Università degli Studi di Padova

TESI DI LAUREA MAGISTRALE IN INGEGNERIA DELL'INNOVAZIONE DEL PRODOTTO

Facoltà di Ingegneria

Input synchronization of a real car and its real-time simulator

Advisor: Prof. Alberto Trevisani[1]

Co-advisor: Prof. Miguel Á. V. Naya[2]

Student: Pasquale Gallo

Matriculation number: 1013490

Dipartimento di tecnica e gestione dei sistemi industriali

Academic year 2011-2012

[1] *Università degli studi di Padova, Dipartimento di Tecnica e Gestione dei Sistemi Industriali, Vicenza.*

[2] *Universidad de La Coruña, Departemento de Ingenería Industrtial II, Ferrol, Spain.*

"It is by logic we prove, but by intuition we discover."
(Leonardo da Vinci)

Abstract

Il presente lavoro riassume il progetto di tesi svolto all'estero, presso *Universidad de La Coruña*, nell'arco di sei mesi. L'attività è stata condotta nel *Laboratorio de Ingeniería Mecánica* (LIM) di Ferrol, punto di riferimento anche a livello internazionale nell'ambito di multibody system dynamics e simulazione dinamica.

Il progetto ha avuto come obiettivo lo sviluppo di un interfaccia tra un prototipo reale di un autoveicolo, ed il suo simulatore dinamico real-time, prestando attenzione alla sincronizzazione in termini di tempo tra le azioni dell'utente sul prototipo e le variabili inviate al simulatore. La simulazione on-board così eseguibile è un passo fondamentale per estendere le applicazioni del simulatore al di fuori del campo scientifico ed accademico. Infatti i benefici dal punto di vista industriale sono molti: primo tra tutti l'incremento della sicurezza dei veicolo terrestri e non solo.

Per simulatore dinamico si intende un software, largamente utilizzato in molti campi dell'ingengneria, che ha la capacità di simulare una moltitudine di modelli multibody, analizzando sia gli aspetti cinematici che quelli dinamici del sistema. Questo aspetto distingue un simulatore dinamico da una comune animazione 3D, la quale è solo approssimativamente capace di simulare la cinematica di semplici sistemi. L'ausilio di un simulatore dinamico permette l'analisi di sistemi complesssi, composti da un elevatissimo numero di elementi, grazie all'implementazione di opportuni metodi numerici capaci di individuare le soluzioni delle equazioni differenziali non-linari risultanti da questo tipo di analisi.

Il prototipo oggetto dello studio è stato dotato di un sistema di acquisizione dati (DAS) per campionamenti ad alta frequenza. Dopo un attenta analisi del simulatore, sono state scelte tre variabili di input: pressione del freno, posizione del volante, posizione dell'acceleratore. Le grandezze in esame sono state monitorate grazie ad opportuni sensori: sensore di pressione ed econders.

Attraverso un linguaggio di programmazione proprio del sistema di acquisizione dati è stata sviluppata un applicazione capace di gestire il DAS, elaborare le informazioni provenienti dai sensori e di inivare le grandezze di interesse al simulatore. Più in dettaglio: il sistema di acquisizione dati riesce a monitorare di continuo le tre variabili di interesse; queste vengono lette e organizzate in blocchi; ogni blocco contiene le tre informazioni che approssimativamente possono essere riferite al medesimo istante temporale; tra tutti i blocchi, l'ultimo, che è il più recente in termini di tempo, viene inviato a delle specifiche funzioni capaci di elaborare le informazioni e determinare le grandezze di interesse del simulatore, ovvero pressione del freno, coppia motrice alle ruote, posizione del volante, velocità angolare del volante, accelerazione angolare del volante. Il simulatore a questo punto ha a disposizione tutte le informazioni per elaborare la simulazione. Un aspetto molto importante da sottolineare è la coerenza in termini di tempo tra l'istante temporale della chiamata del simulatore e l'istante temporale a cui si riferiscono le grandezze. Grazie infatti alla metodologia di trasferimento adottata, si riesce ad ottenere una certa sincronizzazione tra le azione dell'utente e le variabili inviate al simulatore.

I risultati raggiunti sono molto promettenti. In particolare la simulazione on-board è resa possibile: all'esecuzione della manovra del prototipo reale, corrisponde una simulazione in tempo reale visibile su un piccolo monitor a bordo macchina, ottenendo un tempo di trasferimento delle informazioni di circa 0.43 ms.

Acknowledgements

At the end of an hard and long work, spend some words for the people that help us is a pleasure and an obligation.

I would thank, first of all, the Professor Alberto Trevisani, for giving me the opportunity to carry out the present work abroad, and to live an amazing life and educational experience. Moreover he supervised my project. I would thank all my academics friends, for making incredible the last two years. I must mention, moreover, all the teachers of the master Degree, for their commitment and perseverance. A special thanks to Professor Persona, who has always been a supporter of our academic course, to Prof. F. Berto, for his patience had towards me despite my numerous mails due to request of "further informations". Thanks also to Prof. P. Lazzarin, for the passion and commitment invested in education.

I would like to thank the *Laboratorio de Ingeniería Mecánica* of Ferrol. Thanks to Prof. Naya, Prof. Cuadrado and Emilio, for allowing me to do this job. A special thanks is for Roland Pastorino, for his enormous help and for the support. Thanks to Alberto, first of all for his pleasantness, and after for the help in the code programming aspects. A huge thank you to Pedro, Amelia, Florian, David, for giving me the welcome, and for making me feel like at home. I will never forget the sympathy of Urbano, who i hope to meet in a concert, one day. A final thanks to all the people of the LIM for the wonderful moments shared.

I would love to thank my family for supporting me financially and morally not only in the last two years, but during the entire studies career. Thanks to my friends Fabio, Kris, Matt, Piga, who have always been close to me, despite the university often has taken me away for long periods. Thanks to my room-mates Vicentini: Salvo, Ludo, and Raff, for the wonderful and hilarious shared moments. Thanks to Alberto Hyvoz, first of all, a friend of mine, and for being present with his cheerfulness and kindness. Thanks to Paolino, for being a great adventure companion in Spain, and a huge hug to Noelia, Miguel, José, and all my new friends that i have leaved in Spain and beyond.

Dulcis in fundo, the most important "thank you" is for Carlotta, who has always encouraged me, supported, pushed, in the past three years. If i got where i am, i owe it especially to her.

Agradecimientos

Al final de un largo viaje es más un placer dar las gracias a todos los que me han acompañado y han estado a mi lado.

Me gustaría agradecer ante todo al profesor Alberto Trevisani, por haberme dado la oportunidad y la confianza de desarrollar el presente trabajo de tesis en el extranjero, regalándome una experiencia formativa y personal inolvidable. Además le doy las gracias por el trabajo de supervisión del presente proyecto. Agradezco a todos mis compañeros de curso, con los que he pasado dos años increí. Agradezco a todos los profesores del grado de "Innovazione del Prodotto", por el distinguido trabajo desarrollado con empeño y constancia. Un agradecimiento especial al Profesor A. Persona, desde siempre defensor de nuestro curso, y también al Profesor Filippo Berto, por la paciencia demostrada conmigo a pesar del persistente tráfico de emails en este ultimos dos años. Un agradecimiento además al Profesor P. Lazzarin, por la pasión y el empeño invertidos en la enseñanza.

No puedo quedar sin hacer un agradecimiento tambié al *Laboratorio de Ingeniería Mecánica* de Ferrol. Un profundo agradecimiento para el Prof. Naya, el Prof. Cuadrado y a Emilio, por permitirme desarrollar el presente trabajo poniendo a mi disposición todos los instrumentos necesarios. Merece un reconocimiento especial por su enorme ayuda, y por su supervisión, Roland Pastorino, que me ha respaldado siempre. Gracias a Alberto, ante todo por la simpatía, pero tambié para la ayuda que me has dado en todo lo relativo a los códigos de programación. Un gracias de corazón a Pedro, Amelia, Florian y David, por darme la bienvenida ya desde el primer día y por hacerme sentir como en casa. No olvidaré jamás a Urbano por la contagiosa simpatía, indudablemente nos encontraremos en algún concierto! Un gracias final a todas las personas del LIM, por los estupendos momentos compartidos.

Doy las gracias a mi familia por apoyarme económicamente y moralmente no sólo en los últimos dos años, sino durante toda la duración de mis estudios. Gracias a mis amigos, Fabio, Kris, Matt, Piga, que siempre han estado cerca de mi, aunque la universidad a menudo me ha llevado lejos por largos períodos. Gracias a mis compañeros de Vicenza: Salvo, Ludo, Raff, por los muchos momentos compartidos, y por las inmensas carcajadas. Un gracias a Alberto Hyvoz, en primer lugar un amigo, por darme un techo en los últimos dos años, y por haber estado siempre presente con su jovialidad y amabilidad. Un gracias a Paolino, por haber sido un gran compañero de aventura en tierra española. Y un enorme abrazo a Noelia, Miguel, José, y a todos mis nuevos amigos españoles.

Un último agradecimiento, pero sin duda el más importante es para mi novia, que siempre me ha alentado, apoyado e impulsado en los últimos tres años, acompañándome hasta la consecución de esta especial meta de mi vida. Gracias de corazón Carlotta.

Ringraziamenti

Alla fine di un lungo percorso, è sia un piacere, che un dovere, ringraziare chi ci ha accompagnato e stato vicino.

Vorrei ringraziare innanzitutto il Professor Alberto Trevisani, per avermi dato la possibilità (e la fiducia) di svolgere il presente lavoro di tesi all'estero, regalandomi un esperienza di vita e formativa indimenticabile. Lo ringrazio inoltre per il lavoro di supervisione del presente progetto. Ringrazio tutti i miei compagni di corso, con i quali ho passato due anni davvero bellissimi. Ringrazio tutti i Professori del corso di Laurea di Innovazione del Prodotto, per l'egregio lavoro svolto con impegno e costanza. Un particolare ringraziamento al Professor Persona, da sempre sostenitore del nostro corso, ed al Professor Filippo Berto, per la pazienza avuta nei miei confronti nonostante le asillanti mails dovute a richieste di approfondimenti. Un ringraziamento inoltre al Professor P. Lazzarin, per la passione e l'impegno investito nell'insegnamento.

Non posso non ringraziare il *Laboratorio de Ingeniería Mecánica* di Ferrol. Grazie al Prof. Naya, al Prof. Cuadrado e ad Emilio, per avermi permesso di svolgere il presente lavoro, mettendomi a disposizione tutti gli strumenti. Un grazie speciale per il suo enorme aiuto e per la sua supervisione, a Roland Pastorino, che mi ha sempre supportato. Grazie ad Alberto, innanzitutto per la simpatia e poi per l'aiuto negli aspetti di programmazione. Un enorme grazie a Pedro, Amelia, Florian, David, per avermi dato il benvenuto sin dal primo giorno e per avermi fatto sentire a casa. Non dimenticherò poi la simpatia di Urbano, che spero di incontrare a qualche concerto. Un grazie finale a tutte le persone del LIM, per i momenti stupendi condivisi.

Ringrazio la mia famiglia, per avermi sostenuto economicamente e moralmente non solo negli ultimi due anni, ma durante tutta la durata del mio percorso di studi. Grazie ai miei amici Fabio, Kris, Mattia, Piga, da sempre vicini, nonostante l'Università mi abbia spesso portato lontano per lunghi periodi. Un grazie ai miei coinquilini Vicentini: Salvo, Ludo, Raff, per i tanti momenti condivisi e per le risate. Un grazie ad Alberto Hyvoz, prima di tutto amico, per avermi dato un tetto negli ultimi due anni, per essere stato sempre presente con la sua allegria e gentilezza. Un grazie a Paolino, per essere stato un grande compagno di avventura in terra spagnola, ed un enorme abbraccio a Noelia, Miguel, José ed a tutti i miei nuovi amici che lascio in Spagna e non solo.

Un ultimo grazie, ma sicuramente il più importante, è per Carlotta, che mi ha sempre incoraggiato, sostenuto, spinto, negli ultimi tre anni. Se sono arrivato dove sono, lo devo sopratutto a lei.

Table of Contents

List of Figures xv

List of Tables xvii

1 Introduction 1
- 1.1 Background . . . 3
- 1.2 Objectives . . . 5
- 1.3 Thesis structure . . . 5

2 Basic concepts of multibody systems 7
- 2.1 Definitions . . . 9
- 2.2 MBS elements representation . . . 10
  - 2.2.1 Types of coordinates . . . 10
  - 2.2.2 Natural coordinates . . . 12
- 2.3 Introduction to MBS equations of motion . . . 15
- 2.4 Multibody systems simulator and real-time concept . . . 18

3 X-by-wire prototype: features of hardware and simulator 21
- 3.1 Hardware configuration . . . 23
  - 3.1.1 Prototype description . . . 23
  - 3.1.2 Sensors . . . 26
  - 3.1.3 Data acquisition system . . . 30
- 3.2 Simulator and simulation environment . . . 33
  - 3.2.1 Brief numerical methods and analytical considerations . . . 33
  - 3.2.2 Code and software arrangement . . . 34

4 Real car-Simulator communication interface development 39
- 4.1 Overall considerations . . . 41
  - 4.1.1 Previous layout of simulations management . . . 41
  - 4.1.2 Aims and critical factors of the communication interface . . . 41
- 4.2 Simulator input variables . . . 45
  - 4.2.1 Drive wheels torque . . . 46
  - 4.2.2 Position, speed, and acceleration of the steering wheel . . . 50
  - 4.2.3 Brake pressure . . . 58
- 4.3 System communication tools . . . 59
  - 4.3.1 Introduction to DAS management . . . 59
  - 4.3.2 Communication pipes . . . 60
  - 4.3.3 Strategies available for data transfers . . . 62

4.3.4 Strategy chosen and developed for data transfers 63
4.4 Data sampling configuration 71
4.5 Whole application and buffers system 74

5 Conclusions **79**
5.1 Goals achieved, possible applications and future developments 81

Appendix A: main.cpp and guiado.f90 changes **85**

Appendix B: header files **87**

Appendix C: DapControl thread **89**

Appendix D: derived function **93**

Appendix E: compute-par function **95**

Appendix F: brake-pressure and time functions **97**

Bibliography **99**

List of Figures

2.1 Four-bar articulated quadrilateral 9
2.2 RSCR spatial mechanism 10
2.3 Solutions of the position problem in a four-bar mechanism 12
2.4 Types of coordinates used in multibody formulation 12
2.5 Four-bar articulated quadrilateral in natural coordinates 14
2.6 Multibody model of a car suspension and steering system 19
2.7 3D multibody simulator model of a car suspension and steering system 19

3.1 X-by-wire prototype 23
3.2 TBW system assembled 24
3.3 BBW system assembled 25
3.4 Diagram of SBW system 25
3.5 Encoder with spinning codewheel and stationary mask 28
3.6 Encoder sensor of steering wheel 28
3.7 Encoder sensor of throttle 29
3.8 Brake pressure sensor assembled 29
3.9 Connection scheme of DAS, PC, sensors, drivers and actuators 32
3.10 3D prototype-model of simulator 33
3.11 Real test track photo 37
3.12 3D model of test track 37

4.1 Steps scheme of a simulation without the communication system developed 42
4.2 General scheme of the communication system developed 42
4.3 First In First Out approach 44
4.4 Engine torque curve 48
4.5 Working scheme of the drive wheels torque estimation 50
4.6 Working scheme of the steering wheel position, velocity and acceleration computation 51
4.7 Steering angle of the test maneuver 52
4.8 Comparison within the maneuver not smoothed and smoothed 53
4.9 Deviation within the maneuver not smoothed and smoothed 53
4.10 Reference derivatives obtained trough a finite difference method forward 55
4.11 First derivative obtained trough the one sided smooth differentiators $N = 4$ 56
4.12 First derivative obtained through a finite difference method backward . 56
4.13 Second derivative obtained trough the one sided smooth differentiators $N = 4$ 57

4.14 Second derivative obtained trough a finite difference method backward 57
4.15 Working scheme of the brake pressure estimation 59
4.16 Bytes transferred during 1000 loops 70
4.17 Bytes transferred during an entire maneuver 70
4.18 Working detailed scheme of the application developed 76
4.19 Values assumed by the first and last block stored in bufferIn 77
4.20 Correspondence between the most recent data and the value stored in bufferOut 78

List of Tables

3.1 Gear ratios of the prototype 26
3.2 List of the sensors fitted in the prototype 27

4.1 Variables monitored and simulator inputs 46
4.2 Limit values of the function f, and equivalent digital and angle forms . 49
4.3 One sided smooth differentiators formulas 54
4.4 Sensor output, digital value, and pressure value correspondences 58
4.5 Data transferred and execution time for each test maneuvers 69

Chapter 1

Introduction

1.1 Background

The present paper was developed at *Universidad de La Coruña*[1] (UDC), collaborating with *Laboratorio de Ingeniería Mecánica* (LIM), during a six months working period. The main work area of LIM is multibody system dynamics and simulation environment. Nowadays, the term multibody systems is related to a large number of engineering fields like robotics, dynamics and vehicles. The reduction in cost, risk and time during the development is one of the most relevant contributions of MB techniques. The power of a multibody system is the possibility to create an algorithmic, computer-aided way to model, analyse, simulate and optimize arbitrary motion of possibly thousands of interconnected bodies. Different approaches are available to model a vehicle but the more detailed the model is, the more accurate the simulation predictions of the future vehicle dynamics are. However, the multibody models and formulations are strictly related to the final target pursued (e.g. handling analysis, ride analysis, durability analysis, real-time applications, crash analyses) (Rauh, 2003).

Natural coordinates and a self-developed multi-body formulation (Cuadrado et al., 1997; García de Jalón and Bayo, 1994; Bayo et al., 1991) that enables the simulation of complex systems to run in real-time with efficiency and robustness are the preferred choices to model vehicles in the LIM. The research lines of the laboratory are in detail: the investigation of *efficient methods for multibody system dynamics* (Bayo et al., 1991; Orden et al., 2007), which focus on achieving fast simulations, the purpose being either to run human- or hardware-in-the-loop applications which require real-time performance, or computationally intensive algorithms like those appearing in dynamic optimization; the implementation of *simulators of vehicles and machinery*, indeed efficient methods for the dynamics of multibody systems may be applied to the construction of simulators of vehicles and machinery for personnel training or evaluation, interfaces assessment; the study of *virtual reality in the product life-cycle*, which deals with being able to compute the real physical behaviour of virtual entities, may serve to enhance virtual reality applications with new features, which enable their efficient use in the different stages of any industrial product life-cycle like design, analysis, testing, manufacturing, assembly, maintenance, and end-of-life (currently, an application for the virtual assembling and disassembling of mechanisms is being developed); the study of the *human body* (Alonso et al., 2012; Font-Llagunes et al., 2011), that can be considered as a multibody system composed by rigid links (the bones) connected by joints and actuated by muscles; *naval and oceanic applications*, indeed multibody dynamics techniques have also application to these two fields which are so relevant for the local economy, so that the Laboratory is especially committed to develop research on them; at least, efficient methods for the dynamics of multibody systems find application in the *control of ground vehicles* (Cuadrado et al., 2012; Pastorino et al., 2011, 2010), since they serve to build models which can be used either to design and test controllers or to make part of the controllers themselves.

The project presented in this work is part of a broader research, which has been under development for many years, regarding the handling and the automation of land vehicles. Especially, the main objective broader research was to develop a car prototype and its simulator, in order to execute a real-time simulation. The validation of the multibody models, theory and implementation aspect, were other issues addressed.

[1] Campus of Ferrol, Spain.

Moreover, the LIM has recently begun a research on the use of real-time vehicle MB models in state observers. Using state estimation techniques and highly-detailed vehicle models should provide information to the controllers that is not available when using classical vehicle models (Pastorino, 2012). Three papers describe all the steps of the broader project: Naya (2007); Sanjurjo (2011); Pastorino (2012).

Naya (2007) asserts that the real-time multibody dynamic is the key to achieve better results in simulations field and in land vehicles control. Moreover he shows how it is possible to exploit a multibody formulation in order to realize detailed models of vehicles, which are suitable for real-time simulations (human- or hardware-in-the-loop) of complex maneuvers also. In his work, it has been presented the following topic: the built operations of a car prototype provided with sensors and actuators; the development of a PC application that is able to find solution of a multibody equations of motion reaching a very high correctness; implementation of a control sub-system able to repeat maneuvers; experimental validation of the application developed.

Sanjurjo (2011) developed a multibody model related to state observers theory. In fact, also a good dynamic model has differences in the results if compared to the actual prototype. These divergences are resulted from simplifications of the model, from external noises or from errors due to impossibility to know every parameters with such precision. A good way to check these errors, when the real prototype and real-time simulation are available, is to evaluate the errors comparing the behaviours of the real prototype and of the simulator. After the evaluation of the errors it is possible to introduce some adjustments in order to obtain consistent results.

Pastorino (2012) focuses on the study of the validity of real-time vehicle multibody models. For this purpose, a vehicle prototype has been built and automated in order to repeat reference maneuvers. The numerous sensors on the prototype gather the most relevant magnitudes of the vehicle motion (e.g. roll-pitch-yaw rates, wheel speeds). Two low speed maneuvers involving the longitudinal and lateral vehicle dynamics have been repeated several times in a test area. A real-time multibody model of the vehicle prototype has been prepared as well as a simulation environment that includes a close graphical environment, a true road profile and collision detection. Subsystems like brakes or tires have also been modeled. Both test maneuvers have been repeated with the developed multibody model in the simulation environment using inputs that have been measured experimentally. Selected simulation variables have then been compared to their experimental counterparts provided with a confidence interval that characterizes the field testing process errors. The results of the comparisons have then been interpreted to extract useful guidelines to build real-time vehicle multibody models. Once a real-time vehicle model is validated, it not only raises the possibility to be used in hardware or human-in-the-loop applications but also in on-board stability controllers. Nowadays simplified vehicle models coming from the classical vehicle dynamics theory are commonly employed in on-board stability controllers. At least, he shows the developed implementation of the Extended Kalman filter, a common state observer for non-linear systems, with multibody models and, after that presents several new implementations using this filter and other filters coming from the family of the sigma-point Kalman filters.

Summarizing, in this context, an X-by-wire car prototype has been built and provided with a data acquisition system. Both are described in § 3. On the other side, through natural coordinates and a self-developed multibody model formulation, a

simulator of the prototype has been developed.

The present paper deals with the current stage of the development, when the need of a synchronize communication interface between the simulator and the real prototype has begun meaningful. Indeed, send a coherent data in term of time is an unavoidable condition to make available the execution of a real-time simulation on-board; moreover, of this way, different behaviours of the virtual prototype and the real prototype gain evidence. The on-board execution is an essential step to extend the utility of the simulator outside the scientific field. In fact, benefits from an application point of view, in the automotive fields, are several, for example to improve security issues: not only (or no longer) a passive system which can execute the commands necessary to bring the vehicle back under control *after* the detection of unstable handling (like all the common equipments), but a system which is able to *prevent* an event and takes decisions in real-time. At least, thanks to the communication interface developed and to the on-board simulation execution, it is possible trough a state observers technique to fix potential simulation errors and to obtain more informations about the vehicle handling than the number of magnitudes monitored. All others future avails and the results obtained are discussed in § 5 exhaustively.

1.2 Objectives

The main object of this work was to establish a communication interface between a prototype multibody model simulator and the real buggy-prototype, in order to execute the virtual model on-board, employing the signals from sensors through a data acquisition system and, most important, paying attention to the synchronization in term of time between the user actions on the real prototype and the values sent to the simulator. *Synchronization* means: the real car and the simulator receive the same inputs at the same time. Obviously, this is an ideal condition, and only an approximation is achievable. The target was to obtain a data transfer time as small as possible, or at least, less than the integration time of the simulator (i.e. 5 ms). Furthermore, it is important to reach the main objective using an as small as possible number of sensors, cause it is essential to obtain a low-cost solution.

