $y_{\bar{b}}$ to 1, where $\bar{b} = \arg\min_{b \in B} \|x_{ij} - x_b\|$. That is, we store the information that the blob has been observed in the $i$-th image.

After this procedure, the user has full information about the blobs in the images. Then, one may identify the presence of another Duckiebot as follows: - Analyzing the frequency spectrum of the signal y of each blob. If the known emission frequency and the detected frequency using the Fast Fourier Transform match, then we can conclude that a car has been detected. - Using some heuristics. For instance, for each blob one may compute

$$m_b = \frac{1}{N} \sum_{i=1}^{N} [y_b]_i$$

and act upon this number. This algorithms is run three times: To detect Duckiebots on the right, to detect Duckietbots on the left, and to detect traffic lights. To increase the robustness and reduce the computational demand, the image is cut accordingly. Hence, the output of the algorithm are three Booleans indicating the detection on the right, on the front, and for the traffic light respectively.

### 2) Coordination

Our coordination algorithm allows the hybrid management of situations with and without a traffic light.

In the situation without the traffic light, the algorithm is based on the concept of the exponential backoff, cited above. The basic concept is really simple: a Duckiebot can act on its own and decide wether to enter the intersection or to wait, without the help of a centralised system. The Duckiebot arrives at the intersection

and recognises its type. Once it stops in enters the state AT_STOP_CLEARING, which represents the action of deciding what to do. When the Duckiebot is in the state AT_STOP_CLEARING, it starts blinking at the specific defined frequency. This makes it recognisable for the potential other Duckiebots waiting in the other lanes. The other task of a Duckiebot in this state, is to check the existence other waiting Duckiebots. Since we assumed that the Duckiebot is only able to see front and right, these are the two regions of its visual where it checks if other Duckiebots are present. In order to check if a Duckiebot stays in the other lanes, we use the defined command SignalsDetection. If the Duckiebot sees other Duckiebots waiting in front or right to it, it sacrifices itself by entering in the state SACRIFICE. This consists in the first place in stopping blinking and looking, allowing other seen Duckiebots to coordinate. This state lasts for a random bounded time, defined with the variable random_delay. Once this random time has passed, the Duckiebot re-enters the state AT_STOP_CLEARING. If instead the Duckiebot does not see any other Duckiebot waiting, it enters in the state KEEP_CALM. This state makes sure that the Duckiebot waits a different random time before deciding to navigate the interection, decreasing the chance of a possible crash due to errors in the navigation. During this period, the Duckiebot checks if other Duckiebots are blinking: if yes, he sacrifices itself and enters the state SACRIFICE; if not, it enters the state GO, which corresponds to the decision to navigate the intersection.

The coordination algorithm in the situation with the traffic light, is simpler. As the Duckiebot arrives to the intersection, it recognises its type and enters the state TL_SENSING. In this state, he checks for the traffic light signal which allows it to navigate the intersection. In this case it enters the state GO, which corresponds to the decision to navigate the intersection. If not, it waits until its turn comes.

### 3) Logical architecture

Our job starts when Duckiebots are stationary at the red-line of the intersection (this is communicated to us via controllers/ parking). By clicking “start” the LED-coordination-node tells the LED-emitter-node to turn the LEDs white for all Duckiebots. Afterwards, the LED-detector-node checks for each Duckiebot if other LEDs are seen and tells it to the LED-coordination-node. Note that here there is, at least in a first approach to the problem, no turning, i.e., LEDs of Duckiebots on the left are not identified. The LED-coordination-node estimates the coordination move (either “hold on” or “go”) for each Duckiebot. The final output is a signal, named move_intersection, that will be used by the Navigators to start the procedure to navigate the intersection. Thereafter, we are not going to intervene until the Duckiebot finds itself at another intersection. Should the explicit coordination fail (for instance, because of Duckiebots not equipped with LEDs), the task of coordinating the intersection is given to the implicit coordination.

Our LED-detection, LED-emission and LED-coordination nodes affect only the Duckiebots behavior at intersection. Surely, our LED-signal could be seen from other Duckiebots in Duckietown but, at least for now, no group (except for the fleet planning group, see below) needs LEDs-based communication in other situations. A LED-signal will be used by fleet-planning to indicate the status of each vehicle (free, occupied, waiting, etc.). The Fleet planning will be using one LED for implementing this functionality (back-right one) while the other LEDs remain available for coordination purposes.

The following assumptions are made about other modules:

1. When the Duckiebot is made to stop at the red line by the Controllers at an intersection a flag “at_intersection” will be set and that is when the coordination will start. Most likely this flag will be sent out by the Parking group after it has been verified that after the intersection there is no parking.
2. Controllers guarantee that the Duckiebots will stop at the red line within the agreed tolerances (i.e., Min. 10cm behind center of red line; Max. 16cm behind center of red line; +/- 10° of rotation ; +/- 5 cm offset from center of the driving lane.).
3. Fleet planning and neural-SLAM are the ones responsible to give information about where the Duckiebots should go at the intersection (information that will not be used in any case for determining how the intersection will be cleared).
4. Navigators will take over once the Duckiebot has received the order that it can proceed to navigate the intersection (a signal “go”), moment from which our team, explicit-coordination, will no longer intervene.
5. If the Fleet planning and neural-SLAM decision is not available, the Navigators are responsible to generate a random choice for the direction that each Duckiebot will have to follow in the intersection navigation, once again, the direction that the Duckiebot will take is not of interest for the coordination part that is performed regardless of this information.
6. Explicit coordination and implicit coordination will never run at the same time on a Duckiebot.

## 4) Software architecture

Nodes:

1. coordination_node:
   - Input: From Finite State Machine group “you are at an intersection” and apriltags detection
   - Output: Duckiebot move (“go”/ ”not go”)
   - Subscribed topic:
     - apriltags_detections from apriltags_node
     - trigger from finite state machine
   - Published topic: intersection_go
     - string message: go/no_go
2. led_emitter_node:
   - Input: Communication is needed
   - Output: LEDs turn on or stay off
   - Subscribed topic:
     - LED_switch from LED-coordination, string message: on/ off
   - Published topics: None
3. LED_detection: second option:
   - Input: camera_image (possibly after anti-instagram) and trigger
   - Output: LED detected/LED not detected with position
   - Subscribed topic:
     - Trigger from finite state machine
     - camera_image from anti-instragram, CompressedImage
   - Published topic:
     - string message: LED_detected/ no_LED_detected with position

A diagram of our nodes is shown below.

Figure 20.2. Nodes

We subscribe to the following topics:

- Corrected image with maximum assumed latency 1s;
- Trigger with maximum assumed latency 1s.

The following topics are published:

- Flag go/no_go with maximum latency 60s (this is the time needed to make sure that the intersection can be navigated safely).

## 20.5. Formal performance evaluation / Results

TABLE 20.1. PERFORMANCE EVALUATION

| Situation | Performance measure | Required | Obtained |
|---|---|---|---|
| One Duckiebot at the intersection | Clearing time | 60s | 12s |
| One Duckiebot at the intersection | Success rate | 90% | 98% |
| Two Duckiebots at the intersection | Clearing time | 60s | 25s |
| Two Duckiebots at the intersection | Success rate | 80% | 89% |
| Three Duckiebots at the intersection | Clearing time | 60s | 50s |
| Three Duckiebot at the intersection | Success rate | 70% | 89% |
| One Duckiebot at a traffic light type intersection | Clearing time | 60s | 25s |
| One Duckiebot at a traffic light type intersection | Success rate | 90% | 92% |

Failures were mainly caused by the following reasons: - Duckiebots detecting the wrong sign (i.e., expecting a traffic light instead of a normal intersection). - Blobs not properly detected. This is mainly due of failures in the parameters and in the camera calibration. - Duckiebots crashing because of poor intersection navigation.

## 20.6. Future avenues of development

There is room for improvement for the coordination part of this project. Our approach, in the case of an intersection without traffic light, prioritises robustness rather than efficiency (in some cases all the Duckiebots at an intersection could turn off and restart the whole protocol again) and it is easy to imagine a scenario with an improved efficiency (tradeoff with complexity).

An idea would be to encode in the signal also the intentions of the Duckiebot and, by doing so, allow multiple Duckiebots to navigate the intersection at the same time if their directions are compatible. In fact, if two Duckiebots wanted to go straight they could move at the same time. The clearing time could also be reduced in the case of an intersection with traffic light if the latter was able to see where the vehicles are (prevent the light to turn green in a direction where no vehicle is waiting to cross).

UNIT N-21

# Implicit Coordination: preliminary report

## 21.1. Part 1: Mission and scope

### 1) Mission statement

Formation keeping and collision avoidance using implicit communication

### 2) Motto

ALIIS VIVERE

(Live for others)

### 3) Project scope

*What is in scope:*

- Relative pose estimation
- Formation keeping: formalize controller that keeps constant distance from vehicle ahead
- "Follow the leader demo"
- Avoiding traffic waves
- Intersection coordination
  - **Detection**: position (bounding box) + pose estimation per bot in FoV.
  - **Tracking**: trajectory (sequence of pose) estimation per bot tracked, parametrized in time.
  - **Prediction**: predict bots behavior through intersection by integrating tracking information and rules of the road (no explicit communication allowed).
  - Implement traffic rules
  - Intention of the other car has to be predicted
  - Priority traffic rule: First come first serve. Crossings possible without explicit communication
  - Designing Fiducial markers (April tags)
- (Designing a special T intersection with only one stop)

*What is out of scope:*

- Adding hardware
- Intersection navigation
- Communication with LEDs / explicit communication

*Stakeholders:*

- System Architect
- Software Architect
- Knowledge Tzarina
- Anti-Instagram

- Controllers
- Navigators
- Explicit Coordinators

## 21.2. Part 2: Definition of the problem

### 1) Problem statement

**Mission:** Formation keeping and intersection navigation with Duckiebots just using implicit communication.

**Problem statement:** Detect other Duckiebots, follow the leader in a secure distance through Duckietown and coordinate intersections using implicit communication.

### 2) Assumptions

- Duckiebots do not use explicit communication, e.g. LEDs, WLAN ...
- Duckiebots have different appearance (different configuaration, LEDs on/off, ...)
- All Duckiebots are autonomous, not remote controlled
- All Duckiebots use the same formation and implicit control algorithm

### 3) Approach

Formation:

- Vehicle detection
  - April tags
    - Analyze already existing code for vehicle detection (Software, Duckumentation)
  - CNN-based
    - Image annotation
- Tracking
  - model based
  - learning based
- Prediction
  - model based
  - learning based
- Distance Control

Implicit Coordination

- Data Collection
  - Annotation tool: if only detecting bots, any custom manual approach will work, if not, then we will use thehive.ai API.
  - Types of intersections: 4-way, 3-way, 3-way with 1 stop sign.
  - No pedestrians.
  - Motions: moving vs. static traffic, moving vs. static self.
  - Including “look-around” behavior.
  - Illumination variances: add a predominant light to Duckietown so we can have strong shadows and different lighting conditions.
  - Appearance variances: different bots configurations (e.g: with/without the shell, a duck, etc.)

  - Frames will be extracted from videos at a 3 fps framerate.
- Detection -Instance-level 2D/3D Bounding box + pose estimation.
  - Explore OpenCV built-in detection capabilities.
  - Explore supervised learning methods for detection, including existing models (e.g: YOLO2, SDD, etc).
- Tracking
  - Estimate prior trajectory of each bot based on the instance level detection generated.
  - Explore OpenCV tracking algorithms (Doc)
  - Explore MOT algorithms based on Deep Learning (Paper)
- Prediction
  - 3D bounding box + orientation + velocity + Position prediction (1, 5, 20) + estimating a policy ("behavior") applying rules of the road.
  - APPROACH: TBD.

#### 4) Functionality-resources trade-offs

*Functionality provided:*

Formation keeping and intersection coordination

- Detection of other Duckiebots within field-of-view
- Tracking
- Prediction

*Resources required / dependencies / costs:*

- Multiple duckiebots
- Duckietown
- April tags

*Performance measurement:*

Robustness of Formation keeping and Vehicle Detection

- Number of duckiebots in the "Follow the leader" demo
- Duration of the demo (time)
- Transient error of distance and perpendicular offset

Implicit coordination

- **Detection**: bounding-box (intersection over union, precision + recall), uncertainty in pose estimation
- Throughput rate of duckiebots through intersection
- Waiting time at intersection Number of successful crossings at intersection before a collision happens
- Robustness to duckiebot pose, appearance, scale, orientation ambiguity and LED-lightning

## 21.3. Part 3: Preliminary design

#### 1) Modules

- Detection of Duckiebot
- Estimation of relative pose
- Controller for following the leader
- Intersection behavior

### 2) Interfaces

Detection of Duckiebot

- Input: April tag
- Output: Position in formation and duckiebot parameters

Estimation of relative pose

- Input: Camera image
- Output: Estimate of Relative pose between bots

Controller

- Input: Estimated pose
- Output: Command to motors

Intersection behavior

- Input: State at stopping line
- Output: Preferences at intersection

### 3) Preliminary plan of deliverables

*Specifications:*

(Special T-intersection with only one stop in Duckietown)

*Software modules:*

- Detection / Estimation node
- Tracking node
- Controller node
- Implicit coordination behavior node

*Infrastructure modules:*

None

## 21.4. Part 4: Project planning

TABLE 21.1. SCHEDULE

| Date | Task Name | Target Deliverables |
|---|---|---|
| 17/11/2017 | Kick-Off | Preliminary Design Document |
| 24/11/2017 | Review state of the art, theoretical formalisation | List of what has to be implemented and how and what can be reused from last year. Benchmarking current state |
| 1/12/2017 | Implementation | - |
| 8/12/2017 | Implementation and Testing | Code |
| 15/12/2017 | Evaluation, Optimization | Performance measurement |
| 22/12/2017 | Buffer | - |
| 29/12/2017 | Duckumentation | Duckuments |
| 05/01/2018 | End of project | - |

End of project

### 1) Data collection

Annotated images

### 2) Data annotation

Duckiebots in bounded boxes under different conditions (Illumination, LEDs on/off, Duckiebot configurations, ...)

*Relevant Duckietown resources to investigate:*

Exisiting intersection safty rules (Duckumentation)

Existing vehicle detection algorithm (Code, Duckumentation)

*Other relevant resources to investigate:*

Robotics Handbook: Ideas for controller for 'follow-the-leader' or driving in traffic on pg. 808

### 3) Risk analysis

Likelihood from 1-10, where 10 is very likely and 1 very unlikely.

Impact from 1-10, where 10 is a huge negative impact and 1 not so bad.

TABLE 21.2. RISKS

| Event | Likelihood | Impact | Risk response Strategy (avoid, transfer, mitigate, acceptance) | Actions required |
|---|---|---|---|---|
| False Negative in Vehicle detection can lead to crash | 4 | 9 | mitigate | The effect of FPs are less bad than those of FNs. Punish FNs more in an eventual cost function. |
| False Positive in Vehicle detection makes bot stand still. | 4 | 2 | accept | - |
| Inaccurate estimation of distance between bots | 3 | 7 | mitigate | Early testing, big enough safety margin, improve controller |
| Collision of 2 duckiebots in an intersection | 3+4 | 9 | mitigate | Combination of Event 1 and 3 |

UNIT N-22

# Implicit Coordination: intermediate report

## 22.1. Part 1: System interfaces

### 1) Logical architecture

The first part of the project is to detect and track Duckiebots in general. In addition the motion of the Duckiebot should be predicted in a short time horizon. With this detection it should be able to get information of how many Duckiebots are at an intersection and if the way is free to cross the intersection. We will provide a protocol to trigger the driving through intersections.

The detection is also used to implement a demo in which a number of Duckiebots will follow a leader-Duckiebot. The leader will be controlled with forward navigation in Duckietown. The architecture is divided into the following logical submodules:

Detection

- Object bounding boxes and class outputted in a topic (in image space - per image no tracking) (2pts inside the image 640x480 + class)
- on your laptop you should see in rqt_image_view an image with bounding boxes around robots
- For Follow-the-leader:
  - Easy: with fiducial marker detection
  - Hard: With detection see (i) and (ii)

Tracking

- Output of the 2D trajectory (x,y) (stretch - pose) of each of the detected objects
- visualization of the trajectories in RVIZ

Prediction

- Output of a 2D class of trajectories with probabilities for future motion of objects
- In RVIZ a class of future object trajectories weighted by something

Intersection Coordination

- An agent (Duckiebot) infers when it is safe to go without explicit communication. The agent follows a protocol known to each agent on the road.
- Duckietown Intersection Coordination Protocol (DICP):
  - Inspired by CSMA/CD algorithm
  - Constraint: only one Duckiebot at a time crossing the intersection

Follow-controller

- Controller gives wheel velocity commands to follow a detected/tracked Duckiebot to keep a constant distance behind the Duckiebot in front.
- For centering the Duckiebot behind the one in front of him we will use the lane controller from the “Controllers”.
- Velocity control fells into the scope of our tasks.

Figure 22.1. The Coordination Module

Figure 22.2. The Formation Keeping Module

Easy: Own robot is not moving

Hard : Own robot is moving

**Assumptions about other modules**

Intersection Coordination:

- Duckiebot knows when it is at an intersection, coordination_node receives msg when at stopline. Given by Controllers.
- Duckiebot knows how to navigate through an intersection, only needs a wait/go msg from our side (interface with navigators) Formation Keeping:
- We can use the lane controller from “Controllers” to center a Duckiebot behind the Duckiebot in front. We can use the obstacle avoidance for the leader Duckiebot to avoid duckies. Given by Savior.
- The Duckiebot will exit the Formation Keeping mode as soon as the Duckiebot in the front leaves the lane (overtakes or avoids an obstacle) and at intersections.

## 2) Software architecture

The software architecture is divided into the following software nodes:

Detection:

- One node - input camera image, output detections define a msg type Tracking:
- One node - input detections, output trajectories - define message type
  - For Hard need the odometry of robot

Prediction : later...

Intersection Coordination:

- One node - input msg from tracking node, msg stop-line filter - output boolean

msg go/wait
  - Input: flag_at_stop_line -> True if Duckiebot is at intersection
  - Input: Tracking_msg -> Are other Duckiebots at intersection or will arrive at intersection in the prediction horizon
  - Output: flag_wait_go -> True if Duckiebot can navigate through intersection

Controller (Follow-the-leader/Formation Keeping):

- Easy:
  - One node - input pose of Duckiebot - output car control msg
    - Input: geometry_msgs/PoseStamped -> Estimated pose from Duckiebot in front to follower
    - Output: Msg for lane controller from "Controllers -> Msg for generating motor commands
    - flag_follow set true
- Hard:
  - One node - input pose of Duckiebot - output car control msg
    - Input: Tracking/detection msg
    - Output: Msg for lane controller from "Controllers -> Msg for generating motor commands

**Latency introduced by the subscribed modules**

Will be added

**Latency introduced by our modules**

- We need to do some back of the envelope math for the functionality: turn, look, detect, turn back, go.
- Latency for Follow-the-leader controller: max 15ms
- Latency for boolean msg at intersection: to be benchmarked

## 22.2. Part 2: Demo and evaluation plan

### 1) Demo plan

Dream scenario intersection coordination: Duckiebots loop through the map as indicated in Figure 22.3. At the intersections the Duckiebots coordinate implicitly. Demo can run indefinitly without collision. Tunable: density of the traffic (fleet size) and the speed.

Dream scenario follow-the-leader: Leader drives with indefinite navigation through Duckietown and other Duckiebots follow him. The Duckiebots are able to keep up with the leader.

Required Hardware Components

- Circle grid on Duckiebots
- The tiles to build the map in Figure 22.3

Figure 22.3. The Demo Map

### 2) Plan for formal performance evaluation

TABLE 22.1. PLAN FOR PERFORMANCE EVALUATION

| What is evaluated | How | Required | Collected quantities | Performance measure(s) |
|---|---|---|---|---|
| Coordination implementation performance | Run the demo for intersections | Up to four Duckiebots at an intersection, different configurations have to be analyzed (three or four way intersection) | Time required to clear the intersection | Mean clearing time for each configuration separately and for all combined. |
| Vehicle detector | Run the detection node | Two to four Duckiebots at an intersection | Number of TP, FP, FN | -Precision -Recall |
| Follow the leader | Run the demo for follow the leader | More than 4 Duckiebots Duckietown (no special requiremtens) | Distance between Duckiebots | -Number of Duckiebots in formation -Mean error for desired distance |

## 22.3. Part 3: Data collection, annotation, and analysis

### 1) Collection

8000 frames are logged. Sent to the hive.

We don't need extra help in collecting the data from the other teams.

**Method for taking logs**

Two robots are taking logs and there are 6-7 other robots around town. Different sizes

colors. Shells/no shells.

## 2) Analysis

# Unit N-23
# Implicit Coordination: final report

## 23.1. The final result

Figure 23.1. Implicit Coordination Video

Figure 23.2. Follow the Leader Video

The operation manual can be found here.

## 23.2. Motivation, Mission and Scope

### 1) Implicit Coordination Motivation:

Intersection coordination is a crucial task when it comes to safety and avoiding congestions, since it is a bottleneck in road traffic. We built a system for implicit coordination which is able to function without traffic lights and explicit communication e.g. communicating over WLAN or LEDs with other vehicles. The advantage of our coordination algorithm is that it doesn't rely on the latter. Hence we can still guarantee a fluid and save road traffic at intersections, even if the communication or coordination hardware is perturbed or dead.

### 2) Follow the Leader Motivation:

A fluid and smooth road traffic is beneficial in many ways: It saves time and leads to better energy efficiency and more safety. Therefore wavelike road traffic behaviour has to be omitted. Our idea was to create an algorithm that tries to homogenize the road traffic by forcing the vehicles to keep a certain distance to each other.

## 23.3. Opportunity and Existing solution

### 1) Existing Solution and Opportunity Implicit Coordination

There was no previously existing solution for the implicit coordination problem. As stated in the paragraph above, a solution for this problem is very desirable if not absolutely necessary for an autonomous driving system. Since implicit coordination at intersections was rather a tabula rasa for us, we gathered our ideas from various fields including game, communication and network theory.

### 2) Existing Solution and Opportunity Follow the Leader

The idea to use fiducial tags for the follow the leader problem on the other hand already existed. However, this task was implemented in a rather crude way. The Duckiebots were just ought to perform a full stop, whenever they detected another Duckiebot. We added a pose estimation of the leader and thus were able to create a much more sophisticated controller.

## 23.4. Definition of the problem

### 1) Definition of the Problem Implicit Coordination:

The final objective for this part was that, when two, three or four Duckiebots arrive at the same time at an intersection, they are able to handle the challange of who is allowed to drive first, autonomously and without any means of explicit communication. They are however allowed to use implicit communication. Which means they are allowed to observe the other Duckiebots and draw conclusions about the intents of the other Duckiebots from these observations. For this, we assumed that: - Duckiebots do not use explicit communication, e.g. LEDs, WLAN etc. - Duckiebots have different appearance. - All Duckiebots are autonomous, not remote controlled - All Duckiebots use the same formation and implicit control algorithm. For evaluating the performance, we decided to test our algorithm at an intersection and judge by how many Duckiebots can be handled and in what time it does so.

### 2) Definition of the Problem Follow the Leader:

The final goal here, was that the Duckiebots can follow another Duckiebot in front of them and adjust their velocity accordingly. Meaning ideally, they slow down if the leading Duckiebot does so and accelerate analogously. The assumptions here were: - All Duckiebots use the same algorithm - All Duckiebots are equipped with a fiducial tag that allows us to estimate their relative position and pose. The success can be easily evaluated by how many Duckiebots can follow their respective leader at the same time. Furthermore keeping an equal distance between the Duckiebots performance criterion.

## 23.5. Contribution / Added functionality

### 1) Contribution Implicit Coordination:

Our implicit coordination algorithm is inspired by the Carrier Sense Multiple Access/Collision Detection (CSMA/CD) algorithm which handles the access of different parties on a shared resource. In our case the Duckiebots represent the parties and the shared resource correlates with the intersection. This CSMA/CD not just guarantees us, that all Duckiebots are crossing the intersection safely, but is also enables us to give insightful estimates of the maximum throughput and the average waiting time at the intersection, given by the rich theory behind CSMA/CD. Our implementation of CSMA/CD for intersection coordination works the following: 1. Drive towards the intersection and stop at the stopline 2. Wait a random timespan and check if a Duckiebot in your field of view is driving using the Duckiebot detection algorithm 3. If no other Duckiebot is driving cross the intersection. Else repeat Step 2.

Additionally we have implemented rigth priority option in order to accelerate the traffic at the intersection. Rigth priority doesn't allow a Duckiebot to drive and as lang as another Duckiebot is standing right to them at an intersection.

Figure 23.3. Process Flow Chart Implicit Coordination

## 2) Contribution Follow The Leader:

The Duckiebots have two steering inputs, the forward velocity and the angular velocity. The angular velocity is calculated by the lane following algorithm and not changed here. All our algorithm does is calculate the velocity required to maintain a predefined distance to the vehicle in front:

*Pose Estimation:*

This algorithm consists of roughly two parts: The first one is the pose estimation of the leading Duckiebot and the other one is the adjustment of the velocity according to this estimate. We used an OpenCV library blob detector that detects fiducial tags from the camera. From the pixel coordinates of the detected blobs, i.e. the markers of the fiducial tag, the transformation matrix between the tag coordinate frame and the camera coordinate frame is calculated. The OpenCV algorithm gives us the the transformation C_T_CP, which is the homogeneous transformation from camera to tag in the camera frame. We map the this transformation in 3D to the 2D case, where the pose of the Duckiebot will be represented by ρ, ψ and θ.

First we calculate P_T_PC, via the following two steps:

P_R_PC = C_R_CP^T = C_R_CP^(-1)

P_t_PC = -P_R_PC*C_t_CP,

where we get P_R_PC AND P_t_PC FROM P_T_PC = [P_R_PC, P_t_PC; 0 0 0 1].

*In the following lines A(a,b) will denote the entry at row a, column b of matrix A and a(b) the b-th entry of vector a.*

We assume that the tag is centered on the Duckiebot's rear such that the Z-axis of

the tag coordinate system is aligned with the X-axis of the Duckiebot and the X-axis of the tag with the Y-axis of the leading Duckiebot. (The view of the image is from above.)
With these assumptions we can calculate the pose of the leading Duckiebot in the coordinate system of the following Duckiebot. ρ is the distance between the Duckiebots, calculated from the translation vector. ψ is the rotation of the tag around its Y-axis And finally θ can be calculated from the calculated ψ.
ρ = sqrt(P_t_PC(1)²+P_t_PC(3)²)
D_t_DP = - P_R_PD^T*P_t_PD = [d_1;d_2]
θ = arctan2(d_2, d_1)
ψ = arctan2(-P_R_PC(3,1), sqrt(P_R_PC(3,2)²+P_R_PC(3,3)²))
The blob detector outputs the distance ρ between the camera and the tag, the angle θ and finally, the angle ψ of the tag relative to the camera. The velocity of the leader is calculated according to the difference in distance to the leader between two consecutive detections, divided by time between the two detections.
(ρ_1 − ρ_2)/ΔT
Thus, we get the two ouputs of the black box in the picture, d_Leader and v_Leader.

Figure 23.4. Coordination Trafo1

Figure 23.5. Coordination Trafo2

*Velocity Control:*

The following calculations are also illustrated in the picture. First, the actual distance between the two Duckiebots is subtracted from the desired distance d*.
e_d = d* - d_Leader
e_d holds information whether the distance to the leading Duckiebot is too large, too small or just right. From here, we calculate a velocity to adjust this distance to the desired one.

Δv = K_D/T * e_d
e_d/T is the velocity required to compensate the missing distance till the (presumed) next measurement. K_D is a design parameter. Δv is then added to the estimated velocity of the leader.
e_v = Δv + v_Leader
Δv adjusts the distance between the two Duckiebots and by adding it to v_Leader we ensure that the two vehicles drive with roughly the same velocity. The final velocity e_v is then again multiplied with a design parameter K_p. This helps to dampen the the rather noisy pose estimation of the leader.
v_Duckiebot = K_p*e_v
The resulting v_Duckiebot is then used as the input for the Duckiebot.

*Further Details:*

Lastly, there are some precautions not shown in the picture: If the velocity v_Duckiebot is smaller or equals to 0, both the velocity and omega input of the Duckiebot are set to 0. It is undesirable, that the Duckiebots start to drive backwards, as they cannot follow the lanes or avoid obstacles that way. If omega is not set to 0, the Duckiebots start rotating on the spot which – besides looking bad – causes them to lose track of the fiducial tag of the Duckiebot in front of them which in turn causes them to collide.
Finally, if the distance d_Leader falls under a certain threshold, an emergency brake is performed.

Figure 23.6. Controller

## 23.6. Results and Performance Evaluation

### 1) Results and Performance Evaluation Implicit Coordination

Omitting possible errors which might occur in case of the implicit coordination at intersections, one should take the following precautions.
You need the correct april tags at the intersection, otherwise the Duckiebot won't know what kind of situation (intersection) it is dealing with. When the stopline isn't detected the algorithm doesn't start, so all Duckiebots should stop at the stopline. Furthermore you can get problems with twisted coordination systems for the detected position of other Duckiebots if your extrinsic camera calibration is wrong on the laptop (assuming your running the detection node on your laptop).Sometimes a robot is detected if there isn't actually one, which could slow down the traffic at the intersection. We agreed on this with our Canadian friends who did the detection, since we would otherwise risk to overlook a real Duckiebot which would be fatal. In rare cases the detection does not detect a robot. In order to assure the detection works as good as possible I would suggest relaunching the multivehicle detection node regularly, since it seems to start lagging the longer it is running. If you would like to keep track of the detection you can run rostopic echo /robotname/multivehicle_tracker_node/tracking.
The algorithm is designed for up 4 robots at the stoplines, but since we depend on

the indefinit navigation, we are prone to navigation mistakes. If the tracking and navigation are correct, we are able to coordinate 4 way intersections for up to 4 Duckiebots. Empirical tests showed that our algorithm never needed more than 30 seconds to clear a four way intersection with 4 Duckiebots.

