Foundations of Robotics

Part I: Models and Mechanisms

Dr CW Fox

November 26, 2021

About this module

This module introduces the mathematical and practical foundations of robotics which underlie most advanced systems. Mathematical foundations include relevant engineering mathematics, dynamics and kinematics, probability and elementary control, while practice includes understanding sensors and actuators used in real-world systems. Indicative topics of study will include (but are not limited to): sensors, actuators, dynamics, kinematics, introduction to control, introduction to trajectory planning, probability theory for robotics.

Notes on MSc learning

MSc learning is different from undergraduate. You are expected to self-direct your own reading and learning based on the topics introduced in class, including seeking out interesting textbooks, exercises and examples, and research papers. You need to do this in order to learn enough details to earn Distinction. The class is intended only as a jumping-off point for these activities. The class introduces core concepts and provides some starting points for your reading, computer and maths learning activities but beyond this it is up to **you** to go to the library or internet and discover what interests you personally. Exam questions will be quite open ended and designed to enable you to show off what you have learned in these adventures; if you get deeply into something beyond the scope of the module and exam then you will likely have the chance to show it off during your project assessment later. You are unlikely to get the most of of the module if you only study material that is given directly to you in class. If you get an MSc level job in robotics then a large part of your working life will be likely be spent doing similar self-directed finding things out, and gaining the ability to do this well and lead the intellectual direction of a team is why many companies want to hire MScs.

Learning outcomes

- **LO1** Critically appraise the benefits and costs of commonly used sensors and actuators for use in real-world systems
- **LO2** Understand and analyze dynamic physical robotic systems
- **LO3** Understand the key features of basic hardware and control software for a robotic system

Assessment

Assessment for the whole module is by one open book, take-home, time constrained assesment (TCA) during Semester A exam period. See University exam period timetable for date.

Assessment for this Part I of the module is by the first of the three questions on the written exam:

Question 1 (50%) covers Criterion 1 and Criterion 2 of the Criterion Reference Grid (CRG) below. You should refer to the CRG when writing your answer to ensure that you are meeting all of the requirements to get the best mark. The CRG tells you what you need to write in order to get each grade. The question will ask you to write about the practical work you have done in the module, and to link this to the theoretic concepts that you have learned to demonstrate your knowledge and skill.

It is expected that you will work either in groups, or by yourself if you prefer, on one of the mini-projects listed in the last chapter of these notes. The question will list these specific mini projects and ask you to write about the theory and practice of exactly one of them. You may work in groups on the practial work during the module and you are strongly encouraged to do so, in particular to share your existing skills as much as possible.

You are strongly encouraged to write an ICRA conference publication-style paper together with your group, so that you have a single, group-editable, and definitive version of the design for the project. You can use overleaf.com or a similar site to write together in latex and use the ICRA template. These papers are not assessed but may be submitted for publication.

However the TCAs are written and assessed individually and you must not communicate during the TCA or submit text that is not written yourself unless it is credited and referenced to its author. Submissions will be automatically scanned for plagurism including similarity to others students' submissions. TCAs are written from scratch on the assesment day. They may take inspiration from your conference paper drafts but must not cut and paste text from it.

Neither publications nor MSc level assessments care about your project management methods or group working skills. These documents should not include undergraduate-style sections such as Gantt charts, project plans, or group reflections. They should only have Abstract, Introduction, Methods, Results, Conclusion.

The other 50% of the exam will cover Part II of the module. It will be assessed using two further questions worth 25% each, and covering Criterion 3 and Criterion 4 from the CRG.

Lincoln School of Computer Science

CMP9765M-1819
Foundations of Robotics Assessment 1 (100% one TCA)

	Learning Outcome	Criterion	Pass (50)	Merit (60)	Distinction (70)
LO1	Critically appraise the benefits and costs of commonly used sensors and actuators for use in real-world systems	Criterion 1: mechatronics practice (25%)	Demonstrate basic understanding of key concepts in real-world sensors and actuators	Demonstrate basic appraisal of key concepts in real-world sensors and actuators with some analysis of their uses and limitation	Demonstrate critique of key concepts in real-world sensors and actuators with detailed critique of their uses and limitations
LO2	Understand and analyse dynamic physical robotic systems	Criterion 2: mechatronics theory (25%)	Demonstrate basic understanding of key concepts in mechanical and electromagnetic systems	Demonstrate working understanding of key concepts in dynamic physical robotic system, show evidence of ability to apply them	Demonstrate detailed understanding of key concepts in mechanical and electromagnetic systems with accurate quantative models
LO2	Understand and analyse dynamic physical robotic systems	Criterion 3: control and planning theory (25%)	Demonstrate basic understanding of key concepts in control and planning	Demonstrate basic understanding of key concepts in control and planning and evidence of use in analysis	Demonstrate detailed understanding of control and planning and evidence of use in accurate and detailed analysis
LO3	Understand the key features of basic hardware and control software for a robotic system	Criterion 4: control and planning practice (25%)	Demonstrate basic understanding of key concepts in robotic hardware and software	Demonstrate working understanding of key concepts in robotic hardware and software	Demonstrate detailed understanding of key concepts in robotic hardware and software

Homework allocation

4 hours per week model.

Timetable

See timetable.lincoln.ac.uk

Workshops

Each lecture has a set of workshop questions at the end, for use in the workshops or at home. It is expected that students will come from a wide mixture of backgrounds, and as this is a Foundations modules, that every topic will have some student who already has some experience in it. These students are asked to assist others so that existing knowledge in the class can be shared around. Workshops can used to explore these questions and/or for mini project work.

Errata

Like most lecture notes these probably contain some errors and typos, please let me know if you find any so they can be corrected. (Reporting errors is a good way to look clever!)

Module part II overview

Part II of the module will spend the second half of the semester and will cover topics such as:

- Robot motion
- Odometry
- PID controller details
- Intro to localisation
- State space
- Kinematics
- Planning
- Reinforcement learning
- (Reactive behaviours)
- Part II teaching may be based on this textbook:
  - www.researchgate.net/publication/322177050_Elements_of_Robotics

Acknowledgments

Thanks to Chris Waltham, Rob Lloyd, Andy Ham, Ed Gummow, Pete Williams and Lincoln Hackspace for mechatronics ideas. Lecture 3 was transcribed and formatted by Danielle Scullion.

About your lecturer

Dr Charles Fox is a Senior Lecturer in Robotics and Automation at the University of Lincoln, UK. He researches agricultural robotics, autonomous vehicles, pattern recognition and data. Current projects include the ARWAC weeding robot, LASERBOOM boom sprayer optimisation, INTERACT human interactions with self-driving cars using game theory, and data science analytics with geographic data. Dr Fox obtained a first class MA in Computer Science at the University of Cambridge, MSc with Distinction in Cognitive Science at the University of Edinburgh, and his DPhil in Engineering from the Robotics Research Group at the University of Oxford, nominated for the BCS Distinguished Dissertation award by examiners Prof Sir Mike Brady FRS and Prof Kath Laskey. He is a fellow of the Higher Education Academy. He worked as a data-driven high-frequency hedge fund trader in London then as a researcher at the Sheffield Center for Robotics and as an autonomous vehicle fellow at the Institute for Transport Studies, University of Leeds. Project partners have included the BBC, NHS, BMW, Volvo, Nissan, Highways England, Midlands Connect, and various agricultural engineering SMEs. He is the author of the Springer textbook "Data Science for Transport" and was the Programme Chair of the UK's national robotics conference, TAROS2021.

For publications see: https://scholar.google.co.uk/citations?user=dQ7RURYAAAAJ&hl=en&oi=sra

For biography see https://uk.linkedin.com/in/charles-fox-4714746 .

Contents

1 Robotics as physics and perception **12**

1.1 Scope of Robotics . . . 12
1.2 Types of "world" . . . 13
1.3 Physical theories in Computation . . . 14
1.4 Grounding and simulation . . . 16
1.5 Computing with Time . . . 17
1.6 Required mathematics . . . 18
1.6.1 Scalar numbers . . . 18
1.6.2 Vectors . . . 19
1.6.3 Matrices . . . 19
1.6.4 Rotation and translation matrices . . . 20
1.6.5 Change of basis . . . 21
1.6.6 Cross product . . . 21
1.6.7 Tensors . . . 21
1.6.8 Analysis . . . 22
1.6.9 Complex numbers and Quaternions . . . 23
1.6.10 Probability and causality . . . 24
1.7 Further reading . . . 25
1.8 Exercises . . . 25

2 Mechanics for Robotics **27**

2.1 Newtonian mechanics . . . 27
2.1.1 Ontology . . . 27
2.1.2 Vectors as coordinates . . . 28
2.1.3 Laws . . . 29
2.1.4 Units . . . 29
2.2 Folk axioms for particles . . . 30
2.2.1 Agent forces . . . 30
2.2.2 Gravity force . . . 30
2.2.3 Spring linkage forces . . . 30
2.2.4 Particle collisions . . . 31
2.3 Finite Element Methods (FEM) . . . 31
2.4 Folk axioms for rigid bodies . . . 32
2.4.1 Collision . . . 32
2.4.2 Rotation axioms . . . 33
2.4.3 Rigid bodies folk axioms summary . . . 34
2.4.4 Friction . . . 34

2.4.5 Rotary springs . . . 35
2.4.6 Joint constraint forces . . . 35
2.5 Derived quantities from Newton . . . 36
2.5.1 Momentum . . . 36
2.5.2 Energy . . . 36
2.6 Ontology in ODE . . . 37
2.7 Physics theories as AI . . . 38
2.8 Exercises . . . 40
2.9 Further reading . . . 41
2.9.1 Essential . . . 41
2.9.2 Advanced . . . 41

3 Computation for Robotics 42
3.1 Computer architecture overview . . . 42
3.2 Embedded Systems for robotics . . . 43
3.2.1 Typical features of embedded Systems . . . 44
3.3 Embedded systems components . . . 46
3.3.1 Micro-controllers . . . 46
3.3.1.1 CPU . . . 47
3.3.1.2 Harvard architecture memory . . . 47
3.3.1.3 Timers and counters . . . 47
3.3.1.4 IO ports . . . 48
3.3.2 Analog-digital (A/D) conversion . . . 49
3.3.3 IO protocols . . . 50
3.3.3.1 Serial port (RS232) . . . 50
3.3.3.2 I2C bus . . . 52
3.3.3.3 Controller Area Network (CAN) bus . . . 52
3.4 Embedded systems examples . . . 53
3.4.1 CPU-like systems . . . 54
3.4.1.1 Arduino . . . 54
3.4.1.2 PIC Microcontrollers . . . 58
3.4.1.3 RISCV microcontrollers . . . 58
3.4.1.4 Digital Signal Processors (DSPs) . . . 59
3.4.1.5 Raspberry Pi . . . 60
3.4.2 Field programmable gate array (FPGA) . . . 61
3.5 Summary . . . 63
3.6 Exercises . . . 64
3.7 Further reading . . . 65
3.7.1 Required . . . 65

4 Electromagnetism for robotics 66
4.1 Folk electromagnetism . . . 66
4.1.1 Folk electric force . . . 66
4.1.1.1 Charge Ontology . . . 66

4.1.1.2 Charge force . . . 67
4.1.1.3 Application: undirected path following . . . 67
4.1.2 Folk magnetic force . . . 67
4.1.2.1 Application: directed path following . . . 67
4.2 Maxwell's equations . . . 68
4.3 Electronics . . . 70
4.3.1 Ontology . . . 70
4.3.2 Water analogy . . . 70
4.3.3 Simple analog machines . . . 72
4.4 AC circuits . . . 73
4.5 Electromagnetic Actuation . . . 74
4.5.1 Types of Motors . . . 74
4.5.1.1 DC Brushed . . . 74
4.5.1.2 DC Brushless . . . 75
4.5.1.3 AC motors . . . 76
4.5.1.4 Gear motors . . . 76
4.5.1.5 Encoder motors . . . 77
4.5.1.6 Hub motors . . . 77
4.5.1.7 Linear actuators . . . 77
4.5.1.8 Hydraulic actuators . . . 78
4.5.1.9 Pneumatic actuators . . . 78
4.5.2 Drivers . . . 78
4.5.3 Controllers . . . 80
4.5.4 Stepper controller . . . 81
4.5.5 Arduino control . . . 81
4.5.6 Motor Interference . . . 82
4.6 Electromagnetic Communication . . . 82
4.7 Remaining theory limitations . . . 85
4.8 Exercises . . . 86
4.9 Further reading . . . 86
4.9.1 Required . . . 86
4.9.2 Recommended . . . 86

5 Sensors **87**
5.1 Basic sensors . . . 87
5.1.1 Switches . . . 87
5.1.2 Position . . . 88
5.1.3 Light . . . 88
5.1.4 Pressure . . . 89
5.2 Odometry . . . 90
5.3 Touch sensors . . . 91
5.4 Microphones . . . 92
5.5 Cameras . . . 93
5.5.1 Sensors . . . 93

5.5.2 Colours . . . 93
5.5.3 Image formation . . . 96
5.6 Depth sensors . . . 98
5.6.1 Lidar . . . 98
5.6.2 Depth from light patterns . . . 99
5.6.3 Stereo cameras . . . 100
5.6.4 Acoustic ranging (sonar) . . . 100
5.6.5 Radar . . . 100
5.7 Location sensors . . . 101
5.7.1 Global Navigation Satellite System (GNSS) . . . 101
5.7.2 Inertial Measurement Units (IMU) . . . 104
5.7.3 Real Time Localisation Systems (RTLS) . . . 105
5.8 Chemical sensors . . . 105
5.8.1 Mouth-like sensors . . . 105
5.8.1.1 Moisture sensors . . . 106
5.8.1.2 NPK fertilizer sensors . . . 106
5.8.1.3 DNA sensors . . . 106
5.8.2 Nose-like sensors . . . 107
5.9 Exercises . . . 108
5.10 Further reading . . . 108
5.10.1 Recommended . . . 108

6 Building robots 109
6.1 Open Source Hardware (OSH) . . . 109
6.2 Tools . . . 111
6.2.1 3d printing . . . 111
6.2.2 FreeCAD . . . 112
6.2.3 Power supply and measurement . . . 113
6.2.4 Measurement . . . 114
6.3 Materials . . . 114
6.3.1 Aluminum extrusion profiles . . . 114
6.3.1.1 Plastics . . . 115
6.3.2 Steel . . . 116
6.3.3 Wood . . . 116
6.3.4 Enclosures and IP ratings . . . 116
6.4 Connectors . . . 117
6.4.1 Nuts and bolts . . . 117
6.4.2 Mounting motors . . . 117
6.4.3 Connectors and soldering . . . 118
6.4.4 Welding . . . 120
6.4.5 Lego . . . 120
6.4.6 Last resort attachment . . . 120
6.5 Physical circuits . . . 120
6.6 Links . . . 124

6.7 Exercises . . . 125
6.8 Further reading . . . 125

7 Alternative physics models **126**
7.1 Configuration space . . . 126
7.2 Lagrangian mechanics . . . 127
7.2.1 Euler-Lagrange equations . . . 127
7.2.1.1 1-dimensional case . . . 127
7.2.1.2 N-dimensional case . . . 129
7.2.2 Principle of least action . . . 130
7.2.3 Lagrangians in configuration space . . . 131
7.2.3.1 Rotating reference frame . . . 131
7.2.3.2 Ramp and string . . . 133
7.2.3.3 Robot arm . . . 134
7.2.4 From Lagrangians to robot path planning . . . 135
7.3 Naïve Physics as AI . . . 135
7.4 Exercises . . . 137
7.5 Further reading . . . 137
7.5.1 Recommended . . . 137

8 Mini projects **139**
8.1 Robot radio: communication and localisation . . . 139
8.2 Physics simulation: the joy of friction . . . 139
8.3 Motor control: Build a physical robot . . . 140
8.4 Robot music: Automated guitar . . . 140

1 Robotics as physics and perception

Autonomous robots are physical agents which use **computation** to perform **perception** and **reasoning** to **act** in the **physical world**. Traditionally, the study of the physical world is Physics, the study of acting on it is Engineering, the study of reasoning is AI and the study of computation is Computer Science. Autonomous Robotics is an inter-disciplinary study which combines aspects of each of these.

This module gives an introductory review of each of these areas, as a foundation for studying Robotics as a whole. It is likely that students are from a mixture of backgrounds including specialisms in some of them. As such, some areas will be known already to some students. The module provides an opportunity for these students to spend time getting up to speed in the other areas. It would be very helpful if students could help each other to get up to speed in their own specialisms and learn from each other. This is a masters-level module so where we review "basic" material from other fields, such as **"high school**" physics, we will try to explore it in a **more critical** way than you may have previously seen. Even "high school" physics contains a plethora of assumptions and potential self-contradictions which can be challenged not only by modern physics but also by philosophical and computational needs in Robotics. So please try to engage in these debates even in areas which you think you already understand.

1.1 Scope of Robotics

Autonomous robotics differs from, but is related to:

Mechatronics. When we discuss Robotics we mean "Autonomous Robotics" which extend mechatronic systems with significant intelligent behavior. We need to understand basic mechatronics because our systems are build on it. But we are not professional mechatronics engineers and will typically work as part of a team which includes them. We need to know enough about Mechatronics to be able to collaborate at this level. Mechatronics means the fusion of Mechanical Engineering and Electronic Engineering, applied to the design of products and systems. It covers non-robotic product design (e.g. design of video cassette players, ABS braking systems, production line systems) which incorporate both motion and electronics, as well as robotics hardware. (Due to digitization, it is less common to find such consumer products now than in the days of video and cassette players which used to be strongly associated with the field, especially in Japan.) When Mechatronics covers robotics, it will typically work on designs such as robot arms and mobile vehicles, at the level of physical design and motor control, presenting an interface for Automation programmers to use.

AI. In current non-academic contexts it is common to use "robot" to describe any kind of software AI automation, including non-physical "robots" such as senders of email spam

or automated text processing systems for sorting CVs, insurance claims, or other company data ("robotic process automation"). Current media hype around "robots taking all the jobs" typically includes much of this in discussions of replacing office workers processing forms and data. However we do not consider these to be "robotics" in the academic sense, because there is no physical component. Autonomous Robotics usually includes a large AI software component, but requires it to be interfaced with the physical world. Classical (1980s) academic AI focused on logical representations of the world. While this may yet come back into fashion, current academic AI research is dominated by its subfields *Machine learning* and *Machine Vision* which take an approach based on continuous mathematics and (Bayesian) statistics rather than logic.

Roughly then, Robotics is Mechatronics plus AI. A Robotics engineer or researcher may work across both areas or specialize in one as part of a team that covers both. The relationships between IT, AI, Mechatronics and Robotics are shown below,

	no agency	*has agency*
no physical world	Information Technology	AI
has physical world	Mechatronics	(Autonomous) Robotics

1.2 Types of "world"

Autonomous robots are physical agents which use computation to performing reasoning to act in the physical world. But as autonomous robots deals with perception during reasoning as well as with physics, we will also encounter other types of "world" and it is important to be clear about the concept of the physical and these other worlds. They are well known in AI, Psychology and Philosophy but less to Engineers and Physicists.

Our human **perceptual world** is experienced directly as containing everyday macroscopic objects such as chairs, tables and cups, all extended (occupying space) in three dimensional space and at a single point in time. As humans, we experience ourselves reasoning about this world, making actions in it, and observing their effects. Some philosophers (idealists) try to argue that this is the only world that we know to exist. (There is also a philosophical distinction between a "perceptual world" and a "**phenomenal world**", where the former is a functional definition which might for example be implemented by a non-conscious machine, and the latter is a perceptual world that is being perceived by a conscious subject.)

Most people would however agree that another kind of "real" world exists independently of their perception of it, which is linked somehow to their perceptual world and also in the same way to the perceptual worlds of everyone else. In principal, we can never know if (or perhaps even define what it would mean for) such a **noumenal world** exists. Our percepts would be exactly the same if a noumenal world existed and gave rise to them and was made of superstrings or evil demons or the Mind of God or a computer simulation. (Famous thought experiments demonstrating this include Plato's allegory of the Cave; Descartes's evil demon, Putman's Brain in a Vat, and most recently The Matrix movies.)

The best we can do instead of knowing the noumenal world is to construct **physical world** models which assume that such a world exists, that it is made of certain things,

and that those things behave in certain ways. Physical worlds here are *theories* or *models* which describe and predict our perceptual worlds with varying degrees of success during different tasks. When we say that a robots is situated in, and acts in, the physical world, we mean that it appears to be so in such models.

We will often construct **simulation worlds** using computers, which are executable implementations of physical theories. For example, many video games are based in simulation worlds, and simulation worlds are used for testing robotics systems.

When we construct autonomous robots "reasoning to act", this usually requires them to have their own representations of the physical world. Such representations may or may not be designed to be similar to our own human representations. We can call these **robot perceptual worlds**.

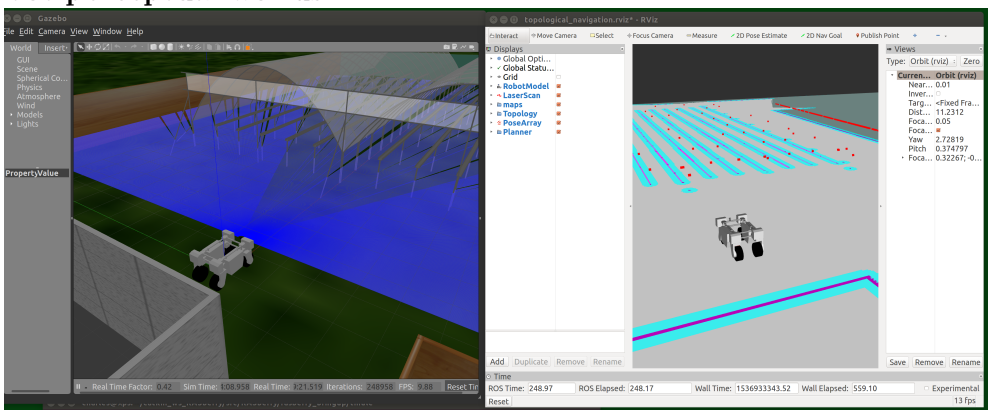


Figure: Simulated and perceptual worlds of an agricultural robot in Gazebo and rviz.

To perform reasoning, humans and robots will often make use of **counter-factual worlds** which represent possible, usually future, states of their perceptual or physical worlds, if certain things were different from its present actual state. For example, a possible future state assuming that the agent has performed a sequence of actions. A counter-factual perceptual world might be obtained using logical reasoning from the present actual state, while a physical counter-factual world might be obtained by the agent running its own simulation world.

1.3 Physical theories in Computation

When Roboticists and AI researchers study physics, it is usually for the purpose of simulation, machine perception and/or automated reasoning about the world. These are different goals from those of Physicists, Engineers, and other users of physical theories, so we have our own view of what a physical theory is and how it can be represented and used for our purposes.

No-one knows, or can ever know, what the noumenal world is "really" made of, of even if such a thing exists. Rather, we experience perceptions via our senses, and we invent theories which try to predict what will happen if we observe or do certain things. Like programming languages, these theories may be though of as having two parts, **ontology** and **laws**. Ontology refers to axiomatic beliefs about what exists (in the

theory). Laws refer to how the things that exist (in the theory) behave. Ontology and laws correspond to the data structures and algorithms used in computations on the theory. Many professional physicists still operate in an outdated philosophical view of the world dating from 1920s **logical positivism**, a failed philosophical movement which like many before it aimed to banish ontology from science and deal only with observed data. Such an opinion is especially common in quantum mechanics but only because we don't possess any clear ontology to think about QM in any other way (there are a few possible ontologies but there are deeply weird so students prefer to "shut up and calculate"). However, AI and Robotics must take a very different view because we need ontologies both to build and understand physics simulations of the world in our programming languages – which require the simulation to be expressed in terms of entities such as classes and objects; but also because building AI systems to reason about them requires them to appear in the conceptual scheme of an AI system, such as logical statements or database schemas. As such, teaching and learning physics for AI will have more emphasis on ontology than it does in pure Physics.

Pragmatism. For Robotics and Engineering, the "truth" of a physical theory is generally a less useful concept than it's usefulness for helping to perform some task of interest. We will use different theories in different tasks. In addition to choosing a physical theory for our use, it is then often possible to express the same theory using different ontologies and laws in ways that make the same predictions of what will happen in some, most, or all cases. Different forms have different effects on human programmers. They may make computation simpler or harder. They may make problems simpler or harder to think about – both for human programmers and for AI systems.

The ontologies of some theories and forms may include concepts of **causation** and **action** (or more philosophically, "free will") which are problematic in Physics but are required for robotics systems to make interactive interventions in the world. Such theories allows us to think as if "the robot's action a at time will change the state of the world at $t+1$ but cannot change the state at $t-1$". It is important to note that many theories of physics do not include these concepts. For example, General Relativity views the world as a fixed, predetermined, and unchanging region of space-time with no notion of agents making decisions or actions. Less well known is that classical Newtonian physics can be written in forms quite similar to this. Whether we include decisions, action and causation in a theory might thus depend on choices in our ontology rather than in the laws, which will affect how we view the ability of robots to make effects in the world.

From an AI perspective, it is notable than any attempt to try to properly define axioms for physics theories is very hard, and almost certain to lead to **contradictions** or other "blowing up" of systems such as via quantities going to zero. We would like to define physics this way as we do mathematics and programming languages. But Physics generally takes a weaker view of inconsistency than formal logic and symbolic AI, and part of the "art" of being a Physicist is to know where a theory is likely to "work" and when to use a different theory. Fully **formalizing** these ideas remains a major unsolved research challenge to AI itself.

Physicists and engineers tend to have more interest in "solving" mechanics problems by finding nice **analytical solutions** – maths equations describing them over time – while

in computing we tend to just assume that such solutions are unlikely to exist for real problems and go straight to **computational simulation** "solutions". We are usually happier to use discretized approximations than continuous calculus. We write less math but lots of code.

1.4 Grounding and simulation

AI in particular has developed formal methods for representing theories as symbolic axioms and rules, in systems such as Metamath and Isabelle. If we wanted to get deeply into this then we could express both ontologies and laws in them as completely formal computational structures.

Less formally, but still rigorous enough to provide executable simulations of physics, we have software implementations of **physics engines**. Rather than stating axioms and laws directly, these provide software APIs to construct worlds, take actions in them, and simulate the effects over time. Internally, they might, but usually don't, represent axioms and laws; but usually these are more implicit in their programmers code. They do however still require a concept of ontology shared with their users, as their APIs will be based on objects and classes from such ontologies.

One view of AI, popular in robotics, is that physics engines or preferably the actual noumenal world, are a better foundation than pure symbolic logic, because they ensure all our work is **grounded**. We might often use a physics engine simulation not just to test an AI system (to save time and money on real world experiments) but also inside the "brain" of the robot itself, when inside a real environment. The physical simulation becomes the robot's own model of the world and is used to predict what will happen in different possible futures.

We see a difference between logical statements such as,

"The block is on the table" or *on(block1, table1)*

and the "grounded" ontology used in a simulation such as,

Block(location=(x=5.0, y=6.7, z=2.3; shape=cuboid(height=1.0, width=2.0, depth=5.0) ; Table(location=(x=5.0, y=3.4, z=2.3; shape=cuboid(height=1.0, width=2.0, depth=5.0)

The first statement is not a complete description of the world. Rather it is a constraint on the set of all possible world which narrows them down to a smaller, but still multiple, set of possible worlds. It acts like a "sense" in ontology, picking out a block and a table but without fully specifying what they are. Forms such as this were used in "good old fashioned AI" in the 1960-80s. The general failure of such systems to find useful applications in the real world was attributed in part to the **symbol grounding problem**, i.e. that the symbols did not refer fully to the real world.

The second form, being the state of an executable simulation, is (arguably) a complete state of the world with no ambiguity. It acts as a referent in ontology being the precise and exact entity under discussion with no ambiguity. Classical AI worked with symbolic senses, making deductions and inferences about them. Simulation-based AI and robotics usually work with referents. (Note that we could use logical statements to describe referents as well as senses. "logical AI" is thus a misnomer to describe what we should

call something like "sense-based AI" vs "referent-based AI".)

1.5 Computing with Time

There are two main ways to think about and compute with time in robotics.

Presentism considers that only the present moment "really" exists. The past is gone and only memories of it remain. The future has not yet happened and only predictions and plans about it exist in the present. Computationally: this view is reflected in both logical and simulation based systems which represent the state of the word at a time t, then act either on the physical world in real time, or on counter-factual perceptual or physical worlds, to step forward through time, usually in discrete time steps Δt . Each representation of the world is for a single moment in time.

Eternalism is the view that time is "just another dimension", so all of "space-time" exists "eternally" in some sense. Computationally, this view is reflected in temporal data structures which represent states of the world across time. These can include lists, graphs, and databases. Mathematically, this can be a more useful view than Presentism when making plans for the future. For example we might consider the future path of a robot moving from A to B as a line in space-time, then perform optimization over possible lines to find the best one. While is is possible to approximate this by say, tree-searching discrete time simulation states for possible action sequences, only the space-time view can make use of continuous mathematics to find perfectly optimal solutions.

Neither view necessarily commits us to a belief in predetermination about the future. Presentism more naturally fits with ontologies that include action selection. But in eternalism we can just as well allow action selection if we assume that our agent will *choose* the best action at each moment and that optimizations over space-time provide a way to perform this decision making.

When modeling objects in the world, two similar options are well-known.

Perdurantism is the view that an object has distinct temporal parts which together comprise it. A table is made of a "table today" plus a "table tomorrow" and so on, in the same way that a song is make of a verse plus a chorus.

Endurantism is the opposing view that an object is singularly and wholly present at each moment in time. The same table is present today and tomorrow.

In computation this distinction is again manifested in data representation – do we choose to have the same data structure representing the table today as the table tomorrow, or should they be two separate data structures.

Consider also the interaction between time and types of worlds. A robot might capture some raw data from the physical world in January, and perceive it in its perceptual world in terms of the models it knows about in January. In December, it may replay either the captured data or its old perceptions to create counter-factual worlds. These do not always give the same results. The robot may have learned new models and ways of interpreting data during the year. A human example: suppose as a **teenager you wore a web-cam** on your head which recorded your whole life. Replaying your memories of what happened (during teenage angst events of epic proportions) would be very different

to watching and reinterpreting the video as an adult (where these events would mostly seem ridiculous). (Proust's novel, Remembrance of Things Past, is a very long study of this effect in literature.)

1.6 Required mathematics

To make physical theories we need to represent space and time. For the kinds of physical theories used in robotics (folk and classical mechanics), space and time are best represented by continuous vector mathematics.

The following is not intended to teach you these concepts. Rather, it is intended to list the concepts which are pre-requisite from undergradate level study, so that you can check that you have them and go and learn about them in your own time if you are missing anything. (This usually needs to be done by Computer Scientists, while the Engineers are similarly catching up on undergraduate level programming.)

1.6.1 Scalar numbers

Natural numbers are the numbers 0,1,2,3,4... . Formally, they can be defined by an inductive axiom scheme. First, 0 is a natural number. Second, if n is a number, then $S(n)$ ("successor of n") is a natural number. Natural numbers can be represented and used for computation by tallies or by encoings such as binary codes.

Integers are the numbers $... -3, 2, -1, 0, 1, 2, 3...$. They can be defined as pairing natural numbers with signs, where +0=-0.

Rationals are defined as pairs of integers a/b with $b \neq 0$. Examples include 1/2, -3/4, 50/2, -150/2, 0/2. Many rationals are equivalent to one another such as 4/2 and 2/1.

Fixed point numbers such as 4.56, 136.78 and -14.23 are intuitively numbers with a limited number of digits before and after the point. Formally, they are a subset of the rationals. They can be easily represented in computers.

Floating point numbers such as 4.56e34 and -1.23e-2 are comprised of a fixed point mantissa and an integer exponent and are also easily represented in computers.

Computable real numbers are all the points along a line whose locations at a given precision can be described by a (terminating) computer program. For example, we can write a function *d=pi(n)* which returns the nth digit of pi (3.141...). It is rare for robotics to use them. (As they have quite strange properties.)

Mathematicians' "real numbers" are even stranger, having a different definition and properties, and almost no practical use.

Note that for all of these beyond rationals, it is difficult or impossible to perform arithmetic exactly. Approximations will be made, such as cutting off digits of floats. These approximations may give rise to small or large errors, especially when dividing by small numbers close to zero. These often cause problems in simulation and probability computations. In particular, it is not usually possible to test if two numbers are equal to each other.

1.6.2 Vectors

A vector is not simply a computational "array" data structure because it comes equipped with rules for manipulating it in certain ways:

The **dot product** operator is defined between pairs of vectors as

$$A.B = \sum_r a_r b_r$$

It measures "how much the vectors have in common in the same direction".

The **length** of a vector is defined as,

$$|A| = \sum_r a_r^2$$

Vectors are useful to represent coordinates in space. For example a point in 3D space can be written as a 3 element vector, and a point in a 2D space such as a map as a 3D vector. For vectors to have physical meaning, you need to specify the coordinate system they are working in.

In Object Oriented Programming, a 1D array class can be made into a Vector class by adding methods to compute the above, and prefereably a field to say what coordinate system it's in. In ROS you will see something similar. (To be *really* correct, you also need to include a boolean to say whether the vector covaries or contravaries with any change to the coordinate system).

1.6.3 Matrices

Mathematically, an $R \times C$ matrix (standing for numbers of rows and columns), M, is a data structure of values $\{m_{r,c}\}_{r=0:R-1,c=0:C-1}$, together with a specific multiplication and addition operators defined between pairs of matrices. We often display matrices as 2D tables (though it is important to understand the difference between the mathematical object and its pictorial representation), such as,

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

The specific multiplication rule for two matrices, A_{rL} and B_{Lc} is, for each r, for each c,

$$(AB)_{rc} = \sum_i a_{ri'} b_{ic}$$

(which is usually thought of as a sweep across a row of A and a column of B, and known as "chung-kerchung"). This produces an $r \times c$ matrices result.

Note that $AB \neq BA$ – unlike multiplication of scalar numbers. (Matrix multiplication is not "commutative").