The results obtained are very promising, better than expected, with a transfers data time less than 1 ms, monitoring only three magnitudes (i.e. brake pressure, steering wheel angle, accelerator pedal angle).

1.3 Thesis structure

The thesis is structured as follows:

- ***Chapter 1*** is a brief introduction about the present work. More precisely, an overview about the past-works that make this project achievable, the motivations and the thesis structure are presented.

- ***Chapter 2*** defines the basic concepts of multibody system dynamics, which are helpful to contextualize the present thesis and to give a ready-to-use content to understand some key-words used.

- ***Chapter 3*** deals with the hardware instruments and the simulator involved in the present project. The test buggy, the digital acquisition system and the sensors utilized are extensively described. Moreover, the simulator of the buggy is presented, paying attention on the numerical method used and on the code-software arrangement.

- ***Chapter 4*** treats in depth the heart of the present work: the development of the interface between the real prototype, the digital acquisition system and the simulator. An introduction about the aims and critical factors is given, in order to better understand the choices made during the development; after, the simulator input variables are presented, paying attention on the data processing and the configuration of sampling; subsequently, all the strategies available and the strategy chosen for data transfers are explained, discussing the results at the end; lastly, the general structure of the application developed is shown, in order to better understand how the implemented functions work together, and how the buffers system is structured. Moreover, all the sections are provided with the relative implemented code as example.

- ***Chapter 5*** draws the conclusions and the possible future developments.

To complete the thesis with further details, the code developed is collected at the end, in different appendixes.

Chapter 2

Basic concepts of multibody systems

2.1 Definitions

In the book by García de Jalón and Bayo, a multibody system (MBS) (e.g. Fig. 2.1) is defined as follows:

> "*It is an assembly of two or more rigid bodies imperfectly joined together, having the possibility of relative movement between them*" (García de Jalón and Bayo, 1994).

The term *rigid body* implies that distance between any two given points of the body under consideration remains constant at all time, in other words, the body deformation is neglected. Relative motion of the interconnected bodies is kinematically constrained because joints allow one or more degrees of freedom and constraint others. For example, in the case of plane systems, a *prismatic* joint allows one relative translation while a *revolute* one allows one relative rotation. Sometimes, the bodies are not directly connected, but related by force transmission deformable elements, like springs or dampers. Because the variety of joints and interconnection possibilities, a system could

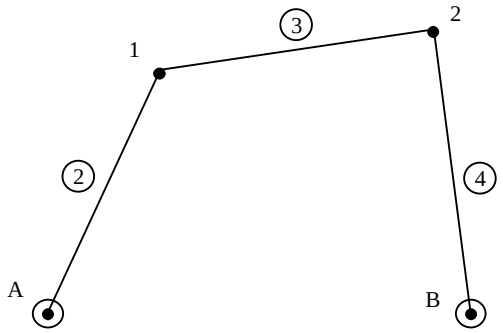

Fig. 2.1: Four-bar articulated quadrilateral

be very complex, even more if it is a *3D* model, as shown in Fig. 2.2, which involves the third dimension.
Overall, the multibody systems are classified as follows:

- Closed-chain: MBS composed of bodies that are connected to other elements in order to create only closed loops. The four-bar mechanism is an example.
- Open-chain[1]: one or more bodies of MBS doesn't create a closed loop, such is a *double pendulum* or a robot-hand.

In order to study a MBS, the following analyses can be performed :

- Kinematic analysis. It deals with the study of system motion independently of the forces that produce it. Only the position, velocity and acceleration of MBS elements are involved, so it is useful mostly to study the trajectory of bodies. Merely, the interaction between geometry and motions is obtained and/or analysed. In a kinematic analysis, a *driving element*[2] must be kinematically prescribed, while the motions of all the other elements are obtained using kinematic constraint

[1]Open and closed chain can be furthermore *simple*, *composed*, *planar*, *three-dimensional*.
[2]More than one driving element can be identified and described.

equations that describe the topology of the system. In some simple cases, the use of trigonometric formulas could be adequate. The main kinematic problems are: *initial position*, *finite displacement*, *velocity and acceleration analysis*, *kinematic simulation* (García de Jalón and Bayo, 1994).

- Dynamic analysis. It deals with the study of the system motion as response to the forces that act on it. Usually, the motion of the system is the unknown factor which must be determined through the analysis. This kind of study can be more difficult than the kinematic one because it involves all kind of forces (i.e external and internal reaction forces, moments) and inertial characteristic parameters (i.e. inertia tensor and mass) of each element. The main dynamic problems are: *static equilibrium position*, *linearized dynamics*, *inverse dynamic*, *forward dynamics* (García de Jalón and Bayo, 1994).

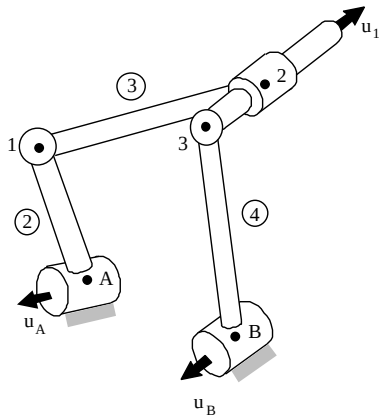

Fig. 2.2: RSCR spatial mechanism

2.2 MBS elements representation

2.2.1 Types of coordinates

The first issue to be considered in order to perform the analyses presented in § 2.1 is the selection of some coordinates which allow to define at all time the position, velocity and acceleration of the system. Several alternatives are available and none of them can be classified as the best or the worst: each method has pros and cons, the choice is highly dependent on the problems and the investigator's final target.

Even though the same multibody system can be described with different types of coordinates, this does not mean that they are all equivalent in the sense that they will allow for formulations that are just as efficient or as easy to implement. In fact, there are differences in computational efficiency and simplicity of implementation when using different sets of coordinates. The different dynamic formulation may also benefit from the characteristics of a particular set of coordinates.

Consequently, the first important question encountered at the time of modeling the motion of a multibody is that of finding an appropriate system of coordinates. The

choice is between a set of *independent coordinates* (ICs) or *dependent coordinates* (DCs). The number of independent coordinates is equal to the system *degrees of freedom* (DOF), and so it is the smallest possible. Conversely, the number of dependent coordinates is greater than the DOF and the relations among them are defined through *constraint equations*, which are usually non-linear and play a main role in the kinematics and dynamics of multibody systems. The constraint equations number r can be obtained through the algebraic difference between the coordinates number n and the degrees of freedom g, as shown by Eq. (2.1).

$$r = n - g \tag{2.1}$$

Using IC generally it is not possible to define unequivocally the position of all MBS elements, as shown in Fig. 2.3: for a certain value of the angle φ (independent coordinate) two solutions of the position problem are possible for the elements 3 and 4. Anyway, for some particular applications, independent coordinates can be very useful to describe with a minimum data set the actual velocities or accelerations and small variations in the position. In addition, they may lead to the highest computational efficiency. Instead, using a set of DCs the position of each and every element is achievable and they may lead to the most easy implementation. The conclusion is that the DCs are much more suitable to describe a MBS while the ICs are not an acceptable solution. In fact, dealing with complex multibody systems, it is preferable an easy and fast implementation. Three main types of dependent coordinates are available:

- *Relative* coordinates. They were the first ones used in the general purpose planar and three-dimensional analysis programs. Relative coordinates define the position of each element in relation to the previous element in the kinematic chain by using the parameters or coordinates corresponding to the relative degrees of freedom allowed by the joint linking these elements (García de Jalón and Bayo, 1994). Relative coordinates make up a system with a *minimum number* of dependent coordinates. This involves a good numerical efficiency. On the other hand, the mathematical formulation can be more involved, cause the absolute position of an element depends on the positions of the previous elements in the kinematic chain; they lead to equations of motion with matrices that, although small, are full and sometimes expensive to evaluate; they require some processing work and post processing.

- *Reference point* (or Cartesian) coordinates. They try to remedy the disadvantages of the relative coordinates by directly defining, using three coordinates or parameters, the absolute position of each one of the element (the so called *reference point*, which often is the center of gravity) with two Cartesian coordinates, and by determining with an angle the orientation of the body in relation to a system of inertial axes. The reference point coordinates require a much larger number of variables that the relative coordinates and do not take into account at all if it is an open chain configuration or not. This means that for some particular cases, and from numerical efficiency point of view, reference point coordinates may not be the most suitable ones. An advantage of these coordinates is that the matrices appearing in the equations of motion are spare, meaning that they have very few non-zero elements. On the other hand, the apparent disadvantages are their large

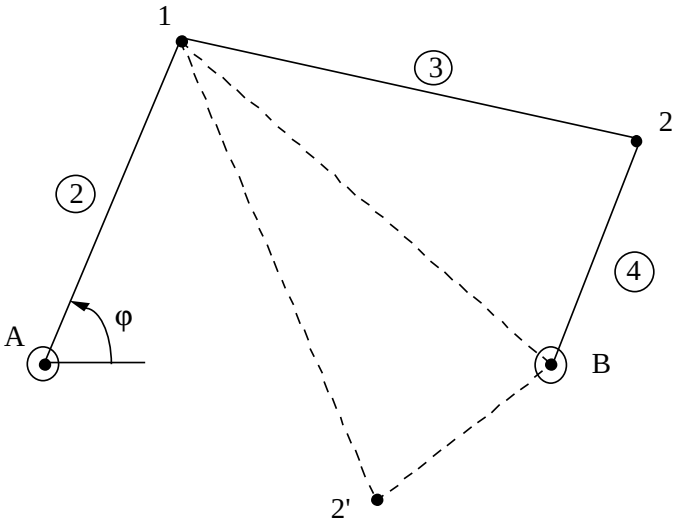

Fig. 2.3: Solutions of the position problem in a four-bar mechanism

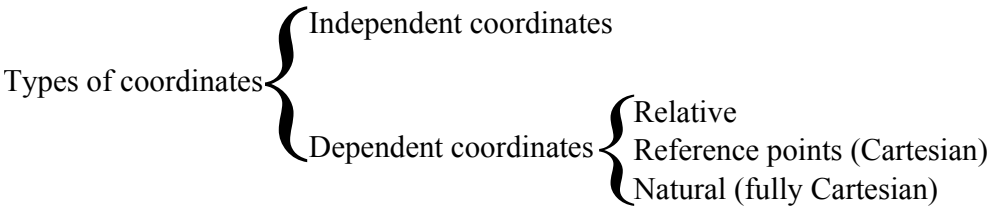

Fig. 2.4: Types of coordinates used in multibody formulation

number and the difficulty to be adapted for a particular topologies such as open kinematic chains.

- *Natural* (or fully Cartesian) coordinates. The prototype MBS under investigation in this paper was modelled using mixed coordinates of natural (almost the model totality) and relative types. For the sake of completeness, the natural coordinates are better defined in § 2.2.2, and also an example is given in the same section. These coordinates, as will be presented in the next section, present numerous advantages from computational point of view.

Anyhow, a comprehensive discussion about all the coordinate systems types is exhaustively treated by García de Jalón and Bayo (1994); Cuadrado (2012); Flores et al. (2008). Figure 2.4 summarizes the previously presented coordinates types. The same types of coordinates discussed above for planar multibody systems also apply to three-dimensional ones. Although the formulation is at times substantially more complicated, the basic concepts hardly differ, therefore the explanations tend to be quite straightforward.

2.2.2 Natural coordinates

The natural coordinates in the case of planar multibody systems are made up of Cartesian coordinates of points, called *basic points*, which are distributed throughout the entire mechanism. They can be considered like as an evolution of the reference point

coordinates in which the points are moved to the joints or to other important points of the elements, so that each element has at least two points. It is important to point out that since each body has at least two points, its position and angular orientation are determined by the Cartesian coordinates of these points, and the angular variables used by reference point coordinates are no longer necessary. This will simplify the formulation of the constraint equations cause the points can be shared at the joints. The criteria to chosen the points proposed by García de Jalón and Bayo (1994) are the follows: at least two basic points for the motion to be defined for each element; basic points should be sheared at the revolute joints; in addition to the basic points that model the body, any other important point of any body can be selected as a basic point, and its coordinates would then automatically become part of the set of unknown variables; each prismatic joint P links two bodies, and the two basic points at one of these determine the direction of the relative motion. Although one of the basic points of the other body can be located on the segment determined by the two basic points of the first one, this is not absolutely necessary.

The number of natural coordinates tends to be an average between the number of relative coordinates and the number of reference point coordinates. The reason for the decrease in the number of coordinates is due, on one hand, to the elimination of the angular coordinates and, on the other hand, to the sharing of the basic points by two or more bodies. Thus, they have the advantage of describing the position of bodies with a reduced number of unknowns. Finally, it should be pointed out that perhaps the most important advantage of natural coordinates is their easy formulation and implementation from a programming standpoint, cause the constraint equations and their Jacobian matrix are very easy to evaluate. These advantages can be translated into some reductions in calculation times, which is very useful for a real-time application.

Figure 2.5 displays a four-bar articulated quadrilateral described through natural coordinates. The variables which define its geometric configuration are shown in Eq. (2.2).

$$q = \{x_1, y_1, x_2, y_2\}^T \tag{2.2}$$

The system has only one degree of freedom and four dependent coordinates. Therefore, according to Eq. (2.1), three constraint equations are required.

$$(x_1 - x_A)^2 + (y_1 - y_A)^2 - L_1^2 = 0 \tag{2.3}$$
$$(x_2 - x_B)^2 + (y_2 - y_B)^2 - L_3^2 = 0 \tag{2.4}$$
$$(x_2 - x_1)^2 + (y_2 - y_1)^2 - L_2^2 = 0 \tag{2.5}$$

Constraint equations (2.3), (2.4), (2.5), are obtained by imposing a null variation of the elements length, or in other words imposing the *rigid body condition*. It may be seen that they are non-linear equations (quadratic in this case). It clearly appears that natural coordinates considerably simplify the formulation of constraint equations along with the fact that points can be shared at the joints.

In the case of three-dimensional multibody systems, the natural coordinates describe the position of each element by means of the Cartesian coordinates of the basic points distributed throughout the elements and by means of the Cartesian components of several *unit vectors*. Each element of the system should have a sufficient number of points and vectors linked to it; so that their motion completely defines that of

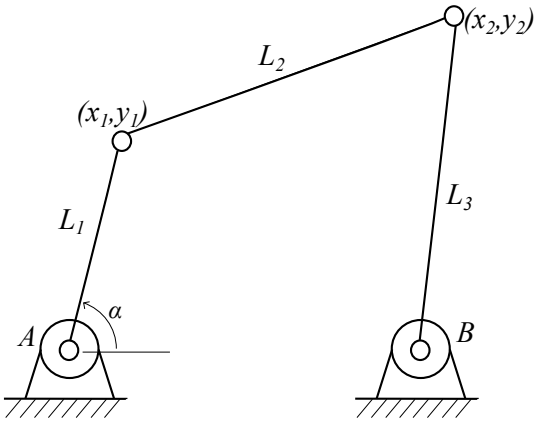

Fig. 2.5: Four-bar articulated quadrilateral in natural coordinates

the element. Also in the case of three-dimensional multibody systems, the natural coordinates provide a simple formulation and implementation. The complexity of the mathematical formulation increases linearly when moving from 2D to 3D applications, cause it only suffices to add new points to the model and a new term to the equations coming from the scalar product of vectors. So, as in the case of planar multibody systems, the need for preprocessing and postprocessing is minimal when using natural coordinates. At least, in the case of 3D multibody systems, the constraint equations with natural coordinates also originate in two ways: from the rigid body condition of the element and from some of the kinematic joints that exist among them (García de Jalón and Bayo, 1994).

Summarizing, the most interesting features of the natural coordinates, that make their representation useful for simulation multibody formulation and for real-time applications are:

- Natural coordinates are composed of purely Cartesian variables and therefore are easy to define and to represent geometrically.
- The rotation matrix of rigid body whose motion is described with natural coordinates is a linear function of these coordinates; while with the reference point coordinates the rotation matrix is a quadratic function of Euler parameters and a transcendental function (sine and cosine) of Euler angles.
- Natural coordinates can be defined at the joints and then shared by contiguous bodies, contributing to define the position of both bodies and significant simplifying the definition of joint constraint equations. At the same time, the total number of variables is kept moderate.
- A single set of variables define the geometry and the position of the body directly in the global reference frame.
- The constraint equation that arise from the rigid body and joint conditions are quadratic (or linear); so their Jacobian matrix is a linear (or constant) function of the natural coordinates.
- Natural coordinates can be complemented easily with relative angles and distances defined at the joints to yield a mixed set of Cartesian and relative coordinate.

Driving an angles or a distance, and defining forces and/or torque in joints become rather straightforward. Relative coordinates also simplify the task of defining the constraint equations for some particular joints.

- The design variables (e.g. length, angles) appear explicitly in the constraint equations.

2.3 Introduction to MBS equations of motion

In order to determine the motion of an entire system, it is necessary to establish the dynamic equilibrium condition that leads to a system of second order differential equations generally called the *equations of motion*.

At current state of the art, many methods are available to derive the equations of motion. The two more popular are: Newton-Euler's method (Nikravesh, 1988), and Lagrange's method (Shabana, 1989). The main difference between them is that with the Newton-Euler formulation, all forces, which are acting on or within the system, must be considered. This is particularly cumbersome when dealing with a system of interconnected bodies and when many of these forces are forces of reaction and constraint, which generally not concern when wanting only to describe motion, but which constitute additional unknowns to the problem. In the Lagrange method, instead, all the workless forces and many constraint forces are automatically eliminated. As consequence the method can be easy, methodical and suitable for an implementation. Furthermore, the derivation is simpler and more systematic than in the Newton-Euler's method. In spite of this, the generated analytic model could be very complex and the selection of the correct coordinates is not a simple step. The Lagrange equations for a constrained mechanical system, described through a set of dependent coordinates has the form expressed in Eq. (2.6).

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L}{\partial \dot{\mathbf{q}}}\right) - \frac{\partial L}{\partial \mathbf{q}} + \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\lambda} = \mathbf{Q}_{\mathrm{ex}} \tag{2.6}$$

Where $L = T - V$ is the Lagrangian function, $\mathbf{q}$ are the dependent coordinates, $\mathbf{\Phi}_{\mathbf{q}}$ is the Jacobian matrix, $\boldsymbol{\lambda}$ contains the Lagrange multipliers. The kinetic energy of a multibody system can be written as shown in Eq. (2.7), where the mass matrix $\mathbf{M}$ is constant as long as all the bodies have at least two points and two non-coplanar unit vectors or an equivalent structure. Otherwise, the mass matrix is dependent on the position $\mathbf{q}$. For general case, in which the kinetic energy depends on $\mathbf{q}$, Eq. (2.6) becomes Eq. (2.8), where $\mathbf{Q}_{\mathrm{ex}}$ is the vector of external forces and L the Lagrangian.

$$T = \frac{1}{2}\dot{\mathbf{q}}^{\mathrm{T}}\mathbf{M}(\mathbf{q})\dot{\mathbf{q}} \tag{2.7}$$

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\lambda} = \mathbf{Q}_{\mathrm{ex}} + L_{\mathbf{q}} - \dot{\mathbf{M}}\dot{\mathbf{q}} \tag{2.8}$$

At time of formulating the equations of motion, it is possible to do it with both dependent or independent coordinates. There is not a consensus among the experts as to which method is the best for all cases. A method can be advantageous over another under certain conditions and vice versa. Usually, the Lagrange's method is applied

with a set of dependent coordinates, while the Newton-Euler's approach with a set of independent coordinates.

One advantage of the independent coordinates is precisely an important reduction in the number of equations to be integrated. Most important is the disappearance of the instability problem in the integration of the constraint equations using ODE solvers. However, this has a price in terms of computational effort since the position and velocity problems need to be solved after the function evaluations. Some of the numerical integration algorithms and in particular the more stable implicit algorithms are difficult to implement. In addition, the formulation and implementation of these methods become more involved than those which use dependent coordinates. At least, one important point is the choice of the right set of independent coordinates.

Instead, when using a set of redundant/dependent coordinates (e.g. natural coordinates presented in § 2.2.2), the motion of MBS is described by *differential algebraic equations* (DAEs) which consist of second-order differential equations plus algebraic constraints (Nikravesh, 1988). DAEs present some differences with respect to *ordinary differential equations* (ODEs) obtained trough a set of independent coordinate. In general, a set of first-order ODEs can be expressed as shown in Eq. (2.9).

$$\mathbf{F}(t, \mathbf{x}, \dot{\mathbf{x}}) = \mathbf{0} \tag{2.9}$$

The derivatives of the dependent variables $\mathbf{x}$ are expressed explicitly in terms of the independent variable t and dependent variables $\mathbf{x}$. As long the vector valued function $\mathbf{F}$ has sufficient continuity, a unique solution can be found for an initial value problem where the values of the dependent variables are given at a specific value of the independent variable. Conversely, considering DAEs, the derivatives are not (in general) expressed explicitly. In fact, derivatives of some of the dependent variables typically do not appear in the equations.

The general form of DAEs system for a constrained mechanical system is expressed by Eq. (2.10), which represents the Lagrange's equation, and by Eq. (2.11), which represents the constraints. $\mathbf{M}$ is the mass matrix, $\mathbf{\Phi_q}$ is the constraints Jacobian matrix, $\boldsymbol{\lambda}$ are the Lagrange multipliers, $\mathbf{Q}$ is the external forces vector, $\mathbf{\Phi}$ are the constraints. Its solution yields the values of n_d dependent coordinates as well as the m Lagrange multipliers.

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{\Phi}_\mathbf{q}^\mathrm{T}\boldsymbol{\lambda} = \mathbf{Q} \tag{2.10}$$

$$\mathbf{\Phi}(\mathbf{q}, \mathrm{t}) = \mathbf{0} \tag{2.11}$$

This kind of equations can be solved through sundry ways, one of them is to transform the set of DAEs into a set of ODEs through one of the methods presented in the literature. The preferred method from a computational point of view is the Penalty method (Bayo and Ledesma, 1996). In fact, the Lagrange multipliers technique allows for the solution of the dynamic problem at the expense of solving for an augmented set of (n_d+m) unknowns: $\boldsymbol{\lambda}$ plus $\mathbf{q}$. The penalty formulation instead, eliminates the Lagrange multipliers from the equations of motion and leads to a set of n_d ordinary differential equations with $\ddot{\mathbf{q}}$ as the only unknowns. This method is very interesting cause it has been successfully extended to real-time dynamics within the context of fully Cartesian coordinates (García de Jalón and Bayo, 1994). Unfortunately, Penalty method bring forth the problem of choosing the right penalty number. While large penalty values will

ensure convergence to the constraint within a tight tolerance, those values may also lead to a numerical conditioning problems and develop round-off errors. It is therefore important that the analyst be supplied with a method that converges, regardless of the size of the penalty values, to the right solution within specified tolerances in the constraints. To this end, it is possible to extend the augmented Lagrangian method commonly used in optimization analysis to improve the numerical conditioning of the proposed penalty equations. The resulting equations, shown in Eq. (2.12) yield the *augmented Lagrangian (AL) formulation* (García de Jalón and Bayo, 1994), where the penalty terms are zero if the constraints are satisfied. In this method, in order to avoid using explicitly Eq. (2.11), the Lagrange multipliers are calculated iteratively, as shown in Eq. (2.13). This last equation represents the progressive introduction of forces that help to fulfill better the constraints of Eq. (2.11). Finally, the iterative process of Eq. (2.13) can be introduced in Eq. (2.12), leading to Eq. (2.14).