#### 2) Results and Performance Evaluation Follow the Leader

We tested our follow the leader with up to four Duckiebots in duckietown and there doesn't seem to be an upper limit on the number of Duckiebots following each other. Regarding the equal distance we are somewhat restricted by the computational power of the Duckiebots and hence the time needed for the detection of the antecedent Duckiebot. The detection time can vary from image frame to image frame however, 0.4 seconds used to be an appropriate upper bound. We found that this delay lead to deviations of maximally 20% from our optimal reference distance. In order to function properly the gain of the wheel calibration should be set to 0.6 as proposed by the Controllers to assure a smooth interplay between our controller and the lane following algorithm. Note that very high gains can dramatically worsen the deviations from the reference distance. Additionally, as always, a correct camera and wheel calibration are crucial for a fluid traffic.

## 23.7. Future Avenues

#### 1) Future avenues Implicit Coordination

Here, the detection algorithm could be improved. As described above, it starts to lag after a certain time and needs to be restarted time and again. Otherwise the algorithm is not very robust. Additionally, the tradeoff between false positives and false negatives could be tuned. Right now, the Duckiebots are far more likely to detect vehicles that are not there then to not detect vehicles that are there. While this makes sense in order to avoid collisions, it can also lead to a Duckiebot waiting for a long time at a free intersection. Also, the detection requires a lot of computational power from the Duckiebots that is currently not available other than on a laptop. This leads to the aforementioned lagging. Maybe, there is a different solution?

#### 2) Future Avenues Follow the Leader

An improvement for the Follow the Leader algorithm could be to git rid of the fiducial tags and try to follow each other solely depending on detecting the other Duckiebots. This could be done with the detection node we used for the imlicit coordination, however the detection is a lot less robust.

# Unit N-24
# PDD - Distributed Estimation

## 24.1. Part 1: Mission and scope

### 1) Mission statement

Enable Duckiebots to communicate with each other wirelessly

### 2) Motto

UBI UNUM IBI OMNES

(Where there is one, there is everybody)

### 3) Project scope

*What is in scope:*

**Communication in a centralized network:** *Communication between Duckiebots in one Duckietown* Define interfaces for communication *Performance testing on the communication system* One network for one Duckietown * TBD: does anything need to be synchronized?

+ Resource error

```
I will not embed remote files, such as https://github.com/duckietown/duckuments/blob/
devel-distribution-est-fleet-wireless-communication/docs/atoms_85_fall2017_projects/
16_distrubuted_est/Duckietown_Project_Image.png:
```

Figure 24.1. System Layout

**Option 1:** Communication in a centralized network with a redundant centralized component (multiple routers)

**Option 2:** Communication in a de-centralized network (ad-hoc)

*What is out of scope:*

- Shared network communication between Duckietowns in case of a centralized network
- Communication redundancy with another physical layer
- Data processing → Message data is just de-serialized and provided to other components

*Stakeholders:*

- Distributed Estimation
- Fleet planning
- Integration Heroes

## 24.2. Part 2: Definition of the problem

## 1) Problem statement

**Mission = Enable Duckiebots to communicate wirelessly**

**Problem Statement = Create a communication framework for the Duckiebots**

## 2) Assumptions

- Duckiebots can connect to a wifi network
- Data to be sent is already synchronized ???? TBD
- Duckiebots leave network when they are out of range or switched off
- If a Duckiebot is not connected to the communication network, it is not in Duckietown

## 3) Approach

**Step 0:** Define contracts with distributed estimation / fleet planning teams

**Step 1:** Create the communication framework for the duckiebots and test it on a centralized network

- Create wifi communication network (physical hardware e.g. wifi router, configuration, etc.)
- Create a software component that serializes data (create/format datapackets)
- Create a software component to send and receive messages to/from other duckiebots
- Integrate Serialization and Messaging software components into THE communication software component (e.g. into a ROS node)
- Testing:
  - Live visualization of bandwidth (and/or latency, etc. → Network performance measures) of network
  - Live visualization of network topology
  - Are any messages being dropped without arriving at their destination?
  - Etc.

**Step 2 (Optional):** Test the communication framework using a redundant centralized network

- Same as step 1, except for the wifi network → redundant centralized network

**Step 2 (Optional):** Test the communication framework using a de-centralized network

- Same as step 1, except for the wifi network → AD-HOC network

## 4) Functionality provided

- Enable duckiebots to join the network.
- Enable duckiebots to pack data and send it to other duckiebots in the same network.
- Enable duckiebots to receive and unpack data such that the data can be used in other software modules.
- Other functionality TBD

## 5) Resources required / dependencies / costs

**Bandwidth definition:**

- Number of duckiebots in the network
- Size of the messages
- Sending frequency
- Latency in message transmission
- Extra computation on duckiebots

#### 6) Performance measurement

- Visualize the network topology → number of duckies
- Visualize messages (wireshark) → message size, latency
- Visualize HW resources → processor, memory, etc.

## 24.3. Part 3: Preliminary design

#### 1) Modules

**Libraries:** *Serialization* Messaging

**Integration:** *Libraries integration into component* Communication framework into Duckietown (repository, duckumentation, etc. → contact Integration Heroes) * Creation of the WiFi network (centralized vs. decentralized)

**Testing:** * Measuring and visualization of performance metrics

#### 2) Interfaces

**Fleet planning** *Send data (SLAM, etc.)* Distribute data through the network

**Distributed Estimation** *Receive data (local maps built from each Duckiebots)* Send data (local maps and/or global map)

**Integration Heroes** *Duckumentation* Contract negotiation * External libraries (Protocol buffers, ZMQ, etc.)

#### 3) Specifications

- WiFi specs

#### 4) Software modules

- Library for Serialization
- Inputs:
  - Data to be serialized
  - Data to be de-serialized
  - Type definitions for serialization → Contract with distributed-estimation and fleet-planning teams
- Outputs:
  - Serialized data
  - De-serialized data
- Functionality:
  - Serialize data
  - De-serialize data
- Library for Messaging
- Inputs:

  - Destination
  - Port
  - Type of socket
  - Serialized data
  - Optional: priority
- Outputs:
  - Serialized data
- ROS node for messaging
- Inputs:
  - Messages from ROS topics
  - Messages from other duckiebots
- Outputs:
  - Messages to ROS topics
  - Messages to other duckiebots

#### 5) Infrastructure modules

- Wifi router for Duckietown
- Network configuration (centralized/decentralized)
- Testing
- Visualize the network topology → number of duckies
- Visualize messages (wireshark) → message size, latency
- Visualize HW resources → processor, memory, etc.
- ...

## 24.4. Part 4: Project planning

#### 1) Project plan

*Week 9: 13/11/2017:*

- **Tasks:**
  - Project kick-off and planning
- **Deliverables:**
  - Preliminary Design Document

*Week 10: 20/11/2017:*

- **Tasks:**
  - Contract negotiation with the relevant groups
  - Research on testing, redundant centralized and ad-hoc networking
  - Design and implementation of Libraries
  - Design and implementation of ROS node
  - Configuration of centralized network
- **Deliverables:**
  - Contracts

*Week 11: 27/11/2017:*

- **Tasks:**
  - Find suitable tools for testing and test the ROS node (incl. libraries) using these tools
  - Configuration of redundant centralized network
  - Code reviews

  - Duckumentation
- **Deliverables:**
  - Network configuration working
  - Libraries (tested)
  - ROS node (tested)
  - First test results

*Week 12: 04/12/2017:*
- **Tasks:**
  - Ad-hoc networking
  - Duckumentation
- **Deliverables:**
  - Redundant centralized network working

*Week 13: 11/12/2017:*
- **Tasks:**
  - Ad-hoc networking
  - Duckumentation
- **Deliverables:**
  - Testing "framework" complete

*Week 14: 18/12/2017:*
- **Tasks:**
  - Ad-hoc networking
  - Duckumentation
  - Solve networking problems
  - Testing
- **Deliverables:**
  - None

*Week "15" : 25/12/2017:*
- **Tasks:**
  - Ad-hoc networking
  - Duckumentation
  - Solve networking problems
  - Testing
- **Deliverables:**
  - Communicating Duckiebots over ad-hoc network

*Week "16" : 01/01/2018:*
- **Tasks:**
  - Duckumentation and Buffer
- **Deliverables:**
  - Duckumentation

### 2) Task distribution

- Libraries (incl. Testing): Luca and Antoine
- ROS node (incl. Testing): Leonie
- Testing of the framework: Pat and Francesco
- Redundant centralized network: Pat and Francesco
- Ad-hoc networking: Leonie, Luca and Francesco

### 3) Data collection

TBD –> other groups

### 4) Data annotation

No data to be annotated.

### 5) Relevant Duckietown resources to investigate

WiFi specs (duckumentation)

### 6) Other relevant resources to investigate

**Suggestions on Slack channel:** 1. MAVLink (born for UAVs also used for other robots) 2. ROMANO (not so good...) 3. DDS (standard for ROS 2.0)

**Computer Networks:**

- Introduction to computer networks: http://intronetworks.cs.luc.edu/

**Serialization and Messaging:**

- Protocol Buffers: https://developers.google.com/protocol-buffers/
- ZeroMQ: http://zeromq.org/

**Google python coding style guide:**

- https://google.github.io/styleguide/pyguide.html

**Redundant routers (rollover, cascading):**

- http://www.tomshardware.co.uk/forum/21591-43-design-redundant-wireless-network
- https://www.linksys.com/ca/support-article?articleNum=132275 (to be verified)

**Static code analysis in python:**

- https://www.pylint.org/

**Pylint configuration:**

- https://stackoverflow.com/questions/29597618/is-there-a-tool-to-lint-python-based-on-the-google-style-guide

### 7) Risk analysis

**Possible Risks?** *Network problems (ad-hoc: unstable network, low bandwidth, high latency, ...)* No ad-hoc network solution *Sizable amount of redundant data sent over wifi (chaos)* Synchronization

**How to mitigate the risks?** *Synchronization not part of networking → contract* Contracts to prevent redundancy

# Unit N-25
# Distributed Estimation: intermediate report

## 25.1. Part 1: System interfaces

### 1) Logical architecture

Robots can broadcast messages to every other Duckiebot in the network (centralized network for sure, and we'll try to make ad-hoc networking work (as ad-hoc or mesh network)). Robots will post the received message data to the corresponding ROS topic.

Ideally the network should scale from 2 to a town of Duckiebots. The network should also be robust to Duckiebots actively leaving and entering the network.

We will measure the latencies and other performance metrics but do not plan on optimizing the performance in regards of it. This can be improved over the next iteration.

Fleet communication: 1. Assume modules that need to use the communication capabilities e.g. multi-robot SLAM, fleet planning are able to publish and subscribe to ROS topics. 2. Assume modules sends data of serializable type and reasonable size 3. Assume communication is not time critical on sub second scale 4. Assume modules self synchronize (if applicable)

### 2) Software architecture

messaging node:

**subscribed topics:** individual outgoing communication (and by outgoing communication, we mean messages we send over wifi) topics published by: - fleet planning (TBD) - Multi-Robot SLAM (TBD).

These teams publish their data to be sent to these topics.

**published topics:** individual incoming communication (and by incoming communication, we mean messages we get over wifi) topics subscribed to by: - fleet planning (flag_fleet_planning_inbox) - distributed estimation (flag_multi_slam_inbox) - maybe other groups, since anyone can subscribe to these topics

Outwards (wifi) communication is realized with protobuffs and zmq

Optimizing for latency will be of low priority, since the primary single goal is to pipe through the data. If the data comes in faster that in goes back out from the ROS node, it shall be solved in a next iteration.

We're going to try and make the node configurable such that the code will not needed to be changed in the future (maybe just optimized, since it is quite complex and we do not have much time to implement it). If not, we will hard code the message conversion (ROS message ←→ ZMQ message).

How does this work?

Once:

- Team A wants to communicate between bots
- Team A tells us their message structure

- We build serialization
- We define ROS topics: teamAout, teamAin

Perpetually (as long as message structure doesn't change:

- Team A bot A posts message to teamAout
- We automatically serialize message with corresponding serialization
- We send on bot A
- We recieve on all other bots
- We deserialize and post to teamAin
- Team A other bots can retrieve message from teamAin

## 25.2. Part 2: Demo and evaluation plan

*Please note that for this part it is necessary for the VPs for Safety to check off before you submit it. Also note that they are busy people, so it's up to you to coordinate to make sure you get this part right and in time.*

### 1) Demo plan

Multi-Robot SLAM and fleet planning relies on communication to work, therefore the communication demo is implicit in the SLAM and fleet planning demo. We think a sole communication demo would not be too impressive to the casual demogoer.

At least three Duckiebots (configured for mesh networking i.e. with additional wireless adapters installed, if it works.) Otherwise, a global network (e.g. Duckietown) for the centralized structure.

### 2) Plan for formal performance evaluation

There are three main criterias that have to be evaluated: 1. Message transport: 1. Centralized Network: test if a simple message e.g. a string can be sent from one duckiebot to another reliably. 2. Decentralized Network: test message propagation. In a mesh network two Duckiebots (nodes) may not be directly connected, therefore we must test if a message can be propagated through the network to the correct receiver. 3. Check of dropped messages 2. Network traffic: accurately monitor network traffic 3. Network topology: visualize nodes entering and leaving the network reliably

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Message Transport | Messages cannot sent or ceived | Strings can besent and re-ceived on tcp | Messages canre-be serializedand sent andreceived tcp | Messages canserializedbe andmulticastand onand receivedon pgm | Messages can serialized and multicast and received on pgm on a mesh network |
| Network Traffic | No trafficmonitored | Can see trafficon the network but useful information ex- | Able to iso-late duckiebot traffic noi | Able to identi-fy specificpackets | Able to visu-alize routing of specific packages |

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Network Topology (centralized and decentralized) | Network cannot be established | tracted Initial Net-work can be established, but no nodes can connect to the network, not robust to con-nection loses | Net-Initial Net-work can be established, but not reli-ably, not robust to con-nection loses | Net-Initial Net-work can be established, new nodes can connect but not ro-bust to con-nection loses | Net-Initial Net-work can be established, new nodes can connect/can connect/leave dynami-cally, and ro-bust to con-nection loses |

## 25.3. Part 3: Data collection, annotation, and analysis

*Please note that for this part it is necessary for the Data Czars to check off before you submit it. Also note that they are busy people, so it's up to you to coordinate to make sure you get this part right and in time.*

### 1) Collection

Initially a set of compiled dummy messages is used to build the fleet-communication. For further implementation and evaluation one ROS-bag of broadcasted messages is needed from the fleet-planning team and from the multi-robot-SLAM product each.

### 2) Annotation

For the fleet-communication no data annotation is needed.

### 3) Analysis

UNIT N-26

# Fleet Messaging: final report

## 26.1. The final result

Demo Video

see the operation manual + Ref. error `I do not know the link that is indicated by the link '#demofleet-messaging'.` to reproduce these results.

README

## 26.2. Mission and Scope

With this project we enable Duckiebots to communicate with each other on centralized and decentralized wireless networks.

### 1) Motivation

In the previous state of Duckietown, Duckiebots were individual, autonomous agents, roaming around Duckietown with no way to communicate with each other explicitly (the only method in existence is with the help of LED patterns, resulting in long interpretation times). Since the ultimate goal being an automated taxi system: Duckiebots working together picking up and dropping off customers in the optimal way; the Duckiebots need to be able to communicate with each other efficiently.

One important part of this communication setup is that it can be decentralized, using a mesh network and Duckiebots can join and leave the system without putting the whole network at risk of failing. This also allows for the network to be scaled, given network limitations.

Due to the current state of Duckietown, the communication is needed, but not limited to, fleet planning control to coordinate the fleet in a town with a predefined map.

### 2) Existing solution

There was no prior work to build a communication system upon. Everything was implemented from scratch.

### 3) Opportunity

Without any existing work on wireless communication, we came up and built a whole new addition to Duckietown. We implemented a fleet-messaging package that builds an ad-hoc mesh network and lets other teams send messages

### 4) Preliminaries

We specifically picked libraries and modules that encapsulates their respective functionalities well. Therefore to fully understand what is going on under the hood, you simply need to read up on the documentation of each package used: -

batman-adv - zeroMQ - protobuf

## 26.3. Definition of the problem

The final goals of the project were to: 1. Create a robust wireless network that can easily be scaled to a larger fleet size and to a bigger Duckietown. 2. Build a communication framework for the Duckiebots that enables the sending and receiving of messages to and from any Duckiebot, which is connected to the above mentioned network. 3. Have a communication framework that is reusable and scalable.

For this we made the following assumptions: 1. Duckiebots can connect to a wifi network. 2. Duckiebots leave network when they are out of range or switched off. 3. If a Duckiebot is not connected to the communication network, it is not in Duckietown. 4. In a first step, the communication network is used by the fleet-planning team.

To evaluate the new framework we: 1. Compared the messages sent and received between two Duckiebots connected over the network and looked for messages dropped. 2. Tested the range of the wifi adapters to see if it is able to cover the size of a demo-sized Duckietown. 3. Test the robustness of the network by taking a Duckiebot out of range of the network and back and restarting the Duckiebot in to see if it would reconnect.

## 26.4. Contribution / Added functionality

### 1) Added Functionailities

Added Functionalities are as follow: - Duckiebots are now able to communicate with one another directly without a central station to relay the messages. - The network has no centralized point of failure. If a Duckiebot fails (runs out battery, breaks, disconnects abruptly) for any reason, the rest of the Duckiebots remaining in the system are still able to communicate without any noticeable disruption, given they are in range of each other.

**Consequently, we believe we have achieved the gold medal outcome for this project: “Ad-hoc networking”.**

### 2) Package Infrustructure

The package infrastructure is as follows: 1. A mesh network that dynamically connects all of the Duckiebots that are currently in Duckietown. 2. A messaging algorithm that allows the sending and receiving of messages over this network. 3. A message encoder that packs the data before it is sent. 4. A message decoder that unpacks the data after it is received. 5. A framework that allows other Duckietown packages to send and receive messages of their defined types.

These individual parts were implemented as described below.

*Mesh Network:*

Batman-adv is the backbone of the mesh network. In short, it is a specialized linux kernel module that implements a network routing protocol. It emulates a virtual network switch of all nodes participating. Hence, all nodes appears to be linked locally and are unaware of the network’s topology and is also unaffected by any net-

work changes. Once setup properly, batman-adv manages the mesh network for us.

*Messaging Algorithm:*

DuckieMQ is based on zeroMQ, a framework used to send messages over sockets. The serialized messages (protobuf) are broadcasted on a specified port into the network. For this to work, we need to know the name of the network interface, the desired port initialization of a messaging socket, which then can either be used as receiver or sender. Multiple sockets can and usually will run on one bot. Also because we use a multicast protocol (epgm) multiple sender and receiver sockets can run on one port.

Moreover, messaging features of the platform is decoupled from the implementation of the network achitecture.

*Message encoder and decoder:*

In order for the messages to be sent and received, they have to be encoded into ROS ByteMultiArrays. A serialization library was implemented to pack the ROS messages into a byte array such that the data could be sent through a zeroMQ message. After the message is sent, the data is then parsed back into a ROS message and published to the correct inbox_topic specified by the package that sent the message. We chose to use ByteMultiArray for its flexibility and because it is a `std_msg` of ROS. This means that other packages must only publish ByteMultiArray to fleet messaging.

*Framework:*

For easy use of the messaging algorithm, a ROS package with two ROS nodes was implemented. The two nodes are the receiver_node and the sender_node.

The sender_node subscribes to the outbox_topic and sends this data to the receiver_node on all other Duckiebots on the network via the messaging algorithm using zeroMQ. The receiver_node then publishes the received data to the inbox_topic.

To use the framework, one simply has to publish to the ROS topic outbox_topic (specified by the config file) and subscribe to the inbox_topic (also specified by the config file) and listen to the specified message port.

The complete structure of the fleet-messaging package is illustrated below. System Infrustructure

+ Resource error

```
I will not embed remote files, such as https://github.com/duckietown/duckuments/blob/
devel-distribution-est-fleet-wireless-communication/docs/atoms_85_fall2017_projects/
16_distrubuted_est/Simple%20Fleet%20Messaging%20Flow%20Diagram.png:
```

## 26.5. Formal performance evaluation / Results

There are three main criterion that have to be evaluated: 1. Message transport: 1. Centralized Network: test if a simple message e.g. a string can be sent from one duckiebot to another reliably. 2. Decentralized Network: test message propagation. In a mesh network two Duckiebots (nodes) may not be directly connected, therefore we must test if a message can be propagated through the network to the correct receiver. 3. Check of dropped messages. 2. Network traffic: accurately monitor network traffic. 3. Network topology: visualize nodes entering and leaving the net-

work reliably.

To test the first criteria: - Compared the messages sent and received between two Duckiebots connected over the network and looked for messages dropped

To test the second criteria: - Analyse packets sent on wireshark

To test the third criteria: - Tested the range of the wifi adapters to see if it is able to cover the size of a demo-sized Duckietown - Test the robustness of the network by taking a Duckiebot out of range of the network and back and restarting the Duckiebot to see if it would reconnect.

Our conclusions are summarized in the following table that was established for evaluation during the project phase. The performances in bold letters were the states the messaging-package achieved at the end of this project.

| | **0** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|---|
| Message Transport | Messages cannot be sent or received | Strings can be sent and re-ceived on tcp | Messages can be serialized and sent and received on tcp | Messages can be serialized and multicast and received on pgm | **Messages can be serialized and multicast and received on pgm on a mesh network** |
| Network Traffic | No traffic monitored | **Can see traffic on the net-work but no useful information extracted** | Able to iso-late duck-iebot traffic | Able to identi-fy specific packets | Able to visualize routing of specific packages |
| Network Topology (centralized and decentralized) | Network cannot be established | Initial Net-work can be established, but no new nodes can connect to the network, not robust to con-nection loses | Net-Initial Net-work can be established, but not reli-ably, not ro-bust to con-nection loses | Net-Initial Net-work can be established, leave dynami-cally, but not ro-bust to con-nection loses | Net-**Initial Net-work can be established, new nodes can connect/leave dynamically, and robust to connection loses** |

## 26.6. Future avenues of development

### 1) One push solution for package setup/installation

*Current Issue:*

Mesh networking can be very finicky because it depends on drivers for the wifi adapters and batman-adv working correctly. From experience, even with the unified Duckiebot hardware this it was very much a case by case basis. This made it difficult to develop a one push solution. Nonetheless we implemented one - see operation manual - which seemed to work on most occasion but we still had to do some on the spot debugging.

*Possible Solution:*

Use wireless adapters where the mesh network is currently working (edimax). A refined the bash script already implemented.

### 2) Improve Developmental setup

*Current Issue:*

When developing this package we needed three networks running simultaneously: one connected to the internet (for git purposes), one connected to the local network created by the Duckiebot (for ssh), and lastly the mesh network itself.

*Side note: Technically you can ssh into the Duckiebot through the mesh network, however this becomes problematic if you want to debug the mesh network itself. If your mesh network doesn't work then you can't ssh into your Duckiebot anymore. That is why it is better - at least when developing - to have three different networks running.*

*Possible solution:*

There are already two wifi adapters on the current configuration, and there lies the problem. There are two workarounds in use currently: 1. Use a centralized network created by an access point. This allows you to both ssh and develop your communication platform. However, it is no longer a decentralized mesh network. 2. Use an additional, third (mesh capable) network adapter strictly for mesh networking. There are several drawbacks. Firstly, this means that you need an additional wifi adapter the Duckiebot totalling up to three. Secondly, you also need one for your laptop to connect to the mesh network if you don't want to use your inbuilt adapter.

Both workarounds have their drawbacks, so it would be prefreable to find a robust solution for laptops connecting to the mesh network.

### 3) Reducing the number of additional hardware

*Current Issue:*

Following from the last point, adding an additional wifi adapter is costly.

*Possible Solution:*

We discovered late into the project that the edimax has mesh capabilities. We tried it and found that it works but never fully tested it to a point that we were confident with its viability.

*Remark: There were some driver problems with some WiFi adapters with respect to mesh network capabilities. It works with the edimax, so it might be of advantage to have two edimax adapters: one for the duckiebot and one for the laptop. With this setup, the edimax adapters can be used to create the mesh network (and the connected laptops would be a part of the network as well). It is also important to know that at this moment, it is not possible to get a connection to the internet through the duckiebot via the mesh network.*

### 4) Improving usability of platform

ZeroMQ allows to filter messages according to strings (eg. botname) at the start of messages. Our duckieMQ implementation already supports filtering, however it has not been used by the teams yet as the whole setup gets cumbersome to a cerain degree if it has to be implemented on every bot. Also serialized messages need to

be manually prepended by the filterstring. It would be useful to extend duckieMQ with the capability to prepend a filterstring to serialized messages automatically.

The configuration files for the different channels are somewhat cumbersome to create for bigger groups of bots. It would be preferable if the bot could create its own config file according to rules laid down in a central config file where all the different groups specify their communication architecture.

E.g fleet level planning want every bot to listen to port 23334, filter it with their name and publish to /taxi/commands and subscribe to /taxi/location and send it on port 23333.

As a final step we could let the config file generator handle ports by itself, so there is no need to keep track of used ports.

### 5) Network Visualization

*Current Issue:*

With the current implementation there is no way to visualize the topology of the network. This was very possible but unfortunately we ran out of time.

*Possible Solution:*

A very useful function would be to implement a real time visualization of the network. To visualize the network involves installing batadv-vis. Batadv-vis can be used to visualize the batman-adv mesh network. It reads the neighbor information and local client table and distributes this information via alfred - a user-space daemon for distributing arbitrary local information over the mesh/network in a decentralized fashion - in the network. By gathering this local information, any vis node can get the whole picture of the network. But this would have only taken us half the way there as it only gave static snapshots of the network. So to improve on this would be to continuously update/generate the graph so it appears to be live.

UNIT N-27

# Demo instructions Fleet Communications

This is the description of a communication setup between multiple Duckiebots.

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** At least two Duckiebots in configuration DB17-w or higher.

**Requires:** One additional wireless adapter per Duckiebot and laptop. (e.g. TP-Link TL-WN822N or TL-WN821N).

**Requires:** A laptop.

## 27.1. Duckietown setup notes

For this Demo, no Duckietown is needed.

## 27.2. Duckiebot setup notes

For this demo, additional wireless adapters are needed that allow mesh networking (e.g. TP-Link TL-WN822N or TL-WN821N).

## 27.3. Pre-flight checklist

This pre-flight checklist describes the steps that ensure that the installation and demo will run correctly:

Check: The additional Wifi adapter is installed and works.

```
$ sudo apt-get install build-essential linux-headers-generic dkms git
$ git clone https://github.com/Mange/rtl8192eu-linux-driver.git
$ sudo dkms add ./rtl8192eu-linux-driver
$ sudo dkms install rtl8192eu/1.0
```

Check: Duckiebots have sufficient battery charge.

## 27.4. Demo setup

+ Naming

```
More than one element with id
'sub:demo-template-run-249-demoInstructions_fleetcommunication-33102ba4'.
```

Some packages are needed to enable the communication beween the Duckiebots, namely Protobuf, ZeroMQ and B.A.T.M.A.N.

To install them, ssh into the Duckiebots and source the environment

```
$ cd duckietown
$ source environment.sh
```

pull the necessary files from devel-distributed-est-master.

Then find the name of the wifi interface you want to use with iwconfig. (eg. wlx7c8bca1120e0).

```
$ iwconfig
```

Next specify a static IP adress and subnet and write it on a piece of paper, be carefull to not use the same IP on two bots. However, the subnet should stay the same on all bots. (eg. 192.168.15.38/24)

Change to dependecie directory

```
$ cd ~/duckietown/catkin_ws/src/30-localization-and-planning/fleet_messaging/dependencies
```

and install everything with one handy script!

```
$ ./install_fleet_messaging <wifi-iface> <ipaddr>
```

Now you need to alter your network config, for this open the interfaces file:

```
$ sudo vim /etc/network/interfaces
```

Change all four instances of wlan0 to wlan1.

After a reboot you are ready to make your Duckiebots talk to each other.

+ Naming

```
More than one element with id
'demo-template-run-249-demoInstructions_fleetcommunication-33102ba4:section'.
```

## 27.5. Demo instructions

+ Naming

```
More than one element with id
'sub:demo-template-run-249-demoInstructions_fleetcommunication-33102ba4'.
```

To run the demo ssh into the bots, then in your duckietown repository run:

```
$ source environment.sh

$ roslaunch fleet_messaging tester.launch
```

and enjoy the show!

+ Naming

```
More than one element with id
'demo-template-run-249-demoInstructions_fleetcommunication-33102ba4:section'.
```

## 27.6. Troubleshooting

It's networking. If it doesn't work try reinstalling while letting 99 duckies swim in the bathtub and lighting magic candles.

## 27.7. Demo failure demonstration

terminal_full_of_errors.avi

# Unit N-28
# Single SLAM Project

## 28.1. Part 1: Mission and scope

### 1) Mission statement

Enable a bot to map duckietown and know its position in the town

### 2) Project scope

*What is in scope:*
- Map visualization
- SLAM super-class
  - EKF SLAM
  - Rao-Blackwellized

*What is out of scope:*
- SLAM with lane segments
- April tag detector
- Controlling the bot / finding path to do SLAM with

*Stakeholders:*
- April Tags
- Control

## 28.2. Part 2: Definition of the problem

### 1) Problem statement

Every Duckiebot is different in configuration.

