6

A Framework for Research and Prototyping in Robotics: From Ideas to Software and Hardware Development

Konstantinos Machairas, Spyridon Garyfallidis, Iosif S. Paraskevas and Evangelos Papadopoulos

Department of Mechanical Engineering, National Technical University of Athens, 15780 Athens, Greece

Abstract

Development and prototyping of robotic systems requires the involvement of many people and many hours of design, development, and cooperation; significant time and effort overhead is required for evaluating conceptual ideas in design, control and technology, and for bringing them fast into reality for testing. Based on the important advances of the last decade in hardware and software, a simple and low-cost framework and its underlying ideas are presented, with steps that aim at accelerating robotics research work in academia and industry. The framework's functionality is validated and illustrated by two application examples concerning the control systems of a single-legged hopping robot and an instrumented treadmill. The software required to conduct the same experiments is provided, with the intention to help the reader reuse it in similar applications.

6.1 Introduction

Cyber Physical Systems (CPSs) are electromechanical and mechatronic systems interacting through physical actions with their environment (e.g., motorized motion), controlled by interconnected embedded computers. The field of CPSs builds on embedded systems theory with a great focus on how

subsystems communicate to form complex systems with more capabilities. CPS research aims at a deep understanding of small embedded components, and at their best possible integration into a whole system with increased functionality [1]. The fields of dynamics, mechanical design, electrical/electronic design, software design, control, and networking, assume equally important roles and compose the multidisciplinary nature of the field of CPSs [2]. Apparent is the relation to robotic systems theory, which strongly builds on advances in these same fields, and also to the recent developments of the Industrial Internet of Things (IIoT) [3] and Industry 4.0 [4], and their promise for smarter industrial procedures. During the last decades, CPSs have gained attention as an important domain, bridging multiple theoretical and applied technological fields toward promising application goals ranging from smart devices to smart cities, and aiming to have strong impacts on most sectors of human life (e.g., health, transportation, etc.) [5]. This chapter focuses on CPS design, control, simulation and development.

In regard to the recent advances in robotics, an increasing tendency of the community is witnessed to provide low cost and easy to implement solutions concerning the theoretical analysis of a system (modeling, design, simulation, and control), as well as its development (embedded hardware and software). Already, important advances have led to big strides in software (e.g., the ubiquitous use of the Robotics Operating System or ROS [6]), and in hardware (e.g., the constantly increasing number of embedded single board computers with high computational power and low cost [7]). However, the robotics community still lacks a complete framework involving all the necessary stages that lead from ideas to prototype development and testing. There is a lack of standard procedures to help research teams create and share design and development tools and build upon existing work to minimize time and maximize research quality. The challenge here is to connect the pieces from diverse science and technology fields toward coordinated steps in the field of CPSs.

In this chapter, an attempt is made to provide new tools and also combine existing ones in software and hardware with the goal to create a complete framework to aid researchers bring conceptual ideas into reality in minimum time and at minimum cost. To this end, the focus is on widely adopted tools and techniques. Also all necessary material for anyone to reuse it in a similar or modified manner is provided. First, the modeling and simulation stages are addressed for robots with multiple degrees of freedom (dof) subject to frequent collisions with the environment – a typical class

of problems – and template simulation software is provided. Second, simple steps toward the design and development of prototypes are proposed, based on the idea of easily interconnected, and low cost embedded computers. Finally, two application examples – mostly concerning motion control – are presented to exhibit the simulation and prototyping aspects of the proposed framework.

6.2 A Framework for Simulation and Prototyping

The proposed simulation and prototyping framework provides solutions for the stages of modeling, simulation, control, and system development. In this section, each stage of the procedure is properly described, and simple software and hardware solutions are provided for a class of problems commonly addressed in robotics, namely motion control and interaction with the environment.

The first stage includes finding the most important unsolved problems and proposing concepts and ideas toward their solutions. In this scope, this stage mostly concerns ideas in the areas of design, control and technology. Next, simulation experiments must be conducted to provide the first proof of concept, hardware prototypes must be designed and built, and finally hardware experiments must validate the previously derived theoretical results.

6.2.1 Modeling, Dynamics, and Simulation

6.2.1.1 Dynamics derivation and simulation methods

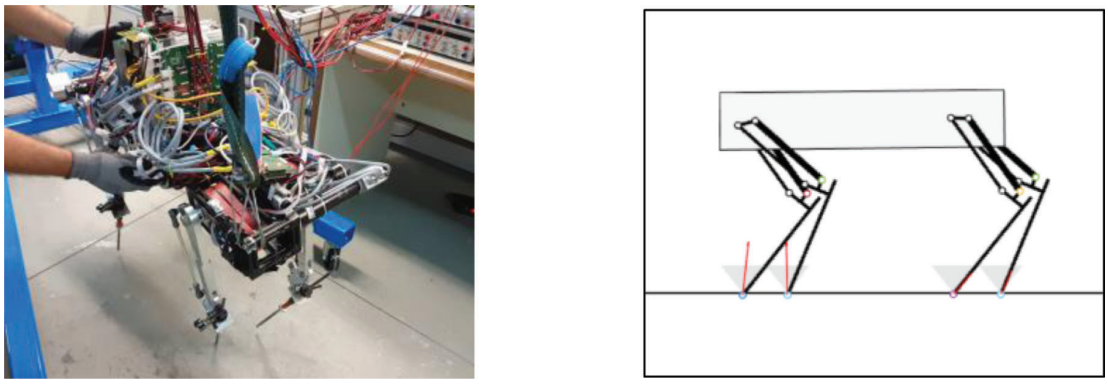
Before conducting hardware experiments, simulation experiments are conducted to first evaluate the theoretical idea or concept. To this end, a model of the system must be built, and then simulated. Two options are typically followed and both present their own advantages; deriving the respective system of differential equations and then solving it with numerical analysis software, such as Mathematica and Matlab, or use 3D simulation software packages such as Gazebo and Adams [8], for which a physical description with mass properties is only required.

In the first option, the models are kept simple but representative, to aid the dynamics derivation phase, to allow the understanding of each component's role in the equations, and also to be used in the design of model-based controllers. This approach provides a simulation environment, but also tools for design and control. The difficulties that arise concern the derivation of

the often intricate and multi-dimensional system of equations, the tedious programming, and the art of selecting the most important features to model.

Template Mathematica and Matlab files for deriving and solving the Equations of Motion (EoM) for a model of the quadruped robot Laelaps (Figure 6.1) – designed and built in the Control Systems Laboratory (CSL) of the Mechanical Engineering Department of NTUA – are provided as supplementary material for this chapter [9]. The EoM are derived in symbolic form using Mathematica, and then imported for integration and animation in Matlab; this approach was followed in designing and simulating the trotting controller proposed in [10]. The software is designed such as to be easily reconfigured for application in other systems of the same class, namely multibody dynamical systems subject to impacts with the environment. The reader is encouraged to employ the code as a tool for dynamics derivation and simulation in related applications.

In the second option, in which a 3D dynamics simulation software package is used, a detailed model can be built easily. The focus here lies on tuning the simulations so as to eliminate, if possible, the expected differences from the experiments. A comparison of the best known and most used 3D simulation packages for use in robotics can be found in [8], where Gazebo appears to be a prevalent solution in terms of cost, community support and ability for interconnection with other software like ROS, ubiquitously used for robot control [6]. This exact combination – Gazebo for modeling and simulation, and ROS for control – is proposed here for 3D simulations. Application examples are presented next to show the packages' ease of use, functionality, and compatibility with the real robot's firmware.



(a) (b)

Figure 6.1 (a) The quadruped robot Laelaps, designed and built at CSL. (b) The 2D model of Laelaps animated in the simulation environment built in Matlab.

6.2.1.2 Modeling the environment

Depending on the application, one has to model the environment with the proper level of detail. In robotics, this includes, among others, the modeling of walls, gravitational acceleration, terrain, aerodynamic or hydrodynamic forces, impacts, etc. The descriptions are such that follow simple or complex forms of physics equations: for example it may be enough to describe a wall as a static object; then, a mobile robot has to avoid certain values in the location of the object in order to avoid any interaction with the object. On the other hand, it is common to require a description of interactions, because the robot operates in a way that these interactions are unavoidable; the foot-terrain interaction is a perfect example which will be analyzed a bit further to give an insight on how to work on these cases.

The interest in this example is how to model impacts between a legged robot and terrain, using a method which shall describe the interaction between different materials, compliant (i.e., able to have deformations) and non-compliant, retaining a high level of fidelity. The answer to this question is important, because a well-established model is necessary for the accurate representation of impacts on simulations. In principle, the impacts can be modeled via three methods: the stereomechanical theory method, the Finite Element Method (FEM), and the compliant/viscoelastic approach [11]. Each method has its pros and cons but the use of the viscoelastic method seems more appropriate, as the impact between different materials can be described by lumped parameter models with suitable characteristics. There are various viscoelastic models in the literature with more prominent the Hunt–Crossley (HC) model, [12]; in fact the majority of the viscoelastic models use the HC model as a basis and this will be also the basis here. In HC, the interaction force is calculated by,

$$F_g(y_g, \dot{y}_g) = k_g \cdot y_g^n + b_g \cdot \dot{y}_g \cdot y_g^n \tag{6.1}$$

where k_g and b_g are the stiffness and damping coefficients respectively, n in the case of Hertzian non-adhesive contact is equal to 1.5, and y_g is the depth of interpenetration. In Figure 6.2(a) the shape of a typical HC impact is given. The area inside the curve is the non-recoverable energy dissipated during impact inside the materials due to mechanisms like internal vibrations and local plastic deformations.