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\alpha}(\ddot{\mathbf{\Phi}} + 2\boldsymbol{\omega}\boldsymbol{\zeta}\dot{\mathbf{\Phi}} + \boldsymbol{\omega}^2\mathbf{\Phi}) + \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\lambda}^* = \mathbf{Q} \tag{2.12}$$

$$\boldsymbol{\lambda}^*_{i+1} = \boldsymbol{\lambda}^*_i + \boldsymbol{\alpha}(\ddot{\mathbf{\Phi}} + 2\boldsymbol{\omega}\boldsymbol{\zeta}\dot{\mathbf{\Phi}} + \boldsymbol{\omega}^2\mathbf{\Phi}) \quad \text{with } \boldsymbol{\lambda}^*_0 = \mathbf{0} \tag{2.13}$$

$$(\mathbf{M} + \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\alpha}\mathbf{\Phi}_{\mathbf{q}})\ddot{\mathbf{q}}_{i+1} = \mathbf{M}\ddot{\mathbf{q}}_i - \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\alpha}(\dot{\mathbf{\Phi}}_{\mathbf{q}}\dot{\mathbf{q}} + \dot{\mathbf{\Phi}}_t + 2\boldsymbol{\omega}\boldsymbol{\zeta}\dot{\mathbf{\Phi}} + \boldsymbol{\omega}^2\mathbf{\Phi}) \tag{2.14}$$

Where i is the index for the iterative process, $\mathbf{\Phi}$ are the constraints, $\mathbf{M}$ is the mass matrix, $\mathbf{\Phi}_{\mathbf{q}}$ is the constraints Jacobian matrix, $\boldsymbol{\lambda}$ and $\boldsymbol{\lambda}^*$ are the Lagrange multipliers, $\mathbf{Q}$ contains the external forces, the velocity-dependent inertia forces and those obtained from a potential, $\mathbf{\Phi}_t$ is the partial derivative of the constraint with respect to time and $\boldsymbol{\alpha}$, $\boldsymbol{\zeta}$ and $\boldsymbol{\omega}$ contain the penalty factors, the dimensionless damping ratios (usually$\cong$1) and the natural frequencies for each constraint. At first, this procedure might seem to be at a disadvantage since an iteration process and thus extra computation are required. However, the extra numerical effort is practically insignificant, since an iterative procedure is usually necessary to solve a system of non-linear differential equations. A major advantage obtained in return for this additional computation is that the analyst does not have to be concerned with the value of the penalty number that simultaneously assures convergence and avoids round-off errors. Moreover, the numerical integration algorithm has the advantage of solving a set of n_d equations as compared to $(n_d + m)$ needed by the Lagrange multipliers method. Constraint stabilization is implicitly considered within the algorithm and is implemented more simply than the methods that use independent coordinates. The penalty formulation has the advantage over the formulations in independent coordinates, in that the appearance or disappearance of constraints can be done accommodates automatically without changing the coordinates. This in turn avoids the restarting procedure of the numerical integrator. The penalty formulation is also more suitable when the multibody system goes through a singular position. In these cases the Jacobian matrix changes its rank, and the use of independent coordinates requires a sudden change of coordinates. Unless special provisions are made, the formulation in independent coordinates and even the Lagrange's equations in dependent coordinates tends either to crash the simulation or introduce sudden large errors. However, with the penalty formulations, the term $(\mathbf{M} + \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\alpha}\mathbf{\Phi}_{\mathbf{q}})$ of equation Eq.(2.14) is free of singularities and the integration becomes be very stable under these circumstances. This fact also makes the penalty formulation go through kinematic singular position without problems, an advantage not shared by the classical Lagrange's method with Baumgarte stabilization (Baumgarte, 1972).

From all the considerations explained emerges that the Lagrange's method, a set of dependent coordinates and AL technique are the best solutions when dealing with a multibody system implementation and with the development of a multibody real-time simulator, in spite of the Newton-Euler's method in dependent coordinates. A review of advantages and disadvantages of different formulations can be found in Nikravesh (1988); García de Jalón and Bayo (1994), and in the most recent book by Arnold and Schiehlen (2009).

2.4 Multibody systems simulator and real-time concept

A multibody simulator is a software widely used nowadays in most engineering fields, which is able to simulate a large variety of MBS. First of all, the difference between a 3D animation (such is a video-game) and a MBS simulator must be clarified: the multibody simulator is able to *analyse kinematic and dynamic aspects* of the system under consideration; conversely, a 3D animation is only roughly able to *elaborate kinematic aspects*.

Traditionally, the kinematic and dynamic analyses of multibody systems were undertaken by assuming heavy simplifications, for example, that the bodies were perfectly rigid without any influence due to joints behaviour. For simple cases, analytic and/or graphical solutions were possible. Nowadays, because the improvement of technology and increasing necessity to study complex systems behaviour (e.g. cost reduction, security enhancement, high-level optimization) computer aided assistance is unavoidable. Fortunately, this requirement was/is supported by a continuous improvement of computers and, especially, of numerical methods.

The need to resort to a computer-aided software is due to the hard and complex mathematical instruments involved in a MBS analysis. In fact, studying a large scale model (e.g. a car prototype, see Fig. 2.6 and Fig. 2.7) involves hundreds bodies linked by different joints. From a mathematical standpoint hundreds of differential equations of motion need to be solved, which is not possible through the analytic way. It can be asserted that *multibody system simulation*, *numerical methods and code programming are closely intertwined*. Flores et al. (2008) present an exhaustive introduction about all computer codes for a large variety of purposes that have been developed during the last fifty years (e.g. ADAMS, DRAM, SIMPACK, LINCAGES, KINSYN). Using computer-aided or self-developed software mandatorily introduces some simplifications and then errors, the first and most evident one being that to compute the variables in the model as a time function on a digital computer, time has to be discretized, and the variable will be available as a discrete series of values, not as a continuous function. The time distance between two adjacent instants is called *time step* or *integration interval*. It is crucial to consider that besides errors due to time discretization, errors are also generated because numerical integration formulas are only approximations of the real integration function. The time step is strictly related to one of the numerous and different purposes (affecting the numerical methods involved and code programming style) of a MBS simulator: *the real-time challenge*. Some words deserve to be spent on this topic in order to understand the actual meaning of "real-time".

The real-time simulation is a special case of conventional simulation, mainly used

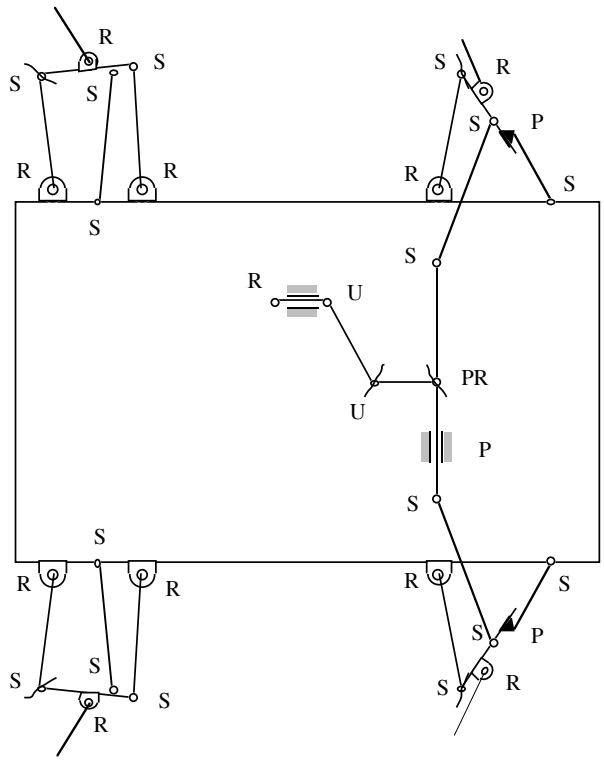

Fig. 2.6: Multibody model of a car suspension and steering system

Fig. 2.7: 3D multibody simulator model of a car suspension and steering system

in *human-in-the-loop* applications (HITL) (e.g. faithful driving simulators, flight simulators) and in *hardware-in-the-loop*[3] applications (HIL) (development and test of complex real-time embedded systems). As already mentioned, solutions of hundreds (for complex systems) nonlinear differential equations are needed to perform an analysis and are accomplished through numerical methods and computer aid. According to García de Jalón and Bayo (1994), in order to work in real-time, a multibody system simulation needs that the analysis time (composed of the integration time plus the time for graphical display), is smaller than the physical time taken by the actual motion of the multibody system. In other words, the software modules must be able to *process all actions according to a predetermined time* (Korkealaakso, 2009). This is the key that controls most the choice of coordinates presented in § 2.2, the algorithms/code implemented in the simulator and, moreover, the interface between the simulator and the data acquisition system developed and presented by the undersigned. Indeed, some numerical methods are available in the literature for each purpose, but only few of them are suitable for real-time simulations. It is not the final target of this paper to present and describe all kind of method, but it must be emphasized that fully Cartesian dependent coordinates and implicit integration methods seem to be suitable for the real-time challenge, specially for their computational *rapidity* and *inexpensiveness*. Anyhow, different solutions are presented from different authors, using implicit and explicit methods. For example Hong et al. (2011) introduced a real-time simulator using an explicit integration method to improve the solving performance for the dynamic analysis of a wheeled vehicle. For further details, the following papers are available in literature: García de Jalón et al. (1986); Nikravesh (1988); Haug (1989); Cuadrado et al. (1997); Bayo et al. (1991); Bae et al. (2000); especially related to the present paper, Naya (2007); Sanjurjo (2011); Pastorino (2012).

Independently of the methods used, it is important to keep in mind that real-time solution requirements often force to simplify the computational model, so that the real-time model must be always considered as a trade-off between efficiency and accuracy.

[3] A HIL simulation must include electrical emulation of sensors and actuators.

Chapter 3

X-by-wire prototype: features of hardware and simulator

Fig. 3.1: X-by-wire prototype

3.1 Hardware configuration

3.1.1 Prototype description

The prototype represented in Fig. 3.1, has been built over several years by Laboratorio de Ingeniería Mecánica (LIM) of Universidad de La Coruña (UDC), and it is very similar to a little buggy. The development is described carefully by Pastorino (2012) and Naya (2007). The choice to build a prototype by themselves entails several advantages:

- The design is completely free, and the prototype could be plan in order to satisfy all the specific necessities (e.g. dimensions, materials, technical features).
- The *cost*, which is always an important parameter, is under control.
- Easy and fast future re-design.
- All the prototype aspects and characteristics are known (e.g. mass of components, materials properties, geometry configuration), and all dynamic/static properties could be easily determined.

On the other hand, the development could be very hard and complex.

The test-buggy is an *X-by-wire* (XBW) vehicle prototype. XBW technology consists in the replacement of traditional control systems with electronic control systems, using electromechanical actuators and human-machine interfaces, such as pedal and steering feel emulators. In the concerned prototype, three by-wire systems have been implemented and described by Pastorino et al. (2010):

1. Throttle-by-wire (TBW) system (Fig. 3.2). The throttle pedal is provided by a geared stepper motor and controlled by a bipolar drive. The control of the

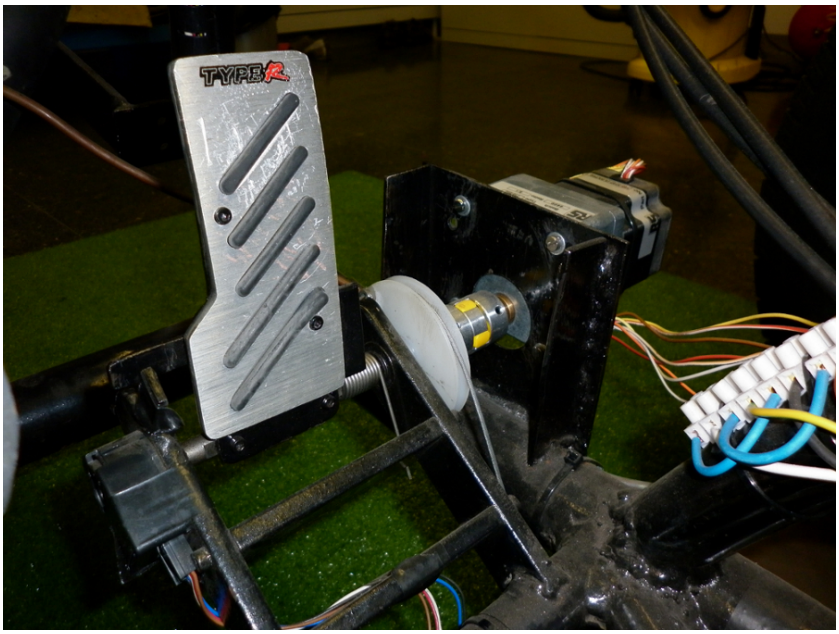

Fig. 3.2: TBW system assembled

throttle angle is performed on its rotation through an encoder (see § 3.1.2 for prototype sensors).

2. Brake-by-wire (BBW) system (Fig. 3.3). The idea is that commands are transmitted electronically through wire. The BBW system, also, actuates on the vacuum servo through an actuator. The control of the brake motion is performed on its position by an encoder, but this solution does not perform a very accurate value of the brake pressure. For this reason, a pressure sensor (see § 3.1.2 for prototype sensor) is mounted in order to obtain a better measurement.

3. Steer-by-wire (SBW) system (Fig. 3.4). It implies that the steering column between the steering wheel, rack and pinion, is eliminated. Thus, two electrical motors realize the steering operations. The first, called *road wheel motor* (RWM), steers the front wheels following the steering wheel angular position provided by an encoder (see § 3.1.2 for prototype sensors). The second one, called *steering wheel motor* (SWM), provides an operator tactile feedback at the steering handwheel. Then the driver can *feel* the resistance of the maneuvers like in the common mechanical systems.

Such configuration gives to the vehicle the possibility to repeat automatically maneuvers, and to be driven without the human control. The systems are controlled by DAS through sensors and controller loop implemented and presented by Pastorino (2012). This characteristic had should be mentioned, because was an unavoidable step to make achievable the present project, making it a future development.

Concerning the technical features, the vehicle is a rear-wheel drive, provided with an internal combustion engine with four cylinders (Chrysler 150 SX) 1600 c.c., a two-barrel carburetor and an automatic gearbox transmission. The following technical specifications are available:

- Engine power of 88 CV (DIN) at 5400 rpm.

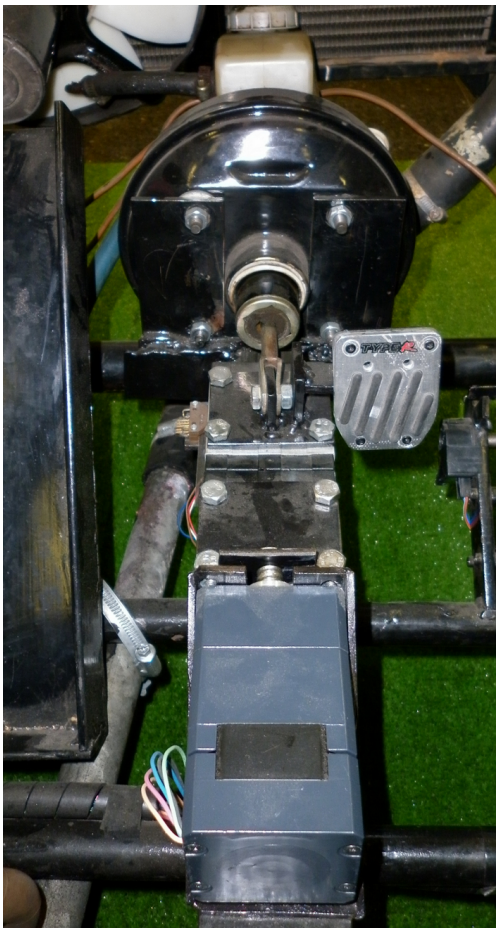

Fig. 3.3: BBW system assembled

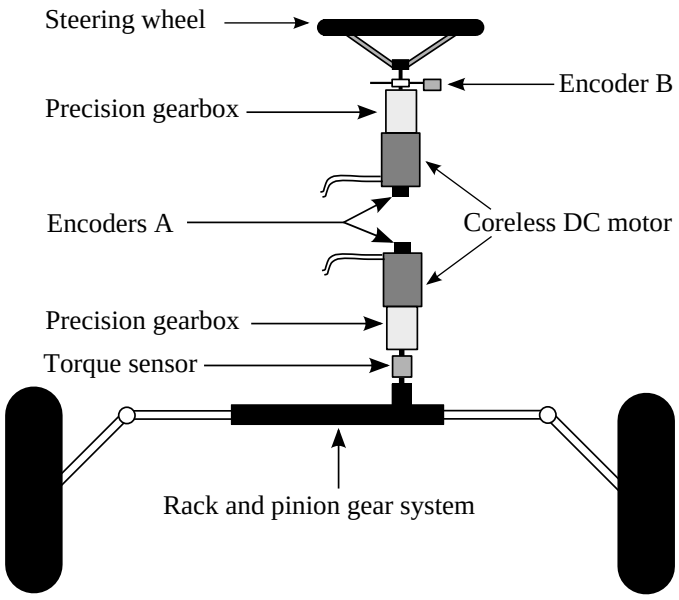

Fig. 3.4: Diagram of SBW system

Gear	Ratio
1st	2.475
2nd	1.475
3rd	1

Table 3.1: Gear ratios of the prototype

- Maximum engine torque of 145 Nm.
- The transmission has three speed automatic gearbox. Table 3.1 summarizes the gear ratios.
- Differential reducer ratio of 3.673.
- Final transmission performance: 29.4 km/h, 3rd gear.

The engine is installed in the rear of the buggy, and linked to the chassis through vibration dumpers that, unfortunately, can't completely dull disturbing vibrations. The frame has been made of tubes. The rear suspension is of MacPherson type, while the front one is of double wishbone type. The tyres are four Michelin 155/80 R13, and the brake system is of disk brake type on rear and front wheels. Moreover, the vehicle comes with an on-board 14'' LCD screen, a personal computer and the *data acquisition system* (DAS). The subsystems are supplied by four batteries: common 12 V battery, one mini-battery of 12 V restricted to the sensors, and two serial batteries of 12 V each one.

3.1.2 Sensors

The vehicle is a very complex system. A lot of sensors are fitted for different purposes, such as to monitor the position, the speed or in general to sense the vehicle dynamics. Table 3.2 summarizes all sensors mounted in the vehicle. A complete description of them can be found in Pastorino (2012); Pastorino et al. (2010); Naya (2007). Related to the present work, only the throttle, brake and steer sensors are presented, cause are the sensors used to monitor the simulator input parameters introduced in § 4.2.

Steer and throttle encoder

A digital optical encoder, is a device that converts motion into a sequence of digital pulses. By counting a single bit, or by decoding a set of bits, the pulses can be converted to relative or absolute position measurements. Encoders have both linear (for displacement measurement) and rotary (for rotation measurement) configurations, though the most common type is rotary. The rotary encoders are manufactured in two basic forms:

- *Absolute encoder*. The main characteristic of this encoder is that a unique digital word corresponds to each rotational position of the shaft.

Measured magnitudes	Sensors
Vehicle accelerations (X, Y, Z)	Accelerometers (m/s^2)
Vehicle angular rates (X, Y, Z)	Gyroscopes (rad/s)
Vehicle orientation angles	Inclinometers (rad)
Wheel rotational angles	Hall-effect sensors (rad)
Brake line pressure	Pressure sensor (kPa)
Steering wheel and steer angles	Encoders (rad)
Engine speed	Hall-effects sensor (rad/s)
Steering torque	In-line torque sensor (Nm)
Throttle pedal angle	Encoder (rad)
Rear wheel torque	Wheel torque sensor (Nm)

Table 3.2: List of the sensors fitted in the prototype

- *Incremental encoder*. The main characteristic of this encoder is that produces digital pulses as the shaft rotates, allowing measurement of relative position of shaft. It consists of two tracks and two sensors, whose outputs are called channels A and B. As the shaft rotates, pulse trains occur on these channels at a frequency proportional to the shaft speed, while the phase relationship between the signals yields the direction of rotation. By counting the number of pulses, and knowing the resolution of the disk, the angular motion can be measured. The channels A and B are used to determine the direction of rotation by assessing which channels *leads* the other. The signals from the two channels are a 1/4 cycle out of phase with each other, and are known as *quadrature signals*. Often, a third output channel, called *index*, yields one pulse per revolution and is useful in counting full revolutions. It is also helpful as a reference to define a home base or zero position.

Most rotary encoders are composed of glass or plastic dotted disk. As radial lines in each track interrupt the beam between a photoemitter-detector pair, digital pulses are produced. Figure 3.5 shows an encoder with spinning codewheel and a stationary mask.

Regarding the prototype, the steer encoder is an *HEDS 5500 A06* from *Agilent*, while the throttle encoder is an *HEDS 5540 A06* model from the same producer. They are incremental type, and provided with two quadrature output channels with optional index pulse. These encoders emphasize high reliability, high resolution and easy assembly, maintaining a very low cost. The operation-range temperature is about from $-40\,^{\circ}$C to $100\,^{\circ}$C. The encoder used to monitor the steering wheel angle is designated as *encoder A* in Fig. 3.4 and shown in Fig. 3.6. It has a resolution of 0.18° with 500 cycle per revolution (CPR), and output pulses resolution setted to 4X. Also the throttle encoder in Fig. 3.7 has a resolution of 0.18° with 500 CPR, and output pulses resolution setted to 4X. The differences between them are the output channel and the mounting system, while both are encoders with metal code-wheels.