> Mission = we need to make control robust to different configuration
>
> Problem statement = we need to identify kinematic model to make control robust enough

### 2) Assumptions

- Robot will move only in horizontal plane
- No lateral slipping of robot
- The body fixed longitudinal velocity and angular velocity will be provided as well as the timestamp of each measurement
- April tags don't move
- Gaussian errors for EKF slam (relaxed for Rao-Blackwellized)

### 3) Functionality provided

- A ROS node that subscribes to image feed and controls
- Publishes estimate of robot pose and uncertainty relative to starting point
- Publishes estimates of april tag poses and associated uncertainties relative to

starting point

### 4) Resources required / dependencies / costs

- Requires camera calibration
- Process model
- April tags
- A town

### 5) Performance measurement

- We will qualitatively evaluate the generated map (Visualization) against town
- For finer tuning we may consider pairwise distance between april tags and compare our estimate to the actual town

## 28.3. Part 3: Preliminary design

### 1) Modules and interfaces

- State data-structure holding poses of robot/features and associated uncertainties
- `Visualize :: State -> Image of map`
- `ProcessImage :: Image -> Poses` (in the map frame)
  - Get relative feature pose given robot pose
  - Add new feature to state
- `Predict :: velocity -> new robot pose`
- `Update :: list of poses, old distribution -> new distribution`

### 2) Subclasses and specific methods

- Kalman Filter
  - 3 sigma circles around feature/robot mean
- Particle Filter
  - Distribution of robot/features heatmap

### 3) Specifications

Duckiebots with different hardware configurations for testing

### 4) Software modules

- Parameter estimation:
  - runs calibration protocol
- Velocity translation: (Node)
  - get velocity as input and translate it to voltage as output

### 5) Infrastructure modules

None

## 28.4. Part 4: Project planning

What data do you need to collect?

### 1) Data annotation

Performances of the current implementation

*Relevant Duckietown resources to investigate:*

- Current State Estimation
- Calibration files

*Other relevant resources to investigate:*

Probabilistic Robotics Chapter 3, 10

### 2) Risk analysis

What could go wrong?

- We may not complete the project in the alloted time
- Uncertainty in map may be so high that it is useless

Mitigation strategy:

- Focus on EKF SLAM early

UNIT N-29

# Fleet Planning: Preliminary Report

## 29.1. Part 1: Mission and scope

### 1) Mission statement

"Create the mobility-on-demand service for Duckietown."

### 2) Motto

VICTORIA CONCORDIA CRESCIT

(Victory through harmony)

### 3) Project scope

*What is in scope:*

- Send mobility commands to each duckiebot
- Receive each duckiebot's location
- Calculate a set of mobility commands for all duckiebots
- Implement status LED patterns (i.e. like a taxi)
- (commands to arrange duckiebots in a pre-specified pattern)
- Implement customer request's for pickup and destination at a desired location

*What is out of scope:*

- Hardware modifications
- Fleet wireless communication
- Improvements to existing graphical representation of map and locations
- Fleet localization improvements (e.g. accuracy..)

*Stakeholders:*

- Fleet-comms: for querying their API (contact: )
- Devel-coordination and multi-slam:
- The Architects (smart city): accurate map of city, sufficiently big map to accommodate ~25 duckiebots at once

## 29.2. Part 2: Definition of the problem

### 1) Problem statement

We need to combine parts of many different stakeholders to achieve planning for a fleet of duckiebots.

### 2) Assumptions

- Sufficiently large duckietown to accommodate all duckiebots
- Collision avoidance and navigation works well to allow duckiebots to reach target destination
- Duckiebots can never park (i.e. stop still anywhere, unless waiting for other duckiebot at intersections etc.).

## 3) Approach

Figure 29.1. Fleet-level planning diagram

Necessary steps:

- See Part 4: Project planning

## 4) Functionality-resources trade-offs

Functionality includes:

- Visualization of N duckiebots
- Pick up and drop-off on request
- Functional standby distribution of duckiebots waiting for a pickup/ drop-off request
- Ability to arrange duckiebots in formations related to christmas videos
- Taxi status lamps

Metrics:

- Minimize time for each service request to be completed

## 5) Functionality provided

- Average handling time for each request
- Get the location for all duckiebots via the fleet-comms team and agree on the interface
- Get topological map of duckietown for planning

## 6) Resources required / dependencies / costs

- Calculate time taken to complete request
- Number of requests served per time

#### 7) Performance measurement

- Calculate time taken to complete request
- Number of requests served per unit of time

## 29.3. Part 3: Preliminary design

#### 1) Modules

See Figure 29.1.

#### 2) Interfaces

Duckiefleet - request handling server:

- List of duckiebots and corresponding locations and statuses - will be sorted with the fleet-wireless-communications team, see Resources required / dependencies / costs

Customer - request handling server:

- Pickup location and desired target location via clicking on map

Request handling server - Duckiefleet:

- List of target locations for each duckiebot such that request is completed
- Each duckiebot displays its status via its LEDs

#### 3) Specifications

No revision of existing duckietown specification necessary.

#### 4) Changes to existing Software

Revisit visualization of Duckiebots on map and adapt it for visualization of N Duckiebots

#### 5) Software modules

- ROS node for the request handling server
- Offers ROS service for customer requests
- (platinum) map on which customer can click to issue request, potentially as a separate ROS node

#### 6) Infrastructure modules

None

## 29.4. Part 4: Project planning

TABLE 29.1. FLEET-LEVEL PLANNING PROJECT PLAN

| Week of | Task | Deliverable |
|---|---|---|
| 13.11.2017 | Project kick-off and planning | Preliminary Design Document |
| 20.11.2017 | Look at state of current infrastructure | Running visualization of 1 duckiebot on map as currently implemented |
| 27.11.2017 | Visualization of n duckiebots | |
| 04.12.2017 | Mission planner, implement m-stochastic queue median policy (or similar, tbd with Claudio) | |
| 11.12.2017 | ...continued | Run test cases (e.g. send n reference locations to n duckiebots) |
| 18.12.2017 | ...continued | Run test cases (e.g. send n reference locations to n duckiebots) |
| 25.12.2017 | Implement customer request handling | Run test cases to establish reliable customer request handling routine |
| 01.01.2018 | Physical visualization of status, ETH formation | Verify that it works |

### 1) Data collection

None

### 2) Data annotation

None

*Relevant Duckietown resources to investigate:*

According to meeting notes:

- Click and send for a single duckiebot is (probably) possible –> find corresponding code
- Graphical representation is running –> find corresponding code
- Read current documentation on tile-level localization
- We want to be able to send a go-to-position to a Duckiebot.
- Already implemented. Video: https://www.dropbox.com/s/93pbcktmwln4fqo/dp6b-draft.mov?dl=0
- DP6 : link to the report
- Navigation to a point already implemented –> look at the code
- Visualization of 1 Duckiebot on a 2D map
- Look at code from Claudio re m-stochastic queue median policy + paper

*Other relevant resources to investigate:*

Papers:

"Fundamental performance limits and efficient polices for Transportation-On-Demand systems" by Marco Pavone, Kyle Treleaven and Emilio Frazzoli link

### 3) Risk analysis

- Dependency on the Fleet-communications project. Closely work together with that team to get notified early about any upcoming problems that could delay the delivery of the needed parts for this project.
- See Part 4: Project planning

# Unit N-30
# Fleet Planning: Intermediate Report

This document describes system that the Fleet-planning team is planning to implement. Part 1 describes the interfaces to other parts of the system. I.e. How it communicates with all of them. In Part 2 a plan for the demo and evaluation is presented. And finally part 3 focuses on data collection, annotation and analysis. As for this project not much annotated data is used, part 3 is rather short.

## 30.1. Part 1: System Interfaces

### 1) Logical Architecture

The fleet planning system shall provide a graphical interface for visualization of n Duckiebots in a Duckietown. Duckiebots shall provide their location regularly to a central "planning node" (not running on a Duckiebot). Furthermore, an interface should exist to generate "taxi" commands (i.e. pickup guest at tile k and bring him to tile m). For such a request the system shall react with a command sent to one of the duckies to pick that customer up and transport him.

See Figure 1 for an overview of the logical structure of the fleet planning system.

In detail the complete process consists of the following steps:

1. Enter desired pickup and drop off location in GUI.
2. Planning node (see Figure 1) finds a Duckiebot that is available for a pickup. The matching is based on the availability of Duckiebots and their current locations. The planning node knows the location of each Duckiebot because they broadcast their position on a regular interval.
3. The location of the pickup is sent to the selected Duckiebot and that Duckiebot changes its fleet planning status to picking customer up as soon as it receives the message.
4. Based on the Duckiebot's location and the received target location, the Duckiebot calculates locally a shortest path. This shortest path is nothing else than a list of directions for all the intersection that will be crossed on the path.
5. When the Duckiebot arrives at an intersection (realized by listening to flag), it publishes the instruction for this intersection.
6. Once the Duckiebot arrives at the location where it picks up the customer, the planning node sends the target location to the Duckiebot. Then steps 3, 4, and 5 are repeated until the Duckiebot arrives at the dropoff location.
7. The fleet planning state of the Duckiebot is set back to rebalancing.

Figure 30.1. Fleet-level planning Logical System Architecture

*Assumptions:*

The communication between Duckiebots and the central planning node relies on the communication team of the distributed estimation project. To exchange messages on a fleet level we need this system to work reliably (i.e. no message loss) and with as little latency as possible (i.e. as little delay as possible between sending and receiving a message). We assume that the distributed estimation team can provide such a system within a reasonable timeframe. In part 2 of this document the interface to the communication layer is described. Based on this interface we can mock the communication and work out the fleet level planning part without a already working communication.

We further assume that the system has a map of Duckietown available. This map can be created by hand or with the system implemented by the distributed estimation team.

As described by the preliminary design document, the localization of the Duckiebot within Duckietown is out of scope. We utilize the existing april tag based localization from last years Duckietown course. First experiments are very promising and give good results. So we assume that the Duckiebot is able to localize himself within Duckietown if it is given a map of Duckietown. Furthermore, the localization can be enhanced by using the current speed information from the controllers. With that we can get an estimated position between intersections.

General assumptions that collision avoidance, line detection etc. function flawlessly

are also made. Furthermore we ignore parking spots and parked Duckiebots and possible parking actions as they are not represented in our map. Lastly we assume that the clocks of the Duckiebots are reasonably in sync.

## 2) Software Architecture

*New Nodes:*

**Fleet planning node [central Laptop]**

This node knows the position and status of each Duckiebot in the network. It does the actual planning for the fleet. This consists of matching incoming transportation requests with available Duckiebots in such a way that the overall fleet is used in an optimal way.

Subscribed Topics:

- "Location": To get the location messages from every Duckiebot.
- "Transportation Requests": Every transportation request posted on this topic should be handled by the fleet planning node.

Published Topics:

- "Transportation status": A topic that will get messages whenever something is updated within the fleet planning node. Example messages ""Robot" Picked up customer x at y", ""Robot" Received transportation order from k to m". This topic will introduce neglectable latency. As soon as the information is acquired from the sources it will post the message to this topic.
- "Target Location": here messages are published that contain a robot name and its target location. Based on this the robot will calculate its path to the target location locally. The latency between getting a transportation request and sending out a target location to one of the robots can not be determined offline as it is dependent on the current state of the system. If there is a Duckiebot immediately available, there is no delay. However, it might be that all Duckiebots are busy and therefore no Duckiebot can be assigned to that target location at the moment.
- "Fleet planning active": Boolean flag indicating that the fleet planning node is active and wants to actively provide instructions at intersections.

**Visualization Node [central Laptop]**

This node is responsible of visualizing n Duckiebots on a map.

Subscribed Topics:

- "Location": The stream of locations that comes in from all the Duckiebots.
- "Target Location": Combining this knowledge with the messages from the "Location" topic, the calculated path of each Duckiebot can be visualized. The user can decide if the paths shall be shown. (The shortest path is also calculated locally on the Duckiebot, in order to reduce communication overhead).

Published Topics:

- "Visualization": The rendered visualization as an image

**Taxi command execution node [local]**

This node will run on the Duckiebot and listen to commands from the central fleet planning node. Whenever it receives a command it starts the appropriate actions.

A Duckiebot can be in one of three fleet-planning states: - Cruising/Rebalancing - Picking up Customer - Transporting Customer

A Duckiebot receives target locations from the central fleet planning node. It then calculates the shortest route to this location. For this the existing A*-path planning node is used. Given the Duckiebot's current location and the target location, the path planning node can calculate instructions for how to get there. These instructions are then passed (on a per intersection basis) onto lower level navigation nodes (i.e. handled by navigator team).

Furthermore this node also handles the back right LED which we are allowed to indicate the taxi status of the Duckiebot. Its status is communicated by the central fleet planning node. Additionally, when a customer is picked up a pattern is played on all the LEDs with very low intensity.

Subscribed Topics:

- "Location": The location of the Duckiebot on the map
- "Stopline" and "April tag": Whenever we are at a stop line with the according april tag we know that we are in front of an intersection. As a reaction to being at a stop line this node will publish the instruction that tells the Duckiebot what to do at this intersection. This instruction is based on the path it had calculated to reach its target location.
- "Taxi status": indicates if the taxi is driving to a customer, is carrying a customer or is in idle (cruising/rebalancing) mode.

Published Topics:

- "Crossing_instructions": Whenever the Duckiebot is at an intersection the node will publish on this topic what direction should be taken. As the path the Duckiebot takes was calculated previously there is no latency introduced. I.e. as soon as the Duckiebot gets the message that it stopped at an intersection it will publish the message with the instruction for this intersection to this topic. The message consists of a single integer. To have backwards compatibility with the current system this is one of the following values:
  - 0: left turn
  - 1: straight
  - 2: right turn
  - 3: random

*Modified Nodes:*

Localization is based on last year's "localization" package. For this purpose the map data was updated to match this year's Duckietown.

The fleet planning package is also based on last year's "navigation" package. It provides software to handle the path planning and a GUI that allows to select start and target nodes and displays the calculated path for a single Duckiebot. Multiple Duckiebot handling does not exist. The Duckiebot is then made to follow these commands. By now, we were not able to reproduce this feature in a stable manner. Also, in this package no location information is taken into consideration, the path planning and execution is executed in an open-loop manner. This shall be closed loop this year.

## 30.2. Part 2: Demo and evaluation plan

### 1) Demo plan

The demo from last year consisted of a single Duckiebot. It was possible to click on the node in the graph where the Duckiebot currently is and a target now where the Duckiebot should go. A path planning node then calculated a shortest path to the target location and the Duckiebot drives to that location.

For this years demo we envision a system that builds on top of that. A map is presented to the user that contains the current locations of all Duckiebots. The user can generate a transportation request by using a GUI. The system then assigns one of the Duckiebots to that task. That Duckiebots drives to that location, picks the customer up, drives to the target location and drops the customer again. The pickup and dropoff action are visible in the visualization. Further, the pickup and drop off can be visualized using the LEDs by showing a fancy pattern. There is no physical interaction planed. The system will be able to handle multiple of such requests at the same time. Also, the system shall be robust in the face of dying Duckiebots (may they rest in peace), thus a customer shall be assigned to a new Duckiebot if the original one is lost on it's way. A Duckiebot counts as out of service if he does not publish a new location within a certain time window. No customers waiting forever. Unfortunately we cannot guarantee safety for a customer that is on a lost duckie-taxi.

Setting up this demo is as quick as starting all Duckiebots with the correct mode of operation and putting them on the map.

**Required Hardware**

- Duckietown (At least the same size as ML J44, depending on number of Duckiebots on duty)
- Several Duckiebots
- One laptop to act as the central fleet planning server (provided by fleet-planning group member)

## 2) Formal performance evaluation

This project will introduce metrics that will be used to evaluate the performance of the fleet planning, providing a baseline for future groups working on further optimization. The metrics, introduced below, will be applied to a test-setup, which is described in the "proposed performance evaluation" section. The aim of the setup is to test the system's reliability on the one hand (first scenario) and the capability to handle multiple requests at very high frequency and at the load limit, that is, at full capacity (number of requests at a given time == number of Duckiebots) on the other hand.

**Customer requests fulfilled per minute for given number of Duckiebots (ie. throughput).** This metric measures how many customer request are handled by the server and fulfilled by all Duckiebots combined. This would allow for the evaluation of different path planning algorithms and testing how well the rebalancing works.

**Mean distance of closest Duckiebot to the origin of a request for a set of requests and a given number of Duckiebots.** In the optimal case the Duckiebots will be distributed as homogeneously as possible in the Duckietown, minimizing the expected distance to each customer. This metric will enable the evaluation of how well a given implementation does this.

**[Stretch goal] Lost customers** In certain circumstances Duckiebots will fail (e.g. battery dies, Duckiebots gets stuck, etc.) and if this occurs while a Duckiebot is on the way to pick up a customer the fleet planning system should be able to send another available Duckiebot to fulfill the request instead. For the purpose of evaluation,

several Duckiebots can be removed from the Duckietown at random and the system should still be able to fulfill all open requests. (A lost duckiebot can be detected using a timeout on the localization.

*Proposed performance evaluation:*

In the Duckietown in ML-J44, that is a 5x6 Duckietown, 4 Duckiebots will be placed at the intersections in the following locations:

```
- (0,3)
- (2,3)
- (4,3)
- (2,5)
```

See picture below for initial locations.

The Duckiebots should operate at a predefined speed which is consistent across all tests for comparability. There should be two scenarios running 10 minutes each with 10 customer requests per scenario.

Scenario 1: 10 requests evenly spaced out across 10 minutes. Scenario 2: 4 requests within the first minute, then 6 requests at 4,5,6,7,8,9 minutes respectively.

For each scenario, the method is the following:

1. Place Duckiebots in the Duckietown and have them move around in an idle state (i.e moving around randomly).
2. Send a command to move all Duckiebots to predetermined locations to ensure repeatability of the performance evaluation. The locations are
   - (0,1)
   - (2,5)
   - (4,1)
   - (1,5)
3. Start the evaluation by sending the series of customer requests during a 10 minute interval.

Figure 30.2. Initial position

Figure 30.3. Final position

## 30.3. Part 3: Data collection, annotation, and analysis

### 1) Collection

We need data to do the formal performance evaluation. As the central fleet planning node has all the information about the Duckiebots (i.e. location at every point in time and taxi status) it is enough to log the information flowing through the topics to and from the central fleet planning node.

These logs will be used for the formal performance evaluation as described in Part 2 of this document.

### 2) Annotation

None needed.

### 3) Analysis

Analysis is done by hand on the acquired logs. As a stretch goal, a set of functions is made available that automates the process such that future teams working on im-

proving this system can use the same evaluation strategy.

UNIT N-31

# Fleet Planning: Final Report

## 31.1. 3.1 The final result

Figure 31.1. The Fleet Planning Demo Video

See the operation manual to reproduce these results.

## 31.2. 3.2 Mission and scope

### 1) 1 Motivation

As cities such as Duckietown grow, their inhabitants can no longer walk from each point of the city to the next; especially after a long night out. With the tremendous growth of Duckietown, the need for a mobility-on-demand (MOD) service became unbearable and was addressed in this project. Duckietown being a city in which autonomous Duckiebots roam the streets, it makes sense for them to provide the much-needed MOD service. If Duckiebots share their positions as well as their next destinations, with a central dispatcher node they can work together to deliver an efficient service. The development of a multi-duckiebot MOD service is thus reliant on many other Duckietown components such as localization, mapping, navigation, collision-avoidance and communication.

### 2) 2 Existing solution

The existing codebase was set up to support localization, navigation and visualization for a single Duckiebot in a fixed, hard-coded Duckietown, represented as a tile-map image; see the specification for the available tile-types.

By combining a set of rotated tiles, a map of any given Duckietown can be formed. In the existing codebase, such a tile-map was supplied as an image and a CSV file of tile-types and orientations; however neither code for composition of a tile-map nor documentation as to how this was generated in the first place were supplied.

The entire code for localization, navigation and visualization ran on the Duckiebot and was linked to the other modules using a Finite State Machine (FSM).

The codebase can be found here.

*Localization:*

Localization was performed by placing AprilTags at each intersection and having

the Duckiebot identify each unique AprilTag through image analysis. The AprilTags are defined in the duckiebook signage section. A Duckiebot could thus compute at which (x,y) coordinate of the map it was and estimate its rotation. The position on the topographic map was not mapped to the corresponding topological graph representation which is required for path planning.

*Navigation:*

A graph was computed from the tile-map and given a start and an end node, A* search was performed on that graph to provide the list of nodes along the shortest path. This list of nodes was then converted into a set of instructions the Duckiebot could follow. These instructions were of the form “straight”, ”left”, ”left”, ... They were executed in open loop manner, i.e. once the entire path was calculated, the Duckiebot was sent off to execute it without any intermediate checks whether the Duckiebot was doing this correctly. Thus any deviation of the Duckiebot from the path, or noisy FSM-transitions led to failure of the system. The python package graphviz was used to compute the graph and generate a visualization, i.e. an image of the graph.

*Visualization/GUI:*

The GUI consisted of a list, from which the user could select the desired start- and destination nodes for one single Duckiebot as well as an overlay of the tile-map and the graph image. The location was indicated by highlighting the corresponding node in a different color, and the path by highlighting the graph edges. No localization was integrated in the demo, and thus the start node had to be set manually and the subsequent execution of the path was completely open loop.

When the user hit the start button, a request was sent to the navigation ROS node, upon which the start and end graph nodes were highlighted and the computed path was indicated on the image.

### 3) 3 Opportunity

The existing solution could benefit from scalability, adaptability and localization. It has been set-up, and often hard-coded, to work only with a single Duckiebot in a completely open loop manner. Without human interaction the Duckiebot would not move.

For fleet-level behaviour and our MOD service, the solution needed to be scaled up to work with n Duckiebots. Also, in order to being able to implement sophisticated fleet planning algorithms, an API handling and tracking the statuses of the Duckiebots and customer requests appeared highly desirable.

Building this infrastructure was a priority to us, since a good software foundation will make future endeavours in the field more productive and will allow to focus on more complex fleet planning concepts. Nevertheless, inspiration for intelligent fleet-level behavior and automatic rebalancing for more efficient request handling were drawn from [1].

Critical components such as the request-handling were desired to move away from running on a single duckiebot. The previous solution also had no stack and could thus handle only one request at a time. A new request simply overwrote its predecessor, which led to incomplete jobs. These were unacceptable limitations to a true MOD service and could be overcome through the implementation of a central node dealing with coordination, called the taxi central node.

In coordination with the fleet-communications team, a method of sharing information and sending commands to multiple Duckiebots was introduced. The GUI was also changed so as not to run on the Duckiebot, but rather directly on the customer's machine to allow multiple customers to utilize the service at the same time.

Furthermore, the existing solution had been set up to work with one specific Duckietown layout only. Since a MOD service should be capable of running in every Duckietown, this issue was also addressed. The tile-map and graph are now automatically generated off of a CSV based description, allowing for easy adaptations to an existing Duckietown and the application of our MOD service in new Duckietowns.

The GUI was also rewritten to scale well with varying Duckietowns and number of Duckiebots and now allows the MOD customer to issue requests more easily. The GUI workflow was also improved with respect to intuitive usability and faster issuing of customer requests. It also has the capability to show all available Duckiebots and their locations. The improved solution is also more flexible with regards to localization. Once this is properly merged with the localization teams' efforts, the resolution of localization will be dramatically improved.

To verify our work, a test environment and virtual Duckiebots were introduced. The previous solution had no means of demonstrating functionality virtually; i.e. without a physical demo.

## 31.3. 3.3 Definition of the problem

### 1) 1 Problem Statement

Integrate expectations of different stakeholders to achieve planning for a fleet of Duckiebots. This can be broken down into the following tasks:

- Receive location information and status from n Duckiebots
- Visualize n Duckiebots on the current Duckietown map
- Receive and process pick-up and drop-off requests which are issued using a GUI
- Assign customer requests and rebalancing targets to Duckiebots
- Standby distribution ("rebalancing") of Duckiebots waiting for customer requests
- Send target location (customer requests or rebalancing targets) to n Duckiebots
- Indication of each Duckiebot's status using their LEDs. A Duckiebot's status indicates whether it is on its way to a customer, currently transporting a customer or idle.
- Local path planning and execution on Duckiebots

### 2) 2 Assumptions

Per the preliminary and intermediate design documents, we assume that the Duckietown is large enough to accommodate all Duckiebots and that collision avoidance, line-detection etc. function reasonably well. We further assume that the system has a CSV map representation of Duckietown conforming to the conventions defined here This map can be created by hand or with the system implemented by the distributed estimation team. The communication between Duck-

iebots and the central planning node relies on the communication team of the distributed estimation project. To exchange messages on a fleet level we need this system to work reliably (i.e. without message loss) and with sufficiently small latencies, i.e. less than roughly a second.

#### 3) 3 Contracts

*Distributed estimation and communication team::*

The fleet communication team provides a means of transporting arbitrary data as a byte array from one Duckiebot to another Duckiebot or to a computer. Communication channels are set up via a configuration file and messages are sent/received by publishing/listening to messages on a ROS topic. This was integral to the functioning of our system, as we required multiple ROS master nodes running on each Duckiebot locally, nonetheless sharing their information with a separate ROS master on a central computer.

*The Architects::*

Initially the idea was for the Architects to design a Duckietown sufficiently large to accommodate a large number of Duckiebots. Closer to the demo day, it emerged that several smaller Duckietowns would be used so this was no longer needed.

#### 4) 4 Performance Evaluation and Metrics

The performance of our project was primarily evaluated qualitatively, since the focus of the project was on providing a working framework to easily implement further fleet planning strategies. The results are discussed in 3.5.

## 31.4. 3.4 Contribution / Added functionality

The work on the 2017 fleet planning project was distributed in the following manner: 80% software infrastructure, 10% algorithms and 10% package integration of other teams. The final software can be divided into five parts:

#### 1) 3.4.1 Path planning and execution

Main component can be found here.

This nodes listens to location updates from the april tags localization package and target destinations from the taxi central node.

The (x,y) location information is then mapped to the topological graph representation. Using the rotation of the Duckiebot and by finding the red line with the minimum distance to the Duckiebot, the location of the Duckiebot is set to the node that best explains this configuration.

Once the node has received both location and a mission target, it executes A* path planning and publishes the next intersection instruction, i.e. “left”, “right”, etc., to be received by the intersection navigation package (implemented by the 2016 team). The path is recomputed at each intersection such that deviations from the original plan do not lead to failure of the entire system; this guarantees a certain robustness to errors of other software components. The calculated path and current location is then reported to the central planning node, called taxi central. This node runs locally on each duckiebot

### 2) 3.4.2 Fleet planning aka taxi central node

Core component can be found here. Duckiebot and Customer logic is defined here.

- Runs on a central laptop, handles all the fleet planning logic
- Maintains a set of active Duckiebots.
- Receives location updates from each Duckiebot and stores them. If a Duckiebot does not update its location within a certain time window, it is considered dead and removed from the map. Customers onboard are re-assigned to a new Duckiebot, with their pick-up location corresponding to the last known location.
- Receives customer requests from GUI, assigns them to a nearby Duckiebot based on FIFO breadth first search. For details, see below.
- Tracks execution status of customer requests, and thus tracks whether customer start or target location has been reached and stores the timestamps for each state transition and each request. This allows evaluation of fleet planning metrics such as time-to-customer or execution times of the whole request.
- Duckiebots that are not busy are assigned rebalancing goals. These are currently random locations on the map. This appeared to be a reasonable probabilistic approximation to an optimal strategy, assuming a large number of Duckiebots on a uniform map which will then spread rather homogeneously. The software was designed in a way that allows easy extension of the current approach. Also, it is possible to switch between different approaches by setting a single enum variable which can be very useful for testing and comparing rebalancing strategies.

**FIFO breadth first search:** The taxi central stores a list of pending (i.e. not yet assigned) customer requests and idle Duckiebots. Our algorithm then selects the closest Duckiebot to each customer in First-In-First-Out manner:

For each customer in pending_customer_requests:

- Find closest Duckiebot via breadth-first search on the map graph
- Assign customer to Duckiebot
- Remove Duckiebot from list of idle Duckiebots

Although this approach is not fully optimal, it is a reasonable approximation for the 2017 Duckietown setup with a low frequency of new customer requests and a small number of operating Duckiebots. Our software is designed in a way that makes it easy to add more sophisticated approaches. This could include strategies that take into account expected distributions of customers and send Duckiebots to anticipated hotspots ahead of time (e.g. dealing with rush hour customer spikes).

### 3) 3.4.3 GUI

The code can be found here.

Since the existing GUI was running directly on the Duckiebot and was laid out for a single Duckiebot system, it had to be completely rewritten. To design the front end, the QtGui module was utilized; the GUI itself runs as an rqt module.

To keep the GUI scalable and extensible along with the rest of our solution, it is able to run on multiple devices at the same time, as long as each device can communicate with the ROS master that the taxi central node is running on. The GUI communicates with other modules through ROS messages and topic listeners/subscribers and runs largely independently of all other components of the fleet planning module.

The source code is located in this following folder.

The source code is located in this following folder.

In this section, the GUI components and their interactions with the other modules are described. The overall layout follows design principles outlined in Galitz' "The essential guide to user interface design: an introduction to GUI design principles and techniques" [2].

Please note that components (2) through (5) are re-positioned depending on the Duckietown map's size.

GUI without customer.

5
3
1

Figure 31.2. Map of Duckietown in GUI showing Duckiebot _Harpy's_ current location.

GUI with assigned customer.

5
3
1

Figure 31.3. Map with icons for a customer at node 7. Duckiebot "Harpy"'s target location is also at node 7 to pickup the customer.

GUI with assigned customer.

5
3
1

Figure 31.4. Harpy travelling with the customer to the target location.

*1 Duckietown Map:*

- A map of the current Duckietown as an image, received from the drawing node
- Displays a selected Duckiebot's location
- Displays the start and target location of the user's last issued request once the start button (4) has been hit
- Displays the calculated route between start and target locations
- The user can intuitively set start and target location by simply clicking on the map
- The first click sets the start location
- The following click sets the target location
- Both can be cleared by clicking the clear button (5)

*2 Display of start and target location:*

Serves to provide the user with feedback on the state that the GUI is in Also provides a way to check correctness of the start/target location before issuing a request

*3 List of active Duckiebots:*

The selected Duckiebot's location is displayed on the map (1) Used for testing and debugging during development List received as ROS message from the taxi central node

*4 Start button:*

Triggers a customer request to the taxi central node Only triggered if start and target location are set

*5 Clear button:*

Clears the start and target locations, which are then removed from the map (1) as well as the numerical display (2)

### 4) 3.4.4 Map drawing

Code can be found here.

