WROCŁAW UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF ELECTRONICS

FIELD: Control Engineering and Robotics (AIR)
SPECIALIZATION: Embedded Robotics (AER)

MASTER OF SCIENCE THESIS

Control system for
two HOG wheel mobile robot

System sterowania robota mobilnego
z dwiema wirującymi półsferami

AUTHOR:
Jędrzej Boczar

SUPERVISOR:
dr inż. Robert Muszyński

GRADE:

WROCŁAW 2019

Streszczenie

Głównym przedmiotem pracy jest analiza procesu implementacji algorytmów sterowania robotem mobilnym na podstawie jego modelu matematycznego. Temat ten przeanalizowany został na przykładzie robota Hogger^2, będącego wynikiem poprzednich prac prowadzonych na Politechnice Wrocławskiej. Robot ten, ze względu na niekonwencjonalny sposób poruszania się wykorzystujący dwie wirujące półsfery, wymaga złożonych algorytmów sterowania, których implementacja nastręcza problemów niespotykanych w typowych robotach mobilnych. Z tego względu, metodyka potrzebna do implementacji sterowania poddana została wnikliwej analizie.

Praca skupia się przede wszystkim na aspektach związanych z procesem generacji kodu źródłowego na postawie modelu matematycznego robota. Na początku przedstawiona została koncepcja robota Hogger^2, wraz z jego modelem matematycznym i algorytmami sterowania. Następnie opisane zostały teoretyczne aspekty implementacji algorytmów sterowania w systemach wbudowanych oraz samego procesu generacji kodu. Powyższe informacje wykorzystane zostały do implementacji sterowania na konkretnym przykładzie robota Hogger^2. W ostatniej części pracy, poruszone zostały dodatkowo aspekty sprzętowe związane z implementacją oraz przedstawiono konieczne do wprowadzenia w robocie zmiany.

Contents

For typesetting this thesis, the LaTeX document preparation system has been used. LaTeX has been developed by L. Lamport [3], and is an overlay on top of the TeX system [1]. Mathematical fonts called AMS Euler which have been used in this document, have been commissioned by the American Mathematical Society and designed by H. Zapf [2] with the assistance of D. Knuth and his students. The URW Palladio font, used for roman text, is a clone of H. Zapf's old-style typeface called Palatino [5]. Typesetting of sans-serif monospaced text has been done using Inconsolata font, created by R. Levien [4].

[1] D. E. Knuth. *The TeXbook*. Computers & typesetting. Addison-Wesley, 1986.

[2] D. E. Knuth and H. Zapf. *AMS Euler — a new typeface for mathematics*. Toronto: University of Toronto Press: Scholarly Publishing, 1989.

[3] L. Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, 1994.

[4] R. Levien. *Inconsolata*. URL: https://levien.com/type/myfonts/inconsolata.html.

[5] *Linotype Palatino nova: A classical typeface redesigned by Hermann Zapf*. Linotype Library GmbH. 2005.

Glossary

ADC	analog-to-digital converter
API	application programming interface
BLDC	brushless direct current [electric motor]
CAS	computer algebra system
CPU	central processing unit
CSE	common subexpression elimination
DAC	digital-to-analog converter
DC	direct current
DSP	digital signal processing
FPU	floating point unit
GCC	GNU compiler collection
HAL	hardware abstraction layer
HOG	hemisphere omnidirectional gimbaled
I^2C	inter-integrated circuit [bus]
IMU	inertial measurement unit
MCU	microcontroller unit
OS	operating system
PC	personal computer
PWM	pulse-width modulation
RC	remote control
RTOS	real-time operating system
SLAM	simultaneous localization and mapping
SPI	serial peripheral interface
UART	universal asynchronous receiver-transmitter
WUST	Wrocław University of Science and Technology

Chapter 1

Introduction

The primary motivation for this work has been Hogger^2 robot developed at the Wrocław University of Science and Technology (WUST). This robot is quite exceptional because its motion is based on the concept of hemisphere omnidirectional gimbaled (HOG) drive. Although today this drive system is almost completely forgotten, its invention dates back to year 1938 [55]. Because of its unique properties, research on HOG drive robots has been done at WUST [16, 20, 21, 39], as well as in other works, e.g. [1, 9].

Hogger^2 motion system has a great potential, however its capabilities come at the cost of complicated control. As a continuation of previous research, the algorithms developed in [21] were to be investigated and implemented in this work. In the course of the investigation however, the complexity of control algorithms required for Hogger^2 control and the number of aspects that have to be taken into account during the process of implementation has been realised. This led to a shift in the main subject of this work to further investigate the methodology of the process of control algorithms implementation in a real robotic system.

The theoretical background for designing control algorithms for mobile robots is a wide, yet already extensively covered subject. Notable works, including [30, 42, 45, 46], treat about solving the control task for general and special cases, providing algorithms that perform well, even for highly nonlinear control objects. Even so, the majority of control implementations use simple solutions that often underperform, but are easier to implement, and as a result are chosen over more sophisticated algorithms. This is often the case, because complex solutions require careful design and may easily become overwhelming, when other aspects of the implementation emerge, e.g. real-time operation and numerical precision. In majority of works, the process of algorithm implementation is often pushed to the background or even passed over, since it is considered trivial.

In practical applications, well established, certified systems, for instance MATLAB [54] or the de facto standard in automotive industry, AUTOSAR [4], are being used and provide powerful and convenient tools for building complex control systems and implementing them on real hardware. One notable part of the implementation process is the usage of tools that can generate low-level code for the target devices. Some examples where code generation plays an important role in the implementation process include the DC motor control laboratory stand at WUST [27] or Kugle ball-balancing robot developed as a part of an exemplary master thesis project [19]. Then again, in smaller projects such extensive tools may not always be of choice, bringing

a lot of development overhead, when compared to relatively straightforward tasks that have to be implemented.

This thesis aims to explore and describe the methodology that can be used to implement standard robot control algorithms, considering the problems originating from the uncovered complexity of seemingly simple procedures and from hardware related issues that may be easily overlooked, when the mathematical description becomes the primary focus area. The main field of the implementation process covered in this work is the procedure of generating computer program code corresponding to the previously developed mathematical description. The process of code generation is often used during the broader process of implementing a control algorithm.

The content of this work is organised in the following way. In Chapter 2, Hogger^2 robot is introduced to give a better understanding of the kind of problems that are to be faced. The mathematical model and control algorithms developed for the robot in previous works are outlined and the results of already performed simulations are summarised. Chapter 3 treats about the methodology that can be used for implementing control algorithms using the code generation approach, as well as it discusses common design considerations when using embedded systems for the task of robot control. The background introduced in Chapters 2 and 3 is then applied in Chapter 4 for the specific problem of implementing Hogger^2 robot control, while describing the subsequent steps of the process. In Chapter 5, the aspect of hardware required for the task of robot control is explored and the hardware available in the existing Hogger^2 prototype is analysed. The final summary is presented in Chapter 6, providing some possible directions of future work related to the issues that have been raised in this thesis.

Chapter 2

Hogger^2 robot

Implementation of most control algorithms requires knowledge about the controlled object and its mathematical model. The subject of this work is to implement control algorithms for Hogger^2 robot. The concept of Hogger^2 was already analysed in previous works [16, 20, 21]. This chapter summarises the findings, focussing on the aspects that are important from the perspective of control algorithms implementation.

First, the concept of Hogger^2 robot will be discussed. Later the robot kinematic models will be described, along with the control algorithms proposed for specific models. Finally, the results of already performed simulations will be summarised.

This chapter includes relatively spacious mathematical expressions. For the sake of legibility, the t argument in functions dependant on time (e.g. $q(t)$) will be omitted (resulting in just q). Moreover, trigonometric functions $\sin(x)$, $\cos(x+y)$ and $\tan(x)$ will be written in shorter form, s_x, c_{x+y} and $\tan_x$ respectively.

2.1 HOG drive

Hemisphere omnidirectional gimbaled drive (HOG) is a concept dating back to the year 1938 or even earlier [55]. A gimbal is a mechanical part that allows the rotation of an object around a single axis [50]. HOG drive uses a spinning hemisphere mounted on two gimbals. This makes it possible to rotate the hemisphere around the two axes that are perpendicular to its spinning axis [51]. The concept of HOG drive has been illustrated in Figure 2.1.

In HOG drive it is assumed, that at any point in time, the hemisphere has one contact point with the ground. The linear velocity of this point determines the resulting velocity relative to the ground that causes movement of the vehicle. When the spinning axis is perpendicular to the ground, the resulting linear wheel velocity is zero. Manipulating the rotation of the hemisphere allows to generate velocities in any direction. Additionally, because the spinning movement holds substantial kinetic energy, it is possible to convert the angular movement velocity into linear velocity in a short time, which gives the possibility to achieve high acceleration.

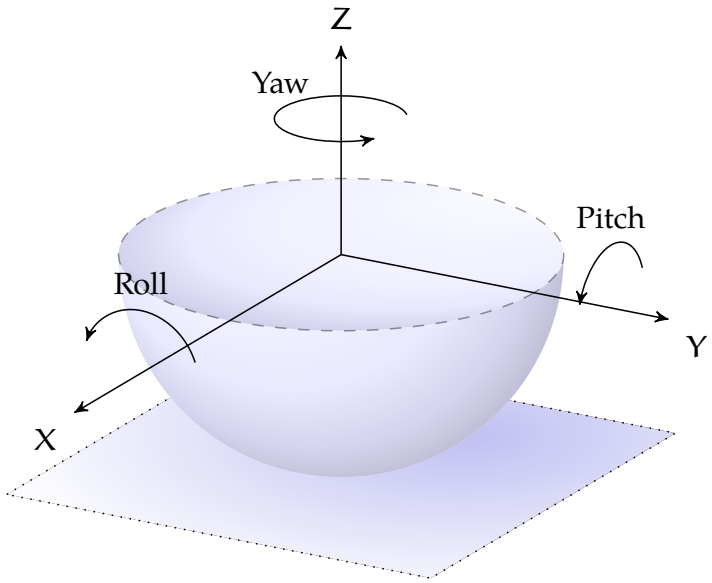


Figure 2.1: The concept of a HOG wheel. The hemisphere spins around Z axis and can be rotated around X and Y axes

2.2 Two HOG wheel robot

The concept of the vehicle described in [55] assumes usage of only one HOG wheel and two regular wheels. Based on this vision, a real robot called Hogger has been created at WUST. The robot has been shown in Figure 2.2a and was described in [39]. This idea is further extended in the work [20] by considering a vehicle that moves using two HOG wheels. The vehicle has been called Hogger² and its prototype has been built and then described in [15, 16]. Hogger² robot has been shown in Figures 2.2b and 2.2c. Its construction overcomes the limitations caused by having regular wheels, allowing to fully benefit from the unique properties of the HOG drive. On the other hand, such a drive complicates the task of robot control, requiring additional coordination of the movement of both HOG wheels, which can be done using proper mathematical models. Such models have been developed for Hogger² in [20–22] and will be summarised below.

2.2.1 Full kinematic model

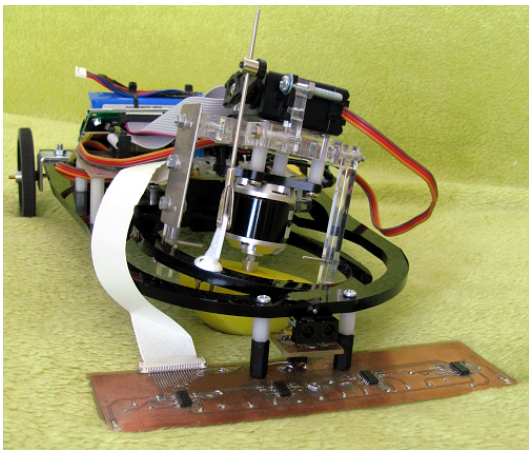
As shown in [20], Hogger² configuration can be described using 9 variables denoting its position and orientation in the world frame (x, y, θ_0), and the angles of rotation of each HOG wheel $(\varphi_i, \theta_i, \psi_i)$. The variables ψ_i denote the angle of the spinning movement of a hemisphere (yaw angle) and φ_i, θ_i are the angles of hemisphere rotation (roll/pitch respectively). The resulting vector is denoted by

$$q = (x, y, \theta_0, \varphi_1, \theta_1, \psi_1, \varphi_2, \theta_2, \psi_2)^T. \tag{2.1}$$

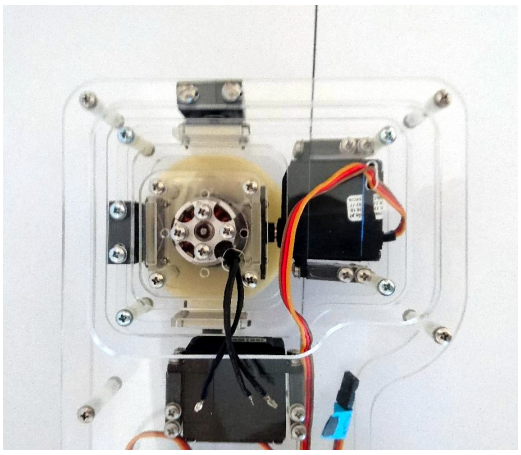
The configuration variables have been illustrated in Figures 2.3 and 2.4.

The kinematic model of a nonholonomic mobile robot is described by the equation

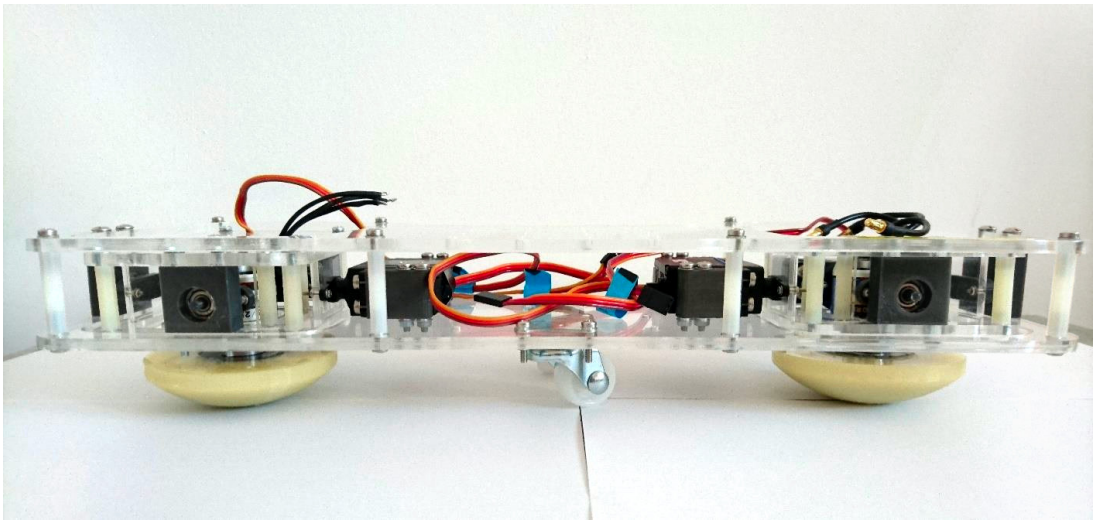
$$\dot{q} = G(q)\eta, \tag{2.2}$$



(a) Hogger robot



(b) Hogger^2 robot — top view



(c) Hogger^2 robot — front view

Figure 2.2: HOG drive based robots

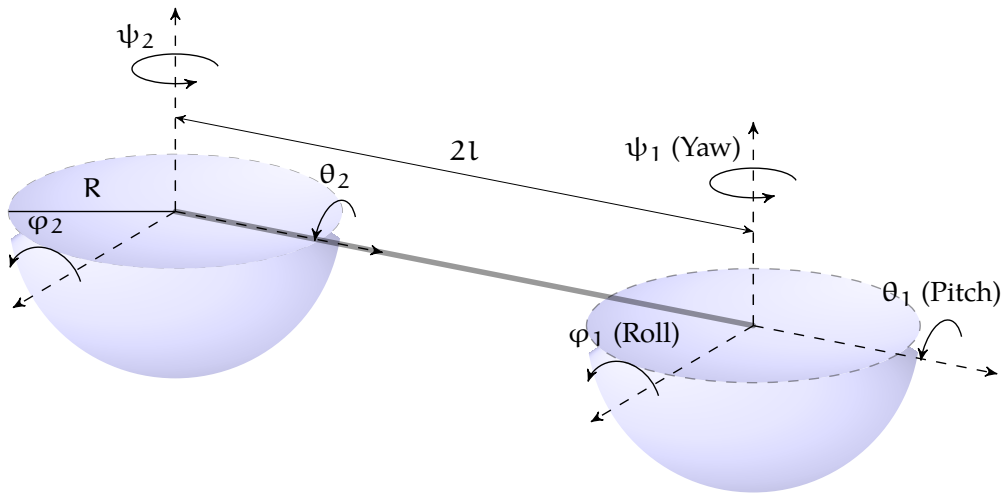


Figure 2.3: Hogger^2 hemisphere angles description

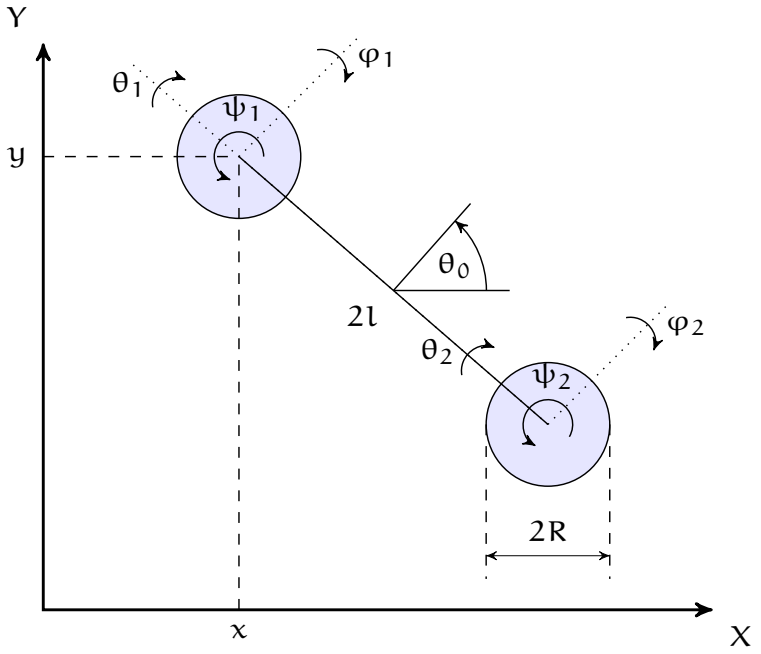


Figure 2.4: Hogger[2] configuration variables

where η denotes the control inputs vector, $\dot{q}$ is the velocity vector in the robot configuration space and $G(q)$ is a matrix composed of generator vectors available for the given nonholonomic constraints [46].

The kinematic model of Hogger[2] robot has been derived with the assumption of nonholonomic constraints of no slip at the contact point of hemisphere with the ground. Two different models have been proposed in [20], depending on the choice of the physical meaning of control inputs vector η. The models, later referred to as *Full Models*, are described by the following equations.