However, the behavior of real materials is somehow different according to the experimental results in the literature. In Figure 6.2(b) the experimental results of the impact of a metallic sphere on a rigidly supported thin laminate

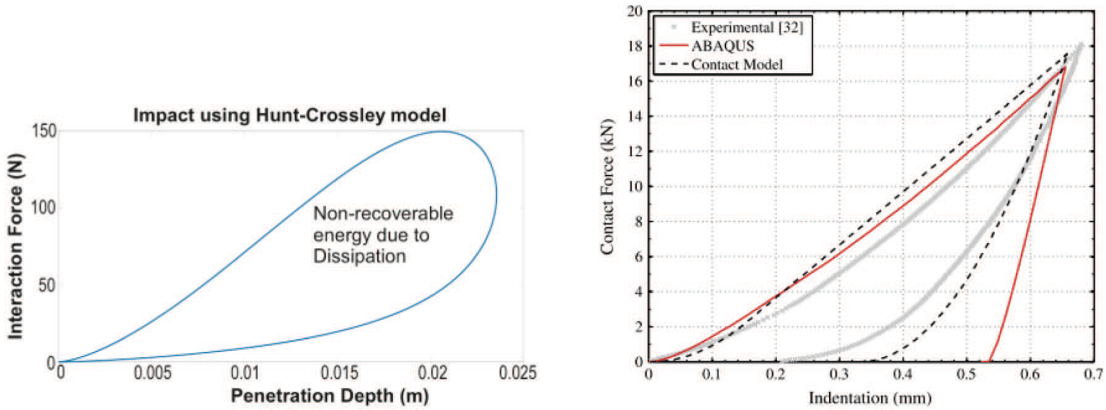


Figure 6.2 (a) Typical HC interaction force–penetration depth diagram. (b) Force–indentation response of a metallic sphere impacting a rigidly supported thin laminate.

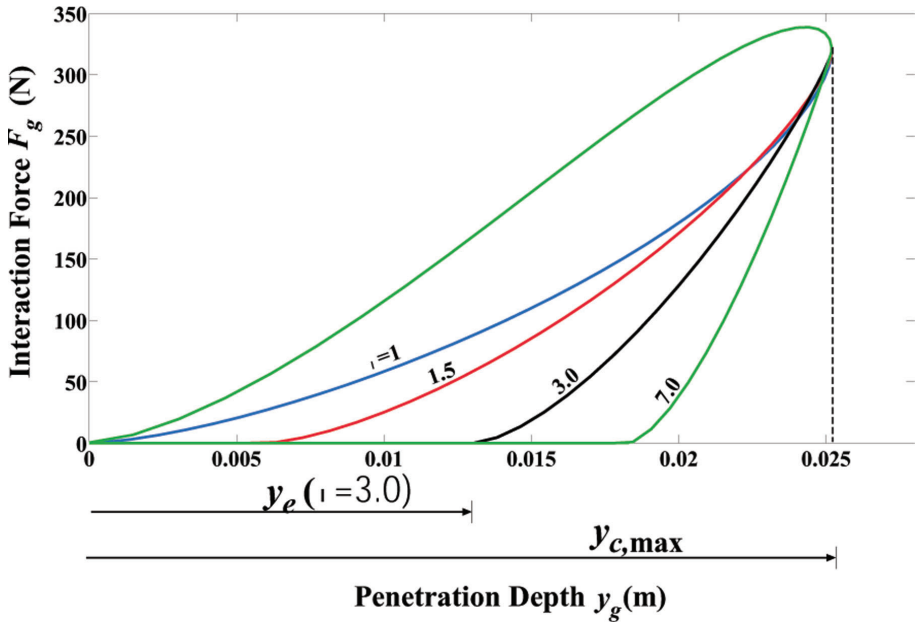


Figure 6.3 Impact curves for the proposed impact model (2) for various λ.

are shown. The qualitative similarity of the HC model is apparent; however, the HC model fails to predict the permanent deformation analytically [13].

In order to tackle the issues that other impact descriptions have, the authors have proposed a novel impact model which has viscoplastic characteristics. This model shows very good correlation with experimental results found in the literature and it can efficiently describe a large number of interactions that occur in robotics (in terrestrial applications and also in space). Briefly, the model calculates the interaction force F_g at impact instance i by,

$$F_g^i\left(y_g, \dot{y}_g\right) = \begin{cases} F_c^i = \left(\lambda_c^i k_g + b_g \dot{y}_g\right)\left(y_g - y_e^{i-1}\right)^n, & \dot{y}_g \geq 0 \\ F_r^i = \left(\lambda_r^i k_g + b_g \dot{y}_g\right)\left(y_g - y_e^i\right)^n, & \dot{y}_g < 0 \end{cases} \tag{6.2}$$

where subscript c stands for compression, r for restitution, y_e is the penetration depth, and the index i identifies the impact instance, while the term *Coefficient of Permanent Terrain Deformation* λ is proposed. The result of the model can be seen in Figure 6.3, however the interested reader should consult [14] for details.

6.2.2 System Development: Hardware and Software

6.2.2.1 Introduction

In this section, methods for fast prototyping of robotic systems of various structures and complexity levels are presented, which are scalable and can be implemented relatively easy. Since hardware experiments provide the final and most convincing proof of the claims made in the analysis and simulation phases, it is critical to be able to design, build, and test a prototype in a fast, low cost, and debuggable way. The focus here is on hardware and software development for robotic applications like legged robots, which include modules with low level read and write capabilities, e.g., microcontroller units (MCUs), controlled through a network. Such systems, also known as *Networked Control Systems* (NCSs) [15], are inextricably linked to CPSs and will be the main subject of the following paragraphs. An example of an NCS is shown in Figure 6.4, where sensor measurements that are used as feedback, and control signals that are used as controlled reactions, are exchanged via the network.

NCSs consist of distributed nodes and are typically controlled by one or more *Single Board Computers* (SBCs) running an operating system, or even a higher level meta-operating system like ROS. The number of nodes

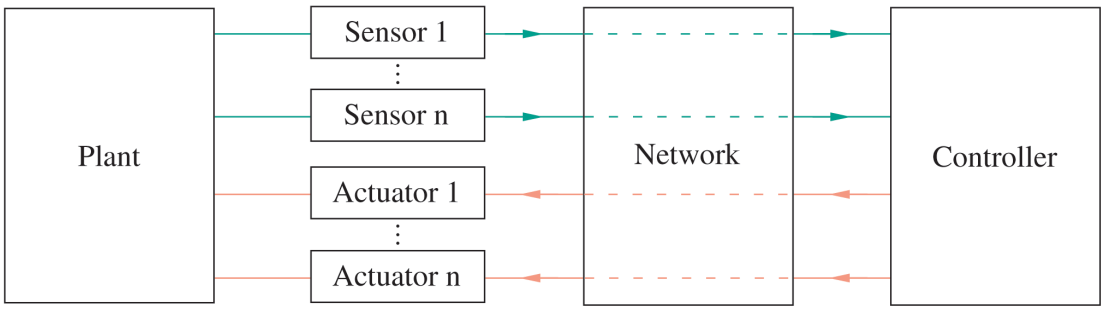


Figure 6.4 The structure of a typical Networked Control System (NCS).

comprising an NCS is indicative of the system's complexity. Common applications for a node include controlling a DC motor, interfacing with sensors like Inertial Measurement Units (IMUs), LIDAR, or force sensors, and more. These low level devices come with specifications for communication, with common interfaces including Quadrature Encoder Interfaces (QEIs), Pulse Width Modulation (PWM) modules, serial interfaces such as Serial Peripheral Interfaces (SPI), etc. Thus, in the design phase of an NCS, suitable embedded computers must be selected such as to first satisfy these low level requirements. Note that some I/O capabilities like QEIs, or Digital to Analog Converters (DAC) are only found in application specific MCUs, whereas others like SPI are ubiquitous. Besides communicating with the devices, the embedded computers must be able to connect to the network, through which they will communicate with the other nodes. This network can have thousands of nodes in the case of a factory or can be much smaller in the case of a small mobile robot. Therefore, depending on the network type, another requirement is added to the process of selecting the appropriate MCUs for each case. For instance, if it is for a node to be connected to a CAN [16] or an EtherCAT network [17], it must have the respective specifications.

Networking of embedded devices has been a subject of meticulous studies for many years, with industry showing early the way by introducing numerous communication protocols for use in factories, vehicles, medical devices, etc. [18]. Interestingly, there is not yet a universally adopted protocol for industrial communications, as there is, e.g., for the Internet. As networks and related technology fields advance, many new communication technologies emerge, many old ones become obsolete, and the future goals get higher in terms of speed, safety and determinism.

The following paragraphs gradually introduce and discuss the fundamental concepts behind industrial communications, aiming at setting the basis on which the proposed methods will be presented. The main focus is on the aspects of communication speed and determinism, which are critical for real-time closed loop applications. Lastly, the hardware and software aspects of the proposed NCS are described.

6.2.2.2 The automation pyramid

The topics discussed are strongly connected with the old *automation pyramid* idea [19], which separates an industrial network in different logical and topological hierarchical layers. The network types in the pyramid differ in size, complexity, requirements and purpose. On the top, the *Global Area Networks* (GANs) are met, able to cover even intercontinental distances, next

are the *Wide Area Networks* (WANs) followed by the *Local Area Networks* (LANs), and below are the *Field Area Networks* (FANs), (Figure 6.5). During the last decades, important advances occurred in both LANs and FANs, aiming at serving different communication purposes; high bandwidth for large data packets delivery for the former, compared to real-time delivery of small data packets for the latter. However, this difference gradually lost importance, as Ethernet started occupying an increasingly wider range in the pyramid, penetrating every level and replacing almost all of the older variants.

Here the interest lies in the lowest levels of the automation hierarchy, where high accuracy embedded control takes place. There, depending on the number of nodes connected to the network, complexity rises and the need of systematic approaches concerning the communication system emerges. Regarding communication at this so called field level, the seminal works on *Fieldbus* technologies, which have been advancing for the last 30 years, have already set a rigid basis under the idea of connecting multiple devices via a bus-like shared medium [20]. This *Fieldbus* revolution pushed industrial communications to a high level, with applications spreading across all technology domains under various standards. Interestingly though, a second promising revolution has recently begun, building upon the widespread *Ethernet* technology [21]; *Ethernet*-based control systems will be rigorously studied herein.