The addition rule is simply, for two matrices with equal r and equal c,

$$(A + B)_{rc} = a_{rc} + b_{rc}$$

Square matrices may be inverted to find M^{-1} such that

$$MM^{-1} = M^{-1}M = I$$

where I is an identity matrix having 1's on its diagonal and zeros elsewhere. Usually inversion is performed with a software library. The effect of inversion is that if you transform the co-ordinates describing a shape by M, then transforming by M^{-1} brings them back to their original locations. (Hence the determinant of the inverse is 1 divided by the determinant of M) .

The **determinant** of a matrix is the scale factor which a shape made of vector coordinates experiences when they are transformed by the matrix. It is usually computed by a software library.

The **eigenvectors** of a matrix are the vectors whose directions are unchanged when they are multiplied by the matrix. The corresponding **eigenvalues** are the scaling factors experienced by the eigenvectors after this multiplication.

(To be really correct, we should also specify the input and output coordinate systems that the matrix is intended to operate on, and have a Boolean for each to say whether it covaries or contravaries with them.)

(A vector may be viewed simple as a matrix having $c = 1$. When it is clear than an object is a vector, we will omit the "c" value from its notation, and write its name in lower case. For example, vector a having elements a_r rather than A having $a_{r,0}$.)

1.6.4 Rotation and translation matrices

Matrices can be used, amongst other things, to represent and compute with rotations and translations. To use them, you pre-multiply a vector by the matrix.

2D rotation matrix,

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

In 3D, we can perform similar 2D rotation in different axes,

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

1.6.5 Change of basis

A coordinate system in which each coordinate in independent of the others is called a **basis**.

The effect of pre-multiplying a vector by a matrix is called a **linear transform**. These include scaling, rotation, and flipping of the coordinates represented by vectors. If you represent a line drawing as a sequence of coordinates and transform them all with a matrix, you will see these transformations.

If we combine a rotation matrix multiplication of a vector with an addition of the resulting (or preceding) vector with another vector (called the "origin") then we get a **change of basis**.

For example: Suppose we have the coordinate of a cow in a field, in the 2D coordinate system where y is North, x is West, and (0,0) is the farmhouse. A robot tractor is roaming the farm and wants to know the position of cow relative to itself, in a different coordinate system having the robot's own heading as y, its own left as x, and its own location as the (0,0) (origin) point. Change of basis is the process of converting the cow's coordinate from the "world" (or "allocentric") coordinate system into the robots own (or "egocentric") coordinate system.

As the robot moves around it will have to update this representation at each step – even if the cow is stationary, its coordinate will change due to the continual changes of basis.

(This is also known as "coordinate transform" and occurs in ROS as the *tf* library. *tf* is a major library and often much of a computers power is used to run many *tf* calls during robotics simulation and planning.).

1.6.6 Cross product

For vectors of dimension 3 only (e.g. representing 3D space), we may define,

$$a \times b = |a||b| \sin(\theta) n$$

where θ is the angle between vectors, and n is the unit (length one) vector which is orthogonal to both a and b . The cross product is highest when the two vectors are orthogonal to each other.

1.6.7 Tensors

Tensors are arbitary-dimensional generalisations of matrices. Like matrices, they begin with a N-dimensional array, then equip it with rules for multiplication, change of coordinates, and other operations. The multiplication rule is a generalisation of the one for matrices. You need N booleans to keep track of how each coordinate covaries with a change in coordinates.

(The name 'tensor' is horribly misused in neural network programming to mean only N-dimensional arrays, sometimes with a multiplication operation but rarely with the required coordinate transform machinary. But then 'NDimensionalArrayFlow' wouldn't sound as saleable for a library name.)

1.6.8 Analysis

We often to want to measure how much some y changes as a result of some x. For example y might be distance and x time, which would mean the change in distance as a result of a change in time is what we call "speed".

Suppose the two variables change by a small amount, such as 0.0001. Write their new values as $x + \Delta x$ and $y + \Delta y$. Then define "$\Delta-$speed" as

$$\frac{\Delta y}{\Delta x}$$

We could compute various values of Δ-speed by using different values of Δx , such as 0.1, 0.001, 0.00001, 0.0000001 etc. If we made such a list of Δ-speeds for a real object then we would see that the values become more and more similar as Δ gets smaller. If we could write down the number that these similar number converge to, then we may define its value as

$$\frac{dy}{dx}$$

Conceptually, the "d" means a Δwhich is "infinitesimal" or "infinitely small". Exactly how to define this mathematically is quite complex and belongs to the field of **analysis** in mathematics which studies exactly how such values can be obtained, and some of its results are a set of simple rules that can be used to manipulate them. The learning of the simple rules and their application is called **calculus** and is taught in high schools.

Integration is the opposite process to differentiation, where we want to add up lots of very small pieces to find the area under a curve,

$$I = \sum_i y(x_i)\Delta x$$

As the width of the pieces gets closer to zero, the result gets closer a number written as,

$$\int_x y(x)dx.$$

Generally in practical robotics programming we work just with Δ rather than d. For example most physics engines use discrete "ticks" of time, such as $\Delta t = 0.1$seconds , and compute directly with these values as floating points. However it is also common to use calculus on paper to simplify calculations before implementing the resulting equations with Δ's again.

If you want to try to follow the course without learning high school calculus, then you can sometimes get away with replace "d" equations with equivalent "Δ"ones to get a (limited) flavor of what is taking place in the mathematics.

An alternative "dot" notation is sometimes used for calculus equations involving time, which writes[1]

$$\dot{x} = \frac{dx}{dt}, \ddot{x} = \frac{d^2x}{dt^2}$$

A vector containing gradients in (usually) three dimensions is defined by this symbol (prounced 'del'):

$$\nabla = \begin{bmatrix} \frac{d}{dx} \\ \frac{d}{dy} \\ \frac{d}{dz} \end{bmatrix}$$

1.6.9 Complex numbers and Quaternions

Complex numbers and quaternions are alternative (to matrix) **representations** of 2D and 3D rotations respectively. We distinguish the abstract algebraic structure of rotations themselves from the mathematical symbols used to represent them. A rotation is not a matrix or a complex or quaternion number; but they share the same structure so can be represented by them.

We will find in terms of abstract algebra that this set of rotations forms a **group:** a group is a set of elements together with an operation (multiplication) defined on them, having properties of closure (ab always exists in the group); associativity ($a(bc) = (ab)c$) ; identity (there exists an element I which "does nothing", $Ia = a$); and inverse (for every element a there exists a^{-1} such that $aa^{-1} = I$). Furthermore, they also form a **Lie Group**, (pronounced "Lee group") which is a infinite, continuous valued group (e.g. given any two rotations, you can always have an additional rotation which is half way between them). The Lie group of 2D rotations is called **SO(2)** and the Lie group of 3D rotations is called **SO(3)**. (SO for "special orthogonal").

Complex numbers extend a regular number system (e.g. rational, floating point, computable real) by defining a new number

$$i = \sqrt{-1}$$

This number does not exist in the underlying number system but is *posited* to exist in the extended system. Multiples of i are called **imaginary numbers**. Sums of regular and imaginary numbers are called **complex numbers**, such as

$$z = 3.56 + 4.45i$$

[1] Historically, the "d" notation was used by Newton and the "dot" notation by Leibniz, with both discovering calculus independently at the same time. Their supporters each pushed their notations to become the standard. Newton mostly won but there are a few places where dots are convienent and/or conventional to use, and even where both notations are used at the same time as in Lagrangian mechanics.

We can represent complex numbers as 2D coordinates, with the regular part on the x axis and the imaginary part on the y axis. If we do this, we will find that they can also be represented as

$$z = r\exp(i\theta)$$

with $r = \sqrt{(x^2 + y^2)}$, the radius from the origin to the 2D point, and $\theta = atan(\frac{y}{x})$, the anticlockwise angle from the axis to the point.

If we consider the restricted set of complex numbers having $r = 1$, the unit circle about the origin, then each member of this set can be used to represent a 2D rotation. Multiplying two rotations together gives

$$z = \exp(i\theta_1)\exp(i\theta_2) = \exp(i(\theta_1 + \theta_2))$$

which correctly models how 2D rotations are combined.

Quaternions extend the idea of complex numbers by adding two more imaginary components, j and k, such that

$$i^2 = j^2 = k^2 = ijk = -1$$

Quaternion values are then written using four numbers such as

$$q = 4.56 + 6.34i - 8.25j + 3.23k$$

As with the SO(2) complex number case, we can again restrict the set of quaternions to only those having unit radius. If we do this then this set can be shown to represent SO(3). This representation of 3D rotations is often used in robotics, physics engines, and 3D video games.

SO(3) has a more complicated topology than SO(2). Think for a moment about exactly how an object such as a cup can be rotated. There are thee basic rotations about the x, y, z axes. But consider what happens if perform any two of them in sequence, or the same two in the opposite order!

1.6.10 Probability and causality

Modern (Bayesian) probability theory makes use of a counter-factual world ontology. We suppose there are N possible worlds and that some statement X is true in n_x of them. Then we say the probability $P(X) = n_x/N$.

If we know that some other statement Y is true, then this "knocks out" all the possible worlds where Y is not true from our consideration. Consider the numbers of possible worlds where X and Y are true and false:

	$\sim X$	X
$\sim Y$	$n_{\sim x \sim y}$	$n_{x \sim y}$
Y	$n_{x \sim y}$	n_{xy}

Here,

$$P(Y|X) = \frac{n_{xy}}{n_{xy} + n_{\sim xy}}$$

We often ask what is the probability of X *given* Y , which can be shown (exercise; Bayes' theorem), to be,

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)}.$$

The probability of X being true given that we passively observe Y to be true may be different from the probability of X being true given that our robot acts to *cause* Y to be true,

$$P(X|Y) \neq P(X|do(Y))$$

For example, if we passive observe a garden, we might see that $P(rain|wetGrass)$ is very high. But if we send a robot agent to sprinkle water onto the grass, this does not make the probability of rain higher than when the grass was dry. When we work with agency this requires modeling of causation, which requires its own rules of probability beyond the usual ones (Pearl, Causality, 2000).

1.7 Further reading

In general, the recommended books try to give you at least a quick look at the famous key textbooks in all these fields. While you are unlikely to read them all in detail it will be useful to have handled them and know there are there for your whole career in robotics.

- W. Keith Nicholson. *Linear Algebra with Applications*. Chapters 1-4. Creative Commons Licence, available free of charge from http://people.cst.cmich.edu/marci1t/223/nicholson-linalgwapplications-2019a.pdf. (The other chapters are are also interesting and useful in robotics.)
- Riley, Hobson, Bence. *Mathematical Methods for the Physical Sciences*, CUP.
- Complex numbers and quaternions for 3D games programmers,
  - https://www.3dgep.com/understanding-quaternions

1.8 Exercises

Masters level study is different from undergraduate study. You are expected to self-direct your studies in order to learn what is required. We provide suggested books and activities but we recommend you also find your own activities to help understand the material overviewed in class.

- Find 10 interesting "robotics" jobs on the internet and check if they meet our definition of robotics. What are the current trends in the robotics job market? Some places to look include: *discourse.ros.org/c/jobs/* , *www.indeed.co.uk* , and *jobs.ac.uk* .
- All excersizes from chapters 1-4 of Nicholson, Linear Algebra with Applications (this is a Creative Commons textbook which is freely available and downloadable).
- Make up a 2D coordinate for a mobile agricultural robot, and a 2D coordinate for a cow in the environment. Calculate the position of the cow in the robot's egocentric frame using trigonometry, then using matrices. Repeat for a 3D robot and object.
- Make up three lengths for the components of a three-element robot arm, with a rotating base. Find the coordinate of the hand in the coordinate system where the base of the arm is the origin, e.g. using rotation matrices.
- The members of SO(2) can be visualized as points on a unit circle – how might you visualize or otherwise imagine the shape (topology) of SO(3)?
- Install Ubuntu 18.04 on your computer, or find and learn to use it on a department computer. (essential for next week).
- A fun way to practice programming – which also quite closely resembles many robotics concepts – is to try writing a simple 1980s style 2D video game in Python using the pygame library. You can find several tutorials to help you do this here: *https://www.pygame.org/wiki/tutorials* . You might also like to look at some C++ tutorials as C++ may also be useful in some modules and projects. To write a simple game in C++, try it with the SDL library as in these tutorials: *https://lazyfoo.net/tutorials/SDL/* .
- Short essay: "Modern AI robotics should be based on physical simulation rather than logic. Discuss." e.g. starting points:
  - *https://en.wikipedia.org/wiki/History_ of_ artificial_ intelligence*
  - *https://en.wikipedia.org/wiki/Embodied_ cognition*

2 Mechanics for Robotics

2.1 Newtonian mechanics

Generally in AI and simulation, as in human reasoning and perception, we work at the "perceptual" level of ontology. We want to represent **macroscopic** objects which are important in our perception (and survival) – unlike the Physicists who want to understand ever-lower level models of reality. The ontology of Newton's theory is based on macroscopic "objects" of the kind we encounter in everyday perception, such as chairs and tables, which persist over time, have spatial extent and various other properties. Newton's laws then describe how such objects, once assumed to exist, move around. In object-oriented computation, we would typically represent a Newtonian objects via a class with member variables for spatial and other standard properties. (Modern theories of physics are not based on this kind of ontology, and instead involve strange entities such as quantum fields which do not have the same kind of individuality or properties.) This is roughly how you make 3D physics engines for video games and robotics simulations, and can also be used inside robot perceptual worlds to predict and plan.

2.1.1 Ontology

The Newtonian world is thus not just made of objects and properties – it also includes entitites of space, time, and force as fundamental:

Space is assumed to be a continuous (or at least, floating-point valued), Cartesian (position in each dimension is independent of each other, and dimensions are infinite and do not wrap around) vector space of three dimensions.

Time is **presentist** – not a dimension but rather a sequence of states of the space. It may be modeled (in Physics) as continuous but in computation it is almost always modeled as discrete **ticks**.

Objects are persistent, discrete entities having a spatial extension (**position** and **shape** in space), numerical **properties** including **mass** (also coefficients of friction, damping, and more), and vector **velocity** and **angular velocity**. This notion of objects contrasts with atomic and field theories. We consider a "table" is a single object, not as a collection of atoms or other parts. It is a different though related use from "object" in Object Oriented Programming, and most OOP physical simulations will end up defining a subclass of "Object" called "PhysicalObject" containing these properties.

Forces are 3d vector-valued entities which can be generated by objects and which affect other objects. (This contrasts with field ontologies in which this role is mediated by a field entity rather than by forces; in modern physics these fields also have particle-like behaviors).

2.1.2 Vectors as coordinates

Robotics research generally uses two standards (ROS REP 103) to represent points in space as vectors. A vector used in this way is called a **coordinate**. The three elements of a 3D vector used as coordinates are always labeled $a_0 = x, a_1 = y, a_2 = z$. If we are dealing with coordinates in physical **geographic** space (such as locations on a farm), then we use,

$$x = East, y = North, z = Up$$

like (x, y) coordinates on a regular map. In some applications where the actual **geographic north** is unimportant, we may pick a different direction to function like "north". For example if we are working in a greenhouse which has a rectangular structure but has not been built to align with geographic north, then it may make more sense to pick one of its axes as a **virtual north**. However, we must be careful doing this because one day someone may ask us to drive our robot out of the greenhouse and around the surrounding farm area, which will make our coordinate system very confused.[1]

If we are working in the robot's - or a sub-component of the robot's - **egocentric** space, then we use

$$x = Forward, y = Left, z = Up$$

One way to visualize this is to picture the robot always driving left along a page, then its system looks like a geographic one; the other way is picture the robot driving up the page but with the axes rotated. I prefer the second because it looks more like the "robot's eye view" with objects in similar - but different due to perspective - locations on the page to in a frame from a forward-facing camera.

Sometimes you will encounter large matrices used to represent visual images of pixels, either from robot cameras or from overhead images used as maps. To make their visualizations on screen correspond to the way we write them on paper as arrays of numbers, we use the convention for **geographic images**, that rows and columns of $M_{r,c}$ represent

$$rows = South, columns = East$$

and for robot forward facing camera images,

$$rows = Backwards, columns = Right$$

Note that these are different from the coordinate standards! Hence, it is important to be clear when we are using "rows and columns" of a matrix versus "x and y" of actual space. In typed languages such as C++, this distinction is quite easy to maintain, because "rows and columns" are discrete ints while x and y are floats, and compiler errors will occur if they are confused. In untyped languages like Python this is a quite common

[1] If you operate robots over really large geographic spaces such as self-drive cars and tractors in large cities or farms, then this gets more complicated due to the fact that the Earth's surface is not a flat 2D space but part of a sphere. See Fox 2018 *Data Science For Transport,* GIS chapter, for details.

source of bugs. I like to use variables called r and c to denote rows and columns in images to make this difference as clear as posible.

2.1.3 Laws

First law: An object either remains at rest or continues to move at a constant velocity, unless acted upon by a force. (This law fixes a frame of reference in which the other laws are valid. For example, they prevent us working in an accelerating frame of reference. We might think of the first law as prefixed by "if" and the others by "then".)

$$x(t + \Delta t) = x(t) + v(t)\Delta t.$$

Second law: The sum of the vector-valued forces F on an object is equal to the scalar mass property m of that object multiplied by the vector acceleration a of the object:

$$F = ma,$$

where acceleration acts to change the velocity as,

$$v(t + \Delta t) = v(t) + a(t)\Delta t.$$

Third law: Every action has an equal and opposite reaction: When one object exerts a vector force F on a second object, the second object simultaneously exerts a vector force equal in magnitude and opposite in direction, $-F$, on the first object.

2.1.4 Units

We will work in standard (SI, *Système International*) units:

The unit of time is the Second (s) (defined as the time taken by a standard atomic process). The inverse unit measuring the number of something *per* second is the Hertz, 1Hz=1s^{-1}.

The unit of length is the Meter (m) (defined as a fraction of distance traveled by light in 1 second)

The unit of mass is the Kilogram (kg) (defined as the mass of a lump of metal in Paris). A (metric) tonne is a non-SI unit equal to 1000kg.

The unit of force is the Newton (N) (defined by the above and Newton's laws).

SI defines standard prefixes for powers of 1000:

name	symbol	value	name	symbol	value
kilo	k	10^3	milli	m	10^{-3}
mega	M	10^6	micro	μ	10^{-6}
giga	G	10^9	nano	n	10^{-9}
tera	T	10^{12}	pico	p	10^{-12}
peta	P	10^{15}	femto	f	10^{-15}
exa	E	10^{18}	atto	a	10^{-18}
zetta	Z	10^{21}	zepto	z	10^{-21}

2.2 Folk axioms for particles

In addition to Newton's basic ontology and laws, we then need to add further axioms to model particular forces. These are rather more messy than Newton's basic setup. Our choice of a particular set of forces to model further specifies the "theory" in use. Newton's theory is like a "core language" and these additional forces are like a library to be imported on top of it. Some of these are used in unrealistic physical ways to achieve robotics goals, e.g. charge models of joint attachment. Hence they look different in robotics than in Physics. Some forces can be "explained away" by replacing them with applications of more basic forces in some theories. (For example, friction is usually modeled as an elementary force in robotics, but may be explained away via microscopic electromagnetism). Such axioms that are blatantly not intended as theories of fundamental reality but useful in the everyday world are known as "folk physics". The simpler ones are applied to small simple objects or Newtonian 'particles' which are assumed to look like small, solid spheres.

2.2.1 Agent forces

Agents are objects - possibly limited to "I" - which can make decisions about when and how to apply additional forces. In robotics programming, we assume that the robot "I" may produce forces at parts of its body (or sometimes elsewhere) at "will".

2.2.2 Gravity force

Gravity applies a downwards force to all objects in proportion (constant $g = -9.81$) to their mass m,

$$F_g = mg.$$

2.2.3 Spring linkage forces

Often we wish to extend folk physics to model objects which are joined together. For example, a tractor pulling a trailer has a "joint" which connects them, and allows them to rotate whilst constraining their positions. The real wheels of the tractor are separate objects from the tractor. They can be modeled by a "joint" which constrains their centers to coincide with point on the tractors, but enables them to rotate about one axis. The front wheels are more complex as they can rotate in two axes (steering as well as rolling along the ground).

We will usually model these effects using **springs**. Springs are an addition to the basic Newtonian ontology as well as adding a type of force. Assume initially that we are dealing with "small" objects whose shape is not important. For such objects, a spring is a **relation** between them, with a strength k_s called the "spring constant", and a preferred vector offset location d for the second object's position x_2 relative to the first object's position x_1. (Note that "relations" are also ontologically new here: unlike a "property", a relation is something involving more than one object.) Such a spring applies a vector

force (equally and oppositely as required by Newton's third law) to the two objects it connects,

$$F_s = -k_s(x_2 - (x_1 + d))$$

This forces the two objects towards the relative position d.

Springs may also exert a second **damping** force as

$$F_d = k_d(v_1 - v_2)$$

Which forces the two objects towards having similar velocities via a "damping constant" k_d.

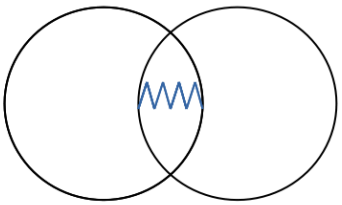
2.2.4 Particle collisions

Next we want to add forces which repel objects from one another when they "collide". In Physics there are very complex models of what happens at the atomic level during collisions, but in folk physics we seek simpler models.

Suppose first that we live in a world of "small" spherical objects bouncing around. They excerpt no force on each other except when they "collide".

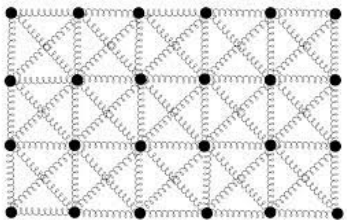
Defining a "collision" is quite subtle. Suppose A and B have a gap of d between them. Does a collision exist only when d is exactly zero? Or when d is small and positive, or when d is small and negative? It is uncomputable to test if a real number is exactly zero, and similarly "messy" to test if a floating point number is zero. So in practice we must model a collision as any "small" positive or negative d.

If we pick some arbitrary small d, then we may add an axiom to our theory which creates a temporary repulsive (negative spring) force between the two objects. Empirically, this works pretty well as a model of the collisions we see in our perception. (There are also ways to explain it in terms of more complex sub-atomic physics theories).



2.3 Finite Element Methods (FEM)

One way to model a macroscopic object is as a mesh of simple particles connected by springs, such as this rectangular block:



Such a model is able to deform and vibrate when it is bent, streched, or collided with another object. It is called a Finite Element Method (FEM) model. FEM models are very compute intensive and usually have to run on large compute clusters for long periods, not in real time. They are used in Soft Robotics, such as for bending robot whiskers and tentacles. In the ontology of these models, there is no table, just a bunch of particles and springs.

2.4 Folk axioms for rigid bodies

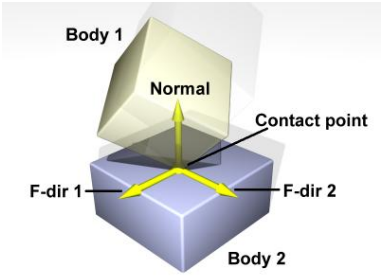
A rigid body is a macroscopic object having a location and a shape. 'Rigid' means it does bend or otherwise deform. Rigid bodies can be modelled with much less compute power than FEM by a few additional folk axioms.

Some of these axioms may be *derived* from the simpler Newtonian theory by assuming the body is made of a mesh as in FEM. The particles connected by springs which is allowed to deform only for *very short* periods of time. However once we implement the rigid body folk rules they are used as axioms rather than emerging from simulation of such meshes: there *is* a table, and there are no particles or springs.

2.4.1 Collision

We can make a convincing model of our experiences of seeing collisions between large objects (known as "rigid bodies") having shapes, if we briefly consider them as being made up of meshes of small spherical objects held together by springs.

A collision between two such meshes is (almost) always one point: a corner of large object 1 enters large object 2:



(Fig: ODE documentation)

Consider a single particle in the corner of 1 which is entering 2. We can model it as experiencing a spring due to the particles in 2 pushing it back out, in the direction orthogonal to 2's surface. By Newton's law, this also creates an equal and opposite

reaction force on 2. The spring might have damping. Different choices of spring and damping forces will model different pairs of surfaces' collision properties. For example a rubber tyre might rebound a metal bullet, while wood will absorb its energy via damping.

2.4.2 Rotation axioms

We may define point called the **center of mass** as the mass-weighted "average" location of a small object in a mesh object,

$$R_{CM} = \frac{\sum m_i r_i}{M}$$

and a **total mass** for the mesh object

$$M = \sum m_i$$

It can be shown that the mesh object then behaves similarly to a particle with position R_{CM} and mass M, if all forces act only at the center of mass.

The effect of a vector force applied at location d from the center of mass, on the acceleration of the center of mass, is,

$$F.d = Ma$$

However it may also spin about its center of mass, an effect which is not so important for small spherical objects. Spinning can be shown to depend on a quantity called **moment of inertia**, which measures the average squared distance of mass from the center of mass,

$$I = \sum m_i (r_i - R_{CM})^2$$

We may also define **moment** (also known as **torque**, especially in practical robotics and vehicle mechanics) as

$$\tau = F \times d$$

where d is the vector from the center of mass to the point on the shape that a vector force F is applied. If we do this then it can be shown that a rotational analog of Newton's second law ($F = m\frac{dv}{dt}$) emerges,

$$\tau = I\frac{d\boldsymbol{\omega}}{dt}$$

where ω is the angular velocity.

We can thus separate the effect of a force on a rigid body into a particle-like acceleration of its center of mass, and an angular acceleration.

2.4.3 Rigid bodies folk axioms summary

Rather than model rigid bodies as actual spring meshes of particles, it is common to use a theory which instead just takes the above results as axioms:

Shape ontology 1: Every object has a point property called center of mass

Shape ontology 2: Every object has a scalar quantity called moment of inertia

Shape ontology 3: Every object has a shape

Shape ontology 4: Every object has an angular rotation

Shape ontology 5: Every object has an angular velocity

Shape law: A force acting on an object at location d from its center of mass acts to (a) accelerate the center of mass as $F.d = Ma$, (b) accelerate the angular rotation as $\tau = I\frac{d\omega}{dt}$.

Collision law: when two object's shapes overlap, a temporal spring is created to push them apart, as in the particle case. The force acts at the center of the overlap, and at right angles to the tangent of the overlap surface.

Getting these into rigorous logical form (e.g. where we might try to prove they don't lead to contradictions) is quite hard. They are more naturally implemented in a physics engine just as simulation code. This is what we find actually implemented in engines such as Open Dynamics Engine (ODE) used in robotics simulators such as Gazebo.

2.4.4 Friction

We observe in our perceptual world that when a pulling force is applied to an object which is resting on a surface under gravity, it encounters an opposing force which depends on its mass, the type of surface S', and on the type of its own surface S. We call this force "friction", and may model it by adding to our physics model a **Coulomb friction** force,

$$F_f = \min(k_{SS'}F_n, F_a)$$

where F_f is the friction force, acting opposite to resist the applied force F_a trying to move the object, $k_{sS'}$ is a coefficient of friction specific to the pair of surface types (e.g.. for "teflon on teflon" is very low; rubber tracks on grass is high), and F_n is the "normal" force which is pushing the objects together and which is usually (but not always) gravity, $F_n = mg$ with m the mass of the object being pushed around. The Coulomb friction force acts to match and resist the applied pulling force up to a limit (hence the "min" where it gets overcome by F_a. At this point the applied force is said to have "gained **traction**" and the object begins to move. A "tractor" is a large (potentially autonomous) vehicle designed to provide traction.

(This model is used in ODE which in turn is used in Gazebo, often with ROS. In ODE there are two separate coefficients for x and y directions of motion, e.g.. to allow for an object to glide easily forwards but experience heavy friction side to side. In ODE, each object stores its own coefficients as a property, and when they interact it takes the smaller of the two objects involved. This is an object ontology: a more realistic model would consider the coefficient to be a relation between the two objects but this is harder to represent.)

2.4.5 Rotary springs

We also sometimes need to model the equivalent behavior for rotary systems, using rotary springs. These are springs which apply spring moment, rather than spring force, in proportion to an axis which has rotated angle θ away from its "home" orientation. They may also apply a damping moment analogous to the damping force, in proportion to the angular velocity.

2.4.6 Joint constraint forces

All robotic systems act by applying forces to component parts of a robot, while also constraining these parts to remain part of the whole robot. For example, a wheel or a robot arm segment is constrained by fixing it to an axle, but able to rotate about this axle as a motor applies torque to it. An autonomous tractor pulling a trailer will be hitched to it by a pin which similarly allows rotation about its axis but prevents motion away from the axis, though this joint is passive rather than actuated (and robotic planning especially for reversing motions to park the trailer may be quite complex in taking account of its motions).

In theory, these joints represent **hard constraints** on motion. It should not be possible for any of the rigid body parts to move in a way with violates the constraint. For example, a wheel may not move to any pose in which its center is not at the axle pose.

However, such joints come with a modern form of **Zeno's paradoxes** of motion. Most physics engines work by considering each object in turn and updating its position while holding the others constant. (Even parallel implementations of engines such as GPU based make similar assumptions while the others are being updated.) Consider a model of tractor moving forwards. We want to alternatively update the pose of its body and its wheels. But they are constrained to match the axle pose. So when we try to update the wheel, it will never move because this would require moving a small distance away from the current axle location. Similarly, the body will also never move because it would violate the same constraint.

To handle this problem, joint constraints are instead modeled as **soft constraints**. These allow *very small* deviations from the constraints to occur at each step. They can be modeled as very strong (and strongly damped) springs which strongly encourage the parts to be in their desired places and pull them into this form a soon as possible after any small deviation. This allows each time step of the simulation to violate the constraint a little bit – the wheel can make progress, then the body, then the wheel, and so on.

These constraints are the cause of the many **exploding simulations** which you will inevitably experience as bugs in your physics engine models. If your robot appears to explode, scatting its component parts all over the world, is is likely because a these constraint spring forces have reached extreme values. The usual remedy for this problem is to use a smaller time step in your simulation which reduces the size of the constraint forces at each tick, at the expense of needing more ticks and hence more compute power.

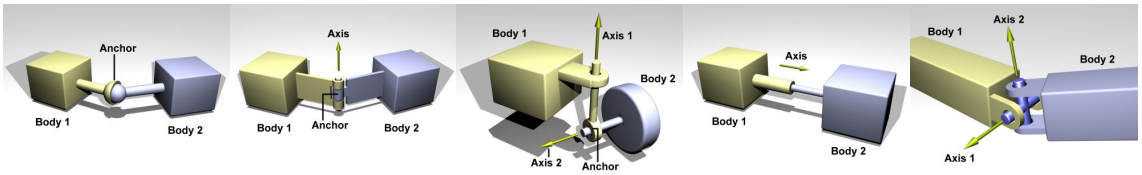


Figure 2.1: Joint models from the Open Dynamics Engine (ODE documentation) (See also video at https://bitbucket.org/osrf/gazebo_models/pull-requests/241/added-model-demo-joint-types/diff)

2.5 Derived quantities from Newton

The presentation of Newtonian mechanics seen previously is just one possible way of describing the theory. Its ontology is based on objects with position, shape and mass; and forces. Under this presentation, the following quantities are simply defined from them, rather than being ontologically "real" themselves. In Computing we care more about this distinction than in Physics, because the ontologically primary quantities are what are implemented in code. (There are other theories of Physics in which they are considered primary, for example quantum mechanics emphasizes momentum over velocity, and the Standard Model is written in terms of energy.)

2.5.1 Momentum

If we define for each object a *momentum* as its mass times its velocity, mv, then it follows from Newton's laws that the sum of momentums of all objects involved in a collision will be conserved.

This is because the force acts equally on both objects, $F = m_i a$ for some period of time T. Resulting in

$$\Delta v_i = (F/m_i)T$$

$$\Delta m v_i = FT$$

in opposite directions. These have the same magnitude and opposite directions.

This provides a mathematical shortcut when calculating outcomes of collisions because we know that the total momentum must be conserved. If two balls collide and we know (somehow) the resulting motion of one, then momentum helps find the other's.

2.5.2 Energy

Define kinetic energy for an object as

$$K = \frac{1}{2} m |v|^2$$

And the kinetic energy of a closed system as

$$K = \sum_i \frac{1}{2} m_i |v_i|^2$$

If we do this, then we can also define total energy for a closed system as a (arbitrary) constant, then define potential energy V of the system as the difference between total and kinetic energy,

$$V = E - K$$

We will then find that when forces act on objects and change their (ontologically primary) velocity, they also makes a change in the balance between KE and PE.

For example, when an object is decelerated by a spring, the contracted spring can be said to contain potential energy, which must by these definitions be,

$$V_s = \frac{1}{2} k x^2$$

Or if an object is moved through a distance h where (folk) gravity is acting, the PE change is opposite to its KE change from the acceleration,

$$\Delta V_g = mgh$$

(This view of energy change as force times distance works the same for movement through all other forces too).

As with momentum, these definitions can provide mathematical shortcuts to calculations when we know things are conserved.

If two balls collide in collision without losing energy, then the two conservation laws, of kinetic energy and momentum, can be sued together to find the resulting motions.

Most systems of macroscopic objects however will appear to lose energy during collisions, friction, electrical resistance, and other energy conversions. This is only appearance: the "lost" energy is usually transformed into heat. "Folk Heat" can be modeled as a new property, or in physics it is modeled as movement of energy into motions of small particles beyond the folk object ontology. In simulations we usually ignore heat and model lossful collisions and interactions, for example a friction force which simply slows an object without conserving its energy.

The SI unit of energy is the Joule, 1 J = 1Newton*1meter. In practical settings (such as home energy bills) the kilowatt hour is also used. (1kWh ~= 3.6MJ).

Power is defined as energy per time. Its SI unit it the Watt, 1 Watt = 1 Joule per second. Most mechanical domestic appliances (lawn-mover, washing machine), big horses, and agricultural robots are around 1kW.

2.6 Ontology in ODE

The ontology used is as follows. Visual and physics models are stored separately, though may use the same meshed and values in some cases. A common bug is for them to

diverge. It is an object-oriented ontology, so even entities would be more clearly modeled as relations appear as objects with properties (such as a joint between two links):

world:

- gui
  - camera
- model(s)
  - pose
  - link(s)
    - pose
    - intertial
    - collision
      - geom
      - surface
      - friction
    - visual
      - geom
      - material
  - joint(s)
    - type
    - pose
    - parent link
    - child link
  - plugin(s)

Fig. 2.6 shows an example of a humanoid robot modelled using this ontology.