Full Model 1. Direct control over rotations and one spinning movement

$$\dot{q} = \begin{bmatrix} Rs_{\theta_0} & Rc_{\theta_0}c_{\varphi_1} & R\left(s_{\theta_0}s_{\theta_1} - s_{\varphi_1}c_{\theta_0}c_{\theta_1}\right) & 0 & 0 \\ -Rc_{\theta_0} & Rs_{\theta_0}c_{\varphi_1} & -R\left(s_{\theta_0}s_{\varphi_1}c_{\theta_1} + s_{\theta_1}c_{\theta_0}\right) & 0 & 0 \\ -\frac{Rs_{\varphi_2}}{2l\tan\theta_2} & -\frac{Rc_{\varphi_1}}{2l} & \frac{R\left(-\frac{s_{\theta_1}s_{\varphi_2}}{\tan\theta_2} + s_{\varphi_1}c_{\theta_1}\right)}{2l} & \frac{Rs_{\varphi_2}}{2l\tan\theta_2} & \frac{Rc_{\varphi_2}}{2l} \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ \frac{1}{s_{\theta_2}} & 0 & \frac{s_{\theta_1}}{s_{\theta_2}} & -\frac{1}{s_{\theta_2}} & 0 \end{bmatrix} \eta. \quad (2.3)$$

Full Model 2. Direct control over both spinning movements and three rotations

$$\dot{q} = \begin{bmatrix} Rs_{\theta_0} & Rc_{\theta_0}c_{\varphi_1} & R\left(s_{\theta_0}s_{\theta_1} - s_{\varphi_1}c_{\theta_0}c_{\theta_1}\right) & 0 & 0 \\ -Rc_{\theta_0} & Rs_{\theta_0}c_{\varphi_1} & -R\left(s_{\theta_0}s_{\varphi_1}c_{\theta_1} + s_{\theta_1}c_{\theta_0}\right) & 0 & 0 \\ 0 & -\frac{Rc_{\varphi_1}}{2l} & \frac{Rs_{\varphi_1}c_{\theta_1}}{2l} & \frac{Rc_{\varphi_2}}{2l} & -\frac{Rs_{\varphi_2}c_{\theta_2}}{2l} \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & s_{\theta_1} & 0 & -s_{\theta_2} \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \eta. \quad (2.4)$$

It is important to note that in *Full Model 1* (2.3) the angles $(\varphi_1, \theta_1, \psi_1, \varphi_2, \theta_2)$ are controlled directly (i.e. $\dot{\varphi}_1 = \eta_1, \dot{\theta}_1 = \eta_2, \dots$), while in *Full Model 2* (2.4) we have direct control over $(\varphi_1, \theta_1, \psi_1, \theta_2, \varphi_2)$.

2.2.2 Simplified kinematic model

Due to the complicated mathematical description of the robot movement, a simplified model has been derived in [20]. The model uses a concept of a hypothetical steerable wheel with variable diameter. Such a wheel can represent the HOG wheel, with the exception that the horizontal movement of the HOG wheel mounting point caused by hemisphere rotation is ignored. This simplification effectively transforms the model to the case of a robot of class (1, 2).

Simplified Model (2.5) has been derived using a transformation of the configuration variables of *Full Model 2* (2.4), and uses a configuration vector

$$q = (x, y, \theta_0, \theta_{u1}, \varphi_{u1}, \theta_{u2}, \varphi_{u2}, r_{u1}, r_{u2})^T,$$

where r_{u1}, r_{u2} denote the current radii of the hypothetical wheels, θ_{u1}, θ_{u2} are the orientations of the wheels, and φ_{u1}, φ_{u2} are the angles of wheels' spinning motion. The variables in *Simplified Model* have been illustrated in Figure 2.5 and the control is described by the following equation.

Simplified Model Direct control over the velocities $(\dot{\theta}_{u1}, \dot{\varphi}_{u1}, \dot{\theta}_{u2}, \dot{r}_{u1}, \dot{r}_{u2})$

$$\dot{q} = \begin{bmatrix} 0 & r_{u1}\, c_{\theta_0+\theta_{u1}} & 0 & 0 & 0 \\ 0 & r_{u1}\, s_{\theta_0+\theta_{u1}} & 0 & 0 & 0 \\ 0 & \frac{r_{u1}\, s_{\theta_{u1}-\theta_{u2}}}{2l s_{\theta_{u2}}} & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & \frac{r_{u1}\, s_{\theta_{u1}}}{r_{u2}\, s_{\theta_{u2}}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \eta. \quad (2.5)$$

When applying control derived for *Simplified Model*, the results must be transformed

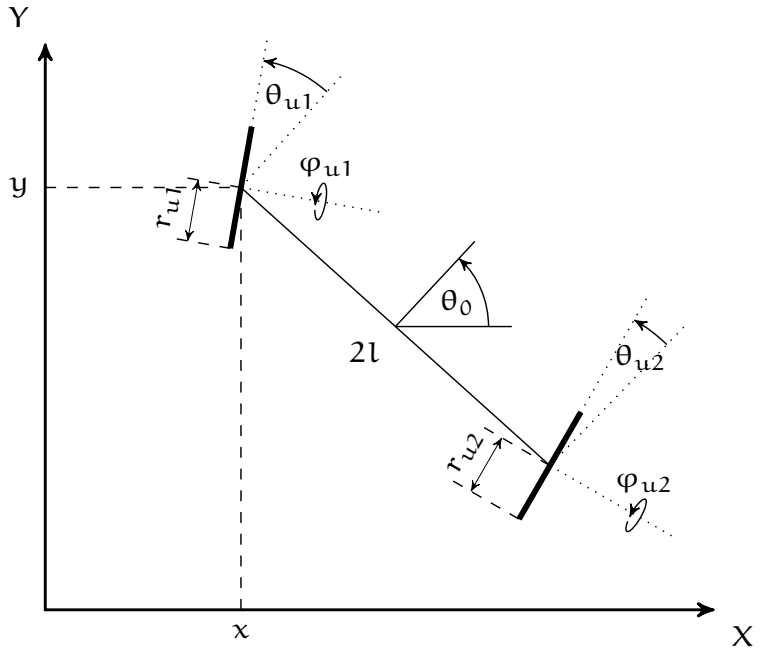


Figure 2.5: Robot configuration in the simplified model

back to the configuration space of *Full Model* (2.1). In [21] two approaches have been described: 'offline' and 'online' modes. While both approaches can be applied in simulations, the 'offline' mode is impractical in real-world control, because it additionally requires simulating the simplified model operation to measure its output. When using the 'online' mode, both measurement of state and application of control are done in *Full Model* space and are transformed from/to *Simplified Model* to apply the rest of the algorithm. The transformations between state variables of *Full Model* and *Simplified Model* are described by the equations

$$\begin{cases} \varphi_{ui} &= \psi_i \\ r_{ui} &= R\sqrt{c_{\varphi_i}^2\left(s_{\theta_i}^2 - 1\right) + 1}\,, \\ \theta_{ui} &= \operatorname{atan}\frac{s_{\theta_i}}{s_{\varphi_i} c_{\theta_i}} \end{cases}$$

and the inverse transformation

$$\begin{cases} \psi_i &= \varphi_{ui} \\ \varphi_i &= \pm\operatorname{acos}\left(\dfrac{\sqrt{r_{ui}{}^2 - R^2}\sqrt{\tan_{\theta_{ui}}^2 + 1}}{\sqrt{-R^2 - R^2\tan_{\theta_{ui}}^2 + r_{ui}{}^2\tan_{\theta_{ui}}^2}}\right)\,. \\ \theta_i &= \pm\operatorname{asin}\left(\dfrac{r_{ui}\tan_{\theta_{ui}}}{R\sqrt{\tan_{\theta_{ui}}^2 + 1}}\right) \end{cases}$$

2.2.3 JPTD kinematic model

JPTD algorithm is a heuristic-based simplification of *Full Model 2* (2.4), developed in [21]. In this model, the least significant parts of the kinematic control matrix $G(q)$ of *Full*

Model 2 have been replaced with zeros. The JPTD model has proved to yield good results in simulations, so it should be considered as potentially better suited for the task of robot control in this particular case. The model is described by the following equation.

JPTD Model Direct control over the velocities $(\dot{\varphi}_1, \dot{\theta}_1, \dot{\psi}_1, \dot{\theta}_2, \dot{\varphi}_2)$

$$\dot{q} = \begin{bmatrix} 0 & 0 & R(s_{\theta_0}s_{\theta_1} - s_{\varphi_1}c_{\theta_0}c_{\theta_1}) & 0 & 0 \\ 0 & 0 & -R(s_{\theta_0}s_{\varphi_1}c_{\theta_1} + s_{\theta_1}c_{\theta_0}) & 0 & 0 \\ 0 & 0 & \frac{Rs_{\varphi_1}c_{\theta_1}}{2l} & 0 & -\frac{Rs_{\varphi_2}c_{\theta_2}}{2l} \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & s_{\theta_1} & 0 & -s_{\theta_2} \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \eta. \tag{2.6}$$

2.3 Control algorithms

Several different control algorithms for Hogger[2] have been derived in [21]. The algorithms have been tested in simulations only. They will be presented below, along with summary of simulation results.

2.3.1 Input-output decoupling and linearisation

Because all the Hogger[2] models described in Section 2.2 have 5 control inputs and 9 state variables, to apply the state feedback linearisation algorithms, first we have to use the input-output decoupling algorithm to construct linearising output function with 5 outputs [45]. In the cases of *Full Models* (2.3), (2.4) and *JPTD model* (2.6) a trivial function

$$h(q) = (x, y, \theta_0, \psi_1, \psi_2)^T$$

can be used. For *Simplified Model* (2.5) a more sophisticated function

$$h(q) = \begin{pmatrix} x + d\cos(\theta_{u1} + \theta_0) \\ y + d\sin(\theta_{u1} + \theta_0) \\ \theta_{u2} \\ r_{u1} \\ r_{u2} \end{pmatrix} \tag{2.7}$$

has been chosen for the static linearisation algorithm, to avoid model singularities, and two different trivial ones, namely

$$h(q) = (x, y, \theta_{u2}, r_{u1}, r_{u2})^T, \tag{2.8}$$

and

$$h(q) = (x, y, \theta_{u2}, \varphi_{u1}, \varphi_{u2})^T, \tag{2.9}$$

for dynamic linearisation.

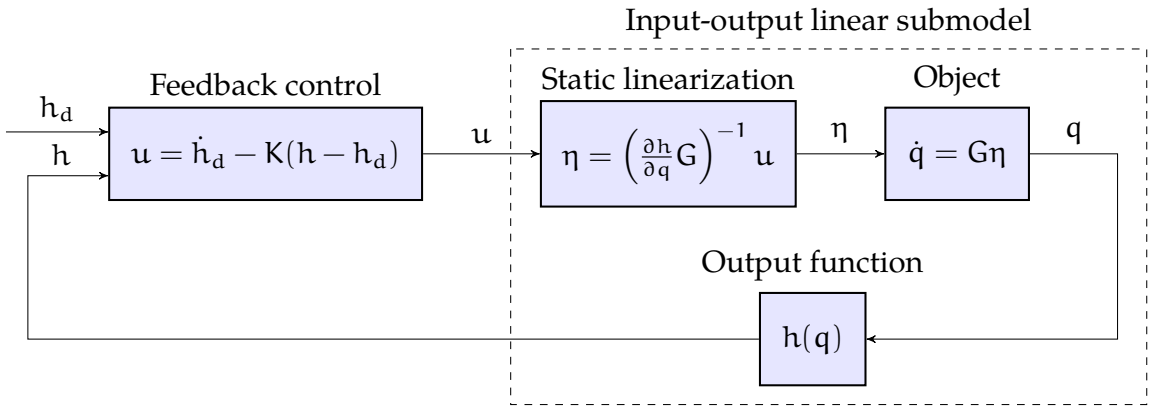


Figure 2.6: Static state feedback linearisation diagram with input-output decoupling. The submodel effectively behaves like an integrator (u is homogeneous to a velocity)

2.3.2 Static feedback linearisation

Given that the robot is modelled by the Equation (2.2), the static linearisation algorithm works by defining new control signal

$$u = \frac{d}{dt} h(q) = \frac{dh(q)}{dq} G(q)\eta, \tag{2.10}$$

resulting in a submodel that is a linear integrator when considering its input-output relation between u and h. For such a system, a standard feedback control algorithm in the form

$$u = \dot{h}_d - K(h - h_d) \tag{2.11}$$

is used, with a positive definite matrix K and desired output h_d. The actual control signal for the robot can be then calculated from (2.10), obtaining

$$\eta = \left(\frac{dh}{dq} G\right)^{-1} u,$$

thus the algorithm cannot work, when the matrix

$$D = \frac{dh}{dq} G$$

is singular. The concept of static linearisation has been depicted in Figure 2.6.

2.3.3 Dynamic feedback linearisation

In general, the dynamic state feedback linearisation algorithm is used when static linearisation cannot be applied ($\frac{dh}{dq} G$ is singular) [45]. The algorithm works by 'delaying' some or all of the control inputs, by introducing additional integrators in the system. For a system described by (2.2), a new control signal is defined,

$$\nu = \frac{d^2 h(q)}{dt^2} = K_{dd}(q)u + P(q, \eta), \tag{2.12}$$

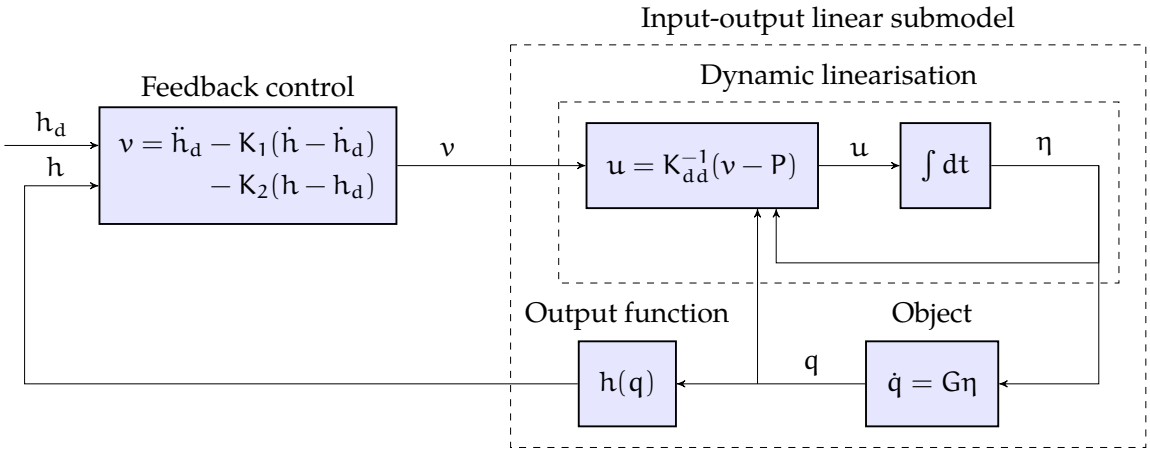


Figure 2.7: Dynamic state feedback linearisation with input-output decoupling. The submodel behaves like a system of two chained integrators (ν is homogeneous to an acceleration)

where u is an intermediate control signal, which is either equal to $\dot{\eta}$, or consists partially of elements of $\dot{\eta}$ and partially elements of η. The matrix K_{dd} and the vector P are obtained by factorization of the whole expression with relation to u. The resulting submodel has an input-output relation of a double-integrator (in particular, it is linear). The standard control algorithm for a system of two chained integrators is defined by

$$\nu = \ddot{h}_d - K_1(\dot{h} - \dot{h}_d) - K_2(h - h_d),$$

where K_1, K_2 are positive definite matrices and h_d is the desired output of the system. The control signal obtained from (2.12) can be written as

$$u = K_{dd}^{-1}(\nu - P),$$

where some or all elements of u have to be further integrated to obtain η. The matrix K_{dd} must have non-zero determinant for the algorithm to work. The concept of dynamic linearisation has been depicted in Figure 2.7.

2.3.4 Samson's algorithm

Another algorithm used to control Hogger[2] robot is the path tracking algorithm proposed in [33]. It is designed specifically for robots of class (1, 2), and as such can be used for *Simplified Model* (2.5). The algorithm models the problem of path tracking using Frenet frame and applies feedback linearisation for the model defined in such a way. The concept of Frenet frame has been depicted in Figure 2.8.

2.4 Simulation results

Results of the simulations described in [21] have been summarised in Table 2.1. The simultaneous model has been omitted, because it was mainly introduced to verify the correctness of calculations. The following factors, important from the perspective of control algorithm implementation, have been considered:

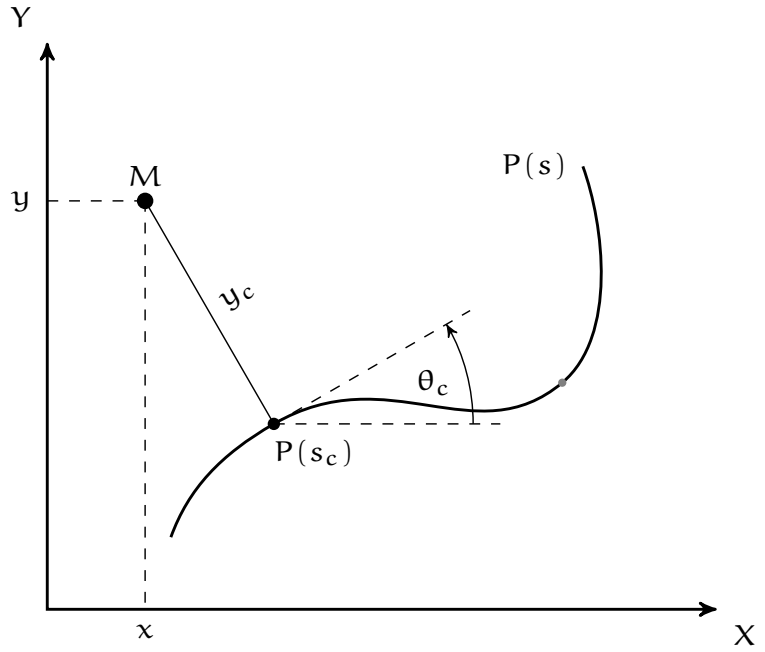


Figure 2.8: Path tracking in Frenet frame. The path is defined by the curve $P(s)$, M represents the robot position and the tracking errors are denoted by y_c and θ_c

Table 2.1: Short summary of simulation results. The numbers (1) and (2) written on the right of each model name correspond either to the number of *Full Model* used, or to the number of *Full Model* on which the given simplified model is based

				Control signal	
	Model	Algorithm	Tracking error	extrema	osc. freq.
1.	Full (1)	static	zero	v. high	v. high
2.	Full (2)	static	zero	v. high	v. high
3.	Full (2)	dynamic	zero	v. high	v. high
4.	JPTD (2)	dynamic	small	small	v. low
5.	Simplified (1)	static, offline	medium	medium	low
6.	Simplified (2)	static, offline	high	medium	low
7.	Simplified (1)	static, online	small	medium	low
8.	Simplified (2)	static, online	small	high	low
9.	Simplified (1)	dynamic, offline	small	medium	low
10.	Simplified (2)	dynamic, offline	high	medium	low
11.	Simplified (1)	Samson, offline	small	medium	v. low
12.	Simplified (2)	Samson, offline	high	high	v. low

- overall trajectory tracking accuracy,
- control signal extrema values,
- control signal oscillations frequency.

Among the simulation results presented in Table 2.1, all the algorithms using *Full Model* directly (1-3) are inapplicable in practice. Furthermore, algorithms 6, 10, 12 have unacceptable tracking errors. The algorithm 8 should also be considered unpromising due to high control signal values. The best results have been achieved using algorithm 4, while algorithms 5, 7, 9 and 11 should also be considered for application on the real robot.

Chapter 3

Code generation methodology

Translating mathematical description of a control algorithm into code is an important step of the process of creating the robot control system. The task of implementing the control algorithm can be trivial, e.g. in a proportional regulator case, however for complex problems writing code by hand becomes long and error-prone process. Moreover, embedded systems often have restricted computational resources, that impose restrictions on algorithms' complexity and the programming languages that can be used for their implementation.

A common approach to implementing complex control algorithms on embedded systems, is to develop the problem description symbolically in a computer algebra system (CAS) and later use that description, along with the tools available in that symbolic environment, to generate source code that will be incorporated into the application [8]. The actual code generation procedure is usually discussed in the documentation of the specific environment used.