6.2.2.3 The OSI model and Media Access Control (MAC) methods

Regarding the structure of an industrial network, studies refer to the physical and the logical topology as the two key characteristics [22]. The physical topology describes the installation of the nodes (in a star, ring or other formation), the cabling and the hardware required, while the logical topology – not necessarily similar to the physical one – defines the protocol used by the signals to transfer data. Of great importance in networks history was the establishment of the *Open System Interconnection* (OSI) model, as a seven-layer open reference model describing the required hardware and software components for connecting all the – incompatible in the past – types of networks [22]. A clarification needed is that the numerous standards occupying the different layers are not considered parts of the OSI model. According to the model, in a typical communication channel, the transmitting side encodes each data packet following the steps from layer 7 down to layer 1 before sending the package to the receiving side, where the inverse process

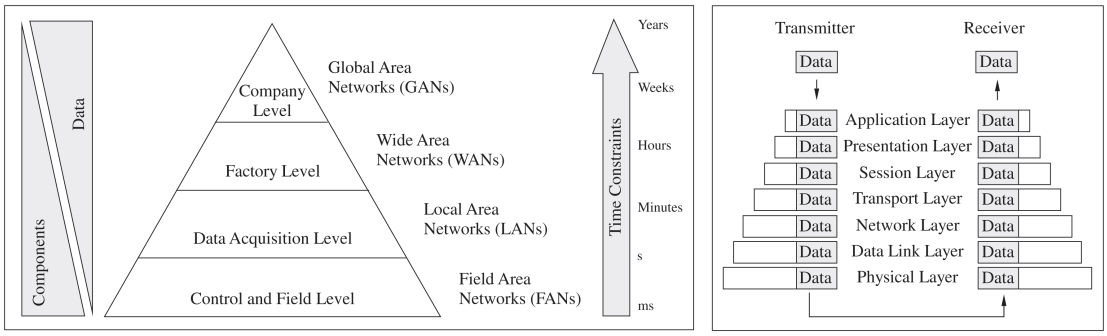


Figure 6.5 (a) The automation pyramid and (b) the OSI model.

takes place. Each layer, in the transmitter prepares data to be read by the peer layer on the receiving side, and then transmits the data to the lower layer to do the same until the packet is finally sent out from the physical layer (Figure 6.5(b)).

The *Media Access Control* (MAC) methods occupy a part of the data link layer of the OSI model, as mechanisms to access the network and manage the bandwidth for achieving the desired communication characteristics. The MAC sub layer has a major role in defining the determinism of the communication; depending on the application, the method may favor large data packet transfer with high and unbounded delays, or small data packets with low and bounded delays. Known MAC methods include *Frequency Division Multiple Access* (FDMA), *Code Division Multiple Access* (CDMA), *Space Division Multiple Access* (SDMA), and *Time Division Multiple Access* (TDMA) [20]. The applications studied here typically use the latter, with which the nodes use the medium sequentially. Referring to TDMA methods, the available sub-strategies include *Polling*, *Token Passing*, *Time-Slot-Based*, and *Random Access* methods [20]. Ethernet adopts the *Random Access* strategy, and thus it is of our main interest in this chapter.

Random Access methods include various *Carrier Sense Multiple Access* (CSMA) methods; some are modified for collision avoidance (CSMA/CA), as for example in CAN, and others for collision detection (CSMA/CD), as in Ethernet's original version with shared medium [23]. Focusing on CSMA/CD, collisions are immediately detected by the sending nodes, which monitor the bus while sending. After collision detection, the nodes abort the data transfer and wait for a random time before trying again, clearly showing a nondeterministic behavior, unsuitable for control applications. However, with the introduction of switched *full-duplex* Ethernet, collisions can now be avoided as described next.

6.2.2.4 Networked Control System design

When designing an NCS, there is no rule of thumb that one should always follow. Taking into account the available software and hardware and their costs, merits and demerits come to the surface and point toward the final solution. In this chapter, we propose a simple but powerful control network structure based on Ethernet and the *User Datagram Protocol* (UDP) [22]. The Ethernet standard is the most common way to connect devices in a local area network, and it occupies layers 1 and 2 of the OSI model. So far, Ethernet was considered mostly for the upper levels of the automation pyramid, but recently it has gained ground toward the lower field levels with industrial standards ensuring high-speed, reliability, and determinism [23]. Moreover, besides the usually costly industrial solutions, simple and cost-effective nonindustrial UDP implementations have been also shown for real-time embedded systems [24–26].

In general, Ethernet technology presents many advantages for use in CPSs; here, we sum up some critical aspects. Firstly, it is a fast, inexpensive technology that can suit many purposes. It can transfer data ranging from short messages to big files and over long distances, it has minimum requirements in hardware and software design – since related hardware modules and software packages are widely available – and it can take advantage of numerous higher-level protocols such as IP and UDP, or other industrial variants like EtherCAT [27]. Finally, most computers have Ethernet support built in, which favors the direct interface of an Ethernet-based CPS with a personal computer.

Throughout the next paragraphs, a generic case is analyzed to show how an NCS can be built using the herein discussed toolsets. Consider a common application, where n DC motors are connected to n MCUs and are controlled through a switched Ethernet network by a computer running ROS, Figure 6.6. To analyze the proposed NCS, the important aspects of Ethernet and UDP in terms of determinism and system performance are examined.

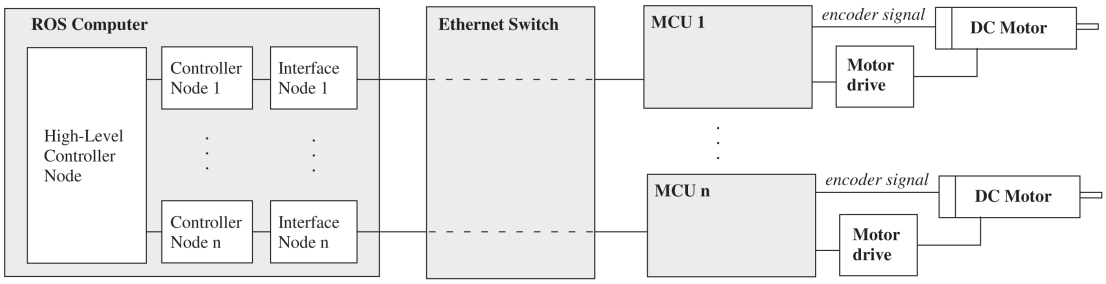


Figure 6.6 The architecture of the proposed NCS.

Also, software design methods for the MCU nodes and the ROS nodes are rigorously discussed.

6.2.2.5 Switched Ethernet and determinism

Before switches, Ethernet networks used hubs. In a hub-based network, packets received by the hub were broadcasted to all ports, raising the risk of message collisions, and packet retransmissions after randomly long waiting times, as defined by CSMA/CD. On the other hand, a switch is a device that can learn the network's topology, divide it into different collision domains, and send packets only to their destination ports. The main switch types are *store and forward*, which examine the whole packet before transmitting, and *cut through*, which immediately forward the packet to its destination port. Store and forward is slower, but it is the most common type of switching.

Regarding sending and receiving packets through the same wire, another risk for collisions emerges even for switched Ethernet. In the early *half-duplex* Ethernet, where a node could not send and receive data at the same time, emerging collisions were addressed with CSMA/CD resulting again in a nondeterministic communication type. However, when *full-duplex* Ethernet was standardized, every node was provided with a unique collision domain allowing for sending and receiving simultaneously, while doubling the available bandwidth [28].

Conclusively, switched Ethernet can assume a deterministic nature, since collisions can be prevented, and thus no random waiting times are required. Yet, a last point requires the designer's attention to ensure the desired real-time character. In case a switch port receives a large number of packets, the port's buffer may overflow and lead to unexpected delays or data loss [29]. By default, switches use first-in-first-out (FIFO) queues. That said, to avoid the risk of data loss and to ensure a high level of determinism (i.e., bounded latency), a control network should be designed such that it would not need large buffers in nominal operation. To this end, congested segments must be avoided early in the design phase. Depending on the hardware components used and the network architecture, the proper data rate for each communication path must be defined in software. For instance, say the network is badly designed, and network traffic for the nominal case is intense at several nodes finally leading to always full buffers. Apart from packet dropping, there is also unwanted latency introduced to the system, equal to the time required to empty the respective buffers. Along these lines, the software developer must carefully take into account all the limitations regarding hardware data rates to achieve optimal performance.

6.2.2.6 Quality of Service (QoS)

The QoS parameters refer to measures of how well a network performs according to criteria of timeliness, reliability and other [30]. The bottlenecks concerning these parameters must be identified and taken under consideration in the design phase. To this end, the most important QoS parameters are shortly presented here.

- **Bandwidth**: a measure of the maximum amount of data bits that can pass in a given interval between two network nodes – measured in bps (bits per second).
- **Throughput**: the actual rate that the data are transferred.
- **Delay**: a measure of how long it takes for a unit of data to be transferred from the source node to the receiver node.
- **Jitter**: the variation in packet delay.

6.2.2.7 Latency in switched Ethernet

Adding to the above, **determinism** can be considered a QoS parameter of great significance when referring to NCSs. It refers to the unbounded delays that may occur in any layer of the OSI model. An analysis on the delays introduced in the various stages of Ethernet communication is useful here. The total delay for a communication channel is the main indicator for determinism and can be calculated as follows based on the analyses conducted in [23] and [29]:

$$T_{\text{delay}} = T_{\text{pre}} + T_{\text{wait}} + T_{\text{frame}} + T_{\text{prop}} + T_{\text{switch}} + T_{\text{post}} \tag{6.3}$$

with the time components defined as follows.

Preprocessing and Postprocessing times: T_{pre} is the time required for the sender to encode the data for sending over the network, and T_{post} is the time required for the receiver to decode the received data. These depend on the devices and they can be the prevalent cause of delay.

Waiting time: T_{wait} is the waiting before transmitting time in case the network medium is unavailable. It depends on the MAC strategy followed and the network traffic.

Frame time: T_{frame} is the time required to send a packet, and can be calculated as:

$$T_{\text{frame}} = (N_{\text{data}} + N_{\text{ovhd}} + N_{\text{pad}})8T_{\text{bit}} \tag{6.4}$$

where N_{data} is the data size in bytes, N_{ovhd} is the number of bytes used as overhead, N_{pad} the number of bytes required to reach the minimum frame

size, and T_{bit} is the time to transmit 1 bit, calculated as the inverse of the data rate (e.g., $T_{\text{bit}} = 10$ ns for 100 Mb/s Ethernet).