Brake pressure sensor

A pressure sensor usually acts as a transducer: it generates a signal as a function of the pressure imposed (the signal is electrical). Pressure sensors can vary drastically in

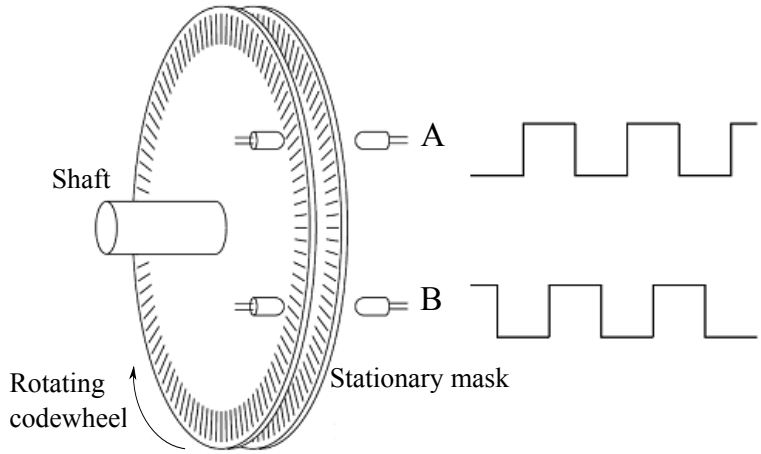

Fig. 3.5: Encoder with spinning codewheel and stationary mask

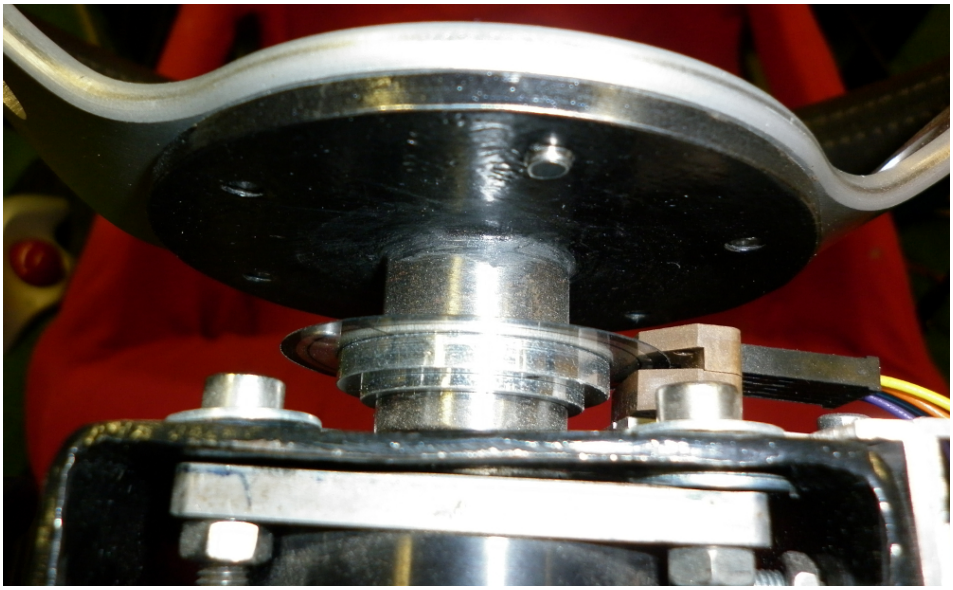

Fig. 3.6: Encoder sensor of steering wheel

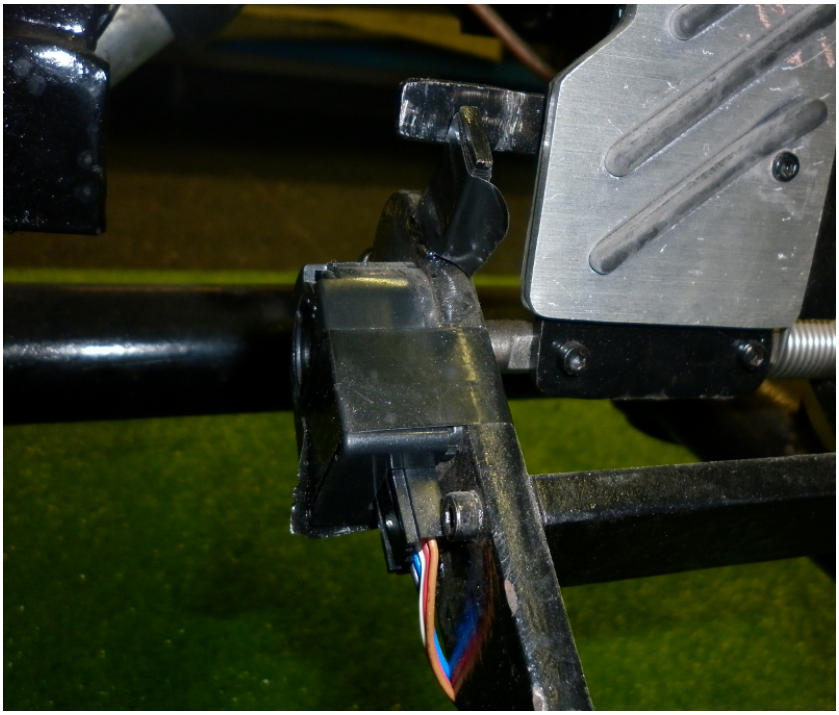

Fig. 3.7: Encoder sensor of throttle

Fig. 3.8: Brake pressure sensor assembled

technology, design, performance, application, suitability and cost. The pressure sensor used on the prototype is a *piezoresistive strain gauge*, which uses the piezoresistive effect of bonded or formed strain gauges to detect the strain due to applied pressure. Common technology types are Silicon, Polysilicon Thin Film, Bonded Metal Foil or, such is the buggy sensor, a Sputtered Thin Film. Generally, the strain gauges are connected to form a Wheatstone bridge circuit to maximize the output of the sensor and to reduce sensitivity to errors.

The concerned pressure sensor, shown in Fig. 3.8, is *3100R0040G0LB00* from *Gems*. The maximum pressure detected is 40 barG, and the output voltage range is 0-5 V. The operation range temperature is from −40 °C up to 125 °C, and the accuracy is ±0.25% FS (Temp O/P ±3% FS).

3.1.3 Data acquisition system

First of all, *data acquisition* is the process of sampling signals that measure real world physical conditions, and converting the resulting samples into digital numeric values that can be manipulated by a computer. As technology has progressed, this type of process has been simplified and made more accurate, versatile and reliable through electronic equipment. A *data acquisition system*, called also DAS or DAQ, typically converts analog waveforms into digital values for processing, and is the hardware what usually interfaces between the signal from transducers and the PC. Nowadays, numerous DASs are available on the market, built for different purposes and with different characteristics. The choice is strictly related to the applications involved and to the final results pursued, such as number and type of sensors or phenomenons to be measured. Another parameter, which can not be overlooked, is obviously the cost. However, being the system always in change (e.g. a sensor could be unthinkable at the beginning of the project and indispensable at the end), it is not possible to own all the informations before purchase. Furthermore, has to be considered which kind of operating system will be used, indeed most DASs have better tools for rapid developing only under Windows or only under Linux (or other OSs). Therefore, DAS for vehicle research purpose must be flexible, modular, expandable and programmable.

Considering these characteristics, a PC-based DAS with PCI host interface has been employed and mounted on the prototype. Figure 3.9 p. 32 shows the connection scheme of on-board DAS, PC and sensors. The main board is a *DAP4200a* (Microstar Laboratories, a) from *Microstar Laboratories* (ML) and is installed in a standard computer. This DAS is designed for high speed data transfers, real-time data sampling and:

- Has an Intel i486 DX4 processor on-board.
- Provides 16 bits A/D converter resolution.
- Works with the 5 V PCI bus for Pentium/Pentium II platforms.
- Comes with 16 M of DRAM on-board memory.
- Transfer data to PC at high rates (up to 3.2 M samples per second).
- Offers low latency for fast response (0.2 ms task time quantum).

- Offers sampling period resolution to 100 ns.
- Samples or updates the digital section at up to 1.66 million values per second.
- Samples analog inputs at up to 769 k samples per second at 12 bits accuracy.
- Updates analog outputs at up to 833 k samples per second each.
- Has expandable analog and digital inputs/outputs.

The on-board multitasking operating system, *DAPL 2000*, runs on the DAP4200a, and ensures that hardware-level differences are transparent. *DAPL 2000* (Microstar Laboratories, c) is a complete software environment for real-time data acquisition. Tasks that perform averaging, triggering, PID control, fast Fourier transforms, filtering, arithmetic operations and many other functions are pre-coded in DAPL. These tasks are chained together to form a complete data acquisition application. More important, user-defined processing commands can be created in `C/C++` language for special tasks and a `C++` library, called *DAPIO32* (Microstar Laboratories, b), is available to interface the main board with the computer applications.
Referring once again to Fig. 3.9, five expansion boards all from ML are mounted:

- *MSXB 037*, which provides sixteen single-ended or eight differential analog inputs with 14 bits A/D converter resolution.
- The second and third boards are *MSXB 056* and have eight analog outputs with 16 bits D/A converter resolution.
- The fourth board, *MSXB 036*, is an high speed counting board used for rotational speed. It has ten independent counter inputs with 16 bits resolution. Two inputs have a maximum input frequency of 100 MHz while the others 6.8 MHz.
- The fifth board, *MSXB 050*, is a quadrature decoder board used for high speed angle counting. It's provided with four input channels with 16 bits resolution and a maximum frequency of 1 MHz for each counter.

The expansion boards are located in a separate rack under the driver's seat, and all the complete manuals can be found in Microstar Laboratories (d). At least, the PC OS is Windows XP Professional (despite that it's not a real-time OS), and the processor is an Intel(R) Core(TM)2 Duo, CPU E8500 @ 3.16 GHz with 2 Gbyte of RAM.

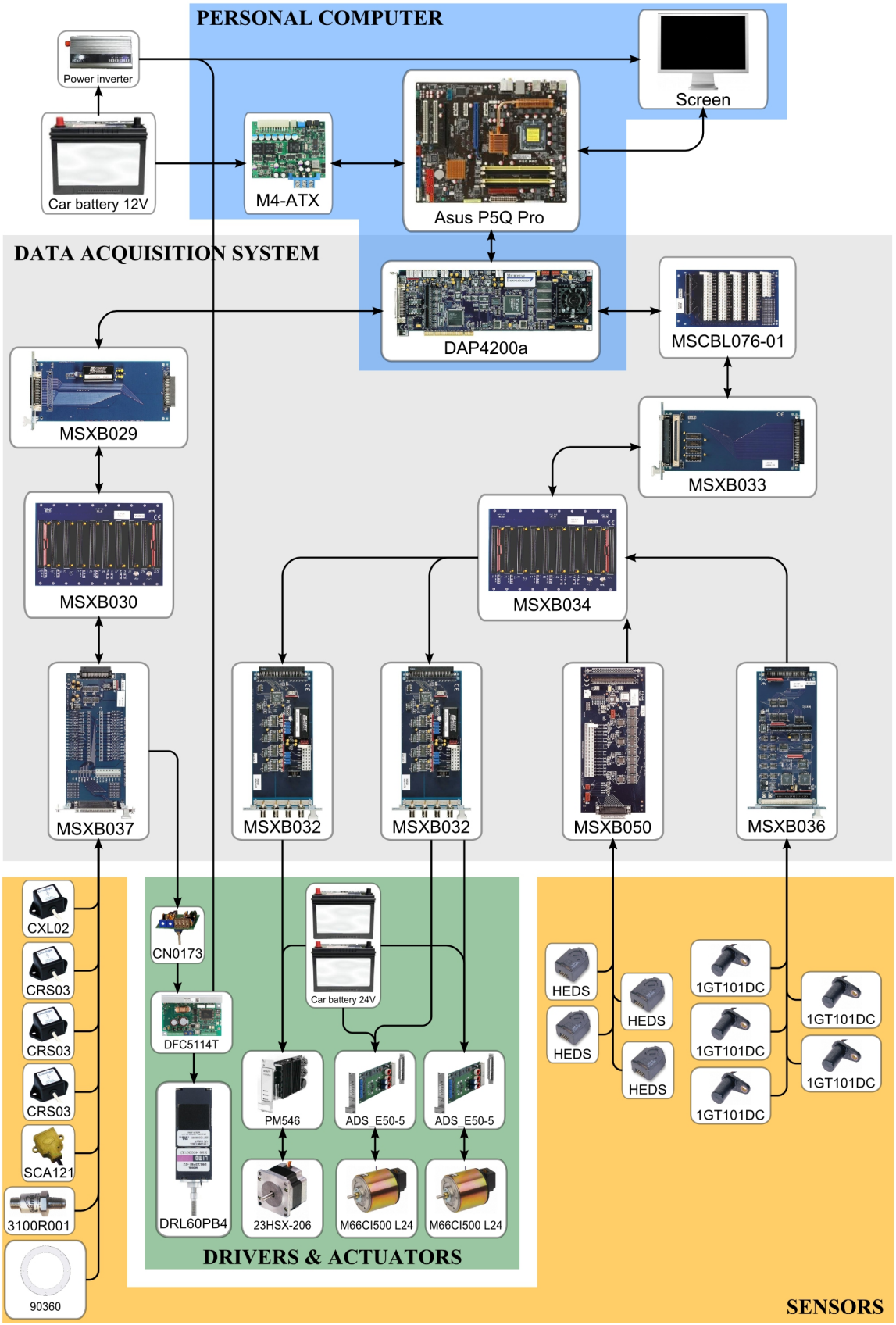

Fig. 3.9: Connection scheme of DAS, PC, sensors, drivers and actuators

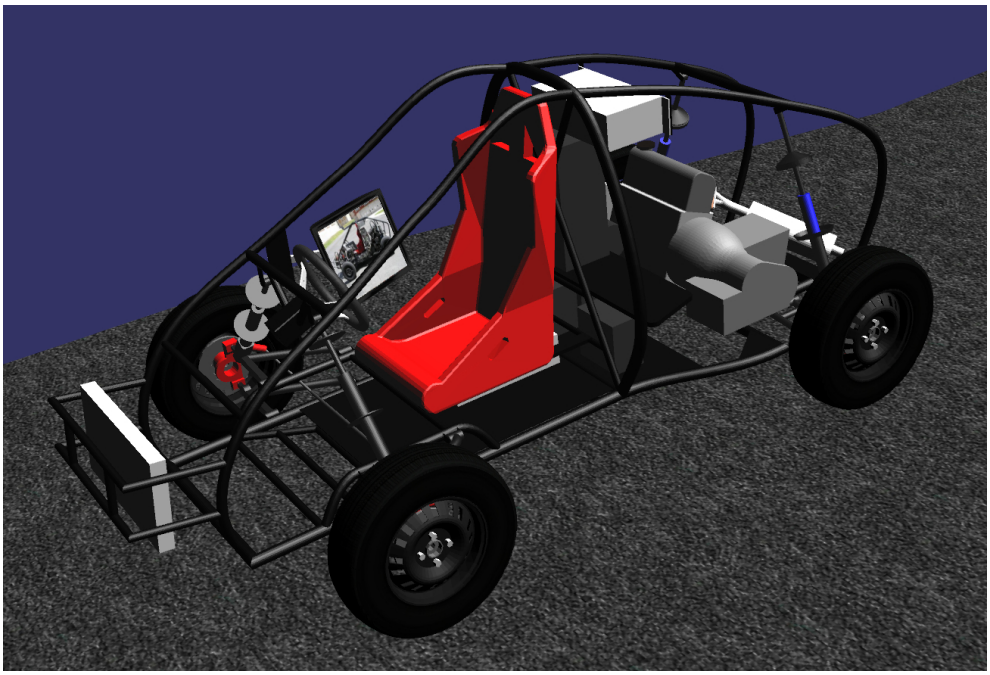

Fig. 3.10: 3D prototype-model of simulator

3.2 Simulator and simulation environment

The MB vehicle model has been completely developed by Laboratorio de Ingeniería Mecánica of Universidad de La Coruña. In order to understand how it works, some basic considerations about the numerical methods, and in general about the simulator, must be done.

3.2.1 Brief numerical methods and analytical considerations

The XBW prototype, presented in the previous sections, has been modelled using fully Cartesian dependent coordinates explained in § 2.2. As asserted in § 2.3, some methods are available to derive the equations of motion, but regarding the simulator the chosen MB formulation is an *index 3 augmented Lagrangian formulation with mass-damping-stiffness-orthogonal projections in velocities and accelerations* (I3AL). The integration scheme is a *predictor-corrector* type: at first, at time t_0 a *prediction* of the system-state of the time t_1 is done through the implicit single-step trapezoidal rule; subsequently, using the prediction and the last system-state known, the iterative Newton-Raphson method is applied in order to obtain the correct system-state of time t_1 (Cuadrado et al., 2001). An integration time-step of 5 ms is imposed.

Concerning the prototype model, it is schematized by eighteen rigid bodies. For every body are defined points plus some vectors in order to completely define its motion, and the mass is known, in order to calculate the chassis total mass. However the obtained value is not accurate, due to some difficulties taking into account the mass of small elements (e.g. wires, screws, fixing elements). For these reasons, the real mass is comparing with the calculated mass, and the difference is taking into account. Finally, with the aid of a CAD software, the center of gravity (COG) and the inertia of each

body (and moreover of the complete chassis) are calculated.

All the details, about the numerical methods employed and the models, are explained in Pastorino (2012). Since these aspects were not objects of this paper, are not treated in depth here.

3.2.2 Code and software arrangement

The simulator consists in a self-developed code implementing and solving the MBM equations of motion. It is composed of some subroutines, programmed in `C/C++`, that covers the 3D outputs, the collision detection, the communication with DAS developed at this work and numerous modules programmed in `Fortran90` (the most widely used language for scientific programming) that cover the vehicle MB. Moreover, a Fortran-library, called *MBSLIMf90*, has been developed by LIM and used. This choice is made in order to take advantage of both languages according to the needs, indeed the `Fortran90` is more suitable for scientific applications, for management matrix of big dimension and in general for easy-develop of numerical methods; while the `C/C++` language is suitable for general purposes. Both languages provide some commands and structures in order to interface them, as consequence it's allowed to call a `C/C++` function in a `Fortran90` programs and vice-versa.
The modules implemented are:

- `CONSTANTES`. In this module are implemented all the constant parameters. Two subroutines are included, the first one, called `inicializa_costantes` initializes the following parameters:
  - `time_step`. This is the time-step of the integration process, setted to 5 ms.
  - `formulation`. It includes the implemented methods: I3AL, matrix R, penalty coefficients or matrix R with Kalman filter.
  - `penaltycoef`. It deals with the coefficients for the penalty formulation.
  - `numerip`, `numeriv`, `numeris`, `numeria`. Which are the numbers of points, vectors, distances and angles.
  - `gravity`. It is the gravity-vector.

  and the second one, called `inicializa_callbaks`, which operates on:
  - `forces`. It is a pointer to the subroutine of forces.
  - `stiffness_damping_matrices`. It is a pointer to the subroutine of matrix R.
  - `ptocolision_normal`. It is a pointer to the subroutine of collisions-detection.
- `CONSTRAINTS_MOD`. It operates on all the model constraints, creates the constraints of all rigid bodies, manages the constraint vector, the Jacobian matrix and all derived parameters.
- `ESTADO`. In this module are included the informations about all the variables used by the solver for every time-step.

- `FORCES_MOD`. It manages all the forces of the model, e.g. contact forces, brakes forces, shock absorbers forces, tyres forces. For each kind of force, a specified function is implemented.

- `Formulations_MOD`. It contains the generic functions which call the desired numerical formulation implemented in the Fortran-library MBSLIMf90.

- `GENERALIZED_FORCES_MOD`. This is the module of the generalized forces which are dependent of the coordinates. This module assembles the generalized force in a global vector and adds their contributions to the rigid/damping matrix.

- `SOLIDOS_MOD`. It contains all the functions responsible of the solids creation. For example it adds solids in the model, adds inertia moments and gravity center to the solids.

In addition, some `C/C++` and `Fortran90` functions and subroutines have been implemented, but only the most interesting for the present work are presented below:

- `main.cpp`. This is the *main* code, the reference tree. Its job is to call the other functions or subroutines implemented, in order to organize the operations and make the simulation start. For example, in the first part, all the functions and files for the graphical display are called, all the modules presented above are declared and called at the right moments.

- `guiado.f90`. It is the subroutine that *operates* on the input and output parameters of the simulator (that is crucial for the present work), it calls the modules and parameters it needs, declares the pointers to input data file, to output one also, and computes all the operations and computations related to the motion of prototype (e.g position, velocity and acceleration of the wheels taking into account eventually gearbox, steer offset, rack, and having as input data the steering wheel angle, velocity or acceleration).

- `lectdatos.f90`. This is the subroutine that *declares* all the input and output data of simulator, and initializes them, if necessary, through functions implemented in the modules presented in the previous page. Moreover, it calls functions which define the points and vectors of each body. The subroutine is executed only one time, at the simulation-start.

All these subroutines-functions-modules are strictly related and interconnected, so the resulting program is very complex and not easy to understand. On the other hand, implementation of different subroutines-functions-modules in different files and specified for an unique need, facilitates vastly maintenance and debugging. Moreover, every code-part can be tested before aggregation to the main project. These considerations justify the programming style chosen. As already mentioned, in order to not complicate the general scheme, all the other subroutines are omitted.

Since the programming languages have no convenient graphical output, an open-source 3D graphical toolkit, *OpenSceneGraph*, has been used to obtain a realistic 3D graphics which reproduce the real environment of the maneuvers road. Particularly, a *true* road profile was realized through a topographical survey. Figure 3.11 shows the real test track, while in Fig. 3.12 a 3D surrounding of test track is presented. As is

possible to see from the figures, the test maneuvers take place in the school campus in Ferrol. Figure 3.10 shows the 3D model of prototype which runs in the simulator.

For further discussions the reader can relate Sanjurjo (2011) for the implemented modules and Pastorino (2012) for analytic aspects and simulator environment (e.g. topographical survey, bodies modelling, subsystems model, collision detection, graphical environment).

Fig. 3.11: Real test track photo

Fig. 3.12: 3D model of test track

Chapter 4

Real car-Simulator communication interface development

4.1 Overall considerations

The communication interface developed at this project faces numerous troubles due to the complexity of the hardware, software and to the variables involved. For these reasons, some overall considerations must be done before to present in detail the interface developed, in order to better understand it. So, a presentation of the previous layout of the simulations management, with respect the present work, is done in § 4.1.1; next the main aims and troubles are presented in § 4.1.2. These remarks will give an essential overview.

4.1.1 Previous layout of simulations management

The previous layout (compared to the present work) of simulations management presents the setting shown in Fig. 4.1: the maneuvers and simulations are performed in two different moments. During a maneuver, the interested variables are sampled through the data acquisition system and stored in a text file. At a later time, the data are reprocessed by a PC: the functions are smoothed and, eventually, the derivatives are calculated and re-smoothed through the aid of *MATLAB*, which is a programming environment for algorithm development, data analysis, visualization, and numerical computation. Since the complete trends of the interested variables, relative to the executed maneuver, are completely known, it is allowed to compute derivatives with simple methods like *forward finite difference method*, obtaining some very good results effortlessly. Lastly, where appropriate, the values are sent to the simulator as input data and the simulation is executed. This allows to state that *no direct communication interface was developed*. For the sake of completeness, another approach is practicable: the X-by-wire systems of prototype presented in § 3, can be not only monitored, but also *controlled* by DAS through its software sampling. More in detail, it is possible to execute a maneuver, save data, and repeat the same identical maneuver without the human participation.

These approaches, from a scientific investigation standpoint, don't present any disadvantages and are suitable for test the simulator, do analyses and especially useful for the development. Moreover, it is not strictly necessary to start the engine for both methods, so most of the operations could be done easily on site. On the other hand, a direct interface raises the possibilities and the applications of the simulator, obtaining a significant step forward from an application point of view, which is exactly the objective of the present work: to make available the execution in real-time of the multibody models on-board.