If you are interested to see an example of a robotics simulation build using raw ODE physics and OpenGL graphics libraries (as would more usually be wrapped by platforms such as Gazebo) there is one here which you can also extend to make your own simulations:

https://github.com/charles-fox/freebots

2.7 Physics theories as AI

We have examined high-school Newtonian physics from a robotics and AI perspective. Robotics needs such physics for simulation and planning, while AI may consider possibilities for representing and working with the theory itself as a form of knowledge representation and reasoning.

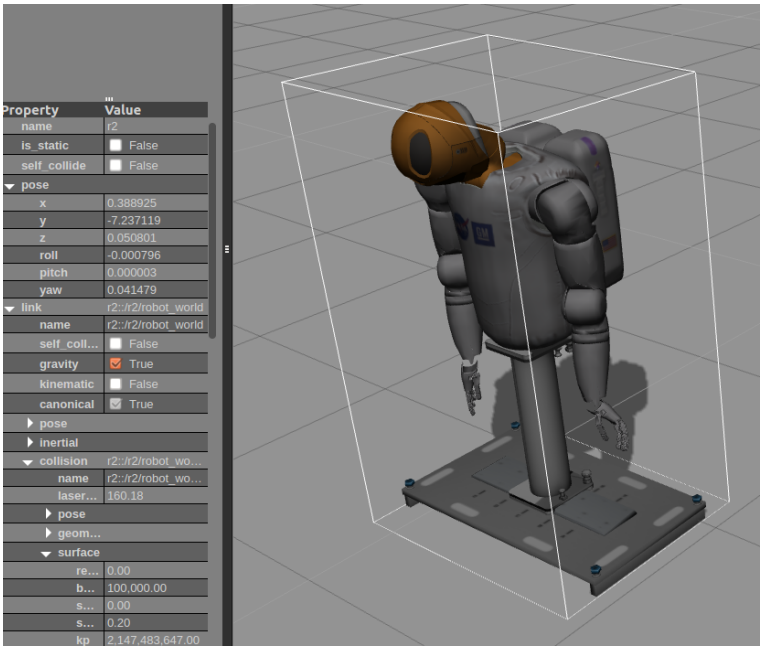


Figure 2.2: ODE ontology in use by Gazebo to model a humanoid robot.

We have seen that Newtonian physics is non-trivial once non-point objects are introduced, and relies on many "folk" forces to make realistic simulations. Such forces are typically found in physics engines such as Open Dynamics Engine, ODE, as used both in robotics simulation (Gazebo) and in computer games.

Some of the famous **failure cases** for Newtonian models include: objects are able to accelerate to infinite speeds; communication is possible over infinite distances; trying to model small particles rather than macroscopic objects will fail; rigid bodies can't bend, split into smaller objects, or merge into larger ones. For everyday robotics these are often obscure cases, but if you need to work with, say, space robotics, nano-robotics, or soft robotics then they may cause problems.

From an AI viewpoint, considering Newtonian physics as just one possible theory which could be built from axioms in a formal system is interesting. How is an AI robotics system built on a simulation model different from one built one the logical axioms of the theory? Do we lose anything by relying on **simulation** rather than **logical reasoning** about physics?

Especially when we add additional forces such as Electromagnetism (in later lectures) we will see that there are **inconsistencies** in and between the theories, and trying to formalize even "high school" physics into formal axioms raises questions about these inconsistencies, and how AI systems and humans might deal with them as they arise. Human students of physicals are well-trained from an early age to ignore these inconsistencies by choosing which theory to work with in which case. But an AI system left to its own devices will likely be able to prove two different outcomes of the same event

using different theories, leading to a contradiction, then potentially to the blow-up of it's whole inference engine. (As a contradiction in standard logic can then be used to prove anything).

The choice of ontology and laws seems highly arbitrary. While modern physics theories have tried to reduce the number of these axioms, they too must postulate other apparently **arbitrary axioms,** pushing back the **cosmological question** of why our world happens to appear under such structures and laws.

2.8 Exercises

- Mechanics questions from Open University "are you ready for Electromagnetism" question sheet. If this is too easy then please help other students to develop your team skills. If it is hard, use standard engineering maths textbooks and your colleagues to help.
- Suppose a 250kg tracked agricultural robot has broken down and needs to be towed back to the farm. It has two tracks about 1m long and 200mm wide. The tracks are stuck and do not move. Estimate the force needed to gain traction to pull it home. What is the smallest commercial tractor you can find able to provide this force? How many horses (1 "big horse" power is about 1kW; a traditional "horsepower" is a bit less, around .75kW) is it equivalent to? Suppose the robots themselves are 1 big-horsepower. Can they be used to rescue one another?
- Using Gazebo tutorials, design and run a simulated robot:
  - http://gazebosim.org/tutorials?tut=build_robot&cat=build_robot
- Apply torques to the robots wheels in Gazebo. Find settings and experiment with friction on the wheels.
- Extend the basic model to a car-like vehicle where the rear wheels move together at a speed, and the front wheels are coupled by a steering linkage. (Use different joint types to do this).
- Describe each of the fields in the Gazebo XML and how does Gazebo's XML ontology map to the folk physics we have discussed?
- Do as many of the exersizes from LaValle, *Planning Algorithms*, chapter 3 as you can.
- Short essay: Physics engines do not use formal axioms, and make use of apparently arbitrary "folk physics" rules implemented in code. What connection do these simulations have to Physics and to the real world, and why should we trust them to provide predictions of what happens in the real world?

2.9 Further reading

2.9.1 Essential

- Bourg, D. (2001) *Physics for Game Developers*, O'Reilly. Chapters 1-5. (If you are a computer person, study the physics; if you are a physics person, try the computer implementations.)
- LaValle, *Planning Algorithms*, chapter 3. (Open-source book available online at http://planning.cs.uiuc.edu/)

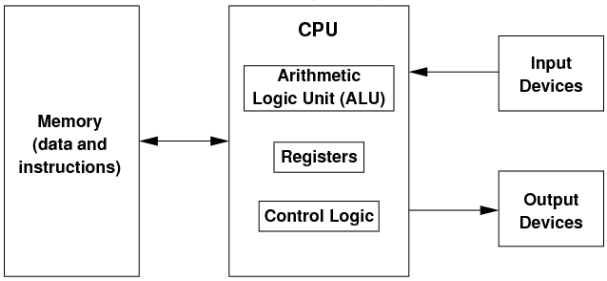
2.9.2 Advanced

- *The Feynman Lectures on Physics*, Volume 1 (Mechanics). Available online at *http://www.feynmanlectures.caltech.edu/* .
- Herb Simon (1947) *The Axioms of Newtonian Mechanics.* The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science, Series 7, 38(287). *https://www.tandfonline.com/doi/abs/10.1080/14786447708561148?journalCode=tphm18*
- C. J. Papachristou. (2012) *Foundations of Newtonian Dynamics: An Axiomatic Approach for the Thinking Student.* https://arxiv.org/abs/1205.2326

3 Computation for Robotics

3.1 Computer architecture overview

Most current computers are comprised of a **CPU**, **memory**, and **IO**. The CPU contains an Arithmetic Logic Unit (ALU), registers, and control logic:



The memory can contain both programs and data. Each location in memory has an address, like a mail pigeonhole.

The registers on board the CPU are a fast, small form of memory.

The ALU acts on data in the registers, for example to add two registers together and store the result in a third. It performs a range of basic arithmetic and logical operations on the registers.

The control unit schedules the work of all the other parts of the CPU. This includes copying data and program instructions in from memory addresses to registers, copyign between registers, sending data and results between registers and the ALU, and checking results of registers to implement 'if' statements by jumping to other parts of the program.

Programs are made of sequences of machine instructions. In memory they are represented in binary machine code. But we can easily translate them to and from human readable assembly language to see instructions such as

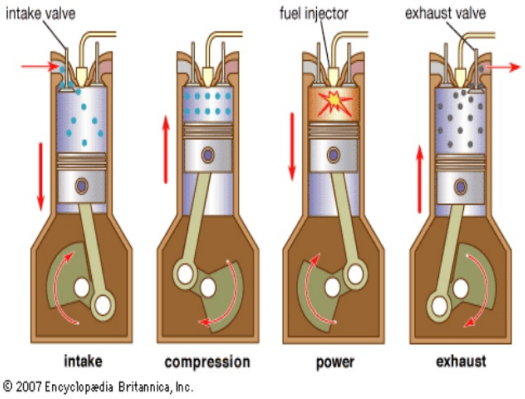
ADD R1 R2 R3 #add register 1 to register 2 storing result in register 3

LDR 4353 R1 #move data from main memory address location 4354 into register R1

CMP R1 R2 #compare registers R1 and R2, set a flag if they are equal

JE 345 #jump to program line 345 if the equal flag is true

A CPU operates in a self-sustaining cycle a bit like a car engine's "suck squeeze bang blow" cycle, whose lower **crank** shaft power output feeds back to drive the **cam** shaft at the top to control the cycle:



The control unit, like the engine, goes through a series of states which usually include fetching instructions, decoding them, and executing them (by passing control temporarily to the ALU and other components). The control unit is (conceptually at least) hardwired so that, like the engine, the completion of each stage triggers the next one in a cycle.

A **compiler** is a program which translates a higher level language such as C into assembly language. An **assembler** is a program which converts the assembly language into machine code. A **loader** is a program which loads machine code into memory and adjusts its addresses to make it executable, and executes it. A **linker** is program which combines several component programs (libraries) together and adjusts their addresses to make them work together. Adjustment of addresses is usually need when running an operating system which presents a virtual memory interface to the program rather than physical addresses.

There are two general styles of CPU architecture. **CISC** (Complex Instruction Set Code) is found in desktops and servers and uses lots of silicon to impliment many specialised instructions in hardware. For example specific instructions to handle vector arithmetic, 3d manipulation, signal processing and cryptography. **RISC** (Reduced Instruction Set Code) is 'lean and mean', it reduces the instruction set to the smallest size, containing only generic instructions, so that the complex instructions found in CISC are implemented instead in software. RISC can execute instructions faster than CISC because of its simplicity, though requires more instructions to do the same work. RISC is usually found in embedded systems on robots because it is lower power, cheaper, and simpler, which fits the general criteria of embedded systems.

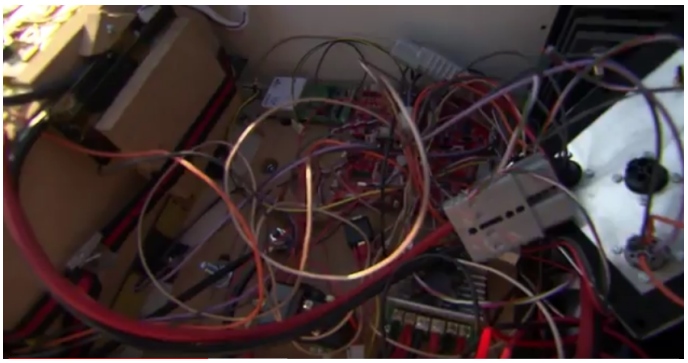
3.2 Embedded Systems for robotics

An **embedded system** is an entire computer system built-in into a larger device. This could be a robot or could also be something like your washing machine. A washing machine was never really intended to be a computer. It was supposed to do washing, rinsing and drying, but there's probably a computer in there somewhere nowadays.

In robotics, embedded systems usually occur inside the robot, at the level of devices which take serial or ROS commands and convert them into actual actuations of the robot, or commands which read from the sensors and return their results at serial or ROS level.

They are seen physically as circuit boards and other devices which are not normal PCs on-board the robot. Often there will be one or more PCs on-board the robot as well, and make further PCs at a base station linked to the robot, but these are not embedded devices. A a Computer Science based roboticist, you may work with embedded systems in two distinct ways. Most of you will interact with them as **users**, for example sending ROS or serial messages to any from them. You will need to have a basic idea of what they do and when to blame bugs onto their designers. Some of you might also work as their **designers**. Typically they are designed by a mixture of electrical engineers and C programmers; the engineers make the physical circuit boards and select micro-controllers, then you will write low-level C code and compile it for the micro-controller. If you do this, expect to work at the level of bits, bytes, and pointers, rather than object orientation.

Figure 3.1: Agricultural robot embedded systems for motor control, power management, sensing, and comms.



3.2.1 Typical features of embedded Systems

Application-specific. An embedded system is going to be something application specific, usually designed for one purpose and only one purpose. This is different to a general computer which can be used to do anything.

Encapsulation. Typically the complexity of the computation is hidden from the user, it's embedded inside the device, probably the user doesn't even realize that there's a computer in the device. In robotics, an embedded system will present some kind of interface to the higher level AI programmers. This is often in the form of a serial port protocol. This interface also forms the level of discussion between the embedded system designer and the user and will be the main topic of meetings between them.

Mobility. Embedded systems typically have much more emphasis on physical size and ease of mobility. A lot of these devices are designed to be carried on your person like your phone for example. An embedded system is often doing something in real-time in the real world such as controlling a machine in a factory and the precise timing of that output is much more important than on a typical home computer system.

Efficiency becomes very important, we put tight sets of constraints including cost constraints on designing the system as cheaply and as simply as possible.

Things have to be **fast** (real-time applications) and they have to be reliable. Very often these devices aren't connected to mains electricity and instead are battery powered. This is perhaps the biggest bottleneck in modern computing now. How much power you draw and how large the capacity of the batter is. This consideration is made in large scale robots as well as pocket sized systems like your mobile phone.

Price is always an issue, there are often many companies competing to make the same device/product and really just competing on price rather than features. Size is an issue with embedded systems as a lot of these devices are to be carried on your person, they've got to feel okay to carry around and be portable.

Safety and reliability. Embedded systems are usually the last line of defence before a robot goes dangeriously, such as human life-threateningly, wrong. Any small bug or glitch at this level can be catastrophic, for example sending a high voltage to a powerful robot motor making a vehicle drive backwards at 100km/h, or smashing a robot arm into a wall. Systems need to be as simple as possible to enable precise human understanding of how they behave, and to reduce the scope for bugs.

Hardware-software interface. Another design consideration is how the hardware will interact with the software. Desktop computing has moved away from computer architecture as the main programmer interface to the operating system. So in general computing there's been a move to commoditise the hardware. You can buy your processor, your memory from different places and run the same **OS** on the top. In embedded systems this can be very different as usually you're designing a custom device and the hardware and software are designed together by the same team of people. They're going to be closely matched together and generally are not usable on other people's hardware and software. In particular this becomes a really contentious issue for the open source political movement.

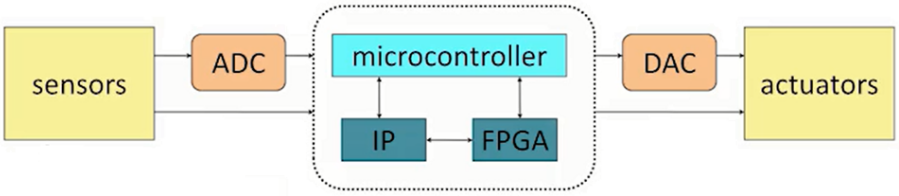
Culture. Their design is typically led by Engineers, not Computer Scientists. Culturally the big difference here is if you work with this stuff you're typically working with and led by Engineers rather than Computer Scientists in this field. This is often seen as the domain of engineering. This can lead to some tension between the approaches of the two fields there

Open source hardware. If you an open source advocate you want all of your software to be open source so that anyone can use it but you typically don't make such a big noise about hardware. You're typically more happy to buy a commercial graphics card as long as your open source software runs on top. When you get to this level the boundary between hardware and software becomes very blurry. It's not completely clear if this kind of software should come under open source definitions at all if it's only ever going to be used with this specific hardware that is already closed source. So often here you'll get a Computer Scientist and an Engineer working on together and having very different ideas about this, with Engineers typically having a more proprietary, commercial view.

3.3 Embedded systems components

Typically an embedded system is something that interfaces with the real-world. We begin by thinking about input, this is gathered by **sensors** on the device that receive data from the world. A sensor is anything that takes a real-world (usually analogue) quantity and turns it into an electrical quantity, this quantity could relate to anything you want to measure.

Figure 3.2: Dataflow of an Embedded System from input (sensors) to output (actuators)



Often those sensors are analog sensors and need some kind of **analog-digital conversion** (ADC) before we can perform actual computing on it. Typically this is done with a type of processor called a micro-controller, there are also things like FPGA and IP cores that are used in this space also. Then you come back, converting the processed information back into analog before sending the information to actuators which cause events to happen in the real world.

The other end of the chain is the **actuator**. "Actuate" is an engineering term meaning 'to put into action or motion'.

3.3.1 Micro-controllers

Figure 3.3: Micro-controller chip

It is hard to define a microntroller (also known as a micro controller unit, MCU, or μC) versus a regular CPU, as it is primarily a marketing term used to describe any form of

processor chip sold with embedded use in mind. But typically the processors marketed in this way will have a few features in common such as:

3.3.1.1 CPU

- **lower power** than a desktop, often shorter word length (eg. **8-bits** still common)
- **fixed-point arithmetic** - often there is no floating point hardware because floating point is very complex and requires lots of silicon to implement and power to run. It is also bug prone at the hardware level. It comes as a big shock to many programmers to have to go back to fixed point!

3.3.1.2 Harvard architecture memory

The memory on these devices then is much smaller than in desktop PC's to fit it all on one chip. Typically the memory on the chip is split into two, forming a **Harvard architecture**:

RAM is fast read/writable memory, usually SRAM, volatile (cleared on power-off), used to store temporary data.

ROM: Non-volatile (remains on power-off) 'read-only' memory used to hold the program and configurations, also known as 'firmware'. In prodiction systems, this information is "burned" into the ROM usually just once at manufacture, or occasionally by the user as a firmware update, by connecting the device to a PC and using special hardware and software. Traditional ROM was physically difficult to reconfigure (e.g. requiring UV light to be shone onto the silicon) but modern electrically erasable ROM (EEPROM) as found on Arduino make it easier.

3.3.1.3 Timers and counters

Timers and counters are often built into microcontrollers, specifically this is because we're dealing with the real world and the system needs to respond in these metrics. What's the difference between a timer and a counter then?

A **counter** counts number of times a particular event or process occurred, with respect to a clock signal (changes, pulses, etc.). It can be used to count the events happening outside the microcontroller. A register is incremented corresponding to an external signal, rather than the clock. Whereas a counter is counting the number of discrete occurrences of a particular event, possibly the number of incorrectly shaped items on an assembly line. Incrementing a value whenever this scenario occurs.

A **timer** measures precisely time intervals or elapsed time, typically using a register incremented for every machine cycle. A timer is measuring time as a continuous variable, how much time has elapsed between one event and another. Typically you'll do this using the clock in your microcontroller, measuring the amount of cycles and using the frequency of the cycles to discern time passing.

A **watchdog timer** is a special timer which automatic resets the MCU in the case of failure, for systems that have to be reliable in the real world. If something goes wrong

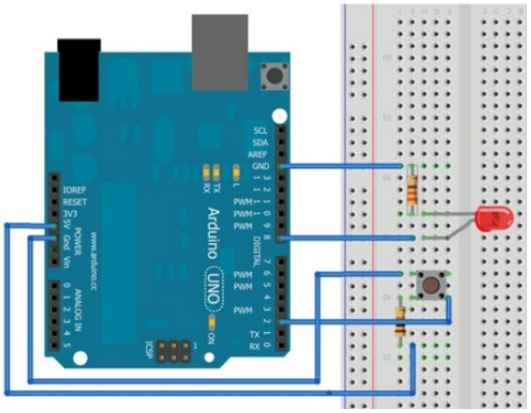
you need a way to reset it (Preferably without needing surgery in medical cases!) A watchdog timer is going to automatically monitor if somethings going wrong and do a physical reset of the device to bring it back up again. That's done in the hardware itself and isn't part of the CPU's program.

3.3.1.4 IO ports

The other major specialised function of a microcontroller is communication. Embedded systems rarely run by themselves in robotics – they are usually going to be communicating either with other embedded systems, an external device or a host PC system. So we usually find IO modules, ports, and some very basic, slow serial communication built into the chip itself.

Let's look at the IO ports of a typical microcontroller, on an Arduino board. Unlike a regular desktop computer IO is not done using a bus, and instead raw voltages are sent directly out of **pins** on the microcontroller. This enables us to connect wires straight from the microcontroller pins into electronic breadboard circuit without needing to worry about busses, addresses, or IO modules.

Figure 3.4: I/O: Circuit connecting a breadboard LED and button to an Arduino



It's often the case that we try to **minimize the number of pins** on these devices. We might build something like the Arduino with a small array of analogue and another of digital pins. Instead of saying half are for input and half are for output we might have them configured as dual purpose. If you have a device that's only taking input you switch everything to input mode etc. A very common mistake in embedded systems programming is having your pins configured incorrectly, trying to send stuff out from an input pin and so on.

IO is especially important in embedded systems to provide a way to **upload programs** to them. Unlike with desktops, it is not usually possible to do the development work itself on an embedded device, as this would require graphics, keyboard, operating system and compiler to all run on the low power device. Instead, we do development work on a desktop, and perhaps test our programs there too using simulation or emulation – before

transferring the final binary executable onto the embedded device. Microcontrollers have special modes for doing this, usually they can be connected to a desktop via USB, serial port, or other means, then put into "**firmware upgrade**" mode to copy the executable into their non-volatile program memory via this connection together with a software device driver on the desktop machine.

3.3.2 Analog-digital (A/D) conversion

We are always dealing with signals coming in and going out that are analogue and inside the controller we want digital signals, this requires conversion on both ends. These converters may be found outside the microcontroller, connected to its pins (such as our favourite Aduino compatible MCP4725, *www.adafruit.com/product/935*), or in some cases on the microcontroller silicon itself.

The classic case of A/D conversion is audio processing. An analogue signal from a microphone is sent to a digital processor and adding effects (processing) the signal before sending an analogue signal back out to the speakers. This is done by taking an analogue signal wave and quantising it, turning it into a digital signal over time by sampling. You can do this at different **resolutions**, fig. 3.5 shows a 2 bit resolution and gives you 4 possible values. This gives the signal 4 possible heights to try and represent the Analog signal with and probably won't sound so nice when we convert it back post-processing. In fig. 3.6, we see the 8 levels available in 3 bit resolution, giving the digital signal representation more accuracy when compared to the Analog wave.

Figure 3.5: Quantising an Analog signal to Digital with 2 bit resolution

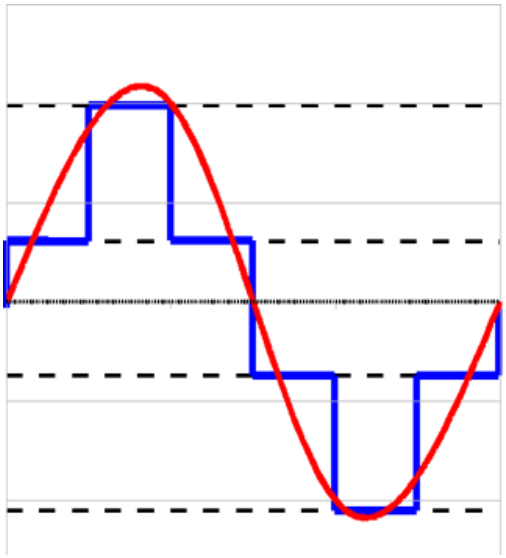
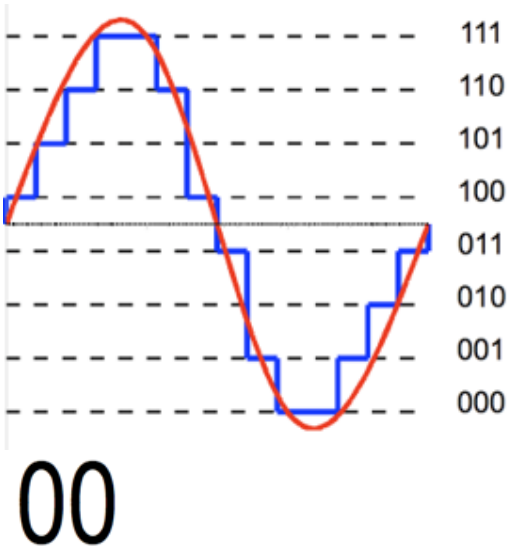


Figure 3.6: Quantising an Analog signal to Digital with 3 bit resolution



When converting the other way, D-A Conversion (DAC), some devices (e.g. Arduino Due) do true conversion of digital integers to analog voltages. Cheaper ones (such as

Arduino Uno) approximate it with **Pulse Width Modulation** (PWM). Here the output is only ever 0V or 5V. If 3V is asked for, the output oscillates rapidly between 0V and 5V, spending 3/5 of its time at 5V and 2/5 at 0V, to give a temporal average of 3V. For some applications this creates no noticable difference but in others it can play havoc with the output.

3.3.3 IO protocols

We will very often find the following communications protocols onboard embedded systems and accessible via their pins. Sometimes there will be hardware implementations of the protocols in silicon and other times the protocols will be implemented as software libraries called from their main programs:

3.3.3.1 Serial port (RS232)

A really ancient and classic technology called the serial port is still very relevant in real engineering and robotics. Our agri robots probably have about seven of these on them.

The core of a serial port is two wires called **RX and TX**, which stands for receive and transmit. These use digital voltages over time to transmit 0s and 1s, so there is one wire sending information one way and another wire sending information the other way. A historical serial port has many other wires as well, in the old days they were used as controls for many things but nowadays we tend to use RX and TX to send data itself. As with a lot of things in computer architecture, these things have a history, and that history accumulates so sometimes you still have to worry about those other 20 wires being there as part of the standard even though they are not really used for anything anymore. Serial ports can run at **different speeds**, this is something you do very practically in robotics, you have to make sure the thing at one end of the wire is talking at the same speed as the other end of the wire. There are again a few historical conventions to do with **error checking**, how many bits per character, the standard is to send ASCII style characters which are 1 byte each but you don't have to do that. There are bits to do with error correction and there are bits to do with showing where the ends of characters are, if you just received the streams of 1s and 0s you need something to tell you how to chop it up into the individual ASCII letters, you need to know where to draw the boundary to take those 8 bits together so those bits are called stop bits. Probably in the projects you do here you are more likely to see this convention in a virtualised form. So don't so often see a physical serial port on a modern computer, but you do see things which plug in e.g. via USB which emulate the old fashioned serial port protocols. You will often see virtual serial port running over USB.

If you do work with long range robotics like we do you will have to use radio communications to talk to your robot. Your robot might be a mile away and your going to your going to send it motor commands from over a mile away, and you do this using a serial link on a radio frequency. There is a protocol called **Zigbee** which gives you another kind of virtual serial port running over a specific radio frequency for that kind of communication.

Figure 3.7: Serial port voltages over time

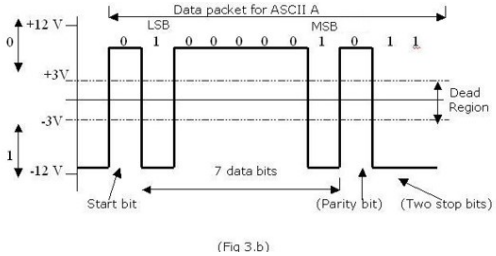


Figure 3.8: A serial port connector

Figure 3.9: I2c bus

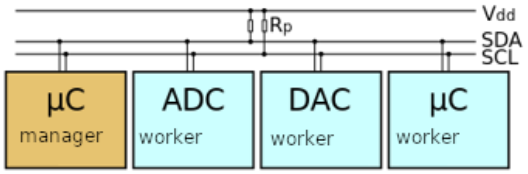
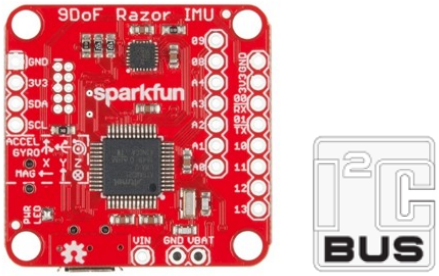


Figure 3.10: I2C device and logo



3.3.3.2 I2C bus

The Inter-Integrated Circuit bus (I2C, pronounced "eye-two-see" or sometimes "I^2C" pronounced "eye-squared-see") is used for connecting chips together, and is very common in robotics. The standard is owned/licenced by NXP (formerly Phillips).

I2C communication is done on just **two wires**: data and clock. It can use 5V or 3.3V as its high voltage, and run in various speed 100kb/s-3Mb/s modes. They may be multiple devices on the bus, each having 7 bit licenced device addresses. One node must take on the role of manager, which generates the clock and initiates communications. Others are workers which reply to the manager. Basic message collision avoidance is implemented by the rule "only talk if the bus is free".

In practice, I2C can be accessed via a standard **FTDI** (Future Technologies Devices International Ltd; cost around 5USD) chip, which provides a hardware and software interface to it, ususally via a serial connection (which itself is usually via a USB port). Hence, no extra device drivers needed – it behaves as a serial port to the user.

3.3.3.3 Controller Area Network (CAN) bus

A vehicle bus is a specialized internal communications network that interconnects components inside a vehicle such as an automobile, train, ship, aircraft, or robot.

CAN is a true serial bus, having a single public serial channel shared by all devices. CAN has no standard connector as it is not intended for use by consumers, but for the internals of vechiles. So usually its wires are soldered directly into the PCBs of the many devices in the vehicle. CAN usually has just 4 internal wires, which use differential voltages (like USB) to protect against strong external electromagnetic fields expected in

Figure 3.11: FTDI interface for I2C

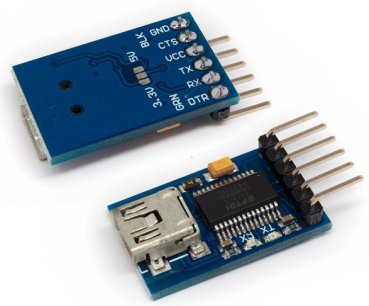
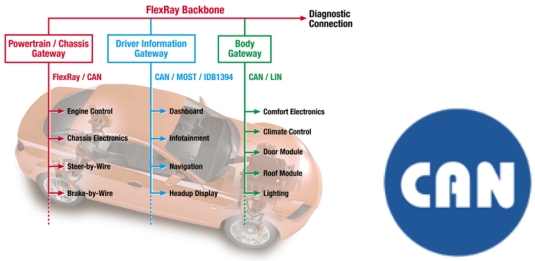


Figure 3.12: CAN bus automotive applications and log



vehicles, especially around electric motors and engines.

CAN security is a current concern. By it nature as a bus, all devices can read and write to it. This may create problems when safety-critical equipment such as anti-lock brakes are connected to the same bus as non-critical devices such as media players. The concern is that security in media and similar devices is typically less rigourous than in in the safety equipment. So for examples, crackers could take control of these devices and use them to send malicious commands to critical devices, or deny service to them on the bus my filling it with junk messages. For new autonomous vehicles where steering and accelleration are also via CAN bus, the consequences could be even worse.

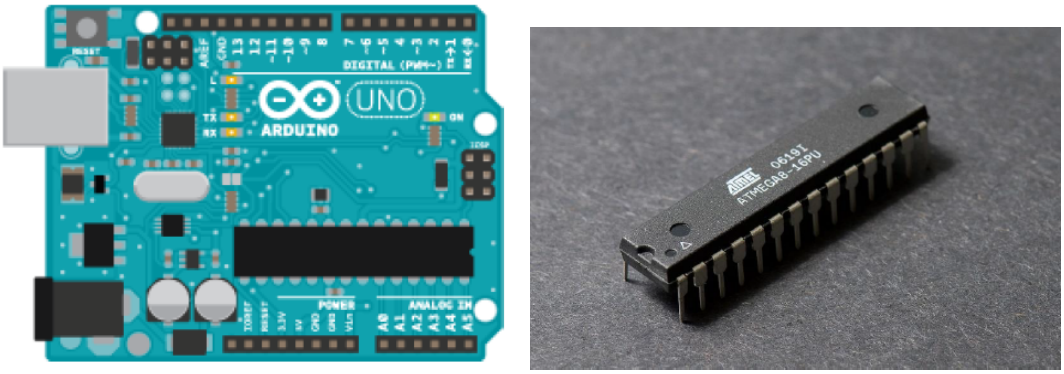
3.4 Embedded systems examples

Modern embedded systems come in two basic flavours: CPU-like microcontrollers and non-CPU-like devices.

3.4.1 CPU-like systems

3.4.1.1 Arduino

Figure 3.13: Arduino board (left) and ATMEGA microcontroller (right)



Arduino is the definitive system for hobbyists, hackers and makers. It is available for a few pounds from some manufacturers. Arduino board is based on the **Atmel AVR family** of microcontrollers, and everything else on the board is basically just I/O. The board design (though not the MCU) is **open source hardware**, so many different manufacturers produce them legally.

The MCU is 8 bit, using a RISC architecture running at around 20MHz, at 2-5V. RISC is commonplace in embedded systems as it's all about being small and simple, this fits very well with the embedded system philosophy (reliability through simplicity).

Arduino is designed specifically to be easy for anyone new to electronics to use and play around with. If you only bought the chip you'd need to do a lot of **soldering**. Computer scientists are usually afraid of soldering so having all of the connectors and I/O ready to connect to power and your PC is a big help.

Arduino has a **Harvard architecture**, the program you send to the board is programmed into ROM using software on your host PC. A USB connector is used to program the device. Your programs is send over a virtual serial port, down through USB, then go through an FTDI converter into I2C bus protocol before ending up in the chip ROM that way. This is a very common way to talk to embedded systems through a host computer.

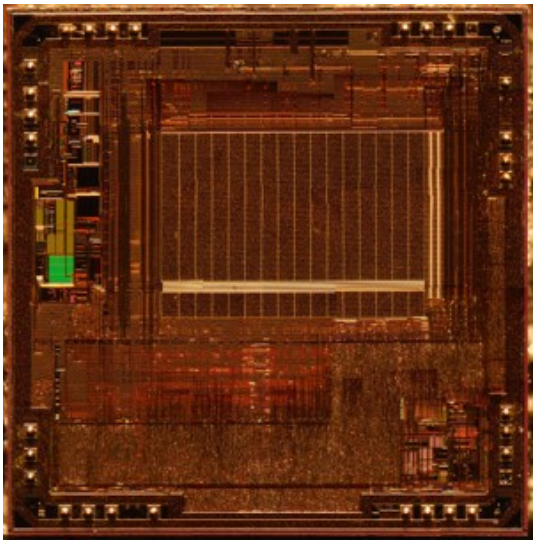
The **I2C bus** also provides the ability to plug in extra extensions to the Arduino. You can get other physical boards ("**shields**") that plug into ports on the I2C bus, in a nice stackable way.