Propagation time: T_{prop} is the time required to propagate a message between two devices. It is defined by the velocity factor (VF) of the transmission medium, which is the ratio of the speed of a wavefront passing through the medium, to the speed of light in a vacuum [31]. Depending on the insulating material, typical VF values lie between 0.4 and 0.7, with 0.7 corresponding to approximately 2.1×10^8 m/s. T_{prop} is less than 1 µs for distances below 100 m for most networks.

Switch delay: is the time a frame is delayed at the switch, and can be calculated as $T_{\text{switch}} = T_{\text{mux}} + T_{\text{queue}}$, where T_{mux} is the multiplexing delay, after which the switch starts to transmit the frame once it is received, and T_{queue} is the time the frame waits in the switch queue plus the time required to transmit it [29].

6.2.2.8 Message exchange using Ethernet, IP, and UDP

A computer sending a UDP message, first places the message in a UDP datagram consisting of a UDP header followed by the data payload, and then places the datagram in the data area of an IP datagram. The IP address does not contain information about the physical location of the destination, and therefore the datagram is placed in an Ethernet frame that contains this kind of information. Finally, an Ethernet driver sends the packet on the network. Conversely, on the receiving side, the Ethernet layer passes the IP datagram to the IP layer, which removes the IP header and passes the data included in the UDP datagram to the port specified in the datagram's header – Figure 6.7 for a visualization of the procedure.

At this point, a detailed description of the Ethernet frame and the resulting message is needed to better understand the basics of Ethernet communication. Referring to Figure 6.7, the *Preamble* field (PRE) consists of seven bytes of the form 10101010, and is used for bit synchronization. The *Start Frame Delimiter* (SFD) is the byte 10101011, which indicates the start of a frame. The *source* and *destination* MAC *(Media Access Control)* addresses (SA and DA) are the physical hardware addresses consisting of 6 bytes each. The *Ethertype* indicates the sort of data contained in the frame; for an IP datagram, the field would contain the value 0x0800. The *Data* field contains a message of size ranging from 46 to 1500 bytes, including information for IP, UDP or other. Note that if the data size is less than 46 bytes, the remaining bits are padded as zeros. Finally, the *Frame Check Sequence* (FCS) is used to detect errors in a received frame, and the *Interpacket gap* (IPG) contains 12 bytes

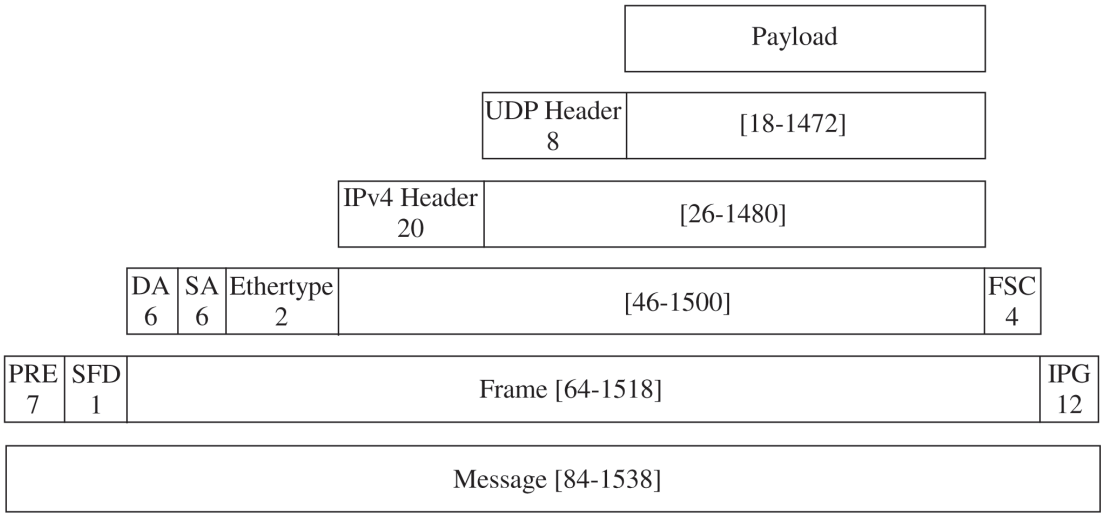


Figure 6.7 The UDP/IP stack.

to cause the required pause between the network frames. Based on these, the size of an Ethernet message always lies between 84 and 1538 bytes.

Details for the path of the data through the *Network*, the *Transport* and the *Application* layers of the proposed architecture are given next.

6.2.2.9 Network layer: The Internet Protocol

The *Internet Protocol* (IP) helps the data packet find the way to its destination on the Internet. Several communications also use IP in local networks to employ its companion protocols, TCP and UDP. It is a connectionless and unreliable protocol, since it doesn't provide flow control and error checking of the payload. The structure of an IP datagram and its header is well described in Kurose and Keith [22]. A protocol field is included in the datagram so the IP layer will know where to pass the received data, Figure 6.7. For instance, for UDP, decimal 17 is used.

6.2.2.10 Transport layer: The User Datagram Protocol

Both UDP (*User Datagram Protocol*) and TCP (*Transmission Control Protocol*) communications are established between logical endpoints called sockets and existing only in software. A socket is defined by a port number and an IP address. The port numbers identify the sending and receiving processes running on the communicating devices. Although in an Ethernet frame, the address fields already identify the communicating interfaces, UDP and TCP precisely specify the source and destination nodes by naming the respective ports. Here we focus on UDP, which only adds port addressing and optional error detection to the message being sent. Unlike TCP, it is a connectionless

protocol, meaning that a computer can send a message without establishing if the target computer is available on the network. These make it unreliable but simpler to implement and faster. UDP can also send a message to multiple destinations at once, while TCP cannot. Those said, and based on the considerations made about the protocol's determinism and reliability, it is considered more suitable than TCP for the discussed control applications.

6.2.2.11 Application layer

The data payload contained in the UDP datagram (Figure 6.7) must also follow a protocol to let the receiving application know what to do with the incoming data. An application may use a standard protocol such as the hypertext transfer protocol (HTTP) for requesting and sending Web pages, the file transfer protocol (FTP) for transferring files, or the simple mail transfer protocol (SMTP) for exchanging e-mail messages. Other applications however may use simpler custom protocols, e.g. in an embedded system like the one shown in Figure 6.6, an application may periodically receive sensor measurements and use the data to control motors, relays, or other circuits.

Before further discussing the proposed software structure at the application layer, putting the possible communication types into categories will help define the nature of the proposed NCS and understand the logic behind the design. A first categorization can be made into *cyclic* and *acyclic* communication types. Connectionless services – like the ones used here – are typically used for cyclic data exchange in the sense that lost packages are not being sent again delayed and outdated, favoring new data to be sent. In practice, buffers are used in a FIFO logic at all nodes to store the most recent data. On the contrary, acyclic communications typically use protocols with reception acknowledgement and packet retransmission mechanisms. A second categorization would be into *time-triggered* and *event-triggered* communication types, with the former mostly used for periodic data exchange, as in the presented case. Finally, regarding the way the central control node communicates with the distributed slave nodes, the communication types can be based on the *client–server* or the *publisher-subscriber* paradigms [20]. For instance, the processes (nodes) in a ROS system typically communicate using a publisher-subscriber type of communication.

6.2.2.12 Software design for the MCU node

On the MCU side, the software must handle the signal level communications, and also set up the node's connection to the network. Specifically in the motor control case, the MCU must read the signals from the encoder attached to the motor, and send a PWM signal to the motor drive. Here, instead of closing the

loop on the MCU, it is preferred that the loop is closed on the ROS computer in order to use its rich toolsets for control. To this end, the MCU must send the encoder measurements via UDP to the ROS computer to calculate the control output and send it back to the MCU also via UDP.

Along these lines, a suitable MCU is selected such as to have motor control features and Internet connection capabilities. Interestingly, a very limited number of ready-to test and low cost development boards with such specifications are available in the market. The Tiva$^{\mathrm{TM}}$ C Series TM4C1294 Connected LaunchPad [32] by Texas Instruments is a good candidate, since it features an 120MHz 32-bit ARM Cortex-M4 CPU, Ethernet connectivity and a QEI module, at the price of $20. Also provided by Texas Instruments is the TivaWare library, which significantly accelerates the software development. Those said and without loss of generality in the design process, TM4C1294 is used in the applications presented herein.

In the *n*-motor control case shown in Figure 6.6, *n* TM4C1294 boards are used. Encoder measurements are sampled from the boards and sent with a desired frequency to the ROS computer via UDP to calculate and send back the corresponding control outputs. The choice of the sampling frequency is of great importance to achieve the desired system performance; a too slow sampling rate will give low resolution and a slower control loop, while a too fast sampling rate will lead to a saturated network and data loss [23]. In terms of software design, a timer is used to strictly define the sampling and the transmission rate.

In general, the achievable data rate is limited by several factors; an important one is the speed of the connection. Typical Ethernet speeds are 10 Mb/s, 100 Mb/s, and 1,000 Mb/s. For a full-duplex 100 Mb/s connection and a minimum message size of 84 bytes (Figure 6.7) the minimum frame delay T_{frame} is $(84 \times 8)/(100 \times 10^6) = 6.72\ \mu s$, corresponding to a maximum frequency of 148.81 kHz. This delay is fixed if the useful data packet size is less than the minimum size of 18 bytes. For a full message of 1538 bytes the theoretical minimum delay is $(1538 \times 8)/(100 \times 10^6) = 123.04\ \mu s$, corresponding to a maximum frequency of 8.1274 kHz. In control applications the messages are short and so the frame delays are expected to be close to 10 μs, based on the previous calculations. Interestingly, there is no difference in delay when sending 1 and 18 bytes of useful data; the optimal exploitation of the frame would be to send 18 bytes of information.

Although, the above discussed network delay can become a bottleneck as the system size increases and the performance requirements raise, the node processing delays are those that typically dominate in small networks. The processing power of the hardware, and the software design determine

Table 6.1 The structure of the message contained in the UDP data frame

Message ID	Data	Description
0x31	1 byte (PWM duty cycle)	PWM and DIR command
0x42	4 byte (encoder)	Encoder measurement

how fast the data can be processed at the nodes. Referring to (3), besides the already discussed T_{frame}, the components T_{pre}, T_{post} and T_{switch} are the most important for estimating the total delay, since the waiting time T_{wait} and the propagation time T_{prop} are negligibly small. To achieve the desired T_{pre} and T_{post}, suitable embedded computers must be properly selected and programmed. Finally, to keep T_{switch} to a minimum the data rates must be set in software such that segment congestions are always avoided.