Code can be found this file.

The map drawing node deals with drawing the Duckietown map according to the specifications in the csv file, overlaying the graph on top of the map and drawing the active Duckiebots at the correct locations. As localization only occurs at intersections where Apriltags are located, Duckiebots are only ever drawn at intersections. The Duckiebot is identified by it's name, displayed below the Duckiebot icon. When a customer request is assigned to a Duckiebot, a customer icon is drawn at the specified location and an icon is displayed at the target location as well. Once the customer is picked up, his or her icon is drawn alongside the Duckiebot acting as a taxi. See screenshots above for the different states.

### 5) 3.4.5 Messaging / Serialization

As described in a previous section, the existing system runs completely on the Duckiebot, including the user GUI. To make the system scalable we needed to have communication between multiple Duckiebots as well as a central planning node. This change in the architecture requires a communication system for reliable communication. We acquired this functionality by setting up a contract with the fleet communication team. See here.

The outcome was a system which consists of one ROS node on each participant in the network. Via a configuration file you can define which node communicates with which other node. The interface the fleet communication team provides accepts a byte array and transfers this byte array to the endpoint of the communication channel. For a more detailed explanation of how this is transferred we refer readers to their final report

To send data such as target locations and localization results a way to serialize this data to a byte array was needed. A general framework was setup to serialize the basic data types using python's pickle [3] module. Based on this we implemented serializer and deserializer classes specifically for the messages we needed to send over the network.

### 6) 3.4.6 Virtual Duckiebot / Simulation

Only at the very end of the project all projects we depend on reached a state where we could integrate them all to have a functional system. Therefore we needed a way to test the system, especially the central dispatcher node, without relying on physical Duckiebots. We solved this by implementing a virtual Duckiebot ROS node that acts as if it were a real Duckiebot.

Duckiebots mainly perform two actions, they report their location to the central dispatcher node and they receive commands (customer pickup, target location, ...). The virtual Duckiebot node mimics this behaviour by regularly sending a message with the current position. Additionally it prints all the information it receives and sends to the console for easier debugging. To the central dispatcher node it looks as if it were interacting with a real Duckiebot.

The virtual Duckiebot node can be run in one of two ways:

*Manual mode::*

The virtual duckiebot is started with an initial location. The user uses a ROS service call provided by the virtual Duckiebot to tell it to send a location update message. The user can specify the location it should send. This mode gives power to the user to test specific situations.

*Autonomous mode::*

In this mode, the user only adds a Duckiebot at a desired node. The virtual Duckiebot node then takes care of all the things a real Duckiebot would do: it receives target locations from the taxi central, calculates the path it should take, notifies the taxi central of the calculated path, and then every few seconds it publishes a location update, as if it were really following the calculated path. The user can add as many Duckiebots as desired and can remove them as well. This way, the whole fleet planning software can be tested and simulated from a single laptop, without the need of a Duckietown or a Duckiebot.

## 31.5. 3.5 Formal performance evaluation / Results

As mentioned previously, the largest portion of the work that needed to be done involved implementing an operational infrastructure that supports actual fleet planning functionality. In summary, this included:

- Generating a Duckietown map from a CSV description of the map Tracking n Duckiebots on the map, and displaying their locations, their targets and their

paths

- Creating customer requests through the GUI by clicking on the image of the map directly, thus removing the need to use tedious drop-down menus
- Tracking customer requests and their assignments through completion (useful classes for easy debugging and clean coding)
- Visualization of the taxi status directly on the Duckiebots via LEDs
- A virtual Duckiebot node, that simulates a real Duckiebot in order to test the fleet planning software.

These functionalities can primarily be evaluated in a binary manner, observing whether these components work or not. Furthermore, the overall performance can be analyzed in a qualitative manner. The components described above all work in the set up used for the 2017 ETH demo day. Our fleet planning system is robust to malfunctions in the execution of the planned path (i.e. the system can re-compute paths and is tolerant towards situations where Duckiebots deviate from the optimal trajectory). Furthermore, the loss of a Duckiebot during the transport of a customer is covered by assigning a new, nearby Duckiebot to pick up the customer from where he or she was last seen.

In comparison to the state-of-the-art, where no fleet planning was possible and visualization of only a single Duckiebot on a map was implemented, a large number of new features was added and the infrastructure put in place to develop more advanced re-balancing algorithms.

Our system is relatively high-level in the sense that it requires many other Duckiebot components to work smoothly for successful operation. For example, we need the Duckiebots to reliably find the correct Apriltag at an intersection and the turns at an intersection to work flawlessly. The principal components our project requires are lane following, intersection navigation, collision avoidance, localization and fleet communication. While our system simply ignores malfunctioning Duckiebots and drops them from the roster of active Duckiebots, it is nonetheless important to ensure most Duckiebots work as expected. On a map the size of the 2017 demo day collisions are inevitable if Duckiebots do not stay in their lane and relatively quickly manual intervention using a joystick becomes necessary. Of course, during manual operation of a Duckiebot the fleet planning system cannot meaningfully assign customers to Duckiebots. Thus, if manual control becomes necessary too often, fleet planning cannot do its job.

At the time of the demo day most other projects of this year that we relied on were not yet ready so we resorted to using last year's implementations for some components, notably (open-loop) intersection control, lane following as well as last year's FSM.

What this meant is that very often the system did not function smoothly and manual intervention was often necessary when Duckiebots missed a turn or lane following abruptly stopped working. Coupled with mercurial joysticks this made testing of new features challenging at times. Initially, we had expected to spend far less time on integrating previous features and get the all running in parallel. Instead, we had thought we would need more time advancing the capabilities of the fleet planning package itself. In the end, a basic Duckiebot that could navigate and communicate with reasonable reliability was paramount to doing any development on our package.

Quantitative evaluation of the sort initially planned and described in the PDD did

not lead to any sensible results and insights. The metrics suggested were the 'customer requests fulfilled per minute for given number of Duckiebots' and the 'mean distance of closest Duckiebot to the origin of a request for a set of requests and a given number of Duckiebots'. Both require a large map, a large number of Duckiebots and, most importantly, a certain amount of reproducibility in the results. As Duckiebots frequently veered off the side of the road or collided with sign posts, the same experiment could not be repeated in a meaningful way and the proposed metrics did not make a lot of sense. Nonetheless, once this year's projects have all been merged and a greater overall stability in the system is achieved these metrics would provide a useful method of evaluation.

## 31.6. 3.6 Future avenues of development

As seen in the previous sections we were able to provide a framework for a fleet level planning system within Duckietown which can serve as a basis for interesting research questions. However, it goes without saying that the current state of the system has some of room for improvement. This sections lists possible extensions and improvements.

### 1) 3.6.1 Integration with other improvements from 2017

As all the teams were working on their projects in the same time frame with unclear finishing dates we made the decision in mid-January to use lane following and intersection maneuvering from the duckietown class of 2016. Therefore the performance of the system is not as good as it could be with the improved implementation of these two components. Integrating these two improved components into the system to replace the old ones would be a low effort, high gain development.

### 2) 3.6.2 Distributed fleet planning

The fleet planning system currently requires the taxi central node to run on a computer in the same network as all the Duckiebots. This is a single point of failure. If either communication between the Duckiebots and that computer breaks down or the computer itself fails the whole system fails. This is a very undesirable property. The fleet communication system not only supports communication with a central node but also point to point communication between the Duckiebots. Therefore it is possible to implement the system in a decentralized way. One possible implementation strategy:

- New customer requests are broadcasted in the network using a flooding algorithm
- Available Duckiebots close to the customer propose that they pick him up
- The duckiebots that broadcasted a proposal reach consensus using an algorithm such as Cheap Paxos [4]
- The decision is again flooded in the network such that every Duckiebot can accordingly update its own knowledge of the system.

Under the assumption of a connected network (i.e. no partitions) such a system is able to achieve the same performance as a system with a centralized node that coordinates all the Duckiebots. However, the implementation of such algorithms is more demanding because of the increase of complexity in the system which makes

it harder to debug.

### 3) 3.6.3 Switch to Mesh Network Communication

The system as is requires a router for passing the messages in the network around. This can be a standard wifi router or a mobile phone used as a hotspot. This is additional hardware that is required to run the system and needs to be setup the right way. I.e. further points of a possible error while running the system. The library from the fleet communication team which we already use for communication also supports a mesh network configuration. In this configuration each Duckiebot uses its wifi stick to create the same Wifi network. The duckiebots can then communicate using this network. Therefore, no additional hardware is needed. The change to enable this kind of network communication is rather small. However, it has not yet been tested. So further development could include the inclusion and testing of the mesh network configuration.

### 4) 3.6.4 Parking of unused Duckiebots

In a real world scenario of an autonomous taxi system, the demand for vehicles changes over the course of a day, i.e. there are peaks around the morning commute time as well as the evening. One kind of optimality is to only have as many vehicles on the road as needed based on the current demand. Obviously this should be combined with a prediction of the future demand to minimize waiting time for the customer. (The interested reader is referred to [5] and [6] for a more thorough analysis of fleet size and rebalancing strategies).

The current implementation of the system forces all Duckiebots to continuously drive around the city. If they are not fulfilling a customer request they are driving around according to the currently activated rebalancing strategy (random by default). This is not perfectly efficient. Using the outcome from the parking team the two systems can be combined to allow dynamic resizing of the currently active fleet by parking vehicles that are currently not needed.

### 5) 3.6.5 Implementation and evaluation of rebalancing strategies

Unused vehicles drive to locations according to a rebalancing strategy. The default rebalancing strategy is “random” and sends the vehicles to random locations. The software architecture allows to easily implement further strategies and use them within the system. As a further development more rebalancing strategies can be implemented and evaluated for their performance in Duckietowns of different sizes. The reader is referred to [6] for a list of possible rebalancing strategies that can be implemented.

### 6) 3.6.6 Location estimation and visualization between intersections

The current implementation updates the location of the Duckiebots only at intersections. Using wheel encoder information the locations could be estimated in between intersections and thus deliver a more fluid user interface and allow customer pick up in between intersections. A more high powered version might use SLAM to localize Duckiebots at any given point in time, allowing for more fine-grained localization and consequently better fleet planning.

### 7) 3.6.6 Online Map Generation

The current implementation of the MOD system depends on a map of the Duckietown generated a priori. It can be extended with the SLAM functionality implemented by the team in Montreal. Using the map that's generated while traversing the map would remove the step of manually creating the map and thus make the system more user friendly.

# Unit N-32

# 4 Conclusion

+ missing status

```
Status not found for this header:

  <h1 id="fleet-planning-report-conclusion">4 Conclusion</h1>

 in file /home/duckietown/scm/duckuments/docs/atoms_85_fall2017_projects/
21_fleet_planning/30-fleet-planning-project-report.md

Please set the status for all the top-level headers.

The syntax is:

      # My section    {#SECTIONID status=STATUS}

These are the possible choices for the status:

           status=draft   This is a draft. Drafts are typically hidden from front-facing
versions.
            status=beta   This is ready for review.
           status=ready   This is ready to be published.
       status=to-update   This is out-of-date and needs a refresh.
      status=deprecated   This part is deprecated and will be eventually deleted.
status=recently-updated   This part has been recently updated (<1 week).
```

In summary, the fleet planning project at current allows for the high-level control of a large number of Duckiebots, the visualization of the duckiebots on the map in a GUI, the assignment of customer requests an the execution of taxi services. The system works but relies heavily on smooth functioning of other components and is only as robust as these components are. The Duckiebot classes are extensively documented and designed in a way that allows easy extension with different fleet planning and rebalancing algorithms. This paves the way for future updates, some of which were discussed in the previous section.

## 32.1. References

[1] M. Pavone, K. Treleaven and E. Frazzoli, “Fundamental performance limits and efficient policies for Transportation-On-Demand systems” 49th IEEE Conference on Decision and Control (CDC), Atlanta, GA, 2010, pp. 5622-5629.

[2] W. Galitz , “The essential guide to user interface design: an introduction to GUI design principles and techniques” John Wiley and Sons, 2007.

[3] https://docs.python.org/2/library/pickle.html, Accessed: February 2017

[4] L. Lamport and M. Massa, “Cheap Paxos,” International Conference on Dependable Systems and Networks, 2004, pp. 307-314.

[5] K. Spieser, K.Treleaven, R. Zhang, E. Frazzoli, D. Morton and M. Pavone, “Toward a Systematic Approach to the Design and Evaluation of Automated Mobility-on-Demand Systems A Case Study in Singapore” Chapter in Road Vehicle Automation, Gereon Meyer, Sven Beiker (Editors). Berlin: Springer, 2014, pp.229-245

[6] Pavone, M., S. L. Smith, E. Frazzoli, and D. Rus, “Robotic load balancing for

mobility-on-demand systems." The International Journal of Robotics Research 31, no. 7

UNIT N-33

# Transferred Lane following

This is the description of transferred lane following demo.

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Wheels calibration completed.wheel calibration

**Requires:** Camera installed in the right position.

**Requires:** Joystick demo has been successfully launched.Joystick demo

**Requires:** PyTorch installed on duckiebot and laptop. (On duckiebot, you can either build from source(link), or download the pytorch-0.2 wheel file we built)

## 33.1. Video of expected results

link 1 of lane following link 2 of lane following

## 33.2. Duckietown setup notes

A duckietown with white and yellow lanes. No obstacles on the lane.

## 33.3. Duckiebot setup notes

Make sure the camera is heading ahead. Tighten the screws if necessary.

## 33.4. Pre-flight checklist

Check: turn on joystick.

Check: Enough battery of the duckiebot.

## 33.5. Demo instructions

Here, give step by step instructions to reproduce the demo.

Step 1: On duckiebot, in /DUCKIERTOWN_ROOT/ directory, run command:

```
duckiebot `roslaunch deep_lane_following lane_following.launch`
```

Wait a while so that everything has been launched. Press R1 to start autonomous lane following. Press L1 to switch to joystick control.

The following is the same as demo-lane-following: Empirically speaking, no duckiebot will successfully run the demo for its first time. Parameter tuning is a must. The only two parameters that you can modify is the gain and trim. The parameter

pair which makes your bot go straight will unlikely work for the lane following due to the current controller design. Facts show that a gain ranging from 0.5 to 0.9, as long as paired with a suitable trim, will all work on this demo. Start with your parameter pair obtained from wheel calibration. Increase gain for higher speed. Increase trim to horizontally move the bot to the center of the lane. Decrease will do the inverse.

Step 2: On laptop, make sure ros environment has been activated, run command:

```
laptop `rqt`
```

In rqt, the images can be visualized are /(vehicle_name)/camera_node/image/compressed

## 33.6. Train the network from scratch

Step 1: Download the simulator docker from link, and launch it by following the similar instructions in link

Step 2: Clone link and run

```
laptop `python collect_data.py`
```

This will generate ~10100 images under `images` folder.

**Note**: you can skip the first two step If you want to download the images we collected. Here is the link)

Step 3: Clone link, Assume duckietown_project_pose and gym-duckietown are under the same folder.

Step 4: Pretrain on simulation images:

```
laptop `cd duckietown_project_pose_estimation; python main.py --augment --use_model2`
```

You can add `--cuda` if you have gpu available.

Step 5: Download real images from link, and decompress under gym-duckietown folder.

Step 6: Finetune on real images, by run: laptop `python finetune.py --use_model2 --epochs 200`

Step 7: Checkout branch cbschaff-devel and copy model. duckiebot `git checkout cbschaff-devel` duckiebot `catkin_make` duckiebot `roscd deep_lane_following` duckiebot `cp model_file include/deep_lane_following/model.pth.tar`

Step 8: Run the ros package.

```
duckiebot `roslaunch deep_lane_following lane_following.launch`
```

## 33.7. Troubleshooting

Contact Chip Schaff or Ruotian Luo(TTIC) via Slack if any trouble occurs.

UNIT N-34

# Transfer Learning in Robotics

KNOWLEDGE AND ACTIVITY GRAPH

**Results:** Understanding transfer learning and the domain randomization technique for transfer learning.

This unit introduces the concept of Transfer Learning and how it can be applied to Robotics.

## 34.1. Transfer Learning Definition

Transfer learning is a subfield of machine learning that focuses on using knowledge gained while solving one problem to solve a related problem.

## 34.2. Why is transfer learning important in autonomous driving (or duckietown)

A known problem for autonomous driving (in real world or duckietown) is the lack of data. Today, the most successful methods use a form of machine learning called deep learning. Deep Learning is extremely powerful but is known to require a large amount of data to achieve good performance. However, it is time intensive and expensive to collect labeled data on a real system. Additionally, in reinforcement learning, the agent learns by trial and error, which can lead to large safety concerns for the vehicle and the people around it.

A solution to the data problem is to build a simulator, in which we can safely collect data and train deep learning systems. However there is a discrepancy between simulation and reality because the simulator does not perfectly model the world. So, we need transfer learning techniques to utilize models trained in simulation on real systems.

## 34.3. Transfer Learning in duckietown

In our case, the simulator has clean background and rooftop, but real duckietown has cluttered background. And also, the texture of the road are not exactly the same as duckitown, and there's no illumination changes in the simulator. However the lane width, camera setting are similar. Additionally, the dynamics in the simulation will not directly match the real Duckietown.

(a) Simulator image

(b) Real Image

Figure 34.1. Simulator images and real images

Need to have images with the correct filename in the folder

### 1) Domain randomization

Domain randomization is a common technique to enable transfer from simulation to the real world. The idea is to continually randomize the dynamics and look of the simulator. The intuition behind this idea is simple: the real world is going to look and act unlike the simulator, so we should force our policy to be robust to these factors. For example, let's imagine we wanted to train a policy to control the duckiebot directly from images. As long as you can determine the location of the lane lines, the exact look of the environment is unnecessary for the task. However, it is easy for the model trained in simulation to rely on the exact look of the simulation, making it useless in Duckietown. By forcing the policy to be robust to the lighting, coloring, and textures specific to the simulator, it will focus on details such as the position and shape of objects, which are equivalent to that in Duckietown. A policy which only

relies on this information will then work when deployed in Duckietown.

(a)

Figure 34.2. Simulators after domain randomization

### 2) Specific task to transfer

Here we have a simple example of training neural network for pose estimation using transfer learning. We replace the pose estimation module in current lane following package with the trained network.

The network takes the camera image as input, and outputs pose d and theta. (d and theta are the angle with respect to the front and )

### 3) Training pipelines:

Collect images by placing duckiebot in the simulator with different poses (The poses are recorded and later used to train the model). Train the neural network on simulator images. Each image is augmented differently by domain randomization technique during training. We use a CNN which contains 5 convolution layers and 2 fully connected layer. We use mean square error and use Adam to do optimization. Fine-tune on real images. (We collected by driving the bot around the duckiebot, and manually label the d and theta by bare eyes) Deploy on real bots.

### 4) Reference.

https://arxiv.org/pdf/1703.06907.pdf

# Unit N-35
# PDD - Supervised-learning

## 35.1. Part 1: Mission and scope

### 1) Mission statement

To learn policies which match the results from recorded data from agents in the real world, so that the vast volumes of the data in the real world can be made useful.

### 2) Motto

In Rete Tuo videbimus lumen

(Anything that is said in Latin sounds important)

### 3) Project scope

*What is in scope:*

- Verifying whether Deep Learning can be used successfully in duckietown.
- Motivated by the concept of 'data processing inequality', using supervised and imitation learning to control the duckiebot end-to-end with data from a recorded policy.
- Using supervised or unsupervised learning to model specific aspects of the autonomous driving task.
- Focusing on indefinite lane navigation by learning based tools.

*What is out of scope:*

- Doing on-policy RL (i.e. running the robot with our learned policy to collect data).
- Collecting our own datasets by running with either our own policy (by hand).

*Stakeholders:*

**All current teams** * Those who wish to use deep learning in the real world could benefit from our pipeline and experiments in using DL in the duckietown.

**For future teams** * If a future team does on-policy RL in the duckietown, initializing with our imitation learned policy could be smart.

## 35.2. Part 2: Definition of the problem

### 1) Problem statement

We have recorded data of the lane following algorithm running smoothly in the duckietown. Our goal is to learn a policy which performs as well as, or better than

the policy which produced the data.

### 2) Assumptions

- The policy used to collect the data is reasonably good.
- The training data can be updated when other groups formulate policy which have better performance.
- The errors in imitation learning are sufficiently small to allow a straightforward approach to learn a decent policy.
- Our trained policy can improve the robustness of overall performance.

### 3) Approach

- Initial approach is to take image and control parameter data from the lane navigation checkoff/log data from all schools.
- Start with indefinite navigation and lane following data.
- First DL approach is to take the last k-frames (probably could use a smarter selection strategy which picks some older frames) before the control parameters are recorded, and train a neural network to predict the control parameters from the state.
- We will manually downsample the image frames to find the smallest resolution where the lane is clearly visible by inspection.
- We will use an absolute error to predict the control parameters, and measure relative error on held-out data, to figure out if we can learn a network which generalizes.
- We will train ALI(adversarially learned inference) and VAE(Variational autoencoder) on the images, for the purposes of intellectual curiosity as well as semi-supervised learning (if overfitting is a serious issue).

### 4) Functionality provided

1. Offline metrics
2. Loss for one step ahead prediction on recorded data, with point predictions for the two control parameters.
3. Likelihood under a continuous distribution over the predicted control parameters.
4. Likelihood under a fixed discrete distribution over the predicted control parameters.
5. Online metrics:
6. Actually run the duckiebot in real world duckietown with our learned policy.
7. Visual inspection of trajectories.

### 5) Resources required / dependencies / costs

- Requires neural compute stick on the duckiebot to run. (already got it)

- GPUs for training models (available through MILA and IDSC in ETH).
- Data for training the imitation learning algorithm (ideally use as little as possible).

#### 6) Performance measurement

- We can use out-of-sample evaluation for the offline metrics, with care taken so that the train, validation, and test sets cover non-overlapping groups of students.
- Online evaluation will be qualitative, and will be done in the Udem duckietown.

## 35.3. Part 3: Preliminary design

#### 1) Modules

- Data collection: a raw collection of the images (state) and control signals.
- Data alignment: Create sets of actions approximately aligned with states (as they're recorded at different frequencies).
- Data Example Construction: Create tuples of (state[t], action[t], state[t-1], state[t-2]).
- Data Train/Valid/Test Split: split the dataset randomly but with different students going into different datasets.
- Model trainer: From collected data trains a model to predict actions from states.
- Model actuator on duckietown: Runs the duckiebot using actions from the trained model.

#### 2) Interfaces

- Data collection: Takes student actions and returns a list of ROS bag files saved to some mila filesystem.
- Data alignment: Performed in-memory, takes the ros-bags as inputs and returns a list of aligned (state[t], action[t]) pairs.
- Data Example Construction: Also performed in-memory, and produces (action[t],state[t],state[t-1],state[t-k]) k-tuples.
- Data Train/Valid/Test Split: Uses a fixed random seed to split the examples into different groups, which are then saved to different files on mila filesystem.
- Model trainer: Takes data as input and returns a saved model binary for running on the intel compute stick (todo: figure out just how small binaries need to be).
- Model actuator on duckietown: Takes a trained model as input and is a ROS module which listens for camera data, and sends control signals to the motors.

#### 3) Specifications

- No changes to duckietown specification.
- Duckuments for using neural compute stick.

#### 4) Software modules

- Python script or small collection of python scripts for data processing.
- Python script using Tensorflow for training model.
- ROS node for running a fixed model on the duckiebot.

#### 5) Infrastructure modules

- We don't think any of the modules are infrastructure.

## 35.4. Part 4: Project planning

#### 1) Data collection

- Collect data generated by other navigation policies.

#### 2) Data annotation

- Generally speaking, no data annotation required.
- Might need to annotate video to remove crashes, stalls.

*Relevant Duckietown resources to investigate:*

- Taiwan group has done imitation learning with 3-camera setup - we may be able to reuse some of their code or at least learn from their experience.

*Other relevant resources to investigate:*

- See the following papers.

title title title

#### 3) Risk analysis

- Raw imitation learning may perform badly in practice due to "exposure bias / exploration" issue.
- Simplest solution might be some sort of data augmentation which moves the

lane in the training data to create “correction” examples.

- May be smarter ways to improve generalization.
- We may have a hard time figuring out what metrics to trust for offline evaluation of the learned model.
- Workaround might be to report many different metrics and always make sure that the simplest metrics (like per output relative error) are reasonably small.
- Some metrics are hard to interpret: making it difficult to know when to declare success.
- May require some intermediate online evaluation to figure this out.

UNIT N-36

# Supervised Learning: intermediate report

## 36.1. Part 1: System interfaces

The system architecture is shown below.

### 1) Logical architecture

- Please describe in detail what the desired functionality will be. What will happen when we click "start"?
- The desired functionality is the deep imitation learning network. How the function works is the following: the node of trained deep imitation learning network subscribes to the image published by compressed image node, does computation, then publishes control commands(orientation and velocity) to the inverse kinematic node.
- Please describe for each quantity, what are reasonable target values. (The system architect will verify that these need to be coherent with others.)
- The robot does lane following with a learned end-to-end deep imitation learning system. Look at the activations of the layers and try to understand them. The target can also be extended to Indefinite navigation realized by deep imitation learning if the lane following is fulfilled.
- Please describe any assumption you might have about the other modules, that must be verified for you to provide the functionality above.

- The time latency of other modules are within reasonable range.

### 2) Software architecture

- Please describe the list of nodes that you are developing or modifying.
- We will develop one node, the trained deep imitation learning model, that maps the compressed images to control commands(orientation and velocity). All other nodes will remain unchanged.
- For each node, list the published and subscribed topics.
- The deep imitation learning node subscribes to /VEHICLE_NAME/camera_node/image/compressed.
- The node publishes /VEHICLE_NAME/car_cmd_switch_node/cmd
- For each subscribed topic, describe the assumption about the latency introduced by the previous modules.
- The latency of image topic can be measured. But during the model training process, we would like to use the map between multiple images and one control command to cover the latency.
- For each published topic, describe the maximum latency that you will introduce.
- The latency that our node introduces will be settled by the computation capability of the Intel Neural Compute Stick and the scale our model. Tests are required before the latency can be finalized.

## 36.2. Part 2: Demo and evaluation plan

### 1) Demo plan

The demo is a short activity that is used to show the desired functionality, and in particular the difference between how it worked before (or not worked) and how it works now after you have done your development.

- How do you envision the demo?
- In the final demo, we hope to implement end-to-end deep imitation learning on Duckiebot and make it work for the aim of lane following.
- What hardware components do you need?

- The Intel Movidius Neural Compute Stick.

### 2) Plan for formal performance evaluation

- How do you envision the performance evaluation? Is it experiments? Log analysis?
- The performance can be first evaluated manually: 1)Compare the time of lane following by traditional approach and the end-to-end deep imitation learning approach. 2)Observe the deviation of lane following by deep imitation learning approach.
- Other formal evaluation approach will be updated.

## 36.3. Part 3: Data collection, annotation, and analysis

### 1) Collection

- How much data do you need?
- A few thousands of image taken by the camera and the corresponding label pairs from images to control commands (orientation, velocity).
- How are the logs to be taken? (Manually, autonomously, etc.)
- The logs will be taken manually. The make log-minimal in branch 'dev-eth-sup-learning' will be enough for the data collection.
- Describe any other special arrangements.
- None.
- Do you need extra help in collecting the data from the other teams?
- None.

### 2) Annotation

- Do you need to annotate the data?

- None, The data itself makes enough sense.
- At this point, you should have you tried using thehive.ai to do it. Did you?
- We have collected the data and extracted the things we need already.

### 3) Analysis

- Do you need to write some software to analyze the annotations?
- None. But we did write a python script to write images and data pair from ros bag.
- Are you planning for it?
- None.

UNIT N-37

# PDD Neural Slam

## 37.1. Part 1: Mission and scope

### 1) Mission statement

Build a map from a duckie driving around the road autonomously that will be used for planning

### 2) Motto

> À l'idiot sans mémoire tout lui paraît nouveau et miraculeux

### 3) Project scope

**What is in scope**

Train agent to learn to explore an entire map efficiently and leep a representation of the current knowledge of the map.

**What is out of scope**

Need to run live on the duckie. We do not use real images but assume we have a tile detector

**Stakeholders**

Transfer learning team - tbd

## 37.2. Part 2: Definition of the problem

### 1) Problem statement

We want the to keep a representation of the current map in memory that could used for downstream task such as planning

### 2) Assumptions

- We work on the tile level, and do not worry about low-level control.
- We assume there are only two types of tiles, road or not road

### 3) Approach

Stage 0: Create an environment where we can a simulated agent

Stage 1: Train an agent using a reinforcement learning method paired with an external memory

Stage 2: Deploy on the real robot and see how the method perform

Stage 3: Use a decoder that can recover the knowledge of the map

### 4) Functionality provided

- A exploration policy

- A map when the exploration is done

### 5) Resources required / dependencies / costs

- GPUs to train our policy
- Chip to run the neural network policy
- A "tile predictor" to infer the tile type from an image
- A simulator to train the agent

### 6) Performance measurement

- Check whether or not the agent actually explores the whole map
- Compare the decoded map with the ground truth map

### 7) Functionality-resources trade-offs

- Robust obstacle detection (many filters,...) vs. computational efficiency
- Maximizing speed (e.g. controlles might want to do that) vs. motion blur

## 37.3. Part 3: Preliminary design

### 1) Modules

- **A grid map environment to train the agent.**
- **A policy network with an external memory**
- **A decoder to decode the external memory into a map**
- **A tile detector ??**

### 2) Interfaces

**Simulator**

- takes the map size as input and generate a environment with agent output

**Policy network**

- take the current position of the robot w.r.t its original position, reads/write to the memory and give a control action.

**Decoder**

- takes the internal memory as input and give the actual map as output.