Arduino is famous for having a particularly easy to use development toolchain. Because Arduino is designed as a completely standardized unit of hardware, it has been possible to produce a similarly standardized set of **software tools**. The Arduino tools run on Linux and other OSs and include graphical code editing and debugging and integrated drivers for the USB link to transfer executables onto the device with a couple of mouse clicks.

I recommend Arduino for beginners wanting to get into embedded systems for the first time. Although the board is open source hardware and produced by many companies, it is best to buy a **"genuine Arduino"** starter kit at first because this is made directly by the designers and therefore has a reputation for working at the highest quality. Quality of those by other comapines may vary and require some Arduino skills to determine if a bug is your own fault or the hardware's.

The Arduino's microcontroller, the Atmel ATmega328, looks superficially somewhat similar to an old style 8-bit system like a 6502. It's almost simple enough to see individual bits of what's going on, right on the borderline of visual intelligibility. Unlike the 80's 8 bit systems there's more going on here than just a CPU.

Figure 3.14: Die shot of the ATMega328

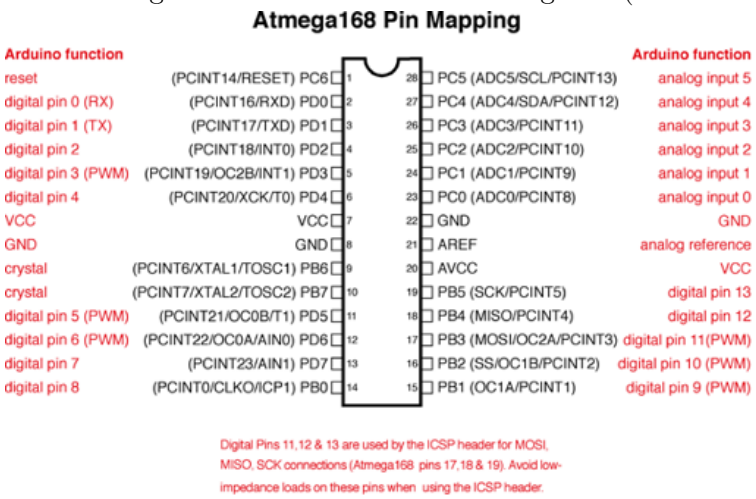


There is a lot of I/O directly on the chip, including digital input and output pins and also analog input via ADC, none of which you would usually find in a 1980's CPU. I/O is thus easy to program as you don't need to use busses, addresses, and **I/O modules**. There is no external memory or bus, you just put that small program onto the **ROM**, the **RAM** memory is all on the chip. It's actually a bit more powerful than an 80's device, you've got 32 user registers rather than 3 on the 6502, allowing you to do more meaty calculations here. 1kB EEPROM, 2kB SRAM and 32kB flash memory. So it's really more comparable to an entire 80's PC rather than just a CPU.

There are eight status bits telling you the result of arithmetic calculations to allow branching. The ISA includes indirect addressing and a hardware stack.

Looking at the pinout diagram then you can see it's very different to a typical CPU as there are **no bus pins**. A typical CPU has many pins dedicated to the address and data lines of the bus heading out to different areas of the system. Whereas here there's just direct inputs and outputs. You can just connect your embedded sensor or actuator directly here and it's all self-contained.

Figure 3.15: Pinout of the ATMega168 (note the lack of bus IO pins)



The most important thing to know about Arduino is that **the same ports are used for both inputs and outputs**. You have to set the mode before using them, this is done by writing to a special couple of registers in memory. There's the Direction register and the Port register. In the direction register if we set the values to 0 the pins are outputs and setting to 1 for inputs. For output we program the port registers with the pin values we want to output (1 is on, 0 is off). You can do this physically with an LED, register and battery. If you set the direction register the other way the same Port register becomes an input register, reading values from this gives us the values from our electrical switches (see right image).

The other important thing to know is that while it has ADC analog input, it fakes analog output using **PWM** which can cause problems for some applications.

Figure 3.17: Ruggedino

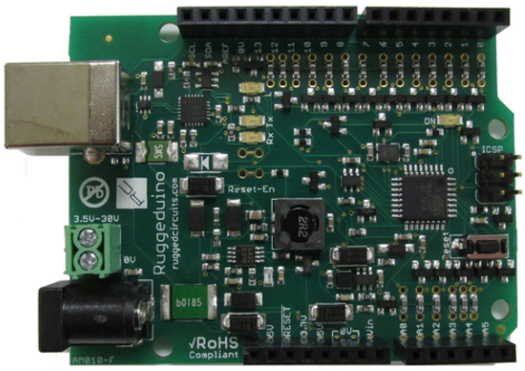
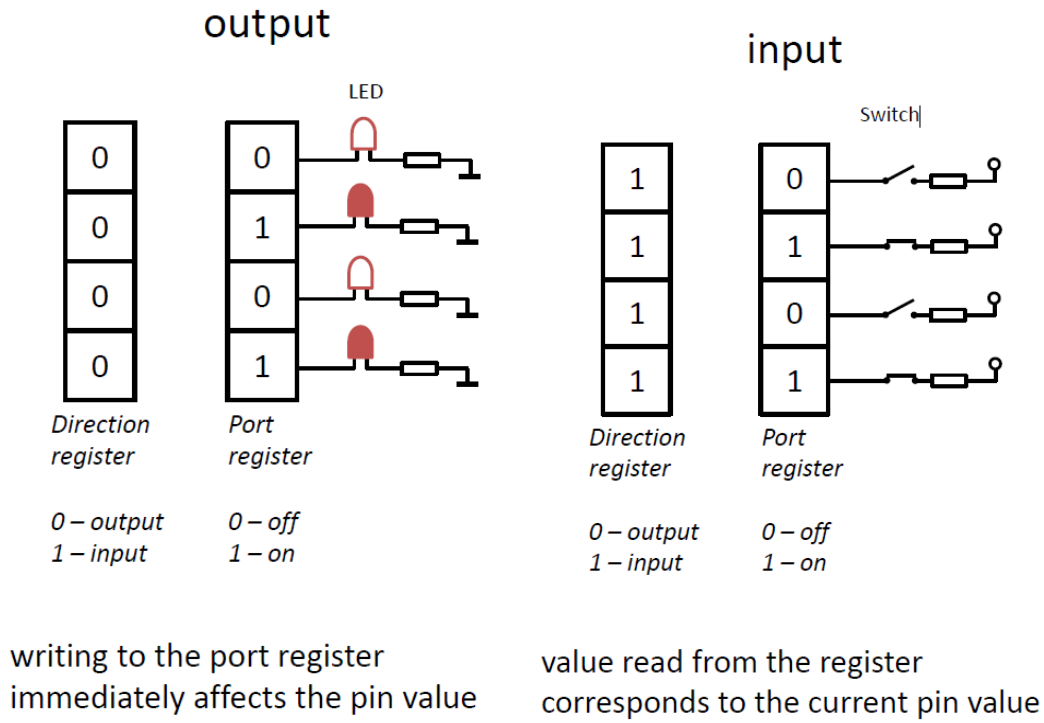


Figure 3.16: Showing the different mode configuration of pins in the Arduino



As it is open source, Arduino has been modified by many designers, for example the **Ruggedino** is a hardened, and thus more expensive, version which includes extra safeguards to prevent you from blowing it up in stupid ways, and manufacturered and certified to higher standards to enable use in more challenging environments. There are also versions with more IO pins and true analog output via DACs.

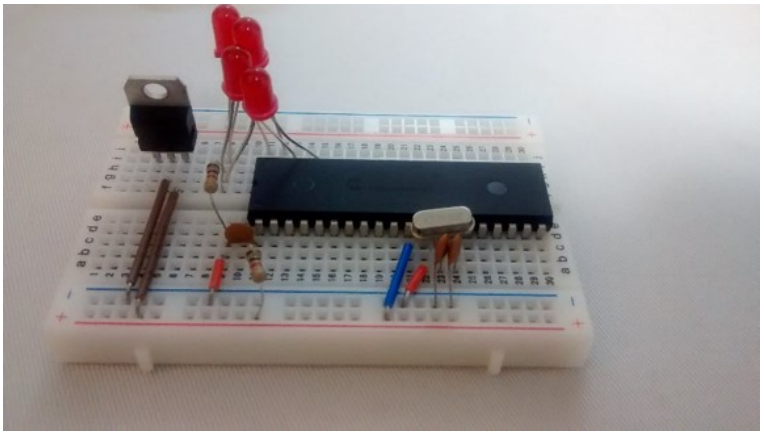
3.4.1.2 PIC Microcontrollers

A PIC is very similar to the AVR microcontrollers, but made by a different company – "Microchip" is in fact a brand name of this American corporation. PICs are found in much real-world industrial and consumer engineering systems, such as your washing machine and your car. There's a large series of them and you decide on which to buy based on the trade-offs between speed, power, cost and physical size. You'll find these probably in most of your consumer embedded devices. If you look in your washing machine you'll probably find a PIC. They're very similar to the AVR but they've come from a different community. More popular with the engineering community, possibly this is just a cultural reason. The Arduino gives us a very nice way of not having to do any soldering, you don't have to burn yourself too much. It's not a big deal for an engineer to find this stuff, having a lot of the required components in a drawer somewhere and they can knock it together with a PIC, program the PIC and be done. Whereas it's nice from most Computer Scientists' perspective to have it all in a nice package when you start off.

If you want to program these you'll need a breadboard; then here is a tutorial for PIC assembler:

http://groups.csail.mit.edu/lbr/stack/pic/pic-prog-assembly.pdf

Figure 3.18: Breadboard using a PIC chip (and other electronic components)



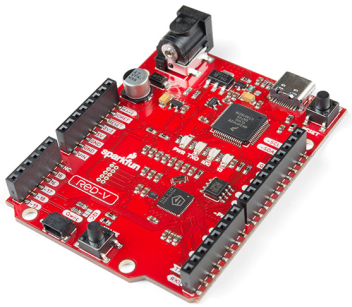
3.4.1.3 RISCV microcontrollers

The Arduino board is open source, but it's MCU and instruction set are not. To take open hardware to the next levels, we need an open instruction set and then an open silicon MCU implementation.

RISCV is an open instruction set. There are now several Arduino-like boards available at Arduino-like prices based on RISCV processors, some with the same form factor and pinouts as Arduino. The first generation of these still have propriatory silicon, such as the SiFive HiFive and SparkFun RED-V. A second generation is currently in progress based on open hardware silicon, such as LowRisc.

There are plugins in progress to make these RISCV boards compatible with ArduinoC and the Arduino IDE, and other projects creating similar IDEs for RISCV boards using regular C, such as Freedom Studio IDE.

Figure 3.19: SparkFun Red-V



3.4.1.4 Digital Signal Processors (DSPs)

Figure 3.20: Example of a Sound FX DSP Chip

Related to microcontrollers are Digital Signal processors (DSPs). These are a more specialized class of embedded system that are specifically for handling real-time signals such as audio/music signals. If you're specialized in that market you're going to put a few different things into your architecture. You're fundamentally dealing with a sequence of continuous valued data like a sound-wave. DSPs are used to process digital signals, such as audio, video, and radar. For example in a guitar effects processor box you'll find several stages like this: De-noising, graphic equaliser, distortion, chorus effects and reverberation (from left to right) with each stage running independently from one to the next. Specifically in this architecture there's not much branching ("if" statements) going on, rather the data just flows through a smooth pipeline from one stage to the next, without branching in the code. DSPs are designed to take advantage of that. Probably

de-emphasising doing clever branch prediction and instead spending that money and the space on the chip that are more useful for that purpose. DSPs use their available silicon to provide additional CISC-like instructions dedicated to signal processing. For examples, **special instructions** for fast hardware filtering and Fourier transforms. Unlike desktop CISC CPUs, these are usually still done in fixed point rather than floating point. If you're building a guitar processor you probably only need to program it when it's being manufactured so Harvard architecture is used here again. Storing the program in ROM makes sense here.

Figure 3.21: Sound FX Processor

3.4.1.5 Raspberry Pi

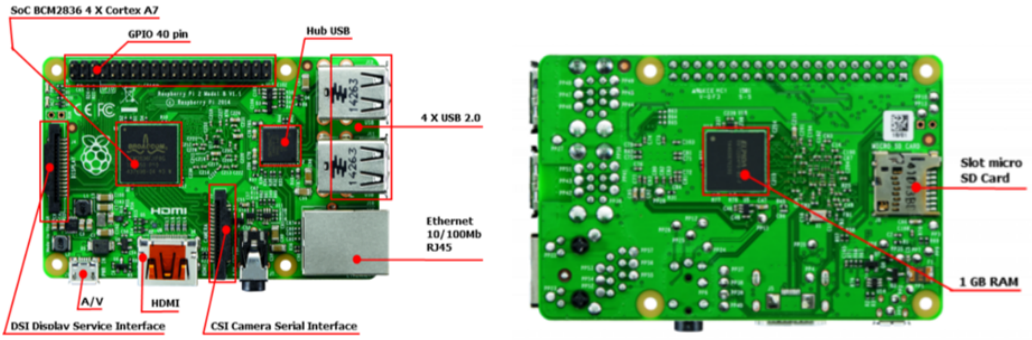
Let's look at some more complex processors. These are also great for on-board robotics and sensor networks. A Pi is more like a very small normal computer than an Arduino. When you work with a Raspberry Pi even though it's a tiny cheap thing it still behaves a lot more like a regular desktop computer. In particular is it intended to run an operating system – usually Linux. This makes programming much easier, but at the cost of losing the safety and reliability guarentees provided by simpler MCUs. If your program runs on Linux (and maybe other middleware such as ROS) then the safety of your system now depends on the absolute bug-freeness of the tens of millions of lines of code in those operating systems as well as yoru own small program. So this is not recommended for safety critical applications such as controlling motors and other potentially dangerious actuators.

Figure 3.22: Raspberry Pi Zero

The Pi has an ARM core on it, it's still a system on a chip, you can see there's just one big chip on here (centre of board). You're basically just buying that chip with the required I/O and A/D conversion on the board. So on this chip you've got an ARM core, that's the CPU part of the chip, inside the core though it looks very similar to a RISC CPU. This is a 32 bit machine, until pretty recently this was the current desktop architecture.

It's running an ARM instruction set, and it runs at around 1GHz which is faster than desktops of the 1990s PC era. It's got a GPU onboard, half a Gb in RAM, a USB and a HDMI connector all on the chip. And the whole thing costs 5 dollars!

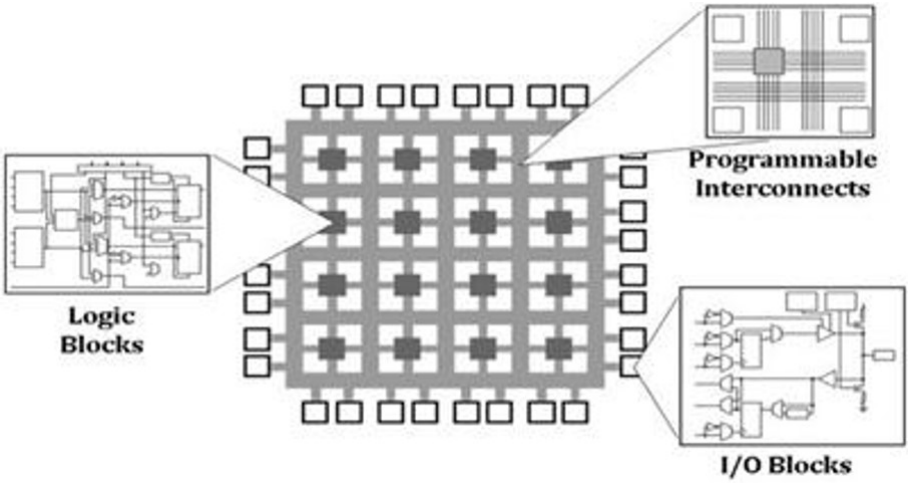
Figure 3.23: Annotated upper and under sides of the Raspberry Pi 2



3.4.2 Field programmable gate array (FPGA)

FPGA is a collection of physically reconfigurable hardware circuits on a chip. You can see there are standard **logic blocks** all over the chip, which can be physically configured to create any small Boolean function using the configuration of the inputs and outputs inside it. We can do that in every one of those logic blocks.

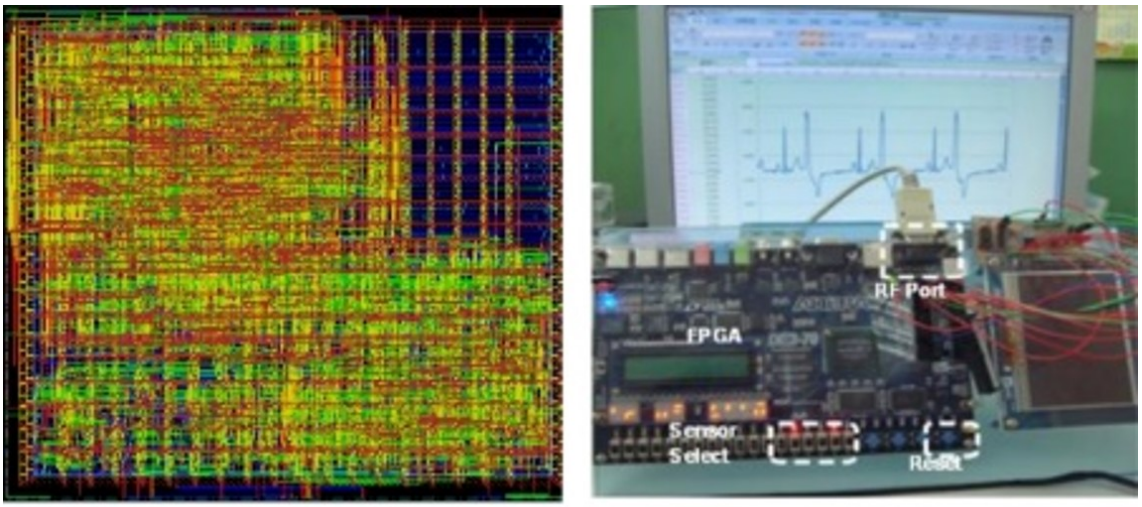
Figure 3.24: Logic level diagram of an FPGA Chip



AND gates with a control line that tells the block whether to make a connection between two wires or not. So this is almost going back to software now, you're not physically changing the chip. People who write these programs are still considered hardware designers but you're not physically blowing the wires anymore.

These are very nice you can buy them on cheap consumer boards. I bought one for around 20 quid I think. (Fig60b) As with the Arduino and Raspberry Pi you typically buy it with a board that helps with the I/O etc. And you can take a look at the state of the chip when it's been loaded with a "program" and you can see it's physically taking up space on the chip (in colours that aren't blue on Fig60a). Again it's not a program like a recipe/ sequence of steps, it's just setting up the logic and connecting the appropriate wires by setting it up with control signals.

Figure 3.25: A display of the logic configurations inside an FPGA (left) and an annotated Image of the board itself(right)



There are only 2 main manufacturers of this now, XILINX and ALTERA (now part of Intel). They make a variety of sizes, the larger of these chips are used for things like neural networks in hardware. However the most common uses for these boards are in Embedded devices and in prototyping chips before photolithography (making a chip mask).

There are compilers and languages like Verilog and Chisel that let you design hardware at a fairly abstract level that can then be compiled in a variety of different ways. It can be compiled into a real chip design (for making masks). But the same design can also be used to describe the logic in your hardware to be put onto an FPGA. This is extremely valuable as you can compile it to run on your FPGA before spending the millions of pounds needed to make a real chip mask. This is how CPU's are still developed and tested.

3.5 Summary

Embedded computers are usually found in robots for low level harware control. Usually has he same basic CPU based as other computers, executing an assembler like language. But designed with more emphasis on the need to operate in the real world, to be cheap, reliable, real-time, and low-power.

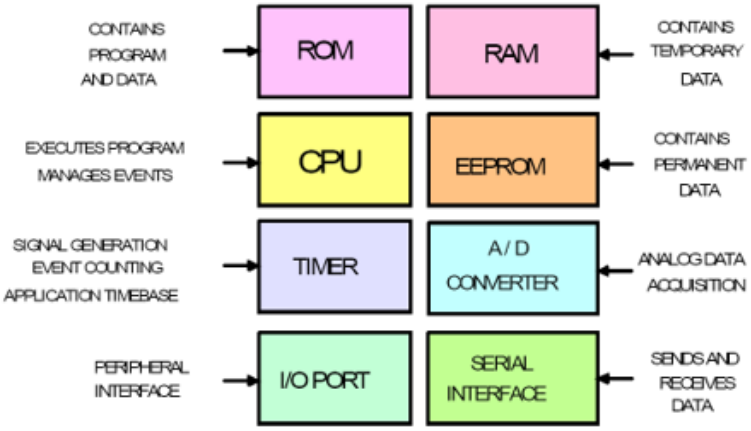
In response to those needs there are specific architectural engineering designs that tend to appear in the CPU: We often see Harvard architectures, using ROM to store the program and RAM to store program data, both of these being on the same chip as the CPU (These are SoC: System On Chip designs).

We often see different choices about data representation, such as ommitting floating point arithmetic operations in favour of fixed point or just integers. DSPs often use fixed

point arithmetic and assume that signals will lie in its range.

We often see I/O and A/D conversion done on silicon, as well as watchdog timers, counters and communication on the chip through slow serial interfaces. If there's any single way to characterise an embedded architecture it's probably finding the I/O ports directly on the chip with no bus coming out of the microcontroller.

Figure 3.26: Diagram showing the components of a typical Embedded System



This is the conventional microcontroller approach. We've just seen this other idea: what if we just use digital logic (ladders, FPGAs). In these architectures there is no program anymore. It's not an ISA anymore and in this section we've seen it as a design for low power embedded systems. However you might ask what if we did this on a larger scale? What if all computing was like this? In a ladder or FPGA system we can do everything at once based on digital values in a form of parallel computing.

3.6 Exercises

- Arduino tutorials from the Elegoo kit, see Blackboard download. Do as many as you can from lessons 1-7.
  - NB: From some Ubuntus you may errors like "Error opening serial port /dev/ACM0 when trying to connect to the device, which can be fixed by giving your user account permission to access it: "sudo chown <username> /dev/ttyACM0".
  - **Take care not to explode the LEDs:** LEDs must be connected together with a suitable resistor and in the right direction. The longer leg of an LED is the positive end. The diode symbol in the circuit has a bar on the negative end (as it prevents current, rather than electrons, flowing backwards). We will learn more about LEDs and resistors in the next session.

3.7 Further reading

3.7.1 Required

- Bolton, W. Mechatronics. Pearson 6th Ed 2015. Chapters 10-12.

4 Electromagnetism for robotics

Electromagnetism occurs in Robotics in several worlds. In the **physical** world, it is a physical theory explaining how robots' motors, circuits, sensors, and communication systems work and how they can be controlled. In the robot's **perceptual** world, the robot may need to perceive and reason about electromagnetic objects and actions. Finally, as part of a robot's action planning, eletromagnetism is sometimes used as a **metaphor**, for example imaging a "virtual magnet" pulling a robot around in order to generate a trajectory. So it is useful to study theories of electromagnetism as a foundation for all of these applications.

Electromagnetism can be modelled in several ways. In a basic form, it can be viewed (by computer scientists) as a "**plugin**" to the Newtonian mechanics model, providing additional ontology and forces. Like all physics theories, this has some limitations which appear in certain settings. Some can be handled within Newtonian mechanics and others may require a large shift in ontology, to a field ontology.

This chapter will introduce:

- Electromagnetism as a folk theory extension to Newtonian mechanics
- Maxwell's equations and field ontology
- Electronic circuits
- Electric motors
- Electronic communitation

4.1 Folk electromagnetism

4.1.1 Folk electric force

Electromagnetism is usually presented using a field ontology, but here we will examine a version based on the object ontology presented so far, which is more in tune with physics engine implementations.

4.1.1.1 Charge Ontology

To each (macroscopic) object, we will assign an additional scalar real-valued property, *charge*, q, measured in units of Coulombs (C). In a basic model for interaction at long distances, we assume that the charge is concentrated at a point such as the center of mass of the object.

4.1.1.2 Charge force

We then add a new force given by Coulombs' law and using Coulomb's constant, $k_e = 9 \times 10^9 Nm^2C^{-2}$,

$$F = \frac{k_e q_1 q_2}{r^2}$$

acting between two objects, where r is the distance between them and a positive force means repulsion along the direction between them.

4.1.1.3 Application: undirected path following

It is unusual to encounter strongly charged macroscopic objects in everyday robotics, but it is very common to use the folk theory as a metaphor during robotic path planning.

Suppose we want a mobile **wheeled robot to follow a particular positional path**, $\theta(t)$, and the robot's actual position or pose is $x(t)$. If we pretend that the two points $\theta(t)$ and $x(t)$ are centers of objects having a positive and negative charge respectively, and that $\theta(t)$ has a very high mass compared to $x(t)$,we can then compute a resulting force which will act to pull the robot from its current position to the desired one. In practice, this can work very nicely and smoothly, as long as the robot has enough "degrees of freedom" to actually move according to the force.

4.1.2 Folk magnetic force

The folk electric force is based on two points having scalar charge which attract or repel. In our perceptual worlds we also observe the presence of magnetic objects. These also attract and repel but cannot be modeled simply by scalar charge. Two magnets each have a vector which describes an orientation, or North and South poles. Two poles of the same type repel one another, and two poles of opposite types attract.

A simple way to model this effect in a folk theory is assume that the **North and South** pole of two objects $i \in 1, 2$ are at locations offset from a macroscopic object's center of mass at some radius d_i at angles $\pm\theta_i$ and strength μ_i. Then add forces of the same for as Coulomb's law between each of the four pairs of poles.

4.1.2.1 Application: directed path following

A nice application of this model is again to robot path planning. The charge model above pulls a robot from a current position to a target position. A magnetic model can pull it from a current *pose* to a target *pose*. A pose is a position plus an orientation. We imagine the robot is an *oriented* magnet, say with a north pole at its front and south pole at its rear. The target pose is defined similarly. Then we compute virtual magnetic forces between a target (of high mass) and current pose (of low mass), and apply them as real forces via the robot's control system to produce smooth motion between the poses.

4.2 Maxwell's equations

The folk electrostatic and magnetism theories above are often very useful in simple simulations and robot perceptual models. But they are not a complete theory of all the phenomena which we observe around electricity and magnetism in the world.

Particular limitations include:

1. If we go and observe a **charged object moving past a magnet**, we will see that the magnetic field puts a new force on the object.

2. If we make a **charged object go round and round in a small circle**, we will see that this acts on magnets as if it was a magnet itself.

3. The electric and magnetic forces in the folk theory **act instantly** between their objects. This would allow, for example, **instant communication** between a Mars Rover and a base station on earth. This would remove the need for automation of Mars rovers, which we observe in the real world to have a several minute response time to any such signals we send from earth. Early Mars rovers were commanded to move just a few centimeters once per minute and driving them was extremely tedious for this reason. The need to give them higher-level commands to perform work for minutes at a time has been a major driver of autonomous robotics in general.

These, and more, limitations can be fixed by moving from the folk theory to Maxwell's theory of electromagnetism.

It is probably possible to express Maxwell's theory in an objects-and-forces ontology but it would be difficult and tedious to do so. Instead it is usually expressed using a **field ontology** via Maxwell's equations (also known as Gauss' law, Gauss's magnetism law, Faraday's law, and Ampere's law):

$$\boldsymbol{\nabla}.\boldsymbol{E} = \frac{\rho}{\epsilon_0}$$

$$\boldsymbol{\nabla}.\boldsymbol{B} = 0$$

$$\boldsymbol{\nabla} \times E = -\frac{\delta \boldsymbol{B}}{\delta t}$$

$$\boldsymbol{\nabla} \times \boldsymbol{B} = \mu_0(\boldsymbol{J} + \epsilon_0 \frac{\delta \boldsymbol{E}}{\delta t})$$

It is beyond the scope of this lecture to go into the details of these. But we will note that **E** and **B** are and vector **fields**. (ρ is charge density, ϵ_0 and μ_0 are constants and J is electric current density).

Gauss's law says that the total electric flux through a sphere is a function of the charge contained inside the sphere.

Gauss's magnetism law says the the total magnetic flux though a sphere is always net zero. For example, if you draw a circle anywhere round or near a magnet with N and S poles, there will be the same amount of arrows coming in as going out.

Faraday's law says that a time varying magnetic field causes a spatially varying electric field. For example you can generate electricity by spinning a magnet, causing electric fields to change which in turn cause currents to flow.

Ampere's law shows how a current causes a magnetic field around it. For example you can create an motor as an electromagnet by making current flow around a coil of wire.

These are functions of 3D space, which given a coordinate in it, return a scalar or a vector respectively. The electric field **E**'s value means how much force a unit charge would experience if it was placed there, and the the magnetic field **B**'s value means roughly similarly for a "unit magnet" placed there,

$$\boldsymbol{F}_e = \boldsymbol{E}q$$

From these equations, the interactions between electrical and magnetic forces and objects emerge naturally. We also see that it takes time – the **speed of light** – for a signal from a moving charge or magnet to arrive at a destination,

$$c = \sqrt{\frac{1}{\mu_0 \epsilon_0}}.$$

Furthermore, light itself can then be modeled as being such waves moving along the two fields together. (Maxwell's theory was originally intended only to describe electricity and magnets, but this theory of light dropped out of it "for free" as a side-effect, in one of the greatest moments in the history of Science.)

Field ontology is very different from objects-and-forces. Rather than our model of the world containing a few discrete objects with properties and relations, its space is now filled with continuous, vector-valued fields. There is no such thing as a "vacuum" because the fields are always present in this ontology.

We may also think of **gravity** under a field ontology, in which a gravitational field gradient causes forces in masses.

Working engineers will freely mix object ontologies with field ontologies in their calculations. They intuitively know when to switch between them to avoid contradictions in their reasoning. This mental juggling can be very complex but intuitive – how to think of forces as "real" at one moment, with the fields just describing what real forces would apply to particles in counter-factual worlds, and then switch to fields as "real" and the forces emerging from them.

Physics simulations tend not to use field ontologies unless they are specifically built to model some electromagnetic aspect (such as radio communications). Building AI systems to try to replicate the human engineer's reasoning forms an interesting case study of how AI should to manage contradictions and models in general – it is still an open research question how to do this well.

4.3 Electronics

4.3.1 Ontology

The electromagnetism we have seen so far exists in continuous 3D space, where distance is space is important as the field strengths vary with it.

When we work with electronic circuits, space becomes less important.

Some materials, conductors, such as metal **wire**, allow small charged **electrons** to flow, to an approximation, completely unencumbered through them. Hence, if we alter the electric field at one end it will be transmitted almost instantly to the other end, regardless of physical distance.

Other materials, such as air and wood, are insulators which to an approximation, do not allow electronics to flow through them at all.

So we can consider a **topological ontology** for electronics – one in which physical space and location are unimportant but what matters are the connections between objects.

4.3.2 Water analogy

When working with electronics circuits, it is often useful to make use of the water analogy. There is a precise, formal analogy between what happens in an electric circuit and what happens in an analogous hydraulic (eg. plumbing) water circuit:

electrons $\Leftrightarrow$ water

voltage $\Leftrightarrow$ height

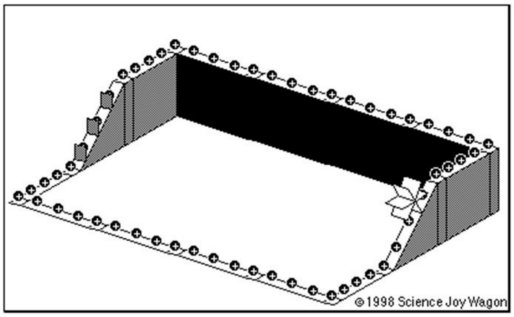
The following circuit represents a power supply (electric battery or water pump) and a load (hydraulic or electric motor). The power supply pumps electrons (water) up a **voltage** (height) gradient (both voltage and height are forms of **potential energy**). The electrons (water) move around the circuit and when the reach the motor, then fall through it, transferring their energy to the motor and landing in a lower potential energy region.

(If the wires did not form a circuit, then the electrons (water) would physically accelerate as they go down the potential gradient and decelerate as they are pumped up it. But because of the circuit form, they must all move at the same speed, and energy lost and gained at the gradients must go out to or be supplied from the external motor and pump.)

The **current** is the amount of electrons (water) passing per second at a point. (By historical accident electrical current is it given a opposite sign to electron flow, so electrons flowing east are considered as a negative current in the east vector direction.) Write current as the flow of charge per second, as

$$I = \frac{dq}{dt}$$

In terms of field ontologies: the electric field is analogous to the gravitational field, and may be viewed as the source of the potential energy.



A motor is just one of many possible devices or "loads" which can be placed in the circuit to convert electrical power to something else, such as motion, heat and light. The **power** drawn relates to the current flowing and the resistance R (units: Ohms, Ω) of the load,

$$P = I^2 R$$

where resistanceis defined by the current observed to flow across the device when a voltage difference is placed across it,

$$R = \frac{V}{I}$$

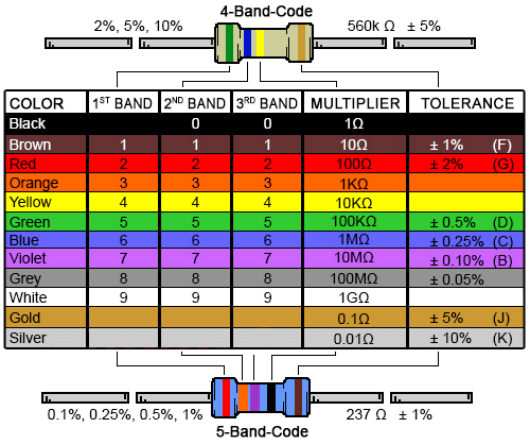
when loads R_i are placed in series in a circuit, the total resistance R is observed to be

$$R = \sum R_i$$

and if placed in parallel it is observed to be

$$\frac{1}{R} = \sum \frac{1}{R_i}.$$

Resistors are mass produced for use in electronics and are painted with a standard 4 or 5 stripe color code to show their resistance and tolerance.