The calculations made set the first limitation for the maximum achievable frequency at 150 kHz. Adding also the node and the switch delays this frequency is further reduced. However, since it is hard to estimate the total delay, this is found experimentally in most cases. This is a critical point as the developer must guarantee that the data rates in all network segments are always kept below the maximum values. Real experiments are presented in the last section showing realistic high frequency control loops and the resulting performance.

As far as software design at the application layer is concerned, a custom protocol is used on top of UDP, for a node and a ROS computer to exchange sensor measurements and control outputs. This includes a byte for defining the message ID, and a number of bytes of data depending on the message ID, see Table 6.1. The protocol is easily adjustable to other applications, as will be shown in the example cases.

6.2.2.13 Software design for the ROS computer

In system design, there is always a discussion concerning the software tool chains that should be used. Also with most platforms, the software created for specific hardware components cannot be transferred easily and reliably to a version consisting of different parts. On the other hand, ROS is a platform including a large database of drivers for devices and sensors, allowing for easy cross communication between processes; this way the need for custom software to handle communication is eliminated. ROS runs on Linux, thus allowing for running the same software on most computers in a technology lab. Also, software for common applications, such as motor control can be used off-the shelf with minimum modifications.

Communication between processes is one of the first challenges a developer faces when designing a robot. ROS provides a messaging system that manages all the communication details, eliminating this way the need for setting up communication protocols, defining data exchange rates etc. Specifically, ROS's basic communication system is an anonymous, asynchronous publish/subscribe mechanism, using nodes (the ROS form of executable files, written in C++, Python or other). A node can publish messages on a bus called topic, to which other nodes can subscribe and receive them. This organization leads to less complex and more readable code. ROS also includes other communication structures like services and actions that can be used depending on the case.

Those said, a ROS running computer is selected here to play the role of the master of the NCS shown in Figure 6.6. In this generic case, the application layer software in the ROS computer side needs to be designed such as to control n motors by exchanging messages with n MCUs, as defined by the protocol described in Table 6.1. To this end, two ROS nodes are required for each motor's low-level control, and a third node to play the coordinating role of a higher level controller.

The first node handles the UDP communication with an MCU; it receives encoder measurements and transmits control outputs, i.e. PWM signal values, back to the MCU. It also communicates with the nodes that implement the controller, the user interface etc. In this interface node, an asynchronous server is set up to receive and send packets, and the IP and the send/receive ports are also defined. The incoming encoder data are read by the node using a *sigaction* function. Every time it receives a message, the signal handler is triggered, the execution of the main function is interrupted, the data are read, and then the execution of the main function continues from where it stopped. Next, the encoder value is published to a topic usually named */state*, as a way of sharing the data with other nodes, e.g. the controller node. Depending on the case, publishing can be done in two ways. If it is done inside the signal handler, the data are published at the same rate they are received, and thus the MCU transmission rate determines also the publishing rate of the node. Alternatively, publishing can be done inside the main function. In this way, and since ROS allows the user to set the loop rate of the main function, the publishing rate is determined by this predefined loop rate and is different from the MCU's transmission rate. Actually, the achieved rate of the node may deviate from the predefined one, since it also depends on the transmission rate of the MCU, the computer resources, the number of running processes and other factors.

Except for sending data, the interface node also needs to receive data from other nodes; by subscribing to a topic named */control_effort*, it reads the calculated by the controller node PWM duty cycle and sends it back to the MCU via UDP. This second controller node operates at the rate it receives messages, i.e. at the rate the */state* topic is published. The publishing rate of the */state* topic is in fact the rate of the control loop. Finally, a third node is typically used as a higher level controller, e.g. to coordinate *n* actuators to perform a task.

Great attention must be paid to the data rates defined in software design. As can be seen in Figure 6.6, given that *n* motors are controlled at the same time by the ROS computer, the segment connecting this computer with the switch is where the maximum traffic will appear. Consider a message transmission rate f_{transm} defined for each MCU and the maximum achievable data rate for a segment $f_{\max}$ as defined in the previous paragraphs, then for *n* MCUs the inequality $f_{\mathrm{transm}} < f_{\max}/n$ must always hold to ensure good system performance.

A last critical point concerning the software design of the ROS computer is the processing delay introduced in the total control loop delay, e.g. it is big if graphics are running or data are printed. Also, the non-real time character of the Linux-ROS system gives a non-deterministic character to the system, and thus attention must be paid to the number and the kind of processes running each time. Graphical user interfaces or similar tasks should better run on a different ROS computer connected to the network; ROS allows for easily distributing the application nodes to run in multiple computers.

6.3 Application Experiments

The methods discussed theoretically in the previous sections are applied here to real applications. Two examples concerning legged robotics are presented. The first describes the control system of an instrumented treadmill, on which a single actuated hopping robot – presented as the second application example – can be tested.

6.3.1 Treadmill Control

6.3.1.1 System description

The first example concerns the velocity control of a treadmill (Figure 6.8) placed in the Control Systems Lab (CSL) of the National Technical University of Athens. It is 6 m long and driven by two 3-phase induction

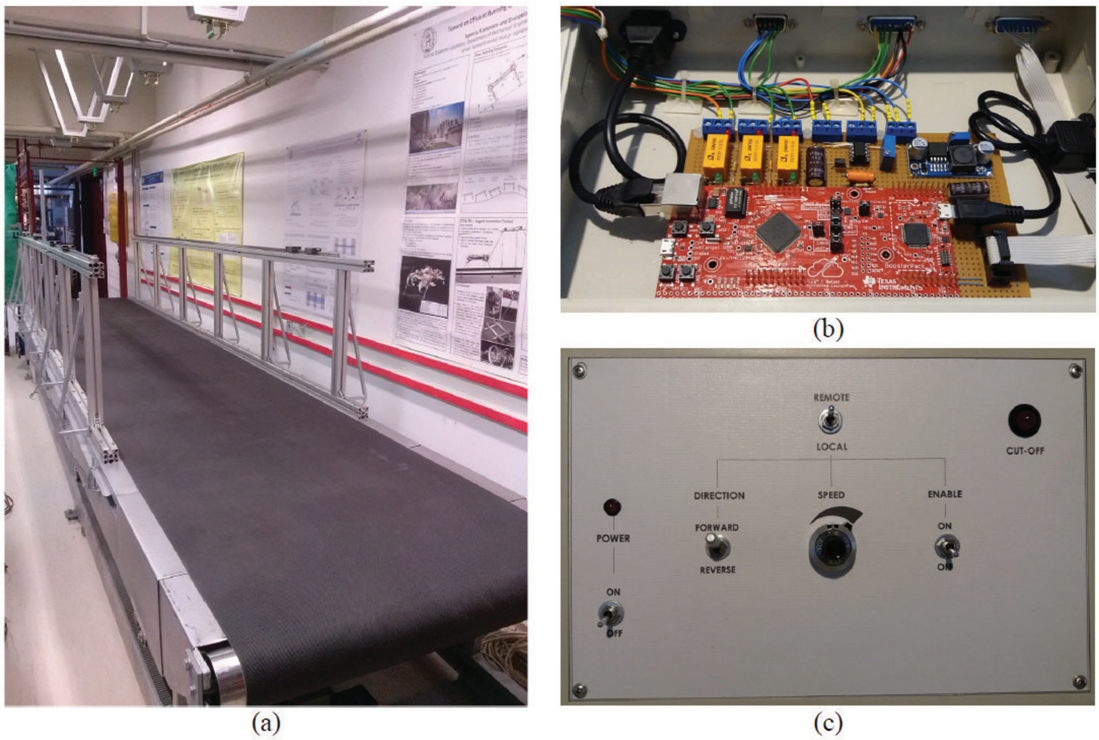


Figure 6.8 (a) The treadmill placed in the Control Systems Lab (CSL) of the NTUA, (b), (c) The treadmill's control system.

motors. The first motor (model MS 100L 2-4, XIUSHI) drives the belt's main pulley achieving a maximum running velocity of 12.6 m/s. The second motor (model FC80-4, Electro Adda) actuates on an endless screw and a rack-pinion system that regulates the treadmill's inclination. Both are driven by inverters; an EMERSON M200-022 model for the belt's motor and a SIEMENS SINAMICS G110 for the inclination motor. These control the motors' velocity according to the formula $n = 120f/P$, where f is the AC current frequency and P the number of poles.

Motor control can be achieved using the control inputs (terminals in Figure 6.9) or the control panel provided by the inverters. While in terminal control mode, at minimum three connections are required (Figure 6.10): system activation through the enable terminal 11, selection of rotation direction (terminals 12 or 13 for forward or reverse respectively) and reference input voltage (terminals 1–2). To drive the motor, both direction and enable terminals have to be connected. With these connections established the motor runs in a minimum frequency predefined in its memory (7–8 Hz). To make the system controllable the terminals need to be connected/disconnected electronically. As the manufacturer specifies, they require a 24-V input, which