In this chapter the details of code generation process as a whole will be explored. First, the real-time operation problems will be summarised. Then the embedded environments restrictions will be discussed, along with the available choice of programming languages that can be used for implementation. The process of code generation and optimisation will be described, both in general and on the example of SymPy environment. In addition, the structure of code and integration with the rest of the application will be considered. Finally, the issues related to numerical differentiation and integration in real-time applications will be addressed.

3.1 Real-time computing

Real-time computing is a term which describes the software that has to meet requirements of a real-time system [36]. A real-time system (also: reactive system) is a system that responds to stimuli from the environment at the speed dictated by that environment. This is contrary to transformational systems that perform only data processing, or interactive systems, for which the interaction speed is dictated by the system itself. It is important to note, that real-time operation is not concerned about the actual computation speed, but the system predictability and strong guarantees of the response time.

All digital control systems have natural real-time restrictions. This is due to the process of discretization, in which it is assumed that the control and sensing happen at regular time intervals. When the control system is not able to deliver control signals on time, its performance degrades significantly.

The process of digital control can be divided into four main parts: A/D conversion, computation, D/A conversion, and actuation [5]. When converting from continuous to discrete model, it is often assumed that data processing is instantaneous, but it is not the case in real-world scenarios. While the computation time highly depends on the implementation details of the algorithm used, the conversion and actuation times are usually constant and dictated by hardware. This imposes restriction on the time of the system response. This time has to be shorter than a single control period or the system will not be able to meet the assumed frequency of control events. Furthermore, minimising processing time allows to minimise approximation errors due to divergence between continuous and discrete models.

3.1.1 Real-Time Operating Systems

In its simplest definition, an operating system (OS) can be defined as a piece of software designed to provide a level of abstraction over the hardware (e.g. input and output devices or memory allocation), as well as to control the scheduling of multiple programs running in parallel [52]. A real-time operating system (RTOS) is a system designed for real-time applications. Contrary to regular OSs, RTOSs do not always provide good hardware abstraction. RTOSs are mainly targeted at small microprocessor devices and have to provide minimal overhead to be applicable on resource constrained systems. Because of this, the main task of an RTOS is to control system multitasking and resource sharing, adhering to the guarantees required for real-time operation [40].

Parallel operation on a single central processing unit (CPU) device can be achieved by creating multiple 'small programs', called threads, that are executed one at a time, but are frequently switched by the scheduler. The task synchronisation is commonly implemented using queues, semaphores and mutexes [40]. These constructs allow to avoid data-race problems that can emerge if multiple threads try to use the same resource at the same time. [40] Multitasking provided by an RTOS can be preemptive or cooperative. Cooperative multitasking does not allow for task preemption, i.e. a task will never stop running until it signalises to the system that it has performed required work and can be stopped, to allow other tasks to do their work. Most RTOSs implement a combination of preemptive and cooperative multitasking, as it is the most flexible approach.

3.2 Embedded systems programming

Electronic devices that perform complex algorithms are usually realised on microcontrollers. A microcontroller unit (MCU) is a microprocessor with peripherals that allow it to communicate with other devices, e.g. sensors, motors. MCUs have highly restricted hardware capabilities (memory, processor frequency, etc.) when compared to desktop computers. This imposes restrictions for tools and libraries that can be used on MCU-based devices.

3.2.1 High-level languages*

It is often convenient to write down the algorithm computations in a matrix form. Programming languages like MATLAB or Python, that are meant for scientific computing, provide us with concise syntax for such operations — one that reflects the way how we would write the computations down on paper. But implementing algorithms on embedded systems using these programming languages is often impossible. The main problem is the size of the runtime environment that must be installed on a device with relatively small memory, as well as the requirement of a strong CPU, because these languages usually perform significantly slower than compiled languages, such as C.

Another problem that often arises when programming embedded systems is dynamic memory management. It is connected to the real-time operation requirements. High-level languages usually use garbage collector to manage allocated memory. Garbage collector deallocates memory at unpredictable times and the user has no influence on when garbage collection is performed, which may freeze the program execution at an important moment. This can lead to the program missing real-time deadlines, so, to avoid this, manual memory management should be used.

3.2.2 Hardware access

MCUs are equipped with peripherals for external communication and signal processing. These are often difficult to configure and use. Moreover, microcontrollers from different vendors have different hardware and require a lot of processor-specific configuration. Developing low-level software for all the possible hardware platforms is impractical, so vendors provide libraries, so called hardware abstraction layers (HALs), that ease hardware configuration and allow to develop code using provided abstractions. These libraries are often available only in the C programming language, because over the years it has become a standard in the industry.

The omnipresence of C libraries for embedded systems and the resulting lack of support for other languages, makes it difficult to use other languages than C in embedded applications, where hardware access is required. It is often inconvenient to use HAL written in C when programming in another language, even though a lot of languages provide dedicated bindings for integration with C. The C++ language is a notable exception, as C code can be used directly in a C++ application.

3.3 Code generation

Many CASs have built-in tools for converting the expressions developed symbolically into code written in another programming language. The resulting code can have different complexity and extendability possibilities. In the process, several code optim-

*High level of abstraction is a relative concept and some may consider languages on the level of C and C++ to be high-level programming languages, relative to e.g. assembly language, because they free the user from thinking about hardware details, such as usage of processor registers. In this work however, the term 'high-level programming language' refers to languages that provide the user with features such as automatic memory management or dynamic type deduction. The term 'dynamic languages' would probably be more suitable, however the author decides to stick to the term 'high-level programming language' for the rest of the work.

isations can be performed, to reduce the complexity of computations that have to be performed during runtime.

Some popular symbolic environments for code generation will be described below. Based on the reasons described in Section 3.2, the only considered target language will be the C programming language.

3.3.1 Mathematica

Wolfram Mathematica is a powerful technical computing system, which mainly targets the area of symbolic computations [57]. Mathematica pricing depends on the usage area, starting from £105 (or £55/year) for non-professional, personal usage for students. It is widely used for computations and simulations on desktop computers, but also offers two options for generating code for embedded platforms.

The first option is the `CCodeGenerator` package, which provides tools for converting functions defined symbolically in Mathematica into C code. Mathematica automatically applies optimisations to the generated functions and generates complete C source and header files. The downside of the package is that it assumes linking the program with Wolfram runtime libraries. This makes it hard to cross-compile the generated code for embedded targets, although it could be possible to manually extracts parts of code that perform the necessary computation and include them only in the rest of the application.

The second method available is the `Microcontroller Kit`, which is a set of tools that provide high abstraction over hardware, allowing to easily develop complex control applications. The greatest advantage of the package is its completeness, i.e. simple symbolic description of the application can be converted into a complete C project, compiled and flashed onto the device with just a few lines of code. However, this is also its major downside, as the package supports mainly boards with ATmega32 MCUs from Arduino, Pololu and Adafruit. The lack of support for other hardware is very limiting, especially because most of the MCUs used in the industry today have ARM CPU cores.

3.3.2 MATLAB

MATLAB is an industrial standard numerical computing environment [54]. It offers the Simulink package, that allows to design control algorithms visually as a system of blocks. It also provides many additional tools, among which there are the `MATLAB Coder` software for generating code in other programming languages and the `Symbolic Math Toolbox` for symbolic computations and code generation. The licence pricing starts from 35€ for student licence (non-commercial usage).

The `MATLAB Coder` is a tool for generating C code based on numerical code written in MATLAB language. It allows to generate portable code with no runtime dependencies, which can be used on embedded platforms. The `Simulink Coder` can be used to generate complete C applications based on visual system model. The `Embedded Coder` is an additional extension, that allows for optimising the code specifically for embedded platforms and includes much wider support for target platforms than Mathematica's `Microcontroller Kit`. `MATLAB Coder` is a comprehensive package for many control applications, but it does not support symbolic expressions, so another package has to be used instead for such cases.

Although MATLAB is primarily a numerical computation environment, it supports symbolic computations though the `Symbolic Math Toolbox`. The developed expressions can be than optimised and converted to simple C code. The generation of C using `Symbolic Math Toolbox` is not as comprehensive as with the `MATLAB Coder` and may require additional processing to integrate with the rest of application.

3.3.3 SymPy

SymPy is a library for the Python programming language, providing tools for performing symbolic computations [32]. It is BSD licensed, so it is free for usage, even commercial. SymPy is cross-platform and easily extensible because it is written in pure Python. This however has negative impact on its performance, when compared to e.g. Mathematica. To increase computation speed, the SymEngine [44] is planned to be used as the backend for SymPy in the future, but now it covers only a subset of SymPy functionality.

SymPy provides tools for converting symbolic expressions into code written in other languages, including C. It also provides ways to simplify and optimise the expressions, as well as to customise the generation process, e.g. by changing the floating-point types used for variables to 32-bit, instead of the default 64-bit.

3.4 Code optimisation

Whenever code optimisation is considered, it is important to remember the famous quote by Donald Knuth [25]:

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: *premature optimization is the root of all evil*. Yet we should not pass up our opportunities in that critical 3%.

Indeed, premature optimisation can cause many problems when developing a computer program. However, automatic code generation is a different scenario, because most often we do not have to maintain the actual generated code, but rather the code that is responsible for the whole process of code generation. This results in much lower probability of introducing human errors during the optimisation process. For this reason, code generation is a natural phase when code optimisation should be applied.

3.4.1 Compiler optimisations

With computers becoming integral part of modern technology, programming becomes vital for achieving high income in many industries. This leads to constant development of the tools that are used to translate code into CPU instructions — the compilers.

One can argue that modern compilers are among the largest software projects that exist [31], with GNU compiler collection (GCC) having over 14.5 million lines of code in its codebase [26]. Modern compilers incorporate an increasing number of code

optimisation techniques, making some of the optimisations that a programmer could perform unneeded, or even undesirable — it is often better to write simpler, more maintainable code and let the compiler optimise it.

3.4.2 Basic types of optimisation

Code optimisation is a large area of research. There are many different types of optimisations [3, 23], taking a look at available GCC flags [14] reveals that they all cannot be described in this document. To provide reasonable view of what types of optimisation can be applied, some of the basic types will be discussed below.

Local and global optimisations

Optimisations can be local or global [3]. Local optimisations are performed only on basic blocks, i.e. parts of code with no control flow (no jump instructions). These are fairly simple and usually provide little savings but their application is also simple and does not take a lot of time.

Global optimisations, on the other hand, take into consideration larger blocks of code. They usually act on whole functions. Those optimisations provide greater potential benefits, but require more computations involved, slowing down the compilation process.

There is also another type of optimisations, called interprocedural optimisations, that work on the program as a whole. Not all compilers support this type of optimisations. Interprocedural optimisations are applied at the end of optimisation process, and allow to inspect complex dependencies between different parts of the program, giving opportunity to optimisations that would be impossible without that information, but come at the cost of even more computations required to apply them.

Compile-time evaluation

These types of optimisations, including constant folding and constant propagation, work by identifying parts of code that can be computed at the time of compilation, i.e. involve operations on constants [3]. An example of constant folding has been provided below.

Before constant folding:

```
float a = sinf(M_PI / 2) * x;
```

After constant folding:

```
float a = 1 * x;
```

Common Subexpression Elimination

Common subexpression elimination (CSE) is a technique that requires analysing the computation graph to find parts of expressions that are computed multiple times and modifies the code, by computing them once before and storing a result in a new temporary variable, which is then used in all the expressions [3]. The process can be repeated recursively — different compilers have different depths of search for common subexpressions.

CSE can provide significant benefits, especially if the subexpressions computation is time consuming, however sometimes, when the subexpressions can be computed fast, it may be beneficial not to apply CSE, if this would require storing the values in memory instead of using CPU registers.

The concept of CSE has been illustrated below.

Before CSE:

```
float p = sinf(a)*cosf(b) * x;
float q = -sinf(a)*cosf(b) * y;
```

After CSE:

```
float tmp = sinf(a)*cosf(b);
float p = tmp * x;
float q = -tmp * y;
```

Dead code elimination

This type of optimisations analyses the control flow, identifying parts of code that can never be reached and may be safely removed [3], for example:

Before dead code elimination:

```
int pow2(int x) {
    int y = x * 3;
    return x * x;
}
```

After dead code elimination:

```
int pow2(int x) {
    return x * x;
}
```

Loop unrolling

Loop unrolling is a technique that compensates for the time required for conditional branch instructions in the loop condition block, which is much longer when compared to simple arithmetical operations [3]. By rewriting loop body multiple times, the number of tests of loop condition decreases, which can increase program performance at the cost of increasing binary size. A loop with condition known at compile time may even be fully unrolled, but this may lead to significant increase of program size.

Below an example of partial loop unrolling has been provided. In this example, the number of loop condition evaluations is reduced by the factor of three.

Before loop unrolling:

```
int q = x;
for (int i = 0; i < 30; ++i) {
    q = q * x;
}
```

After loop unrolling:

```
int q = x;
for (int i = 0; i < 30; i += 3) {
    q = q * x;
    q = q * x;
    q = q * x;
}
```

Function inlining

Function calls impose significant overhead, because of the requirement to store registers' contents on the stack, before jumping to the function body [3]. This is especially true for functions performing simple operations. Function inlining works by copying the function body directly into the places in code where it is used. This also increases overall binary size. An example of function inlining has been provided below.

Before function inlining:

```
int pow3(int x) {
    return x * x * x;
}

int a = 2, b = 3;
int y = pow3(a) + b;
int z = a + pow3(y);
```

After function inlining:

```
int pow3(int x) {
    return x * x * x;
}

int a = 2, b = 3;
int y = a * a * a + b;
int z = a + y * y * y;
```

3.4.3 Data locality

Memory latency is the main bottleneck of computing these days [56]. This problem, although more important for programs that process large amounts of data, is so crucial from the performance point of view, that it has been described shortly below.

Modern processors, even those in embedded systems, can perform multiple arithmetic operations during the time of one memory access. This effectively means that every time some data from memory is required, the CPU has to stall for several processing cycles. To remedy this, computers include multi-level cache memory. Cache memory is much faster than regular memory, but has much smaller capacity, because it is much more expensive. It stores recently accessed data, so that the most frequently used data can be read faster. Whole cache lines are loaded from the main memory at once when a single access occurs, so to reduce the number of situations when the required data is not already in cache (so called 'cache misses'), CPU should read memory at memory addresses that are close to each other.

3.4.4 Floating-point arithmetic

Processors can differ in the type of floating point unit (FPU) available, or they can not have any FPU at all. In the later case, floating-point arithmetic can be implemented using software constructs, which is done by default when using C compiler. This however, results in much slower computations and big increase in the final binary size. Because of this fact, on such devices fixed-point arithmetic is often used. Porting floating-point calculations to fixed-point is a difficult topic and will not be described here.

Many modern microprocessors, especially 32-bit ones used for implementing complex mathematical algorithms, are equipped with FPU, thus it is not always necessary to avoid floating-point. However it has to be noted, that microprocessors used in embedded systems often have 32-bit FPUs, as compared to desktop computers with 64-bit

precision FPUs. In C, different floating-point precisions are represented by the 32-bit `float` type and the `double` type (so called 'double-precision').

An important type of optimisations consists of transforming the code written using 64-bit floating point arithmetic to 32-bit one [47]. For example, this involves the default C standard library functions, such as `sin` or `cos`, which perform arithmetic using double-precision floating point numbers. There exist standard library functions for single-precision operations, having `f` suffix (e.g. `sinf` instead of `sin`).

After applying such optimisations the resulting precision of mathematical operations decreases, so the algorithm results have to be verified to check if the computation errors stay in the required range. Even so, single-precision operations are generally sufficient for most algorithms.

3.4.5 Automatic compiler tuning

As the actual number of optimisations available in the compilers is much larger that the ones mentioned earlier, it is difficult to decide which optimisations should be enabled for the given hardware platform and algorithms used in the application. Compilers usually provide different optimisation levels, which combine a set of optimisations to provide sensible defaults, which then can be further modified with other compilation flags to selectively enable/disable optimisations.

As an example, GCC provides optimisation levels numbered from 0 to 3 (`-O0` to `-O3`), where 0 means no optimisations and 3 enables almost all of them. Furthermore, there are some special optimisation levels: optimisations that do not increase resulting binary size (`-Os`), optimisations that do not cause unexpected results when debugging the program (`-Og`) and optimisations that do not conform to mathematical operation standards, e.g. IEEE floating point standard (`-Ofast`) [14].

While the presets provided are more convenient, they may not be optimised for a particular hardware configuration. As a solution, many techniques of automatic compiler flags tuning have been proposed, including iterative methods with different types of sampling of the available compilation flags space, and even machine learning based approaches [6, 24]. These methods allow to find optimal configurations, but require additional benchmarking, which increases development time and efforts, and the results may not be significant enough to be worth the costs in all applications.

3.5 Code structure

A robotic system usually requires implementing multiple tasks, such as sensing, localisation, path planning, motor control, etc. Multiple parts of the system have to perform a sequence of operations with delays between them, e.g. reading an inertial measurement unit (IMU) might involve reading the accelerometer data, reading the gyro data, filtering the readings and waiting appropriate time for the next reading. Furthermore, different modules often require different service frequencies and often have to exchange data between each other.

The issues mentioned above are not of concern when implementing a system with a single task, however, the work involved into extending the system can be minimised by

implementing the task in a modular way. Modular programming is a concept of splitting the module interface from its implementation, which can be done in C by writing the application programming interface (API) in header files and moving the algorithms implementation to source files [18]. The modules are later connected depending on the system architecture, whether it is a bare-metal application or an OS/RTOS based one.

3.5.1 Control loop

Control loops are a key component of many embedded systems. A typical feedback control loop consists of three main blocks: measurement of inputs, control output calculation and its application [5]. As described in Section 3.1, this imposes a delay between measurement and actuation, which should be minimised to improve control quality. Moreover, if the delay becomes noticeable, the system should be treated as a control system with delays, which makes the control process much more complicated [17] In many algorithms it is possible to calculate the output faster, by delaying some calculations that are not required for computing the current output value, but are used for the next value calculation. This involves some computations that update the internal state or operations that can be precomputed before the next algorithm iteration, e.g. calculating the next point of the desired trajectory. In such cases, the control loop can be modified by splitting it into four parts instead of three, as shown in Figure 3.1.

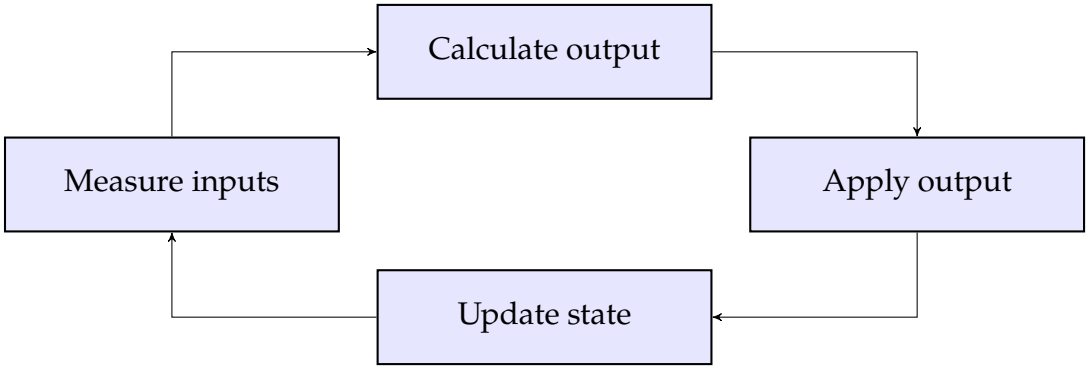
In Figure 3.1b the delay T_i between measurement and actuation has been illustrated. In practical applications, measurement and actuation may involve reading from or writing to multiple devices. This complicates the situation even more, because measurements are taken at different time moments, as well as the outputs are not updated at the same instant. Consequently, it is important to design the control loop in such a manner, that the time difference of different measurements/actuations is as low as possible.