COLOR	1ST BAND	2ND BAND	3RD BAND	MULTIPLIER	TOLERANCE
Black		0	0	1Ω	
Brown	1	1	1	10Ω	± 1% (F)
Red	2	2	2	100Ω	± 2% (G)
Orange	3	3	3	1KΩ	
Yellow	4	4	4	10KΩ	
Green	5	5	5	100KΩ	± 0.5% (D)
Blue	6	6	6	1MΩ	± 0.25% (C)
Violet	7	7	7	10MΩ	± 0.10% (B)
Grey	8	8	8	100MΩ	± 0.05%
White	9	9	9	1GΩ	
Gold				0.1Ω	± 5% (J)
Silver				0.01Ω	± 10% (K)

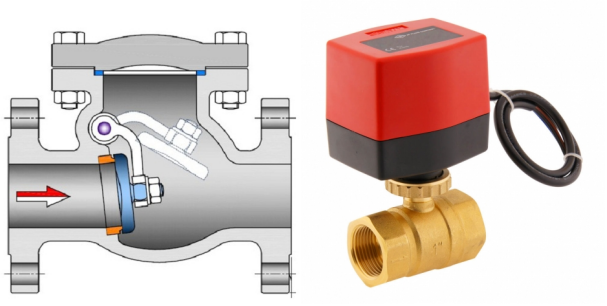


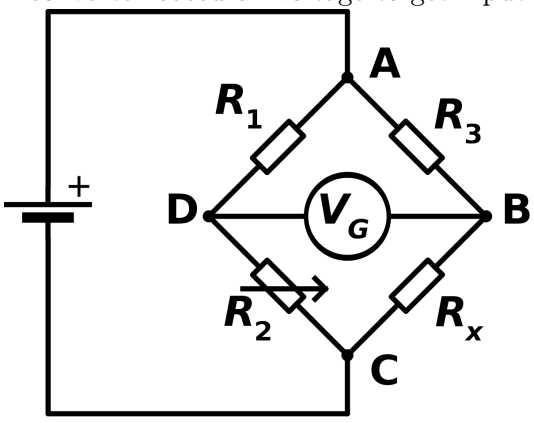
Figure 4.1: One way valve; electric one-way relay valve.

4.3.3 Simple analog machines

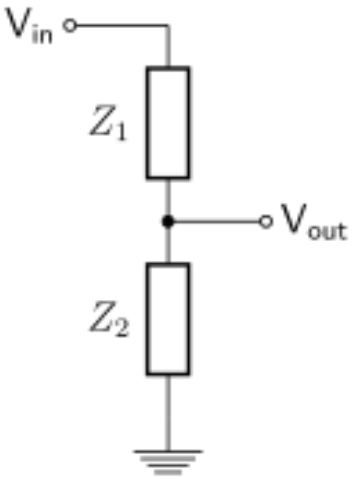
A **diode** is a device which enables current to flow in one direction only. It is analogous to a one-way water valve as found in plumbing. Such water values operate using a spring loaded trapdoor which the water can push open in one direction and not the other. A light emitting diode (LED) is a diode which emits light as a side effect, commonly used as a visual display. (LEDs tend to explode if if don't put them in series with a resistor.)

A **transistor** is an electrical value which can be turned on and off by a control current. It is analogous to the electromechanical water relays found in plumbing systems to turn hearing on and off. Transistors are the building blocks of CPUs and other integrated circuits (chips).

A **Wheatstone bridge** is a simple circuit that converts changes in resistance to changes in voltage. This is useful is you have a sensor which measures resistance and an AD converter based on voltage to get input to your computing (see Bolton for maths):

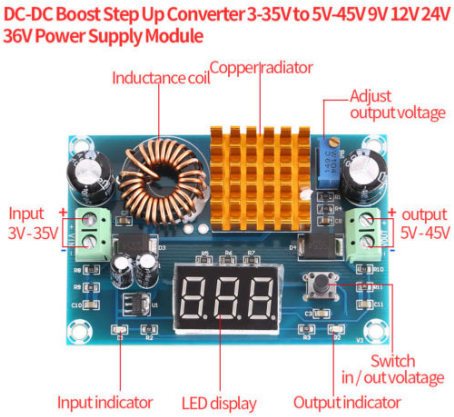


A voltage divider is a simple circuit which creates a lower voltage than its input. For example to convert a 12V power supply (often available on robots and vehicles) into a 5V one (used by many embedded systems):



To convert voltages upwards is harder and requires an amplifier. Op-amps (operations amplifiers) are standardized amplifier components which can be wired in a variety of ways do perform basic amplification and other signal conditioning (such as integrating and differencing signals, inverting and subtracting signals).

To convert power supply voltages on robots you usually want to buy a cheap "DC-DC" converter for a few pounds on ebay. "Step up" converters increase the voltage, "buck" or "step-down" converters decrease it. Modern converters often have the ability to set the desired voltage within some range so can be reused for different applications:



4.4 AC circuits

The electrical circuits we have seen so far are DC (direct current) which means that the electrons move round the circuit in a single direction. Each electron goes around the whole circuit. AC (alternating circuits) are different. In AC current, each electron remains pretty much where it started, but the electrons vibrate back and forth rapidly and by a very small distance. This oscillation is another way to transmit power around the circuit. It is similar to a sound or water wave as the constituent parts do not move

around in bulk, only the wave moves around. As with DC circuits, they can be imagined via the hydraulic analogy.

AC circuits can be modeled using **complex numbers** to represent the amplitude and phase of the wave. Complex valued versions of "resistance", called more generally "**impedance**", include capacitors and inductors which affect the motion of the wave rather of the underlying electrons. A capacitor can be imagined as a large rubber sheet which completely blocks the flow of water in a circuit, but allows vibrations to travel though it. An inductor can be imagined as a large heavy waterwheel which requires time and work from the water flow to get it moving, then continues to spin under its own angular momentum.

AC is used to transmit mains power more efficiently than DC in the national grid and in homes (using a 50Hz frequency which can be heard as a hum in some equipment). In the grid, and in large scale industrial factory robots, *three-phase* AC is used for efficiency, formed of three separate AC cables which are out of phase with one another. Small scale robotics will not make use of AC, instead using a transformed DC version of mains, or DC from batteries onboard a mobile robot.

If you are working at the level of mechantronics presented here then you should not be touching AC systems including mains electricity at all. In most countries including the UK it is illegal for uncertified people to work with mains. So always ask an "adult" (your technician, certified electrician or electrical engineer) to help if you need to work with AC.

4.5 Electromagnetic Actuation

"**Drivetrain**" or "drive" means everything between the engine and wheels of a vehicle, e.g. including shafts, gearbox, axles, motors.

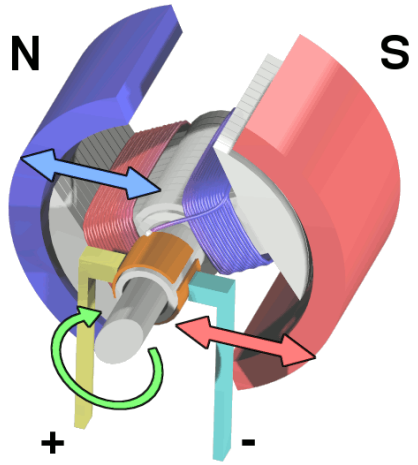
"Powertrain" means the drivetrain together with the power source, e.g. an engine.

As computational roboticists, you are unlikely ever to design actuators but you will very likely spend a lot of your career working closely with people who do and it is useful to be able to talk to them a bit in their language. Most of your robotic commands are ultimately actuated in the world via motors, so it is useful to have some understanding of how motors work under the hood at the end of your programs.

4.5.1 Types of Motors

4.5.1.1 DC Brushed

DC Brushed motors use physical contacting brushes, alternating positive and negative sides, to make current flow in opposite directions as the motor rotates. The **stator** is two fixed permanent magnets, north and south poles. The **rotor** (generic name for rotating part in any motor) is an **armature**, a rotating wire loop which makes and breaks contacts via brushes.



(figure: Wikipedia, Brushed motor)

We apply a (DC) current to the armature. The mechanical brushes then swap this current's direction twice within each rotation.

Applying (via a motor driver) current I at V voltage = power P.

Power causes of is supplied by rotational acceleration (torque, τ) of the rotor, at angular velocity ω,

$$P = IV = I^2R = \omega\tau$$

(This equation related to gearing, as on a bicycle, showing how power can be used "more torquey" or "more fast").

As the (V, I) input causes torque rather than speed or position, *control* is needed to obtain desired rotational speed or position.

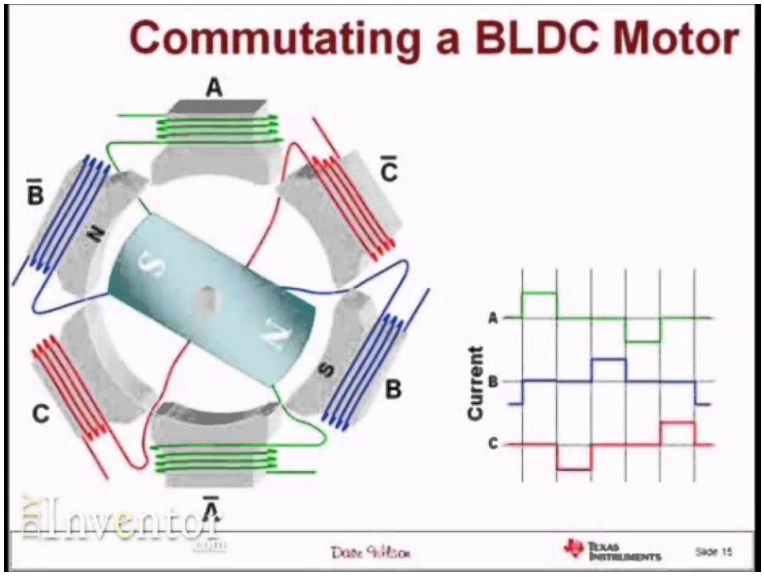
Brushed motors are **cheap** and **simple**, but the brushes **wear out** and must be replaced.

4.5.1.2 DC Brushless

A brushless motor avoids the problem of contact braking and breaking brushes that wear out. It replaces their mechanical role in swapping the current direction with an electronic version. This is more complex to build but then more reliable.

We swap the role of the stator and rotor: the rotor is now the permanent magnet, and the stator is made of several coils of wire. These coils produce magnetic fields. Current in the coils is reversed during the rotation of the rotor, using a (eg. Hall effect) sensor to monitor the rotor's position and choose the direction needed to make it accelerate or decelerate as needed.

ODrive (https://odriverobotics.com/) is an open source hardware brushless driver which does this.



4.5.1.3 AC motors

AC motors are fancy and expensive, based on AC power. They require complicated controllers as with brushless motors, to manage synchronization and feedback issues. Efficient design is still something of a research area.

4.5.1.4 Gear motors

Gearboxes can be added to motors to change the torque to speed ratio (as with bicycle gears). Some "geared motors" have a gearbox built into the same casing as the motor, making them easier to mount on robots.

Motion Control Products MCP Series are large geared motors suitable for mobile agri-robots with gearings around 50:1.


MCP series

4.5.1.5 Encoder motors

"Encoder motors" have built-in sensors of various types (see Sensors chapter) which provide a feedback voltage telling us where the motor currently is. Without such monitoring we don't know what the motor is doing – when we send power we know that it will try to accelerate the motor, but this will often be done again a load on the motor, such as friction if driving a wheel, or weight, if lifting an object.

4.5.1.6 Hub motors

Hub motors are brushless DC motors which are designed to fit inside the hub of a wheel. Hub motors are mostly produced to retrofit bikes to make them electric, but are also useful in agricultural robots such as Bosch's Bonirob where the wheels are required to be on legs to minimize contact with the crops. Hub motors are available on Alibaba for 50GBP. Like all **brushless** motors they require quite complex feedback timing controllers.

4.5.1.7 Linear actuators

Are electric motors with a worm gear that converts the rotation to linear motion (just like Technic Lego!).

Linear actuator to automate steering of Lincoln's pod car.

They are also available in servo and stepper forms: eg. search ebay for "linear servo" (e.g. 300GBP); "linear stepper", and/or supplied with built in potentiometer sensors for doing your own feedback control. (Search ebay for "linear actuator potentiometer" or "linear actuator encoder", e.g. *http://www.phidgets.com/products.php?product_id=3572* , 120GBP). Usually, these pot sensors are rotary, and are geared down from the shaft (which is itself is geared down from the motor.)

4.5.1.8 Hydraulic actuators

As an alternative to electric motors, is is possible to use hydraulic (fluid) pressure to drive both linear and rotary actuators. A power source such as a diesel engine (or electric motor) is first used to pressurize the fluid in a tank. Then the pressurized fluid is released under control into the actuators to make them move.

Linear hydraulics are used in agricultural and construction vehicle machinery.

Rotary hydraulic motors can also be made by forcing the fluid to push a rotating device.

Open Source Ecology (OSE) uses commercial hydraulic drives for everything in its fully Open Source hardware ecosystem, which includes a complete open source tractor, LifeTrac, which anyone can put together. It uses hydraulics because they are physically simpler and easier to service and replace than electrics, which is important in developing countries in particular. OSE is currently working on an open source design for a hydraulic motor itself.

Recently hydraulic hub motors have also appeared (eg. hydrapac.net).

LifeTrac tractor with hydraulic loader. (From https://wiki.opensource

4.5.1.9 Pneumatic actuators

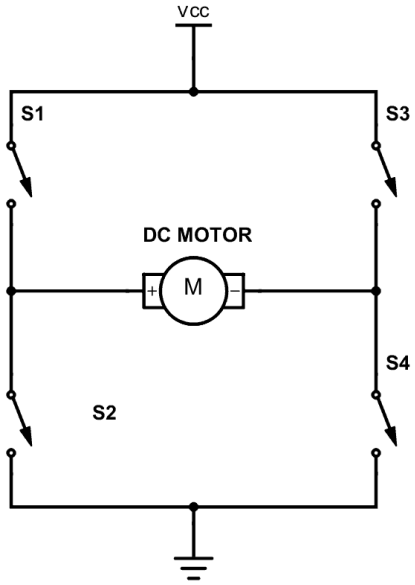
Pneumatic actuators are similar to hydraulics but use compressed air instead of fluid. They are found in *Robot Wars* and *Walking With Dinosaurs* live shows where very high power is needed but for very short periods of time, such as a three minute combat robot battle.

Soft Robotics is a subfield of Robotics research, often using pneumatics to inflate, move, and shape-change various flexible materials. For example, tentacle-like actuators and haptic feedback systems. *FlowIO* (*www.softrobotics.io*) is a recent open source hardware project which aims to be an Arduino-like cheap and multipurpose pneumatic power source, controller and sensor interface for soft robotics projects.

4.5.2 Drivers

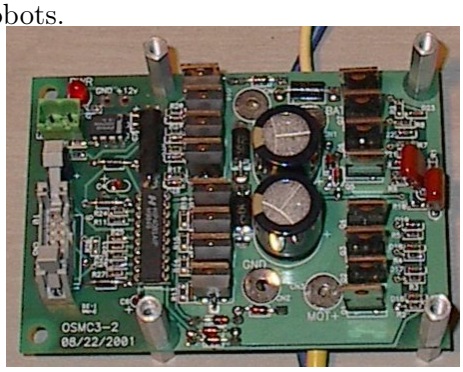
A driver is basically an amplifier, like an audio amp. It takes a small command voltage or binary signal (eg. from a serial port protocol), specifying the desired motor speed, and outputs serious power to the motor. (Software "device drivers" are named by analogy to them.)

Often you also want to send a second signal to say which *direction* you want the motor to turn. A simple standard circuit known as a *H-bridge* is then included in the driver to flip the direction using four switches. The name is because the circuit takes the form of a letter "H" as below. (Whole drivers are sometimes informally called "H-bridges" after this component, though it's still the amplifier doing the main work.)



If you just want to specify how much power to send to the motor then driver is all that is needed.

OSMC ("Open source motor controller" [*sic* - it is a driver not a controller!]) is a fully open source hardware driver with H-bridge, of suitable size for typical RobotWars style robots.



OSMC motor driver

Fancier drivers like Dimension Engineer's Sabretooth 2x25 (*www.dimensionengineering.com/products/sabertooth2x25*) have extra features like:

- safety power management - cut off power if about to fry the motor - via thermal and current sensing.
- regenerative charging (turning kinetic energy from brakes back into electrical po-

tential energy, rather than heat)

- lithium battery cutoff (safety feature, prevent catching fire)
- digital control input protocols, eg. RS-232
- managing fast changes in commands, eg between fwd and reverse, which can damage things
- "has independent and speed+direction operating modes, making it the ideal driver for differential drive (tank style) robots and more."

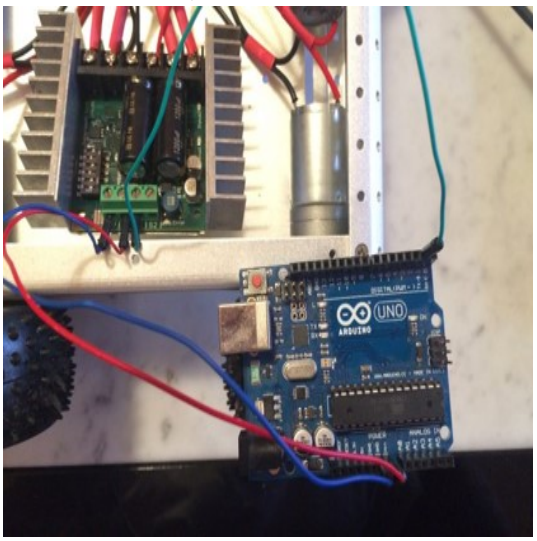
Another example is the SyRen motor driver, 100USD, also made by Dimension Engineering.

4.5.3 Controllers

A driver sends power, which causes rotational acceleration (torque) against whatever the motor's current load is. To choose power to send to achieve a certain motor speed for this load, you need a controller, and usually some closed-loop feedback from sensors on the wheel,shaft, or motor. Controllers usually takes as their command input a either a **desired motor speed** (e.g. for controlling wheels), or a **desired motor position** (e.g. for controlling robot arms).

This is usually done using a microcontroller such as a general purpose Arduino, or a dedicated board (eg. Pololu).

Figure 4.2: Motor Controller Arduino board (lower-right) attached to a driver (upper-left) controlling the power input to a motor (upper-right)



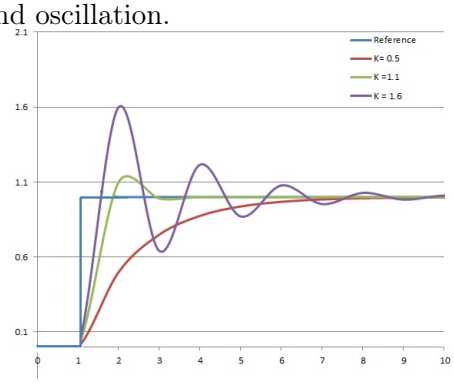
Usually these controllers implement **PID control** (proportional–integral–derivative).

P is like a spring force. It measures how far from out target value (eg. position, speed) we are, and applies a force proportional to the distance.

I is an integrated spring force over time. Its strength is proportional to the sum of the errors over a short time, rather than just the present error.

D is like a spring damping force. It acts against the rate of change of the error term, as a "brake" or "damper" on the system. It is useful to prevent overshooting and oscillation.

In theory there is a complicated theory of how to set these parameters optimally. In practice, it may be implemented into a software tuning tool, or often the designer just applies trial and error with a bit of intuition to find values which "behave". Behaving means getting to where you want to go reasonably quickly but while preventing overshoot and oscillation.



PID controllers with different settings.

Servo motors - are motors with encoder and (usually PID) controller all built in.

(for Sabertooth driver, there is a free Arduino code controller on Sabertooth website. Sabertooth has an optional add-on called Kangeroo x2 Motion controller using self-tuning PID. 24GBP, *www.dimensionengineering.com/products/kangaroo*.)

(There is PID control build into Gazebo 8. Create a revolute joint, pull out the right hand side menu, change from Force to Velocity tab. Here we can set the desired wheel speed and also the PID parameters used to achieve it.)

4.5.4 Stepper controller

Stepper motors are designed to have a discrete set of configurations which "step" into place, and their controllers count the number of these steps precisely. They are a discrete alternative to continuous servos.

Servos are usually better suited to high speed, high torque, high power applications, with dynamic changes in load. Steppers are cheaper and better suited to lower power and load-holding systems.

4.5.5 Arduino control

Some video examples of Arduino and drivers for motor control can be seen at:

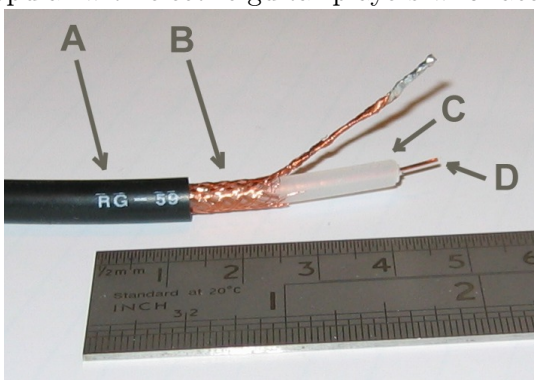
www.youtube.com/watch?v=OW-Bf3yjUyE

pharos.ece.utexas.edu/wiki/index.php/Controlling_the_SyRen_25A_Motor_Controller_Using_an_Ar

(Arduino commands given to the driver are: ST.motor(s) where s in 0-255 speed.)

4.5.6 Motor Interference

In practical robotics, **interference** in the electromagnetic fields is a major cause of robot bugs. For example, a large motor on a robot will make large changes to the local fields, which will affect any sensors on the robot which rely on these fields. The problem can affect electrical wires as well as the sensors themselves, as Maxwell's equations show that current is induced when the field changes. Special cables are available to reduce this effect by shielding or co-axial signals which cancel out external field effects. (Also popular with electric guitar players who face similar interference challenges.)



4.6 Electromagnetic Communication

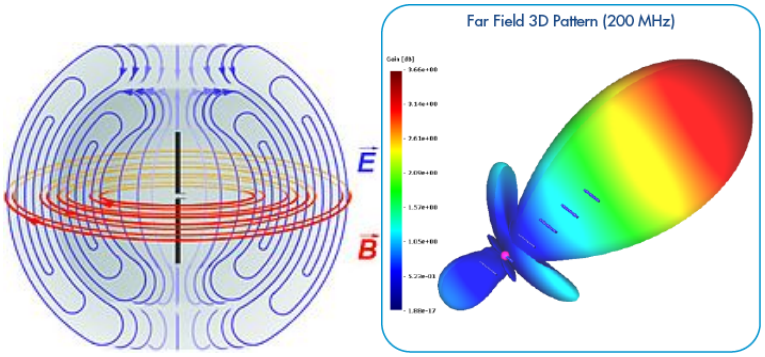
Maxwells equations give rise to propagating waves in the EM fields, which can take any frequency. This electromagnetic (EM) spectrum allows communications on different frequencies to exist simultaneously and independently, at the speed of light. Visible light is a small part of the spectrum. Much of the spectrum is used for radio frequency (RF) communications, such as for mobile robots and medium and long distances from a base computer. Range and bit-rate depend on frequency, power of transmission as well as geometry of the ground and antenna. In theory arbitrary rates are achievable but as the spectrum is a shared, public resource, there are tightly controlled legal limits on powers and frequencies that may be used. High rates are thus available but at high monetary licensing costs (eg. the 3G spectrum auction raised billions of pounds by selling the rights to use a set of new frequencies across the UK.) Radio communications is a thus complex subject which involves both physical and legal constraints.

From Maxwell's equations, the **Friss equation** for radio transmission may be derived which gives the ratio of transmitted P_t and received P_r power for a wavelength λ at range R, in free space:

$$\frac{P_r}{P_t} = G_t G_r (\frac{\lambda}{4\pi R})^2$$

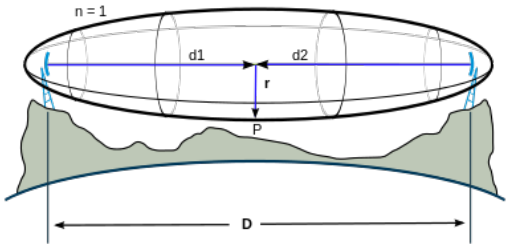
where G_t and G_r are the "gains" of the transmitter and reciever antennae. Gain can be created by electrical amplification and by the use of shaped beamforming antennae such as the **Yagi** shaped antenna commonly used for TV reception. Beamformed antennae

Figure 4.3: *Dipole and beamformed Yagi antenna fields*



focus the beam in a single direction at the expense of the other directions. (A single wire antenna produces a donut shape of waves around it; a Yagi combines many of these held in a row to focus in one direction.)

This is an inverse square law, and depends on the frequency, with low frequencies (longer wavelengths) going further. For example, Long Wave AM audio radio crosses oceans, and Extremely Low Frequency comms (eg 30Hz) are used with giant underground antenna to command nuclear submarines all over the world. Egli model This Friis equation holds in free space, eg used for communicating between satellites and interplanetary probes. However it is not a good model for ground based comms due to the effects of the surface of the earth. The **Fresnel zone** describes how different frequencies interact with the ground, and require antennae to be raised up away from it. The effect is bigger at lower frequencies:



The **Egli model** makes assumptions and approximations about the Fresnel zone to modify the Friss equation to:

$$P_R = G_B G_M (\frac{h_B h_M}{d^2})^2 (\frac{40}{f})^2 P_T$$

which includes the heights h of the antenna. This is why transmitters are placed on large towers – it's not just to get line of sight.

At low "surface" (or "groundwave") frequencies (up to about 3MHz, but continuous effect) , spreading of a beam is significant, meaning that waves can follow the curvature of the earth, including following curves of hills and valleys on farms. At higher frequencies, beams retain their shape, for example microwaves travel in straight lines enabling focused

line-of-sight beams to connect mobile phone towers at high bit-rates.

Reflection occurs at joins of different types of surface (as in acoustic and light reflection). Different frequencies have different reflection properties, for example wall and ground penetrating radars are chosen to have frequencies that pick up interesting properties. Generally, low frequencies penetrate more things. (eg ELF penetrates the ocean to talk to subs.) Higher frequencies are more difficult to work with, eg requiring higher clocked electronic circuits running at the actual radio frequencies. Unlike audio work, this gets seriously hard. eg. A radio beam-former DSP needs to be running at GHz like an Intel processor. (Note that infra red and visible light are a bit higher than microwaves; also that microwave type frequencies can heat and burn you if you stand in their beams, just like a microwave oven.)

The **Shannon-Hartley theorem** links the analogue Friss-Egli equations to digital information theory,

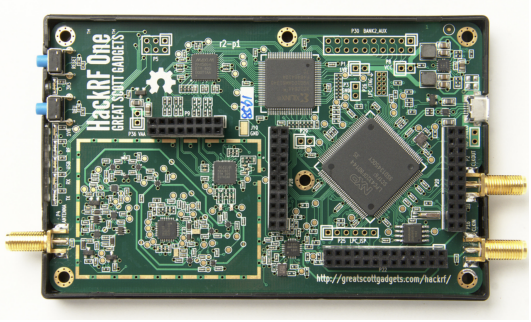
$$C = B \log_2(1 + \frac{S}{N})$$

for bandwidth B, channel capacity C in bits, and S/N are the signal to noise powers. This is independent of the frequency used, although the Signal power is dependent on this, with lower frequencies having more received power.

Due to environmental and technical limits, typical communications applications use frequencies from 30MHz-30GHz. As these frequencies are much higher than the data rates (eg. 50kHz max for music and speech), data about whole bands of audio frequencies may be contained in a single MHz or GHz frequency, as in AM and FM radio. Lower frequencies are used for lower bit-rates over long distances (e.g. submarine Morse code) because they travel through water and other obstacles, and follow earth's curvature; high frequencies for higher rate localized comms (eg. home wifi) because they modulate with higher bandwidths and carry more information. Diffraction says that waves can bend around obstacles smaller than their wavelength. So long waves (eg. 252m music radio) can bend over hills (eg. 100m). Higher frequencies such as GHz only go in straight lines past such obstacles, leaving a radio shadow. GHz as used in domestic wifi is a form of microwave, which is absorbed by water (as used in microwave cooking) so annoyingly is absorbed by trees whose leaves contain water.

Software Defined Radio (**SDR**) is now replacing traditional analog engineering of radio comms systems, and allows almost any frequency or multiple frequencies to be transmitted and receieved via Computer Scientists' programs rather than by adjusting physical hardware. It is the FGPA of radio comms. The **HackRF** is a cheap opensource SDR. Its transmission power is deliberately limited but can be easily amplified using radio amplifiers.

Figure 4.4: HackRF



Policing of spectrum use by the UK **OFCOM** authority is reactionary, triggered by complaints from other users of interference. OFCOM will send antenna vans out to investigate and fine offenders. To use radio communications more powerful and information carrying that wifi generally requires large payments to OFCOM to gain exclusive access to parts of the spectrum. Mobile phone companies are typically the heaviest users and buyers of spectrum from OFCOM, and sale of spectrum to them has brought in large incomes to governments.

4.7 Remaining theory limitations

Mixing Newtonian mechanics with Maxwell fields and forces is not completely straight forward. Human physics students gain intuition about when to use each theory. Some of the issues which would arise in trying to axiomise the theories together, eg. in an AI system, include:

1. Folk object ontologies assumes charge is a scalar property, concentrated at center of mass rather than spread out.

2. Folk object ontologies assumes non-point atom particles. e.g. macro objects. Yet when thinking about circuits we had to introduce the idea of electrons moving around.

3. The folk forces of friction and collision can be explained by lower level physics models, as electromagnetism between particles making up macroscopic objects.

4. Folk forces are transmitted instantly but Maxwell forces take time to travel, when should this be modeled?

5. In Maxwell's theory, a small region of space and time can contain an infinite amount of information. Imagine a millimeter cube of space for a second in the room your are sitting. This contains all of Radio 4, Radio 1, nuclear submarine launch codes, police radio, and an infinite number of other frequencies, all storing information. Computer design could be made much more powerful if we could fit infinite memory into such a small area.

6. Waves in the electromagnetic field also form a theory of light. It shows how light spreads out over space like a wave-front. However this does not explain how lasers are able to focus their energy in non-spreading beams such as usd in robotic lidar sensors. (Quantum mechanics is needed for this).

7. Modern physics (the standard model) further unifies electromagnetism with other subatomic forces, in a purely field ontology having no objects at all (quantum field theory). However unifying this model again with gravity forces remains an open question (e.g. attempted in string theory).

4.8 Exercises

- Electromagnetics questions from Open University "are you ready for Electromagnetism" question sheet; and circuits worksheet. If this is too easy then please help other students to develop your team skills. If it is hard, use standard engineering maths textbooks and your colleagues to help.

- Arduino tutorials from Elegoo, lessons 9 (servo motors), 21 (DC motors), 22 (relay), 23 (stepper motor).

4.9 Further reading

4.9.1 Required

- Bolton, Mechatronics, chapters 3 (signal conditioning); chapter 9 (motors).

4.9.2 Recommended

- Ross, Shamieh and McComb, 2009. *Electronics for Dummies.* Wiley. (basic/catchup).

- Horowitz and Hill, 2015, *The Art of Electronics,* 3rd Ed. (This has been the classic introduction and advanced reference to electronics for many years so the 3rd edition update was a big event in 2015.).

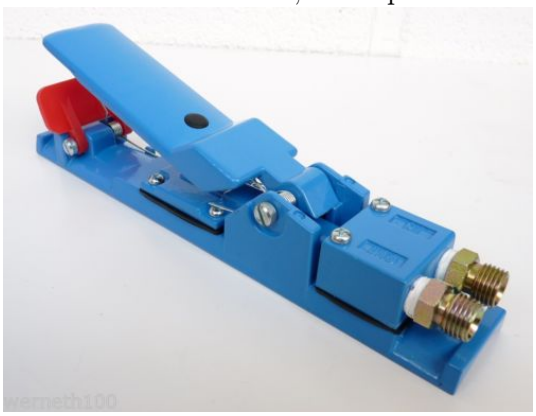
5 Sensors

5.1 Basic sensors

5.1.1 Switches

A switch is perhaps the simplest possible sensor. A conducting wire is either open or closed. We put a voltage on one end of the wire (relative to ground) and measure if it appears on the other end.

A **dead person's handle** is a particular type of switch used in safety such as when operating robots. It is any switch designed to be open unless a human is actively pressing it down, wither with their hands or sometimes feet. If anything goes wrong the human releases the switch – either by choice, or instinct, or if they are rendered dead on unconscious. The power system of the robot is wired so that power is only supplied to anything when the switch is closed, so all power to the robot is cut if anything goes wrong.

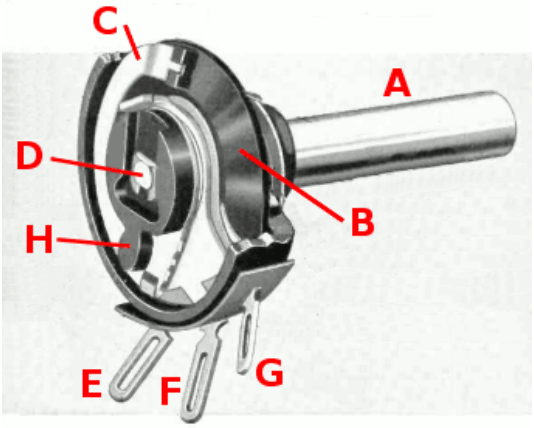


Switches are not completely trivial however. Any mechanical switch is likely to physically **bounce** on first contact. The moving part does not just fall into place to complete the circuit, but hits it and bounces off one or more times. This can cause havoc if your system is trying to count the number of switching events, or if it has a safety timer (as in a dead man handle) that resets things for some time after each switch event. **Debouncing** means the writing of code (or sometimes hardware design) that filters out bounce events from the signals. For example, an Arduino program might consider all switch events within some fraction of a second after a first one as artifacts of bouncing, and remove them.