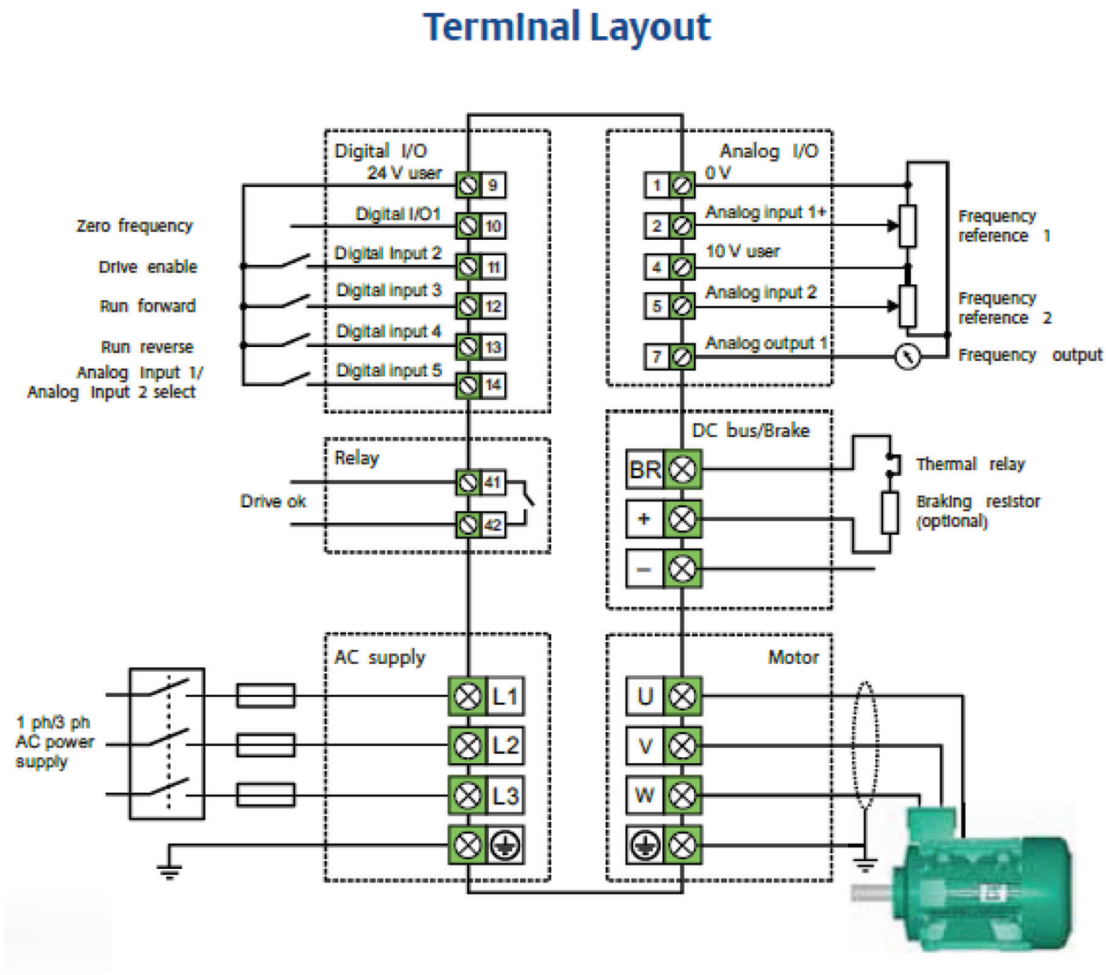


Figure 6.9 The terminal layout of the treadmill's inverter.

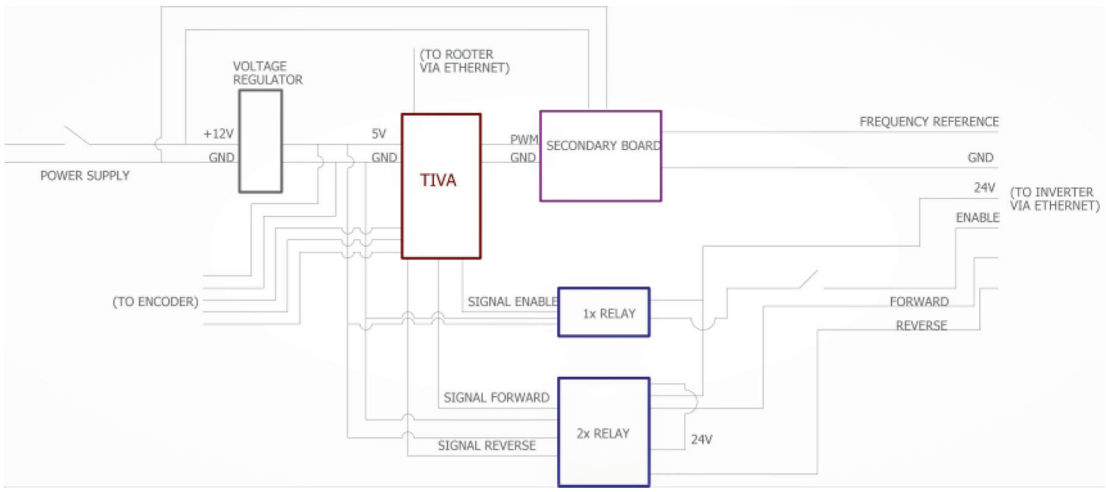


Figure 6.10 The electronic schematic of the treadmill's control system.

can be provided by terminal 9; one needs to short-circuit terminals 9–11 and 9–12 or 9–13 according to the desired direction of motion. To define the frequency, a 0- to10-V input is required on terminal 1 and GND on terminal 2. Three relay modules are used for this, which require a 5-V power supply and a logical signal. To measure the speed of the belt an HEDL-5540 incremental encoder is installed.

To satisfy all requirements, the basic components used are a TM4C1294 Connected LaunchPad, a TP-LINK AC1750 router, and a regular PC running Ubuntu 16.04 and ROS Kinetic. With this setup the treadmill can be controlled also by any other PC on the network. The system built is shown in Figure 6.8.

6.3.1.2 Software design: MCU side

On the MCU side, one needs to enable the QEI and PWM modules and to set up the UDP communication. Moreover, the IP addresses of the MCU and the ROS computer, the device subnet, the device gateway and the send and receive ports must be defined. The software is available on [33].

6.3.1.3 Software design: ROS side

The software designed for the ROS computer consists of three nodes. The first is named *ros_speed_enable* and it is the interface between ROS and the MCU. It sets up the UDP communication, receives velocity measurements from the

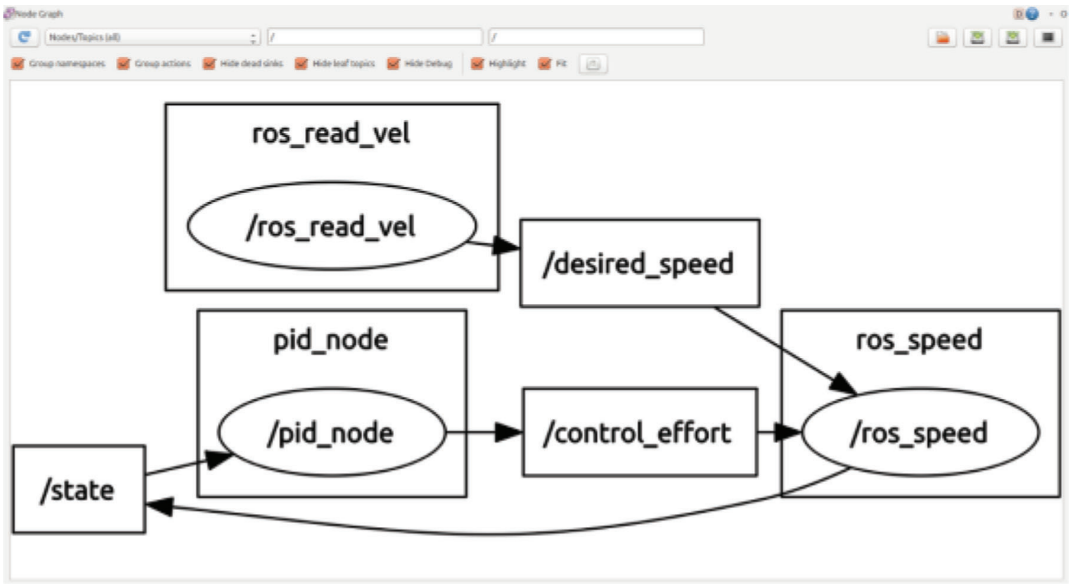


Figure 6.11 The communication system as a ROS graph.

encoder, and publishes them on the topic named*/state*. Also, it subscribes on the topic named */control_effort* to receive the calculated PWM commands, which are sent back to the MCU. Note that the send port of the MCU is the receive port for ROS and vice versa. The last topic is published by the control node *pid_node*, which receives the measured velocity from the */state* topic and the desired velocity from the */setpoint* topic as published by the third node *ros_read_velocity*. The latter is the user interface reading the desired speed from the keyboard. The software structure is shown in Figure 6.11 and the software is available on [33].

6.3.1.4 Hardware experiment

An experiment was conducted, where the treadmill was commanded to follow a velocity profile rising linearly with time to 10 m/s, then reducing to 6 m/s, rising again to 10 m/s and finally slowing down back to 0 m/s (Figure 6.12. The sampling frequency was set at 10 kHz, the PID control node frequency was set at 2 kHz, and the velocity was estimated with a frequency of 15 Hz, which however worked well. Also, adjustments of the gains K_{p} and K_{d} had to be made.

As deduced from the results shown in Figure 6.12, this is a particularly slow system mainly because of the inverter, which applies a fixed acceleration rate for security reasons; it takes about 22 s to reach 10 m/s. It also seems that the controller needs modifications to reduce the high-frequency oscillations that appear.

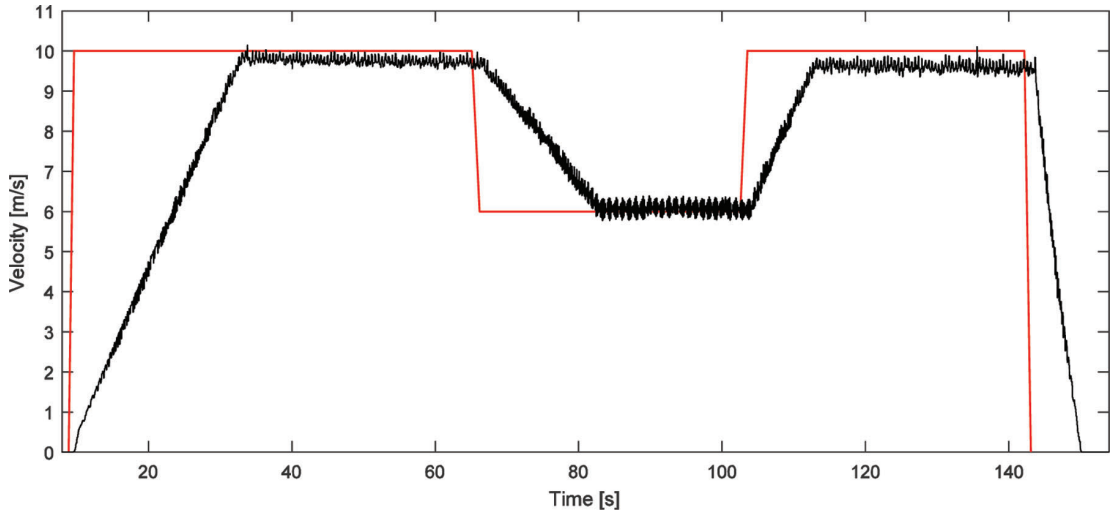


Figure 6.12 Results from the treadmill experiment.