### 3) Specifications

No need to revise duckietown specifications

### 4) Software modules

- Pytorch / Tensorflow
- Grid world simulator

## 37.4. Part 4: Project planning

### 1) Timeline

- Build the environment to train the agent
- Implement the agent to be trained with an external memory for the exploration task
- Train the agent
- Check if the map an actually be decoded

## 2) Data collection

- No need for data collection

## 3) Data annotation

- No need data annotation

*Relevant Duckietown resources to investigate:*

Duckietown simulator

*Other relevant resources to investigate:*

- *Neural SLAM*: Jingwei Zhang, Lei Tai, Joschka Boedecker, Wolfram Burgard, Ming Liu
- *Learning to Navigate in Complex Environments*: Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andrew J. Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, Dharshan Kumaran, Raia Hadsell
- *Hindsight experience replay*: Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, Wojciech Zaremba
- *Neural Turing Machines*: Alex Graves, Greg Wayne, Ivo Danihelka Differentiable neural computer Alex Graves et al (DeepMind)

## 4) Risk analysis

- Might have issues to scale
- RL agent might have stability issues paired with an external memory the whole system could be hard to train

# Unit N-38
# PDD - Visual Odometry

## 38.1. Part 1: Mission and scope

### 1) Mission statement

We will use unsupervised learning to build a depth estimation system for Duckietown. The application could be building a point cloud map for Duckietown for mapping–our overall plan is to have a serviceable deep network that is trained end-to-end with no ground truth data. We will also be using the Movidius chip, hopefully learning to how integrate it well into the current system for future users.

### 2) Motto

CARPE DIEM

(Seize the day)

### 3) Project scope

The scope of the project is to use recent work in unsupervised depth estimation as well as training data we gather in Duckietown to create a fully unsupervised depth estimation system.

*What is in scope:*

The deep network and use of Movidius chip.

*What is out of scope:*

Localization. We plan on utilizing the april tags for localization. Also hardware modification–we plan to do fully monocular depth.

*Stakeholders:*

All of the SLAM teams could (potentially) benefit from our system, though it might be non-trivial to integrate it.

## 38.2. Part 2: Definition of the problem

### 1) Problem statement

We need to train a deep neural network to predict depth from monocular video (with no GT depth!)

### 2) Assumptions

We assume that it's possible to collect diverse enough training data in Duckietown

to train a deep CNN (outside data like KITTI might not transfer to this task).

#### 3) Approach

We will start with the "Unsupervised Learning of Depth and Ego-Motion from Video" paper and modify the framework for duckietown.

#### 4) Functionality-resources trade-offs

There's often a tradeoff beteen size of network and performance–the Movidius chip is more limited than our usual NVidia GPUs.

#### 5) Functionality provided

We will need to develop some confidence measures for our depth map. Time permitting, we would like to build a point cloud map of Duckietown.

#### 6) Resources required / dependencies / costs

We will need the Movidius chip for best performance–our metrics are the size of the network (including activations) and making sure that inference is real-time.

#### 7) Performance measurement

Measuring performance here is tricky since there is no way to obtain ground truth depth data from Duckietown (aside from using a range sensor not available for duckiebots). We will likely develop a surrogate metric based on april tags.

## 38.3. Part 3: Preliminary design

#### 1) Modules

Since it's an end-to-end neural network it's all in one logical module, but we could split up the design of the architecture from the way we use the data (e.g. data augmentation)

#### 2) Interfaces

Input: RGB Image Output: Depth map (potentially relative depth, discretized)

#### 3) Preliminary plan of deliverables

The architecture and the data collecation and augmentation schemes need to be designed. Tensorflow implementation of architecture is what needs to be implemented. There already exists open source code for unsupervised learning of depth paper (linked below).

#### 4) Specifications

Do you need to revise the Duckietown specification? N/A

#### 5) Software modules

The software will be a simple end-to-end CNN going from frames to a depth map

(likely a node publishing a depth map)

## 38.4. Part 4: Project planning

Next phase is to start collecting data and experimenting with architectures in Tensorflow.

### 1) Data collection

Video of duckiebot traversing duckietown.

### 2) Data annotation

No data annotation necessary.

*Relevant Duckietown resources to investigate:*

All of the camera geometry and computer vision notes.

*Other relevant resources to investigate:*

1. https://people.eecs.berkeley.edu/~tinghuiz/projects/SfMLearner/
2. https://github.com/tinghuiz/SfMLearner

### 3) Risk analysis

It's possible that training a deep neural network on only duckietown data will be difficult. We will also consider using the KITTI dataset as additional training data.

UNIT N-39

# Visual Odometry Project

Here we briefly describe the theory behind the model in the visual odometry project. The discussion begins with a review of epipolar geometry and a description of the depth image-based rendering problem, then moves to the description of the deep learning model used.

## 39.1. Epipolar geometry and DIBR

Figure 39.1. Epipolar geometry.

We follow the discussion in Sun et al. (2010). First, consider the stereo setup and recall the relationship between a world point

$$m1 = \frac{1}{Z} K1 \cdot R1[I| - C1]M$$
$$m2 = \frac{1}{Z} K2 \cdot R2[I| - C2]M$$

If we choose the left camera as the reference, we can set $R1 = I, C2 = 0$ in order to get:

$$m1 = \frac{1}{Z} K1 \cdot [I|0]M$$
$$m2 = \frac{1}{Z} K2 \cdot R[I| - C]M$$

and then we can get the relationship with depth $Z$:

$$Zm2 = ZK2RK1^{-1}m1 + K2C$$

Using this relationship, we can learn a prediction for $Z$ and use image $m1$ to predict image $m2$–we describe the experiments below.

## 39.2. Learning Depth Prediction

We consider an end-to-end CNN-based unsupervised learning system for depth estimation, using the paper Unsupervised Learning of Depth and Ego-Motion from Video by Zhou et al., CVPR 2017. This model was initially trained on the KITTI dataset, and we take the pre-trained weights and evaluate them in Duckietown, showing that the results can be significantly improved by fine-tuning with Duckietown images. Our goal is real-time inference, and at test time we only use the depth prediction network:

(a) Training: unlabeled video clips.

(b) Testing: single-view depth and multi-view pose estimation.

Figure 39.2. The general architecture.

The model consists of two networks–a pose estimation network giving us $\boldsymbol{R}, \boldsymbol{t}$ between the source and target frames, and a depth prediction network that gives us $\boldsymbol{Z}$ and allows us to warp the source view to the target view using the pose and the RGB values in the source image. Over time, the depth prediction network begins to predict reasonable depth values. The result of fine-tuning on Duckietown data plus KITTI pretraining versus applying trained model on the KITTI dataset directly:

Figure 39.3. Fine tuning on our dataset.

When the bot turns, motion blur heavily affects the model predictions. One way to alleviate this problem would be to preprocess input images. First deblur them and then feed them to the depth prediction network. Having blurred images in the training set would also slightly improve the results:

Figure 39.4. Results on motion-blurred images.

## 39.3. True Depth

We benchmark our results against the true distance from the camera we get from April tag detection. In the figure below, we show the predicted depth values versus the estimated depth:

Figure 39.5. Comparison to true depth.

Outliers at low depths are due to lack of texture around April tags. As we can see with proper scaling our estimated depth is well aligned with the estimated depth from April tags–using a sparse set of true depth maps, we can rescale our pixelwise prediction into metric units.

## 39.4. Conclusions

We demonstrated a monocular depth estimation pipeline, trained with no annotated data. The approach gives reasonable depth predictions in Duckietown, but we found several notable limitations. The results on motion-blurred frames are poor, even after fine-tuning with a small number of blurred images–for good results in

fast-moving environments, we would likely need to train with more blur. In addition, our approach does not incorporate any traditional depth post-processing, which should significantly improve results.

Our depth prediction node could be used for a variety of tasks, including point-cloud-based SLAM as well as obstacle detection.

## 39.5. References

Unsupervised Learning of Depth and Ego-motion from Video

Depth Image-Based Rendering

**Author:** Igor

**Maintainer:** Igor

**Point of contact:** Igor

UNIT N-40

# Deep Visual Odometry ROS Package

## 40.1. devel-visual-odometry

This package contains a ROS node `dt_visual_odometry` that produces monocular depth estimates on duckiebot. It also contains another node `apriltags_ros_center`, which is slightly modified from `apriltags_ros` to publish pixel locations, in order to benchmark the result on April tags. You need to have Tensorflow installed on your local machine.

To launch the node, clone the repository into `catkin` workspace and download the Tensorflow model checkpoint file from Duckietown Dropbox to `checkpoint_dir`. This Tensorflow model is obtained through taking a pretrained model on KITTI and further train it on Duckietown for 200K iterations with smoothness penalty set as 0.1. Use the command:

```
roslaunch dt_visual_odometry deepvo.launch ckpt_file:=checkpoint_dir robot_name:=robot_name
```

This publishes the depth heatmap the into `robot_name/VO/image/compressed` as well as prints the predicted(from CNN) and actual(generated from April tags node, unit in meters) depth on any detected April tags. The scaling factor is calculated as the average predicted/actual depth for all detected April tags, and published under the topic `robot_name/VO/scale`. To supress April tags detections for a higher refresh rate of depth heatmap, use `apriltags_scaling:=0`.

# Unit N-41
# PDD - Anti-Instagram

## 41.1. Part 1: Mission and scope

### 1) Mission statement

Make the Duckiebot robust to illumination variation.

Line detection includes different colors of lines, we need to be able to detect these colors accurately.

### 2) Motto

SEMPER VIGILANS

(Always vigilant)

### 3) Project scope

*What is in scope:*

- Improve software (anti instagram, line detection, ...) such that the line detection is robust to illumination variation
- Sample ground truth pictures
- Influence future changes on Duckietown (e.g. color of parking spots)

*What is out of scope:*

- Geometric interpretation of the line detection (e.g. where is the middle of the road, distance to certain objects, ...)
- Hardware modifications of the Duckiebot
- Hardware modifications of the current Duckietown set up (colors of lanes, stop line, ...)

*Stakeholders:*

| | |
|---|---|
| System architect | She helps us to interact with other groups. We talk with her if we change our project. |
| The controllers | A.A. is the interaction person to confirm that we fulfil their requests regarding: frequency, latency, accuracy (resolution), maximum false positives, maximum misclassification error (confusion matrix). |
| The Parking | We will determine together the best color for the parking lot lines if needed. |
| The Navigators | At the moment the Duckiebot is controlled open loop at intersections. This should be improved. Probably they need line detection in a certain way. We can help and figure together out what procedure would be the best |

## 41.2. Part 2: Definition of the problem

### 1) Problem statement

One of the central problems in autonomous mode of the duckiebot is the line detection. Line detection though is very sensitive to illumination variation. This problem has been solved by a color transformation called "Anti-Instagram". The current illumination correction algorithm, however, is not working well enough. This affects the extraction of the line segments since the extract-line-segments algorithm is very sensitive to illumination changes (e.g. shadow, bright light, specular reflections). There are several reasons why the current implementation fails:

1. Illumination correction is done only once by user input. So we don't do any online/automatic correction. This will obviously fail in an environment where illumination is changing frequently.
2. The algorithm works by detecting different clusters in RGB space for the colors of lines, thus it fails when it's not able to differentiate adequately. (e.g. in certain lighting conditions yellow and white look quite similar, in addition specular reflections distort all of the colors)
3. The color space is fixed to RGB, but it's unclear that this is best.
4. No geometric information is considered in differentiating colors of lines. The color information is completely decoupled from the place it's actually coming from, thus a red Duckiebot may be detected like a stop line, even though their shapes are quite different. (We also don't know whether a pixel is coming from the "street level" or the "sky level")
5. It is a linear transformation (shift, scale) instead of a possible non-linear transformation, but there is nothing indicating this should be true.
6. Any previous anti-instagram transformation parameters are not taken into account when a new transformation is performed, thus no prior knowledge is leveraged.

### 2) Assumptions

1. Lighting:
   - We assume normal office lighting, including any shadows that may occur because of occlusions, or spatial variance.

- We don't consider outdoor illumination (e.g. sunlight).
- We assume having illumination (no pitch black scenario).

2. Duckietown Condition:
   - We assume the normal colors of lane lines, plus one or two more (for the parking lot).
   - We assume no variation in the shapes of the lines besides what is already constructed.
3. Training data:
   - We expect to have some pictures in different scenarios with correctly labeled segmentation (lines, street, ...), where a polygon is drawn around each region.
   - The pictures will be captured from a Duckiebot camera.
4. We also assume the current method of extracting lane pose from segments is accurate.

### 3) Approach

1. Understand current system
   - Determine false positives, false negatives, true positives, true negatives of current line detection
   - Determine latency
   - Compare other color spaces than RGB to see how it affects performance (e.g. HSV or LAB).
2. Use geometric information to better determine the actual existing colors.
   - The optimal case would be that the system already knows beforehand which areas it should take into account. The relevant color areas are only the dashed lines area, the continuous line area, the stop line area, the parking line area and the street area. Everything else should not taken into account since we have no reference color for the other areas.
   - It should be possible to define a region in the picture where we can find these specific areas. For example the dashed line starts lower left and goes to direction top middle but it should stop at the "line of horizon" (= middle of the vertical length)
   - Distinguish between dashed and non-dashed lane lines to simplify identification of colors.
3. Use time information/parameters from earlier illumination corrections to improve robustness (Online learning).
   - In contrast to starting the color analysis from scratch every time, we could consider using the latest transformation parameters as an initial guess. We could consider to update the color analysis every x-th frame during a session, to be more robust to changing light conditions/shadows.
4. Further improvements:
   - We are using the color transformation to better estimate the lines. So we know after processing (color transformation, edge detection, ...) where we can find the lines. With that information we could update the color transformation. The color transformation now should take into account only the "important" areas. As a result we should have a more accurate color transformation. We repeat until we converge to a minimum.

### 4) Functionality provided

We are assuming to have ground truth pictures. Then it is possible by processing

the same picture with our algorithm and compare it to the ground truth to calculate an error.

We are going to consider true positives, true negatives, false positives and false negatives. This can be done either for only one color/one feature (dashed lines, continuous lines...) or for the whole process at once which means everything should be classified properly. In addition, we would like to measure the accuracy of the lane pose estimation for our algorithm vs. a ground truth, this can be a Euclidean distance.

#### 5) Resources required / dependencies / costs

Costs:

1. Computational cost
   - If the processing is done online we have to take care that it doesn't take too long.
2. Cost of producing ground truth pictures
   - Will be determined when we have some examples done by hand. (Week of 20th of November)

Resources:

1. Functional Duckiebot
   - Getting sample data for ground truth pictures
   - Try out the algorithm in real conditions

Dependencies:

1. Ground truth images
2. Lane pose estimator

#### 6) Performance measurement

1. Identify current run times of algorithms implemented at the moment and compare the algorithms intended to implement with the current ones. If the new ones are way more costly than the current ones and the current ones already use the Raspberry Pi to capacity it is probably not a good idea to implement ours. To sum up: Estimate run time of new implementation and compare to old. See whether implementation is feasible.
2. We can compute percentage of success of identification of a line segment, as well as correct color classification.
3. Measure euclidean distance of lane pose estimation using our algorithm and lane pose estimation without to the ground truth. The performance measurement procedure for the algorithm is described in the section *Functionality provided*.

## 41.3. Part 3: Preliminary design

#### 1) Modules

1. Anti-Instagram module: Takes raw picture from camera and estimates a color transformation. The transformation details are returned.
2. Module to classify geometries (e.g., distinguish between dashed lines, continuous lines and stop lines): This could be a standalone routine, which gets called by the Anti-Instagram, we could also combine this with the anti-instagram as de-

scribed in Approach point 4.
3. Online learning: Takes picture from camera, does Anti-Instagram procedure and transformation. The error (e.g. cluster error of k-means) is estimated and the procedure is repeated until the optimal transformation parameters are found (transformation with the lowest error). The procedure returns the optimal parameters. They are saved and used for the future image processing.

#### 2) Interfaces

1. Anti-Instragram: input: Raw camera image. Output: Transformation parameters
2. Geometry Classifier: Input: Raw camera image. Output: List of the different line segments.
3. Online learning module: input: Multiple camera images/continuous stream. Output: Optimal transformation parameters

#### 3) Preliminary plan of deliverables

1. Take the current algorithm and find best color space for it, estimate the errors and accuracies discussed previously.
2. Search for other clustering method and optimize current version. (Without considering geometry)
3. Consider geometry (as a first step indicate considerable areas by hand) and see what difference it makes compared to the current optimal implementation (Maybe after 1.) and 2.) are done).
4. Distinguish relevant and non-relevant areas (street surface vs. rest of world).
5. Distinguish dashed and continuous lines.
6. Implement and test a online learning system.

#### 4) Specifications

As stated above we are only involved in determining the best fitting color of the (not yet installed) parking lot lines. All the other colors and the environment are assumed to be given.

#### 5) Software modules

1. Anti-Instagram Node: Already exists, will most likely be updated or even changed completely according to our approach. The online-learning (if accomplished) will be folded into this node.
2. Geometry Classifier Node: Needs to be written according to the approach.

#### 6) Infrastructure modules

NO

## 41.4. Part 4: Project planning

| Date [MM/DD/YYYY] | Task | Who | Target Deliverables |
|---|---|---|---|
| 11/20/2017 | Finish the Preliminary Document, Peer Reading of other team members | Milan, Christoph | Preliminary Document |
| 11/27/2017 | Investigate why current algorithm fails. | Milan, Christoph | Create detailed description when the algorithm fails and when it works. Make it understandable why for everyone |
| 11/27/2017 | Create data annotation, check how website works | David, Shengjie | Get 1000 annotated pictures or have a specific date when these images are delivered. |
| 12/1/2017 | Find out what colorspace is the best for the current algorithm | | best color space, performance analysis |
| 12/4/2017 | Find out what's the best clustering method based on best color space, is the best color space still the best? | | Best clustering method, best color space, performance analysis |
| 12/4/2017 | Include geometry in the current color transformation algorithm | | Performance analysis |
| 1/8/2018 | Implement an online system | | Performance analysis of supervised system |

### 1) Data collection

Around 1000 pictures with the Duckiebot camera from a Duckiebot perspective in Duckietown. The pictures have to be from different environment conditions (illumination, specular light)

### 2) Data annotation

The data collected above has to be annotated. The annotations should state what type and what color it is.

1. Dashed lines: Yellow
2. Continuous lines: White
3. Stopping lines: Red
4. Street: Black
5. (Parking Lot: TBD)

*Relevant Duckietown resources to investigate:*

The whole sum of nodes within the '10-lane-control' folder will be within the scope of this project. This is:

- `anti_instagram`
- `ground_projection`
- `lane_control`
- `lane_filter`
- `line_detector`

- `complete_image_pipeline`

*Other relevant resources to investigate:*
1. Color differentiation, like this.
2. Properties of different color spaces
3. OpenCV

### 3) Risk analysis

1. Computationally too expensive algorithms
   - We have to estimate our algorithms carefully and compare them to the existing solutions.
2. No annotated data delivered
   - Build annotated data by ourselves/by hand.
3. Not enough time
   - Create good tasks list to be done. Try to specify time exact for every task.

UNIT N-42

# Transfer: preliminary design document

## 42.1. Part 1: Mission and scope

### 1) Mission statement

The goal is to test transfer learning algorithms trained on simulator and test on the real duckietown.

Solving control problems in reality is hard due to sparse reward signals, expensive data acquisition and the danger of breaking the robot during exploration. It is comparatively easier to train the policy in a simulator as we can "speed up" the reality, and there are no inherent dangers of running arbitrary policies. But policies trained on simulator do not necessarily transfer directly onto the real world, and our goal is try and bridge this gap

### 2) Motto

IPSA SCIENTIA POTESTAS EST,

UT TRANSIRE CALLIDUS EST

(Knowledge is power, transferring that is clever)

### 3) Project scope

Final goal: Implement DARLA or https://arxiv.org/pdf/1703.06907.pdf or https://arxiv.org/pdf/1710.06537.pdf

*What is in scope:*

Testing out different RL algorithms, (or unsupervised feature learning algorithms, if any)

Methods for transferring from simulator to duckietown

Baselining (?)

*What is out of scope:*

Hardware modifications

Design modifications that might be needed

Anything not involving the control of the duckiebot

*Stakeholders:*

Simulator (Maxime and Florian)

Supervised Learning (Rithesh and Alex)

Running NN models on the bot (Neural stick ?) who is in charge?

File with all projects click

## 42.2. Part 2: Definition of the problem

### 1) Problem statement

We will replace the part of the pipeline that uses raw images and give motor controls by using a model trained in a simulator. The model policy will be able to do lane following and navigation. The model can hopefully generalize to imperfect camera calibration, motor calibration and different light conditions.

Mission: Efficient transfer of algos trained on simulators

Problem statement:

- Implement DARLA or
- Domain Randomization Paper or
- Dynamics Randomization Paper
- Apply the algorithms for tasks like, navigation.

### 2) Assumptions

We need the simulation to be close enough to real world so as to be transferable.

We need to run inference on the robot at test time.

Maybe allow for fine-tuning in the real world (on a small dataset)

### 3) Approach

- Get the simulator up and running
- Start with the most basic lane following environment: A straight lane...
- Train a model to control the duckiebot in the simulator
- Transfer the policy to the real world
- Move onto slightly more complicated scenario ... repeat (curriculum learning setting)

### 4) Functionality provided

Lane following or route following (i.e. follow a given route through duckietown and obey traffic laws)

Metrics:

- Quality of navigation (as defined by a reward/loss function)
- Quality of transfer defined by the above reward function, computed automatically or by hand
- Finetuning needed for transfer?

Reward function description (probable):

- Deviation from center (-ve)
- Time taken (-ve)
- Finish position (+ve)
- Collision (-ve)
- Violating traffic rules or conventions (-ve)
- See Socially Aware Motion planning

### 5) Resources required / dependencies / costs

- # of cpu/gpu-hours for training

- Test time computation costs
- Duckiebots/ duckietown

#### 6) Performance measurement

- In simulation, use access to the true state to compute the reward function
- In duckietown, compute the reward function by hand (or develop a heuristic for computing it)
- Qualitative comparison to current control pipeline
- Baseline wrt other RL algos
- Compare with #devel-super-learning for performance wrt imitation learning policies

## 42.3. Part 3: Preliminary design

#### 1) Modules

Module 1: Policy mapping raw images ⟶ actions (or Module 1a: policy ⟶ disentangled representation ⟶ actions)

Module 2: Actions ⟶ motor voltages (Joy Mapper node)

Module 3: Training module (Also introduces domain/dynamics randomization in the simulator)

#### 2) Interfaces

Module 1: subscribe to raw camera images and publish actions (forward/backward/turn left/turn right = float values)

Module 2: already provided in duckietown stack

Module 3:

- Input: simulator parameters, model
- Output: a trained policy to be used in Module 1

#### 3) Specifications

See Modules and Interfaces above.

#### 4) Software modules

During training, the modules will be written in Python.

At test time, modules 1,2 will be deployed as a ROS node during test.

#### 5) Infrastructure modules

Simulator

## 42.4. Part 4: Project planning

#### 1) Data collection

- Real world cam pictures corresponding to simulator states (e.g in front of a

straight lane/at the beginning of a left/right turn/...) : can be useful if we want to assess the quality of the disentangled representation without the need to run it on the robot.

- Use the simulator to generate training data for RL algorithms

### 2) Data annotation

The simulator will annotate data automatically by providing ground truth information about the duckiebot and the environment

+ comment

To be confirmed... We probably need to implement a module that does that.

If needed, then semantically segmented images would useful

*Relevant Duckietown resources to investigate:*

Simulator

*Other relevant resources to investigate:*

PyTorch rl codebase, OpenAI Gym, OpenAI Baselines

DARLA: https://arxiv.org/abs/1707.08475

Transfer by randomization/generalization:

- https://arxiv.org/abs/1703.06907
- https://arxiv.org/abs/1710.06537

UNREAL: https://arxiv.org/abs/1611.05397

Socially Aware Motion planning: https://arxiv.org/abs/1703.08862

### 3) Risk analysis

Deploying the policy trained in simulator will break the bots in real duckietown.

Risk mitigation:

- Train it properly
- Curriculum learning setting to gradually increase difficulty
- Safety on/off switch
- Cushion the sides of the lane

# Part O
# Packages - Infrastructure

TODO: to write

# Unit O-1
# Package duckieteam

## 1.1. Package information

Link to package on Github

### Essentials

**Author:** Andrea Censi (maintainer)

### Description

Code for handling the DB of people and robots.

## 1.2. create-machines

This program creates the machines file, using the data contained in the scuderia database.

Run as follows:

```
$ rosrun duckieteam create-machines
```

## 1.3. create-roster

TODO: this program is unfinished

UNIT O-2

# Package duckietown

## 2.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Mack

### Description

The duckietown meta package

UNIT O-3

# Package duckietown_msgs

## 3.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Mack

### Description

TODO: Add a description of package `duckietown_msgs` in `package.xml`.

UNIT O-4

# Package easy_algo

## 4.1. Package information

Link to package on Github

### Essentials

**Author:** Andrea Censi (maintainer)

### Description

A package to make it easy to abstract about algorithm configuration and create regression tests.

## 4.2. Motivation

The problem that this package solves is the need to easily refer to the configuration of algorithms, and objects in general.

Here's one case: a node requires a line detector; a line detector has a dozen parameters. Do we want to put a dozen parameters in the node?

Or, think about the case where there are different line detector classes. Now, there must be a parameter in the node that tells which class to instantiate.

All of this business can be simplified by the features of EasyAlgo.

## 4.3. Usage

The approach is to declare that there is a "family" of things, and to provide the description of what those things are in YAML files. These files can be anywhere in `catkin_ws`.

To declare the family of line detectors we create a file `line_detector.easy_algo_family.yaml`, with the following content:

```
interface: line_detector.LineDetectorInterface
description: These are the line detectors.
```

This tells the system that there exists a family called `line_detector` (from the filename) and that the objects in the family are instances of line_detector.LineDetectorInterface.

Then, we can write the configuration files that describe the particular instances. These files have the pattern *instance name*.*family name*`.yaml`.

Continuing the example, suppose that there is a `line_detector.ConcreteLineDetector` with two parameters, `alpha` and `beta`.

Then we can define a `baseline` object by creating a file called `baseline.line_detector.yaml`, containing the following:

```yaml
description: The baseline detector
constructor: line_detector.ConcreteLineDetector
parameters:
    alpha: 1.0
    beta: 1.0
```

Similarly, we can define an `experiment` object in a file called `experiment.line_detector.yaml`:

```yaml
description: My experiment with wild parameters
constructor: line_detector.ConcreteLineDetector
parameters:
    alpha: 2.0
    beta: 13.0
```

These configuration files are parsed by EasyAlgo.

## 4.4. User interface

The user can see the list of family by using:

```
$ rosrun easy_algo summary
```

To see the list of instances available for a family, use:

```
$ rosrun easy_algo summary family name
```

## 4.5. The developer's point of view

From the developer's point of view, it is possibly to access the code using the EasyAlgoDB class.

Create an instance using get_easy_algo_db():

```
from easy_algo import get_easy_algo_db

algo_db = get_easy_algo_db()
```

Then create an instance like the following:

```
line_detector_instance = algo_db.create_instance('line_detector', 'baseline')
```

It is also possible to query the database using the function `query`:

```
# iterate over all possible line detectors
available = algo_db.query('line_detector', '*'
for name in available:
    line_detector = algo_db.create_instance('line_detector', name)
```

# Unit O-5
# Package `easy_logs`

## 5.1. Package information

Link to package on Github

### Essentials

**Author:** Andrea Censi (maintainer)

### Description

The `easy_logs` packages manages the Duckietown logs.
It indexes them and has a simple interface to query the DB.
Tests should use the `easy_logs` DB to know what logs to work on.

## 5.2. Overview

This package provides several command line utilities.
Commands to query the log database:

```
$ rosrun easy_logs summary  logs query
$ rosrun easy_logs find     logs query
$ rosrun easy_logs details  logs query
```

Command to download logs from the cloud:

```
$ rosrun easy_logs download logs query
```

Command to create thumbnails:

```
$ rosrun easy_logs thumbnails logs query
```

## 5.3. Command-line utilities `find`, `summary`, `details`

The package includes a few programs to query the database: `find`, `summary`, and `details`.
They all take the same form:

```
$ rosrun easy_logs program  query
```

where *query* is a query expressed in the selector language (Section 5.4 - Selector language).
Use the `summary` program to display a summary of the logs available:

```
$ rosrun easy_logs summary ''*dp3tele*''
| #     Log name                          date          length    vehicle name    valid
| --    ------------------------------    ----------    ------    ------------
-----------
| 0     20160504-dp3tele1-thing           2016-05-04     294 s    thing           Yes.
| 1     20160506-dp3tele1-nikola          2016-05-06     321 s    nikola          Yes.
| 2     2016-04-29-dp3tele-neptunus-0     2016-04-29     119 s    neptunus        Yes.
| 3     20160503-dp3tele1-pipquack        2016-05-03     352 s    pipquack        Yes.
| 4     20160503-dp3tele1-redrover        2016-05-03     384 s    redrover        Yes.
| 5     20160504-dp3tele1-penguin         None          (none)    None            Not
indexed
| 6     20160430-dp3tele-3-quackmobile    2016-04-30     119 s    quackmobile     Yes.
```

The `find` command is similar to `summary`, but it only displays the filenames:

```
$ rosrun easy_logs find vehicle:oreo
.../logs/20160400-phase3-dp3-demos/dp3auto_2016-04-29-19-58-57_2-oreo.bag
.../logs/20160400-phase3-dp3-demos/dp3auto_2016-04-29-19-56-57_1-oreo.bag
.../logs/20160400-phase3-dp3-demos/dp3tele_2016-04-29-19-29-57_1-oreo.bag
.../logs/20160210-M02_DPRC/onasafari/201602DD-onasafari-oreo-RCDP5-log_out.bag
.../logs/20160400-phase3-dp3-demos/dp3tele_2016-04-29-19-27-57_0-oreo.bag
.../pictures/M03_04-demo_or_die/onasafari/oreo_line_follow.bag
.../logs/20160400-phase3-dp3-demos/dp3auto_2016-04-29-19-54-57_0-oreo.bag
.../logs/20160400-phase3-dp3-demos/dp3tele_2016-04-29-19-31-57_2-oreo.bag
```

The `details` command shows a detailed view of the data structure:

```
$ rosrun easy_logs details dp3auto_2016-04-29-19-58-57_2-oreo
- - [log_name, dp3auto_2016-04-29-19-58-57_2-oreo]
  - [filename, ...]
  - [map_name, null]
  - [description, null]
  - [vehicle, oreo]
  - [date, '2016-04-29']
  - [length, 112.987947]
  - [size, 642088366]
  - - bag_info
    - compression: none
      duration: 112.987947
      end: 1461960050.639
      indexed: true
      messages: 15443
      size: 642088366
      start: 1461959937.651054
      topics:
      - {messages: 107, topic: /diagnostics,
         type: diagnostic_msgs/DiagnosticArray}
      - {messages: 8, topic: /oreo/LED_detector_node/switch,
         type: duckietown_msgs/BoolStamped}
      - {messages: 9, topic: /oreo/apriltags_global_node/apriltags,
         type: duckietown_msgs/AprilTags}
      - {messages: 9, topic: /oreo/apriltags_global_node/tags_image,
         type: sensor_msgs/Image}
 ...
```

## 5.4. Selector language

Here are some examples for the query language (Table 5.1).