5.1.2 Position

A **potentiometer** ("pot") is simply a variable resistor whose variability depends on its physical rotation. They are cheap to buy and are found, for example, in the controls of electric guitars.

They are made of a long resistive region (B) whose resistance depends on its length. The moving part (C) makes contact with this region at a variable location in it. So by moving this via an exposed knob (A), we move from a low to high resistance.



Potentiometers can be used to measure both angles and linear distances in robotics. For example, the angle of a rotary servo motor can be senses by instrumenting it with a potentiometer, used as a feedback sensor.

5.1.3 Light

A **photo-diode** is the opposite of a light-emitting diode (LED). An LED is diode which allows current to flow in only one direction, and converts some of its energy into light as it passes through. A photo-diode absorbs light and converts energy into current. At large scales, this can be used to general solar power. At smaller scales, it can be used for sensing. It is common to buy matched pairs of an LED and photo-diode which emit

and receive the same frequency, so they can communicate. Some systems use visible light when it is desirable for humans to see what is happening. In other cases, such as making TV remote controls, you want the light to be invisible so you us infrared. A fun project is to observe the flashing codes used by your TV remote and reproduce them with your own hardware.

Figure 5.1: Photo-diode-LED pair

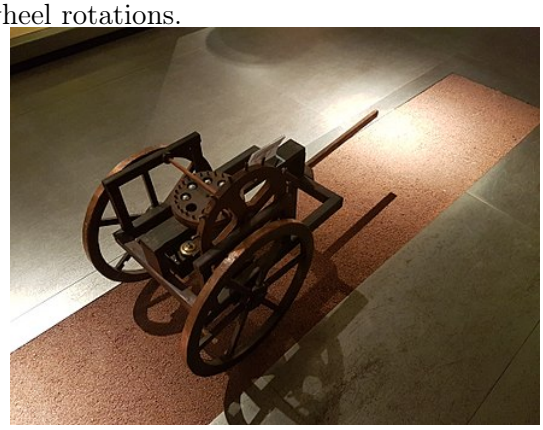
5.1.4 Pressure

Strain gauges are sensors whose resistance changes when mechanical forces are applied to them. They are typically made from a long thin wire of a material whose resistance varies with such forces, tightly wrapped around to amplify the effect in a small area. They are typically used together with a Wheatstone bridge to convert their output to a voltage. The can be used to detect both compression and expansion. e.g. sensing the weight (stress) of an object placed on top of them, or the force trying to pull an object apart (strain) such as experienced in some agricultural machinery. They are also useful for making expressive human interfaces, such as for musical instrument controllers.

5.2 Odometry

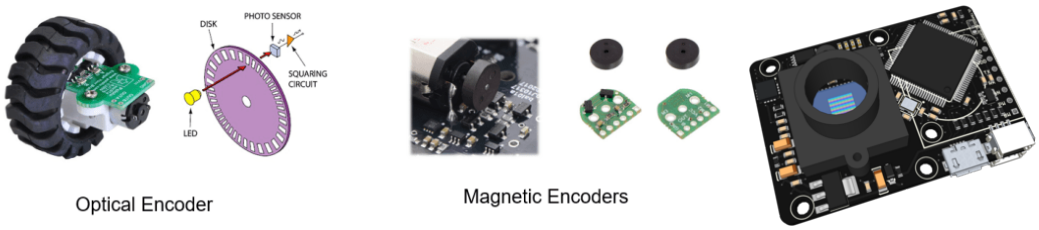
Odometry sensors measure a vehicle's motion relative to the ground.

The first known odometers date from Roman times. Wheels of a cart were fitted with balls and slots such that on each rotation of the wheel a ball was released from one box into another. The number of balls passing through the mechanism counts the number of wheel rotations.



Modern rotary (wheel or shaft) odometry can be done in various ways with different sensor types:

Figure 5.2: Optical encoder; magnetic encoder; optic flow hardware (PXFLOW)



- Rotary potentiometer - for slow moving shafts with limits on rotations – (doesn't work for general wheels, only limited servos such as on robot arms)
- Optical encoder - shine an LED through a rotating wheel with slits, then use a light photo-diode on the other side to count how many discrete flashes are seen as the slits pass by it.
- Magnetic - place a magnet on the moving part and a Hall effect sensor stationary nearby, which measures changes in voltage caused by the magnet. Often used in cars for ABS braking systems.
- Optic flow – measure the visual motion of the ground below. As used in optical computer mice. Available in real-time hardware, or use OpenCV to do in software if you have the compute power available.

- High level machine vision – detect and track the positions of objects relevant to motion control, e.g. place a camera inside a car facing the steering wheels and track the driver's hands on it or its own positions.

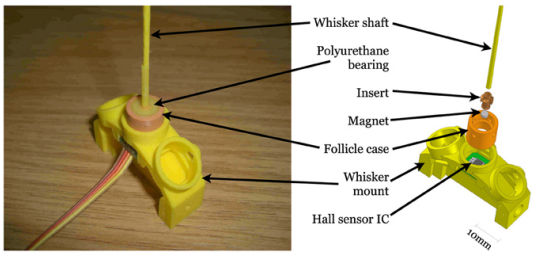
Rotary encoders usually come in two physical formats: ones which look like a rotary pot with a rotating shaft on the device itself; and "hollow shaft" encoders made of two concentric rings, the center one wrapping an existing axel-like shaft and the outer one bolting to something stationary. Hollow shafts are usually what you want for mobile robot odometry. The Alps Alpine series of hollow shaft encoders cost just a few pounds, are available from RS, and are often used in student projects. High end encoders for use on cars and industrial plants can cost hundreds of pounds.

e.g. a motor encoder installation video:
https://www.youtube.com/watch?v=dzV6QVr29vY
e.g. encoder products: *https://www.robotshop.com/uk/encoder-disks.html*
e.g. optic flow product: *https://docs.px4.io/v1.11/en/sensor/px4flow.html*

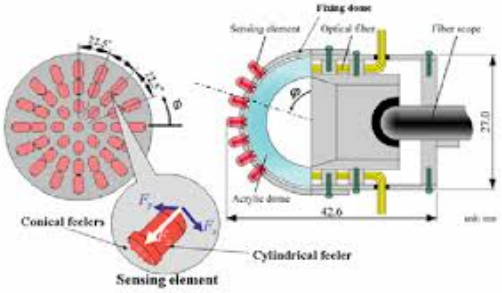
Figure 5.3: Alps Alpine rotary encoders in various forms

5.3 Touch sensors

Whisker-like sensors can be made of a passive 3d-printed bendable plastic whisker, with a magnet and Hall Effect sensor at the base to detect bending at the base only. Beam theory from mechanical engineering is then used to reconstruct the polynomial shape of the bend whisker from the reading at the base, and thus reconstruct the location of a contacting object (from Fox, Evans, Lepora, Pearson, Ham & Prescott. Crunchbot: a mobile whiskered robot platform, TAROS2011):

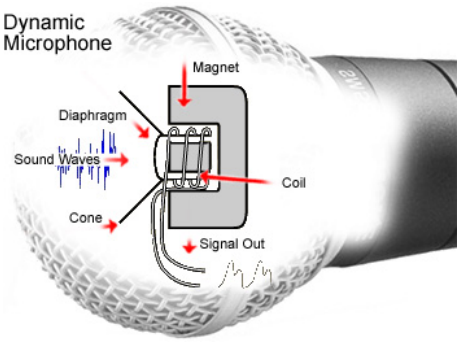


Current research into fingertip sensors includes **optical fingertips**. These put a known matters of dots onto the "skin" of a robot finger. Then a small camera inside detects deformation of the skin from motions of the dots:



(From Optical Three-Axis Tactile Sensor for Robotic Fingers, Ohka et al. 2008, DOI: 10.5772/6619)

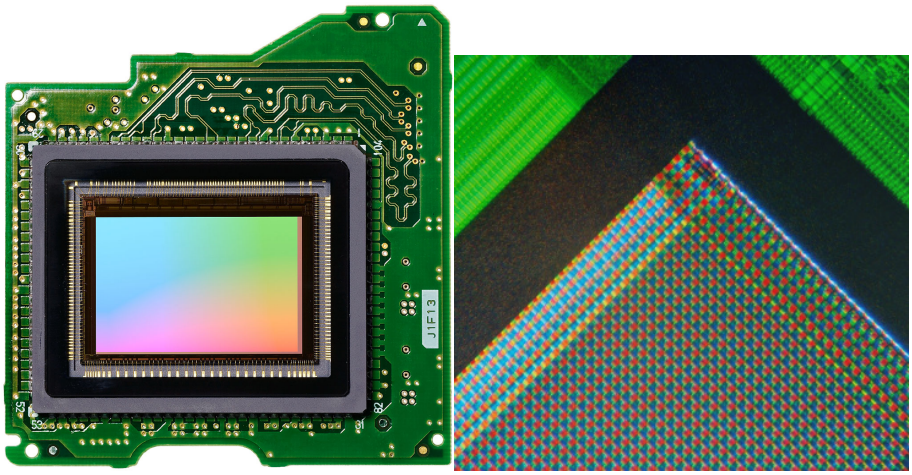
5.4 Microphones



Standard (dynamic) microphones work similarly to electric motors, but in reverse as they are sensors rather than actuators. A static magnet is fixed to the back of the microphone. A movable diaphram is placed over it, which is made to vibrate by incoming sound waves. An coil of electrical wire is built into the diaphram. When this coil vibrates in the presence of the magnet, a small current is induced in it. This current contains the audio signal and can be amplied or recorded electronically.

(A loudspeaker is an actuator which works in the opposite direction – with the current causing the diagrpham to move to create sound waves. The pickups of electric guitar work in a related way to microphones, though with moth magnets and coils fixed in the pickup and only the metal strings vibrating above them. The string moving in the magnetic field

Figure 5.4: *Digital camera sensor, and close up*



include currents in the static coils in this case. Guitar pickups are sometimes used as another simple way to build whisker touch sensors, as a vibrating metal whisker behaves like or can even be made from a guitar string.)

5.5 Cameras

Cameras are sensors which mimic the behaviour of the human eye, or sometimes extend or replace human-like perception with other types of visual response.

5.5.1 Sensors

Digital camera sensors are, roughly, 2D arrays of photodiodes as seen above. They are fabricated via photolithography, like integrated circuits. There are various specific technologies used with a lot of current innovation in the market.

5.5.2 Colours

The human eye does not respond directly to the frequency of light, as the ear does to sound. Rather, it contains four types of receptor, and each type responds strongly to a particular central frequency and less strongly to a wide range of nearby frequencies. Fig. 5.5 shows the response curves for the four types.

Red, green and blue cone receptors are concentrated in the center of the retina, while rod receptors cover peripheral vision in monochrome. The rods are the reason that you can sometimes see stars in the "corner of your eye" which disappear when you look directly at them. This creates a complex, compressive relationship between the underlying spectrum and the response. For example, identical perceptual yellow light can be produced either by a lamp emitting a single wavelength of 550 nanometers, or by two separate

Figure 5.5: Human eye response to colours. (From *www.sciencedirect.com/topics/page/Cone_cell*)

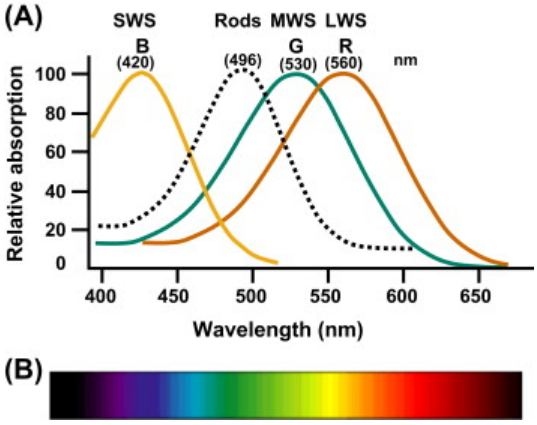
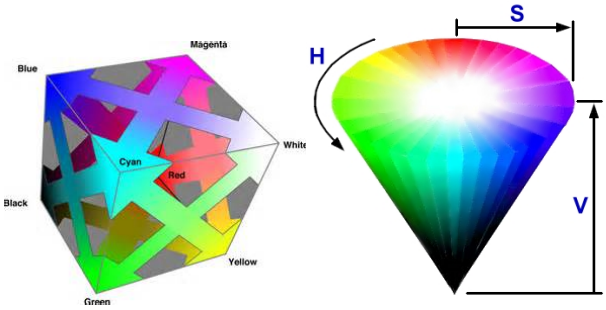


Figure 5.6: Colour space models.



sources at 560nm (red) and 530nm (green) as in a RGB computer monitor. Physically, these are completely different things.

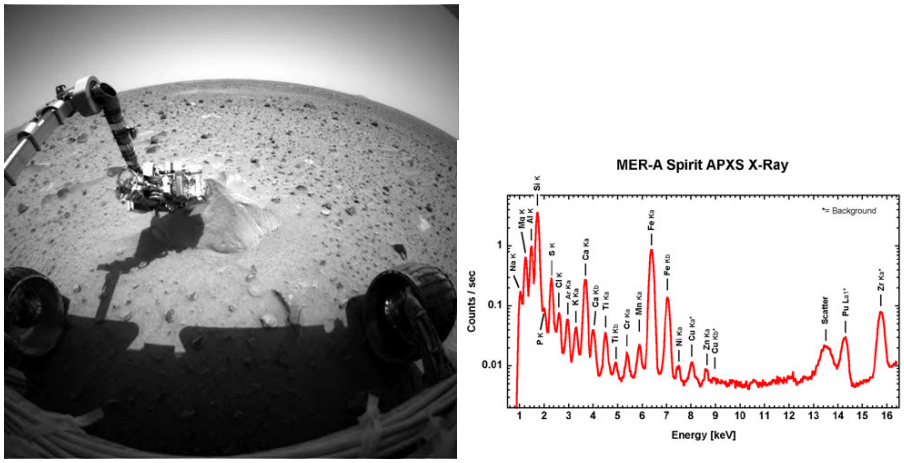
As we have three colour receptor types, we may consider perceptual colour space as consisting of three dimensions. They are sometimes represented as an RGB (red, green, blue) cube (fig. 5.6 left), but also in other ways such as Hue, Saturation, Value (HSV) cones (fig. 5.6 center).

The visual part of the spectrum is only a tiny part, and even within this part we are limited to sampling three specific frequencies rather than seeing the whole spectrum.

While humans get by with this, robots can use many additional frequencies, such as **infra red** (IR), **x-ray**, and **ultra-violet** (UV) channels, especially useful for detecting properties of living things. There is a region of near-IR in particular containign frequencies which are absorbed by H_2O molecules in vegatation. (NIR often penetrates clothing but reflects of skin, enabling NIR cameras to see human bodies through clothes.) Flowers often contain complex and beautiful UV patterns on their petals to attract insects who can see these frequencies. Agricultural robots in particular might thus use cameras with channels responding to these frequencies either replacing or augmenting the usual RGB channels. This can result in the need to process images with more than the usual three

Figure 5.7: NIR can sometimes see through clothing, while UV patterns are present on many flowers.

Figure 5.8: Mars rover *Spirit*'s spectrometer arm, and resulting x-ray spectrum showing chemicals inferred in the soil



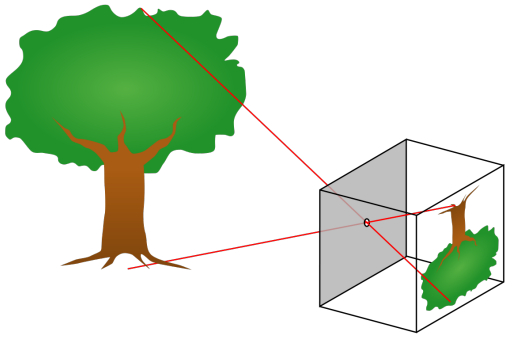
channels, requiring special software tools.

In some cases robots may be fitted with a full **spectroscope** to measure entire chunks of spectrum rather than just sampling channels. These are difficult or impossible to focus on individual pixels like a regular camera. They usually consider the whole sample region as a single 'pixel'. The full spectrum contains detailed information which is often used to infer the chemical structure of new materials. For example we can infer the chemicals in distant stars from their full spectrum, or the content of soil on Mars or farms on Earth.

Spectroscopes are usually large and very expensive, usually based in a static lab, though include on Mars rovers such as Spirit. Recent low-cost, handheld Kickerstarter-style products such as Consumer Physics' *SCiO* may change this soon.

Figure 5.9: SCiO sensor (www.consumerphysics.com)

5.5.3 Image formation



The basic **pinhole camera** equation projecting from 3D (x_1, x_2, x_3) space to 2D plane, assuming camera origin at (0,0,0) and principal plane at $(0,0,f)$:

$$Y = pinhole(X)$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \frac{f}{x_3} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

f is the 'focal length', it is simplest to use $f = 1$. Standing at distance f from the camera, a 1 meter person has height 1 meter in the projection. f behaves exactly like "digital zoom", it scaling the overall scale of the 2D image but does not alter its perspective. For a physical pinhole camera, is measures the horizontal distance between the pinhole and the image plate.

In robotics, we typically own one (or two) camera/s whose f is known along with the camera's location, which allows us to directly estimate the 3D positions of objects on the ground from this equation.

The same equation can be written in **affine coordinates** $(z_i = y_i w)$ as:

$$Y = MX$$

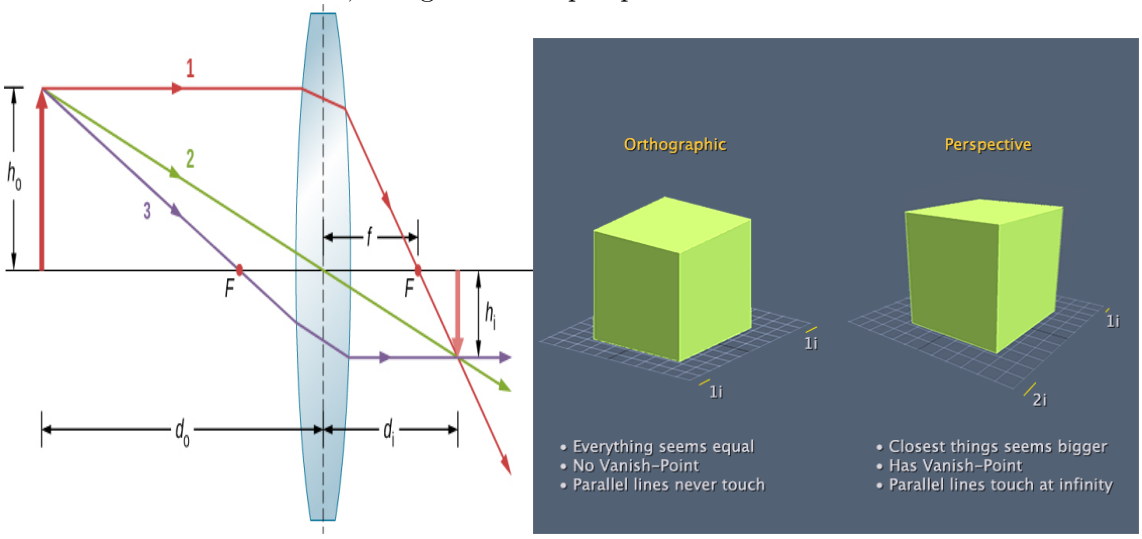
$$\begin{bmatrix} z_1 \\ z_2 \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix}$$

The 3x4 matrix is called the camera matrix M, and will in general take any value. The affine coordinates are defined as,

$$\begin{bmatrix} y_1 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} z_1/w \\ z_2/w \\ w/w \end{bmatrix}$$

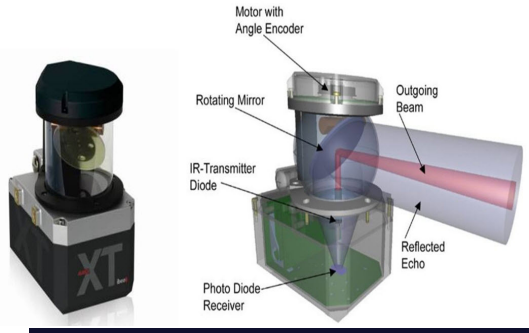
Affine coordinates let us handle the division by x_3 within the matrix multiplications, and also let us do transposition within them. They are able to represent directed points at infinity using $w = 0$, which will be a useful way to represent directed rays when we lose depth information.

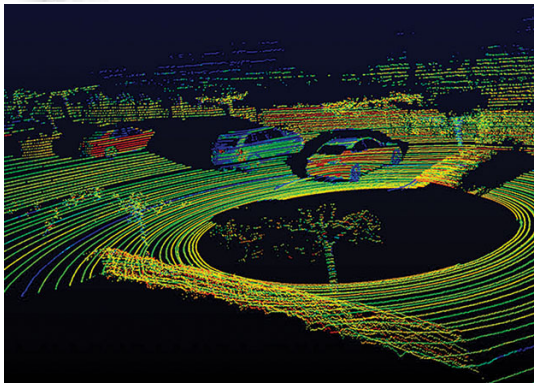
Lenses placed in front of the pinhole have the effect of shifting the camera to a virtual location forward or backward, and give rise to perspective effects:



5.6 Depth sensors

5.6.1 Lidar





Lidar (light detection and ranging) uses reflections of a scanning laser beam to measure locations of points in 3d space. Usually an invisible light frequency is used such as IR or UV.

If the beam is held in one position, it projects out and hits something in the world. This reflects the beam back to the lidar unit.

A simple homebrew lidar can be built using a **laser pointer** and a webcam. Point the laser pointer at an horizontal angle, and use the webcam to look for the bright (usually red) dot in the image. The horizontal distance in the image tells us how distance the contact object is.

Real lidars don't use such position shifts, instead they rapidly (e.g. 100kHz) pulse the laser and use a very precise time-of-flight sensing and calculation to compute the distance as the time taken multiplied by the speed of light. They then **scan** the beam and detector across two dimensions, like an old fashioned CRT television display. The beam is usually scanned horizontally along a row at high resolution. Then after each row, its vertical angle is shifted, for example so as to form 16 or 64 vertical scan lines. These can provide a detailed 3d scan of the environment, e.g. sufficient to detect pedestrians and cars as well as the shapes of buildings.

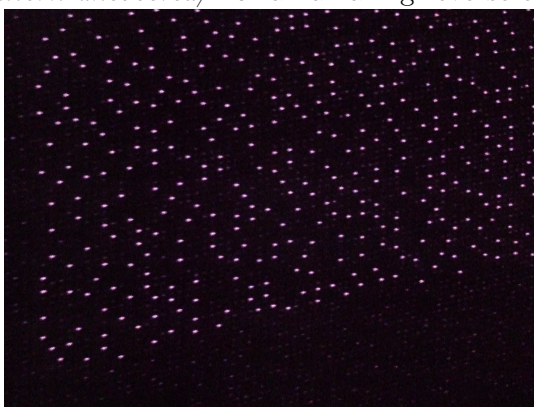
Problems sometimes occur with objects with unusual **reflection** properties, for example glass appearing transparent, or metallic surfaces which deflect the light in unexpected directions.

Self driving cars are very reliant on lidar. A concern is that when everyone has one their lidars may interfere with each other, picking up reflections from other devices.

PCL (Point Cloud Library) is the standard software tool for processing lidar. Its functions can, for example, try to chunk points into surfaces and objects, and identify features in these objects useful for navigation and recognition.

5.6.2 Depth from light patterns

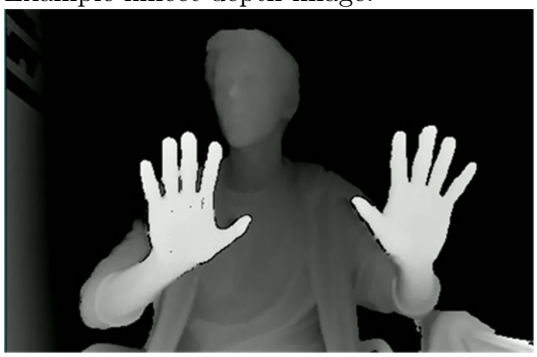
Related to lidar, Kinect and similar sensors project a **fixed pattern** of light out into the world, then look at its 2D projection in a camera image to compute the disparities and thus distances. It is also similar to the laser pointer ranging idea, but with many points at once. The point pattern is chosen to enable the parts of it to be recognizable even after the spatial distortions. (See *https://azttm.wordpress.com/2011/04/03/kinect-pattern-uncovered/* for an amazing reverse engineering of the pattern properties):



Kinect point pattern.

Current Kinects have lower ranges than lidar, using non-laser light sources.

Example kinect depth image:



The OpenNI library is the standard open source system for processing Kinect-like sensors.

5.6.3 Stereo cameras

Two cameras, displaced horizontally from one another are used to obtain two differing views on a scene, in a manner similar to human binocular vision. By comparing these two images, the relative depth information can be obtained in the form of a **disparity map**, which encodes the difference in horizontal coordinates of corresponding image points. The values in this disparity map are inversely proportional to the scene depth at the corresponding pixel location.

An information measure which compares the two images is minimized. This gives the best estimate of the position of features in the two images, and creates a disparity map. This an an intensive but highly parallizable computation which is usually done on a **GPU** or dedicated hardware built into the stereo camera. (We were very annoyed when we bought expensive "stereo cameras" which turned out to be two bog standard webcams glued together plus some code which requires our own GPU to run on.)

Optionally, the received disparity map is projected into a 3d point cloud. By utilizing the cameras' projective parameters, the point cloud can be computed such that it provides measurements at a known scale.

5.6.4 Acoustic ranging (sonar)

An acoustic ranging apparatus directs a burst of acoustic energy towards an acoustically reflective target object. The **time** it takes for an echo to return to the ranging apparatus is measured to determine the distance between the ranging apparatus and the object. The echo is received, amplified and compared with a threshold voltage, which decays in time in a manner which simulates the attenuation of acoustic energy as a function of distance travelled in the medium. The attenuation function is substantially simulated over the distance range that the ranging apparatus is intended to accurately measure by means of a simple, low-cost first order linear circuit. The ranging apparatus responds only to echoes having a magnitude which exceeds the threshold voltage. Spurious or false echoes, such as those due to reflection of acoustic energy in the side lobes produced by an acoustic energy transducer in the ranging apparatus, are ignored.

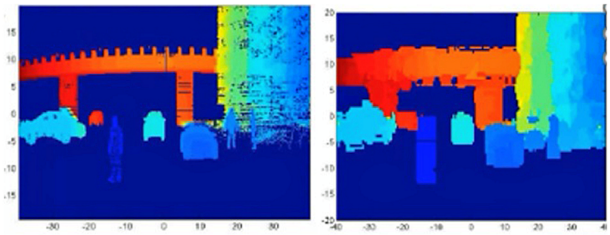
5.6.5 Radar

Radar is an object-detection system that uses radio waves to determine the range, angle, or velocity of objects. It can be used to detect aircraft, ships, spacecraft, guided missiles,

motor vehicles, weather formations, and terrain. A radar system consists of a transmitter producing electromagnetic waves in the radio or microwaves domain, a transmitting antenna, a receiving antenna (often the same antenna is used for transmitting and receiving) and a receiver and processor to determine properties of the object(s). Radio waves (pulsed or continuous) from the transmitter reflect off the object and return to the receiver, giving information about the object's location and speed. (*Wikipedia*)

Unlike sound, radar travels at the speed of light so computing time of flight delays is harder.

Traditionally: lidar has higher accuracy and radar is better in smoke and dust environments. But modern high resolution radar and give almost lidar like precision:



s

The two technologies are fighting in self-driving car markets. For example Google has used Lidar plus cameras while Tesla has used Radar.

See eg. https://www.sensorsmag.com/components/lidar-vs-radar for a comparison.

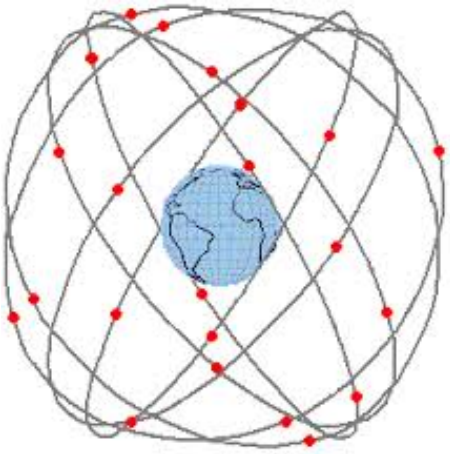
5.7 Location sensors

5.7.1 Global Navigation Satellite System (GNSS)

Global Navigation Satellite Systems (GNSS), include the best known Global Positioning System (GPS, American) and also the alternative GLONASS (Russian), Galileo (EU), and BeiDou(China). GNSS can also refer to fusing data from several of these systems. Details of the systems change over time as new satellites are launched and upgraded so the following is just a sketch of one system. GNSS are important military assets which each country maintains in case the others are disabled. It is likely that more satellites will continue to be launched to build in redundancy to all systems as states also build anti-satellite weapons. The advantage of this for robotics is that in peacetime, we get access to four systems, each with lots of satellites, to fuse together and improve our accuracies.

The original **GPS** system consisted of 24 orbiting satellites about 20,000km altitude, as shown in fig. 5.10. From any point and time on Earth, one can typically see between 6-12 satellites in the *unobstructed* sky. Urban and other obstacle-filled environments are unlikely to see as many, which can be a big problem. The satellites transmit microwave (1.2-1.5GHz) signals, containing their identity and correction information, at the speed of light, timed by atomic clocks. Receivers compare the time delays of signal arrivals, and triangulate their position, using knowledge of the satellites' positions and conditions. As the changing atmosphere affects the signals, a network of base stations at known

Figure 5.10: GPS satellite constellation.



locations monitors the errors, and computes and relays correction information back to the satellites, which include it in their transmissions. Localization accuracy on the Earth's surface from this basic system is around 10m. The satellites move around slowly in the sky, for example taking 30 minutes to appear and vanish over the horizons.

Differential GPS (DGPS) systems give higher accuracy, of the order of 100mm, by installing their own local base station in addition to the global network, and comparing the computed location with that of a moving vehicle. (You can't make a "poor-man's DGPS" from two standard GPS receivers however, because they must use exactly the same set of satellites, which requires extra logic for them to communicate and negotiate with each other.) Some wide regions have public DGPS services which transmit over radio such as WAAS in the USA and EGNOS in the EU.

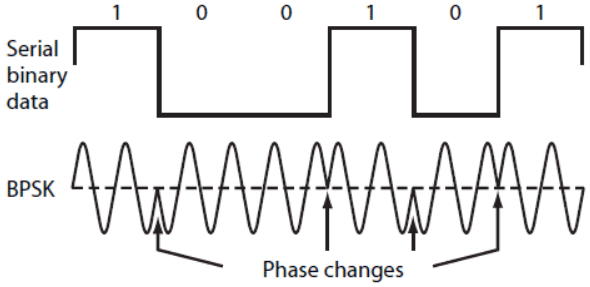
Real-Time Kinematic (RTK) GPS (fig. 5.11) uses the DGPS concept together with additional high-resolution carrier wave phases (fig. 5.12) between signals (rather than delays in the carried signals) to obtain accuracies up to 20mm on a good day. On a bad day, cheaper RTK sensors don't work at all. Some organizations operate local networks of RTK+DGPS base stations, such as *rtkfarming.co.uk* which covers the East of England using farmer volunteers' base stations.

Generally, GNSS is not accurate enough by itself to localize vehicles for autonomous driving, especially in urban areas, and is fused and filtered with many other sensors, including its own historical readings, for this purpose. Similarly for car satnavs, heavy use is made of smoothed sequences of observations and knowledge of its speed and heading. It is easier to predict and smooth a high-speed car's positions on a motorway than its finer movements around intersections. This is why expensive RTK is common for agricultural tractors but is less common for cars. A vehicle's heading can be estimated either from its direction of motion or from a pair of RTK sensors placed at its opposite ends.

Usually robotics will use only the final estimates locations from **NMEA** data sentences. Occasionally, we may work with data from the individual satellites which also appear in the NMEA data, for example when researching new algorithms for fusing data from

Figure 5.11: *RTK/DGPS base station.*

Figure 5.12: *Carried (top) and carrier (bottom) signals.*



different systems or with other localization data sources. Most physical sensor devices come with serial port which streams lines of text to a computer, as a stream of bits chunked into ASCII characters (bytes). If you insert a probe into their serial port cables you can actually see the 0s and 1s over time in the voltage. The format of this data from most GPS-type devices is called NMEA and looks a lot like a CSV file, with lines like,

```
$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
```

You will often receive log files with lots of these lines recorded over time. The first symbol is the type of data. GNSS is very complex and produces lots of information about individual satellites and accuracies, but the lines which begin with "GPGGA" give the best summaries of the actual location as you would see reported in a car satnav. The second field is a time stamp in HHMMSS string format, using UTC time. The third and fifth fields are the latitude and longitude, in degrees, as strings. You will typically filter NMEA log files to get just the GPGGA lines, then pull out these three fields.

5.7.2 Inertial Measurement Units (IMU)

An inertial measurement unit estimates its pose relative to to its starting pose. IMUs are commonly found in mobile phone as well as robots and drones. They usually work by fusing three types of sensor, built on the same PCB or single chip:

Magnetometers are compasses which measure the direction of the local magnetic field. The earth has a magnetic field so like a handheld compass this usually estimates the direction to the earth's north pole.

Accelerometers detect linear acceleration. When you sit an a car which accelerates, you feel a force pushing you back into your seat. If a piezoelectric material was put there, it would also feel squashed, and would change its resistance which could be measured as in a pressure sensor, forming an accelerometer.

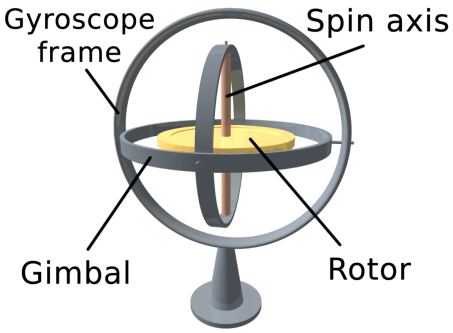
Gyroscopes (gyros) are used to measure rotational velocity. They are physical spinning systems whose (famously difficult, traditionally used to cull first year physics students) physics makes their rotor tend to stay in its absolute pose even when the frame is manipulated to rotate around it. Very high end, million dollar, gyros can retain this pose well over whole missions, so the pose difference between the rotor and the frame gives a near perfect estimate of the absolute rotation. Lower end systems as found in most research robots will have the rotor drift gradually over time. So rather than measuring absolute rotation we use deltas over time to measure only the weaker rotational velocity.