4.1.2 Aims and critical factors of the communication interface

A direct communication, between DAS and simulator, has the main target to make available to the simulator, when necessary, the *most recent data*. This changes drastically the setting of the simulation, inasmuch the maneuver and the simulation are done in the *same time*. In addition, driving the prototype, it is possible to see the output simulation on the on-board monitor. An overall and simple scheme about how the interface worked is shown in Fig. 4.2: the driver and the disturbances (e.g. road profile) act on the prototype; the interested variables (which are presented in § 4.2) are continuously

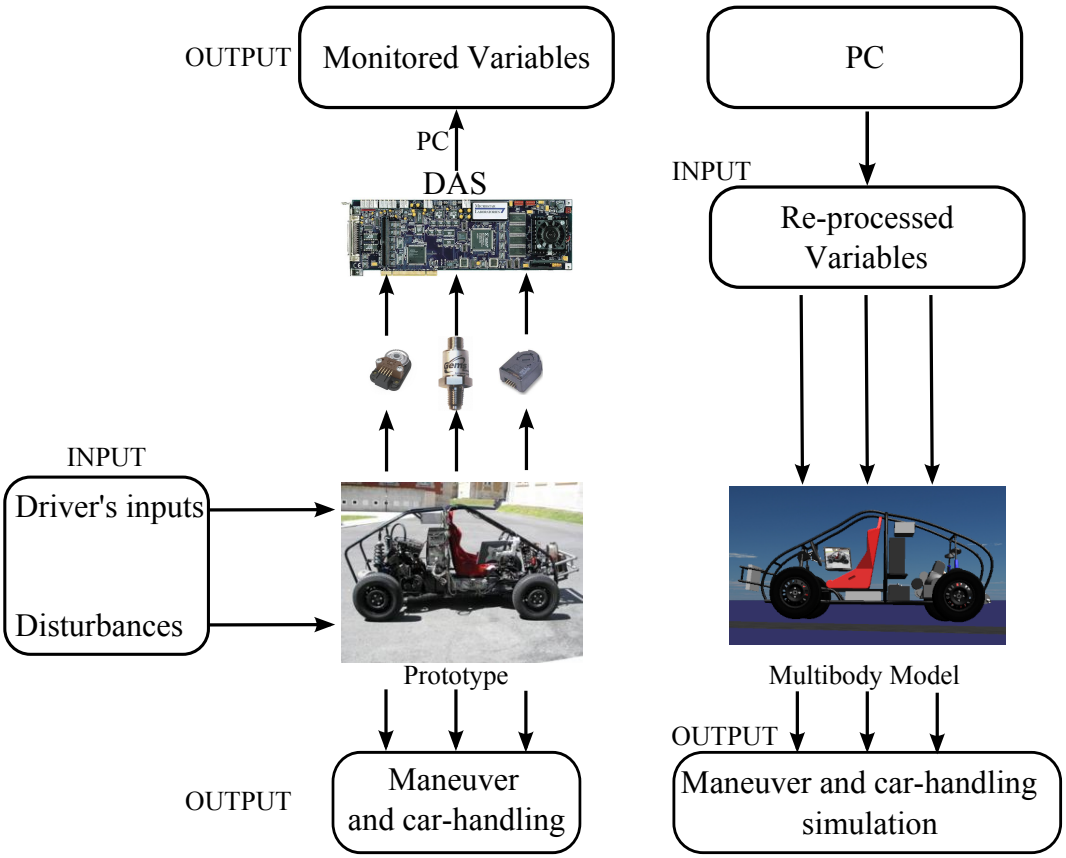

Fig. 4.1: Steps scheme of a simulation without the communication system developed

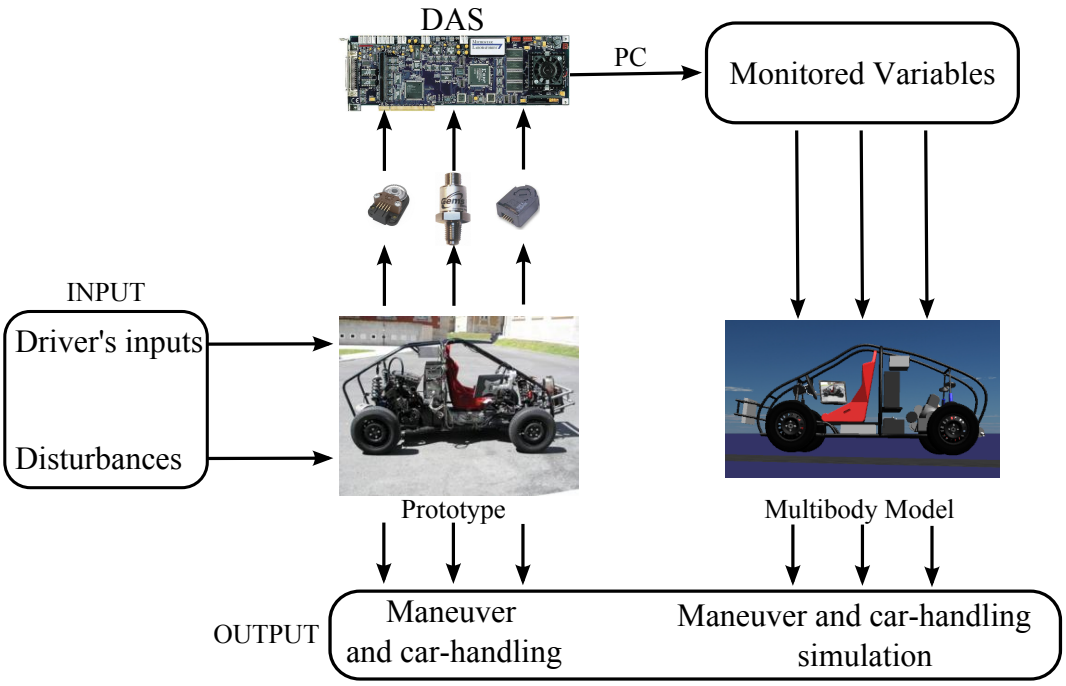

Fig. 4.2: General scheme of the communication system developed

monitored by DAS through the sensors and transferred to the PC; when the simulator needs data, the most recent values are sent to it. These operations, that sound simple at first, after a depth analysis present some difficulties discussed below not easy to be solved:

1. The PC and DAS have different speeds and *times* for data management. In particular, the data acquisition system can generate a lot of values in a small time since offers a sampling period resolution to 100 ns. Conversely, the PC is very slow. Moreover, the DAS generates data predictably, in the manner defined by the user, but the application developed to managed data transfers runs at those moments the operating system scheduler deems appropriate, with an unpredictable delay, depending on other process activity (e.g. graphic displays, disk activity, mouse pointer management). These remarks allow to state that *it is impossible to get a perfect synchronization in term of time between PC and DAS*, while the correct way is to manage data *asynchronously*.

2. Every time step (i.e. integration time), the simulator needs to know the system-state in order to solve the numerical methods implemented. So, when there is the need, for example at time t_1, the data of time t_1 must be sent instantly, without delay. This condition is impossible to achieved due to an inevitable time-cost in performing the operations (e.g. monitor data, transfer data, save data) and to the difficulties to manage data synchronously expressed previously. So, the main aim is to get as close as possible to the aforementioned ideal condition, and it is achieved sending the *most recent values* monitored at the moment of the receiver call since are the nearest available data to time t_1. This is the most important and tricky aspect of all the work. Be able to send the most recent data means be able to supply the simulator with the most faithful state-system in term of time, obtaining a remarkable step forward in the real time direction and in the correctness of the simulations.

3. *Pipes overloading* is an aspect strictly related to the two points discussed previously. A pipe is a communication channel, buffers data from and to the data acquisition processor. It is defined carefully in § 4.3.2, where the communication tools are presented. The pipe must be considered not only like a flow of data, but also like a *stack* which stores unprocessed data, according to *Fist In First Out* (FIFO) methodology shown in Fig. 4.3. This characteristic, combined with the impossibility to flush unwanted data from a pipe, with the DAS high speed sample and with the low PC speed, generates the following two cases:

   - The simulator needs data every time step. During the elapsed time, all unprocessed values that arrive are buffered in the pipe and stacked. Referring once again to Fig. 4.3, suppose that C is the state-system at time t_2 (the most faithful, the most recent data), while A and B are values stacked between t_1 and t_2. At time t_2 the simulator needs data C, but the PC can read values mandatorily observing FIFO approach, so the only data available is A, which is an incorrect old value from a time point of view. Read the most recent data, at first, seems to be impossible. The simulation obtained could be correct, but not related to time t_2, rather to an old state-system.

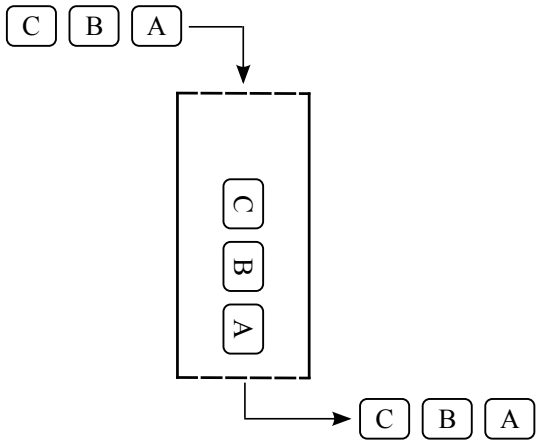

Fig. 4.3: FIFO approach involves that the first value that enters, A, is the first that comes out so the enter sequence is also the output sequence.

As consequence, the maneuver executed are displayed (and simulated) with a delay that could be of few seconds (the best case), or some minutes, depending on PC processing speed and sampling rate: more fast the DAS samples, more the values are stacked. This condition stays surely on the side opposite to the real time concept.

- The worst case occurs when the PC is so slow to process data that the amount of values stacked in the pipe is higher than the maximum admissible. This situation is called *overflow*, and generates an immediate crash of the program. The buffer limit can be reached in a few seconds or in some minutes, depending on PC processing speed and DAS sampling rate.

4. The PC capabilities are limited. For this reason, every operation assigned to it must be simple and, as consequence, inexpensive in term of computational cost. In fact, it has to manage the numerical and graphical outputs of simulator, actions which are obviously unavoidable and that exploit the processor power. So, the interface has to continuously manage and transfer data with the maximum efficiency, in order to obtain almost the same performance achieved reading data by a text file.

5. Lastly, the data processing must be executed without interfering the simulator run. Then, emerges the necessity to deserialize all activities.

Taking into account all the aforementioned remarks, the solution developed, presented in detail in § 4.3.4 and § 4.5, blends together the following aspects:

- Asynchronous data management.
- Empty the pipes in order to avoid the overloading.
- Make available, when simulator needs, the most recent and time-faithful data.
- Deserialize activities to obtain a parallel execution of simulation and data management.

4.2 Simulator input variables

Table 4.1 summarizes the variables involved in the interface developed. First of all, it is important underline that the variables monitored by the DAS are different than those needed by the simulator. The two main reasons are:

- The DAS and its expansion boards, as already explained in § 3.1.3, convert a *continuous quantity* to a discrete time representation in digital form. In other words, the analog input produced by the sensor (e.g. voltage) is converted to a digital form processable by the PC. As consequence, the DAS, from a PC point of view, produces *numbers*. The simulator, on the other hand, needs measurements of specific real physical conditions, for example the brake pressure. For these accounts the digital values must be linked and converted to the real physical conditions monitored before be passed to the simulator.
- Sometimes many ways are available to obtain measurements of the desired physical phenomenon. They have pros and cons, depending on the own needs (e.g. cost, efficiency) one method may be preferable to another. For this reason, a physical condition is not always directly monitored, but could be more suitable to monitor another phenomenon and linked them through a relation.

In the next sections, are presented all the variables needed by the simulator and the relative measurements; moreover it is justified the choices done and the consequences involved. At the end, the DAS configuration of sampling is presented. However, to better understand some aspects, it is appropriate to describe some basic concepts of analog-to-digital conversion:

- Input Volt range ΔV. It is the input electrical tension range admissible by the converter. The tension can vary within a minimum V_{min} and a maximum V_{max} value. It is calculated trough Eq. (4.1).

$$\Delta V = V_{max} - V_{min} \tag{4.1}$$

- Maximum value of the digital output. It is the maximum digital value achievable by the conversion from analog to digital form. It can be easy obtained through Eq. (4.2); n is the ADC resolution in bit, N the maximum number.

$$N = 2^n - 1 \tag{4.2}$$

  It is useful to calculate the digital range that varies within 0 to N (i.e. unsigned integer) or within $-N/2$ to $(N-1)/2$ (i.e. signed integer), depending on the application. Usually the range is approximated by $\pm N/2$.

- Resolution. The resolution Q of the converter is the minimum change in voltage required to guarantee a change in the output digital values. It is computed through Eq. (4.3).

$$Q = \frac{\Delta V}{N} \tag{4.3}$$

For example, considering the DAS of the prototype, n is equal to 16 bits and ΔV admissible is 10 V: the maximum digital value is 65535, while the resolution of the converter is 0.15 mV. The digital values can cover the range 0-65535 or $\pm$32768 (such is the range of the prototype steering wheel encoder).

Monitored variables	Simulator inputs
Throttle pedal angle (deg)	Drive wheels torque (Nm)
Steering wheel angle (deg)	Steering wheel angle (deg)
	Steering wheel speed (deg/s)
	Steering wheel acceleration (deg/s^2)
Brake pressure (bar)	Brake pressure (bar)

Table 4.1: Variables monitored and simulator inputs

4.2.1 Drive wheels torque

The drive wheels torque is an essential magnitude to be sent to the simulator. Different ways are available to monitor it, for example using a specified sensor and obtaining directly the desired magnitude. In despite of this, in the present work the drive wheels torque is estimated using a measurement of the throttle pedal angle and assuming a model for the engine behaviour. The justification of this solution is its appropriateness in an initial development phase. In fact, correlating the throttle displacement to a model of the engine behaviour, it is possible to estimate an approximation of the drive wheels torque and so to drive the virtual-prototype with the real engine off. This is an enormous advantage cause all the numerous tests can be performed in the laboratory, on site, in a very fast and convenient way. The general relation scheme can be summarizes as follows: the throttle pedal angle adjusts the fuel injection which act, in turn, on the engine torque and so on the drive wheels torque. The most critical aspect, presented in depth here, is exactly the correlation between the accelerator pedal and the engine torque. To accomplish this operation, two steps are crucial:

- A model or a function of the engine behaviour must be developed or assumed.
- The throttle pedal angle must be correlated to the engine torque through the model/function assumed.

The development of a model, that faithfully describes and represents the engine behaviour, is a very complex and hard achievement. The advantage of performed tests with the engine off could be incomparable in respect with the cost to develop the model. For this reason, a simply and approximative model must be developed. So, the suitability of the solution assumed is doubtless, but on the other hand the development of an approximative model introduces a very large approximation of the engine torque and of the final drive wheels torque. Anyway, it is an acceptable compromise cause the final target is to develop and test the communication interface in term of transfers of data. The development of a more faithful model, or a choice of another measurement method, should be considered as a future achievement.

In the present paper, the model developed by Naya (2007) has been assumed and is presented in detail below. In general, the engine torque, every instant, is dependent on the following variables:

- The maximum torque of the internal-combustion engine, which is usually a higher value than the actual torque on the wheels.

- The torque of the engine braking.
- The revolutions per minute of the engine.
- The actual gear.
- Fuel injection, that depends on the throttle pedal angle.

Taking into account the engine specifications presented in § 3.1.1, a third degree polynomial has been assumed as a good representation of the maximum engine torque. Three constraint are imposed:

1. Engine torque of 96 Nm at 1000 rpm.
2. Maximum engine torque of 137 Nm at 3000 rpm.
3. A torque value of zero at 6000 rpm.

The equation of the *acceleration torque*, T_a, is obtained in function of the revolutions per minute n and consists in Eq. (4.4). It represents the maximum torque of the internal-combustion engine, when the throttle reaches the maximum displacement.

$$T_a = -9.9444 \cdot 10^{-10} n^3 - 3.2888 \cdot 10^{-3} n^2 + 0.046583 n + 53.7 \tag{4.4}$$

On the other side, when the accelerator pedal is completely released the *engine braking torque*, T_b, has to be considered. Equation (4.5) norms this condition.

$$T_b = -15 \cdot 10^{-3} n \tag{4.5}$$

Figure 4.4 shows the resultant engine torque for the two limit conditions presented above. The revolutions per minutes n, that is an unknown factor, can be obtained indirectly through measurements of the drive wheels angular velocity ω_w. Two different ways are practicable to capture this information:

1. Through a direct measurement of the angular velocity achievable trough the Hall-effect sensor mounted on the prototype.
2. Taking advantage of the simulator itself, exploiting the dynamic model implemented to obtain an approximation of the wheels angular velocity every time step.

In despite of the first solution accuracy, the second one is chosen. Indeed, the use of a direct measurement on the wheels involves to turn the engine on and to drive effectively the real prototype. This condition, obviously, contrasts with the target to drive the virtual model with prototype engine off, on site.

In general, the revolutions per minute n are related to the engine angular velocity ω_e trough Eq. (4.6).

$$n = \frac{60\omega_e}{2\pi} \tag{4.6}$$

Neglecting the transmission losses, the relation between the angular velocity of the engine, ω_e, and of the wheels, ω_w, is shown in Eq. (4.7). Variables T_w and T_e are

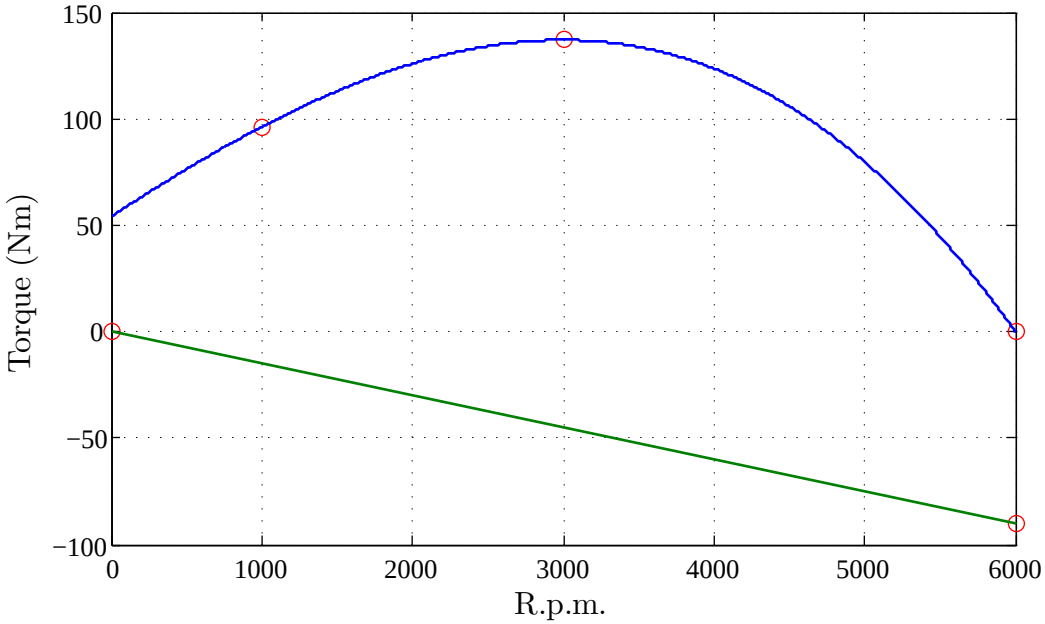

Fig. 4.4: Engine torque curve
The blue curve is T_a, the green one is T_b.

respectively a general drive wheels and engine torque. The parameter τ, computed through Eq. (4.8), is the total speed ratio. It takes into account two contributes: one due to the gearbox ratio ρ_{gb}, which depends on the gear engaged, and one due to the differential speed ratio ρ_d, that assumes the constant value of 3.673. Table 3.1 p. 26 reports all the value assumed by ρ_{gb}.

$$\frac{T_w}{T_e} = \frac{\omega_e}{\omega_w} = \tau \tag{4.7}$$

$$\tau = \rho_{gb}\rho_d \tag{4.8}$$

Taking into account Eq. (4.6) and Eq. (4.7), the revolutions per minute can be defined as Eq. (4.9).

$$n = \frac{60\omega_e}{2\pi} = \frac{60\omega_w\tau}{2\pi} \tag{4.9}$$

Since the angular velocity of the right drive wheel is different from the left one, ω_w has been assumed the arithmetic mean of the two values as shown in Eq. (4.10), where ω_{rw} is the angular velocity of the right wheel while ω_{lw} is the angular velocity of the left one.

$$\omega_w = \frac{\omega_{rw} + \omega_{lw}}{2} \tag{4.10}$$

Equations of T_a and T_b are now completely defined.

The actual engine torque, T_{real}, is a mix of the two limit conditions, depending on the fuel injection and so on the throttle pedal angle. This relation is shown by Eq. (4.11). Where f is a function linked to the accelerator pedal angle.

$$T_{real} = T_a f + T_b(1 - f) \tag{4.11}$$

	Digital form x	f y	Angle form (deg)
Minimum value (x_1, y_1)	0	0	0
Maximum value (x_2, y_2)	420	1	75.5

Table 4.2: Limit values of the function f, and equivalent digital and angle forms

The definition of f, which is a developer choice, is a very important step. This function can vary within a minimum value of 0, when the accelerator pedal is completely released, and a value of 1, when the pedal reaches the maximum angle available. It is evident that, referring to Eq. (4.11), this two values return respectively a real torque equal to T_b and T_a.

The trend of the function must be related to the digital values generated by the encoder (presented in §3.1.2) which is monitoring the accelerator pedal angle. Assuming f as a linear equation, only two values of the function are needed to write the straight line equation. So, the digital values, corresponding to the two limit conditions of $f = 1$ and $f = 0$, have been determined through a test. The minimum digital value registered for a complete release of the pedal was equal to 0; while the maximum digital value registered for the maximum rotation allowed by the hardware configuration was 420. Taking into account the resolution of the encoder, that is 0.18° per pulse, the value can be easy converted in degrees: 76.6°. Table 4.2 summarizes the limit values. However, since the angle value is not strictly necessary, only the digital form is considered.

Having available two points and having defined the trend, the equation has been obtained substituting the four values in the so called two-point form of linear equations shown in Eq. (4.12); where, referring once again to table 4.2, x can be considered as the general actual digital value received from the DAS, y the general actual value assumed by f, (x_1, y_1) are respectively the digital number registered and the value assumed by f for a complete release of the pedal, while (x_2, y_2) are respectively the digital values registered and the value assumed by f for the maximum pedal angle.

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) \tag{4.12}$$

The resulting equation, written in a clear and general form, is Eq. (4.13).

$$f = \frac{\text{actual encoder digital value}}{420} \tag{4.13}$$

The last equation is the link between the digital form of the accelerator pedal angle generated by the encoder and the engine torque. Once f is known, Eq. (4.11) is completely defined, and the actual engine torque T_{real} can be easily computed.

The final step is to estimate the final input variable needed by the simulator, that is the *drive wheels torque*. Taking into account the relation between a general engine and drive wheels torque shown in Eq. (4.7), the drive wheels torque T_{dw} is obtained through Eq. (4.14). Where T_{real} is the actual engine torque and τ the total speed ratio, both already defined previously.

$$T_{dw} = T_{real}\ \tau \tag{4.14}$$

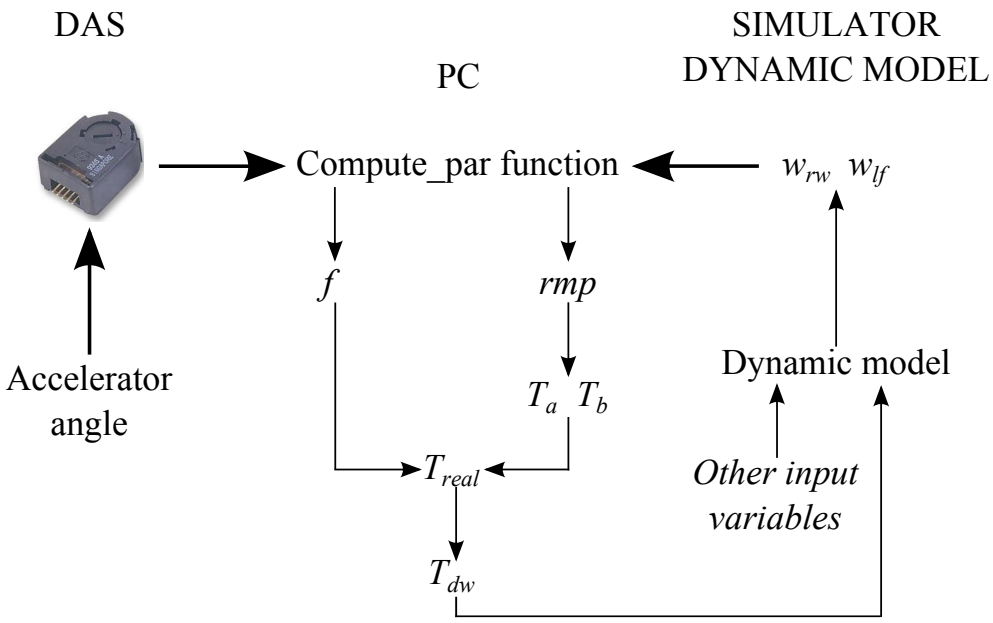

Fig. 4.5: Working scheme of the drive wheels torque estimation

All the equations presented previously are implemented in the communication interface developed, precisely in the `compute_par.cpp` function shown in appendix E. The value of ρ_{gb} is setted to 1.475, which correspond to the 2nd gear, in order to obtain a smooth acceleration that helps to drive the virtual prototype. The gear shift is omitted. Referring to Fig. 4.5, the working scheme can be summarizes as follows: the DAS continuously monitors the accelerator angle through the encoder sensor; when appropriate the value is sent to the PC and managed by the `compute_par.cpp` function. Reading the dynamic model output angular velocity of the wheels (that are called $ap(30)$ and $ap(40)$) the revolutions per minutes is calculated; subsequently, through the equation presented in this section, all the other variables are computed; at the end, the torque of the drive wheels is sent to the simulator as input data; the dynamic model, taking into account the wheels torque generated by the engine motor and the contributes of other variables (e.g. road profile, brake pressure), acts on the virtual prototype run. With the new *dynamic conditions* the angular velocity are recalculated and made available for another estimation of the engine motor torque contribution.