3.5.2 Generated code decomposition

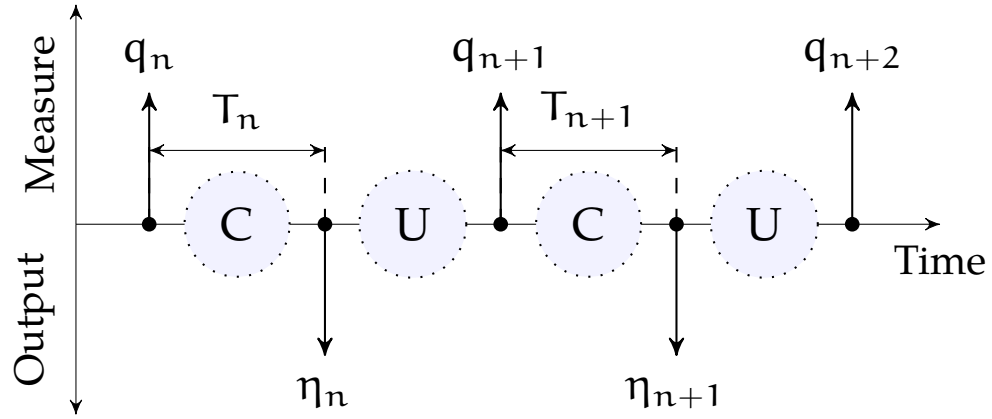
It is a good programming practice to separate the code of a program into many small functions [28]. Such an approach allows for easier code maintainability and better code reuse. Whenever implementing a complex algorithm, the programmer should decompose the problem into smaller subproblems, that can be tested independently.

Although this practice is so important for code written manually, it may be not so desirable when the code is generated automatically. As the generation process is automatic, the code does not have to be manually maintained and any changes to the code are made on the generation algorithm instead. This questions the relevance of algorithm decomposition, because producing bigger functions may lead to more opportunities for code optimisation, e.g. allowing for improved application of CSE by providing a bigger context. Moreover, generating code as a single, big procedure may be easier, as well as it can decrease the amount of work that a programmer has to do in order to incorporate the algorithm into the rest of the application.

On the other hand, decomposed implementation still leads to some benefits. First, people have a tendency to make mistakes, so even if the generation process is perfectly repeatable, there might be subtle errors in the description of the generation process



(a) Loop steps



(b) Timing diagram (C — calculate, U — update, q_i — i-th measurement, η_i — i-th output). The delay T_i may jitter, due to other tasks scheduled in the system

Figure 3.1: Generalised control loop diagram

or in the implemented algorithm itself. Splitting the code allows for easier debugging process and can speed up the process of code development. In addition, code split into separate functions allows for easier timing of separate parts of the algorithm, facilitating the process of code profiling, by allowing to easier locate the performance bottlenecks. Apart from that, the code that has been generated, although it may not be modified, will be easier to read and to reason about, if it is split into smaller functions.

Considering the arguments mentioned above, the final decision on selected approach has to be made depending on the specific problem, and none of the approaches seems to be unconditionally better.

3.6 Differentiation and integration

Another problem that often emerges during implementation of control algorithm is how to properly perform differentiation and integration operations numerically. Background information concerning this issue will be required in following chapters, so the problem has been shortly described below.

From definition, differentiation is written as the limit [34]

$$\frac{d}{dx}f(x_i) = \lim_{h\to 0}\frac{f(x_i+h)-f(x_i)}{h},$$

while integration is defined by the Riemann integral as [34]

$$\int_{x_0}^{x_f} f(x)dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i)h,$$
$$x_i = x_0 + ih,$$
$$h = \frac{x_f - x_0}{n}.$$

Of course in practice it is impossible to achieve h value which is infinitely close to zero. Even more, for discretely sampled data points, h is equal to the sampling period. Because of this, different estimation methods for calculating derivatives and integrals exist [34].

3.6.1 Differentiation

The simplest algorithm for differentiation is the finite difference approximation [34]. There are three variants of this algorithm: forward, backward and central differences. The only one that is applicable for an online algorithm (not having knowledge of future measurements) is backward difference, which leads to a simple derivative approximation defined as

$$\nabla_h f(x_i) = \frac{f(x_i) - f(x_i - h)}{h},$$

for some small h.

The differentiation operation is very sensitive to input noise, because high frequencies are amplified. To illustrate this, take for example a derivative of a sine wave with an angular frequency ω:

$$\frac{d}{dx}\sin(\omega x) = \omega\cos(\omega x).$$

As can be seen, the resulting differentiated signal is a phase-shifted input signal, amplified by the value of frequency ω. This property can have negative impact on the control algorithm performance in presence of high frequency noise. In order to reduce the noise influence, it is common to use filters to smooth the differentiated signal. These could be standard IIR or FIR filters [43], or a more sophisticated statistical-based estimation approach, e.g. Kalman filter [42]. Unfortunately, applying filters introduces a phase lag relative to the real value of the estimated derivative, so the filtering strength should be balanced.

3.6.2 Integration

Simple method for numerical integration is quadrature integration [34]. In this method the integrated function is interpolated using polynomial functions, usually of low degree. In the simplest case a constant function (polynomial of degree zero) can be used. This method is called rectangle rule, and is defined by

$$F_n = F_{n-1} + f_n h,$$

where F_i denotes i-th value of the calculated integral, f_n is the current value of the integrated function, and h is the sampling period.

A simple, yet effective improvement to this method is to use a polynomial of degree one. The method (called trapezoidal rule) can be written as

$$F_n = F_{n-1} + \frac{f_n + f_{n-1}}{2} h.$$

Similarly, polynomials of higher degrees can be used, e.g. of second degree (Simpson's rule), but in practice the accuracy improvements may be unnoticeable due to insufficient measurements precision.

Chapter 4

Control algorithms implementation

After designing the mathematical background for a control algorithm, it is necessary to implement it in a programming language on a target device. This step is often considered as a trivial task and overlooked, which may introduce errors in the implementation or lead to design decisions that can make future extensions of the implementation difficult.

In the previous chapters we have introduced the background essential for implementing control algorithms for Hogger^2 robot. Chapter 2 described Hogger^2 along with the model and the control algorithms derived for this robot in [21]. In Chapter 3 the background of code generation process have been explained, providing choice of tools that can be used for generating and optimising the control algorithms. In this chapter the specific problem of implementing control algorithms for Hogger^2 will be discussed. First, the implementation of control algorithms for Hogger^2 will be presented. The SymPy CAS, which has been described in Subsection 3.3.3, will be used for the code generation process. The algorithms will be implemented in C programming language. The choice of the language has already been explained in Subsection 3.2.1. Finally, the results of performance benchmarks of the developed implementation will be shown and discussed.

4.1 Symbolic modelling

The first step in the Hogger^2 control algorithms implementation is to convert the mathematical description of the robot into scripts in a CAS. For this purpose, the SymPy system introduced in Subsection 3.3.3 has been used.

The kinematic models described in Section 2.2 have been implemented symbolically in SymPy. To verify the correctness of the implementation, *Full Model 1* and *Full Model 2* have been tested if they satisfy the nonholonomic constrains of Hogger^2 robot described in [20]. Given the models, the static and dynamic linearisation algorithms have been implemented using the methods shown in Section 2.3. Finally, all the calculated expressions have been simplified using the methods provided by SymPy library.

4.1.1 Function declarations

Considering arguments presented in Subsection 3.5.2, the decision of decomposing the control algorithms into separate functions has been taken. The main reason for

such a decision was to be able to easily benchmark the performance of different parts of the algorithm.

The decomposition has been based on the diagrams shown in Figures 2.6, 2.7. Each block in the diagrams has only one (vector) output. Since in C language we have to represent vectors and matrices as arrays and it is impossible to return an array from a function without using dynamic memory allocation, the following convention has been used for the functions that have to return vector outputs: the first argument of each function is used as the output argument and all the following arguments are used as inputs and are not modified. The functions do not return any value, so their return type is `void`*. Additionally the functions for calculating the determinants of the matrices that are being inverted during the algorithms have been generated. Providing means of checking the determinants is important, because control algorithm will fail at the points of singularities and it may be necessary to switch to another algorithm or to use methods dedicated to avoiding singularities.

SymPy code generation does not allow to generate code of whole C functions, it is only possible to generate the expressions that can be used inside the function body, however it is not an issue. In fact, it is beneficial, because it allows the user to fully control the interface used in the application and make the order of function arguments fixed. For each model the following functions have been generated:

Static linearisation (Figure 2.6)

```
void feedback_u(float u[5], const float K[25], const float h[5],
    const float hd[5], const float d_hd[5]);

void static_lin_eta(float eta[5], const float u[5], const float q[9]);

float static_lin_det_D(const float q[9]);
```

Dynamic linearisation (Figure 2.7)

```
void feedback_v(float v[5], const float K1[25], const float K2[25],
    const float h[5], const float d_h[5], const float hd[5],
    const float d_hd[5], const float d2_hd[5]);

void dynamic_lin_u(float u[5], const float v[5], const float eta[5],
    const float q[9]);

float dynamic_lin_det_Kdd(const float q[9], const float eta[5]);
```

*A function in a programming language is not understood the same as in mathematics, because it usually can have side effects or can modify its input arguments. This however is not the case in the situation discussed here, so all the generated functions are what is referred to as 'pure functions' in programming theory. And because the functions created in the generation process do not have any constrains on their domain, they can never fail, so their return type can be left `void`.

The chosen naming convention was to specify the output of the block represented by a function as the postfix on the function name. The naming of the variables corresponds to the one used in mathematical description. Greek letters have been replaced with their English names and a prefix have been used to represent derivatives (`d_` for first derivative, `d2_` for the second derivative, etc.).

Additionally, two functions for conversion between the configuration variables of *Full Models* and *Simplified Model* have been defined as:

```
void full_to_simplified(float q_simp[9], const float q_full[9]);
void simplified_to_full(float q_full[9], const float q_simp[9]);
```

4.1.2 Function implementations

The documentation of code generation in SymPy is not extensive, but helpful materials can be found in [10]. The main function for generating C code in SymPy is the `ccode` function. Its basic usage is relatively simple, however the whole generation process in SymPy is not as straightforward as it could be. There are some minor problems that have to be addressed.

First, the C language does not have a native construct for manipulations on matrices and vectors, thus all the calculations have to be performed element by element. Fortunately, SymPy can generate code in that manner if an output symbol of type `MatrixSymbol` is passed to the `ccode` function through the `assign_to` argument. The `MatrixSymbol` is a placeholder for a matrix and proper dimensions have to be specified.

Second problem that emerged is that the robot configuration variables have been used in all the equations using their direct names, i.e. x, y, θ, etc. Consequently, the function would require passing all the nine state variables as separate arguments (and doing the same for other input vectors), which would not be convenient, nor efficient. To remedy this, a custom code printer class had to be implemented. The `ccode` function is in fact a wrapper around `CCodePrinter` class. SymPy allows to derive from this class to easily extend the printing process. In this way the printer has been extended by providing a set of mappings that would cause replacing variable names with indexing operations on the provided vector, e.g. transforming the expression `x` into `q[0]`. The code printer has been configured in single precision floating mode, for the reasons described in Subsection 3.4.4.

4.1.3 Common subexpression elimination

The CSE optimisation has been described in Subsection 3.4.2 while talking about compiler optimisation techniques. As mentioned, modern compliers can perform this type of optimisation, e.g. GCC does this by default at `-O2` optimisation level. That being said, CSE performed by the compiler does not always achieve complete elimination of common subexpressions. This will be later seen in Section 4.2 when the results of benchmarks will be presented.

To increase the performance of the generated code, CSE can be performed using SymPy library. The SymPy function `cse` performs CSE on the expression tree, returning a list of substitutions in their required declaration order, along with the resulting

simplification of the original expression. After performing the optimisation, the code of the subexpression declarations can be generated before the code of the final result calculation, by creating appropriate variables local to the generated function. Code generated in such a way is still subject to compiler optimisations, which in turn may lead to the compiler inlining some of the previously generated local variables into the final expression, effectively reverting the CSE process. However, if this is the case, than most probably such operation will provide performance benefits for the given CPU. As a result, the CSE performed symbolically in SymPy can aid the complier in the process of optimisation, but if needed, the compiler can easily revert the CSE in appropriate places.

4.1.4 C files structure

The generated functions have been organised into standard C header-source file pairs, e.g. `dynamic_lin.h/dynamic_lin.c` for the dynamic linearisation algorithm. The header exposes the interface to be used in the application through the function declarations and includes all the necessary libraries, namely the standard C maths library (through the `<math.h>` header). An additional header file `robot_parameters.h` has been generated and included in all the files. It contains the value of the radius `R` of robot hemispheres, the distance `l` between their centres and, for static linearisation of *Simplified Model*, the distance `d` used in the linearising output function `h(q)`.

4.1.5 Differentiation and integration

As can be noticed, during the code generation process no functions for integration or differentiation have been generated. There are two reasons for this decision.

First of all, in a general case we cannot know what data concerning the robot will be available for direct measurement. While most often the position is being measured directly, it may also be the case that, for some of the state variables, velocity will be measured directly instead. Considering this factor, only general routines for differentiation and integration should be provided, which can be then used to finalise the control algorithm implementation by executing these routines on the measured data appropriately.

Another reason for separating the implementation of differentiation and integration is that multiple problems emerge when performing these operations numerically. There are different methods that allow for increasing precision of differentiation and integration. Some of the methods have been introduced in Section 3.6. More information on this subject can be found in [34].

4.2 Benchmarks

Compared to algorithms designed for typical mobile robots, like these of class $(2, 0)$, the control algorithms developed for Hogger[2] are complex. Because of this fact, the time required for computations is of concern. To estimate the required hardware capabilities, appropriate benchmarks had to be performed. The benchmarks have been divided into two stages. In the first part, the relative performance on three different platforms have

Table 4.1: Naming convention used in benchmark results plots. Function names correspond to the names defined in Subsection 4.1.1

Function name	In-plot name
`feedback_u`	Static u
`static_lin_eta`	Static η
`static_lin_det_D`	Static $\det(D)$
`feedback_v`	Dynamic v
`dynamic_lin_u`	Dynamic u
`dynamic_lin_det_Kdd`	Dynamic $\det(K_{dd})$
`full_to_simplified`	Full $\rightarrow$ Simplified
`simplified_to_full`	Simplified $\rightarrow$ Full

been compared. Later, the time complexity of different models and control algorithms have been tested in details.

The tests have been performed for the algorithms of static and dynamic feedback linearisation described in Section 2.3, using the models listed in Section 2.2. In each test case the time taken by each function for an input consisting of random data has been measured. Special compiler directives have been used to disable optimisations that could be performed due to unused function results. Because the time taken by the benchmarked functions may be small (relative to available timing methods resolution), on some hardware platforms the time was measured for several function calls and then divided by the number of calls. Furthermore, the whole procedure of time measurement was repeated multiple times to compute the mean value and estimate the standard deviation using Welford's online algorithm [48]. The algorithm has been chosen due to limited memory on some of the devices used. The overhead of the timing procedures have been measured using an empty function call and subtracted from the final results. All the results presented later have been compiled using the `-O2` GCC optimisation level. The results naming convention has been described in Table 4.1.

4.2.1 Platforms performance

In this stage, the performance of the algorithms for *Full Model 1* and *Full Model 2* has been compared on different hardware platforms. The benchmarks have been run on three devices: personal computer (PC), Raspberry Pi microcomputer and STM32 Cortex-M microcontroller. Details of the platforms along with C compiler versions have been described in Table 4.2.

The results of the benchmarks have been presented in Figures 4.1, 4.2 and 4.3. As expected, the most time consuming functions are the functions `static_lin_eta` and `dynamic_lin_u`, responsible for the process of linearisation of control signals.

The presented results show that the performance difference between the implementations with and without applying CSE is significant. The relative difference in calculation time is the largest for dynamic linearisation function. This function is the most

Table 4.2: Hardware platforms used for benchmarks

PC	
Operating system:	Linux, kernel version: 4.19
C compiler:	gcc (GCC) 9.1.0
Processor:	Dual Core Intel Core i5-5200U, 2.2 GHz

Raspberry Pi	
Operating system:	Linux, kernel version: 4.14
C compiler:	gcc (Raspbian 4.9.2-10+deb8u2) 4.9.2
Processor:	Broadcom BCM2835, ARM1176JZF-S, 1 GHz

STM32	
Operating system:	none
C cross-compiler:	arm-none-eabi-gcc (Arch Repository) 9.1.0
Processor:	STM32F407, ARM Cortex-M4, FPU, 168 MHz

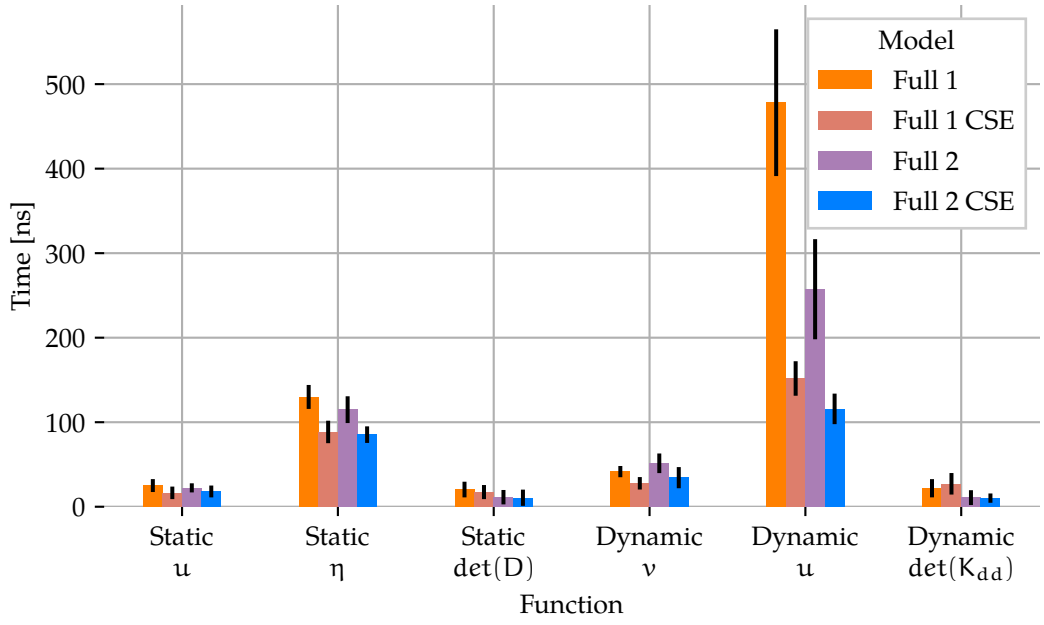


Figure 4.1: Benchmarks results on PC. Computed for 100 000 function calls measuring time after each 100 calls. Black bars represent standard deviation of measurements. Function naming convention has been described in Table 4.1