Show all the Ferrari logs:

```
$ rosrun easy_logs summary vehicle:ferrari
```

All the logs of length less than 45 s:

```
$ rosrun easy_logs summary "length:<45"
```

All the invalid logs:

```
$ rosrun easy_logs summary "length:<45,valid:False"
```

All the invalid logs of length less than 45 s:

```
$ rosrun easy_logs summary "length:<45,valid:False"
```

TABLE 5.1. QUERY LANGUAGE

| expression | example | explanation |
|---|---|---|
| *attribute*:*expr* | `vehicle:ferrari` | Checks that the attribute *[attribute* of the object satisfies the expression in *expr* |
| >*lower bound* | `>10` | Lower bound |
| <*upper bound* | `<1` | Upper bound |
| *expr1*,*expr2* | `>10,<20` | And between two expressions |
| *expr1*+*expr2* | `<5+>10` | Or between two expressions |
| *pattern* | `*ferrari*` | Other strings are interpreted as wildcard patterns. |

## 5.5. Automatic log download using `download`

The command-line program `download` downloads requires logs from the cloud.

The syntax is:

```
$ rosrun easy_logs download log name
```

For example:

```
$ rosrun easy_logs download 2016-04-29-dp3auto-neptunus-1
```

If the file `2016-04-29-dp3auto-neptunus-1.bag` is not available locally, it is downloaded.

The database of URLs is at the file `dropbox.urls.yaml` in the package `easy_node`.

A typical use case would be the following, in which a script needs a log with which to work.

Using the `download` program we declare that we need the log. Then, we use `find` to find the path.

```
#!/bin/bash
set -ex

# We need the log to proceed
rosrun easy_logs download 2016-04-29-dp3auto-neptunus-1

# Here, we know that we have the log. We use `find` to get the filename.
filename=`rosrun easy_logs find 2016-04-29-dp3auto-neptunus-1`

# We can now use the log
vdir ${filename}
```

## 5.6. Browsing the cloud

How do you know which logs are in the cloud?

Run the following to download a database of all the logs available in the cloud:

```
$ make cloud-download
```

You can query the DB by using `summary` with the option `--cloud`:

```
$ rosrun easy_logs summary --cloud '*RCDP6*'

| #    Log name                                  date         length    vehicle name
| --   ----------------------------------------   ----------   ------    ------------
| 0    20160122-censi-ferrari-RCDP6-catliu       2016-02-28    194 s    ferrari
| 1    20160122-censi-ferrari-RCDP6-joe-wl       2016-02-27    196 s    ferrari
| 2    20160228-sanguk-setlist-RCDP6-sangukbo    2016-03-02    193 s    ferrari
| 3    20160122-censi-ferrari-RCDP6-eharbitz     2016-02-27    198 s    ferrari
| 4    20160122-censi-ferrari-RCDP6-teddy        2016-02-28    193 s    ferrari
| 5    20160122-censi-ferrari-RCDP6-jenshen      2016-02-29     83 s    ferrari
```

Then, you can download locally using:

```
$ rosrun easy_logs download 20160122-censi-ferrari-RCDP6-teddy
```

Once it is downloaded, the log becomes part of the local database.

## 5.7. Advanged log indexing and generation

### 1) Shuffle

*expr*`/shuffle` shuffles the order of the logs in *expr*.
Give me all the `oreo` logs, in random order:

```
$ rosrun easy_logs summary  vehicle:oreo/shuffle
```

### 2) Simple indexing

*expr*`/[i]` takes the i-th entry.
Give me the first log:

```
$ rosrun easy_logs summary  "vehicle:oreo/[0]"
```

Give me a random log; i.e. the first of a random list.

```
$ rosrun easy_logs summary  "vehicle:oreo/shuffle/[0]"
```

### 3) Complex indexing

You can use the exact Python syntax for indexing, including `[a:]`, `[:b]`, `[a:b]`, `[a:b:c]`, etc.

Give me three random logs:

```
$ rosrun easy_logs summary  "all/shuffle/[:3]"
```

### 4) Sorting

**TODO:** To implement.

### 5) Time indexing

You can ask for only a part of a log using the syntax:

```
expr/{start:stop}
expr/{start:}
expr/{:stop}
```

where *start* and *stop* are in time relative to the start of the log.
For example, "give me all the first 1-second intervals of the logs" is

```
all/{:1}
```

Cut the first 3 seconds of all the logs:

```
all/{3:}
```

Give me the interval between 30 s and 35 s:

```
all/{30:35}
```

## 5.8. How to set up the backend needed to use files from the cloud

This applies to any resource, not only logs.
First, put the file in the `duckiedown-data-2017` directory on Dropbox.
This is the link.
You need to ask Liam/Andrea because only them have write access.
Then, get the public link address and put it in the file `dropbox.urls.yaml` in the package `easy_node`. Remember to have `?dl=1` instead of `?dl=0` in the url.
In the code, use the function `require_resource()`:

```
from duckietown_utils import require_resource

zipname = require_resource('ii-datasets.zip')
```

The storage area is in `${DUCKIETOWN_ROOT}/cache/download`.
If the file is already downloaded, it is not downloaded again.

(So, if the file changes, you need to delete the cache directory. The best practice is to change the filename every time the file changes.)

The function `require_resource()` returns the path to the downloaded file.

Also note that you can put any URL in the file `dropbox.urls.yaml`; but the convention is that we only link things to Dropbox.

UNIT O-6

# Package easy_node

## 6.1. Package information

Link to package on Github

### Essentials

**Author:** Andrea Censi (maintainer)

### Description

`easy_node` is a framework to make it easier to create and document ROS nodes.

The main idea is to provide a *declarative approach* to describe:

- The node parameters;
- The node's subscriptions;
- The node's publishers;
- The node's assumptions (contracts).

The user describes subscriptions, publishers, and parameters in a YAML file.

The framework then automatically takes care of:

- Calling the necessary boilerplate ROS commands for subscribing and publishing topics.
- Loading and monitoring configuration.
- Create the Markdown documentation that describes the nodes.
- Provide a set of common functionality, such as benchmarking and monitoring latencies.

Using `easy_node` allows to cut 40%-50% of the code required for programming a node. For an example, see the package `line_detector2`, which contains a re-implementation of `line_detector` using the new framework.

**Transition plan**: The plan is to first use `easy_node` just for documenting the nodes. Then, later, convert all the nodes to use it.

## 6.2. YAML file format

For a node with the name *my node*, implemented in the file `src/`*my node*`.py` you must create a file by the name *my node*`.easy_node.yaml` somewhere in the package.

This is the smallest example of an empty configuration:

```
description: My node
parameters:
subscriptions:
publishers:
contracts:
```

### 1) parameters section: configuring parameters

This is the syntax:

```
parameters:
    name parameter:
        type: type
        desc: description
        default: default value
```

where:

- *type* is one of `float`, `int`, `bool`, `str`.
- *description* is a description that will appear in the documentation.
- The optional field `default` gives a default value for the parameter.

For example:

```
parameters:
    k_d:
        type: float
        desc: The derivative gain.
        default: 1.02
```

### 2) publishers and subscriptions section

The syntax for describing subscribers is:

```
subscriptions:
    name subscription:
        topic: topic name
        type: message type
        desc: description

        queue_size: queue size
        latch: latch
        process: process
```

The parameters are as follows.

*topic name* is the name of the topic to subscribe.

*message type* is a ROS message type name, such as `sensor_msgs/Joy`.

*description* is a Markdown description string.

*queue_size*, *latch* are optional parameters for ROS publishing/subscribing functions.

See the ROS documentation.

The optional parameter *process*, one of `synchronous` (default) or `asynchronous` describes whether to process the message in a synchronous or asynchronous way (in a separated thread).

The optional parameter *timeout* describes a timeout value. If no message is received

for more than this value, the function `on_timeout_subscription()` is called.

TODO: implement this timeout functionality.

The syntax for describing publishers is similar; it does not have the the `process` and `timeout` value.

Example:

```
subscriptions:
    segment_list:
        topic: ~segment_list
        type: duckietown_msgs/SegmentList
        desc: Line detections
        queue_size: 1
        timeout: 3

publishers:
    lane_pose:
        topic: ~lane_pose
        type: duckietown_msgs/LanePose
        desc: Estimated pose
        queue_size: 1
```

### 3) Describing contracts

**Note:** This is not implemented yet.

The idea is to have a place where we can describe constraints such as:

- "This topic must publish at least at 30 Hz."
- "Panic if you didn't receive a message for 2 seconds."
- "The maximum latency for this is 0.2 s"

Then, we can implement all these checks once and for all in a proper way, instead of relying on multiple broken implementations

## 6.3. Using the easy_node API

### 1) Initialization

Here is a minimal example of a node that conforms with the API:

```
from easy_node import EasyNode

class MyNode(EasyNode):

    def __init__(self):
        EasyNode.__init__(self, 'my_package', 'my_node')
        self.info('Initialized.')

if __name__ == '__main__':
    MyNode().spin()
```

The node class must derive from `EasyNode`. You need to tell EasyNode what is the package name and the node name.

To initialize, call the function `spin()`.

The `EasyNode` class provides the following functions:

```
info()
debug()
error()
```

These are mapped to `rospy.loginfo()` etc.; they include the name of the node.

## 2) Using configuration parameters

This next example shows how to use configuration parameters.

First, create a file `my_node.easy_node.yaml` containing:

```
parameters:
    num_cells:
        desc: Number of cells.
        type: int
        default: 42
subscriptions:
contracts:
publishers:
```

Then, implement the method `on_parameters_changed()`. It takes two parameters:

- `first_time` is a boolean that tells whether this is the first time that the function is called (initialization time).
- `updated` is a set of strings that describe the set of parameters that changed. The first time, it contains the set of all parameters.

To access the parameter value, access `self.config.`*parameter*.

Example:

```python
class MyNode():

    def __init__(self):
        EasyNode.__init__(self, 'my_package', 'my_node')

def on_parameters_changed(self, first_time, updated):
    if first_time:
        self.info('Initializing array for the first time.')
        self.cells = [0] * self.config.num_cells

    else:
        if 'num_cells' in updated:
            self.info('Number of cells changed.')
            self.cells = [0] * self.config.num_cells

if __name__ == '__main__':
    Node().spin()
```

EasyNode will monitor the ROS parameter server, and will call the function `on_parameters_changed` if the user changes any parameters.

### 3) Using subscriptions

To automatically subscribe to topics, add an entry in the `subscriptions` section of the YAML file.

For example:

```
subscriptions:
    joy:
        desc: |
            The `Joy.msg` from `joy_node` of the `joy` package.
            The vertical axis of the left stick maps to speed.
            The horizontal axis of the right stick maps to steering.
        type: sensor_msgs/Joy
        topic: ~joy
        timeout: 3.0
```

Then, implement the function `on_received_`*name*.

This function will be passed 2 arguments:

- a `context` object; this can be used for benchmarking (Section 6.6 - Benchmarking).
- the message object.

Example:

```
class MyNode():
    # ...

    def on_received_joy(self, context, msg):
        # This is called any time a message arrives
        self.info('Message received: %s' % msg)
```

### 4) Time-out

TODO: to implement

The function `on_timeout_subscription()` is called when there hasn't been a message for the specified timeout interval.

```
class MyNode():
    # ...

    def on_timeout_joy(self, context, time_since):
        # This is called when we have not seen a message for a while
        self.error('No joystick received since %s s.' % time_since)
```

### 5) Publishers

The publisher object can be accessed at `self.publishers.name`. EasyNode has taken care of all the initialization for you.

For example, suppose we specify a publisher `command` using:

```
publishers:
    command:
        desc: The control command.
        type: duckietown_msgs/Twist2DStamped
        topic: ~car_cmd
```

Then we can use it as follows.

```
class MyNode():
    # ...

    def on_received_joy(self, context, msg):
        out = Twist2DStamped()
        out.header.stamp = 0
        out.v = 0
        out.omega = 0

        self.publishers.command.publish(out)
```

### 6) `on_init()` and `on_shutdown()`

Define the two methods `on_init()` and `on_shutdown()` to c

```python
class MyNode(EasyNode):
    # ...
    def on_init(self):
        self.info('Step 1 - Initialized')

    def on_parameters_changed(self, first_time, changed):
        self.info('Step 2 - Parameters received')

    def on_shutdown(self):
        self.info('Step 3 - Preparing for shutdown.')
```

Note that `on_init()` is called before `on_parameters_changed()`.

## 6.4. Configuration using easy_node: the user's point of view

So far, we have seen how to use parameters from the node, but we did not talk about how to specify the parameters from the user's point of view.

EasyNode introduces lots of flexibility compared to the legacy system.

### 1) Configuration file location

The user configuration is specified using files by the pattern

*package_name*-*node_name*.*config_name*`.config.yaml`

where *config_name* is a short string (e.g., `baseline`).

The files can be anywhere in:

- The directory `${DUCKIETOWN_ROOT}/catkin_ws/src`;
- The directory `${DUCKIEFLEET_ROOT}`.

Several config files can exist at the same time. For example, we could have somewhere:

```
line_detector-line_detector.baseline.config.yaml
line_detector-line_detector.fall2017.config.yaml
line_detector-line_detector.andrea.config.yaml
```

where the `baseline` versions are the baseline parameters, `fall2017` are the parameters we are using for Fall 2017, and `andrea` are temporary parameters that the user is using.

However, there cannot be two configurations with the same filename e.g. two copies of `line_detector-line_detector.baseline.config.yaml`. In this case, EasyNode will raise an error.

TODO: implement this functionality.

### 2) Configuration file format

The format of the `*.config.yaml` file is as follows:

```yaml
description: |
    description of what this configuration accomplishes
extends: [config name, config name]
values:
    parameter name: value
    parameter  name: value
```

The `extends` field (optional) is a list of string. It allows to use the specified configurations as the defaults for the current one.

For example, the file `line_detector-line_detector.baseline.config.yaml` could contain:

```yaml
description: |
    These are the standard values for the line detector.
extends: []
values:
    img_size: [120,160]
    top_cutoff: 40
```

### 3) Configuration sequence

Which parameters are used depend on the **configuration sequence**.

The configuration sequence is a list of configuration names.

It can be specified by the environment variable `DUCKIETOWN_CONFIG_SEQUENCE`, using a colon-separated list of strings. For example:

```
$ export DUCKIETOWN_CONFIG_SEQUENCE=baseline:fall2017:andrea
```

The line above specifies that the configuration sequence is `baseline`, `fall2017`, `andrea`.

The system loads the configuration in order. First, it loads the `baseline` version. Then it loads the `fall2017` version. If a value was already specified in the `baseline` version, it is overwritten. If a version does not exist, it is simply skipped.

If a parameter is not specified in any configuration, an error is raised.

Using this functionality, it is easy to have team-based customization and user-based customization.

There are two special configuration names:

1. The configuration name "`defaults`" loads the defaults specified by the node. Note that the defaults are ignored otherwise.
2. The configuration name "`vehicle`" expands to the name of the vehicle being used.

**TODO:** the `vehicle` part is not implemented yet.

### 4) Time-variant configuration

EasyNode allows to describe configuration that can change in time.

The use case for this is the configuration of calibration parameters:

- Calibration parameters change with time.

- We still want to access old calibration parameters, when processing logs.

The solution is to allow a date tag in the configuration name. The format for this is

```
package_name-node_name.config_name.date.config.yaml
```

For example, we could have the files:

```
kinematics-kinematics.ferrari.20160404.config.yaml
kinematics-kinematics.ferrari.20170101.config.yaml
```

Given this, EasyNode will select the configuration to use intelligently. When reading from a bag file from 2016, the first configuration is going to be used; for logs in 2017, the second is going to be used.

**TODO:** not implemented yet.

## 6.5. Visualizing the configuration database

There are a few tolls used to visualize the configuration database.

### 1) `easy_node desc`: Describing a node

The command

```
$ rosrun easy_node desc package_name node_name
```

shows a description of the node, as specified in `package_name.node_name.easy_node.yaml`.

For example:

```
$ rosrun easy_node desc line_detector2 line_detector2
```

shows the following:

```
Configuration for node ''line_detector_node2'' in package ''line_detector2''
=====================================================================

 Parameters

    name                        type   default
    ----                        ----   -------
    en_update_params_interval   float  2.0
    top_cutoff                  int    (none)
    detector                    (n/a)  (none)
    img_size                    (n/a)  (none)
    verbose                     bool   True


 Subcriptions

    name       type                    topic       options         process
    ----       ----                    -----       -------         -------
    switch     BoolStamped             ~switch     queue_size = 1  synchronous
    image      CompressedImage         ~image      queue_size = 1  threaded
    transform  AntiInstagramTransform  ~transform  queue_size = 1  synchronous


 Publishers

    name               type         topic               options
    ----               ----         -----               -------
    color_segment      Image        ~colorSegment       queue_size = 1
    edge               Image        ~edge               queue_size = 1
    segment_list       SegmentList  ~segment_list       queue_size = 1
    image_with_lines   Image        ~image_with_lines   queue_size = 1
```

### 2) `easy_node eval`: Evaluating a configuration sequence

The program `eval` script allows to evaluate a certain configuration sequence.
The syntax is:

```
$ rosrun easy_node eval package name node name config sequence
```

The tool shows also which file is responsible for the value of each parameter.
For example, the command

```
$ rosrun easy_node eval line_detector2 line_detector_node2 defaults:andrea
```

evaluates the configuration for the `line_detector_node2` node with the configuration sequence `defaults:andrea`.
The result is:

```
Configuration result for node `line_detector_node2` (package `line_detector2`)
The configuration sequence was ['defaults', 'baseline', 'andrea'].
The following is the list of parameters set and their origin:
  parameter                    value                              origin
  ---------                    -----                              ------
  en_update_params_interval    2.0                                defaults
  top_cutoff                   40                                 baseline
  detector                     - line_detector.LineDetectorHSV    baseline
                               - configuration:
                                   canny_thresholds: [80, 200]
                                   dilation_kernel_size: 3
                                   hough_max_line_gap: 1
                                   hough_min_line_length: 3
                                   hough_threshold: 2
                                   hsv_red1: [0, 140, 100]
                                   hsv_red2: [15, 255, 255]
                                   hsv_red3: [165, 140, 100]
                                   hsv_red4: [180, 255, 255]
                                   hsv_white1: [0, 0, 150]
                                   hsv_white2: [180, 60, 255]
                                   hsv_yellow1: [25, 140, 100]
                                   hsv_yellow2: [45, 255, 255]
  img_size                     [120, 160]                         baseline
  verbose                      true                               defaults
```

Note how we can tell which configuration file is responsible for setting each parameter.

### 3) `easy_node summary`: Describing and validating all configuration files

The program `summary` reads all the configuration files described in the repository and validates them against the node description.

If a configuration file refers to parameters that do not exist, the configuration file is established to be invalid.

The syntax is:

```
$ rosrun easy_node summary
```

For example, the output can be:

```
package name      node name              config_name    effective     extends    valid
error    description
line_detector2    line_detector_node2    baseline       2017-01-01    ()
yes               These are the standard values for t [..]
```

## 6.6. Benchmarking

EasyNode implements some simple timing statistics. These are accessed using the `context` object passed to the message received callbacks.

Here's an example use, from `line_detector2`:

```
def on_received_image(self, context, image_msg):

    with context.phase('decoding'):
        ...

    with context.phase('resizing'):
        # Resize and crop image
        ...

    stats = context.get_stats()
    self.info(stats)
```

The idea is to enclose the different phases of the computation using the context manager `phase(name)`.

A summary of the statistics can be accessed by using `context.get_stats()`.

For example, this will print:

```
Last 24.4 s: received 734 (30.0 fps) processed 301 (12.3 fps) skipped 433 (17.7 fps) (59 %)
            decoding | total latency  25.5 ms | delta wall   20.7 ms | delta clock  20.7 ms
            resizing | total latency  26.6 ms | delta wall    0.8 ms | delta clock   0.7 ms
          correcting | total latency  29.1 ms | delta wall    2.2 ms | delta clock   2.2 ms
           detection | total latency  47.7 ms | delta wall   18.2 ms | delta clock  21.3 ms
    preparing-images | total latency  55.0 ms | delta wall    7.0 ms | delta clock   7.0 ms
          publishing | total latency  55.5 ms | delta wall    0.1 ms | delta clock   0.1 ms
          draw-lines | total latency  59.7 ms | delta wall    4.0 ms | delta clock   3.9 ms
    published-images | total latency  61.2 ms | delta wall    0.9 ms | delta clock   0.8 ms
pub_edge/pub_segment | total latency  86.3 ms | delta wall   24.7 ms | delta clock  24.0 ms
```

## 6.7. Automatic documentation generation

EasyNode generated the documentation automatically from the `*.easy_node.yaml` files.

Note that this can be used independently of the fact that the node implements the `EasyNode` API. So, we can use this EasyNode functionality also to document the legacy nodes.

To generate the docs, use this command:

```
$ rosrun easy_node generate_docs
```

For each node documented using a `*.easy_node.yaml`, this generates a Markdown file called "*node name*`-easy_node-autogenerated.md`" in the package directory.

The contents are enclosed in a `div` with ID `#`*package name*`-`*node name*`-autogenerated`.

The intended use is that this can be used to move the contents to the place in the documentation using the `move-here` tag.

For example, in the `README.md` of the `joy_mapper` package, we have:

```
## Node `joy_mapper_node`

<move-here src="#joy_mapper-joy_mapper_node-autogenerated"/>
```

The result can be seen at Unit P-3 - Package `joy_mapper`.

## 6.8. Parameters and services defined for all packages

(Generated from configuration `easy_node.easy_node.yaml`.)
These are properties common to all nodes.

### Parameters

No parameters defined.

### Subscriptions

No subscriptions defined.

### Publishers

No publishers defined.

## 6.9. Other ideas

(Add here other ideas that we can implement.)

# Unit O-7
# Package `easy_regression`

## 7.1. Package information

Link to package on Github

### Essentials

**Author:** Andrea Censi (maintainer)

### Description

A package to make it easy to abstract about algorithm configuration and create regression tests.

## 7.2. Design goals

TODO: to write

## 7.3. Formalization

A regression test is defined by the following:

1. A set of logs on which to operate.
2. Zero or more "processors". A processor takes a bag as input and write a bag in output.
3. Zero or more "analyzers": these are objects that analyze the results.
4. Zero or more "acceptance conditions". These are the conditions on the results of the analyzers.

Let's go more in detail, to obtain a formalization that can be used to parallelize the processing.

This is in the spirit of a map/reduce framework.

➡ Map-reduce framework XXX

We assume that a processor has the following signature:

$$\textbf{processor} : \textbf{Bag} \to \textbf{Bag}$$

An analyzer is something that returns a statistics of type `Stat`:

$$\textbf{analyzer} : \textbf{Bag} \to \textbf{Stat}$$

We also assume to have an operation `merge` that allows to merge the results of two statistics:

$$\textbf{merge} : \textbf{Stat} \times \textbf{Stat} \to \textbf{Stat}$$

Finally, the acceptance condition is a test on the statistics. We want to be careful

about the possible conditions, so instead of having only true/false, we give the following results:

$$\mathbf{acceptance} : \mathbf{Stat} \to \{\mathbf{ok}, \mathbf{fail}, \mathbf{nodata}, \mathbf{abnormal}\}$$

The semantics is as follows:

- `ok` means that the conditions is satisfied;
- `fail` means that the condition is not satisfied;
- `nodata` means that there is not enough data to check the condition. For example, there are conditions that reference the results of regression tests at another date; that data might be not available;
- `abnormal` covers all abnormal conditions, including, for example, mistakes in writing the testing conditions.

The regression test engine does the following.

It reads the log test. It makes the logs available. (This might imply downloading the logs.)

A physical log is a physical .bag file. These are generated

```
regression1.regression_test.yaml

description:
Simple regression test.
a
logs:
- "vehicle:ferrari,length:<10"
processors:
- transformer: Identity
stats:
visualizers:




identity.job_processors.yaml
description:
constructor: IdentityProcessor
parameters:



count:



S is a dict of YAML

processor: Bag -> Bag
analyzer: Bag -> S
reduce: S x S -> S
display_test: S -> Display
```

A regression test is characterized by

## 7.4. Example of configuration file

Using an EasyAlgo configuration file, we can describe a simple regression tests that counts the number of messages in a bag in a file `example.regression_test.yaml` as follows:

```
description: Simple regression test
constructor: easy_regression.RegressionTest
parameters:
    logs:
    - 20160223-amadoa-amadobot-RCDP2
    processors: []
    analyzers:
    - count_messages
    checks:
    - desc: The number of messages read should remain the same.
      cond: |
        v:count_messages/20160223-amadoa-amadobot-RCDP2/num_messages == 5330
```

In this example, there is only one log, no processors, and one analyzer.

There is one condition, that checks that the output of `count_messages` on that log is 5330.

```
v:count_messages/20160223-amadoa-amadobot-RCDP2/num_messages == 5330
```

We can run this regression test using:

```
$ rosrun easy_regression run --tests example
```

## 7.5. Language for the conditions to check

There is a flexible language that allows to check conditions.

The simplest checks regarding checking the values:

```
v:analyzer/log/statistics == value
v:analyzer/log/statistics &gt;= value
v:analyzer/log/statistics &lt;= value
v:analyzer/log/statistics &lt; value
v:analyzer/log/statistics &gt; value
```

Check that it is in 10% of the value:

```
v:analyzer/log/statistics ==[10%] value
```

Use `@`*date* to reference the test result at that date. For example,

```
v:analyzer/log/statistics ==[10%] v:analyzer/log/statistic@2017-08-01
```

Use `~branch@date` to reference the value of a branch at a certain date:

```
v:analyzer/log/statistics ==[10%] v:analyzer/log/statistic~master@2017-08-01
```

Use `?commit` to reference the value of a branch at a specific commit:

```
v:analyzer/log/statistics ==[10%] v:analyzer/log/statistic?e9aa5f4
```

# Unit O-8
# Package what_the_duck

## 8.1. Package information

Link to package on Github

### Essentials

**Author:** Andrea Censi (maintainer)

### Description

`what-the-duck` is a program that tests *dozens* of configuration inconsistencies that can happen on a Duckiebot.

### 3) The `what-the-duck` program

The proper usage of `what-the-duck` to debug an environment problem is the following sequence:

```
$ # open a new terminal
$ cd Duckietown root
$ git checkout master
$ git pull
$ source environment.sh
$ ./dependencies_for_duckiebot.sh # if you are on a Duckiebot
$ ./dependencies_for_laptop.sh    # if you are on a laptop
$ make build-clean
$ make build-catkin
$ ./what-the-duck
```

> **Note:** you have to do all the steps in the precise order.

### 4) Seeing the fleet results

The telemetry is collected and available at this URL.

# Part P
# Packages - Teleoperation

TODO: to write

# Unit P-1
# Package `adafruit_drivers`

## 1.1. Package information

Link to package on Github

### Essentials

**Author:** Dmitry Yershov
**Maintainer:** Mack

### Description

**TODO:** Add a description of package `adafruit_drivers` in `package.xml`.

These are the Adafruit drivers.

**TODO:** What is the original location of this package?

UNIT P-2

# Package `dagu_car`

## 2.1. Package information

Link to package on Github

### Essentials

**Author:** Dmitry Yershov
**Author:** Shih-Yuan Liu
**Maintainer:** Mack

### Description

**TODO:** Add a description of package `dagu_car` in `package.xml`.

## 2.2. Node `forward_kinematics_node`

(Generated from configuration `forward_kinematics_node.easy_node.yaml`.)

**TODO:** Missing node description in `forward_kinematics_node.easy_node.yaml`.

### Parameters

No parameters defined.

### Subscriptions

No subscriptions defined.

### Publishers

No publishers defined.

## 2.3. Node `inverse_kinematics_node`

(Generated from configuration `inverse_kinematics_node.easy_node.yaml`.)

**TODO:** Missing node description in `inverse_kinematics_node.easy_node.yaml`.

### Parameters

No parameters defined.

### Subscriptions

No subscriptions defined.

### Publishers

No publishers defined.

## 2.4. Node `velocity_to_pose_node`

(Generated from configuration `velocity_to_pose_node.easy_node.yaml`.)

TODO: Missing node description in `velocity_to_pose_node.easy_node.yaml`.

### Parameters

No parameters defined.

### Subscriptions

No subscriptions defined.

### Publishers

No publishers defined.

## 2.5. Node `car_cmd_switch_node`

(Generated from configuration `car_cmd_switch_node.easy_node.yaml`.)

TODO: Missing node description in `car_cmd_switch_node.easy_node.yaml`.

### Parameters

No parameters defined.

### Subscriptions

No subscriptions defined.

### Publishers

No publishers defined.