The model of the engine and the estimation of the engine torque using the drive wheels angular velocity of the virtual prototype introduce errors in the final result. Moreover, some limits on the engine model can be inferred, like the omission of the gear shift. Despite all, the solution is suitable for testing the communication interface, and from an application point of view the solution presented makes unnecessary the use of a specific and more expensive torque sensor, maintaining a small number of sensors.

4.2.2 Position, speed, and acceleration of the steering wheel

Three variables, related to the steering wheel, are sent to the simulator: the steering wheel position, that is the rotation angle, the steering wheel rotation velocity and lastly the steering wheel rotation acceleration. Despite this, only the position is monitored through the data acquisition system and the encoder shown in § 3.1.2. The other two

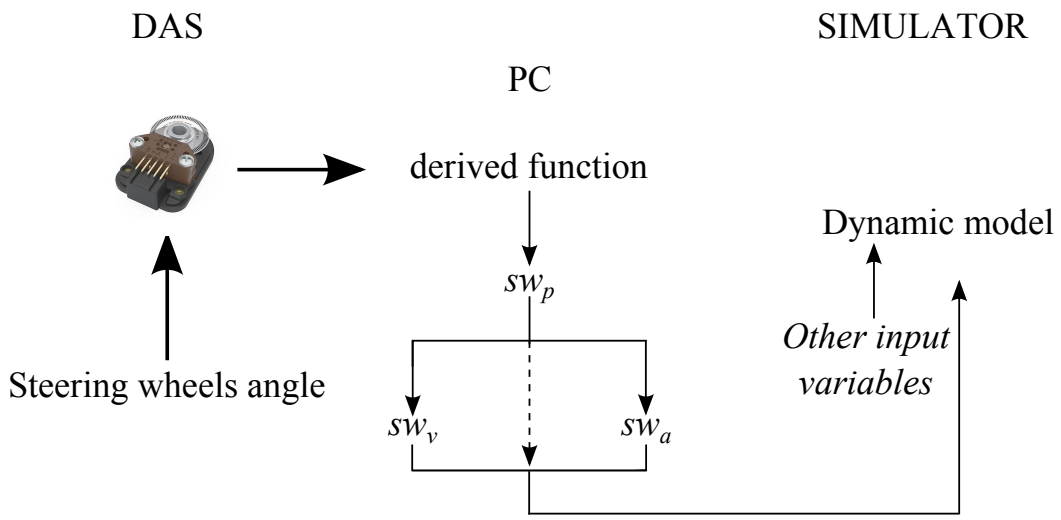

Fig. 4.6: Working scheme of the steering wheel position, velocity and acceleration computation

variables are obtained a posteriori, trough a numerical differentiation method considers only past values. In order to obtain the steering wheel angle position, the digital values produced by the DAS must be related to the angle form. Working with a rotary encoder this operation is very simple. Taking into account the encoder resolution, that is 0.18°, the steering position sw_p is calculated through Eq. (4.15).

$$sw_p = 0.18° \cdot \text{actual encoder digital value} \tag{4.15}$$

So, once the steering wheel digital measurement is transferred to the PC, the function `derived.cpp` shown in appendix D calculates the position, velocity and the acceleration. Subsequent, the three values are stored into a buffer and made available at the simulator. Figure 4.6 draws the working scheme of the `derived.cpp` function.

The numerical differentiation methods used to obtain the first and the second derivatives of the position has been investigated for a long time. In fact, the time discretization of the steering wheel position generates a *stepped* curve shown in Fig. 4.7, although the high encoder resolution. The steps are evident especially if the maneuver is compared to the curve obtained after a smooth operation, shown in Fig. 4.8. Figure 4.9, instead, shows only the deviation within the two curves that has an oscillating trend. Moreover, only past-data are available to estimate the derivatives cause it is impossible to anticipate the user actions. As consequence, the numerical derivatives of this curve, obtained considering only past values, is very *noisy* and wrong. If the simulator receives a too wrong derivatives, the time needed to solve the equations of motion through the integration methods explained in § 3.2.1 increases cause the iterations number to reach the solution steps up. If the final objective is a real-time simulation, it is not allowed an increase in the number of the iterations and, in turn, an increase in the time needed to reach a numerical solution. Rather, a real-time simulation has precisely the opposite aims: low number of the iterations, and an as small as possible time to obtain the numerical solutions. Another aspect must be highlighted, using past values to calculate the derivatives introduces a time delay: more past values are considered, more the delay increases. Fort all these reasons, the best solution must be considered as a trade-off

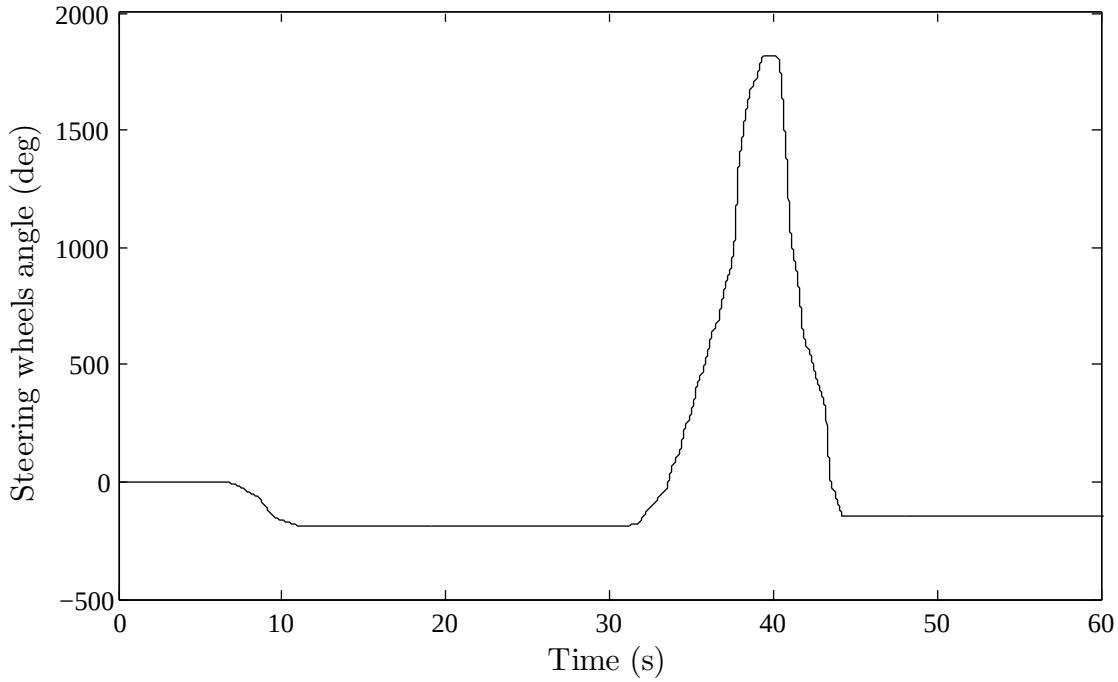

Fig. 4.7: Steering angle of the test maneuver

between a faithful derivatives and the time delay introduced. So, the main objective is: estimate good derivatives with an as small as possible number of past values.

To found the correct solution a test maneuver has been executed and stored, in order to manage the data easily. Some numerical methods has been tested through their implementation in *MATLAB*, and after a comparison of the results, the best method has been chosen.

The derivatives method chosen is treated by Holoborodko (2008). The power of this differentiation method is due mainly to the following reasons:

- It is a backward method.
- It does a noise reduction operation.
- The resulting derivatives are acceptable already considering a small number of past values.
- The results, presented at the end of this section, are incomparable with other methods if considering the same number of values of the past data.

Such differentiators, of any filter length N, can be written as shown by Eq. (4.16).

$$f'(x_i) \approx \frac{1}{h} \sum_{k=0}^{N} c_k f_{i-k} \tag{4.16}$$

Where c_k are the coefficients expressed by Eq. (4.17) and Eq. (4.18), h is the sampling

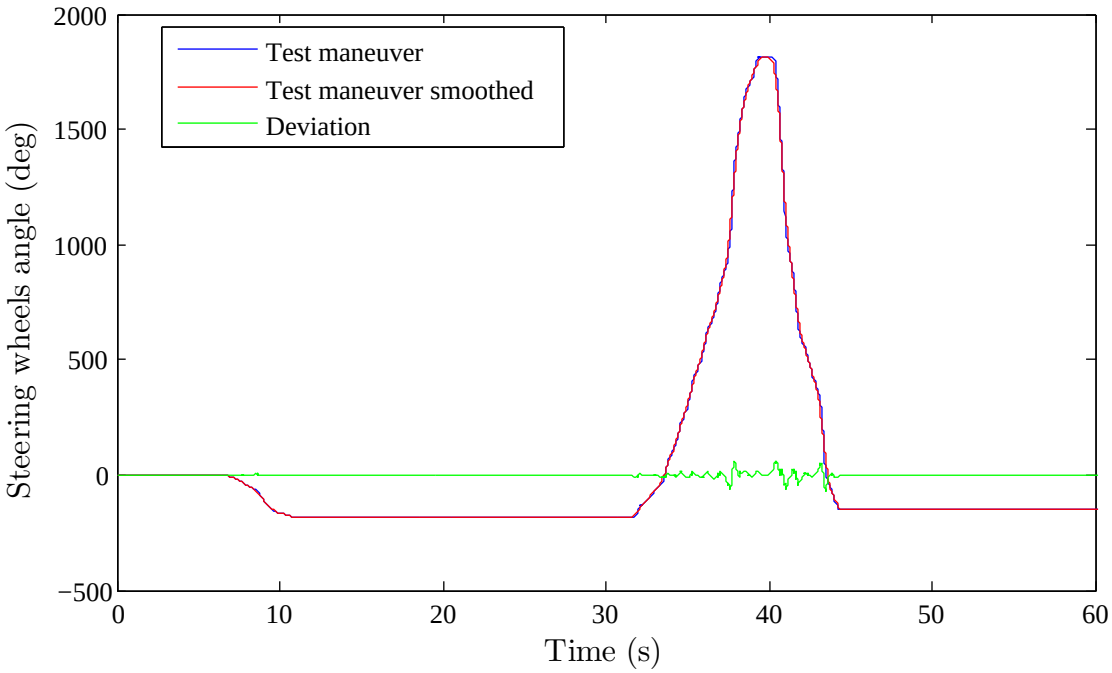

Fig. 4.8: Comparison within the maneuver not smoothed and smoothed

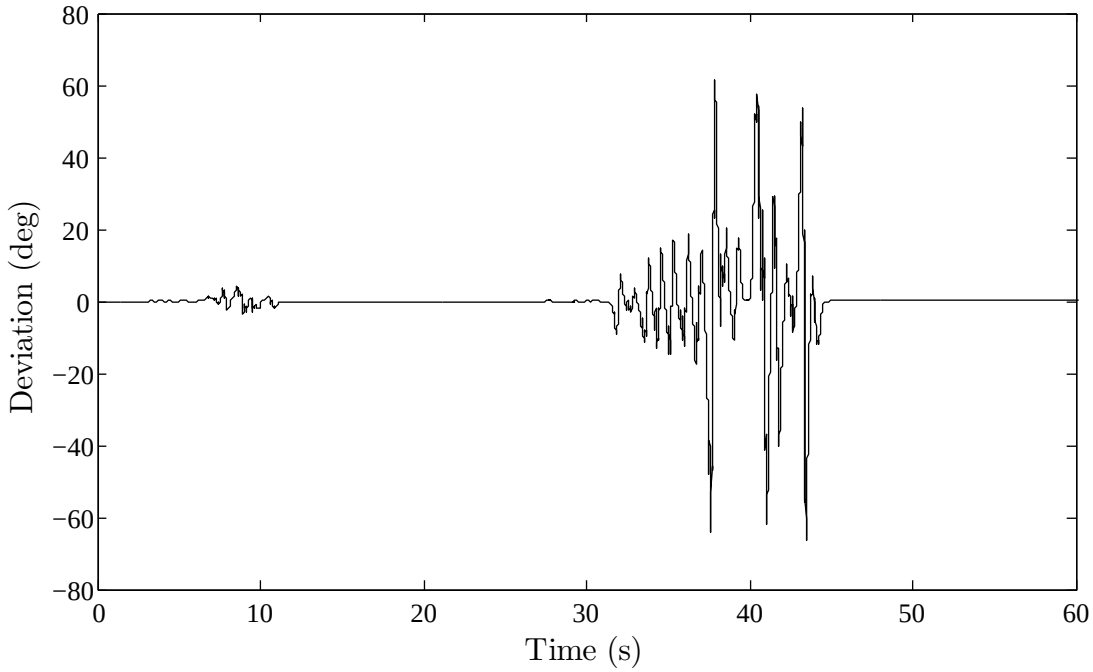

Fig. 4.9: Deviation within the maneuver not smoothed and smoothed

N	One sided smooth differentiators
2	$\frac{1}{2h}(f_i - f_{i-2})$
3	$\frac{1}{4h}(f_i + f_{i-1} - f_{i-2} - f_{i-3})$
4	$\frac{1}{8h}(f_i + 2f_{i-1} - 2f_{i-3} - f_{i-4})$
5	$\frac{1}{16h}(f_i + 3f_{i-1} + 2f_{i-2} - 2f_{i-3} - 3f_{i-4} - f_{i-5})$
6	$\frac{1}{32h}(f_i + 4f_{i-1} + 5f_{i-2} - 5f_{i-4} - 4f_{i-5} - f_{i-6})$
7	$\frac{1}{64h}(f_i + 5f_{i-1} + 9f_{i-2} + 5f_{i-3} - 5f_{i-4} - 9f_{i-5} - 5f_{i-6} - f_{i-7})$
8	$\frac{1}{128h}(f_i + 6f_{i-1} + 14f_{i-2} + 14f_{i-3} - 14f_{i-5} - 14f_{i-6} - 6f_{i-7} - f_{i-8})$

Table 4.3: One sided smooth differentiators formulas

step and $f_{i-k} = f(x_i - kh)$.

$$c_k = \frac{1}{2^{2m+1}} \left[\binom{2m}{m-k+1} - \binom{2m}{m-k-1} \right] \tag{4.17}$$

$$m = \frac{N-3}{2} \tag{4.18}$$

The method combines numerical derivative estimation and guaranteed noise suppression towards upper bound of Nyquist interval. Moreover, it posses preciseness on low frequencies and smooth/suppression of high frequencies. Table 4.3 reported the explicit formulas for some values of N.

The differentiators method presented have been tested for different values of the filter length N, within $N = 10$ to $N = 2$. The upper limit is chosen considering the delay introduced in the derivative. Consider more than ten values to calculate the derivatives involves a time delay bigger than 50 ms, which is unacceptable, even if supported by an improvement of the derivatives. In truth, since no significant improvement of the derivatives quality was registered for high values of N, the filter length $N = 4$ has been adopted as a good choice and trade-off between the number of past values and the derivatives quality. The method is applied on the position to calculate the first derivative, and on the first derivative to calculate the second derivative. To compare the results, Fig. 4.10 shown the test maneuver and the derivatives obtained trough a finite difference method forward in the post processing. One more time must be underlined that such kind of methods are not suitable in an on-board simulation, cause would need the complete trend of the function known. However such derivatives can

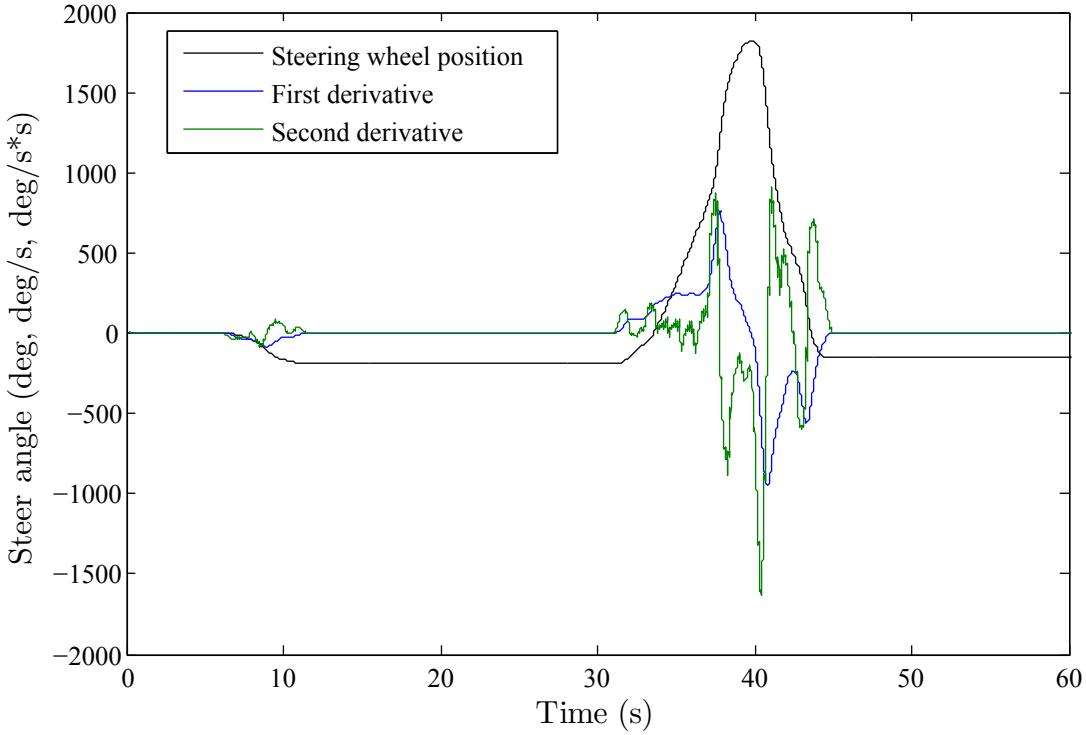

Fig. 4.10: Reference derivatives obtained trough a finite difference method forward

be considered as good references. For the forward method has been considered four points. Figure 4.11 shown the first derivative obtained with the one sided smooth differentiators method, for $N = 4$. It is possible to see that, despite the peaks, the trend is easily recognizable. For the sake of completeness, the good result is emphasized if it is compared with a finite difference method backward shown in Fig. 4.12, which uses the same number of past values and that is suitable for an on-board simulation cause considers only past data. The quality of the first derivative is incomparable: the one sided smooth differentiator gives a better result. For the second derivative there is not a so large improvement on the trend, but however the peaks are correct. Figure 4.13 and Fig. 4.14 shown respectively the second derivative calculated trough the one sided smooth differentiator method and the classic finite difference method four points backward.

The solution to estimate the speed and the acceleration trough numerical methods was adopted in an early phase of the development, when the good performances reached by the data transfer operations was unknown. By this logic, it was thought to limit the DAS operations as much as possible, relying instead on the calculation speed of the PC. At the conclusion of the present project, the positive results give the possibility to trust on the data transfer operation and to exploit the DAS to obtain at least the velocity of the steering wheel. In fact, a simple processing command can be included in the configuration file presented in § 4.4, i.e. `CTRATE`, which can give a measurement of the rotation speed. This option, that deserves to be investigated, could make possible the calculation of the first derivative in an easy an more precise way respect of the current numerical differentiation method. The benefits are also reflected on the second derivative.

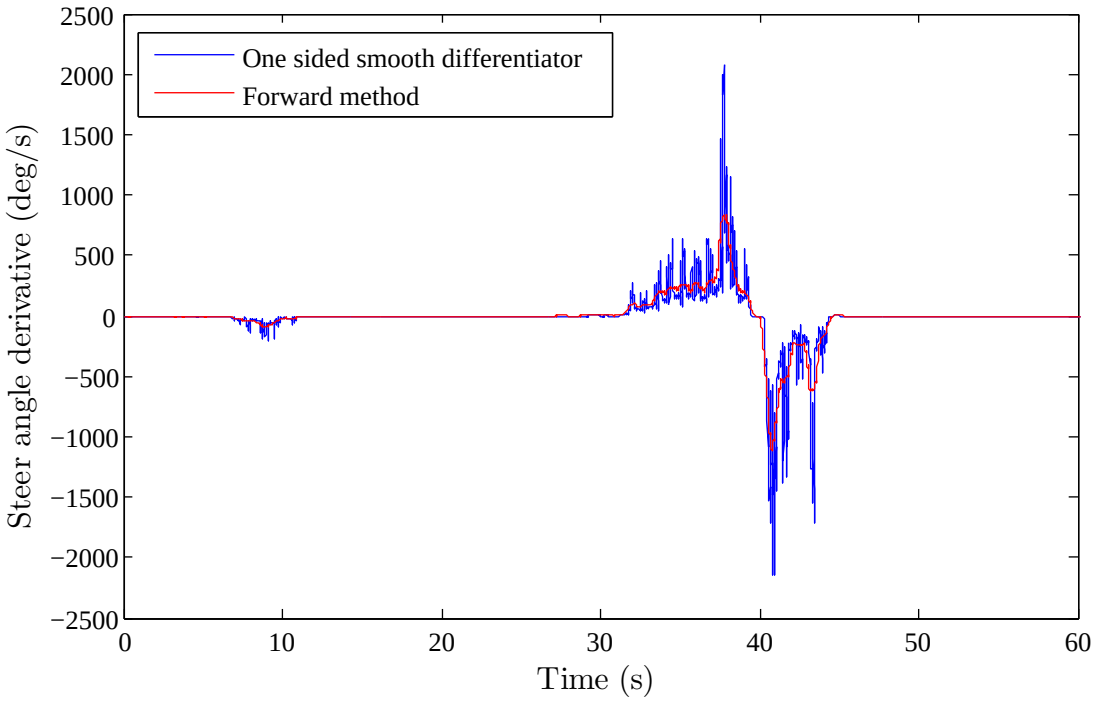

Fig. 4.11: First derivative obtained trough the one sided smooth differentiators $N = 4$