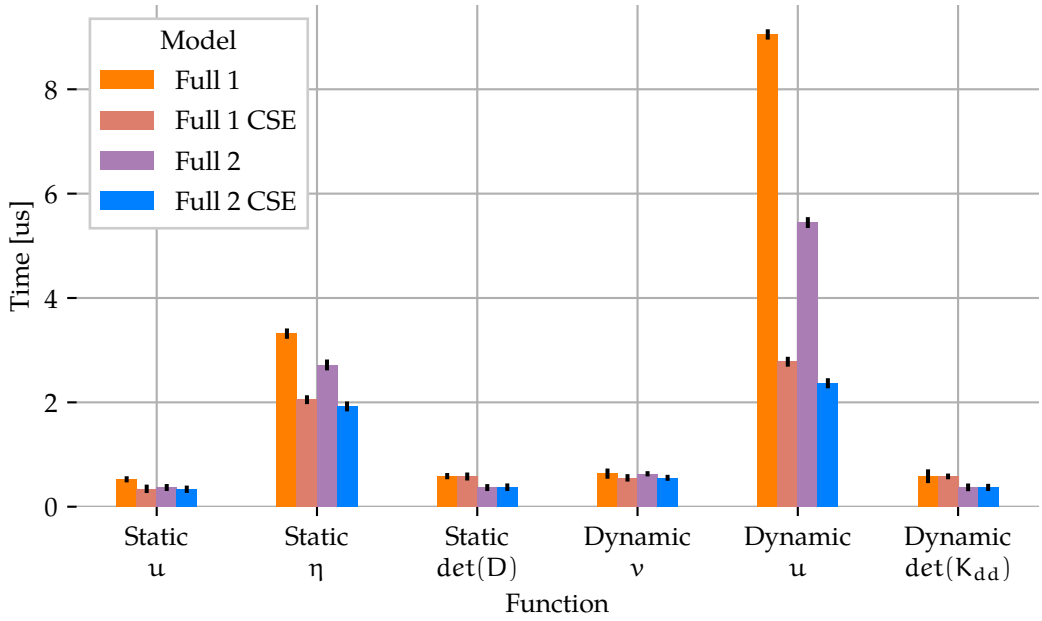


Figure 4.2: Benchmarks results on Raspberry Pi. Computed for 100 000 function calls measuring time after each 100 calls. Black bars represent standard deviation of measurements. Function naming convention has been described in Table 4.1

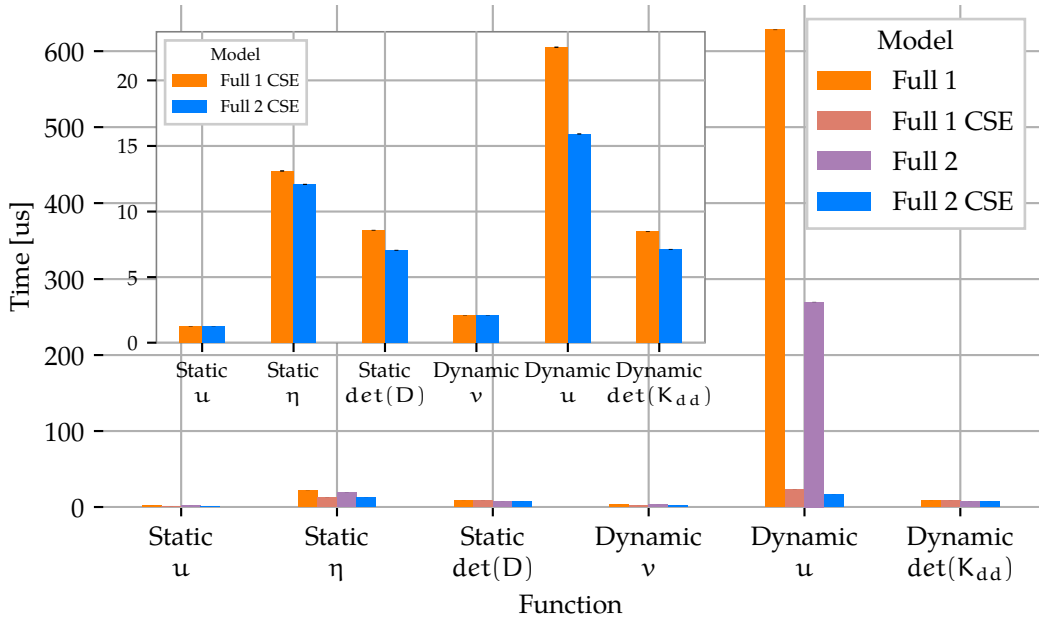


Figure 4.3: Benchmarks results on STM32. Computed for 1000 function calls measuring time after each call. Black bars represent standard deviation of measurements. Function naming convention has been described in Table 4.1. For clarity, the two benchmarks with CSE optimisation applied have been presented separately in the subplot

complex one, and so the optimisation algorithm finds many opportunities to extract common subexpressions. The performance gain is the largest on the STM32 device (up to time decrease by 96.4 % for *Full Model 1*). This may be caused by higher relative time cost of evaluating trigonometric functions on this device, however the exact reasons may be more complicated and have not been investigated.

The standard deviation values seen in the plots and their relevance to real-time behaviour have to be addressed. These values are so high for PC and Raspberry Pi due to the method of time measurement used on these devices. The tested functions computation time is very short relatively to timer resolution on these machines, even though the time has been measured after more than one function call. Besides this, the time cannot be measured precisely because the operating systems are not real-time, i.e. at any moment the system may interrupt the process introducing unpredictable delays. In addition, an OS can often change the CPU frequency, which may be another source of the measurements inconsistency. Despite the fact that the measurements are not precise, they show the order of time required for computations. On the other hand, the time measured on the microcontroller is precise up to single processor cycles. This is because there is no OS, which means that nothing can interrupt the measurement. Also, the ARM Cortex-M core includes a built-in CPU cycles counter that has been used to measure time with single CPU cycle resolution.

What is the most important, the results of benchmarks proved that it is possible to achieve computation times below 1 millisecond on the STM32F4 microcontroller, even without applying CSE. Consequently, there is no need to seek for more powerful CPUs if the control algorithm is the only computationally expensive task that will be performed.

4.2.2 Models performance

This part of measurements have been performed on the STM32 MCU only. The overall time complexity of the static and dynamic linearisation algorithms have been tested for all the models presented in Section 2.2. The results have been shown in Figures 4.4 and 4.5. A few interesting details can be read from the presented plots.

First, the actual performance of *Simplified Model* is much worse than the performance of other models due to very high cost of computing the transformations of configuration variables, even though the linearisation time is relatively low. Of course this does not mean that the simplification may not give better control results then other models, this only shows that the computational cost of using *Simplified Model* will be higher. Nevertheless, it may be possible to reduce the total computation time by incorporating the conversions directly in the linearisation function, which may increase the possibilities of applying CSE.

Second, it can be seen that the time taken by the feedback calculation function for *Simplified Model*, shown in Figure 4.4, is unusually high. This is caused by the non-trivial linearising function $\mathtt{h}(\mathtt{q})$ (Equation (2.7)) that had to be chosen to avoid singular linearisation matrix $\mathtt{D}$.

Finally, an interesting result can be seen in Figure 4.5 for the *JPTD Model*. As described in Subsection 2.2.3, the model is a simplification of *Full Model 2* by setting appropriate expressions in the matrix $\mathtt{G}(\mathtt{q})$ to zero, hence the computation time in this case, without applying CSE, is lower than for *Full Model 2*. However, when the CSE

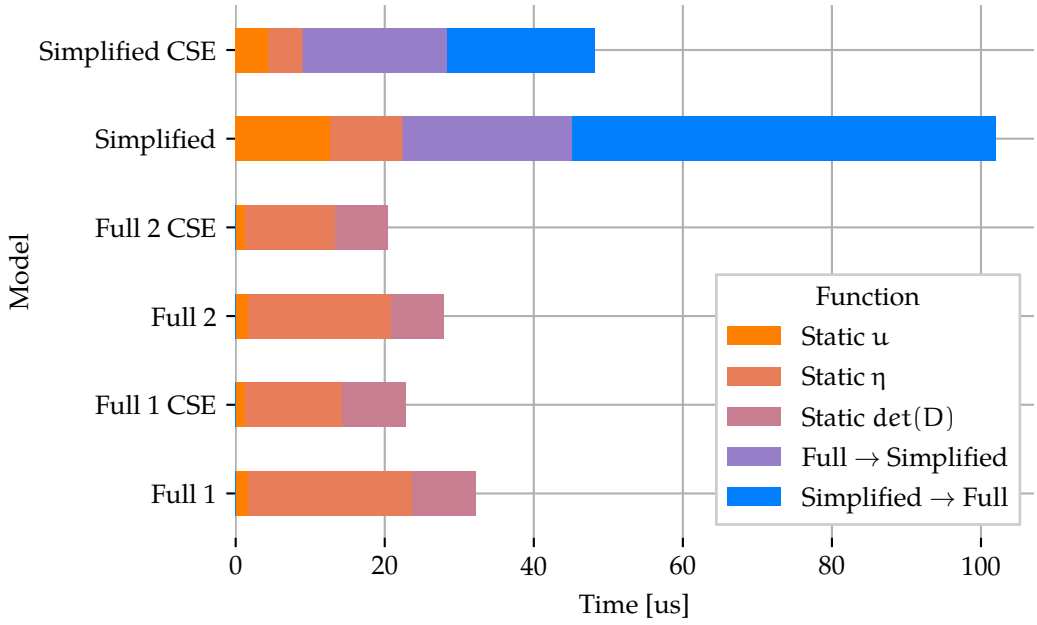


Figure 4.4: Benchmarks results on STM32 for the static linearisation algorithm. The plot shows cumulative times and per-function times for different models. The results for *Simplified model* include additional state conversion times

is applied, the performance gain is much bigger for the original *Full Model 2*, and as a result, the *JPTD Model* calculation end up being actually slower. This phenomenon must be attributed to the fact that, because of the modification of the matrix $G(q)$, some trigonometric expressions cannot be simplified, which results in some `tanf` evaluations that do not exist in *Full Model 2*[†]. In fact, after applying CSE there are 4 `tanf`, 5 `sinf` and 5 `cosf` evaluations in *JPTD Model*, and only 2 `tanf`, 5 `sinf` and 5 `cosf` evaluations in *Full Model 2*. For completeness, without applying CSE the counts are: 116 `sinf`, 164 `cosf`, 2 `tanf` for *Full Model 2* and 70 `sinf`, 64 `cosf`, 20 `tanf` for *JPTD Model*.

4.3 Algorithm implementation

When the functions that implement different blocks of the control algorithms have been generated, they have to be connected in a computer program that realises the task of robot control. The controller program has to perform the same set of operations in regular intervals, as has been previously shown in Figure 3.1. The control loop can be split into a repeated sequence of actions, where the most natural starting operation seems to be measurement, and as such a sequence can be implemented as a function/procedure in a programming language.

An important part of implementing algorithms in a computer program is to identify the persistent state, i.e. the state that has to be preserved between subsequent iterations. This is important, because in each iteration of a stateful algorithm data from previous iterations has to be used and an initial state has to be provided before first iteration.

[†]Note however, that symbolic simplification has been performed in both test cases, only the CSE algorithm has not been applied in one of them.

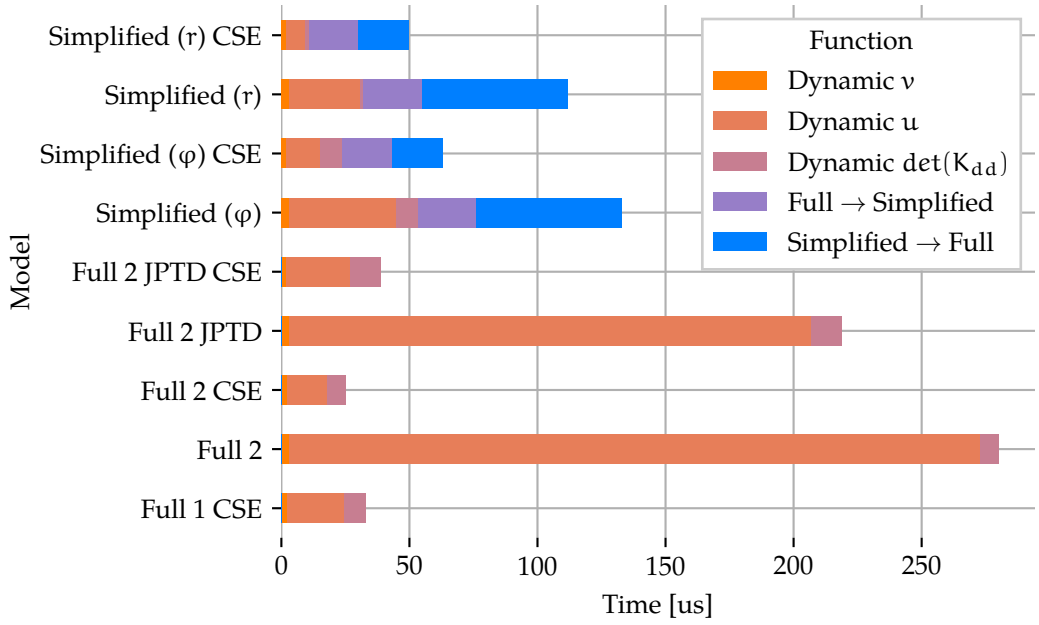


Figure 4.5: Benchmarks results on STM32 for the dynamic linearisation algorithm. The plot shows cumulative times and per-function times for different models. *Full Model 1* without CSE has been omitted, as it takes 640 µs. The results for *Simplified model* include additional state conversion times. Two *Simplified Models* have been tested, depending on the choice of output function $h(q)$ — either from Equation (2.8) with direct radii control 'Simplified (r)' or from Equation (2.9) with control over φ angles 'Simplified (φ)'

The state storage in a computer program can be implemented using global variables, variables local to a module, a structure or an object, depending on the programming style used.

The implementations of both static and dynamic linearisation are analogous, however the case of dynamic linearisation is more complex, so it will be used here for demonstration purpose. An example implementation of dynamic linearisation has been show using pseudocode in Algorithm 1. The pseudocode consists of three main parts:

parameters describe the values that do not change in every iteration (and, most likely, are constant throughout the whole operation of the program),

persistent state represents the variables required and modified in each iteration of the algorithm,

function describes the order of operations performed in each iteration; divided into:

- **local variables** describes the variables required for the scope of one iteration,
- **procedures** further decomposition of the performed operations corresponding to the steps shown in Figure 3.1a.

The types of variables have been provided as arrays of floating point values, e.g. '$\eta_{[n]}$ :: float[5]' denoting an array of 5 floating point values. Each variable is used

Algorithm 1: Straightforward implementation of dynamic linearisation

parameters
K_1 :: float[25]
K_2 :: float[25]

persistent state
$h_{[n]}$ :: float[5]
$h_{[n-1]}$ :: float[5] *» for computing the 1st derivative*
$h_{d[n]}$ :: float[5]
$h_{d[n-1]}$ :: float[5] *» for computing the 1st derivative*
$\dot{h}_{d[n-1]}$ :: float[5] *» for computing the 2nd derivative*
$\eta_{[n]}$ :: float[5] *» for integration of* u

function ControlStep(Δt)

local variables
$q_{[n]}$ :: float[9]
$\dot{h}_{[n]}$ :: float[5]
$\dot{h}_{d[n]}$ :: float[5]
$\ddot{h}_{d[n]}$ :: float[5]
$\nu_{[n]}$ :: float[5]
$u_{[n]}$ :: float[5]
$\dot{\eta}_{[n]}$ :: float[M], $M \in \{1, \ldots, 5\}$ *» depending on the model used*

procedure MeasureInputs
$q_{[n]} \leftarrow$ measure_q()
» linearising output
$h_{[n-1]} \leftarrow h_{[n]}$ *» shift sample numbers*
$h_{[n]} \leftarrow$ calculate_linearising_output($q_{[n]}$)
» trajectory generation
$h_{d[n-1]} \leftarrow h_{d[n]}$ *» shift sample numbers*
$h_{d[n]} \leftarrow$ calculate_next_trajectory_point(Δt)

procedure CalculateOutput
» derivatives calculation
$\dot{h}_{[n]} \leftarrow$ differentiate($h_{[n]}$, $h_{[n-1]}$, Δt)
$\dot{h}_{d[n]} \leftarrow$ differentiate($h_{d[n]}$, $h_{d[n-1]}$, Δt)
$\ddot{h}_{d[n]} \leftarrow$ differentiate($\dot{h}_{d[n]}$, $\dot{h}_{d[n-1]}$, Δt)
$\dot{h}_{d[n-1]} \leftarrow \dot{h}_{d[n]}$
» control algorithm
$\nu_{[n]} \leftarrow$ feedback_v(K_1, K_2, $h_{[n]}$, $\dot{h}_{[n]}$, $h_{d[n]}$, $\dot{h}_{d[n]}$, $\ddot{h}_{d[n]}$)
$u_{[n]} \leftarrow$ dynamic_lin_u($\nu_{[n]}$, $\eta_{[n]}$, $q_{[n]}$)
» integration of some or all of the elements of u
$\dot{\eta}_{[n]} \leftarrow u_{[n]}$ *» take the elements of* u *that have to be integrated*
$\eta_{[n]} \leftarrow u_{[n]}$ *» take the elements of* u *that do not have to be integrated*
$\eta_{[n]} \leftarrow$ integrate($\eta_{[n]}$, $\dot{\eta}_{[n]}$, Δt) *» integrate derivatives of* η

procedure ApplyOutput
$\eta_{[n]} \leftarrow$ prepare_output($\eta_{[n]}$) *» output saturation, optional filtering*
apply_control($\eta_{[n]}$)

to store value of some vector from an iteration relative to the current iteration number. This has been shown using a suffix, e.g. $h_{[n-1]}$ corresponds to the value of linearising output at previous iteration (n is the current iteration). Assignments has been written symbolically using the $\leftarrow$ operator, while the syntax 'function(arguments)' denotes a call to a function that uses the values of arguments as its inputs and possibly returns an output.

Algorithm 1 corresponds to the diagram shown in Figure 2.7. The required persistent state has been identified based on the analysis of implementation of ControlStep — simply, the values that are used before assignment during one iteration have to be saved between subsequent iterations. The algorithm is a sequence of operations and function calls. Some of the functions used are the generated functions, as described in Subsection 4.1.1 and the remaining functions have to be manually implemented. The integration and differentiation functions have been assumed to use simple algorithms of backward difference and quadrature integration with the rectangle rule, as described in Section 3.6. Using other algorithms may require storing additional state and adding additional function arguments to integration and differentiation calls. The implementation of integration requires modifications depending on the dimension of η, which in turn depends on the robot model used. Because of this the integration has been written in a simplified way, not to complicate the pseudocode. In addition, the following functions have to be implemented:

measure_q collects the values of robot configuration variables, involving measuring motor angles and robot global position; some values may be calculated using other program modules (e.g. localisation module) and some may use their previous values if they cannot be measured with the required frequency,

calculate_linearising_output applies the linearising output function $h(q)$, as described in Subsection 2.3.1,

calculate_next_trajectory_point calculates the desired value of h at the current time instant; the function should internally track current time if the trajectory is defined using analytical functions or it could implement calculation of new h_d value based on control provided by the user using a controller device,

prepare_output saturates the control signal to an appropriate range depending on the physical construction of the robot to avoid damaging the motors; this can also involve optional filtering of output to avoid rapid acceleration that could damage the motors,

apply_control sends the values of control signals to robot actuators (motors/motor drivers).

As can be noticed, Algorithm 1 does implement only three of four steps shown in Figure 3.1a. Careful analysis of the algorithm allows to identify operation that can be moved to the end of the ControlStep function in order to decrease the delay between measurement of inputs and applying calculated controls. Most notably, the calculations of trajectory could be moved to the end of the function to calculate the values for

the next iteration[‡]. Apart from this, the shifting of some of the values from current sample to the previous one can be done at the end of the function. Algorithm 2 shows a version of the implementation optimised for minimising the time between measurement and actuation. Although in this case the performance improvements may not be noticeable when compared to the time taken by other operations, in some case such an algorithm timing optimisation procedure can lead to significant improvements.

This example implementation may be further modified. For instance, it is possible to implement an algorithm that combines both static and dynamic linearisation, by first computing the determinant $\det(D)$, and, depending on whether its value is close to zero or not, the dynamic or static linearisation algorithm can be used correspondingly.

[‡]It is the case only if the trajectory is known beforehand, so this optimisation cannot be applied if the trajectory is defined by the signal from controller device operated by the user (e.g. a joystick).