Typical configurations contain one accelerometer, gyro, and magnetometer per axis for each of the three vehicle axes: pitch, roll and yaw. Some IMUs include a **microcontroller** and code to fuse the sensor data together into a single pose estimate, others make the raw data available and expect you to do this yourself.

They are typically fabricated as **MEMS** (Micro-Electro-Mechanical Systems) a (nano)technology which 3d prints layers of mechanical systems, including moving parts, onto silicon or other base using similar photolithograhy methods to chip fabrication.

Magnetometers are sensitive to metal in the environment such as the body of a robot, so have to be **calibrated** for their field readings to make sense as direction estimates.

Figure 5.13: *Gyroscope*



5.7.3 Real Time Localisation Systems (RTLS)

For limited smaller environments, such as a single street, higher precision than GNSS can be obtained using Real Time Localization Systems (RTLS). (Sometimes also known as "indoor positioning systems", though they usually also work outdoors.) These work on the same principle as GNSS but using locally installed beacons rather than distant satellites. Beacons and detectors may be of various types:

Visual – stick easily recognizable objects such as reflectors at beacon locations; or use visible or invisible (IR, UV) lights.

Acoustic (active bat) – beacons might transmit ultrasound, then use echolocation.

RF – beacons transmitting radio signals, use signal strength, direction of arrival (via phased arrays), and/or tim-of-flight (as in lidars) to localise. Like lidar, this processing speed of light signals in real time, which is much more challenging than audio DSP. Software Defined Radio has recently opened up many possibilities here, suitable for use and reaseach in your own projects.

Wifi – some systems have used distance and direction to local named wifi networks to localize (eg. Google maps and cars – they got into legal trouble for recording the names of domestic networks though).

5.8 Chemical sensors

Chemistry is a huge field beyond the scope of this module, but its reactions provide many ways for sensors to detect the presence of chemicals in the robot's environment.

5.8.1 Mouth-like sensors

Lincoln research is currently mounting many types of chemical sensors of agri-robots to monitor soil quality – again Soil Science is a huge research field studying the interactions of many physical, chemical and biological components in soil. Typically such projects involve roboticists working with Chemistry specialists who design the sensors – the roboticists just worry about how to stick them into the ground in the right places.

Figure 5.14: *Marvelmind Ultrasound RTLS*

This type of chemical sensing is a bit like a robot "mouth" as the material being sensed has to be physically brought inside the robot or into contact with the robot. Particular types of mouth-like sensors of interest to agri-robotics include:

5.8.1.1 Moisture sensors

For plant-watering applications we need to measure the current water content of soil. Moisture sensors can be bought for a few pounds and come with some Arduino kits. They work by measuring resistance: pure H_2O doesn't conduct electricity but real world water also contains other ions which do conduct in proportion to how much water is there.

5.8.1.2 NPK fertilizer sensors

Nitrogen (N), Phosporous (P) and Potassium (K) are the three chemicals all required in plant growth, and are widely applied as fertilizers so are often required to be monitored. Measuring them directly is difficult so most so-called "NPK" sensors actually measure side-effects of their presence to infer their levels. These can be bought for around 100GBP. (See also *https://teralytic.com/*)

5.8.1.3 DNA sensors

Biological DNA for humans consists of around 1Gb of data coded as chains of chemicals (named A,T,C and G). Other lifeforms have similar amounts. Genome measurement has very recently falling in price, especially for methods with only detect a few **fingerprints**

Figure 5.15: *Moisture sensor*

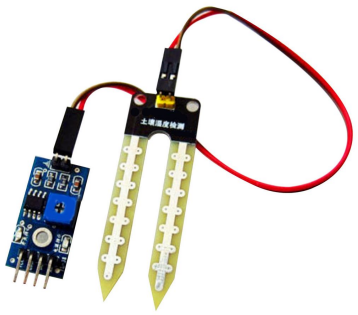


Figure 5.16: *Left: in-field DNA sensor. Right: in-field NPK sensor.*

rather than sequencing whole genomes. For example these are used in labs which run commercial genealogy tracing. Portable versions of this technology such as the Oxford Nanopore MinION (*nanoporetech.com/products/minion*) now enable similar sensing to run on robots in the field.

5.8.2 Nose-like sensors

The other common type of chemical sensor as an "artificial nose" to mimic human smell in detecting multiple types of chemical in the air, at a distance from their source. Often levels of chemicals are used to build gradient maps, which can be used to locate and follow the source of a chemical which is diffusing through the geographic environment. e.g. the source of a pollutant in an agricultural field.

5.9 Exercises

- Search the internet or academic literature and make a comparison table of pros and cons of the different 3D ranging sensors. Present your sensor findings as a written report or presentation.
- Do as much as you can from Arduino sensor tutorials: Lessons 8 (tilt ball switch), 10 (ultrasonic), 11 (temperature), 12 (joystick), 18 (photocell).

5.10 Further reading

5.10.1 Recommended

- Bolton, chapter 2 (sensors), 8 (mechanical actuation), 7 (hydraulics).
- A Malik, 2009. RTLS for Dummies, Wiley.

6 Building robots

The hardest part of mechatronics for many students has traditionally been the physical mounting of sensors and actuators onto robotic systems. There are no widely agreed standards for mounts so most robots will need custom physical components for making them fit together. Luckily, as with most parts of mechatronics, this has recently got a lot easier and cheaper. The following is everything you need to set up your own mechantronics lab.

6.1 Open Source Hardware (OSH)

The use of OSH allows for more effective and accessible sharing and collaboration among researchers, as well as more accessible entry-level projects. OSH brings a number of benefits to a research community, as a common context for results and accessible platform with which to test new ideas allows for more engagement, leading to a more mature field of study.

The precise meaning of open-source hardware (OSH) **remains widely debated**, unlike open-source software which has become widely accepted to be defined by standard documents such as the GNU General Public License and BSD licence (and others including Apache, MIT and Creative Commons licences). The OSH issues debated stem from the differing design process between hardware and software, as the development of hardware is much more expensive to iterate and has a much higher entry-level skill requirement. This topic has been augmented by the combination of "3-D printing with open-source microcontrollers", however is not yet as widely accepted open-source software.

For **software** to be open source, its **entire stack** of compiler-linked dependencies must also be open source, while non-linked dependencies need not be open source. For example, an open source program might depend on a non-open operating system which it calleds through non-compiler linked shared object libraries ("so" or "dll" files). The main debate around OSH definitions is what the equivalent boundary should be for hardware. In conventional product design, a "**product**" is a legally defined entity which may, for example, be registered as an entity via a patent or receive safety certification as an entity. Such a product will specify as part of its legal existence that is it is comprised from, or manufactured using, other products, which themselves have legal definitions. For example, to obtain a CE safety mark, all the component products must themselves have CE marks. The OSH community does not usually use the term "product" as this denotes a commitment to certain commercial worldviews, but may use the term "design" in an analogous way.

It is currently impossible to build almost any large hardware design that does not incorporate non-open designs or make use of non-open designs as tools during construction. So various definitions of OSH of different strengths have been proposed as difference balances between openness and pragmatism. The following are not standard terms but are currently used around Lincoln, not always yet with consistent meanings:

Weak OSH. The weakest form of OSH is based on conventional product design, and considers the design itself to be the only component in need of openness. Only the design for how to combine it with other components is OSH. This means that the new design could be completely dependent on patented component and tool designs, and on the "owners" of these designs allowing their use. For example, the Arduino computer is Weak OSH because its component microcontroller, the Atmel AVR, is a patented design and can only be made and sold by its owner.

Deep OSH. The strongest form of OSH, as aimed for but not yet realised in the Open Source Ecology (OSE) project, would build a complete stack of OSH such that there are no non-open components used at any level. Every motor, screw, transistor in a Deep OSH design would itself be Deep OSH. This is usually impractical and no completed Deep OSH are currently known.

Shallow OSH. This intermediate view of OSH is that openness refers to a design, which may be built from non-open component designs and tools if and only if the *interface* to these designs and tools is non-patented. This enables multiple suppliers to produce competing alternatives which can be swapped in for these components, including Deep or other OSH alternatives. International standards exist to define many such open interfaces, for example ISO and EU standards. This gives some protection to the life of the new design, making it resilient to suppliers of components going out of business, stopping production, or overcharging for patents.

Deeper OSH. Deep and shallow are thus ends of a continuum, in which the depth of an OSH design can increase as more components are gradually swapped for deeper OSH designs, until a fully deep design is reached.

Some other aspects involved in the OSH debate include:

Freedom to close modifications. Analogous to the GPL/BSD debate in software. GPL-style OSH specifies that if you modify the OSH design then you have to controbute your changes back to the community by making them public under the same licence as the original. BSD-style in contrast gives private companies the right to withold their changes from the community and profit from being their sole 'owner'.

Open tooling. Should an OSH design have to be buildable from tools which are OSH, or can propriatory tools be used? This applies to both physical tools such as screwdrivers and bench power supplies, and to software tools. Most OSH people assume that open source software needs to be used in the designs, or at least *can* be used to read and edit the design files. OSH hardware screwdrivers (made on all-OSH tool-making machines!) are usually considered quite an extreme interpretation. (Patents do exist and are enforced on some specialist screwhead and screwdriver head patterns however, which may be a more direct problem.)

Bottoming out. Is there some lowest level of component where everything is assumed to already be OSH? For example standard electronic components are usually implicitly

assumed to form this level. (However there was a time when, for example, transistors had just been invented and patented, and this could happen as new components are invented.)

Build instructions. 'Doing a build' for software requires a cmake file and a couple of commands, but for hardware it's much more complicated and may take days of manual work. You might have a CAD file showing the design but be unable to build it unless you also have easy to follow, Ikea-style, build instructions. These may include a Bill Of Materials, includling web links to buy each component from one or more suppliers, then step-by-step instructions and sometimes pictures showing how to put them together, with what tools, in what order. An emerging convention is for OSH jounral and conference reviewers to focus primarily on your github build instructions rather than your paper, and in some cases actually perform their own build to check that they are clear enough. But how much skill should be assumed on the part of the builder – should OSH mean that *anyone* can do a build, or only if they hire a skilled technican? (In our Lincoln work we usually assume the builder is 'a reasonably smart undergrad STEM student' having access to basic tools as in this chapter.).

How to make money? Unlike software, hardware costs serious money to both design and manufacture, and hardware designers tend to think more in traditional engineering terms of investment, sales, and profit than software designers who have often moved to open source models. OSH makes it hard to profit from being the sole manufacturer and supplier of a design. However it can create more sales as customers increasinly demand OSH products and are willing to pay premiums for them. Customers take a large risk when they design their own product based on a non-OSH component which may go out of production at any time. OSH gives an assurance that other manufacturers or even the customer themself could step in to continue manufacture. As with open software, the market currently seems to reward the original designers by buying goods and support from them at a premium over third party manufacturers, as the original designers have a valuable reputation which they are presumed to want to preserve via quality control in their manufacturing and branding. For example it is usually preferable to buy a 'genuine Arduino' for your project because you know it's going to work, rather than a no-name version which might cause days of bug hunting if it has a fault.

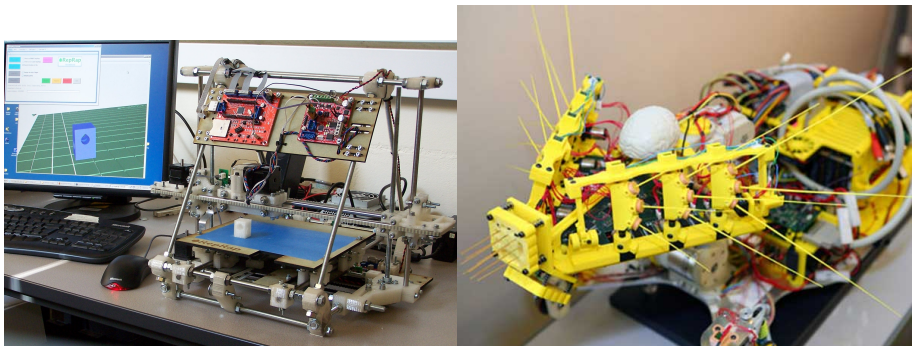
6.2 Tools

6.2.1 3d printing

3d printers such as the self-replicating, open source hardware RepRap (*reprap.org*), have revolutionized prototyping and production for small to medium sized plastic parts of arbitrary shapes. They are ideal for making mounts to connect parts together. 3d printers take a 3d CAD file and print it layer by layer, building the model from the ground up like an ink-jet printer.

You can buy a complete RepRap kit for a couple of hundred pounds. They are self-replicating, make largely from their own printed components, so you if have access to one then you can make your own copy of the plastic parts and just buy a kit of the metal and

Figure 6.1: *RepRap 3 printers. ScratchBot's components are mostly made and mounted using 3d printing .*



electronics parts, which are all commodity and widely available, e.g. on eBay. Building a RepRap is a great way to get some hands-on mechatronics experience in itself.

You can also send your 3d designs for 3d printing by many companies (e.g. *3dprint-uk.co.uk*, *3dprintdirect.co.uk*) who will run a RepRap or similar machines for you. This is very easy to do, usually you upload a *.stl* file to their website, pay a few pounds, then your plastic will arrive in the post in a few days.

Sites such as *www.thingiverse.com* host thousands of downloadable, printable designs and you can add yours to them.

6.2.2 FreeCAD

To design your 3d objects you need CD program. FreeCAD (*www.freecadweb.org*) is an open-source one which is often used with RepRap. (It is also often used to make 3d models for use in Gazebo and games.) Its website has good tutorials to get started.

There are many open source CAD files for standard needs, for example *www.thingiverse.com* hosts a large database of them.

(You might also come across Blender, another open source 3D modeling program more focused on artistic creation than engineering, for example used to design characters for video and tabletop games.)

CAD programs usually use ontologies similar to those in robots simulation, descriping a design as a collection of 'parts' with each part made from basic shapes. It's common to need to represent *holes* in CAD designs, for example if you create a cylinder then want to drill a hole through it. The hole is often represented as an entity in the ontology, which unlike most entities represents the *absence* of physical stuff. Traditional CAD for CNC ('computer numerical control', or control of machines for subtractive manufacturing) interprets holes not as entities but as commands to *do* something, namely move a drill to a certain series of positions.

CAD programs may use parameters, for example to define the position of a components *relative* to another rather than in absolute space. This enables many components to automatically shift around in response to changes in those to which they are releated.

Figure 6.2: *FreeCAD models for Andrew Henry's Lincoln MSc 201920 open source hardware robot guitar player.*

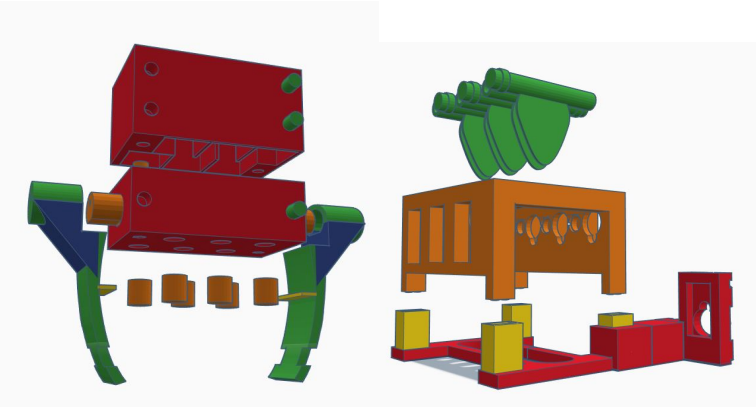


Figure 6.3: Bench power supply and leisure battery

6.2.3 Power supply and measurement

When setting up and testing sensors and actuators you probably want to power them up indpendently from your main robot. While it is possible to use regular batteries for this, it's usually more convient to buy a **bench power supply** (e.g. 50GBP) which can be set to create any voltage and supply any current at it, up to some safe limits.

For mobile robots, you can't usually rely on a bench supply and instead need a battery onboard the robot. The standard choice is a 12V lead-acid **leisure batteries**. These are so-called because they are often used in leisure vehicles such as golf buggies, boats and caravans. Car batteries are also lead-acid 12V but are optimised for the very specific needs of cars – namely providing occasional very large pulses for sparkplugs – while leisure batteries are optimised for more the continuous usage typical of robots.

Battery technology is currently advancing very rapidly, due to consumer demand for long duration batteries in both small low power devices (i.e. phones, UAVs) and large vehicles (i.e. electric cars and bikes). Various new types of battery have been developed based on **Lithium**, such as Lion (lithium ion), Lipo (lithium polymer), and Lifepo (Lithium Iron Phosphate), which can store much more energy per weight than lead-

Figure 6.4: Multimeter and USB oscilloscope

acid. However some may be prone to catching fire, and may require specialist recharging processes which are harder and more dangerous to manage than lead-acid.

6.2.4 Measurement

A **multimeter** (eg. 20GBP) is very useful for measuring voltage, current, and other properties when diagnosing your electronics.

If working with periodic signals – such as musical synthesis or from the sensors in a brushless motor – an **oscilloscope** can be used to visualise them over time or by phase against other signals. These used to be expensive but can now be bought as USB versions, using a computer for the interface, for <100GBP.

6.3 Materials

6.3.1 Aluminum extrusion profiles

For load-bearing robots, you usually need to use metal at least to create the main structure to hold 3d printed parts in place.

The quickest and simplest way to make metal structures is to use aluminum extrusion profiles. These are made by melting aluminum (alu, “ally”) and forcing it though a hole cut to a particular pattern, like a sausage machine. These patterns are made so that the resulting 3d bars are strong, light, efficient, and make it particularly easy to attach bolts to fit pieces together to make corners and frames. Many companies make and sell alu profiles, often as “structural systems” in which they match the profiles to nuts, bolts, brackets, and other Lego-like components which all fit together to make different shapes. Some systems include advanced components such as linear actuators which fit into and move along the profiles. Annoyingly these systems are not usually compatible with one another.

You will often see mobile robotics, lab equipment, and even lab furniture such as desks built from alu profile. You cut alu profile yourself with a suitable saw, but usually it’s easier to order it precut from a supplier. Take your FreeCAD design, measure what lengths and needed, and order them per project.

Figure 6.5: *V-Slot aluminium profile system*

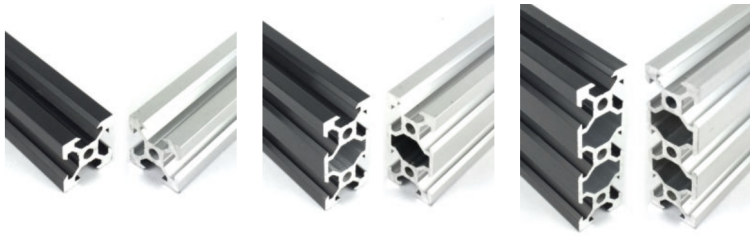


Figure 6.6: Resin Identification Code

An open source hardware alu profile system is V-slot, available in the UK from https://ooznest.co.uk/ .

An extensive, but expensive and patented, alu profile system is Bosch Rexroth (https://www.boschrexroth.
groups/assembly-technology/topics/aluminum-profiles-solutions-components/index). RS lists several other systems including its own.

For heavier robots, alu might not be enough and you will need steel frames. High end steel is a complex subject and there are many different types of steel with different properties. Traditionally steel must be welded together - you can buy a MiG welder for a few hundred pounds but they require training to use and are fatally dangerous if used without correct training. Dexion Speedframe is an interesting recent alternative to welding: it is a profile system which includes alu and steel versions of the same parts. So you can prototype in alu then switch to steel when happy with your design.

6.3.1.1 Plastics

The six most common consumer plastics are always labels with Resin Identifican Code symbols to aid with recycling and disposal – if you look carefully you will find one somewhere on any consumer plastic item (e.g. on the bottoms of plastic bottles, and inside their caps, which are usually a different plastic to the bottle).

(The recycling arrows are misleading because only 1 and 2 are typically recycled – i.e. they are the plastics that you put in your recycle bin, with the others for landfill. There are proposals to replace the arrows with a simple triangle at least for the plastics that are not currently recycled.)

Important properties of plastics are whether they degrade in weather including sun-

light, water, and temperatures; whether they are deformable (so can be used to make moving hinges such as on DVD cases); and how recyclable they are.

RID	name	uses
1 PETE	Polyester	transparent bottles, eg. mineral water bottles
2 HDPE	High density polyethlyene	translucent bottles, cups; low-end 3d printing
3 PVC	vinyl	pipes; signs; flooring; wire insulation; records
4 LDPE	Low density polyethlyene	polythene bags; food wrapping
5 PP	Polypropylene	consumer product bodies, eg. laptop cases, toys
6 PS	Polystyrene	packaging; styrofoam
ABS	Acrylonitrile butadiene styrene	high end 3d printing; crash helmets; trumpets
7 other	nylon	

Codes 1-6 are usually considered "consumer" plastics; ABS and other "engineering plastics" are rarer, more expensive, and have higher quality properties. Consumer plastics are in the most common use so are most important to recycle. Sorting used plastics for recycling is an interesting and relevant robotics challenge. There are good purely mechanical systems for 1 and 2 at least, based on cutting into small pieces, then quickly doing spectrometry with a scanner and using air blowers to separate the pieces. But could we do this more energy efficiently by e.g. deep learning to recognise whole Tesco packages such as milk bottles?

6.3.2 Steel

Heavy duty robots such as for agriulture or transport will usually use steel frames or plates. 'Steel' is not a single material, rather 'steels' are a large category of alloys, all based on the the idea of adding small amounts of carbon and other elements to iron to modify its properties. Steel chemistry is a large subject with its own dedicated MScs. Steels can be bought as 'steel bar' or 'steel plate' from various sellers, then cut into the required shape using metalworking tools – metal drill, lathe and milling machine – or welded together. These are heavy duty and potentially dangerous processes which require extensive operator training.

6.3.3 Wood

Don't underestimate old fashioned wood for prototyping robots! Wood can be very easy and fast to work with, as you can saw it by hand without needing power tools. You can also screw into it for mounts.

6.3.4 Enclosures and IP ratings

The technical term for a "box", "project box", or "case" to house electronics components is an "enclosure". Enclosures are used to keep electronics safe from water, dust, and any other flying objects which would otherwise damage them. Nowadays you can often 3d print a custom enclosure for your electronics which also attaches to your robot.

Standard sized enclosures can also be purchased off the shelf. This may be cheaper than 3d printing as they are mass produced, though they are often strangely expensive. They may also come with standard ratings for protection against dust and water known as IP (ingress protection) ratings. These have two digits, such as IP56, where the first (6) is the level of dust proofing and the second (7) is the level of waterproofing. IP56 is usually what you want for robots research, IP67 for industrial deployment. (IP67 is also found on phones where it means they can survive drops into toilet bowls.)

Another option is to re-use enclosures or packaging from other products, which may be cheaper but lacking official IP ratings. For example, old hi-fi equipment may sell second hand for less than new enclosures of the same 19" rack mount form factor, and be resused as enclosures if you strip out the old electronics. Gardeners' waterproofed power cable extendors such as "dribox" are especially cheap and useful for robots.

level	*dust (first digit)*	*water (second digit)*
1	objects >50mm	vertically dripping water
2	objects >12.5mm	dripping water at <15 degrees
3	objects > 2.5mm	spraying water
4	objects > 1mm	water splash (eg bucket pour)
5	some protection from normal levels of dust	water jets
6	dust tight	powerful water jet (pressure cleaner)
7		submersion <1m
8		submersion >1m

6.4 Connectors

6.4.1 Nuts and bolts

ISO 4017:2014 is the international standard for nuts and bolts. It defines the metric "M" notation for nuts and bolts. It is what ensure that when you have a random bucket of nuts and bolts from different manufacturers, they all fit together. It is one of the great human achievements of our time!

It defines names such as "M3" and "M4" which mean 3mm and 4mm diameter threads respectively.

You can buy large packs of bolts for a few pounds from RS, Amazon etc.

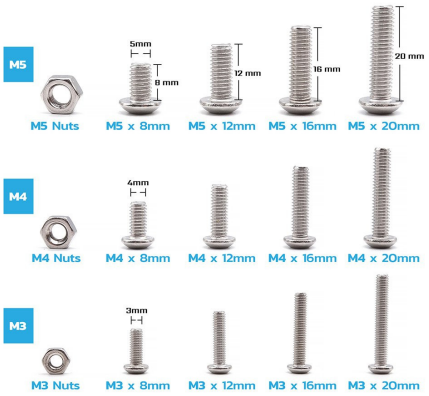
Be careful if using equipment from the USA which is the only country in the world to use inch-based rather than metric-based engineering standards, including bolts. A couple of American bolts in your collection can really mess it up. Lidar systems are often considered to be "optical equipment" which traditionally uses Amercian inch standards so take special care when mounting them.

6.4.2 Mounting motors

Attaching your motor to your robot is a common and important mechanical task. There are many EU standards, including mounting "IM" and others:

http://new.abb.com/docs/librariesprovider53/about-downloads/low-voltage-motor-guide.pdf

Figure 6.7: ISO 4017:2014 bolts



Motor to vehicle chassis mounts. NEMA and IEC are two main organizations issuing standards for mounting points. Floor mounting (for a wheel motor) means the length of the motor cylinder connects to a ground plate below it. Flange mounting (for a wheel motor) means that the circular face of the motor cylinder around the shaft connects to the side wall of the robot. (flange="a projecting flat rim, collar, or rib on an object, serving for strengthening or attachment or (on a wheel) for maintaining position on a rail."). e.g. NEMA23, NEMA17 (steppers used in RepRap design). The standards specify (families of) standard locations for screws to connect.

Motor shaft to wheel attachments also have various standards. Annoyingly, you often need to ask a technician (or if you have suitable health and safety permissions, use large metal-working tools yourself) to make adaptors between them.

6.4.3 Connectors and soldering

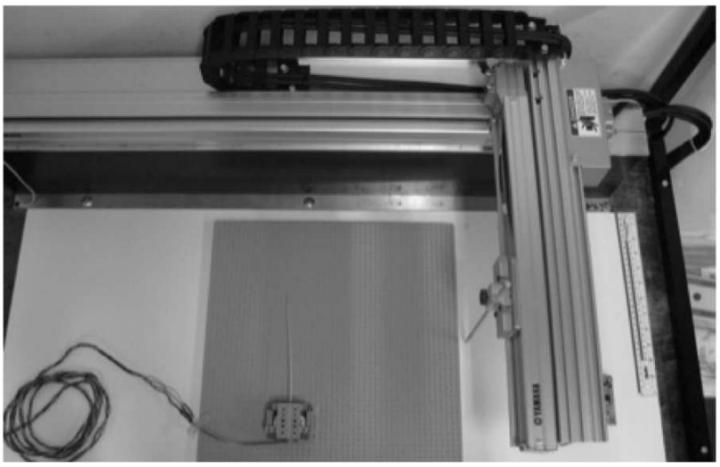
There are many types of connector available which reduce or remove the need for soldering. Classic "choc-block" connectors are plastic mounts with metal screws that make connections. If you don't even want to screw then Wago connectors are even simpler, just using a small lever to hold connections in place.

If you really need to solder: you can buy a soldering iron an some older for a few pounds. Never heat the solder directly, this makes poor connections which break. Instead, heat the metal to be joined, then push the solder into it one it is the same temperature as the soldering iron. Use 'helping hands' to hold everything in place before soldering.

For all these connectors you will need to strip the plastic covering of the wires using a wire stripper tool (don't use your teeth like guitarists do).

Figure 6.8: Choc-block, Wago connectors, wire strippers, helping hands, soldering kit

Figure 6.9: Lego interface for prototype whisker sensor (Evans et al, ROBIO2010)



6.4.4 Welding

Heavy duty robots such as for agriulture or transport will usually use steel frames. 'Steel' is not a single material, rather 'steels' are a large category of materials

6.4.5 Lego

Lego isn't just a toy, actually it's a highly sophisticated interlocking brick system. For example in (Evan, 2010) we needed to make many very precise positioning of objects contacting whisker touch sensors. The original plan involved lots of RS components, metal drilling and bolts, but we found that by 3d printing a Lego-compatible mount for our whisker, we could then mount it on any existing Lego systems to get sub-millimeter precision. Technic and Mindstorms robotics Lego also bring many possibilities for prototyping and student projects. (Mechano has similarly been used by a previous generation.)

6.4.6 Last resort attachment

Cable ties can be used not only to bundle your wires together but also to attach anything to anything else as long as you can drill holes in them both. If all else fails there is gaffer tape.

6.5 Physical circuits

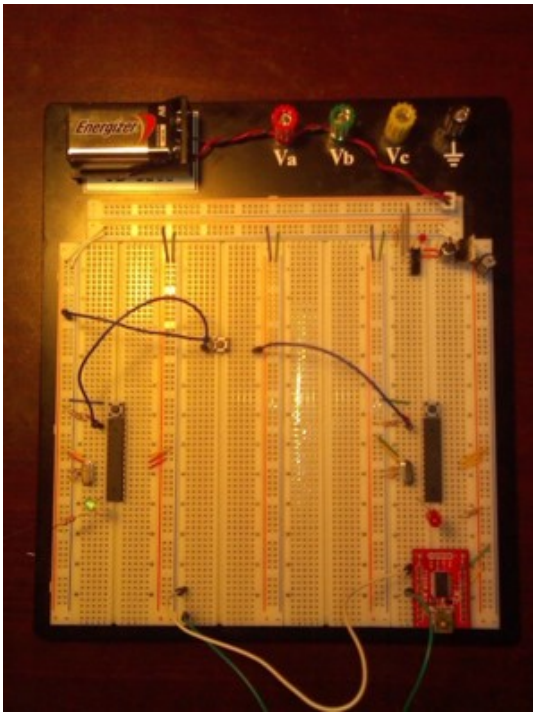
Arduino is designed deliberately to be easy to use for Computer Scientists as opposed to engineers. You wouldn't normally sell a product based on a full Arduino board. You'd probably prototype your system on it before taking just the AVR microcontroller, put it on a real circuit (on breadboard) without the rest of the board before getting it manufactured and selling it.

Figure 6.10: Cable tie; gaffer tape

This is designed to make circuit design a little easier so you don't have to solder things. A **breadboard** is just a big piece of plastic with some holes containing interconnected metal parts underneath. Enabling you to connect bits of wire together without soldering. You just plug stuff into it and if there's two adjacent or in some other related space they'll get connected together.

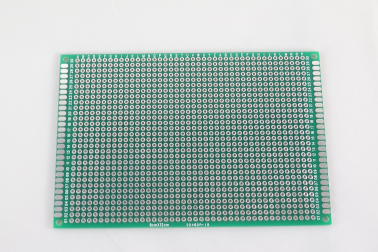
Here you can take the microcontroller (see black bar right side of image) mounted on a breadboard. They use standard wire sizes for the holes meaning you can take pretty much any chip and the pins will fit into the board here. You can try and replace the Arduino's standard ports and electronics with your own (with some additional resistors and capacitors) and you can very easily turn LEDs off and on again using this system.

Figure 6.11: A breadboard implementation using an AVR microcontroller



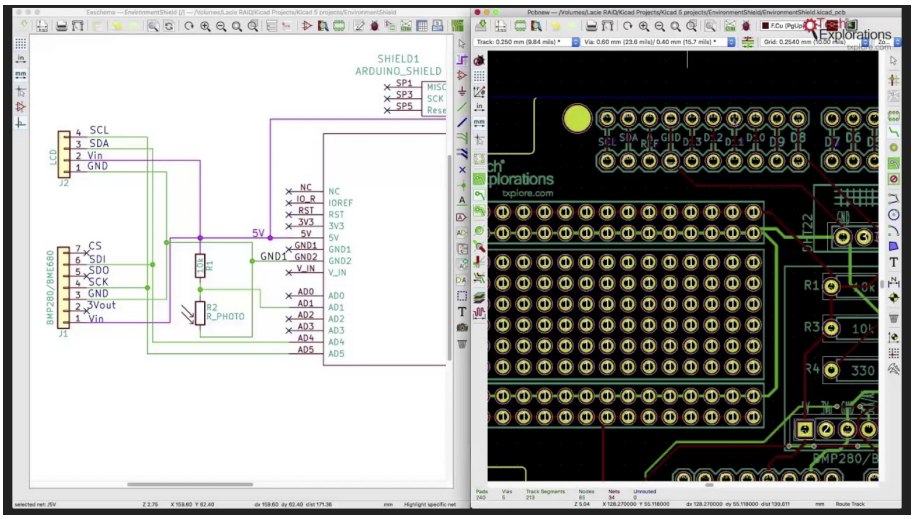
Sometimes you might then take your completed breakboard circuit and rebuit it by soldering it onto **prototype board**, which is a a generic printed circuit board containing just a grid of holes which you can wire together and to components as needed.

Figure 6.12: Prototype board



One you have designed your circuit (or as part of the design work, depending on your style), you then enter the design into a CAD program such as the open source **KiCAD** (*https://kicad-pcb.org/*) which can semi-automatically lay it out onto a 2D **printed circuit board (PCB)** for you. There are now many companies now that have webpages where you can send this PCB file, such as *www.pcbway.com* who take a couple of weeks before sending your printed circuit-boards back with all components already soldered onto them. You don't have to solder anymore, their soldering robot does that for you. (These robots may be interesting to study as robotics: traditionally they have just followed commands like car factory robots, but there is interest in making them more interactive and intelligent, for example by monitoring and actively testing the quality of the work they have done and correcting it as needed.)

Figure 6.13: Transferring a breadboard design to a topological circuit (left) then topographic layout (right) in KiCAD.