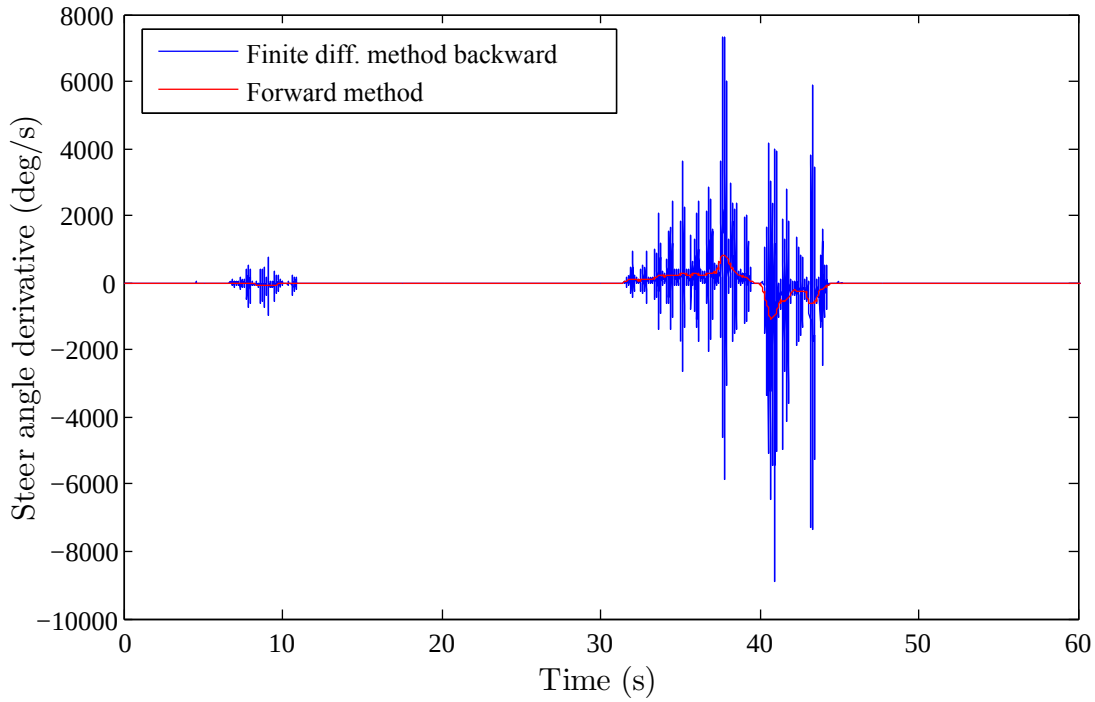

Fig. 4.12: First derivative obtained through a finite difference method backward

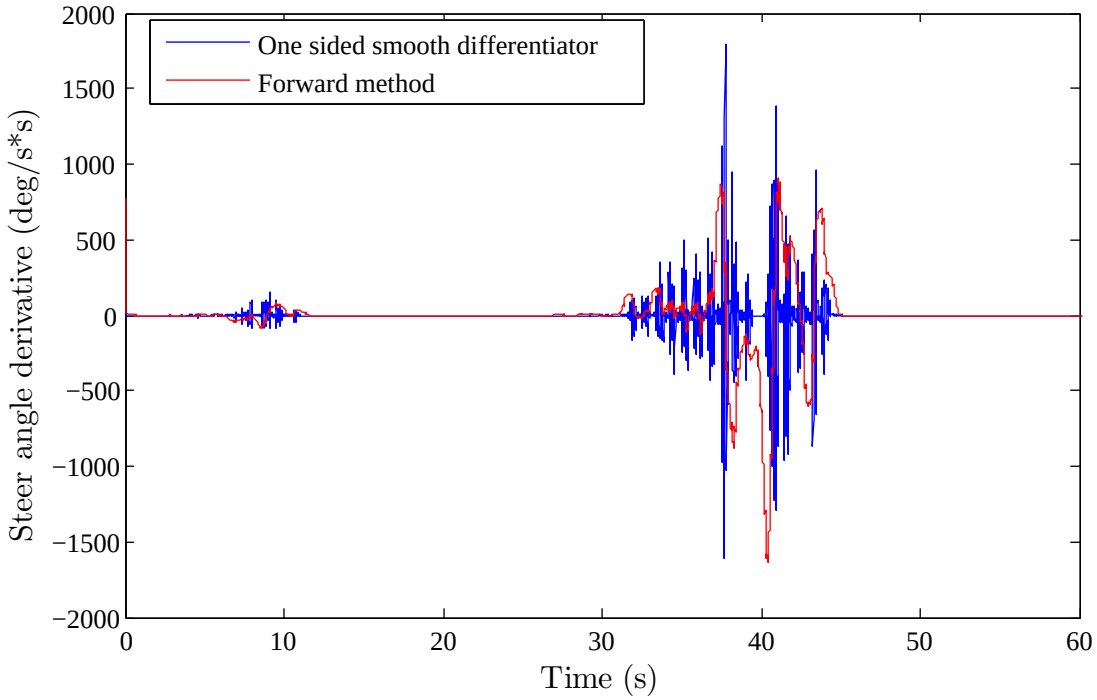

Fig. 4.13: Second derivative obtained trough the one sided smooth differentiators $N = 4$

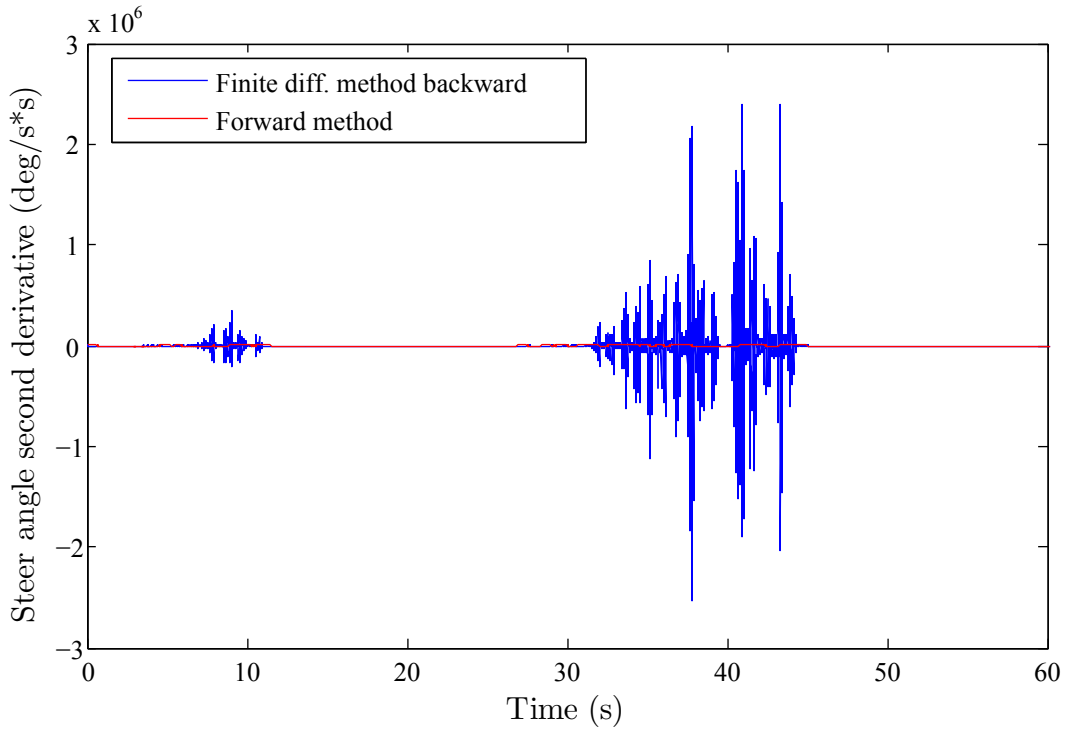

Fig. 4.14: Second derivative obtained trough a finite difference method backward

Sensor output (V)	Digital value	Pressure (barG)
0	0	0
+5	32767	40

Table 4.4: Sensor output, digital value, and pressure value correspondences

4.2.3 Brake pressure

Measurements of the brake pressure have not hard operations. However, it is important to remind the concepts clarified in § 4.2 about analog-to-digital conversion. Indeed, the digital value generated by the DAS must be converted in the desired unit of measurement.

The full voltage range of the DAS pin to which the sensor is connected is ±5 V. The digital precision of the DAS input is 16 bits, that gives a digital range of ±32767. The pressure sensor, presented in § 3.1.2, has a pressure range that is 0-40 barG, while the output voltage range is 0-5 V. As consequence, the sensor has a digital output range of 0-32767, that, in turn, correspond to the pressure range of 0-40 barG. Table 4.4 reports these correspondences. Having available two points, a linear equation has been obtained substituting the four values in the so called two-point form of linear equations, shown in Eq. (4.12); where x can be considered as the general actual digital value received from the DAS, y the general actual value assumed by the pressure (in *bar*); (x_1, y_1) are respectively the minimum digital value of zero and the corresponding pressure value that is 0; (x_2, y_2) are respectively the maximum digital value, that is 32767, and the corresponding maximum pressure value that is 40 barG. The resulting equation written in a clear and general form is Eq. (4.19). It correlates the digital values of the range admitted to the corresponding brake pressure b_p.

$$b_p = \frac{\text{actual sensor digital value}}{32767}\ 40(\text{barG}) \tag{4.19}$$

The last equation is implemented in the `brkPress.cpp` function, shown in appendix F. The function receives the digital value from the DAS and converts it in the correct pressure value. Subsequently, the pressure measurement is made available for the simulator. It truth, inasmuch the digital range is 0-32767, the function managed also the case of a digital value less than zero. In fact, due to noise of the sensor, for a null pressure the digital value is not perfectly zero but a small negative number. In order to avoid an erroneous measurement of a negative pressure, if the digital number received is less than zero, the pressure is setted to a value of zero. Figure 4.15 shows the working scheme of `brkPressure.cpp` function.

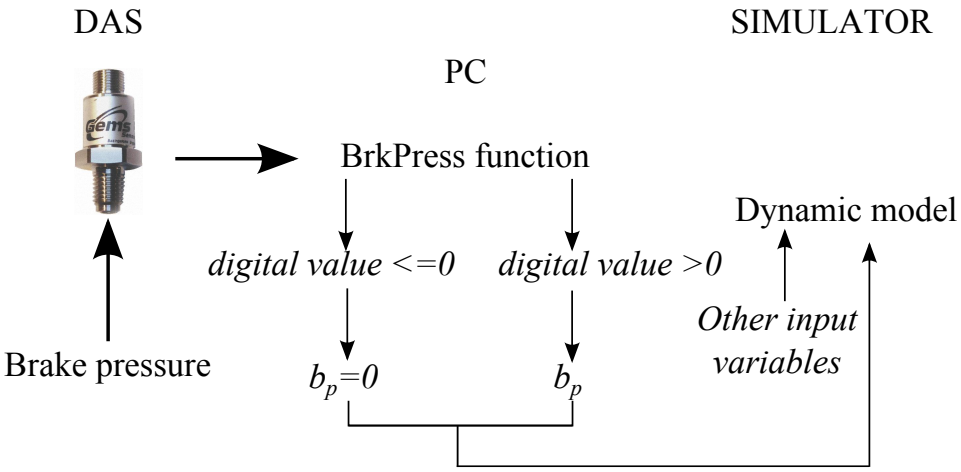

Fig. 4.15: Working scheme of the brake pressure estimation

4.3 System communication tools

The system communication tools available give to the user a large variety of possibilities to communicate with the DAS, as much as give different strategies to manage the data sent. All these instruments are presented in the paragraphs below: a brief introduction on standard software and on DAPIO32 language is given; after, are defined the communication channels first and, subsequently, all the tools applicable to manage data by PC application through DAPIO32; at the end, the strategy chosen by the undersigned is presented in § 4.3.4.

4.3.1 Introduction to DAS management

The DAS used on the prototype is very flexible and gives to the user a lot of instruments to sample and manage data. These possibilities are not only hardware expansions but, above all, software aspects that can improve the standard software given and make DAS adaptive to their own needs or specifications. The data acquisition system, in general, can be managed and setted through the following two standard ways:

- Through software provided with an highly developed graphical interface for input and output operations, such is *DAPstudio*. This applications software is a no-coding-required development environment, that allows to develop a complete data acquisition application rapidly, selecting the settings of the DAS easily (e.g. sampling rate, input and output channels, pre or postprocessing of data). Despite the last assertion, DAPstudio is a powerful and professional software for data sampling management, and can be used for proof-of-concept development or for a full application run-time environment.

- Through software devoid of an highly developed graphical interface, such is *DAPview*. This applications software allows to configure the DAS writing a sequence of commands which are defined in a standard library[1]. In other words,

[1] A library has to be considered like a dictionary where all the commands or functions are defined and all their operations programmed

it is made possible to set the DAS writing the instructions in a text file (like in programming), that can be the input for DAPview. The software sends the file to the DAS and makes the acquisition run. A basic graphical output is given anyhow. The commands sequence is very similar to the so called *machine language*, and makes easy to understand a very important aspect for the present work: the data acquisition system is a *programmable* instrument through a text file, which is provided with commands predefined in a standard library. In fact, this is the method used in the application presented in this work to set the configuration of sampling. The instructions file is explained carefully in § 4.4.

Both the previous methods presented can operate only on the DAS side. Once values are sent in the output pipe, only few simple operations are allowed (e.g. graphical display, data log). On the other hand, before data are sent, some tasks can be done through standard functions or *self-developed modules*. These modules, once added to the standard library, are available for every applications. The usefulness of self-developed modules is, first of all, the possibility to merge multiple operations in one single command, improving performance and computational efficiency. Furthermore, the user can design personal command in `C/C++` languages for his specific aim. A good example on how and why is possible to develop modules is given in Pastorino (2012) and Pastorino et al. (2010).

On the PC side, the most important aim to achieve is the management of data sent by the DAS. The most powerful tool at hand is *DAPIO32* (Microstar Laboratories, b). It is a language interface allows a PC application to control the Data Acquisition Processor through its operating system. DAPIO32 defines functions, commands and complete language rules, which can be utilized during the development of an application in `C/C++`, that can be able to communicate with DAS, and manages data coming from the output DAS channels.

4.3.2 Communication pipes

Application of PC communicates with the DAS through a communication channel structure called *communication pipe*. It is possible to open an handle to a pipe and then use the handle to send and receive data through the pipe, according to First In First Out method (see Fig. 4.3 p. 44). The pipe is also a buffer, in fact DAPIO32 buffers data from and to the DAS in the pipes. There are no dangers that in a pipe could flow data from two or more applications in the same time: pipe can be opened for reading or writing only once; once opened the pipe is reserved until the application closes the open handle. There are communication pipes in the DAPL operating system running on the data acquisition processor and in the PC. They are logically connected on a one-to-one basis in order to obtain a continuous pipe, which is considered an output channel if data flows from DAS to PC, and input channel if data flows from PC to DAS[2]. Each connected pair of pipes (one on DAS and one on PC) form a communication channel between the PC application and the DAS. Four default communication pipes are available for each data acquisition system:

- `$SysIn`. It is used for text commands from the host (in this case the PC) to the DAPL system on the DAP board. The DAS, as already said, is a programmable

[2]To regard a pipe as Input or Output channel, must be considered always a DAS point of view.

instrument through a text file provided with the instructions (commands) to perform the sampling operations. This pipe is useful to send precisely the configuration text file, or every single text command line.

- `$SysOut`. It is used for text messages returned from DAPL system to the application on the host system. This pipe is useful above all for receive error messages.
- `$BinOut`. It is used for binary data transfers from the DAPL system, and its processing configuration to the application on the host system (typically for returning digitized signal data). This is precisely the pipe used to receive data from DAS. Data transfers in binary format were preferred to data transfers in text format because to avoid the change-over from binary to text type, and cause binary format allows higher velocity of transfer. In general, nothing prevents the user to transfer data in text format.
- `$BinIn`. It is used for binary data transfers from the application on the host system to the processing configuration in the DAPL system (typically, for generating output signals). This pipe is not interesting for the present work.

The four default communication channels should be adequate for most applications. Anyhow, additional pipes can be created by the application using DAPIO32 function `DapComPipeCreate`, for a maximum of 32 sets of inputs and outputs.

The data transfers through buffered communication pipes (dynamic object) are not easy like data transfers from a static object, such as a file. In general, the following aspects should be taken into account carefully when working with pipes in any investigation field:

- Place data into the pipe and take data from the pipe are two separated process, which run concurrently. It should be taken into account that the application (on the PC side) is subject to timing constraints of operating system.
- It is impossible to decide, during data sending, exactly when the receiver will take the data; just as it is impossible, during data receiving, to decide exactly when data were sent.
- Simply taking *all* the data that arrive through the pipe implies the impossibility to define exactly the amount of data received: it will vary depending on how much was sent, and how much has arrived.
- Taking a *part* of the data that arrive through the pipe implies the danger to incur in the pipes overloading explained in § 4.1.2 (point three p. 43): some amount of other as-yet-unseen data can remain within the transfer pipe, and/or the maximum buffer capacity can be reached.
- The groupings of data that is taken from the *stream* can be completely different from the groupings as they went in.

For these reasons, for a strong management of pipe, the main targets should be: to be able to receive all data transferred, to avoid waiting for data where not yet sent because is not possible to know the waiting time, to take data in meaningful groups in order to make easy and unequivocal their interpretation.

4.3.3 Strategies available for data transfers

Depending on the goals of the application, DAPIO32 offers a lot of strategies for data transfers. It is important to take into account that every strategies have to be considered only like a guide-line. These methods are presented and commented below, highlighting the base-strategies chosen by the undersigned as references.

One value at a time − This strategy implies that, at each opportunity, a scheduling loop extracts one unit of data from the communication pipe buffers. Being the PC so slow to process values, is obviously the wrong way to transfer a lot amount of data. It is not suitable for the present work because pipe-overloading. Moreover, if for some reason arrival is delayed, the application pauses without control. This strategy is helpful only to manage pipes that are always empty except in certain conditions. For example, a pipe that contains emergency or error warning flags: the function will always wait for something, while other processes of other applications are running; if a warning flag does appear, it is directly transferred.

One block at a time − This strategy, in despite of the poor efficient of the PC operating system in transferring data, allows to manage a lot amount of data. In fact, which each operation many values at a time are moved, then the overhead per transported value reduces dramatically, and it is possible to move a specified number of values. The main problem is precisely the size of block. This method, indeed, attempts to transfer in each operation exactly the number of values defined by the user (block size). If data are not available, the application waits, but it is not possible to know how much time is needed to collect the correct number of values. The consequences are: application blocking (if the waiting time exceeds a maximum value) and/or pipe-overloading, because it is not possible to know how many values are not transferred at each operation. Read the most recent-value is made impossible so this method is not suitable for the present work, while could be helpful for a general purpose of data transfers which are not needed by other processes.

Take everything available − This strategy allows to transfer a lot amount of data, independently of how much data is arrived and when. The application can move everything that arrived with the great efficiency. At each opportunity, the application attempts to receive a full data block. If it does: the data is transferred to other processes and the application attempts to receive another block. If it does not: the data yet arrived is anyway transferred, and the application attempts to receive immediately another block. More over, if arrival is delayed and no data is available, it is permitted to specify a maximum *time-pause*, and let the application do other things. To control data transfers, the DAPIO32 structure called `TDapBufferGetEx` is defined. This strategy *is the guide-line used by the undersigned to transfer data.* Indeed, at first, it has all the characteristics needed:

- The pipe is always kept empty at each operation, independently of how much and when data arrives, so pipe-overloading is surely avoided.

- The computational efficiency is reached, because if no data are available for a specified time, the application can do other things.

- The most recent data is surely available: in fact, transferring for every operations all the values, can be asserted for sure that even the last value entered in the

pipe at the moment of the receiver call is transferred. The key, once data are transferred, is to read the correct value.

After a depth analysis, it is found a big limit due to that in the prototype more than one parameter is sampled (i.e. throttle pedal angle, steering wheel angle, brake pressure). To understand why the strategy fails in this case, some words deserve to be spent on how the DAS works when sampling. The three parameters are sampled and sent to output pipe *one at a time*, in an order that can be chosen by the user. Assume that the order is throttle, steer, brake and that the pipe has a certain amount of data yet collected. The *take everything available* method, at each opportunity, transfers everything arrived, independently of how much and when data is arrived. As consequence, *it is not under control when and where the stream of data will be interrupted and transferred*. Being the pipe a stack of single independent values, there are no guarantees that, at each operation, the stream of data is interrupted precisely at brake value, which is the last one of the three. The certainty that the last three values of transferred data have always the selected order is loosed. It is possible to conclude that, sampling more than one value and transferring data through this strategy:

- It is impossible to know which are the last three values of transferred data at each transfer operation.
- The most recent data of each value isn't surely transferred and available.

Even though this strategy is not directly suitable, must be absolutely taken into account to develop a correct method.

Take what you need – This strategy can be useful when receiving small tagged data blocks, on the sort that occur when there are multiple activities on the DAP board producing small amount of data. Thanks to the *tag* preceding each group of data, it is achievable to identify the type and size. This strategy is not suitable to transfer a lot amount of data and so is not considerable.

Take block if available – This strategy is helpful for data transfers when the data sources generates fixed size blocks with periods of delay between. For example, to monitor one or more parameters every predefined elapsed time. This method is not suitable for continuous sampling at high rate such is the necessities of the prototype. Anyhow, *take block if available* strategy shows how it is possible to transfer data in a fixed size block. Since it is very important for the application developed by the undersigned to be sure to transfer always, at least, three parameters (i.e. steer, brake, throttle), and to avoid the problems that occur in the *take everything* procedure, this strategy can be taken into account for understand the block data transfers. To control data transfers and block size, it is defined the so called DAPIO32 structure `TDapBufferGetEx`.

4.3.4 Strategy chosen and developed for data transfers

As can be inferred in the section § 4.3.3, no one of the basic strategies available is perfectly suitable for the final target of this work. In fact, to accomplish all the goals, a new strategy (only alluded in Microstar Laboratories (b)), which takes the best of some methods, has been developed and thought. The strategy is presented below.

Take all blocks available strategy

Take all blocks available strategy is a mix of two strategies already presented in § 4.3.3: *take everything available* and *take block if available*. From the second method is understood how to transfer data only in a specified block size, from the first one how to transfer all data entered in the pipe. Mixing the two methods, at each opportunity is attempted to transfer all *blocks available*, independently on when and how much data is sent to the pipe. As consequence, the transferred data will be organized in a variable number of blocks of the same size. Obviously, every block consists in the three desired values: steering wheel angle, brake pressure, throttle pedal angle. Through this method, the stream of data entered in the pipe is interrupted always in the same point and, more precisely, after the last value of the last block available. *Take all blocks available* strategy satisfies all the needs, avoiding the problems that occur in the *take everything* method, and without the necessity of a time delay between the block:

- The pipe is always kept empty at each operation.
- If no data are available, for a specified time the application can do other things.
- The most recent data is surely available: in fact, transferring for every operations all the blocks, can be asserted for sure that even the last block entered in the pipe at the moment of the receiver call is transferred. The most recent data will be the three values of the last block.

For the sake of completeness, the assertion "the pipe is always kept empty at each operation" is not completely true. The pipe is effectively kept empty only when a number of values multiple of three is entered in the pipe because the values can be transferred only in groups of three. Such condition is impossible to be controlled. Indeed, it should be kept in mind that process that places data into the pipe is separate from process that takes data out of a pipe. For this reason, the sampled values will enter anyway in the pipe one at a time independently from each other and from takes operation, while the transfer operation takes *only* all blocks of three values available. As implication, it is not guaranteed that the last block has effectively the last values entered in the pipe. For example, could have been left behind in the pipe, in the worst case, two values, cause the last one at the moment of the call of the receiver was not already sent. The most recent data, as consequence, is not the most recent as possible, but this approximation could be doubtless assumed cause the high rate of sampling and so the negligible short elapsed time between two values. Furthermore, this strategy is the best choice (maybe the only one) to transfer the three parameters in a synchronous way (i.e. related approximately to the same time-instant) maintaining an useful sort.

From a code programming point of view, to control data transfers, DAPIO32 gives to the user a structure called `TDapBufferGetEx` and the related function `DapBufferGetEx`. Defining the structure, it is possible to configure all the parameters needed in order to obtain the desired strategy for data transfers through the function. Since this structure is used in the application developed by the undersigned, it is essential see in detail the parameters and the settings for *take all blocks available*.

TDapBufferGetEx structure and DapBufferGetEx function

`TDapBufferGetEx` and `DapBufferGetEx` are strictly intertwined: the structure defines how the function work, but the function defined the input and output parameters to be processed. So, both the commands must be presented (first the function and after the structure).

DapBufferGetEx – It is a DAPIO32 function that, as all the functions in the programming field, operates on input and output defined variables. The function has the following structure:

int_stdcall DapBufferGetEx (
HDAP *hAccel*, **const TDapBufferGetEx** **pGetInfo*, **void** **pvbuffer*);

Three parameters can be defined:

- *hAccel*. It specifies the open handle to the target pipe. The handle can be considered like a pipe on PC, and the *open* operation is nothing but a definition of the logical connection to a specified DAS pipe in order to obtain a continuous communication channel. More details about opening handle are explained in § 4.5. From a point of view of the PC and the application, this is the input channel: data arrive from *hAccel*. Because to define the input/output attributes must be considered always the DAS point of view, the handle must be opened with read access and the target pipe must be an output pipe from the data acquisition processor. Indeed from the DAS standpoint, *hAccel* is the channel from which data is *read*. In the application presented in the present paper, this handle is called *hDapBinGet*, is opened with the read access and has as target one of the default DAS output pipes presented in § 4.3.2: `$BinOut`.