Algorithm 2: Dynamic linearisation implementation optimised for response time

parameters
- K_1 :: float[25]
- K_2 :: float[25]

persistent state
- $h_{[n-1]}$ :: float[5] *» for computing the 1st derivative*
- $h_{d[n]}$ :: float[5]
- $\dot{h}_{d[n]}$ :: float[5] *» for precalculation of trajectory*
- $\ddot{h}_{d[n]}$ :: float[5] *» for precalculation of trajectory*
- $h_{d[n-1]}$ :: float[5] *» for computing the 1st derivative*
- $\dot{h}_{d[n-1]}$ :: float[5] *» for computing the 2nd derivative*
- $\eta_{[n]}$ :: float[5] *» for integration of* u

function ControlStep(Δt)

local variables
- $q_{[n]}$ :: float[9]
- $h_{[n]}$ :: float[5]
- $\dot{h}_{[n]}$ :: float[5]
- $\nu_{[n]}$ :: float[5]
- $u_{[n]}$ :: float[5]
- $\dot{\eta}_{[n]}$ :: float[M], $M \in \{1, \ldots, 5\}$ *» depending on the model used*

procedure MeasureInputs
- $q_{[n]} \leftarrow$ measure_q()
- $h_{[n]} \leftarrow$ calculate_linearising_output($q_{[n]}$)

procedure CalculateOutput
- $\dot{h}_{[n]} \leftarrow$ differentiate($h_{[n]}$, $h_{[n-1]}$, Δt)
- $\nu_{[n]} \leftarrow$ feedback_v(K_1, K_2, $h_{[n]}$, $\dot{h}_{[n]}$, $h_{d[n]}$, $\dot{h}_{d[n]}$, $\ddot{h}_{d[n]}$)
- $u_{[n]} \leftarrow$ dynamic_lin_u($\nu_{[n]}$, $\eta_{[n]}$, $q_{[n]}$)
- *» integration of some or all of the elements of* u
- $\dot{\eta}_{[n]} \leftarrow u_{[n]}$ *» take the elements of* u *that have to be integrated*
- $\eta_{[n]} \leftarrow u_{[n]}$ *» take the elements of* u *that do not have to be integrated*
- $\eta_{[n]} \leftarrow$ integrate($\eta_{[n]}$, $\dot{\eta}_{[n]}$, Δt) *» integrate derivatives of* η

procedure ApplyOutput
- $\eta_{[n]} \leftarrow$ prepare_output($\eta_{[n]}$) *» output saturation, optional filtering*
- apply_control($\eta_{[n]}$)

procedure UpdateState
- *» precompute future trajectory*
- $h_{d[n-1]} \leftarrow h_{d[n]}$
- $h_{d[n]} \leftarrow$ calculate_next_trajectory_point(Δt)
- $\dot{h}_{d[n]} \leftarrow$ differentiate($h_{d[n]}$, $h_{d[n-1]}$, Δt)
- $\ddot{h}_{d[n]} \leftarrow$ differentiate($\dot{h}_{d[n]}$, $\dot{h}_{d[n-1]}$, Δt)
- $\dot{h}_{d[n-1]} \leftarrow \dot{h}_d$
- *» save current linearising output value as the previous one*
- $h_{[n-1]} \leftarrow h_{[n]}$

Chapter 5

Hardware analysis

The primary objective of this work is to design a robot controller that could be implemented on the actual robot. The methodology for controller implementation has already been shown from software perspective, however some unwritten assumptions made about the capabilities of the underlying robot hardware. To complete Hogger[2] robot control considerations, the hardware required for the implementation has to be analysed.

In this chapter hardware required for Hogger[2] robot control will be analysed using the existing robot prototype [16] as the reference. An overview will be provided first, describing the theoretical aspects of hardware interaction from software perspective. Following, the basis for determining the onboard computer requirements will be outlined. Finally, the necessary modifications to the existing Hogger[2] prototype will be described.

5.1 Interaction with hardware

It is helpful to consider hardware requirements from the perspective of the mathematical model developed for the robot. This method allows to identify the input and output operations that have to be implemented in software, which leads to the hardware that has to exist in the robot.

The robot configuration vector that corresponds to Hogger[2] construction is the one used in *Full Model 1* and *Full Model 2*, and shown in Equation (2.1). The configuration can be split into two parts: the global position and orientation of the robot (x, y, θ_0), and the internal angles of motor rotations $(\varphi_1, \theta_1, \psi_1, \varphi_2, \theta_2, \psi_2)$. The angles can be further split into hemispheres rotation angles $(\varphi_1, \theta_1, \varphi_2, \theta_2)$ and the angles of hemispheres spinning movements (ψ_1, ψ_2).

5.1.1 Controller outputs

Identification of controller output is a straightforward task. The controller output vector η is defined by the Equations (2.3) and (2.4), and is homogeneous to a velocity. While the global velocity $(\dot{x}, \dot{y}, \dot{\theta_0})$ of the robot are controlled indirectly, we have to be able to control the velocities of all of the other 6 configuration variables.

In the Hogger[2] prototype (see [16] and Figures 2.2b, 2.2c), servo motors have been used to control the variables $(\varphi_1, \theta_1, \varphi_2, \theta_2)$, and for the variables (ψ_1, ψ_2) brushless

direct current (BLDC) motors have been used. It has to be noted that the standard BLDC motor controllers control the velocity by default. On the other hand, servomechanisms most often take position as their control input, and so the values obtained in control algorithm output η may have to be further integrated to obtain the position values. This also generates another problem, that is, the velocities in η cannot be higher that those stated in servo specifications, and, while this is also true for BLDC motors, the velocities of servo motors are often relatively low, which has to be considered.

5.1.2 Controller inputs

The subject of robot state measurements is much more complicated. In general in a control algorithm we have to be able to measure the whole robot configuration vector q. Analysing the diagrams in Figures 2.6 and 2.7, we can see that this is true. Even though some blocks in the algorithm require only the vector

$$h(q) = (x, y, \theta_0, \psi_1, \psi_2)^T, \tag{5.1}$$

which itself does not contain all elements of q, the matrices $\frac{\delta h}{\delta q}G$, K_{dd} and the vector P depend on other elements of q. Detailed analysis of the expressions shows that for both full models, the linearisation blocks always uses the variables $(\theta_0, \varphi_1, \theta_1, \varphi_2, \theta_2)$, and the rest of q can be found in Equation (5.1).

Global position

Global positioning aims to estimate the position of the robot in an external reference frame. This can be achieved either using dead reckoning or global localisation methods [29].

Dead reckoning is the most convenient methods in the sense that the localisation is based only on the robot internal measurements. Based on the measured robot velocity its position in the world is calculated by transforming the measured velocities to the global reference frame and integrating the results. It is usually faster and less computationally complicated technique than global localisation. On the other hand, dead reckoning suffers from the problem of drift, where small errors in velocity measurements are integrated, resulting in the error of global localisation increasing with time. Because of the weakness of dead reckoning, global positioning methods have to be used. There are a lot of available methods, which can be divided into indoor and outdoor methods [29].

It should be noted that from the point of view of Hogger[2] control, global position error does not influence the local correctness of control, because it is only used for calculating feedback errors ($h - h_d$, etc.), so the localisation errors in practice have the same result as if the desired trajectory h_d was changed.

Servo motors

The measurement of servo motor angles and their control is a problematic task. A servomechanism is a device that already includes a feedback control system [53], so it has to measure its position which is then used for tracking error calculation. However, most

standard servos do not provide output with the information about current position. This means that, in order to measure the angle, another solution has to be found.

The easiest (and most expensive) solution is to use servo motors that provide current position output. These can be divided into two groups. The first one are highly advanced servos that provide multiple configuration registers accessed over serial protocols, e.g. Dynamixel servos [38]. The second one includes cheaper servomechanisms, that provide the signal from the internal potentiometer as an analog output pin. These include FEETECH FS90-FB servo [13] or Analog Feedback Servos by Adafruit [2], however this kind of servos usually have poor motor parameters when compared to other models.

A simple alternative for servomechanisms that provide current angle information by design is to use a modified standard servo. Taking advantage of the fact that each servomechanism has a built-in potentiometer, a wire can be connected to the potentiometer's output signal. This allows to read the voltage directly by the MCU which can be then converted into angle information after proper calibration.

Another approach, which requires no hardware modifications is to simulate the servo movement based on the velocity given in servo specification. Although theoretically applicable, this method does not provide very accurate data, because any disturbances that happen due to insufficient motor torque will not be registered.

As the last proposed solution, servomechanism can be replaced with a custom made servo, i.e. a DC (or other*) motor with an attached encoder. This in practice may be more costly in terms of money and development time, but can be tuned specifically for the controlled system, yielding the best results.

BLDC motors

BLDC motors have multiple advantages over brushed DC motors, at the cost of more complicated controller design. The main advantages of BLDC motors are high power-to-weight ratio, high speed and higher durability due to lack of brushes that wear out [49].

As in the case of servomechanisms, standard BLDC motor controllers do not provide position or velocity feedback. At the same time, this limitation is less significant in the case of BLDC motors in Hogger[2] robot, because of much higher velocity of their operation. The motors are used to spin the hemispheres, and their velocity has been included in the mathematical model in the linearising output h, and thus its value can be controlled using h_d, specifically it can be set to a high constant value. In such case, an assumption can be made that the current velocity of the motor is equal to its desired velocity, and the error introduced by this assumption may be negligible. If however the control results are not satisfactory, a controller with velocity feedback should be found.

*However stepper motors should generally be avoided in high-speed applications, as they are designed to work well in open loop, but in the presence of high, unpredictable torque, they can lose steps, which introduces unrecoverable errors. And even though adding an encoder solves the problem, the motor will still have worse power-to-weight ratio than other types of motors.

5.2 Onboard computer

An important aspect of controller hardware is the choice of computer device which will be used to perform algorithm calculations. The choice depends on what calculations the robot has to perform onboard and what hardware has to be controlled.

Generally, for low latency, high precision operation a bare-metal/RTOS-based MCU is the best choice, as has been discussed in Section 3.2. When considering higher level algorithms, e.g. simultaneous localization and mapping (SLAM), it may be desirable to use more powerful microcomputer alongside the MCU. In such scenario, the lowest level control loop (i.e. the control algorithms as discussed in this thesis) can be implemented on the MCU, while the main computer performs the heavy computations and sends commands to the MCU over some serial interface, by setting the value of h_d (see Figures 2.6, 2.7).

5.2.1 Computing power

It is important to find a balance between computing power and price when choosing an MCU for an embedded system. The computing power required for the application mainly depends on the desired control loop frequency and the complexity of the algorithm itself (and any other computations that have to be performed).

The loop frequency determination in case of feedback linearisation algorithms can be done using standard methods for linear control systems [12]. Feedback linearisation[†] transforms the system into an equivalent linear system. The analysis of frequency response of the system can be then used to tune the gains of the feedback controller, e.g. the K matrix for the control law in Equation (2.11). As a rule of thumb, control frequency between 100 Hz and 1 kHz can be chosen as an initial guess.

To determine the final requirements it is best to perform tests on a similar device, and then, given the required sampling frequency, choose the MCU comparing the device capabilities, considering factors such as CPU clock frequency, FPU and DSP instructions availability, etc. For these reasons, it is more convenient to choose a stronger MCU when developing a prototype to avoid wasting time on hardware replacements.

5.2.2 Hardware interactions

Another, probably even more critical factor that has to be taken into account when choosing MCU, is the availability of peripherals for interfacing with sensors and actuators used in the robot. This involves serial communication interfaces (UART, SPI, I^2C, etc.), timers used for PWM signal generation, and analog signal converters (ADCs/DACs).

5.3 Necessary hardware modifications

The initial tests of the implementation of control algorithms developed in this thesis shall be performed using the existing prototype of Hogger[2] robot. The prototype is a functional construction, however it is not fully prepared for the control task.

[†]In case of Hogger[2]: input-output linearisation, as opposed to full state linearisation.

5.3.1 Existing construction

Hogger[2] prototype described in [16] has been equipped with four PowerHD HD-1501MG servos [35] and two EMAX BL2215/25 BLDC motors [11]. These motors have been chosen based on the required parameters calculated in that thesis[‡]. The construction has been tested using standard remote control (RC) radio system, namely RadioLink T6EAP [37]. Robot control have been simplified to allow for direct control by a human operator without the need for complex motion controller. The servo positions and BLDC motor velocities have been interlocked ($\theta_1 = \theta_2$, $\varphi_1 = \varphi_2$, $\dot{\psi}_1 = \dot{\psi}_2$). Such a model has been described in [20, Section 4.4]. This simplification effectively reduces the number of control inputs to 3. During the tests it was possible to achieve the desired robot behaviour, although potential improvements concerning its mechanical construction have been described.

5.3.2 Feedback measurement method

In the existing prototype no state feedback measurements have been performed, however state measurement is crucial for implementing the robot controller. Because the servo motors used in the prototype do not provide position information, one of the methods described in Section 5.1.2 has to be used. The method of soldering an additional wire directly to the servo internal potentiometer will be used for the prototype implementation. BLDC motor velocity measurements will not be performed, but instead, the velocities will be set to high values and the tracking will be assumed to be perfect, as discussed in Section 5.1.2. Initially, for measuring robot global position, an indoor motion capture system will be used, and a radio transmitter-receiver pair, will be used for communication with the system[§]. Later, more sophisticated solutions can be tested.

5.3.3 Microcontroller unit

In order to be able to test the implementation of control algorithms developed in Chapter 4, the robot must be equipped with an MCU that will be able to control all the motors and read the robot state. The following MCU capabilities are required:

- 4 PWM outputs for setting servomechanisms target positions,
- 2 PWM outputs for setting BLDC motors target positions,
- 4 ADC channels for measuring servo potentiometers positions,
- up to 5[¶] timer channels with input capture functionality ('PWM inputs') for reading signals from RC receiver.

[‡]There is a mistake in [16, Equation (2.9)] — the final value of M_d should be equal to 0.125 N m instead of 0.25 N m.

[§]nRF24L01 modules [41] can be used for this task. The modules have already been tested by the author in [7], and introduce a delay of 158.4 µs, which should be enough for control at the frequency of 1 kHz (delay of 15.8 % of the control period).

[¶]This is the number of control inputs η in the model, however for simple human control 2 or 3 channels will be enough.

- availability of serial interfaces for communicating with other sensors used, e.g. motion-capture system (through radio module), IMU, etc.

Beside the things mentioned above, the selected CPU should include an FPU, because the algorithms have been implemented using floating-point arithmetic.

Estimation of required CPU computing power has been done using the, somewhat arbitrarily chosen, STM32F407 microcontroller (see Table 4.2). As can be seen in Figures 4.4 and 4.5, the most time-consuming computation when using CSE optimisation takes 63 μs for the case of dynamic linearisation of *Simplified Model*. Such period can, in theory, allows for control loop frequency even as high as 15 kHz. Even though this frequency will probably be impossible to achieve due to the time required for measurement, actuation and side computations, the STM32F407 MCU should be capable of operating at control loop frequency well above 1 kHz.

Considering the capabilities of STM32F407 MCU it is a reasonable choice for implementing. It easily meets all the requirements mentioned above, concerning both computing power and peripherals availability, as it includes up to 24 ADC channels, up to 17 timers (PWM inputs/outputs) and multiple serial interfaces. The specific choice of STM32F407 is a result of the author's preference and possession of STM32F4Discovery development board, which can greatly reduce the initial amount of work required for preparing electronic circuits for the controller, however a lot of other MCUs based on ARM Cortex-M4 core would suffice.

Chapter 6

Conclusion

The main goal of this thesis was to design a controller implementing algorithms for Hogger[2] robot, to explore the methodology behind the implementation process and to evaluate hardware required for the robot. The assumed goals have been accomplished.

In the work the specific problem of Hogger[2] control has been described. The algorithms already developed for the robot have been summarised, focussing on the issues related to the implementation on the real robot. The results of previous simulations have been compared, identifying the most promising algorithms.

The implementation considerations have been described, underlining the issues specific to embedded systems. A methodology process which includes symbolic modelling of mathematical algorithms description and low-level code generation have been proposed. It has been shown that free tools supporting this kind of methodology are available, and when needed, more complete commercial solutions can be chosen. Furthermore, the structure of code for the task of robot control has been outlined and decomposed into logical blocks.

To complete the discussion, a selected example of controller design has been shown by implementing specific blocks of the controller for Hogger[2] robot. The process of symbolic modelling in SymPy environment has been explained and the optimisation possibilities available during code generation have been explored. An example of a complete implementation has been shown as pseudocode for the dynamic feedback linearisation algorithm, explaining the data that has to be stored between successive iterations.

The hardware required for proper robot state measurements and controller implementation has been analysed, providing multiple solutions and highlighting potential problems. Finally, the changes to the existing prototype that are required for testing the control algorithms implementation have been proposed.

There is still a lot of work to be done to complete Hogger[2] robot. This thesis focused on the methodology and the process of control implementation, leaving the actual implementation as a future goal. The ground for concrete implementation has been prepared, however several problems have to be solved before. The hardware of Hogger[2] prototype has to be extended, as described in Chapter 5, to allow for reliable measurements of robot state. The task of robot localisation is another issue that has to be considered. First tests should probably be performed using only the motion capture system available at WUST, however for better control IMU measurements should be used too. An interesting work direction may be to test localisation techniques based

on Kalman filter that could use odometry data, even in the presence of vibrations that, until now, have always emerged during robot control.

The benchmarks performed on the generated code are not ideal, due to possibly unrealistic testing data, but should serve as good approximation of the processor capabilities. To make the tests more exact, the object model could be implemented too, but the results may not provide any useful information. If more complex calculations are ever required, it should be considered, whether the CPU used for the robot should be exchanged for a more powerful one or the calculations should be split between more CPUs.

Not all the algorithms presented in [21] have been implemented. In future work Samson's algorithm should be implemented, as it has shown promising results in simulations (see Table 2.1). Furthermore, the algorithms of Hogger^2 control do not consider its unique dynamic properties. This is an important area to be explored, as the spinning hemispheres may produce Coriolis forces that have significant influence on the robot behaviour. Dynamic robot properties have not been tested in the simulations, so the algorithms that have been considered promising may in fact have low performance or even be completely unusable. This however will be tested when the existing algorithms will be finally implemented for the robot prototype. Another algorithmic challenge is designing control that can robustly control the robot near singularities, avoiding unstable zero dynamics that may emerge in some configurations.

HOG drive is unquestionably an interesting field of research into new movement techniques for mobile robots. Surprisingly, it has not yet been deeply examined and we cannot conclude whether it may become successful in practical applications or not. Certainly, there are multiple directions that are yet to be explored, and hopefully this is not the last work related to Hogger^2 robot and other works are to come in near future.

Bibliography

[1] E. Ackerman. *You've Never Seen a Robot Drive System Like This Before*. 7th June 2011. URL: https://spectrum.ieee.org/automaton/robotics/diy/youve-never-seen-a-drive-system-like-this-before.

[2] Adafruit. *Analog Feedback Micro Servo — Metal Gear*. URL: https://www.adafruit.com/product/1450.

[3] A. V. Aho et al. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[4] *AUTOSAR — AUTomotive Open System ARchitecture*. URL: https://www.autosar.org/.

[5] K. J. Åström and R. M. Murray. *Analysis and Design of Feedback Systems, Chapter 10*. California Institute of Technology. 2004. URL: https://www.cds.caltech.edu/~murray/courses/cds101/fa04/caltech/am04_ch10-3nov04.pdf.

[6] C. Blackmore, O. Ray and K. Eder. 'Automatically Tuning the GCC Compiler to Optimize the Performance of Applications Running on the ARM Cortex-M3'. In: *CoRR* abs/1703.08228 (2017). URL: http://arxiv.org/abs/1703.08228.