6.3.2 Single Actuated Hopping Robot (SAHR)

This example refers to the Single Actuated Hopping Robot (SAHR), designed and built in CSL [34]. Practically, it is a realization of the standard Spring Loaded Inverted Pendulum (SLIP) model. Here, the hardware and software are redesigned based on the principles presented in the previous sections.

6.3.2.1 Robot description

The system is designed to move on the sagittal plane and has an actuated revolute hip joint, and a passively compliant prismatic knee joint using a spring of stiffness 5900 N/m. Most parts are made of aluminum, while the leg shaft receiving most of the impact loads is made of steel. A Maxon motor (RE35, 90 W) is used with maximum continuous current 3.36 A, nominal voltage 24 V and maximum continuous torque 0.0933 Nm. A planetary gearhead with gear ratio 26:1 is attached together with a belt drive of reduction ratio 2:1 to move the hip axis. The motor drive is an AZBDC12A8 by Advanced Motion Controls (AMC), supplying up to 6 A continuous and 12 A intermittent current. To use it, one has to send an enabling signal to the corresponding pin, a High/Low signal to the direction pin, and a PWM signal corresponding to the desired current (by default, 100% duty cycle corresponds to 12 A).

Regarding the sensory system, two incremental encoders measure the spring compression and the hip angle, providing three pulses, A, B, and Index. Measurements from the spring encoder can be used to calculate the body CoM position in stance phase, detect transitions from stance to flight phase and vice versa, or even estimate the ground force. However, for a better ground force measurement, a 3-Axis force sensor by BOTA Systems is also installed.

6.3.2.2 Software design: MCU side

Two TM4C1294 boards are used to read the sensors since each board has only one QEI module. The board that reads the hip encoder also sends commands to the motor drive. Their programming is identical to that for the treadmill control system, with small modifications concerning the IP addresses and the encoder related parameters. The software is available on [33].

6.3.2.3 Software design: ROS side

In this case, where two encoders need to be read, the advantages of ROS become apparent. Two identical MCU interface nodes are used for the

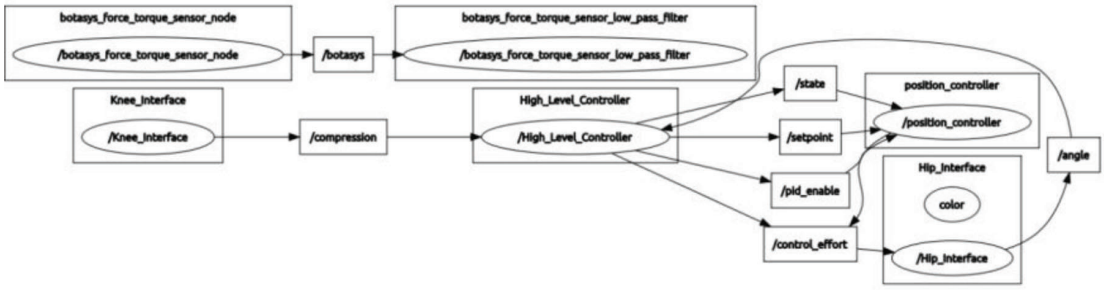


Figure 6.13 The ROS graph for the control system of SAHR.

encoder measurements to become available for every node that needs them. To control the hip angle a PID controller node from the *ros_pid* package [35] is used. The force sensor connected to the computer is ROS-ready and only the driver needs to be built. Extra nodes can be also added depending on the application. For a position control experiment a user interface is required to send commands, and for a force sensing experiment the force sensor driver must run. In the case of a hopping experiment, a high level controller is also required to read the knee encoder or the force sensor data and decide if the leg is in stance or flight phase (Figure 6.13). In flight phase it can position the leg for landing, while in stance it can push the body forward with a predefined torque. However, more complex control algorithms can be used by taking into account the ground stiffness and the energy losses as proposed in Vasilopoulos et al. [36]. The software is available on [33].

6.3.2.4 Simulation experiment

The purpose of the experiment is to determine how close a Gazebo simulation is to reality, but also to test the developed ROS system. A big advantage of the Gazebo and ROS setup is the direct simulation with the software used on the actual hardware. In this case two custom plugins are used; one to publish the joint states and one to subscribe to a topic to receive hip commands (just like the MCUs in the real robot).

The robot was left to fall from a height of 0.1 m, and the leg was controlled to remain vertical. The ground was considered infinitely stiff, and this might be one of the reasons that a small deviation between simulation and experiment was observed. The ODE solver was used with a 0.001 s maximum step size. The simulation results are shown in Figure 6.14 and also in the video presented in SAHR [37].

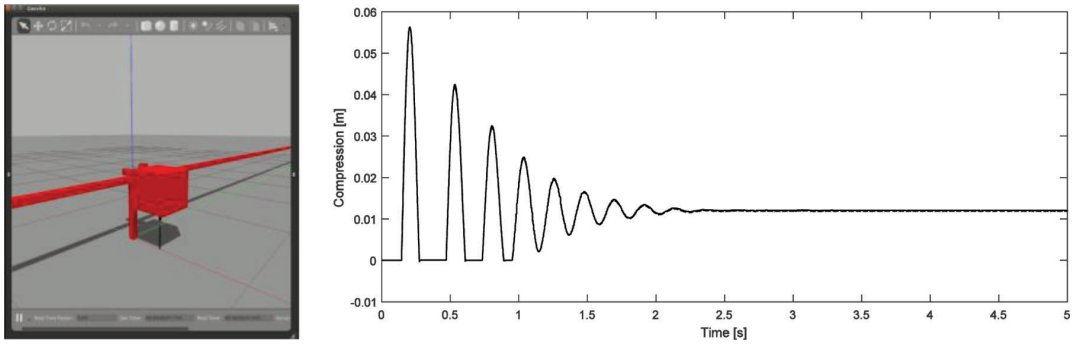


Figure 6.14 Gazebo simulation for the monopod hopper SAHR.

6.3.2.5 Hardware experiment

To constraint the robot to move on the sagittal plane, it is mounted on a supporting mechanism on the treadmill. Like in the simulation, the leg is controlled to stay in the vertical position with a PD controller, as shown in Figure 6.15. Leaving the rest of the ROS system the same, the MCU nodes are added. The transmission frequency of each MCU is set to 15 kHz and the PID frequency to 1 kHz.

Of great importance is the good matching between simulation and experiment for many aspects in robotics research. As deduced from Figure 6.14 and Figure 6.16, simulation and reality were close in this experiment, showing the same number of bounces, the same settling time (about 2.3 s) and the same steady-state compression (0.012 m). A slight difference is observed in the maximum compression appearing higher in simulation, which could be

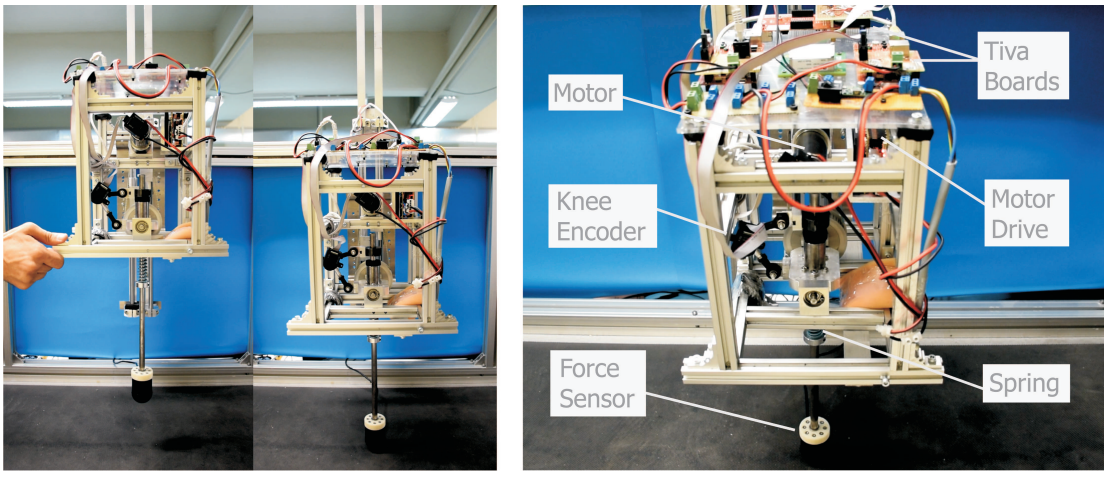


Figure 6.15 The Single Actuated Hopping Robot (SAHR) in a hopping experiment.

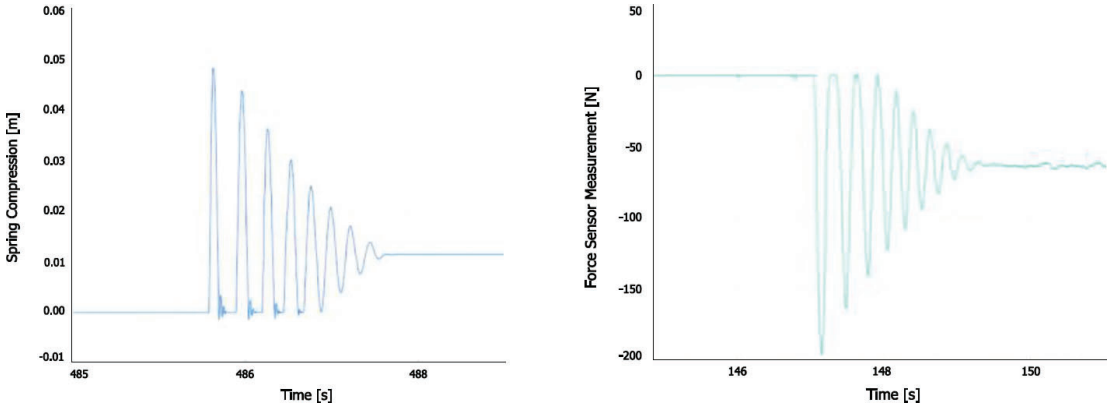


Figure 6.16 Results from the SAHR hopping experiment.

explained by unmodeled frictional losses. A video of the experiment can be found in SAHR [37].