## 2.6. Node `wheels_driver_node`

(Generated from configuration `wheels_driver_node.easy_node.yaml`.)

TODO: Missing node description in `wheels_driver_node.easy_node.yaml`.

### Parameters

No parameters defined.

### Subscriptions

No subscriptions defined.

### Publishers

No publishers defined.

## 2.7. Node `wheels_trimmer_node`

(Generated from configuration `wheels_trimmer_node.easy_node.yaml`.)

**TODO:** Missing node description in `wheels_trimmer_node.easy_node.yaml`.

### Parameters

No parameters defined.

### Subscriptions

No subscriptions defined.

### Publishers

No publishers defined.

# Unit P-3
# Package joy_mapper

## 3.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Mack

### Description

The joy_mapper package for duckietown. Takes sensor_msgs.Joy and convert it to duckietown_msgs.CarControl.

## 3.2. Testing

Connect joystick.

Run:

```
$ roslaunch launch/joy_mapper_test.launch
```

The robot should move when you push buttons.

## 3.3. Dependencies

- `rospy`
- `sensor_msgs`: for the `Joy.msg`
- `duckietown_msgs`: for the `CarControl.msg`

## 3.4. Node joy_mapper_node2

(Generated from configuration `joy_mapper_node2.easy_node.yaml`.)

This node takes a `sensor_msgs/Joy.msg` and converts it to a `duckietown_msgs/CarControl.msg`.

It publishes at a fixed interval with a zero-order hold.

### Parameters

**Parameter** `v_gain`: `float`; default value: `0.41`

TODO: Missing description for entry "`v_gain`".

**Parameter** `omega_gain`: `float`; default value: `8.3`

TODO: Missing description for entry "`omega_gain`".

**Parameter** `bicycle_kinematics`: `int`; default value: `0`

TODO: Missing description for entry "`bicycle_kinematics`".

**Parameter** `steer_angle_gain`: `int`; default value: `1`

TODO: Missing description for entry "`steer_angle_gain`".

**Parameter** `simulated_vehicle_length`: `float`; default value: `0.18`

TODO: Missing description for entry "`simulated_vehicle_length`".

## Subscriptions

**Subscription** `joy`: topic `joy` (`Joy`)

The `Joy.msg` from `joy_node` of the `joy` package. The vertical axis of the left stick maps to speed. The horizontal axis of the right stick maps to steering.

## Publishers

**Publisher** `car_cmd`: topic `~car_cmd` (`Twist2DStamped`)

TODO: Missing description for entry "`car_cmd`".

**Publisher** `joy_override`: topic `~joystick_override` (`BoolStamped`)

TODO: Missing description for entry "`joy_override`".

**Publisher** `parallel_autonomy`: topic `~parallel_autonomy` (`BoolStamped`)

TODO: Missing description for entry "`parallel_autonomy`".

**Publisher** `anti_instagram`: topic `anti_instagram_node/click` (`BoolStamped`)

TODO: Missing description for entry "`anti_instagram`".

**Publisher** `e_stop`: topic `wheels_driver_node/emergency_stop` (`BoolStamped`)

TODO: Missing description for entry "`e_stop`".

**Publisher** `avoidance`: topic `~start_avoidance` (`BoolStamped`)

TODO: Missing description for entry "`avoidance`".

# Unit P-4
# Package `pi_camera`

## 4.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Andrea Censi

### Description

TODO: Add a description of package `pi_camera` in `package.xml`.

## 4.2. Node `camera_node_sequence`

(Generated from configuration `camera_node_sequence.easy_node.yaml`.)
Camera driver, second approach.

### Parameters

No parameters defined.

### Subscriptions

No subscriptions defined.

### Publishers

No publishers defined.

## 4.3. Node `camera_node_continuous`

(Generated from configuration `camera_node_continuous.easy_node.yaml`.)
Camera driver.

### Parameters

**Parameter** `framerate`: `float`; default value: `60.0`
Frame rate
**Parameter** `res_w`: `int`; default value: `320`
Resolution (width)
**Parameter** `res_h`: `int`; default value: `200`
Resolution (height)

### Subscriptions

No subscriptions defined.

#### Publishers

**Publisher** `image_compressed`: topic `~image/compressed''` (`CompressedImage`)

TODO: Missing description for entry "`image_compressed`".

### 4.4. Node `decoder_node`

(Generated from configuration `decoder_node.easy_node.yaml`.)

A node that decodes a compressed image into a regular image. This is useful so that multiple nodes that need to use the image do not do redundant computaon.

#### Parameters

**Parameter** `publish_freq`: `float`; default value: `1.0`

Frequency at which to publish (Hz).

#### Subscriptions

**Subscription** `compressed_image`: topic `~compressed_image` (`CompressedImage`)

The image to decode.

**Subscription** `switch`: topic `~switch` (`BoolStamped`)

Switch to turn on or off. The node starts as active.

#### Publishers

**Publisher** `raw`: topic `~image/raw` (`Image`)

The decoded image.

### 4.5. Node `img_process_node`

(Generated from configuration `img_process_node.easy_node.yaml`.)

Apparently, a template, or a node never finished.

#### Parameters

No parameters defined.

#### Subscriptions

No subscriptions defined.

#### Publishers

No publishers defined.

### 4.6. Node `cam_info_reader_node`

(Generated from configuration `cam_info_reader_node.easy_node.yaml`.)

Publishes a CameraInfo message every time it receives an image.

### Parameters

**Parameter** `config`: `str`; default value: `'baseline'`

TODO: Missing description for entry "`config`".

**Parameter** `cali_file_name`: `str`; default value: `'default'`

TODO: Missing description for entry "`cali_file_name`".

**Parameter** `image_type`: `str`; default value: `'compressed'`

TODO: Missing description for entry "`image_type`".

### Subscriptions

**Subscription** `compressed_image`: topic `~compressed_image` (`CompressedImage`)
If image_type is "compressed" then it's CompressedImage, otherwise Image.

### Publishers

**Publisher** `camera_info`: topic `~camera_info` (`CameraInfo`)

TODO: Missing description for entry "`camera_info`".

# Part Q
# Packages - Lane control

TODO: to write

# Unit Q-1
# Package anti_instagram

## 1.1. Package information

Link to package on Github

### Essentials

**Author:** mfe
**Maintainer:** Mack

### Description

TODO: Add a description of package `anti_instagram` in `package.xml`.

## 1.2. Unit tests integrated with rostest

Unit tests are integrated with `rostest`.
To run manually, use:

```
$ rostest anti_instagram antiinstagram_correctness_test.test
$ rostest anti_instagram antiinstagram_stub_test.test
$ rostest anti_instagram antiinstagram_performance_test.test
```

## 1.3. Unit tests needed external files

These are other unittest that require the logs in DUCKIETOWN_DATA:

```
$ rosrun anti_instagram annotations_test.py
```

## 1.4. Node anti_instagram_node

(Generated from configuration `anti_instagram_node.easy_node.yaml`.)

TODO: Missing node description in `anti_instagram_node.easy_node.yaml`.

### Parameters

**Parameter** `publish_corrected_image`: `bool`; default value: `False`
Whether to compute and publish the corrected image.

### Subscriptions

**Subscription** `image`: topic `~uncorrected_image` (`CompressedImage`)
This is the compressed image to read.

**Subscription** `click`: topic `~click` (`BoolStamped`)
Activate the calibration phase with this switch.

### Publishers

**Publisher** `image`: topic `~corrected_image` (`Image`)
The corrected image.
**Publisher** `health`: topic `~colorSegment` (`AntiInstagramHealth`)
The health of the process.
**Publisher** `transform`: topic `~transform` (`AntiInstagramTransform`)
The computed transform.

# Unit Q-2
# Package complete_image_pipeline

## 2.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Mack

### Description

This package contains functions that allow to call the entire message pipeline programmatically, as a Python function, without having to instantiate any ROS node.

This is useful for regression tests, and quick tests with new logs.

## 2.2. Program single_image_pipeline

This runs the entire pipeline:

```
$ rosrun complete_image_pipeline single_image_pipeline --image image --line_detector
detector --image_pre image_prep
```

where *image* can be a filename or a URL.

For example, this:

```
$ rosrun complete_image_pipeline single_image_pipeline --image "https://www.dropbox.com/s/
r1bpyb8fd5577dm/frame0002.jpg?dl=1"
```

results in an output as in Figure 2.1.

Figure 2.1. Output of `single_image_pipeline`

# Unit Q-3
# Basic Markduck guide

The Duckiebook is written in Markduck, a Markdown dialect.
It supports many features that make it possible to create publication-worthy materials.

## 3.1. Markdown

The Duckiebook is written in a Markdown dialect.

➡ A tutorial on Markdown.

## 3.2. Variables in command lines and command output

Use the syntax "`![name]`" for describing the variables in the code.

*example*

For example, to obtain:

```
$ ssh robot name.local
```

Use the following:

```
For example, to obtain:

    $ ssh ![robot name].local
```

Make sure to quote (with 4 spaces) all command lines. Otherwise, the dollar symbol confuses the LaTeX interpreter.

## 3.3. Character escapes

Use the string "`$`" to write the dollar symbol "$", otherwise it gets confused with LaTeX math materials. Also notice that you should probably use "USD" to refer to U.S. dollars.
Other symbols to escape are shown in Table 3.1.

Table 3.1. Symbols to escape

| | |
|---|---|
| use `$` | instead of $ |
| use ``` | instead of ` |
| use `&#lt;` | instead of < |
| use `&#gt;` | instead of > |

## 3.4. Keyboard keys

Use the `kbd` element for keystrokes.

*example*

For example, to obtain:

Press a then Ctrl-C.

use the following:

```
Press <kbd>a</kbd> then <kbd>Ctrl</kbd>-<kbd>C</kbd>.
```

## 3.5. Figures

For any element, adding an attribute called `figure-id` with value `fig:`*figure ID* or `tab:`*table ID* will create a figure that wraps the element.

For example:

```
<div figure-id="fig:figure ID">
    figure content
</div>
```

It will create HMTL of the form:

```
<div id='fig:code-wrap' class='generated-figure-wrap'>
    <figure id='fig:figure ID' class='generated-figure'>
        <div>
            figure content
        </div>
    </figure>
</div>
```

To add a caption, add an attribute `figure-caption`:

```
<div figure-id="fig:figure ID" figure-caption="This is my caption">
    figure content
</div>
```

Alternatively, you can put anywhere an element `figcaption` with ID *figure id*`:caption`:

```
<element figure-id="fig:figure ID">
    figure content
</element>

<figcaption id='fig:figure ID:caption'>
    This the caption figure.
</figcaption>
```

To refer to the figure, use an empty link:

```
Please see [](#fig:figure ID).
```

The code will put a reference to “Figure XX”.

## 3.6. Subfigures

You can also create subfigures, using the following syntax.

```
<div figure-id="fig:big">
    <figcaption>Caption of big figure</figcaption>

    <div figure-id="subfig:first" figure-caption="Caption 1">
        <p style='width:5em;height:5em;background-color:#eef'>first subfig</p>
    </div>

    <div figure-id="subfig:second" figure-caption="Caption 2">
        <p style='width:5em;height:5em;background-color:#fee'>second subfig</p>
    </div>
</div>
```

This is the result:

first subfig

(a) Caption 1

second subfig

(b) Caption 2

Figure 3.1. Caption of big figure

By default, the subfigures are displayed one per line.

To make them flow horizontally, add `figure-class="flow-subfigures"` to the external figure `div`. Example:

first subfig

(a) Caption 1

second subfig

(b) Caption 2

Figure 3.2. Caption of big figure

## 3.7. Shortcut for tables

The shortcuts `col2`, `col3`, `col4`, `col5` are expanded in tables with 2, 3, 4 or 5 columns. The following code:

```
<col2 figure-id="tab:mytable" figure-caption="My table">
    <span>A</span>
    <span>B</span>
    <span>C</span>
    <span>D</span>
</col2>
```

gives the following result:

TABLE 3.2. MY TABLE

| | |
|---|---|
| A | B |
| C | D |

### 1) labels-row1 and labels-row1

Use the classes `labels-row1` and `labels-row1` to make pretty tables like the following.

`labels-row1`: the first row is the headers.

`labels-col1`: the first column is the headers.

TABLE 3.3. USING CLASS="LABELS-COL1"

| | | | |
|---|---|---|---|
| **header A** | B | C | 1 |
| **header D** | E | F | 2 |
| **header G** | H | I | 3 |

TABLE 3.4. USING CLASS="LABELS-ROW1"

| header A | header B | header C |
|---|---|---|
| D | E | F |
| G | H | I |
| 1 | 2 | 3 |

## 3.8. Linking to documentation

### 1) Establishing names of headers

You give IDs to headers using the format:

```
### header title {#topic ID}
```

For example, for this subsection, we have used:

```
### Establishing names of headers {#establishing}
```

With this, we have given this header the ID "`establishing`".

### 2) How to name IDs - and why it's not automated

Some time ago, if there was a section called

```
## My section
```

then it would be assigned the ID "my-section".

This behavior has been removed, for several reasons.

One is that if you don't see the ID then you will be tempted to just change the name:

```
## My better section
```

and silently the ID will be changed to "my-better-section" and all the previous links will be invalidated.

The current behavior is to generate an ugly link like "autoid-209u31j".

This will make it clear that you cannot link using the PURL if you don't assign an ID.

Also, I would like to clarify that all IDs are *global* (so it's easy to link stuff, without thinking about namespaces, etc.).

Therefore, please choose descriptive IDs, with at least two IDs.

E.g. if you make a section called

```
## Localization  {#localization}
```

that's certainly a no-no, because "localization" is too generic.

✔

```
## Localization {#intro-localization}
```

Also note that you don't *need* to add IDs to everything, only the things that people could link to. (e.g. not subsubsections)

### 3) Linking from the documentation to the documentation

You can use the syntax:

```
[](#topic ID)
```

to refer to the header.

You can also use some slightly more complex syntax that also allows to link to only the name, only the number or both (Table 3.5).

TABLE 3.5. SYNTAX FOR REFERRING TO SECTIONS.

```
See [](#establishing).
```

See Subsection 2.8.1 - Establishing names of headers

```
See <a class="only_name" href="#establishing"></a>.
```

See Establishing names of headers.

```
See <a class="only_number" href="#establishing"></a>.
```

See 2.8.1.

```
See <a class="number_name" href="#establishing"></a>.
```

See Subsection 2.8.1 - Establishing names of headers.

### 4) Linking to the documentation from outside the documentation

You are encouraged to put links to the documentation from the code or scripts.

To do so, use links of the form:

```
http://purl.org/dth/topic ID
```

Here "`dth`" stands for "Duckietown Help". This link will get redirected to the corresponding document on the website.

For example, you might have a script whose output is:

```
$ rosrun mypackage myscript
Error. I cannot find the scuderia file.
See: http://purl.org/dth/scuderia
```

When the user clicks on the link, they will be redirected to Section 4.2 - The "scuderia" (vehicle database).

# UNIT Q-4
# Basic Markduck guide

The Duckiebook is written in Markduck, a Markdown dialect.
It supports many features that make it possible to create publication-worthy materials.

## 4.1. Markdown

The Duckiebook is written in a Markdown dialect.

➡ A tutorial on Markdown.

## 4.2. Variables in command lines and command output

Use the syntax "`![name]`" for describing the variables in the code.

*example*

For example, to obtain:

```
$ ssh robot name.local
```

Use the following:

```
For example, to obtain:

    $ ssh ![robot name].local
```

Make sure to quote (with 4 spaces) all command lines. Otherwise, the dollar symbol confuses the LaTeX interpreter.

## 4.3. Character escapes

Use the string "`$`" to write the dollar symbol "`$`", otherwise it gets confused with LaTeX math materials. Also notice that you should probably use "USD" to refer to U.S. dollars.
Other symbols to escape are shown in Table 4.1.

TABLE 4.1. SYMBOLS TO ESCAPE

| | |
|---|---|
| use `$` | instead of `$` |
| use ``` | instead of `` ` `` |
| use `&#lt;` | instead of `<` |
| use `&#gt;` | instead of `>` |

## 4.4. Keyboard keys

Use the `kbd` element for keystrokes.

*example*

For example, to obtain:

Press a then Ctrl-C.

use the following:

```
Press <kbd>a</kbd> then <kbd>Ctrl</kbd>-<kbd>C</kbd>.
```

## 4.5. Figures

For any element, adding an attribute called `figure-id` with value `fig:`*figure ID* or `tab:`*table ID* will create a figure that wraps the element.

For example:

```
<div figure-id="fig:figure ID">
    figure content
</div>
```

It will create HMTL of the form:

```
<div id='fig:code-wrap' class='generated-figure-wrap'>
    <figure id='fig:figure ID' class='generated-figure'>
        <div>
            figure content
        </div>
    </figure>
</div>
```

To add a caption, add an attribute `figure-caption`:

```
<div figure-id="fig:figure ID" figure-caption="This is my caption">
    figure content
</div>
```

Alternatively, you can put anywhere an element `figcaption` with ID *figure id*`:caption`:

```
<element figure-id="fig:figure ID">
    figure content
</element>

<figcaption id='fig:figure ID:caption'>
    This the caption figure.
</figcaption>
```

To refer to the figure, use an empty link:

```
Please see [](#fig:figure ID).
```

The code will put a reference to “Figure XX”.

## 4.6. Subfigures

You can also create subfigures, using the following syntax.

```
<div figure-id="fig:big">
    <figcaption>Caption of big figure</figcaption>

    <div figure-id="subfig:first" figure-caption="Caption 1">
        <p style='width:5em;height:5em;background-color:#eef'>first subfig</p>
    </div>

    <div figure-id="subfig:second" figure-caption="Caption 2">
        <p style='width:5em;height:5em;background-color:#fee'>second subfig</p>
    </div>
</div>
```

This is the result:

first subfig

(a) Caption 1

second subfig

(b) Caption 2

Figure 4.1. Caption of big figure

By default, the subfigures are displayed one per line.

To make them flow horizontally, add `figure-class="flow-subfigures"` to the external figure `div`. Example:

first subfig

(a) Caption 1

second subfig

(b) Caption 2

Figure 4.2. Caption of big figure

## 4.7. Shortcut for tables

The shortcuts `col2`, `col3`, `col4`, `col5` are expanded in tables with 2, 3, 4 or 5 columns. The following code:

```
<col2 figure-id="tab:mytable" figure-caption="My table">
    <span>A</span>
    <span>B</span>
    <span>C</span>
    <span>D</span>
</col2>
```

gives the following result:

TABLE 4.2. MY TABLE

| | |
|---|---|
| A | B |
| C | D |

### 1) labels-row1 and labels-row1

Use the classes `labels-row1` and `labels-row1` to make pretty tables like the following.

`labels-row1`: the first row is the headers.

`labels-col1`: the first column is the headers.

TABLE 4.3. USING CLASS="LABELS-COL1"

| | | | |
|---|---|---|---|
| **header A** | B | C | 1 |
| **header D** | E | F | 2 |
| **header G** | H | I | 3 |

TABLE 4.4. USING CLASS="LABELS-ROW1"

| header A | header B | header C |
|---|---|---|
| D | E | F |
| G | H | I |
| 1 | 2 | 3 |

## 4.8. Linking to documentation

### 1) Establishing names of headers

You give IDs to headers using the format:

```
### header title {#topic ID}
```

For example, for this subsection, we have used:

```
### Establishing names of headers {#establishing}
```

With this, we have given this header the ID "`establishing`".

### 2) How to name IDs - and why it's not automated

Some time ago, if there was a section called

```
## My section
```

then it would be assigned the ID "my-section".

This behavior has been removed, for several reasons.

One is that if you don't see the ID then you will be tempted to just change the name:

```
## My better section
```

and silently the ID will be changed to "my-better-section" and all the previous links will be invalidated.

The current behavior is to generate an ugly link like "autoid-209u31j".

This will make it clear that you cannot link using the PURL if you don't assign an ID.

Also, I would like to clarify that all IDs are *global* (so it's easy to link stuff, without thinking about namespaces, etc.).

Therefore, please choose descriptive IDs, with at least two IDs.

E.g. if you make a section called

```
## Localization  {#localization}
```

that's certainly a no-no, because "localization" is too generic.

✔

```
## Localization {#intro-localization}
```

Also note that you don't *need* to add IDs to everything, only the things that people could link to. (e.g. not subsubsections)

### 3) Linking from the documentation to the documentation

You can use the syntax:

```
[](#topic ID)
```

to refer to the header.

You can also use some slightly more complex syntax that also allows to link to only the name, only the number or both (Table 4.5).

TABLE 4.5. SYNTAX FOR REFERRING TO SECTIONS.

```
See [](#establishing).
```

See Subsection 2.8.1 - Establishing names of headers

```
See <a class="only_name" href="#establishing"></a>.
```

See Establishing names of headers.

```
See <a class="only_number" href="#establishing"></a>.
```

See 2.8.1.

```
See <a class="number_name" href="#establishing"></a>.
```

See Subsection 2.8.1 - Establishing names of headers.

### 4) Linking to the documentation from outside the documentation

You are encouraged to put links to the documentation from the code or scripts.

To do so, use links of the form:

```
http://purl.org/dth/topic ID
```

Here "`dth`" stands for "Duckietown Help". This link will get redirected to the corresponding document on the website.

For example, you might have a script whose output is:

```
$ rosrun mypackage myscript
Error. I cannot find the scuderia file.
See: http://purl.org/dth/scuderia
```

When the user clicks on the link, they will be redirected to Section 4.2 - The "scuderia" (vehicle database).

UNIT Q-5

# Package `ground_projection`

## 5.1. Package information

Link to package on Github

### Essentials

**Author:** Changhyun Choi

**Maintainer:** Mack

### Description

**TODO:** Add a description of package `ground_projection` in `package.xml`.

## 5.2. Node `ground_projection_node`

(Generated from configuration `ground_projection.easy_node.yaml`.)

**TODO:** node description for ground_projection

### Parameters

No parameters defined.

### Subscriptions

No subscriptions defined.

### Publishers

No publishers defined.

UNIT Q-6

# Package `lane_control`

## 6.1. Package information

Link to package on Github

### Essentials

**Author:** steven

**Author:** Shih-Yuan Liu

**Maintainer:** Liam Paull

### Description

**TODO:** Add a description of package `lane_control` in `package.xml`.

## 6.2. `lane_controller_node`

(Generated from configuration `lane_controller_node.easy_node.yaml`.)

> **Note:** there is some very funny business inside. It appears that `k_d` and `k_theta` are switched around.

### Parameters

**Parameter** `v_bar`: `float`

Nominal linear velocity (m/s).

**Parameter** `k_theta`: `float`

Proportional gain for $\boldsymbol{\theta}$.

**Parameter** `k_d`: `float`

Propertional gain for $\boldsymbol{d}$.

**Parameter** `d_thres`: `float`

Cap for error in $\boldsymbol{d}$.

**Parameter** `theta_thres`: `float`

Maximum desired $\boldsymbol{\theta}$.

**Parameter** `d_offset`: `float`

A configurable offset from the lane position.

### Subscriptions

**Subscription** `lane_reading`: topic `~lane_pose` (`LanePose`)

**TODO:** Missing description for entry "`lane_reading`".

### Publishers

**Publisher** `car_cmd`: topic `~car_cmd` (`Twist2DStamped`)

**TODO:** Missing description for entry "`car_cmd`".

# Unit Q-7
# Package `lane_filter`

> Assigned to: Liam

## 7.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Liam Paull

### Description

TODO: Add a description of package `lane_filter` in `package.xml`.

## 7.2. `lane_filter_node`

(Generated from configuration `lane_filter_node.easy_node.yaml`.)

TODO: Missing node description in `lane_filter_node.easy_node.yaml`.

### Parameters

**Parameter** `mean_d_0`: `float`; default value: `0.0`

TODO: Missing description for entry "`mean_d_0`".

**Parameter** `mean_phi_0`: `float`; default value: `0.0`

TODO: Missing description for entry "`mean_phi_0`".

**Parameter** `sigma_d_0`: `float`; default value: `0.0`

TODO: Missing description for entry "`sigma_d_0`".

**Parameter** `sigma_phi_0`: `float`; default value: `0.0`

TODO: Missing description for entry "`sigma_phi_0`".

**Parameter** `delta_d`: `float`; default value: `0.02`
(meters)

**Parameter** `delta_phi`: `float`; default value: `0.0`
(radians)

**Parameter** `d_max`: `float`; default value: `0.5`

TODO: Missing description for entry "`d_max`".

**Parameter** `d_min`: `float`; default value: `-0.7`

TODO: Missing description for entry "`d_min`".

**Parameter** `phi_min`: `float`; default value: `-1.5707`

TODO: Missing description for entry "`phi_min`".

**Parameter** `phi_max`: `float`; default value: `1.5707`

TODO: Missing description for entry "`phi_max`".

**Parameter** `cov_v`: `float`; default value: `0.5`

Linear velocity "input".

> XXX which units?

**Parameter** `cov_omega`: `float`; default value: `0.01`

Angular velocity "input".

> XXX which units?

**Parameter** `linewidth_white`: `float`; default value: `0.04`

TODO: Missing description for entry "`linewidth_white`".

**Parameter** `linewidth_yellow`: `float`; default value: `0.02`

TODO: Missing description for entry "`linewidth_yellow`".

**Parameter** `lanewidth`: `float`; default value: `0.4`

TODO: Missing description for entry "`lanewidth`".

**Parameter** `min_max`: `float`; default value: `0.3`

Expressed in nats.

**Parameter** `use_distance_weighting`: `bool`; default value: `False`

For use of distance weighting (dw) function.

**Parameter** `zero_val`: `float`; default value: `1.0`

TODO: Missing description for entry "`zero_val`".

**Parameter** `l_peak`: `float`; default value: `1.0`

TODO: Missing description for entry "`l_peak`".

**Parameter** `peak_val`: `float`; default value: `10.0`

TODO: Missing description for entry "`peak_val`".

**Parameter** `l_max`: `float`; default value: `2.0`

TODO: Missing description for entry "`l_max`".

**Parameter** `use_max_segment_dist`: `bool`; default value: `False`

For use of maximum segment distance.

**Parameter** `max_segment_dist`: `float`; default value: `1.0`

For use of maximum segment distance.

**Parameter** `use_min_segs`: `bool`; default value: `False`

For use of minimum segment count.

**Parameter** `min_segs`: `int`; default value: `10`

For use of minimum segment count.

**Parameter** `use_propagation`: `bool`; default value: `False`

For propagation.

**Parameter** `sigma_d_mask`: `float`; default value: `0.05`

TODO: Missing description for entry "`sigma_d_mask`".

**Parameter** `sigma_phi_mask`: `float`; default value: `0.05`

TODO: Missing description for entry "`sigma_phi_mask`".

## Subscriptions

**Subscription** `velocity`: topic `~velocity` (`Twist2DStamped`)

TODO: Missing description for entry "`velocity`".

**Subscription** `segment_list`: topic `~segment_list` (`SegmentList`)

TODO: Missing description for entry "`segment_list`".

## Publishers

**Publisher** `lane_pose`: topic `~lane_pose` (`LanePose`)

TODO: Missing description for entry "`lane_pose`".

**Publisher** `belief_img`: topic `~belief_img` (`Image`)

TODO: Missing description for entry "`belief_img`".

**Publisher** `entropy`: topic `~entropy` (`Float32`)

TODO: Missing description for entry "`entropy`".

**Publisher** `in_lane`: topic `~in_lane` (`BoolStamped`)

TODO: Missing description for entry "`in_lane`".

**Publisher** `switch`: topic `~switch` (`BoolStamped`)

TODO: Missing description for entry "`switch`".

# UNIT Q-8
# Package line_detector2

## 8.1. Package information

Link to package on Github

### Essentials

**Author:** Andrea Censi (maintainer)

### Description

A re-implementation of the line detector node.

This is a re-implementation of the package `line_detector` using the new facilities provided by `easy_node`.

## 8.2. Testing the line detector using visual inspection

The following are instructions to test the line detector from bag files.

You can run from a bag with the following:

```
$ roslaunch line_detector line_detector_bag.launch veh:=vehicle bagin:=bag in
bagout:=bag out verbose:=true
```

Where:

- *bag in* is the **absolute path** of the input bag.
- *vehicle* is the name of the vehicle that took the log.
- *bag out* is the **absolute path** if the output bag.

> Note: you always need to use absolute paths for bag files.

You can let this run for a few seconds, then stop using Ctrl-C.

You can then inspect the result using:

```
$ roscore &
$ rosbag play -l bag out
$ rviz &
```

In `rviz` click "add", click "by topic" tab, expand "`line_detector`" and click "`image_with_lines`".

Observe on the result that:

1. There are *lots* of detections.
2. Predominantly white detections (indicated in black) are on white lines, yellow detections (shown in blue) are on blue lines, and red detections (shown in green) are on red lines.

These are some sample logs on which to try:

```
$ wget -O 160122-manual1_ferrari.bag https://www.dropbox.com/s/8bpi656j7qox5kv?dl=1

https://www.dropbox.com/s/vwznjke4xvnhi9o/160122_manual2-ferrari.bag?dl=1

https://www.dropbox.com/s/y7uljl98punj0mp/160122_manual3_corner-ferrari.bag?dl=1

https://www.dropbox.com/s/d4n9otmlans4i62/160122-calibration-good_lighting-tesla.bag?dl=1
```

Sample output:

```
From cmd:roslaunch duckietown camera.launch veh:=${VEHICLE_NAME}
[INFO] [WallTime: 1453839555.948481] [LineDetectorNode] number of white lines = 14
[INFO] [WallTime: 1453839555.949102] [LineDetectorNode] number of yellow lines = 33
[INFO] [WallTime: 1453839555.986520] [LineDetectorNode] number of white lines = 18
[INFO] [WallTime: 1453839555.987039] [LineDetectorNode] number of yellow lines = 34
[INFO] [WallTime: 1453839556.013252] [LineDetectorNode] number of white lines = 14
[INFO] [WallTime: 1453839556.013857] [LineDetectorNode] number of yellow lines = 29
[INFO] [WallTime: 1453839556.014539] [LineDetectorNode] number of red lines = 2
[INFO] [WallTime: 1453839556.047944] [LineDetectorNode] number of white lines = 18
[INFO] [WallTime: 1453839556.048672] [LineDetectorNode] number of yellow lines = 28
[INFO] [WallTime: 1453839556.049534] [LineDetectorNode] number of red lines = 2
[INFO] [WallTime: 1453839556.081400] [LineDetectorNode] number of white lines = 13
[INFO] [WallTime: 1453839556.081944] [LineDetectorNode] number of yellow lines = 34
[INFO] [WallTime: 1453839556.082479] [LineDetectorNode] number of red lines = 1
```

The output from `rviz` looks like Figure 8.1.