[7] J. Boczar. 'A device for multimodal localization of indoor mobile robots'. Bachelor's thesis. Wrocław University of Science and Technology, 2017.

[8] P. Caspi and O. Maler. 'From Control Loops to Real-Time Programs'. In: *Handbook of Networked and Embedded Control Systems*. Ed. by D. Hristu-Varsakelis and W. S. Levine. Boston, MA: Birkhäuser Boston, 2005, pp. 395–418.

[9] E. Clarke, B. Fite and J. Reyer. 'Design, Development, and Analysis of a Hemispherical Singularity Drive System for Instantaneously Omnidirectional Motion With Kinematic Isotropy'. In: Nov. 2016.

[10] B. Dahlgren et al. *Automatic Code Generation with SymPy*. 2017. URL: https://www.sympy.org/scipy-2017-codegen-tutorial/.

[11] Emax. *Emax BL2215/25 950kv Outrunner Brushless Motor*. URL: https://www.headsuphobby.com/Emax-BL221525-950kv-Outrunner-Brushless-Motor--Discontinued_ep_94-1.html.

[12] F. Fairman. *Linear Control Theory: The State Space Approach*. Wiley, 1998.

[13] FEETECH. *FEETECH FS90-FB Micro Servo with Position Feedback*. URL: https://www.pololu.com/product/3436.

[14] Free Software Foundation, Inc. *Using the GNU Compiler Collection (GCC): Options That Control Optimization*. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[15] D. N. Góral. *Był sobie robot.. Nietypowy projekt wykonany w programie Autodesk Inventor*. PCC Polska. 2017. URL: http://www.pccpolska.pl/platforma-mobilna-napedzana-dwiema-polsferami-studium-tematu/.

[16] D. N. Góral. 'Two HOG wheel mobile robot construction'. Bachelor's thesis. Wrocław University of Science and Technology, 2017.

[17] R. Hametner et al. 'Implementation Guidelines for Closed Loop Control Algorithms on PLCs'. In: Feb. 2013.

[18] J. R. Hayes. *Modular Programming in C*. 2001. URL: https://www.embedded.com/design/prototyping-and-development/4023876/Modular-Programming-in-C.

[19] T. K. Jespersen. 'Kugle — Modelling and Control of a Ball-balancing Robot'. Master's thesis. Aalborg University, 2019.

[20] P. Joniak. 'Analysis of two HOG wheel mobile robot behaviour'. Bachelor's thesis. Wrocław University of Science and Technology, 2014.

[21] P. Joniak. 'Control problem for two HOG wheel mobile robot'. Master's thesis. Wrocław University of Science and Technology, 2016.

[22] P. Joniak and R. Muszyński. 'Path Following for Two HOG Wheels Mobile Robot'. In: *Journal of Automation, Mobile Robotics and Intelligent Systems* 11.02 (2017), pp. 75–81.

[23] A. Karkare. *CS 738: Advanced Compiler Optimizations: Overview of Optimizations*. 2018. URL: https://karkare.github.io/cs738/lecturenotes/02OptsOverviewSlides.pdf.

[24] T. Kisuki et al. *Iterative Compilation in Program Optimization*. 2000. URL: https://pdfs.semanticscholar.org/2b77/619723635067e314871c09ce27a2d5529bcb.pdf.

[25] D. E. Knuth. 'Structured programming with go to statements'. In: *Computing Surveys* 6 (1974), pp. 261–301.

[26] M. Larabel. *GCC Soars Past 14.5 Million Lines Of Code & I'm Real Excited For GCC 5*. 5th Jan. 2015. URL: https://www.phoronix.com/scan.php?page=news_item&px=MTg3OTQ.

[27] J. Malewicz. 'Prototyping environment for robotic subsystems'. Bachelor's thesis. Wrocław University of Science and Technology, 2008.

[28] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[29] R. Mautz. *Indoor Positioning Technologies*. Geodätisch-geophysikalische Arbeiten in der Schweiz. ETH Zurich, Department of Civil, Environmental, Geomatic Engineering, Institute of Geodesy and Photogrammetry, 2012. URL: https://books.google.pl/books?id=BsHpMgEACAAJ.

[30] A. Mazur. 'Sterowanie Oparte na Modelu dla Nieholonomicznych Manipulatorów Mobilnych'. PhD thesis. Wrocław University of Science and Technology, 2009.

[31] D. McCandless, P. Doughty-White and M. Quick. *Codebases: Millions lines of code*. 24th Sept. 2015. URL: https://informationisbeautiful.net/visualizations/million-lines-of-code/.

[32] A. Meurer et al. 'SymPy: symbolic computing in Python'. In: *PeerJ Computer Science* 3 (Jan. 2017), e103. URL: https://doi.org/10.7717/peerj-cs.103.

[33] A. Micaelli and C. Samson. *Trajectory tracking for unicycle-type and two-steering-wheels mobile robots*. Research Report RR-2097. INRIA, 1993. URL: https://hal.inria.fr/inria-00074575/file/RR-2097.pdf.

[34] K. Mørken. *Numerical Algorithms and Digital Representation*. Chapter 11: Numerical Differentiation and Integration. 2015. URL: https://www.uio.no/studier/emner/matnat/math/MAT-INF1100/h08/kompendiet/komp.html.

[35] PowerHD. *Power HD High-Torque Servo 1501MG*. URL: https://www.pololu.com/product/1057.

[36] P. Puschner. *Introduction to Real-Time Systems*. Technische Universität Wien. 2017. URL: https://ti.tuwien.ac.at/cps/teaching/courses/real-time-systems/slides/rts01_definitions.pdf.

[37] RadioLink. *RadioLink T6EAP Instruction Manual*. URL: https://www.manualslib.com/manual/921053/Radiolink-T6eap.html.

[38] ROBOTIS. *DYNAMIXEL*. URL: http://www.robotis.us/dynamixel/.

[39] M. Rybczyński. 'Model of mobile robot with HOG wheel'. Bachelor's thesis. Wrocław University of Science and Technology, 2013.

[40] J. Sauermann and M. Thelen. *Realtime Operating Systems. Concepts and Implementation of Microkernels for Embedded Systems*. 1997. URL: http://dsp-book.narod.ru/DSPROSES.pdf.

[41] N. Semiconductor. *nRF24L01, Single chip 2.4 GHz Transceiver*. URL: https://www.sparkfun.com/datasheets/Components/nRF24L01_prelim_prod_spec_1_2.pdf.

[42] B. Siciliano and O. Khatib. *Springer Handbook of Robotics*. Springer Handbooks. Springer International Publishing, 2016.

[43] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. San Diego, CA, USA: California Technical Publishing, 1997.

[44] *SymEngine*. URL: https://github.com/symengine/symengine.

[45] K. Tchoń and R. Muszyński. *Mathematical Methods of Automation and Robotics*. Lecture Notes in Automation and Robotics. 2017.

[46] K. Tchoń et al. *Manipulatory i roboty mobilne: modele, planowanie ruchu, sterowanie*. Problemy Współczesnej Nauki, Teoria i Zastosowania. Robotyka. Akademicka Oficyna Wydawnicza PLJ, 2000.

[47] Texas Instruments Wiki. *Floating Point Optimization*. URL: http://processors.wiki.ti.com/index.php/Floating_Point_Optimization.

[48] Wikipedia, The Free Encyclopedia. *Algorithms for calculating variance: Welford's online algorithm*. URL: https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_online_algorithm.

[49] Wikipedia, The Free Encyclopedia. *Brushless DC electric motor*. URL: https://en.wikipedia.org/wiki/Brushless_DC_electric_motor.

[50] Wikipedia, The Free Encyclopedia. *Gimbal*. URL: https://en.wikipedia.org/wiki/Gimbal.

[51] Wikipedia, The Free Encyclopedia. *Hemispherical omnidirectional gimbaled wheel*. URL: https://en.wikipedia.org/wiki/Hemispherical_omnidirectional_gimbaled_wheel.

[52] Wikipedia, The Free Encyclopedia. *Operating system*. URL: http://en.wikipedia.org/w/index.php?title=Operating\%20system&oldid=898517811.

[53] Wikipedia, The Free Encyclopedia. *Servomechanism*. URL: https://en.wikipedia.org/wiki/Servomechanism.

[54] The MathWorks, Inc. *MATLAB*. URL: https://www.mathworks.com/products/matlab.html.

[55] Unknown. *Hemisphere Drive Speedster*. Oct. 1938. URL: http://blog.modernmechanix.com/hemisphere-drive-speedster/.

[56] J. Weidendorfer. *Analysis and Optimization of the Memory Access Behavior of Applications*. URL: http://calcul.math.cnrs.fr/IMG/pdf/weidendorfer.pdf.

[57] Wolfram Research, Inc. *Mathematica*. URL: https://www.wolfram.com.

Appendix A

Generated code examples

To give the reader a better view of the complexity of Hogger[2] control algorithms, some examples of generated code have been provided in this appedinx. At the same time, it can be seen how CSE optimisation works for cases more complex than the one provided in Subsection 3.4.2.

Listing A.1: Static linearisation for *Full Model 2* without CSE

```
void full_2_static_lin_eta(float eta[5], const float u[5], const float q[9])
{
    eta[0] = -u[3]*sinf(q[4]) + u[0]*sinf(q[2])/R - u[1]*cosf(q[2])/R;
    eta[1] = u[3]*cosf(q[4])*tanf(q[3]) + u[0]*cosf(q[2])/(R*cosf(q[3])) +
        u[1]*sinf(q[2])/(R*cosf(q[3]));
    eta[2] = u[3];
    eta[3] = u[4]*cosf(q[7])*tanf(q[6]) + 2*l*u[2]/(R*cosf(q[6])) +
        u[0]*cosf(q[2])/(R*cosf(q[6])) + u[1]*sinf(q[2])/(R*cosf(q[6]));
    eta[4] = u[4];
}
```

Listing A.2: Static linearisation for *Full Model 2* with CSE

```
void full_2_cse_static_lin_eta(float eta[5], const float u[5], const float q[9])
{
    float x0 = q[4];
    float x1 = q[2];
    float x2 = sinf(x1);
    float x3 = 1.0F/R;
    float x4 = u[0]*x3;
    float x5 = cosf(x1);
    float x6 = u[1]*x3;
    float x7 = q[3];
    float x8 = 1.0F/cosf(x7);
    float x9 = x4*x5;
    float x10 = x2*x6;
    float x11 = q[6];
    float x12 = 1.0F/cosf(x11);

    eta[0] = -u[3]*sinf(x0) + x2*x4 - x5*x6;
    eta[1] = u[3]*cosf(x0)*tanf(x7) + x10*x8 + x8*x9;
    eta[2] = u[3];
    eta[3] = 2*l*u[2]*x12*x3 + u[4]*cosf(q[7])*tanf(x11) + x10*x12 + x12*x9;
    eta[4] = u[4];
}
```

Listing A.3: Dynamic linearisation for *Full Model 2* without CSE

```
void full_2_dynamic_lin_u(float u[5], const float v[5], const float eta[5], const float q[9])
{
    u[0] = -v[3]*sinf(q[4]) - ((1.0F/2.0F)*R*(R*(eta[1]*cosf(q[3]) - eta[2]*sinf(q[3])*cosf(q[4]) -
                    eta[3]*cosf(q[6]) + eta[4]*sinf(q[6])*cosf(q[7]))*eta[0]*sinf(q[2]) + R*(eta[1]*cosf(q[3])
                    - eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                    eta[4]*sinf(q[6])*cosf(q[7]))*eta[1]*cosf(q[3])*cosf(q[2]) +
                2*l*eta[0]*eta[1]*sinf(q[3])*sinf(q[2]) + (-R*(eta[1]*cosf(q[3]) -
                        eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                        eta[4]*sinf(q[6])*cosf(q[7]))*sinf(q[3])*cosf(q[2])*cosf(q[4]) + R*(eta[1]*cosf(q[3])
                        - eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                        eta[4]*sinf(q[6])*cosf(q[7]))*sinf(q[2])*sinf(q[4]) +
                    2*l*(eta[0]*sinf(q[2])*cosf(q[3])*cosf(q[4]) - eta[1]*sinf(q[3])*sinf(q[2])*sinf(q[4]) +
                        eta[1]*cosf(q[2])*cosf(q[4])))*eta[2])/l + v[1])*cosf(q[2])/R +
        (-1.0F/2.0F*R*(-R*(eta[1]*cosf(q[3]) - eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                            eta[4]*sinf(q[6])*cosf(q[7]))*eta[0]*cosf(q[2]) + R*(eta[1]*cosf(q[3]) -
                            eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                            eta[4]*sinf(q[6])*cosf(q[7]))*eta[1]*sinf(q[2])*cosf(q[3]) -
                       2*l*eta[0]*eta[1]*sinf(q[3])*cosf(q[2]) - (R*(eta[1]*cosf(q[3]) -
                                eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                                eta[4]*sinf(q[6])*cosf(q[7]))*sinf(q[3])*sinf(q[2])*cosf(q[4]) +
                            R*(eta[1]*cosf(q[3]) - eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                                eta[4]*sinf(q[6])*cosf(q[7]))*sinf(q[4])*cosf(q[2]) +
                            2*l*(eta[0]*cosf(q[3])*cosf(q[2])*cosf(q[4]) -
                                eta[1]*sinf(q[3])*sinf(q[4])*cosf(q[2]) -
                                eta[1]*sinf(q[2])*cosf(q[4])))*eta[2])/l + v[0])*sinf(q[2])/R;
    u[1] = v[3]*cosf(q[4])*tanf(q[3]) + ((1.0F/2.0F)*R*(R*(eta[1]*cosf(q[3]) - eta[2]*sinf(q[3])*cosf(q[4]) -
                    eta[3]*cosf(q[6]) + eta[4]*sinf(q[6])*cosf(q[7]))*eta[0]*sinf(q[2]) + R*(eta[1]*cosf(q[3])
                    - eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                    eta[4]*sinf(q[6])*cosf(q[7]))*eta[1]*cosf(q[3])*cosf(q[2]) +
                2*l*eta[0]*eta[1]*sinf(q[3])*sinf(q[2]) + (-R*(eta[1]*cosf(q[3]) -
                        eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                        eta[4]*sinf(q[6])*cosf(q[7]))*sinf(q[3])*cosf(q[2])*cosf(q[4]) + R*(eta[1]*cosf(q[3])
                        - eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                        eta[4]*sinf(q[6])*cosf(q[7]))*sinf(q[2])*sinf(q[4]) +
                    2*l*(eta[0]*sinf(q[2])*cosf(q[3])*cosf(q[4]) - eta[1]*sinf(q[3])*sinf(q[2])*sinf(q[4]) +
                        eta[1]*cosf(q[2])*cosf(q[4])))*eta[2])/l + v[1])*sinf(q[2])/(R*cosf(q[3])) +
        (-1.0F/2.0F*R*(-R*(eta[1]*cosf(q[3]) - eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                            eta[4]*sinf(q[6])*cosf(q[7]))*eta[0]*cosf(q[2]) + R*(eta[1]*cosf(q[3]) -
                            eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                            eta[4]*sinf(q[6])*cosf(q[7]))*eta[1]*sinf(q[2])*cosf(q[3]) -
                       2*l*eta[0]*eta[1]*sinf(q[3])*cosf(q[2]) - (R*(eta[1]*cosf(q[3]) -
                                eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                                eta[4]*sinf(q[6])*cosf(q[7]))*sinf(q[3])*sinf(q[2])*cosf(q[4]) +
                            R*(eta[1]*cosf(q[3]) - eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                                eta[4]*sinf(q[6])*cosf(q[7]))*sinf(q[4])*cosf(q[2]) +
                            2*l*(eta[0]*cosf(q[3])*cosf(q[2])*cosf(q[4]) -
                                eta[1]*sinf(q[3])*sinf(q[4])*cosf(q[2]) -
                                eta[1]*sinf(q[2])*cosf(q[4])))*eta[2])/l + v[0])*cosf(q[2])/(R*cosf(q[3]));
    u[2] = v[3];
    u[3] = v[4]*cosf(q[7])*tanf(q[6]) + 2*l*(-1.0F/2.0F*R*(-(eta[0] + eta[2]*sinf(q[4]) -
                    eta[4]*sinf(q[7]))*eta[3]*sinf(q[6]) - (eta[0] + eta[2]*sinf(q[4]) -
                    eta[4]*sinf(q[7]))*eta[4]*cosf(q[6])*cosf(q[7]) + eta[0]*eta[1]*sinf(q[3]) +
                eta[0]*eta[2]*cosf(q[3])*cosf(q[4]) - eta[1]*eta[2]*sinf(q[3])*sinf(q[4]) +
                eta[3]*eta[4]*sinf(q[6])*sinf(q[7]))/l + v[2])/(R*cosf(q[6])) +
        ((1.0F/2.0F)*R*(R*(eta[1]*cosf(q[3]) - eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                eta[4]*sinf(q[6])*cosf(q[7]))*eta[0]*sinf(q[2]) + R*(eta[1]*cosf(q[3]) -
                eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                eta[4]*sinf(q[6])*cosf(q[7]))*eta[1]*cosf(q[3])*cosf(q[2]) +
            2*l*eta[0]*eta[1]*sinf(q[3])*sinf(q[2]) + (-R*(eta[1]*cosf(q[3]) - eta[2]*sinf(q[3])*cosf(q[4]) -
                    eta[3]*cosf(q[6]) + eta[4]*sinf(q[6])*cosf(q[7]))*sinf(q[3])*cosf(q[2])*cosf(q[4]) +
                R*(eta[1]*cosf(q[3]) - eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                    eta[4]*sinf(q[6])*cosf(q[7]))*sinf(q[4])*cosf(q[2]) +
                2*l*(eta[0]*sinf(q[2])*cosf(q[3])*cosf(q[4]) - eta[1]*sinf(q[3])*sinf(q[2])*sinf(q[4]) +
                    eta[1]*cosf(q[2])*cosf(q[4])))*eta[2])/l + v[1])*sinf(q[2])/(R*cosf(q[6])) +
        (-1.0F/2.0F*R*(-R*(eta[1]*cosf(q[3]) - eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                            eta[4]*sinf(q[6])*cosf(q[7]))*eta[0]*cosf(q[2]) + R*(eta[1]*cosf(q[3]) -
```

```
                            eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                            eta[4]*sinf(q[6])*cosf(q[7]))*eta[1]*sinf(q[2])*cosf(q[3]) -
                        2*l*eta[0]*eta[1]*sinf(q[3])*cosf(q[2]) - (R*(eta[1]*cosf(q[3]) -
                                eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                                eta[4]*sinf(q[6])*cosf(q[7]))*sinf(q[3])*sinf(q[2])*cosf(q[4]) +
                            R*(eta[1]*cosf(q[3]) - eta[2]*sinf(q[3])*cosf(q[4]) - eta[3]*cosf(q[6]) +
                                eta[4]*sinf(q[6])*cosf(q[7]))*sinf(q[4])*cosf(q[2]) +
                            2*l*(eta[0]*cosf(q[3])*cosf(q[2])*cosf(q[4]) -
                                eta[1]*sinf(q[3])*sinf(q[4])*cosf(q[2]) -
                                eta[1]*sinf(q[2])*cosf(q[4])))*eta[2])/l + v[0])*cosf(q[2])/(R*cosf(q[6]));
    u[4] = v[4];
}
```