6.3.2.6 Simulations on interactions with terrains

Another interesting example of simulations with SAHR, is the case of planetary environments with different gravitational accelerations and different terrain types. In the following example [38], a controller presented in Vasilopoulos et al. [36], tries to move the SAHR in three different environments. It is interesting to see how the motion profiles are affected by the abovementioned parameters. The robot behavior was simulated for the

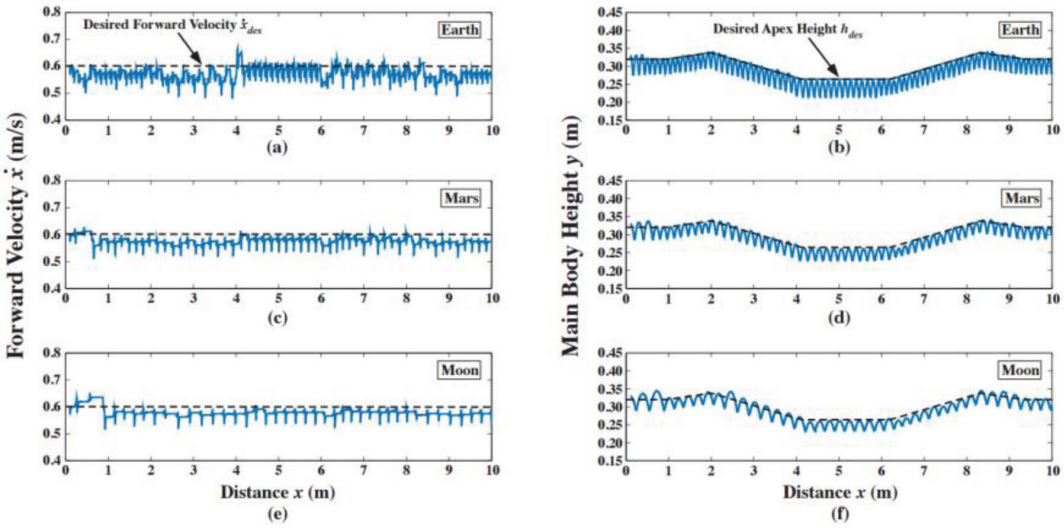


Figure 6.17 Controller performance on shallow crater: (a, c, e) Forward Velocity on Earth, Mars and Moon respectively, (b, d, f) Main Body Height on Earth, Mars and Moon, respectively.

Earth ($g = 9.81\ \mathrm{m/s^2}$), the Mars ($g = 3.711\ \mathrm{m/s^2}$) and the Moon ($g = 1.622\ \mathrm{m/s^2}$). While the controller adapted quickly to each terrain and followed the desired objectives of forward velocity and body height in each case, as it is shown in Figure 6.17, it can be observed that the response converged more slowly to the desired commands as the acceleration of gravity decreased.

6.4 Conclusions

In this chapter, advanced but easy to implement and low-cost methods were proposed for modeling, control, design and development of robotic systems. Widely used software tools such as Matlab, Gazebo, and ROS were combined to form a framework for fast simulation and prototyping of robots seen as Networked Control Systems (NCSs). The focus was on giving guidelines on how to directly test a conceptual idea in design and control using low-cost solutions in software and hardware. Two application examples concerning legged robotics were finally presented as a proof of concept, and all the software was built such as to be easily reused by the reader in similar or modified applications.

Acknowledgment

Funding for this research by the "IKY Fellowships of Excellence for Postgraduate Studies in Greece – Siemens Programme" in the framework of the Hellenic Republic – Siemens Settlement Agreement is acknowledged. The authors wish to thank Stylianos Vagenas for assistance with the treadmill control, and Pavlos Stavrou for assistance with the development of software.

References

[1] Marwedel, P. (2011). *Embedded System Design, Embedded Systems Foundations of Cyber-Physical Systems*. Netherland: Springer.

[2] Rawat, D. B., Rodriques, J., and Stojmenovic, I. (2015). *Cyber Physical Systems: From Theory to Practice*. Boca Raton: CRC Press, Taylor & Francis Group.

[3] Jeschke, S., Brecher, C., Song, H., and Rawat, D. B. (2017). *Industrial Internet of Things, Cybermanufacturing Systems*. Berlin: Springer International Publishing.

[4] Gilchrist, A. (2016). *Industry 4.0: The Industrial Internet of Things*. New York, NY: Apress.
[5] Song, H., Rawat, D., Jeschke, S., and Brecher, C. (2017). *Cyber-Physical Systems: Foundations, Principles and Applications*. Academic Press.
[6] ROS. http://www.ros.org/
[7] Raspberry Pi. https://www.raspberrypi.org/
[8] Ivaldi, S., Padois, V., and Nori, F. (2014). Tools for dynamics simulation of robots: a survey based on user feedback. arXiv preprint. arXiv:1402.7050
[9] https://github.com/kostasmach/Laelaps_sim
[10] Machairas, K., and Papadopoulos, E. (2016). "An Active Compliance Controller for Quadruped Trotting," in *24th Mediterranean Conference on Control and Automation (MED '16)*, Athens, Greece.
[11] Stronge, W. J. (2000). *Impact Mechanics*. Cambridge: Cambridge University Press.
[12] Hunt, K. H. and Crossley, F. R. E. (1975). Coefficient of Restitution Interpreted as Damping in Vibroimpact. *J. Appl. Mech.* 440–445.
[13] Majeed, M. A., Yigit, A. S., and Christoforou, A. P. (2012). "Elasto-plastic contact/impact of rigidly supported composites. *Compos. Part B Eng.,* vol. 43, 1244–1251.
[14] Paraskevas, I. (2015). "Capturing of Orbital Space Systems by Robots." Ph.D. Dissertation, National Technical University of Athens, Greece.
[15] Gupta, R. A., and Chow, M.-Y. (2010). Networked control system: Overview and research trends. *IEEE Trans. Ind. Electr.* 57, 2527–2535.
[16] Valenzano, A., and Cena, G. (2014). "Protocols and Services in Controller Area Networks," in *Industrial Communication Technology Handbook,* 2nd Edn. (Boca Raton, FL:), 52.1–52.49.
[17] Cena, G., Scanzio, S., Valenzano, A., and Zunino, C. (2014). "Ethernet for Control Automation Technology," in *Industrial Communication Technology Handbook, Second Edition* (Boca Raton, FL: CRC Press), 18.1–18.27.
[18] Zurawski, R. (2014). *Industrial Communication Technology Handbook*, 2nd Edn. Boca Raton, FL: CRC Press, 2014.
[19] Sauter, T., Soucek, S., Kastner, W., and Dietrich, D. (2011). The evolution of factory and building automation. *IEEE Ind. Electr. Mag.* 5, 35–48.

[20] Zurawski, R. (2014). "Fieldbus System Fundamentals," in *Industrial Communication Technology Handbook, 2nd Edn*, Boca Raton, FL: CRC Press.
[21] Lounsbury, R. (2008). *Industrial Ethernet on the plant floor: A planning and installation Guide*, ISA.
[22] Kurose, J. F., and Keith, R. W. (2013). *Computer networking: a top-down approach* (6th Edition). Upper Saddle River, NJ: Pearson.
[23] Zurawski, R. (2014). "Networked Control Systems for Manufacturing," in *Industrial Communication Technology Handbook.* Boca Raton, FL: CRC Press.
[24] Axelson, J. (2003). *Embedded Ethernet and Internet Complete*, Lakeview Research LLC.
[25] Enner, F. (2016). *Analyzing the viability of Ethernet and UDP for robot control*. https://ennerf.github.io/2016/11/23/Analyzing-the-viability-of-Ethernet-and-UDP-for-robot-control.html
[26] Prytz, G., and Johannessen, S. (2005). "Real-time performance measurements using UDP on Windows and Linux," in *10th IEEE Conference on Emerging Technologies and Factory Automation, ETFA* (New York, NY: IEEE).
[27] EtherCAT. https://www.ethercat.org/
[28] Zimmerman, J. and Spurgeon, C. E. (2014). *Ethernet: The Definitive Guide*, 2nd Edn. Newton, MA: O'Reilly Media, Inc.
[29] Loeser, J., and Haertig, H. (2004). "Low-latency hard real-time communication over switched Ethernet," in *16th Euromicro Conference on Real-Time Systems, ECRTS* (New York, NY: IEEE).
[30] Parrott, J. T. Moyne, J. R., and Tilbury, D. M. (2006). "Experimental Determination of Network Quality of Service in Ethernet: UDP, OPC, and VPN," in *American Control Conference*, Minneapolis, Minnesota, USA.
[31] Gottlieb, I. M. (1993). *Practical RF power design techniques*. New York, NY: TAB Books.
[32] EK-TM4C1294. http://www.ti.com/tool/ek-tm4c1294xl
[33] https://github.com/ntua-cslep/Legged
[34] Cherouvim, N., and Papadopoulos, E. (2008). "The SAHR robot: Controlling Hopping Speed and Height Using a Single Actuator," in *Applied Bionics and Biomechanic,* vol. 5, 149–156.
[35] http://wiki.ros.org/pid

[36] Vasilopoulos, V., Paraskevas, I., and Papadopoulos, E. (2015). "Control and Energy Considerations for a Hopping Monopod on Rough Compliant Terrains," in *IEEE International Conference on Robotics and Automation (ICRA '15)*, Seattle, WA, USA.

[37] SAHR. (2016). *Single Actuated Hopping Robot experiment.* https://www.youtube.com/watch?v=Dcw0f3NULy4&feature=youtu.be

[38] Vasilopoulos, V. Paraskevas, I., and Papadopoulos, E. (2015). "Monopod Hopping on Rough Planetary Environments," in *13th Symposium on Advanced Space Technologies in Robotics and Automation, (ASTRA '15)*, ESA, ESTEC, Noordwijk, The Netherlands.