Figure 8.1. RViz output

## 8.3. Quantitative tests

TODO: Something more quantitative (to be filled in by Liam or Hang)

## 8.4. line_detector_node2

(Generated from configuration line_detector_node2.easy_node.yaml.)

This is a rewriting of line_detector_node using the EasyNode framework.

### Parameters

**Parameter** verbose: bool; default value: True

Whether the node is verbose or not. If set to True, the node will write timing statistics to the log.

**Parameter** img_size: not known

TODO: Missing description for entry "img_size".

**Parameter** top_cutoff: int

This parameter decides how much of the image we should cut off. This is a performance improvement.

**Parameter** line_detector: str

This is the instance of line_detector to use.

### Subscriptions

**Subscription** image: topic ~image (CompressedImage)

This is the compressed image to read. Note that it takes a long time to simply decode the image JPG.

> Note: The data is processed *asynchronously* in a different thread.

**Subscription** transform: topic ~transform (AntiInstagramTransform)

The anti-instagram transform to apply. See Unit Q-1 - Package anti_instagram.

**Subscription** switch: topic ~switch (BoolStamped)

This is a switch that allows to control the activity of this node. If the message is true, the node becomes active. If false, it switches off. The node starts as active.

### Publishers

**Publisher** edge: topic ~edge (Image)

TODO: Missing description for entry "edge".

**Publisher** color_segment: topic ~colorSegment (Image)

TODO: Missing description for entry "color_segment".

**Publisher** segment_list: topic ~segment_list (SegmentList)

TODO: Missing description for entry "segment_list".

**Publisher** image_with_lines: topic ~image_with_lines (Image)

TODO: Missing description for entry "image_with_lines".

# UNIT Q-9
# Package `line_detector`

## 9.1. Package information

Link to package on Github

### Essentials

**Author:** Hang Zhao

**Maintainer:** Mack

### Description

**TODO:** Add a description of package `line_detector` in `package.xml`.

This package is being replaced by the cleaned-up version, `line_detector2`. Do not write documentation here.

However, at the moment, all the launch files still call this one.

# Part R
# Packages - Indefinite navigation

TODO: to write

# Unit R-1
# AprilTags library

## 1.1. Package information

Link to package on Github

### Essentials

**Author:** Michael Kaess

**Author:** Hordur Johannson

**Maintainer:** Mitchell Wills

### Description

A catkin version of the C++ apriltags library

Detect April tags (2D bar codes) in images; reports unique ID of each detection, and optionally its position and orientation relative to a calibrated camera.

See examples/apriltags_demo.cpp for a simple example that detects April tags (see tags/pdf/tag36h11.pdf) in laptop or webcam images and marks any tags in the live image.

Ubuntu dependencies: sudo apt-get install subversion cmake libopencv-dev libeigen3-dev libv4l-dev

Mac dependencies: sudo port install pkgconfig opencv eigen3

Uses the pods build system in connection with cmake, see: http://sourceforge.net/p/pods/

Michael Kaess October 2012

---

AprilTags were developed by Professor Edwin Olson of the University of Michigan. His Java implementation is available on this web site: http://april.eecs.umich.edu.

Olson's Java code was ported to C++ and integrated into the Tekkotsu framework by Jeffrey Boyland and David Touretzky.

See this Tekkotsu wiki article for additional links and references: http://wiki.tekkotsu.org/index.php/AprilTags

---

This C++ code was further modified by Michael Kaess (kaess@mit.edu) and Hordur Johannson (hordurj@mit.edu) and the code has been released under the LGPL 2.1 license.

- converted to standalone library
- added stable homography recovery using OpenCV
- robust tag code table that does not require a terminating 0 (omission results in false positives by illegal codes being accepted)
- changed example tags to agree with Ed Olson's Java version and added all his other tag families
- added principal point as parameter as in original code - essential for homography
- added some debugging code (visualization using OpenCV to show intermediate detection steps)

- added fast approximation of arctan2 from Ed's original Java code
- using interpolation instead of homography in Quad: requires less homography computations and provides a small improvement in correct detections

todo: - significant speedup could be achieved by performing image operations using OpenCV (Gaussian filter, but also operations in TagDetector.cc) - replacing arctan2 by precomputed lookup table - converting matrix operations to Eigen (mostly for simplifying code, maybe some speedup)

UNIT R-2

# Package apriltags_ros

AprilTags for ROS.

build passing

+ Resource error

I will not embed remote files, such as https://api.travis-ci.org/RIVeR-Lab/apriltags_ros.png:

## 2.1. Package information

Link to package on Github

### Essentials

**Author:** Mitchell Wills

**Maintainer:** Mitchell Wills

### Description

A package that provides a ROS wrapper for the apriltags C++ package

# Unit R-3
# Package fsm

## 3.1. Package information

Link to package on Github

### Essentials

**Author:** Michael Misha Novitzky

**Maintainer:** Mack

### Description

The finite state machine coordinates the modes of the car. The fsm package consists of two nodes, namely `fsm_node` and `logic_gate_node`.

## 3.2. Node fsm_node

(Generated from configuration `fsm_node.easy_node.yaml`.)

**fsm_node**

This node handles the state transitions based on the defined state transition events. Below is a summary of the basic functionality of the `fsm_node`.

- Each state is a mode that the Duckiebot can be in
- Each state has corresponding state transitions triggered by events
- Each event is triggered by a certain value of a certain topic message
- In each state, certain nodes are active
- Each node affected by the state machine can be switched active/inactive by a `~/switch` topic

The current state is published to the `fsm_node/mode` topic. For each state, there is a list of nodes which should be active, which are switched by means of `node_name/switch` topics.

The FSM node publishes on many topics according to the configuration:

```
for node_name, topic_name in nodes.items():
    self.pub_dict[node_name] = rospy.Publisher(topic_name, BoolStamped, ...)
```

where `nodes.items()` is a list of all nodes affected by the FSM, and the `topic_name` is `node_name/switch`. The relevant nodes then subscribe to `~/switch`, and toggle their behaviour based on the value of the switch. Nodes can also subscribe to the `fsm_node/mode` topic if they need to change their behaviour based on the state. An example of how a node named `ExampleNode` can handle this is shown below:

```
class ExampleNode(object):
    def __init __(self):
    ...
    self.sub_switch = rospy.Subscriber("~switch",BoolStamped, self.cbSwitch, queue_size=1)
    self.sub_fsm_mode = rospy.Subscriber("fsm_node/mode",FSMState, self.cbMode,
queue_size=1)
    self.active = True
    self.mode = None

    def cbSwitch(self,switch_msg):
        self.active = switch_msg.data # True or False

    def cbMode(self,switch_msg):
        self.mode = switch_msg.state # String of current FSM state

    def someOtherFunc(self, msg):
        if not self.active:
            return
        # else normal functionality
        ...
        if self.mode == "LANE_FOLLOWING":
            ...
        if self.mode == "INTERSECTION_CONTROL":
            ...
```

## Parameters

**Parameter** `states`: `dict`; default value: ``

States are the modes that the system can be in. Each state has corresponding events (which trigger transitions to specific states), as well as a list of active nodes in the current state.

**Parameter** `nodes`: `dict`; default value: ``

These are the nodes which are affected by the FSM, and also define the `~/switch` topics to switch them between active and inactive.

**Parameter** `global_transitions`: `dict`; default value: ``

These are the state transition events (and corresponding topic) that can be triggered from all states.

**Parameter** `initial_state`: `str`; default value: `'LANE_FOLLOWING'`

This is the initial state that the FSM will be in upon launch of the node.

**Parameter** `events`: `dict`; default value: ``

These are the events and the corresponding topics (and message values) which trigger them, which allow for transitions between states.

## Subscriptions

No subscriptions defined.

## Publishers

**Publisher** `mode`: topic `~mode` (`FSMState`)

This topic gives the current state of the FSM, and can have values from a set of strings indicating the possible state names.

## 3.3. Node `logic_gate_node`

(Generated from configuration `logic_gate_node.easy_node.yaml`.)

**logic_gate_node**

This node handles AND and OR logic gates of events for state transitions. Below is a summary of the basic functionality of the `logic_gate_node`.

- Logic AND and OR gates can be defined
- For each gate, the input events (and their corresponding topics) are defined
- The `logic_gate_node` subscribes to all of these input event topics
- When an input topic is published, the `logic_gate_node` checks whether the AND or OR gate is satisfied
- If the gate is satisfied, the node publishes `True` on the `~/gate_name` topic, else it publishes `False`

The logic gate node publishes on many topics according to the configuration:

```
for gate_name, gate_dict in self.gates_dict.items():
    output_topic_name = gate_dict["output_topic"]
    self.pub_dict[gate_name] = rospy.Publisher(output_topic_name, BoolStamped, queue_size=1)
```

where `gate_dict.items()` is a dictionary of all gates, and `output_topic_name` is `~/gate_name`. The `fsm_node` then subscribes to `logic_gate_node/*`, where each `gate_name` corresponds to

a state transition event.

### Parameters

**Parameter** `events`: `dict`; default value: ``

These are all the events and corresponding topics (and trigger values) which are inputs to a logic gate event.

**Parameter** `gates`: `dict`; default value: ``

These are the logic gate events. Each gate has a gate_type (AND or OR), input events, and an output topic.

### Subscriptions

No subscriptions defined.

### Publishers

No publishers defined.

UNIT R-4

# Package `indefinite_navigation`

## 4.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Liam Paull

### Description

TODO: Add a description of package `indefinite_navigation` in `package.xml`.

# UNIT R-5
# Package `intersection_control`

## 5.1. Package information

Link to package on Github

### Essentials

**Author:** Michael Misha Novitzky

**Maintainer:** Mack

### Description

The intersection_control package implements a dead reckoning controller until we do something smarter.

UNIT R-6

# Package `navigation`

## 6.1. Package information

Link to package on Github

### Essentials

**Author:** igor

**Maintainer:** Mack

### Description

**TODO:** Add a description of package `navigation` in `package.xml`.

UNIT R-7

# Package `stop_line_filter`

## 7.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Liam Paull

### Description

**TODO:** Add a description of package `stop_line_filter` in `package.xml`.

# Part S
# Packages - Localization and planning

TODO: to write

# Unit S-1
# Package duckietown_description

## 1.1. Package information

Link to package on Github

### Essentials

**Author:** Teddy Ort
**Maintainer:** Mack

### Description

**TODO:** Add a description of package `duckietown_description` in `package.xml`.

UNIT S-2

# Package `localization`

## 2.1. Package information

Link to package on Github

### Essentials

**Author:** teddy

**Maintainer:** Mack

### Description

**TODO:** Add a description of package `localization` in `package.xml`.

# Part T
# Packages - Coordination

TODO: to write

UNIT T-1

# Package `led_detection`

## 1.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Andrea Censi

### Description

**TODO:** Add a description of package `led_detection` in `package.xml`.

**TODO:** add authors

## 1.2. LED detector

Pick your favourite duckiebot as the observer-bot. Refer to it as *robot name* for this step. If you are in good company, this can be tried on all the available duckiebots. First, activate the camera on the observer-bot:

```
$ roslaunch duckietown camera.launch veh:=robot name
```

In a separate terminal, fire up the LED detector and the custom GUI by running:

```
$ roslaunch led_detector LED_detector_with_gui.launch veh:=robot name
```

> **Note:** to operate without a GUI:

```
$ roslaunch led_detector LED_detector.launch veh:=robot name
```

The LED_detector_node will be launched on the robot, while LED_visualizer (a simple GUI) will be started on your laptop. Make sure the camera image from the observer-bot is visualized and updated in the visualizer (tip: check that your camera cap is off).

Hit on Detect and wait to trigger a detection. This will not have any effect if LED_detector_node is not running on the duckiebot (it is included in the above launch file). After the capture and processing phases, the outcome will look like:

The red numbers represent the frequencies directly inferred from the camera stream, while the selected detections with the associated signaling frequencies will be displayed in green. You can click on the squares to visualize the brightness signals and the Fourier amplitude spectra of the corresponding cells in the video stream. You can also click on the camera image to visualize the variance map.

## 1.3. Unit tests

To run the unit tests for the LED detector, you need to have the F23 rosbags on you hard disk. These bag files should be synced from [this dropbox link](https://www.dropbox.com/sh/5kx8qwgttu69fhr/AAASLpOVjV5r1xpzeW7xWZh_a?dl=0).

For the test to locate the bag files, you should have the `DUCKIETOWN_DATA` environment variable set, pointing to the location of you duckietown-data folder. This can be achieved by:

```
$ export DUCKIETOWN_DATA=local-path-to-duckietown-data-folder
```

All the available tests are specified in file `all_tests.yaml` in the scripts/ folder of the package led_detection in the duckietown ROS workspace. To run these, use the command:

```
$ rosrun led_detection unittests.py algorithm name-of-test
```

Currently, *algorithm* can be either 'baseline' or 'LEDDetector_plots' to also display the plot in the process.

To run all test with all algorithms, execute:

```
$ rosrun led_detection unittests.py '*' '*'
```

More in general:

```
$ rosrun led_detection unittests.py tests algorithms
```

where:

- *tests* is a comma separated list of algorithms. May use "`*`".
- *algorithms* is a comma separated list of algorithms. May use "`*`".

The default algorithm is called "`baseline`", and its tests are invoked using:

```
$ rosrun led_detection <script> '*' 'baseline'
```

UNIT T-2

# Package `led_emitter`

## 2.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Andrea Censi

### Description

**TODO:** description for `led_emitter` package.

## 2.2. In depth

The coordination team will use 3 signals: CAR_SIGNAL_A, CAR_SIGNAL_B, CAR_SIGNAL_C. To test the LED emitter with your joystick, run the following command:

```
$ roslaunch led_joy_mapper led_joy_with_led_emitter_test.launch veh:=robot name
```

This launches the joy controller, the mapper controller, and the led emitter nodes. You should not need to run anything external for this to work. Use the joystick buttons A, B and C to change your duckiebot's LED's blinking frequency.

Button A broadcasts signal CAR_SIGNAL_A (2.8hz), button B broadcasts signal CAR_SIGNAL_B (4.1hz), and button CAR_SIGNAL_C (Y on the controller) broadcasts signal C(5hz).The LB button will make the LEDs all white, the RB button will make some LEDs blue and some LEDs green, and the logitek button (middle button) will make the LEDs all red

Repeat this for each vehicle at the intersection that you wish to be blinking. Use previous command replacing *robot name* the names of the vehicles and try command different blinking patterns on different duckiebots.

(optional tests) For a grasp of the low level LED emitter, run:

```
$ roslaunch led_emitter led_emitter_node.launch veh:=robot name
```

You can then publish to the topic manually by running the following command in another screen on the duckiebot:

```
$ rostopic pub /robot name/led_emitter_node/change_to_state std_msgs/Float32
float-value
```

Where *float-value* is the desired blinking frequency, e.g. 1.0, .5, 3.0, etc. If you wish

to run the LED emitter test, run the following:

```
$ roslaunch led_emitter led_emitter_node_test.launch veh:=robot name
```

This will cycle through frequencies of 3.0hz, 3.5hz, and 4hz every 5 seconds. Once done, kill everything and make sure you have joystick control as described above.

UNIT T-3

# Package `led_interpreter`

## 3.1. Package information

Link to package on Github

### Essentials

**Author:** qlai

**Maintainer:** Mack

### Description

**TODO:** Add a description of package `led_interpreter` in `package.xml`.

UNIT T-4

# Package `led_joy_mapper`

## 4.1. Package information

Link to package on Github

### Essentials

**Author:** lapentab

**Maintainer:** Mack

### Description

**TODO:** Add a description of package `led_joy_mapper` in `package.xml`.

UNIT T-5

# Package `rgb_led`

## 5.1. Package information

Link to package on Github

### Essentials

**Author:** Dmitry Yershov

**Maintainer:** Mack

### Description

**TODO:** Add a description of package `rgb_led` in `package.xml`.

## 5.2. Demos

To test the traffic light:

```
$ rosrun rgb_led blink trafficlight4way
```

Fancy test:

```
$ rosrun rgb_led blink trafficlight4way
```

To do other tests:

```
$ rosrun rgb_led blink
```

# Unit T-6
# Package `traffic_light`

## 6.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Mack

### Description

**TODO:** Add a description of package `traffic_light` in `package.xml`.

# Part U
# Packages - Additional functionality

TODO: to write

# Unit U-1
# Package mdoap

## 1.1. Package information

Link to package on Github

### Essentials

**Author:** Catherine Liu
**Author:** Sam
**Author:** Ari
**Maintainer:** Mack

### Description

The mdoap package which handles object recognition and uses ground projection to estimate distances of objects for duckie safety. Also contains controllers for avoiding said obstacles

UNIT U-2

# Package `parallel_autonomy`

## 2.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Liam Paull

### Description

**TODO:** description for the parallel autonomy package

UNIT U-3

# Package `vehicle_detection`

## 3.1. Package information

Link to package on Github

### Essentials

**Author:** Andrea Tacchetti

**Maintainer:** Mack

### Description

**TODO:** Add a description of package `vehicle_detection` in `package.xml`.

# Part V
# Packages - Templates

These are templates.

UNIT V-1

# Package `pkg_name`

## 1.1. Package information

Link to package on Github

### Essentials

**Author:** Shih-Yuan Liu

**Maintainer:** Andrea Censi

### Description

The package `pkg_name` is a template for ROS packages.

For the tutorial, see Unit K-6 - Minimal ROS node - `pkg_name`.

## 1.2. How to use this template

This package is a template that contains code

UNIT V-2

# Package `rostest_example`

## 2.1. Package information

Link to package on Github

### Essentials

**Author:** Teddy Ort

**Maintainer:** Mack

### Description

**TODO:** Add a description of package `rostest_example` in `package.xml`.

# Part W
# Packages - Convenience

These packages are convenience packages that group together launch files and tests.

UNIT W-1

# Package duckie_rr_bridge

## 1.1. Package information

Link to package on Github

### Essentials

**Author:** greg

**Maintainer:** Mack

### Description

The duckie_rr_bridge package. Creates a Robot Raconteur Service to drive the Duckiebot.

# Unit W-2
# Package duckiebot_visualizer

## 2.1. Package information

Link to package on Github

### Essentials

**Author:** Shih-Yuan Liu
**Maintainer:** Mack

### Description

TODO: Add a description of package `duckiebot_visualizer` in `package.xml`.

## 2.2. Node duckiebot_visualizer

(Generated from configuration `duckiebot_visualizer.easy_node.yaml`.)

TODO: Missing node description in `duckiebot_visualizer.easy_node.yaml`.

### Parameters

**Parameter** `veh_name`: `str`; default value: `'megaman'`

TODO: Missing description for entry "`veh_name`".

### Subscriptions

**Subscription** `seg_list`: topic `~segment_list` (`SegmentList`)

TODO: Missing description for entry "`seg_list`".

### Publishers

**Publisher** `pub_seg_list`: topic `~segment_list_markers` (`MarkerArray`)

TODO: Missing description for entry "`pub_seg_list`".

UNIT W-3

# Package `duckietown_demos`

## 3.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Liam Paull

### Description

TODO: description of `duckietown_demos`

UNIT W-4

# Package duckietown_unit_test

## 4.1. Package information

Link to package on Github

### Essentials

**Maintainer:** Mack

### Description

The `duckietown_unit_test` meta package contains all the unit test launch files for Duckietown.

UNIT W-5

# Package `veh_coordinator`

## 5.1. Package information

Link to package on Github

### Essentials

**Author:** Michal Cap

**Maintainer:** Mack

### Description

The simple vehicle coordination package

I think is used to fake the vehicle coordination for the FSM.

# Part X
# Packages - Deep Learning

TODO: to write

UNIT X-1

# Last modified

- Wed, Feb 28: Unit N-20 - Explicit coordination: final report (ValentinaCavinato)
- Wed, Feb 28: Unit N-24 - PDD - Distributed Estimation (tanij)
- Wed, Feb 28: Unit N-25 - Distributed Estimation: intermediate report (tanij)
- Wed, Feb 28: Unit N-26 - Fleet Messaging: final report (tanij)
- Wed, Feb 28: Unit N-27 - Demo instructions Fleet Communications (tanij)
- Tue, Feb 27: Unit N-8 - The Controllers: preliminary report (tanij)
- Tue, Feb 27: Unit N-18 - Explicit Coordination: preliminary report (tanij)
- Tue, Feb 27: Unit N-19 - Explicit Coordination: intermediate Report (tanij)
- Tue, Feb 27: Unit N-21 - Implicit Coordination: preliminary report (tanij)
- Tue, Feb 27: Unit N-23 - Implicit Coordination: final report (tanij)
- Tue, Feb 27: Unit N-10 - The Controllers: final report (Jacopo Tani)
- Tue, Feb 27: Unit N-22 - Implicit Coordination: intermediate report (tanij)
- Tue, Feb 27: Unit F-10 - Follow Leader (Kornel Eggerschwiler)
- Sun, Feb 25: Unit N-29 - Fleet Planning: Preliminary Report (tanij)
- Sun, Feb 25: Unit N-30 - Fleet Planning: Intermediate Report (tanij)
- Sun, Feb 25: Unit N-31 - Fleet Planning: Final Report (tanij)
- Tue, Feb 20: Unit C-11 - Software setup and RC remote control (Florian Golemo)
- Mon, Feb 19: Unit N-13 - The Saviors: Final Report (Niklas Funk)
- Mon, Feb 19: Unit F-3 - Demo Saviors (Niklas Funk)
- Fri, Feb 16: Unit C-16 - Wheel calibration (Manfred Diaz)
- Tue, Feb 13: Unit C-10 - Networking aka the hardest part (Julien Kindle)
- Fri, Feb 09: Unit N-3 - Group name: final report (Jacopo Tani)
- Tue, Jan 23: Unit J-31 - How to install PyTorch on the Duckiebot (Nithin Vasisth)
- Fri, Jan 19: Unit C-17 - Camera calibration (Manfred Diaz)
- Thu, Jan 18: Unit F-8 - Explicit Coordination (ValentinaCavinato)
- Thu, Jan 18: Unit F-2 - Lane following (Simon Muntwiler)
- Mon, Jan 15: Unit F-7 - Implicit Coordination (Kornel Eggerschwiler)
- Sun, Jan 14: Unit Q-2 - Package `complete_image_pipeline` (Andrea Censi)
- Fri, Jan 05: Unit O-5 - Package `easy_logs` (Andrea Censi)
- Mon, Jan 01: Section 3.1 - Package information (sonja)
- Sun, Dec 31: Unit O-6 - Package `easy_node` (Andrea Censi)
- Sat, Dec 30: Unit B-2 - Basic Markduck guide (Andrea Censi)
- Sat, Dec 30: Unit B-2 - Basic Markduck guide (Andrea Censi)
- Sat, Dec 30: Unit D-2 - Duckietown Appearance Specification (Andrea Censi)
- Sat, Dec 30: Unit R-3 - Package `fsm` (sonja)
- Fri, Dec 29: Unit K-6 - Minimal ROS node - `pkg_name` (Theodore Koutros)
- Wed, Dec 20: Unit N-41 - PDD - Anti-Instagram (Jacopo Tani)
- Mon, Dec 18: Unit B-1 - Contributing to the documentation (Jacopo Tani)

```
This link is empty:

    <a class="number_name" href="#booktitle"></a>

It might be that the writer intended for this
link to point to something, but they got the syntax wrong.

    href = #booktitle

As a reminder, to refer to other parts of the document, use
the syntax "#ID", such as:

    See [](#fig:my-figure).

    See [](#section-name).
```

Sat, Dec 16: + syntax error (Andrea Censi)

- Sat, Dec 16: Unit F-5 - Indefinite Navigation (Jacopo Tani)
- Sat, Dec 16: Unit F-6 - Coordination (Jacopo Tani)
- Sat, Dec 16: Unit F-9 - Parallel Autonomy (Jacopo Tani)
- Mon, Dec 11: Unit N-16 - PDD - Parking (tanij)
- Mon, Dec 11: Unit N-17 - Parking: intermediate report (tanij)
- Mon, Dec 11: Unit N-39 - Visual Odometry Project (tanij)
- Mon, Dec 11: Unit N-40 - Deep Visual Odometry ROS Package (tanij)
- Sat, Dec 09: Unit N-33 - Transferred Lane following (Chip Schaff)
- Sat, Dec 09: Unit N-34 - Transfer Learning in Robotics (Chip Schaff)
- Thu, Dec 07: Unit N-9 - The Controllers: Intermediate Report (lapandic)
- Wed, Dec 06: Unit N-37 - PDD Neural Slam (lapandic)
- Wed, Dec 06: Unit J-33 - Movidius Neural Compute Stick Install (Michael Noukhovitch)
- Wed, Dec 06: Unit N-7 - System Identification: Intermediate Report (tanij)
- Wed, Dec 06: Unit N-12 - The Saviors: intermediate report (tanij)
- Wed, Dec 06: Unit N-35 - PDD - Supervised-learning (tanij)
- Wed, Dec 06: Unit N-36 - Supervised Learning: intermediate report (tanij)
- Wed, Dec 06: Unit F-4 - Intersection Navigation (Jacopo Tani)
- Tue, Dec 05: Unit N-15 - The Navigators: intermediate report (Theodore Koutros)
- Tue, Dec 05: Part X - Packages - Deep Learning (liampaull)
- Tue, Dec 05: Section 11.7 - Sep 28: some announcements (ValentinaCavinato)
- Fri, Dec 01: Unit C-7 - Reproducing the image (Elias Zoller)
- Tue, Nov 28: Unit C-1 - Duckiebot configurations (Andrea F. Daniele)
- Tue, Nov 28: Unit N-2 - Group name: intermediate report (Jacopo Tani)
- Tue, Nov 28: Unit C-2 - Acquiring the parts (`DB17-jwd`) (liampaull)
- Mon, Nov 27: Unit C-6 - Assembling the Duckiebot (`DB17-wjd` TTIC) (Andrea F. Daniele)
- Mon, Nov 27: Unit A-3 - Duckietown classes (Andrea F Daniele)
- Mon, Nov 27: Part C - Operation manual - Duckiebot (Andrea F. Daniele)
- Mon, Nov 27: Unit C-3 - Soldering boards (`DB17`) (Andrea F. Daniele)
- Mon, Nov 27: Unit C-4 - Preparing the power cable (`DB17`) (Andrea F. Daniele)
- Mon, Nov 27: Unit C-5 - Assembling the Duckiebot (`DB17-jwd`) (Andrea F. Daniele)
- Mon, Nov 27: Part D - Operation manual - Duckietown (Andrea F. Daniele)
- Mon, Nov 27: Part E - Duckiebot - `DB17-1c` configurations (Andrea F. Daniele)
- Mon, Nov 27: Unit E-1 - Acquiring the parts (`DB17-1c`) (Andrea F. Daniele)
- Mon, Nov 27: Unit E-2 - Soldering boards (`DB17-1`) (Andrea F. Daniele)
- Mon, Nov 27: Unit E-3 - Assembling the Duckiebot (`DB17-1c`) (Andrea F. Daniele)

- Mon, Nov 27: Unit E-5 - `DB17-1` setup (Andrea F. Daniele)
- Fri, Nov 24: Unit J-32 - How to install Caffe and Tensorflow on the Duckiebot (Nick Wang)
- Thu, Nov 23: Unit N-1 - Group name: preliminary design document (Andrea Censi)
- Thu, Nov 23: Unit N-4 - The Heroes quests (Andrea Censi)
- Thu, Nov 23: Unit N-5 - PDD - Smart City (Andrea Censi)
- Thu, Nov 23: Unit N-11 - PDD - Saviors (Andrea Censi)
- Thu, Nov 23: Unit N-14 - PDD - Navigators (Andrea Censi)
- Thu, Nov 23: Unit N-38 - PDD - Visual Odometry (Andrea Censi)
- Thu, Nov 23: Part F - Operation Manual - demos (Andrea Censi)
- Thu, Nov 23: Unit F-1 - Demo template (Andrea Censi)
- Thu, Nov 23: Unit N-6 - PDD - System Identification (Andrea Censi)
- Thu, Nov 23: Unit N-42 - Transfer: preliminary design document (Andrea Censi)
- Thu, Nov 23: Unit K-14 - Road Release Process (liampaull)
- Thu, Nov 23: Unit M-28 - System Architecture (sonja)
- Wed, Nov 22: Part N - Fall 2017 projects (lapandic)
- Wed, Nov 22: Unit N-28 - Single SLAM Project (Casper Neo)
- Mon, Nov 20: Unit M-24 - Checkoff: Navigation (saikrishnagv_1996)
- Mon, Nov 20: Unit M-13 - Chicago branch diary (Matthew Walter)
- Fri, Nov 17: Unit M-6 - Logistics for NCTU branch (Jacopo Tani)
- Wed, Nov 15: Unit M-14 - NCTU branch diary (championway)
- Tue, Nov 14: Unit D-3 - Signage (Andrea F Daniele)
- Mon, Nov 13: Unit E-4 - Bumper Assembly (Shiying99)
- Mon, Nov 13: Unit M-12 - Montréal branch diary (liampaull)
- Mon, Nov 13: Unit I-11 - Exercise: Lane Filtering - Extended Kalman Filter (Philippe Laferrière)
- Fri, Nov 10: Unit Q-8 - Package `line_detector2` (Philippe Laferrière)
- Fri, Nov 10: Unit M-25 - Homework: Lane Filtering (Matthew Walter)

Page left blank