Listing A.4: Dynamic linearisation for *Full Model 2* with CSE

```
void full_2_cse_dynamic_lin_u(float u[5], const float v[5], const float eta[5], const float q[9])
{
    float x0 = q[4];
    float x1 = sinf(x0);
    float x2 = q[2];
    float x3 = sinf(x2);
    float x4 = 2*l;
    float x5 = eta[0];
    float x6 = eta[1];
    float x7 = q[3];
    float x8 = sinf(x7);
    float x9 = x6*x8;
    float x10 = x5*x9;
    float x11 = x10*x4;
    float x12 = cosf(x7);
    float x13 = x12*x6;
    float x14 = eta[3];
    float x15 = q[6];
    float x16 = cosf(x15);
    float x17 = sinf(x15);
    float x18 = eta[4];
    float x19 = q[7];
    float x20 = cosf(x19);
    float x21 = x18*x20;
    float x22 = eta[2];
    float x23 = cosf(x0);
    float x24 = x22*x23;
    float x25 = R*(x13 - x14*x16 + x17*x21 - x24*x8);
    float x26 = x3*x5;
    float x27 = cosf(x2);
    float x28 = x25*x27;
    float x29 = x23*x6;
    float x30 = x12*x23;
    float x31 = x1*x3;
    float x32 = x23*x8;
    float x33 = R/l;
    float x34 = 2*v[1] + x33*(x11*x3 + x13*x28 + x22*(x25*x31 - x28*x32 + x4*(x26*x30 + x27*x29 - x31*x9)) +
            x25*x26);
    float x35 = 1.0F/R;
    float x36 = (1.0F/2.0F)*x35;
    float x37 = x27*x36;
    float x38 = 2*v[0];
    float x39 = x11*x27;
    float x40 = x27*x5;
    float x41 = x25*x40;
    float x42 = x25*x3;
    float x43 = x13*x42;
    float x44 = x1*x27;
    float x45 = x22*(x25*x44 + x32*x42 + x4*(-x29*x3 + x30*x40 - x44*x9));
    float x46 = x3*x36;
    float x47 = 1.0F/x12;
    float x48 = x34*x46;
    float x49 = x37*(x33*(x39 + x41 - x43 + x45) + x38);
    float x50 = 1.0F/x16;
    float x51 = x1*x22;
    float x52 = x18*sinf(x19);
    float x53 = x14*x17;
    float x54 = x5 + x51 - x52;
    u[0] = -v[3]*x1 - x34*x37 - x46*(x33*(-x39 - x41 + x43 - x45) - x38);
    u[1] = v[3]*x23*tanf(x7) + x47*x48 + x47*x49;
    u[2] = v[3];
    u[3] = l*x35*x50*(2*v[2] + x33*(-x10 - x12*x24*x5 + x16*x21*x54 + x51*x9 - x52*x53 + x53*x54)) +
        v[4]*x20*tanf(x15) + x48*x50 + x49*x50;
    u[4] = v[4];
}
```

Listing A.5: Dynamic linearisation for *JPTD Model* without CSE

```
void full_2_JPTD_dynamic_lin_u(float u[5], const float v[5], const float eta[5], const float q[9])
{
    u[0] = -v[3]*tanf(q[3])/(eta[2]*powf(cosf(q[4]), 2)) - (sinf(q[2])/cosf(q[3]) +
            cosf(q[2])*tanf(q[3])*tanf(q[4]))*((1.0F/2.0F)*powf(R, 2)*(eta[2]*powf(sinf(q[3]),
                    2)*cosf(q[2])*powf(cosf(q[4]), 2) - eta[2]*sinf(q[3])*sinf(q[2])*sinf(q[4])*cosf(q[4]) -
                eta[4]*sinf(q[3])*sinf(q[6])*cosf(q[2])*cosf(q[4])*cosf(q[7]) +
                eta[4]*sinf(q[6])*sinf(q[2])*sinf(q[4])*cosf(q[7]))*eta[2]/l + v[1])/(R*eta[2]*cosf(q[4])) +
            (sinf(q[2])*tanf(q[3])*tanf(q[4]) - cosf(q[2])/cosf(q[3]))*(-1.0F/2.0F*powf(R,
                        2)*(eta[2]*powf(sinf(q[3]), 2)*sinf(q[2])*powf(cosf(q[4]), 2) +
                            eta[2]*sinf(q[3])*sinf(q[4])*cosf(q[2])*cosf(q[4]) -
                            eta[4]*sinf(q[3])*sinf(q[6])*sinf(q[2])*cosf(q[4])*cosf(q[7]) -
                            eta[4]*sinf(q[6])*sinf(q[4])*cosf(q[2])*cosf(q[7]))*eta[2]/l +
                    v[0])/(R*eta[2]*cosf(q[4]));
    u[1] = -v[3]*tanf(q[4])/eta[2] + (-1.0F/2.0F*powf(R, 2)*(eta[2]*powf(sinf(q[3]),
                    2)*sinf(q[2])*powf(cosf(q[4]), 2) + eta[2]*sinf(q[3])*sinf(q[4])*cosf(q[2])*cosf(q[4]) -
                eta[4]*sinf(q[3])*sinf(q[6])*sinf(q[2])*cosf(q[4])*cosf(q[7]) -
                eta[4]*sinf(q[6])*sinf(q[4])*cosf(q[2])*cosf(q[7]))*eta[2]/l +
            v[0])*sinf(q[2])/(R*eta[2]*cosf(q[4])) - ((1.0F/2.0F)*powf(R, 2)*(eta[2]*powf(sinf(q[3]),
                        2)*cosf(q[2])*powf(cosf(q[4]), 2) - eta[2]*sinf(q[3])*sinf(q[2])*sinf(q[4])*cosf(q[4])
                    - eta[4]*sinf(q[3])*sinf(q[6])*cosf(q[2])*cosf(q[4])*cosf(q[7]) +
                    eta[4]*sinf(q[6])*sinf(q[2])*sinf(q[4])*cosf(q[7]))*eta[2]/l +
                v[1])*cosf(q[2])/(R*eta[2]*cosf(q[4]));
    u[2] = v[3];
    u[3] = -v[3]*tanf(q[3])/(eta[2]*powf(cosf(q[4]), 2)*tanf(q[6])*tanf(q[7])) + v[4]/(eta[4]*tanf(q[7])) +
        (-1.0F/2.0F*powf(R, 2)*(eta[2]*powf(sinf(q[3]), 2)*sinf(q[2])*powf(cosf(q[4]), 2) +
                                eta[2]*sinf(q[3])*sinf(q[4])*cosf(q[2])*cosf(q[4]) -
                                eta[4]*sinf(q[3])*sinf(q[6])*sinf(q[2])*cosf(q[4])*cosf(q[7]) -
                                eta[4]*sinf(q[6])*sinf(q[4])*cosf(q[2])*cosf(q[7]))*eta[2]/l +
         v[0])*(cosf(q[2])/(R*eta[4]*sinf(q[6])*sinf(q[7])) +
             sinf(q[2])*sinf(q[4])*tanf(q[3])/(R*eta[2]*powf(cosf(q[4]), 2)*tanf(q[6])*tanf(q[7])) -
             cosf(q[2])/(R*eta[2]*cosf(q[3])*cosf(q[4])*tanf(q[6])*tanf(q[7]))) + ((1.0F/2.0F)*powf(R,
                 2)*(eta[2]*powf(sinf(q[3]), 2)*cosf(q[2])*powf(cosf(q[4]), 2) -
                     eta[2]*sinf(q[3])*sinf(q[2])*sinf(q[4])*cosf(q[4]) -
                     eta[4]*sinf(q[3])*sinf(q[6])*cosf(q[2])*cosf(q[4])*cosf(q[7]) +
                     eta[4]*sinf(q[6])*sinf(q[2])*sinf(q[4])*cosf(q[7]))*eta[2]/l +
             v[1])*(sinf(q[2])/(R*eta[4]*sinf(q[6])*sinf(q[7])) -
                 sinf(q[2])/(R*eta[2]*cosf(q[3])*cosf(q[4])*tanf(q[6])*tanf(q[7])) -
                 sinf(q[4])*cosf(q[2])*tanf(q[3])/(R*eta[2]*powf(cosf(q[4]), 2)*tanf(q[6])*tanf(q[7]))) +
             2*l*((1.0F/2.0F)*R*(eta[2]*sinf(q[4]) - eta[4]*sinf(q[7]))*eta[4]*cosf(q[6])*cosf(q[7])/l +
                     v[2])/(R*eta[4]*sinf(q[6])*sinf(q[7]));
    u[4] = v[4];
}
```

Listing A.6: Dynamic linearisation for *JPTD Model* with CSE

```
void full_2_JPTD_cse_dynamic_lin_u(float u[5], const float v[5], const float eta[5], const float q[9])
{
    float x0 = q[4];
    float x1 = cosf(x0);
    float x2 = 1.0F/x1;
    float x3 = q[3];
    float x4 = tanf(x3);
    float x5 = v[3]*x4;
    float x6 = q[2];
    float x7 = sinf(x6);
    float x8 = 1.0F/cosf(x3);
    float x9 = x7*x8;
    float x10 = cosf(x6);
    float x11 = tanf(x0);
    float x12 = x11*x4;
    float x13 = sinf(x0);
    float x14 = q[6];
    float x15 = sinf(x14);
    float x16 = eta[4];
    float x17 = q[7];
    float x18 = x16*cosf(x17);
    float x19 = x15*x18;
    float x20 = x19*x7;
    float x21 = eta[2];
    float x22 = powf(x1, 2);
    float x23 = sinf(x3);
    float x24 = x21*x22*powf(x23, 2);
    float x25 = x13*x21;
    float x26 = x1*x23;
    float x27 = x25*x26;
    float x28 = 1.0F/l;
    float x29 = powf(R, 2)*x21*x28;
    float x30 = 1.0F/R;
    float x31 = (1.0F/2.0F)*x30;
    float x32 = x31*(2*v[1] + x29*(-x10*x19*x26 + x10*x24 + x13*x20 - x27*x7));
    float x33 = x10*x8;
    float x34 = x10*x13;
    float x35 = x31*(-2*v[0] + x29*(x10*x27 - x19*x34 - x20*x26 + x24*x7));
    float x36 = 1.0F/x21;
    float x37 = x2*x36;
    float x38 = 1.0F/x16;
    float x39 = 1.0F/tanf(x17);
    float x40 = x39/tanf(x14);
    float x41 = x36*x40/x22;
    float x42 = sinf(x17);
    float x43 = x38/(x15*x42);
    float x44 = x37*x40;
    float x45 = x4*x41;

    u[0] = -x37*(x2*x5 + x32*(x10*x12 + x9) + x35*(x12*x7 - x33));
    u[1] = -x36*(v[3]*x11 + x10*x2*x32 + x2*x35*x7);
    u[2] = v[3];
    u[3] = l*x30*x43*(R*x18*x28*(-x16*x42 + x25)*cosf(x14) + 2*v[2]) + v[4]*x38*x39 - x32*(x34*x45 - x43*x7 +
            x44*x9) - x35*(x10*x43 + x13*x45*x7 - x33*x44) - x41*x5;
    u[4] = v[4];
}
```

Listing A.7: Dynamic linearisation for *Simplfied Model* without CSE

```
void simplified_dyn2_dynamic_lin_u(float u[5], const float v[5], const float eta[5], const float q[9])
{
    u[0] = -(v[0] + (1.0F/2.0F)*powf(eta[1], 2)*powf(q[7], 2)*sinf(q[2] + q[3])*sinf(q[3] -
                q[5])/(l*sinf(q[5])))*sinf(q[2] + q[3])/(eta[1]*q[7]) + (v[1] - 1.0F/2.0F*powf(eta[1],
                    2)*powf(q[7], 2)*sinf(q[3] - q[5])*cosf(q[2] + q[3])/(l*sinf(q[5])))*cosf(q[2] +
                    q[3])/(eta[1]*q[7]);
    u[1] = v[3];
    u[2] = v[2];
    u[3] = -v[3]*q[7]/eta[1] + (v[0] + (1.0F/2.0F)*powf(eta[1], 2)*powf(q[7], 2)*sinf(q[2] + q[3])*sinf(q[3] -
                q[5])/(l*sinf(q[5])))*cosf(q[2] + q[3])/eta[1] + (v[1] - 1.0F/2.0F*powf(eta[1], 2)*powf(q[7],
                    2)*sinf(q[3] - q[5])*cosf(q[2] + q[3])/(l*sinf(q[5])))*sinf(q[2] + q[3])/eta[1];
    u[4] = -(v[0] + (1.0F/2.0F)*powf(eta[1], 2)*powf(q[7], 2)*sinf(q[2] + q[3])*sinf(q[3] -
                q[5])/(l*sinf(q[5])))*q[8]*sinf(q[2])/(eta[1]*q[7]*sinf(q[3])) + (v[1] -
                1.0F/2.0F*powf(eta[1], 2)*powf(q[7], 2)*sinf(q[3] - q[5])*cosf(q[2] +
                    q[3])/(l*sinf(q[5])))*q[8]*cosf(q[2])/(eta[1]*q[7]*sinf(q[3])) - (v[4] +
                    eta[1]*eta[2]*q[7]*sinf(q[3])*cosf(q[5])/(q[8]*powf(sinf(q[5]), 2)))*powf(q[8],
                    2)*sinf(q[5])/(eta[1]*q[7]*sinf(q[3]));
}
```

Listing A.8: Dynamic linearisation for *Simplfied Model* with CSE

```
void simplified_dyn2_cse_dynamic_lin_u(float u[5], const float v[5], const float eta[5], const float q[9])
{
    float x0 = q[2];
    float x1 = q[3];
    float x2 = x0 + x1;
    float x3 = sinf(x2);
    float x4 = eta[1];
    float x5 = q[7];
    float x6 = q[5];
    float x7 = sinf(x6);
    float x8 = powf(x4, 2)*powf(x5, 2)*sinf(x1 - x6)/(l*x7);
    float x9 = 2*v[0] + x3*x8;
    float x10 = cosf(x2);
    float x11 = 2*v[1] - x10*x8;
    float x12 = 1.0F/x4;
    float x13 = x12/x5;
    float x14 = (1.0F/2.0F)*x9;
    float x15 = (1.0F/2.0F)*x11;
    float x16 = q[8];
    float x17 = sinf(x1);

    u[0] = (1.0F/2.0F)*x13*(x10*x11 - x3*x9);
    u[1] = v[3];
    u[2] = v[2];
    u[3] = x12*(-v[3]*x5 + x10*x14 + x15*x3);
    u[4] = x13*x16*(-x14*sinf(x0) + x15*cosf(x0) - x16*x7*(v[4] + x17*x4*x5*eta[2]*cosf(x6)/(x16*powf(x7,
                        2))))/x17;
}
```

Listing A.9: Conversions between *Simplfied Model* and *Full Models* without CSE

```
void simplified_full_to_simplified(float q_simp[9], const float q_full[9])
{
    q_simp[0] = q_simp[0];
    q_simp[1] = q_simp[1];
    q_simp[2] = q_simp[2];
    q_simp[3] = atan2f(sinf(q_full[4]), sinf(q_full[3])*cosf(q_full[4]));
    q_simp[4] = q_full[5];
    q_simp[5] = atan2f(sinf(q_full[7]), sinf(q_full[6])*cosf(q_full[7]));
    q_simp[6] = q_full[8];
    q_simp[7] = R*sqrtf((powf(sinf(q_full[4]), 2) - 1)*powf(cosf(q_full[3]), 2) + 1);
    q_simp[8] = R*sqrtf((powf(sinf(q_full[7]), 2) - 1)*powf(cosf(q_full[6]), 2) + 1);
}
void simplified_simplified_to_full(float q_full[9], const float q_simp[9])
{
    q_full[0] = q_simp[0];
    q_full[1] = q_simp[1];
    q_full[2] = q_simp[2];
    q_full[3] = ((fmodf(q_simp[3], M_PI) > (3.0F/2.0F)*M_PI || fmodf(q_simp[3], M_PI) < M_PI_2) ?
            (1) : (-1)
            )*acosf(sqrtf(-powf(R, 2) + powf(q_simp[7], 2))*sqrtf(powf(tanf(q_simp[3]), 2) + 1)/sqrtf(-powf(R,
            2)*powf(tanf(q_simp[3]), 2) - powf(R, 2) + powf(q_simp[7], 2)*powf(tanf(q_simp[3]), 2)));
    q_full[4] = ((fmodf(q_simp[3], M_PI) > (3.0F/2.0F)*M_PI || fmodf(q_simp[3], M_PI) < M_PI_2) ?
            (1) : (-1)
            )*asinf(q_simp[7]*tanf(q_simp[3])/(R*sqrtf(powf(tanf(q_simp[3]), 2) + 1)));
    q_full[5] = q_simp[4];
    q_full[6] = ((fmodf(q_simp[5], M_PI) > (3.0F/2.0F)*M_PI || fmodf(q_simp[5], M_PI) < M_PI_2) ?
            (1) : (-1)
            )*acosf(sqrtf(-powf(R, 2) + powf(q_simp[8], 2))*sqrtf(powf(tanf(q_simp[5]), 2) + 1)/sqrtf(-powf(R,
            2)*powf(tanf(q_simp[5]), 2) - powf(R, 2) + powf(q_simp[8], 2)*powf(tanf(q_simp[5]), 2)));
    q_full[7] = ((fmodf(q_simp[5], M_PI) > (3.0F/2.0F)*M_PI || fmodf(q_simp[5], M_PI) < M_PI_2) ?
            (1) : (-1)
            )*asinf(q_simp[8]*tanf(q_simp[5])/(R*sqrtf(powf(tanf(q_simp[5]), 2) + 1)));
    q_full[8] = q_simp[6];
}
```

Listing A.10: Conversions between *Simplfied Model* and *Full Models* with CSE

```
void simplified_cse_full_to_simplified(float q_simp[9], const float q_full[9])
{
    float x0 = q_full[4];
    float x1 = sinf(x0);
    float x2 = q_full[3];
    float x3 = q_full[7];
    float x4 = sinf(x3);
    float x5 = q_full[6];

    q_simp[0] = q_simp[0];
    q_simp[1] = q_simp[1];
    q_simp[2] = q_simp[2];
    q_simp[3] = atan2f(x1, sinf(x2)*cosf(x0));
    q_simp[4] = q_full[5];
    q_simp[5] = atan2f(x4, sinf(x5)*cosf(x3));
    q_simp[6] = q_full[8];
    q_simp[7] = R*sqrtf((powf(x1, 2) - 1)*powf(cosf(x2), 2) + 1);
    q_simp[8] = R*sqrtf((powf(x4, 2) - 1)*powf(cosf(x5), 2) + 1);
}
void simplified_cse_simplified_to_full(float q_full[9], const float q_simp[9])
{
    float x0 = q_simp[3];
    float x1 = fmodf(x0, M_PI);
    float x2 = (3.0F/2.0F)*M_PI;
    float x3 = M_PI_2;
    float x4 = ((x1 > x2 || x1 < x3) ? (1) : (-1));
    float x5 = tanf(x0);
    float x6 = powf(x5, 2);
    float x7 = sqrtf(x6 + 1);
    float x8 = q_simp[7];
    float x9 = powf(x8, 2);
    float x10 = powf(R, 2);
    float x11 = -x10;
    float x12 = 1.0F/R;
    float x13 = q_simp[5];
    float x14 = fmodf(x13, M_PI);
    float x15 = ((x14 > x2 || x14 < x3) ? (1) : (-1));
    float x16 = tanf(x13);
    float x17 = powf(x16, 2);
    float x18 = sqrtf(x17 + 1);
    float x19 = q_simp[8];
    float x20 = powf(x19, 2);

    q_full[0] = q_simp[0];
    q_full[1] = q_simp[1];
    q_full[2] = q_simp[2];
    q_full[3] = x4*acosf(x7*sqrtf(x11 + x9)/sqrtf(-x10*x6 + x11 + x6*x9));
    q_full[4] = x4*asinf(x12*x5*x8/x7);
    q_full[5] = q_simp[4];
    q_full[6] = x15*acosf(x18*sqrtf(x11 + x20)/sqrtf(-x10*x17 + x11 + x17*x20));
    q_full[7] = x15*asinf(x12*x16*x19/x18);
    q_full[8] = q_simp[6];
}
```