A particular design idiom useful in OSH is **mounting small PCBs onto larger PCBs**. This is most the literal and physical instantiation of the open source ideal of building your new system from other people's existing modules. For example, you will usually want to mount an Arduino along with other things all in one place. Mounting PCBs on other PCBs is easy, and is done using plaster **headers** and metal **pins**, like the ones already found on your Arduino connectors. (Sometimes called 'board-to-board connectors'.) They are usually in a standard (but American) size, compatible with most breadboards, using 0.1inch squares per pin. You can buy extra headers and place them between the two PCBs. A PCB is just a physical flat board and you can attach pretty much anything you like to it. Sometimes you will need to lift up the smaller board and hold it, floating about the big board, using **spacers** with bolts going through them and both boards to hold the board in place. Plastic spacers, bolts and nuts are often used here, so that if they come loose they don't cause short circuits. Occasionally you may need a bit of 3d printing to convince strange shaped large components to attach to your big PCB.

Figure 6.14: Left: Mounting several components PCBs on a large OSH PCB. (From Open PodCar project. Note the boxes of headers in background right).

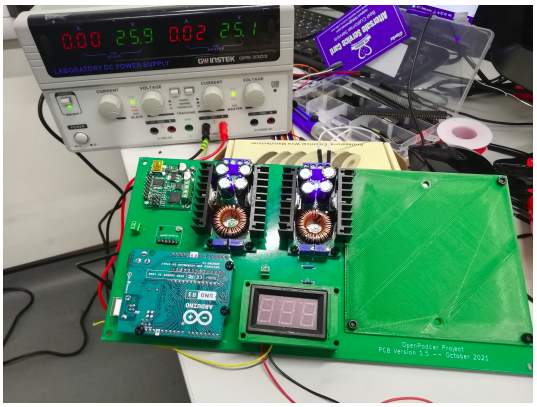


Figure 6.15: Board-to-board connectors. Left:soldered to both boards; Right: each board soldered to male-female connector pair.

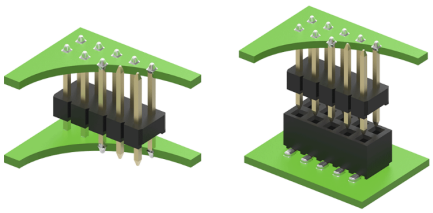
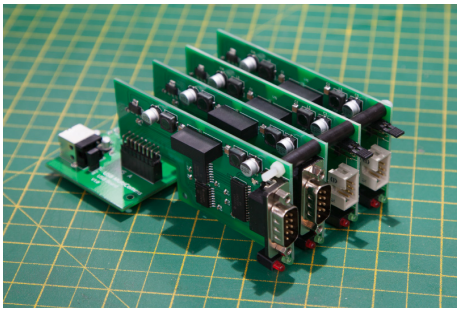


Figure 6.16: Physical board stacking with spacers and plastic bolts



When your OSH design later becomes more established, you might want to obtain the CAD files for the small board components and merge them onto your main PCB to avoid needing multiple boards. This is a macroscopic version of common practice in chip design, where chunks of other people's *designs* incorporated onto your 2D layout are called '**IP cores**'.

There are many small electronics projects on sites like **Kickstarter**. If you have an idea for a new Arduino based controller or musical instrument gadget its now very easy for student projects to raise some funding there to manufacture a batch and start selling them on ebay for real. Some of the components on our robots have been bought from companies like that. For example our inertial measurement unit (containing a gyroscope, accelerometer and compass) was designed and sold by a couple of students who put that together in this way.

6.6 Links

Many mechantronicists buy their electronics and mounting components from one of two major suppliers: RS and Farnell:

- *https://uk.rs-online.com/web/*
- *https://uk.farnell.com/*

RS and Farnell compete directly and often use the same product codes so that you can instantly compare and move orders between them. They are both set up to supply for both fast prototypes and bulk production, and can usually deliver next day.

Newer popular suppliers aimed more specifically at robotics and makers are

- *https://www.sparkfun.com/distributors*
- *https://www.robotshop.com/uk/*

A nice website hosting many OSH mechatronics projects which you can rebuild, extend, or use as inspiration, is:

- *https://www.hackaday.com*

6.7 Exercises

- Do the FreeCAD tutorials from *https://wiki.freecadweb.org/Tutorials*
- Do some KiCAD tutorials from *https://www.kicad.org/help/learning-resources/*
- Essay: What do **you** think is the best definition of and licence for OSH and why? (This is an easy topic to discuss in student reports to claim CRG marks for 'critical analysis' as it's an open and topical question.)

6.8 Further reading

- A Cohen. *Prototype to Product: a practical guide for getting to market.* O'Reilly. (how to scale up lab prototypes and manufacture for real-world use.)
- B. Cantwell Smith, *The Origin of Objects*, MIT Press 1998; and Lewis & Lewis's 1996 article *Holes* discuss the ontological challenges of representing holes in CAD and in philosophy. These challenges become critical when you want to use AI as part of the design process.

7 Alternative physics models

We have so far presented mechanics from a Newtonian perspective, which is useful in physical simulation software such as game engines and robotics simulations. However for other purposes in robotics, it can be useful to consider alternative perspectives.

Newtonian mechanics is located in Cartesian world space, but some robotics problems are more easily viewed in other spaces. **Lagrangian mechanics** takes **energy as fundamental** rather than force as in Newtonian mechanics, and may be viewed sometimes as taking an eternalist rather than sequentialist view of time. A major virtue it than it can operate in coordinate systems other than Cartesian physical space. For example, a multi-jointed robot arm's configuration may be modeled as a point in a high dimensional configuration space, where Lagrangian, but not Newtonian, methods can operate.

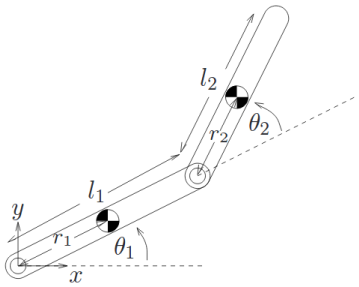
Newtonian and Lagrangian mechanics are equivilent in their predictions and modelling power, and both based on numerical values and locations in space. **Naïve Physics** is an alternative approach to AI and robotics models of the physical world which tries to be more human-like, forgoing exact numbers in favour of conceptual relationships.

7.1 Configuration space

So far we have done mechanics in world space – representing locations by coordinate vectors in the physical world. This works but can be quite time consuming to handle when, for example, dealing with stacks of coordinate transforms as in robot arms where each segment defines a new basis.

An alternative way to represent the state of a robot such as a robot arm is as a vector of parameters describing its internal configuration. For example in the two-element robot arm below, this can be represented as,

$$\boldsymbol{q} = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} \tag{7.1}$$



(2 element arm from
www.cds.caltech.edu/~murray/books/MLS/pdf/mls94-manipdyn_v1_2.pdf)

This representation may make more sense from the robot's own point of view, as for example the angles would be directly controlled and sensed by encoders on the motors.

We may want to convert between configuration space coordinates $\boldsymbol{q}$ and world coordinates $\boldsymbol{x}$ to describe the position of the end actuator. This is most efficiently done using a **Jabobean** matrix, which is a matrix that **converts** one basis into another,

$$\boldsymbol{J} = \frac{d\boldsymbol{x}}{d\boldsymbol{q}} = \begin{bmatrix} \frac{d\boldsymbol{x_1}}{d\boldsymbol{q_1}} & \frac{d\boldsymbol{x}_1}{d\boldsymbol{q_2}} \\ \frac{d\boldsymbol{x_2}}{d\boldsymbol{q}_1} & \frac{d\boldsymbol{x_2}}{d\boldsymbol{q_2}} \end{bmatrix} \tag{7.2}$$

And the derivatives in the matrix can be computed from the trigonometric description of the joints in the world.

The same matrix can be used to **convert velocities** in the two spaces, as

$$\boldsymbol{J} = \frac{\delta \boldsymbol{x}}{\delta t}\frac{\delta t}{\delta \boldsymbol{q}} \tag{7.3}$$

$$\boldsymbol{J}\frac{\delta \boldsymbol{q}}{\delta t} = \frac{\delta \boldsymbol{x}}{\delta t} \tag{7.4}$$

7.2 Lagrangian mechanics

7.2.1 Euler-Lagrange equations

7.2.1.1 1-dimensional case

(This section shows the derivation of the Euler-Lagrange equations. It is not necessary to know this derivation, rather the equation is usually used as a starting point for Lagrangian mechanics. The derivation is provided just so that you can see that it exists. It is based on Boas'.)

Suppose we want to find an optimal path $x(t)$ subject to some conditions, such as nailed down values $x(0) = a$ and possibly[1] also $x(T) = b$, and which minimises a total cost function, C, from summing a **Lagrangian**, L, along the path,

$$C = \int_{t=0}^{T} L(x(t), \dot{x}(t), t)dt \tag{7.5}$$

To find the best path, we consider a small perturbation from it,

$$X(t) = x(t) + \lambda n(t) \tag{7.6}$$

where $n(t)$ is any function subject to $n(0) = n(T) = 0$. This makes $X(0) = x(0)$ and $X(T) = x(T)$ so the start and end points are the same. $X(t)$ represents an alternative

[1] If we have two constraints in the past, such as $x(0) = a, \dot{x}(0) = c$, then we will obtain results for forward-time equations as in Newtonian mechanics. If one of the constraints is in the future, such as $x(T) = b$, then we will obtain a plan for how to get from the present to a particular future.

path from the start point to the end point which is close to, but a little perturbed from, the optimal.

Define,

$$C(\lambda) = \int_{t=0}^{T} L(X(t), \dot{X}(t), t)dt \tag{7.7}$$

Differentiate (the differential of an integral can just be pushed inside it; as the differential of a sum is equal to the sum of differentials of its components),

$$\frac{\delta C}{\delta \lambda} = \int_{t=0}^{T} [\frac{\delta}{\delta \lambda} L(X(t), \dot{X}(t), t)]dt \tag{7.8}$$

By the multivariate chain rule ($\frac{\delta a(b_1 ... b_n)}{\delta c} = \sum_{i=1}^{n} \frac{\delta a}{\delta b_i} \frac{\delta b_i}{\delta c}$),

$$\frac{\delta C}{\delta \lambda} = \int_{t=0}^{T} (\frac{\delta L}{\delta X} \frac{dX}{d\lambda} + \frac{\delta L}{\delta \dot{X}} \frac{d\dot{X}}{d\lambda})dt \tag{7.9}$$

As,

$$\frac{dX}{d\lambda} = n(t); \frac{d\dot{X}}{d\lambda} \dot{n}(t) \tag{7.10}$$

then,

$$\frac{\delta C}{\delta \lambda} = \int_{t=0}^{T} (\frac{\delta L}{\delta X} n(t) + \frac{\delta L}{\delta \dot{X}} \dot{n}(t))dt \tag{7.11}$$

Key point: at the optima, we require,

$$\frac{\delta C}{\delta \lambda} = 0 \tag{7.12}$$

So,

$$\int_{t=0}^{T} (\frac{\delta L}{\delta X} n(t) + \frac{\delta L}{\delta \dot{X}} \dot{n}(t))dt = 0 \tag{7.13}$$

Integrating the second term by parts ($\int a \frac{db}{dc} dt = [ab] - \int b \frac{da}{dc}$),

$$\int \frac{\delta L}{\delta \dot{X}} \dot{n}(t)dt = [\frac{\delta L}{\delta x} n(t)]_0^T - \int \frac{d}{dt} \frac{\delta L}{\delta \dot{x}} n(t)dt \tag{7.14}$$

$$= - \int \frac{d}{dt} \frac{\delta L}{\delta \dot{x}} n(t)dt \tag{7.15}$$

because the the first integrated term is zero as $n(0) = n(T)$ by definition of the perturbation function.

Substitute this back in to the zero equation,

$$\int_{t=0}^{T} (\frac{\delta L}{\delta x} - \frac{d}{dt} \frac{\delta L}{\delta \dot{x}}) n(t)dt = 0 \tag{7.16}$$

As $n(t)$ can be *any* function, this must be true for any $n(t)$. This is only possible if the rest of the integral is zero at every point,

$$\frac{\delta L}{\delta x} - \frac{d}{dt}\frac{\delta L}{\delta \dot{x}} = 0 \tag{7.17}$$

or,

$$\frac{d}{dt}\frac{\delta L}{\delta \dot{x}} = \frac{\delta L}{\delta x} \tag{7.18}$$

This is the **Euler-Lagrange equation**.

It tells us how one quantify (LHS) changes over time, according to another one (RHS). Hence it can be used to simulate a system over time, as an equation of motion, by computing one step at a time in sequence.

7.2.1.2 *N*-dimensional case

Boas shows we get a set of N similar simultaneous equations,

$$\frac{d}{dt}\frac{\delta L}{\delta \dot{x_i}} = \frac{\delta L}{\delta x_i} \tag{7.19}$$

For example in 2D, with path having components $[x(t), y(t)]$,

$$\frac{d}{dt}\frac{\delta L}{\delta \dot{x}} = \frac{\delta L}{\delta x} \tag{7.20}$$

$$\frac{d}{dt}\frac{\delta L}{\delta \dot{y}} = \frac{\delta L}{\delta y} \tag{7.21}$$

And similarly for 3D,

$$\frac{d}{dt}\frac{\delta L}{\delta \dot{x}} = \frac{\delta L}{\delta x} \tag{7.22}$$

$$\frac{d}{dt}\frac{\delta L}{\delta \dot{y}} = \frac{\delta L}{\delta y} \tag{7.23}$$

$$\frac{d}{dt}\frac{\delta L}{\delta \dot{z}} = \frac{\delta L}{\delta z} \tag{7.24}$$

For robots in configuration spaces (joint angles etc) there could be many more dimensions.

7.2.2 Principle of least action

Newtonian mechanics can be reformulated if we take, as an axiom, that systems move in paths minimising the integral of a **particular Lagrangian** whose integral is called the **action**,

$$L = K - V \tag{7.25}$$

where K and V are kinetic and potential energy (V is the traditional symbol for potential energy, as used in Voltage for electrical potential). This works both for single objects in Cartesian space, but also for Configuration Spaces, unlike standard Newtonian mechanics.

Example: for a thrown rock, moving under gravity, in 3D,

$$K = \frac{1}{2}m(\dot{x}^2 + \dot{y}^2 + \dot{z}^2), V = mgz \tag{7.26}$$

$$L = \frac{1}{2}m(\dot{x}^2 + \dot{y}^2 + \dot{z}^2) - mgz \tag{7.27}$$

Create all the derivatives which appear in the 3D Euler-Lagrange equations (we are just a user of the equations now),

$$\frac{\delta L}{\delta x} = \frac{\delta L}{\delta y} = 0; \frac{\delta L}{\delta z} = mg \tag{7.28}$$

$$\frac{\delta L}{\delta \dot{x}} = m\dot{x}; \frac{\delta L}{\delta \dot{y}} = m\dot{y}; \frac{\delta L}{\delta \dot{z}} = m\dot{z} \tag{7.29}$$

Plug these in to the Euler-Lagrange equations and we recover the standard Newtonian second-order differential **equations of motion**,

$$\frac{d}{dt}m\dot{x} = 0 \implies \frac{d\dot{x}}{dt} = 0 \tag{7.30}$$

$$\frac{d}{dt}m\dot{y} = 0 \implies \frac{d\dot{y}}{dt} = 0 \tag{7.31}$$

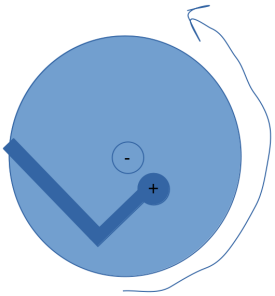
$$\frac{d}{dt}m\dot{z} = mg \implies \frac{d\dot{z}}{dt} = g \tag{7.32}$$

What was the point of all this when we could have just written now the Newtonian version directly? In this simple example we could have done that. But when we have more complex configuration spaces it will be harder. And importantly for path planning – we have shown that we can derive second order differential equations of motion for **any** Lagrangian integral cost function.

For example, for a robot arm in configuration space,

$$\boldsymbol{q} = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}, \tag{7.33}$$

Figure 7.1: Robot arm mounted on spinning disc



we might obtain the equations of motion in terms only of the θ values, without any reference to world coordinates,

$$\frac{d\dot{\theta_1}}{dt} = f_1(\theta_1, \theta_2, \dot{\theta_1}, \dot{\theta_2}) \tag{7.34}$$

$$\frac{d\dot{\theta_2}}{dt} = f_2(\theta_1, \theta_2, \dot{\theta_1}, \dot{\theta_2}). \tag{7.35}$$

(History: "Euler equation" was the pure maths version; "Lagrange's equations" were the 3D mechanics application. Quantum Field Theory turns out to be most nicely written in this form, with the Standard Model usually stated as a very large set of Lagrangian terms defining all the different potential energies produced by its many interactions.)

7.2.3 Lagrangians in configuration space

7.2.3.1 Rotating reference frame

Suppose we have a 2D robot arm *mounted* on the inside of fast-spinning drum (a centrifuge), and rotating along with it, such as might be used for mixing soup in a rotating vat on an industrial scale. The arm has many joints which are free to move. Suppose the end effector (i.e. the last actuator in the chain, usually the "business end" which is going to interact with the rest of the world such as stirring or sampling the soup) has a positive charge, and there is a negative charge in the center of the cylinder. (More realistically, these would be virtual charges being used to plan control of the arm to bring it from its current location to the negative charge location.)

Suppose we stand in the center of the cylinder to observe the motion of the end effector in radial coordinates, creating a rotating reference frame – what does the motion of the end effector look like to us from here?

The frame is rotating with angular velocity ω_0. The (stationary) Cartesian (x, y, z) coordinates are related to the rotating (r, θ, z) coordinates by,

$$x = r\cos(\theta + \omega_0 t) \tag{7.36}$$

$$y = r\sin(\theta + \omega_0 t) \tag{7.37}$$

The velocities in the Cartesian frame are,

$$\dot{x} = \dot{r}\cos(\theta + \omega_0 t) - r(\dot{\theta} + \omega_0)\sin(\theta + \omega_0 t) \tag{7.38}$$

$$\dot{y} = \dot{r}\sin(\theta + \omega_0 t) + r(\dot{\theta} + \omega_0)\cos(\theta + \omega_0 t) \tag{7.39}$$

The kinetic energy of the end-effector is given by,

$$K = \frac{1}{2}m(\dot{x}^2 + \dot{y}^2) \tag{7.40}$$

$$= \frac{1}{2}m(\dot{r}^2 + r^2(\dot{\theta} + \omega_0)^2) \tag{7.41}$$

The Lagrangian is composed from the kinetic energy and the potential energy,

$$L = \frac{1}{2}m(\dot{r}^2 + r^2(\dot{\theta} + \omega_0)^2) - V(r) \tag{7.42}$$

The derivatives needed as ingredients of the Euler-Lagrange equation are,

$$\frac{d}{dt}\frac{\delta L}{\delta \dot{\theta}} = \frac{d}{dt}mr^2(\dot{\theta} + \omega_0) = m(2r\dot{r}(\dot{\theta} + \omega_0) + r^2\ddot{\theta} \tag{7.43}$$

$$\frac{\delta L}{\delta \theta} = 0 \tag{7.44}$$

$$\frac{d}{dt}\frac{\delta L}{\delta \dot{r}} = m\ddot{r} \tag{7.45}$$

$$\frac{\delta L}{\delta x} = mr(\dot{\theta}^2 + \omega_0^2 + 2\theta\dot{\omega}_0) - \frac{\delta V}{\delta r} \tag{7.46}$$

Plugging these into Euler-Lagrange and rearranging gives the equations of motion,

$$\ddot{\theta} = -\frac{2\dot{r}}{r}(\dot{\theta} + \omega_0) \tag{7.47}$$

$$\ddot{r} = r(\dot{\theta}^2 + \omega_0^2) + 2r\dot{\theta}\omega_0 - \frac{\delta V}{\delta r} \tag{7.48}$$

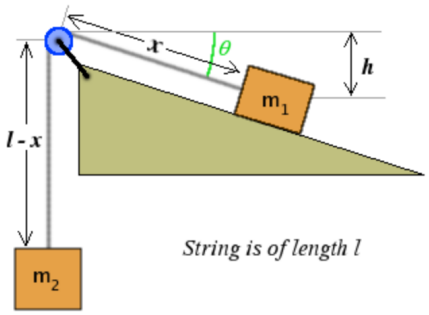
The right hand side of the $\ddot{\theta}$ equation is the **tangential Coriolis force**; it depends on the radial velocity, and the sum of the angular velocity of the particle in the frame and rotation rate of the frame.

The first term on the right of the $\ddot{r}$ equation is the **centrifugal force** due to the particle's own motion and the rotation of the frame. The second term is the **radial Coriolis force**; it is proportional to ω_0, so for a non-rotating frame, it vanishes. The third term in the central force we started with.

The central force that we started with is the only "real" force present. The others are "virtual" forces which "appear to exist" only because of our point of view in the rotating frame. This is a non-Newtonian frame as Newton's first law is not satisfied.

7.2.3.2 Ramp and string

Here we have two weights m_1 and m_2 attached together with a string of length l. One is on a ramp, the other is hanging over the edge by the string going over a pully. Both weights are intially stationary. We assume there is no friction.



With x units of string between the pully and weight m_1, it drops $h = \sin\theta$ units below the level of the pulley.

So the (gravitational) potential energy of m_1 is

$$V_1 = -m_1 g h = -m_1 g x \sin\theta \tag{7.49}$$

When m_1 slides x units down the ramp, weight m_2 must rise x units due to their string connection. So the potential energy of m_2 is

$$V_2 = m_2 g x \tag{7.50}$$

The kinetic energy of the system is

$$K = \frac{1}{2}(m_1 + m_2)\dot{x}^2 \tag{7.51}$$

So the Lagrangian is

$$L = \frac{1}{2}(m_1 + m_2)\dot{x}^2 - gx(m_1 \sin\theta - m_2) \tag{7.52}$$

The partial derivatives needed for the Euler-Lagrange equation are

$$\frac{\delta L}{\delta \dot{x}} = (m_1 + m_2)\dot{x} \tag{7.53}$$

$$\frac{d}{dt}\frac{\delta L}{\delta \dot{x}} = (m_1 + m_2)\ddot{x} \tag{7.54}$$

$$\frac{\delta L}{\delta x} = g(m_1 \sin\theta - m_2) \tag{7.55}$$

We will find that making this substitution helps a lot,

$$M_1 := \frac{m_1}{m_1 + m_2} \tag{7.56}$$

$$M_2 := \frac{m_2}{m_1 + m_2} \tag{7.57}$$

Then the equation of motion, from the Euler-Lagrange equation, is

$$\ddot{x} = g(M_1 \sin\theta + M_2) \tag{7.58}$$

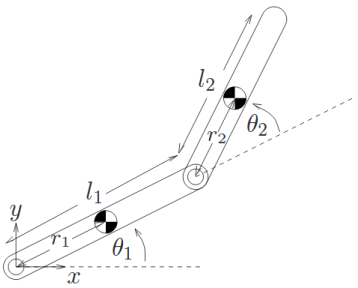
From this, we can see that there are three cases:
First: if $m_1 \sin\theta > m_2$, then weight m_1 slides down the ramp.
Second, if $m_1 \sin\theta = m_2$, then the system is in balance and nothing moves.
Third, if $m_1 \sin\theta < m_2$, then weight m_1 is pulled up the ramp.
We can obtain a closed form solution for the positions by integrating the equation of motion twice to give,

$$\dot{x} = gt(M_1 \sin\theta - M_2) \tag{7.59}$$

$$x = \frac{1}{2}gt^2(M_1 \sin\theta - M_2) + x_0 \tag{7.60}$$

where x_0 is a constant, representing the starting location of m_1.

7.2.3.3 Robot arm



Lagrangians can be used to obtain the equations of motion for robot arms, where each link of the arm creates a new coordinate frame for the next. As we move up the chain of links, similar virtual forces to those seen in the centrifuge will be seen, becoming quite complex as they compound from one another.

To **control** such an arm we will need to compensate for many or all of these forces, which is quite hard... see advanced modules!

See *https://www.cds.caltech.edu/~murray/books/MLS/pdf/mls94-manipdyn_v1_2.pdf* , p164

7.2.4 From Lagrangians to robot path planning

When we use Lagrangians to simulate motion, we know the state at time $t = 0$ but not at the end state $t = T$. The equations tell us how to forward simulate the state by computing accelerations, velocities and positions at each step in sequence.

However if we do constrain the end state at $t = T$, then we can use the Lagrangian method as a robotic path planner. Rather than start at a state and see where it goes, we ask what is the best path to get from the start to a particular goal.

The "best" path is again defined by a cost function. Unlike in mechanics, we can choose whatever cost we like here. For a vehicle such as an autonomous car, we might define costs including sharpness of curves, and a cost of time for the journey, as well as costs for hitting obstacles on a map.

$$C = \int_{t=0}^{T} L(x(t), \dot{x}(t), t)dt \tag{7.61}$$

Again apply Euler-Lagrange equations,

$$\frac{\delta L}{\delta x_i} = \frac{d}{dt}\frac{\delta L}{\delta \dot{x}_i} \tag{7.62}$$

To obtain differential equations of motion. However when both ends of the path are constrained these may be difficult or impossible to solve analytically.

So instead, we might abandon Euler-Lagrange theory and try to solve the Lagrangian equation numerically or heuristically. Here we are just using Lagrangian mechanics as a weak inspiration for robot path planning methods.

A particular class of methods starts with a guessed solution path, and quantises it into a set of discrete blobs. Then iteratively update the position of each blob individually so as to minimize its contribution to the total cost. This is similar to physical models of an elastic band being pulled into its shortest configuration between two points.

7.3 Naïve Physics as AI

As an antidote to the heavy maths (text from Alan Blackwell's thesis,
https://www.cl.cam.ac.uk/~afb21/):

"Naïve Physics" is a term coined by Hayes in 1978, to describe his approach to developing a "large-scale formalism" of ***commonsense knowledge*** *about the world. This concern with real world knowledge can be related to a general awareness amongst AI workers that future progress in AI depends on intensive knowledge being made available to reasoning systems.*

The aim of naïve physics is to formally describe the world in the way that most people think about it, rather than describing it in the way that physicists think about it. This description should attempt ***reasonable completeness*** *- that is, it should describe a significant portion of the way we understand the world, rather than just small pieces.*

The use of the word "naïve" indicates that this description must include commonsense knowledge that is normally taken for granted in formal physics, and it therefore may

include elements outside what we consider to be the field of physics. It may also choose to describe phenomena in a way that is familiar to "the man in the street", but would not be considered appropriate to a physicist. Two examples are the "force" of ***sucking****, and* ***"impetus"*** *theories of motion, both of which adequately describe everyday phenomena, and are prevalent theories amongst intelligent people, even though they are considered to be inappropriate for physicists.*

The "Naive Physics Manifesto", and the its revision in the "Second Naive Physics Manifesto", proposed the construction of a formalisation of commonsense knowledge which covered a broad range of knowledge using a common framework, and included dense factual detail. It specifically did not recommend the construction of programs or new formal description methods. It did dwell on the importance of spatial representation, although it excluded the sort of spatial reasoning that is necessary to plan physical motion.

General principals of Naïve Physics:

- AI should model the world as humans do, with **macroscopic objects** and **laws**. This is far removed from "fundamental physics". Work with everyday sized "objects" and model the logic of the phenomenal world. Newton's laws (and equivalent formulations such as Lagrange's) assume the existence of such "objects" but have difficulties when objects change or go in and out of existence.
- **Qualitative physics** – eg. dealing with signs and magnitudes rather than numbers
  - eg. "falling rocks go down, balloons go up"
  - eg. "if you hit glass with a rock it will break"
  - eg. "releasing the pressure makes the size reduce"
  - working with inequalities, categories, and signs
  - qualitative "places" rather than coordinates, "inside the room", "on the shelf"
  - emphasize boundaries of objects as place boundaries
- **the nature of "objects"** remains poorly or not at all understood in AI (and philosophy)
  - when are two objects **identical** (i.e. the same object) or different?
    - an object can change its properties over time but remain the same object (the cup turns red)
      - an object can change its parts but remain the same object (broom gets a new handle and a new body)
      - how can an object split into two or merge from two ? (branch of an apple tree grows and becomes two)
  - how do objects relate to "**classes**" or "kinds":
    - when do objects fall into membership of a "class" whose properties can be used to predict theirs?

        * when do we form a "class" from several observations of different "objects"?
    - what is an "**event**" vs an "object" and how do they relate? Perduranist approach taking extended object "histories" as fundamental.

- Modeling **common sense** laws:
    - very **hard** to represent and complete
        * common sense can **contradict** Newtonian physics and itself, eg.
            · energy is needed to maintain motion in real-life vehicles (impetus)
            · heavy objects do fall faster than light ones (Galilean physics)
        * and contradict one another too

In general – the physics we have seen in this half module has been very numerical and precise. It is useful for physics simulations and for robot control. But it remains unclear whether robots can or should use such detailed models for higher level planning. The computational complexity of planning may be too great for completely optimal quantative methods at these levels, so heuristics such as Naïve Physics may be useful there, as humans appear to use in their solutions. Qualitative reasoning has become an AI industry with its own conferences but can be hard to merge with the dominant numerical approaches to robotics.

Philosophers have argued over the questions posed by Naïve Physics for thousands of years, famously starting from Aristotle, and have not reached agreement. So it is unlikely that we will be able to build AI from their solutions in the near future. But if AI research does produce workable models, then perhaps that would show that it is superior to academic Philosophy for answering them – game on! The *Naïve Physics Manifesto* confidently predicted this would be finished in the 1980s but it hasn't happened ... yet.

7.4 Exercises

- Invent variations on the rotating reference frame examples based on robot of your own interests, and solve using Lagrangian mechanics
- Essay question: "Naive physics is more realistic for robotics than Lagrangian mechanics – Discuss."

7.5 Further reading

7.5.1 Recommended

Lagrangian mechanics:

- Feynman lectures on Physics, Vol II, Chapter 19, The Principle of Least Action.
    - http://www.feynmanlectures.caltech.edu/II_19.html

- L. Susskind and G. Hrabovsky. *Classical Mechanics: the theoretical minimum.* Penguin, 2014.
- *Structure and Interpretation of Classical Mechanics* book
  - https://groups.csail.mit.edu/mac/users/gjs/6946/sicm-html/
- *From conservation of energy to the principle of least action: A story line* written by Jozef Hanc and Edwin F. Taylor.

Naïve physics:

- PJ Hayes, 1983. *The Second Naïve Physics Manifesto.*
  - www.academia.edu/722721/The_second_naive_physics_manifesto
- Barry Smith and Roberto Casati. *Naïve Physics: An Essay in Ontology.* Philosophical Psychology, 7/2 (1994), 225-244.
  - http://ontology.buffalo.edu/smith/articles/naivephysics.html

8 Mini projects

The assessment will ask you to report on exactly one of these projects. You are encouraged to work in groups to complete one of them during the module, in workshops and in your own study time. We can provide basic hardware for each project. You may also wish to source and purchase additional hardware yourself in some cases. Please discuss your project choices, ideas, and group formation with the lecturer as there are sometimes interesting ways to link them to current research at Lincoln which enables them to access research resources and leverage existing work.

8.1 Robot radio: communication and localisation

Software Defined Radio (SDR) is a recent technology which allows a single hardware radio transceiver to act on almost any radio frequency. An SDR board can be used along with open source software GNURadio to control transmit and receive on single frequencies or bands of frequencies. The project will explore practical SDR by building and explaining an implementation of a protocol of your choice. For example, you could try to control a radio controlled children's toy, commercial robot, or car security key by reverse engineering and re-implementing its protocol. And/or you could use the same hardware to build an RTLS by triangulating position from signal delays or amplitudes. An advanced project could consider how to do both comms and localisation efficiently at the same time, for example by looking at RTK-like temporal shifts in signals otherwise used for comms.

Bill of materials: HackRF board x2, antenna x2, laptop PC. Optional: children's remote control toy.

Example exam question option: Discuss how radio communications can be used to control a robot, giving detailed examples of hardware and software systems from a real application.

8.2 Physics simulation: the joy of friction

The world's most enjoyable activities all involve friction, and it is a current area of interest in physical robot simulation as robotics is moves from 2D to 3D environments. 3D environments have slopes and different surface types such as tarmac and grass. Some robots will roll or slide down a hill if not actively controlled to remain still, while others have different friction properties enabling them to stay still without active control. This project will deep dive into the physics of friction such as the different detailed models available in Gazebo/ODE. It might work with a research project to improve an existing

robot simulation to use the latest state of the art friction models for these situations. Almost every robotics project in the world needs a physical simulation and hires an expert person to build and look after it: this project would make you quite employable as that expert.

Bill of materials: PC running Gazebo.

Example exam question: Describe the process of simulating a physical robot, giving detailed examples of methods and challenges from a real application.

8.3 Motor control: Build a physical robot

Lincoln is working on an open source hardware (OSH) motor control system, including embedded Arduino PID software and the OSMC Open Source Motor Controller driver. We have geared rotary and linear actuators available, suitable for building agricultural sized robots and basic robot arms. In this project you will design and build an open source robot from these components, either a wheeled mobile robot and/or an arm. You can use an aluminium profile system for the main structure and mounts. You might also get involved in improving the exsiting OSH designs, which are public for the world to use. Applications could include agriculture, last mile freight delivery, or an entry for *Robot Wars* or similar tournment.

Bill of materials: Elgoo Arduino dexute kit including motor and motor driver. Larger motors, gears, OSMC, alu profile.

Example question: Describe the theory and practice of motors and motor control, including electromagnetic theory, PID control, and embedded systems used in a real application.

8.4 Robot music: Automated guitar

Our previous MSc student Andrew Henry has published an open source hardware (OSH) design for a robot guitar player . Lots of potential future improvements were identified and could be worked on. At the mechatronics level, we'd like it to play faster and smoother, with better timing, and be easier for new users to build at home. We'd like to add string bends, vibrato, and the ability to pick in different locations and styles. At the AI level, we'd like it to read MIDI input of a polyphonic song, and figure out the optimal guitar fingering to use, and convert them into finger and string commands.

Lecture: https://www.youtube.com/watch?v=QGKTtiIZPBg

Paper: eprints.lincoln.ac.uk/id/eprint/45327/1/AutomatedGuitarICMC.pdf

Source: gitlab.com/Andrew_Henry/automated-guitar

Bill of materials: Elgoo Arduino kit, 3d printer.

Example exam question: Describe the theory and practice of designing and building a musical controller from robotics components.