- *pGetInfo*. Which is a pointer[3] to the `TDapBuggerGetEx` structure, that passes the parameters of the get operation into the function. In the application of communication interface developed this pointer is called *BufControl*.

- *pvBuffer*. It is the pointer to the buffer that receives data, and called *bufferIn* in the application presented in the present work.

Summarizing: `DapBufferGetEx` function reads a block of data from the target pipe *hAccel* and transfers data in the target buffer through the pointer *pvBuffer*; instructions on how data is read and transferred are passed to the function through the pointer *pGetInfo*. If the function succeeds, the return value is the number of data bytes read, while if fails, the return value is -1. One last note: **HDAP**, **const TDapBufferGetEx**, **void**, are the declarations of data type. Below, an extract of appendix C shows the function as appears in the communication interface code developed:

```
92  bytes = DapBufferGetEx(hDapBinGet,BufControl,(void *)bufferIn);
```

[3] A pointer is a programming language data type whose value refers directly to (or *points to*) another value stored elsewhere in the computer memory using its address

TDapBufferGetEx – It is a DAPIO32 structure and defines the behaviour of the `DapBufferGetEx` function. This structure is composed of some parameters and has the following form:

typedef struct tag_TDapBufferGetEx {
int *iInfoSize*;
int *iByteGetMin*;
int *iByteGetMax*;
unsigned long *dwTimeWait*;
unsigned long *dwTimeOut*;
int *iBytesMultiple*;
} TDapBufferGetEx;

Six parameters cab be defined:

- *iInfoSize*. It specifies the size of this information structure.
- *iByteGetMin*. It specifies the minimum number of bytes to get. It can be zero, or a positive integer that is a multiple of *iBytesMultiple*. This parameter is very important. In fact, the minimum number of bytes to get corresponds to the desired minimum number of values to transfer. In the application of the communication interface developed, this parameter is setted as the sum of the three desired variables data type. The brake pressure and the accelerator angle are *short* type (2 bytes), while the steer position is *long* one (4 bytes), so the sum of the three desired variables is 8 bytes. With this setting, at least three value are transferred.
- *iByteGetMax*. It specifies the maximum number of bytes to get. It must be greater than or equal to *iBytesGetMin* and a multiple of *iBytesMultiple*. The value of this parameter must be big enough to allow the transfers of all data at each operation, in order to empty the pipe and avoid the pipe overloading. In the application developed, it is setted as the multiplication within the size of the buffer where data are transferred (in number of values) and value of *iBytesGetMin* (in bytes). The result is the value in bytes of the target buffer.
- *dwTimeWait*. It specifies the longest time in milliseconds to wait for new data to arrive. If no new data arrive in this amount of time, the service aborts the operation. A value of zero implies that the application doesn't wait. In the application developed, this parameter is setted to five milliseconds. Since the sampling rate is setted to 200 µs (for details about sampling configurations refer to § 4.4), the total time necessary to sample three variables is 1 ms. It can be asserted that it is very improbable that no data are available, while it is not sure that, at least, one block of three variables has been collected. So data could be available but not processable. In this last case the time out does not start and the number of bytes transferred is equal to zero.
- *dwTimeOut*. It specifies the longest time in milliseconds to complete the entire operation. If the operation fails to complete in this amount of time, the service aborts the operation. A value of zero implies an indefinitely wait, if necessary. This parameters is setted as ten milliseconds.

- *iBytesMultiple*. It specifies that the number of bytes to get for *iBytesGetMin* and *iBytesGetMax* be restricted to a multiple of this value. This parameter is the key for ensure that the sort of transferred data is maintained and that the flow of data is interrupted always in the same point, transferring always only groups multiple of *iBytesMultiple* size. In the application developed it is setted, obviously, equal to *iBytesGetMin*. As result, only the groups of three values available are transferred.

Summarizing: `TDapBufferGetEx` defines the behaviour of `DapBufferGetEx` function. At each opportunity, the function attempts to read and transfer an amount of data multiple of *iBytesMultiple*; if data are available, the minimum data transferred could be *iBytesGetMin*, while the maximum one could be *iBytesGetMax*; if data are not available, the maximum time to wait before abort operation is specified in *dwTimeWait*; all the operations have to be accomplished, anyway, before the elapsed time reaches *dwTimeOut*. Below, an extract of appendix C shows the structure declaration as appears in the communication interface code developed:

```
67 TDapBufferGetEx BufControl;
68 DapStructPrepare(&BufControl,sizeof(TDapBufferGetEx));
69 //At least 3 samples
70 BufControl.iBytesGetMin = sizeof(Type);
71 //As many as " " of these groups
72 BufControl.iBytesGetMax = BUFFER_DIM*sizeof(Type);
73 Always groups of 3 samples
74 BufControl.iBytesMultiple =sizeof(Type);
75 BufControl.dwTimeWait = TimeMaxWait;
76 BufControl.dwTimeOut = TimeMaxTot;
```

It is very interesting to spend some words about the performance of the function. As already said, the `DapBufferGetEx`, if succeeds, returns the number of bytes sent. This information is very easy to achieve (e.g. printing the variable on the terminal, or in a text file) and, thanks to it, considerations about the data transfers and the time needed can be done. The results obtained show that, every loop, the following three case may occur:

- The number of bytes transferred is equal to zero. It can be deduced that the operations of read and transfer are so fast that, during the elapsed time, no data processable are collected. A so called `if` cycle has been thought and implemented in the application in order to prevent that, if this condition occurs, the null values (or whatever has been interpreted by the PC) reach the simulator. In fact, when a null value of bytes is detected, the function *returns* and attempts another read operation (appendix C, line 114). It must be underlined again that processable data are different from available data, cause the parameters setted in the structure. If the transferred bytes are equal to zero, doesn't imply that the pipe is empty but quite simple that the number of the pipe values is not the minimum one imposed.

- The number of bytes transferred is equal to *iBytesGetMin*, that is 8 bytes. This implies that one block , composed of three values (steer position, throttle pedal

angle, brake pressure), was available and transferred. The block available, being the only one, can be considered as the most recent data and sent to the target buffer.

- The number of bytes transferred is larger than the minimum one imposed. This implies that more than one group of three variables is available. If this condition occurs, according to the `TDapBufferGetEx` structure settings, all blocks available are transferred to the target buffer. Since some blocks are buffered, the last one, which is the most recent data, must be read and made available to the simulator. This step exploits the aid of another buffer, which can store only three parameters, and where the most recent data is always sent and overwritten (more details in § 4.5).

Five test maneuvers, lengthy fifty seconds, have been done in order to analyse more accurately the results. Figure 4.17 shows the number of bytes transferred every loop (transfer operation) during the maneuver number one, while Fig. 4.16 shows the number of bytes transferred for the first 1000 loops. All the three cases (one, more than one, none block processable) explained previously are evident. A preliminary analysis makes think that, in most cases, the number of bytes transferred is equal to the size of one group of three variables, that is 8 bytes. This result is evident mainly in Fig. 4.16. The PC, in despite of its slow management of data, can transfer the values so quickly that, in the subsequent loop, only another block is processable. Undoubtedly, this result is due to the powerful of transfer in blocks. A time estimation is possible: the time needed can be considered, with approximation, as the time needed to collect five values. In fact, the data processable are different from the data available and, as consequence, the operation time can not be considered equal to the one needed to collect three values, but surely, is lower than the one needed to collect two groups (six values). Since the sampling rate is setted to 200 µs, as shown in § 4.4, the total time can be easily obtained: 1.4 ms. Another condition which often occurs, is that the number of bytes obtained is equal to two groups of data, that is 16 bytes. Summing the possible maximum number of value of unprocessable data, i.e. two, the total time obtained is 2200 µs, that is 2.2 ms, for a total of height values. It can be asserted that, as a conclusion of a preliminary analysis, on average, the total time seems to vary within the two times obtained, not a such good result. Fortunately this conclusion is not reliable cause other non evident factors must be taking into account. Nevertheless, the application, and so the transfer operation too, is subject to timing constraints of the operating system scheduler. An unpredictable delay may occur due to other process operations (e.g. disk activity, desktop application). Moreover, the impact on the transfer time of the peaks (which occur cause the last circumstance asserted) shown in Fig. 4.17 must be considered. On the other hand, the zero bytes transferred could be a signal that the function, sometimes, is more fast to transfer data than the DAS to collect it, and must be considered too.

A more correct conclusion can be obtained considering the total time of the maneuver and the total number of loops, which are respectively: 50 s and 125229 loops. Dividing the maneuver time by the maneuver loops, the average time of a transfer operation can be easily calculated: 0.40 ms. Unexpectedly, the value obtained is *lower* than the estimation of the preliminary analysis and moreover is lower than the minimum time needed to collect a block of three value, that is 1 ms. This result is supported

Maneuver	Length (s)	Loops	Bytes	Blocks	Transfer time (ms/loop)
1	50	125229	1117416	139677	0.40
2	50	120857	1080528	135066	0.41
3	50	102720	918152	114769	0.49
4	50	149883	1342368	167796	0.33
5	50	100413	897064	112133	0.50
Avg. time					0.43

Table 4.5: Data transferred and execution time for each test maneuvers

by Fig. 4.16, where it is possible to see that the condition of zero bytes transferred occurs very often. If zero bytes are transferred, the PC is so quickly to transfer the values that in the subsequent loop none block is available. The result is better than expected and very promising, because it remains largely under the integration time step imposed, i.e. 5 ms and moreover under 1 ms. Furthermore, the peaks do not have a significant influence.

Table 4.5 summarizes the number of loops, of bytes transferred, of blocks transferred and the average transfer time for each maneuver. It shows that a variation occurs in the results, due to the impact of the operating system scheduler, which, as already said, involves casual and unpredictable delays. Considering all the five maneuvers, the final average time is equal to: 0.43 ms, largely under the integration time step imposed, i.e. 5 ms. This result, at the moment, is the more reliable obtainable but can not be considered totally the real limit of the achievable performances. In fact, the influence of the operating system scheduler is unquantifiable. A totally reliable measurement should be done on a real-time operating system.

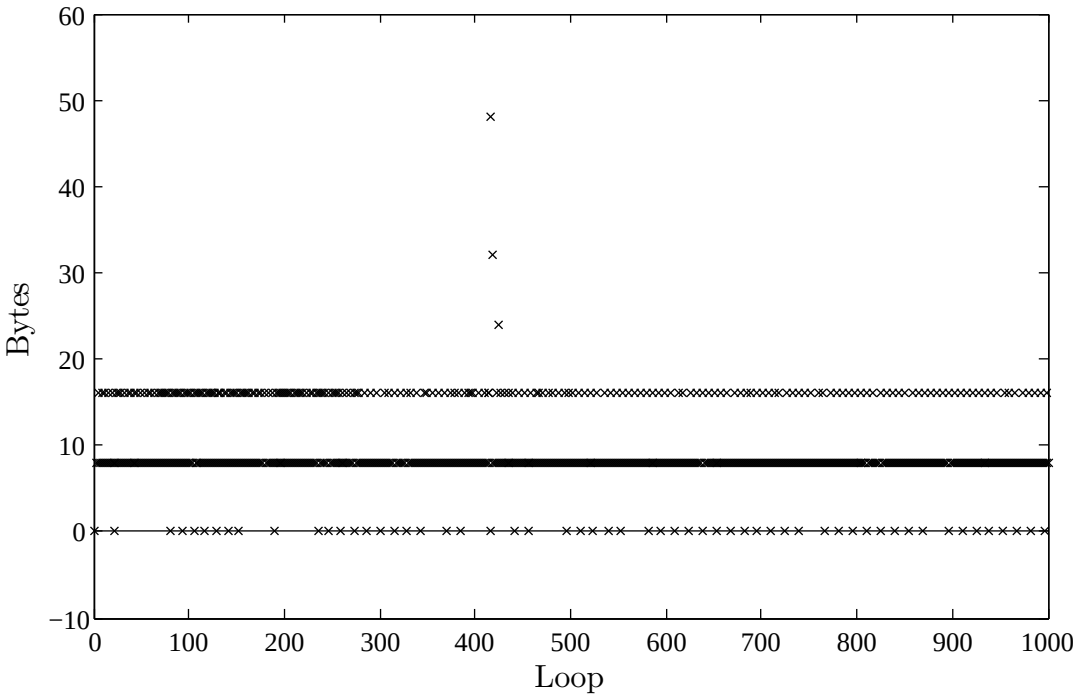

Fig. 4.16: Bytes transferred during 1000 loops

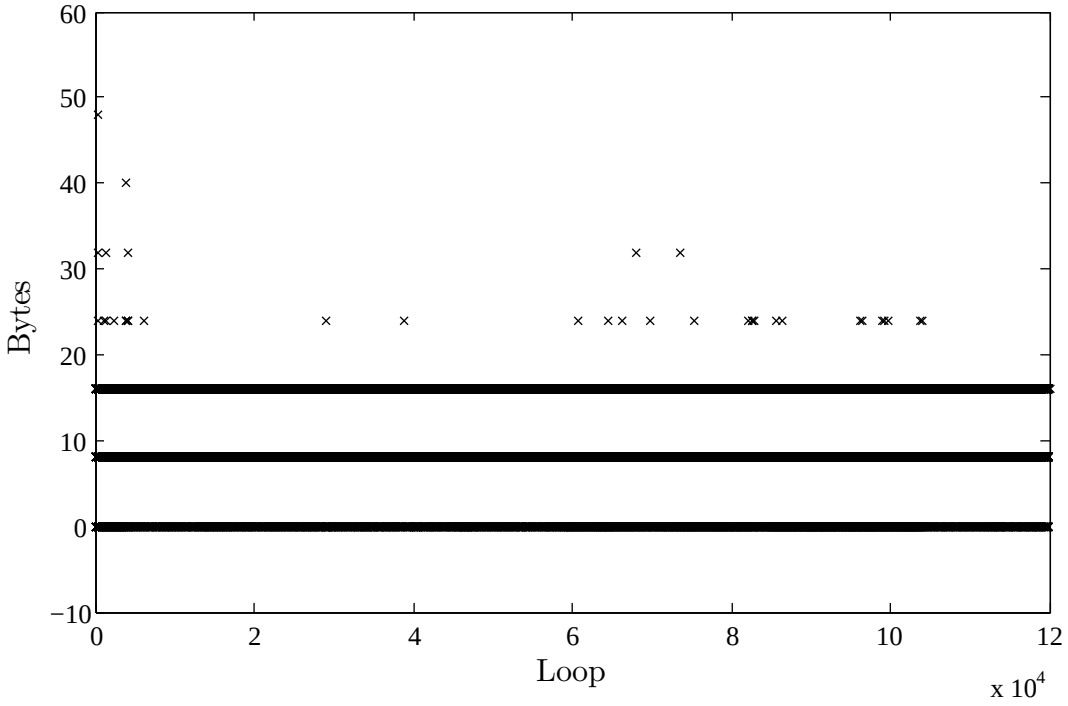

Fig. 4.17: Bytes transferred during an entire maneuver

4.4 Data sampling configuration

As already mentioned, the digital acquisition system can be managed through a text file, provided with the desired commands and options, and sent through the communication pipe `$SysIn`. This is precisely the way adopted at the present project to configure the settings of the sampling. The application developed opens the communication pipe and through the command `DapConfig` sends the instructions file to the DAS (see appendix C, line 64).

The text file, or better the *configuration script*, consists in *procedures*, which are groups of commands that together perform some functions. The commands are recognized by the DAS operating system called DAPL. The DAPL is downloaded into RAM of the data acquisition processor during the boot sequence of the host processor. Once the configuration script is downloaded, it is translated automatically into a set of *tasks*, which consist of commands and its parameters. All the tasks in a procedure execute concurrently when a procedure is active. The procedures can be grouped in the following types:

- Input procedure sets the sampling rate and select the physical input pins on which voltages are sampled.
- Processing procedure deals with the processing of the data. For example it might contain several commands setting up an average tasks that the DAS executes concurrently on different input channel pipes.
- Output procedure defines commands to perform output operations. For example it might contain commands setting up the output port or pipe for the processed data.

An application has no obligation to have all the procedures described, but most common configuration is: one input procedure, one processing procedure and possibly one output procedure.

Since the number of DAPL commands is very large, it would be difficult and useless to describe them all. For these reasons, only the commands used in the configuration script of the communication interface developed are presented below. Further details can be found in Microstar Laboratories (c). The configuration script is called `con_dap.DAP` and is reported below:

```
1  RESET
2  outport 8..11 type=1
3  options buffering=off
4  options quantum=200
5  options scheduling=fixed

6  PIPES P1 LONG
7  IDEF A 5
8  SET IP0 S4    //BRAKE   buffer[1]
9  SET IP1 B3
10 SET IP2 B2    //SWHEELS buffer[3]
11 SET IP3 B2
```

```
12 SET IP4 B2    //THR     buffer[2]
13 TIME 200      //microseconds
14 END

15 PDEF B
16 QDCOUNT(IP2,0,"RELATIVE",P1)
17 MERGE(IP0,IP4,P1,$BinOut)
18 END

19 START A,B
```

As it is shown, the configuration script consists in a preamble, where are specified some system options for controlling the trade off between latency and efficiency, and two procedure: the input procedure called `A` defined by the command `IDEF` with the declaration of the number of channels (such is five) to be sampled (line 7); the process procedure called `B` defined by the command `PDEF` (line 15). Each procedure is closed by its own `END` command. The tasks defined in the procedures remain available, though inactive, until the system command `START`, in the last line, activates the procedure.
In the preamble, the commands useful especially for a real-time application are the follows:

- `RESET` command clears all definitions and errors. It is a good practice to put always this command in the first line.
- `outport` command informs DAPL of the types of output expansion boards in a system and their output port addresses, both depending on the hardware configuration: `type=1` is for analog output expansion board. The output expansion boards, in fact, appear to the data acquisition processor as several input/output ports.
- `option buffering` acts on the buffering of the processed data. With the option *off*, the task pushes individual values through the sequence without buffering.
- `option quantum` sets the maximum time allocation allowed per task, in microseconds. A setting of 200 µs is typical when low response latency is required.
- `option scheduling` deals with the tasks scheduling. When DAPL switches between tasks, the operating system is responsible fro selecting the task to activate. With the option *fixed*, DAPL uses a round robin scheduling algorithm.

The configuration of buffering, quantum and scheduling shown, set the DAS in a low latency configuration.
In the input procedures two commands are defined:

- `SET` command associates an individual physical input pin `S4`, `B3`, `B2`, to the input pipes `IP0-1-2-3-4`. The number of the pin is dependent on the hardware configuration, inasmuch every number correspond to a specific pin on which a specific sensor is plugged. On the other hand, the output pipe numbers are user-defined but when the sampling configuration runs, it will capture samples in order of channel identifier numbers rather than by order of appearance within

the `IDEF` section. The input pin names begin with a character that allows to identify the pin type: S stands for *single-ended* analog input while B stands for *binary* digital input source. Once more, the pin type depends on the hardware. In the input procedure considered, one single-ended input pin, on which the brake pressure sensor is plugged, is defined and associated with the pipe number 0. The other pipes regard the quadrature decoder board, are digital type, and the relation between pins and pipes is not evident like the analog case. In fact, the quadrature decoder board consists of two input ports: a control port (i.e. `B3`) and a data port (i.e. `B2`). Reading the control port *latches* (stores) values of all the counters on the quadrature decoder board for reading. Internally, the counters continue to monitor and count events while latched values remain stable. After latching, each operation reading from the data port obtains one latched counter value. The first read from the data port obtains the latched value from the first counter. Subsequent read operations obtain the latched count values from the second, third and fourth counters in sequence. Reading the control port again ends the read sequence and latches new count values in all channels. Reading the data port again begins the next cycle of reading counter values, starting with the first counter. For example, considering the file `con_dap.DAP`, reading the control port `B3` stores four values (one for each encoder) on the quadrature decoder board and a not useful data is sent to `IP1`. To read the stored data, some read-operations of the input port `B2` must be executed in order to scroll through the values. For this reason, the first lecture of the input port obtains the first value, that is the digital value from the steering wheel encoder, and sends it to the input pipe `IP2`; the second lecture obtains the digital value from another encoder which is not considered in the present work; the third lecture obtains the third value stored, which is the digital value of the throttle encoder, and sends it to the input pipe `IP3`. It is possible to observe that the second lecture is useless. Unfortunately, none skip operation is available, so the only way available to scroll the stored values is to read in order the counters, also of the unwanted data. However, not all counters must be read. If, for example, only the first two counters are needed, the last two can be omitted. In the configuration script shown in this section the last counter is omitted. Lastly, the order of the stored values depends on the hardware configuration and on how the encoders are plugged to the quadrature board.

- `TIME` command sets the sampling time in microseconds unit. In case of multiplex input sampling and with M channels, each channel is sampled every $TIME \cdot M$. Because the sampling time has been set to 200 µs per channel, and there are five channels, each pin is sampled every 1 ms, while the delay between two subsequent channels is 200 µs.

Some optimization of the input procedure are available, for example a more useful hardware configuration could avoid the lecture of the useless counter, saving time. Moreover, the sampling time can be reduced.
In the process procedure the following commands are defined:

- `QDCOUNT` processing command deals with the quadrature decoder board, and maintains a 32-bit representation of the running count compensating for the

numerical overflow conditions, yielding a running 32-bit count with vastly larger effective range. It also provides some helpful features for establishing the initial state of the processing. An additional 32-bit pipe is needed to receive adjusted count values. This expedient is useful to process the steering wheel encoder digital values, since the digital range of ±32767, due to the 16 bit resolution, is too short to describe large rotation angles of the steering wheel. In fact, the numerical overflow condition is quickly reached for small rotations and the count appears to jump instantaneously from a large positive value to a large negative value, or reverse. From a simulator point of view, this is traduced in an instantaneous change of the steering position, from a large positive angle to a large negative value (or reverse), despite the user is turning in the same verse. To avoid confusion that this cause, the `QDCOUNT` command reads the values form the pipe `IP2`, that is a 16 bit representation of the steering wheel count, and sends a 32 bit representation to the user-defined pipe `P1`. This pipe is declared in the line 6 with the attribute *long*, since a 32 bit representation produces long data type. Two options are defined for this command: the initial offset of the count start, setted to *zero*, and the operating mode setted to *relative*, that is an explicit request of the default operating mode. It assures that only the counts change after processing is started are considered, while the counts change between the time that the hardware counters are initialized and the processing begins are unconsidered.

- `MERGE` reads data from one or more input pipes and places the data consecutively into an output pipe. One important characteristic of this command is: data arrival rates in all input pipes must be equal. Thanks to this command, the values of the pipe `IP0`, `IP4`, `P1`, which correspond respectively to the brake pressure, throttle pedal angle and steering wheel angle (long representation), are sent maintaining the order to the output pipe `$BinOut`. The order of the merge operation is very important, because permits to easily identify data once are transferred on the PC, in the buffers system presented in § 4.5. Moreover, the order is not casual, indeed a little trick is to put the long data type in the last position, in order to avoid eventual *data type format* problems.

The last line of the script shown the aforementioned `START` command. It activates the procedures defined.

4.5 Whole application and buffers system

All the functions presented in the previous sections work together to realize the communication interface. They are also linked by a buffers system that is useful to identify and transfer the most recent data at the simulator call. Figure 4.18 shows the general organization of the whole application. Referring to this figure, the application is divided in two main parts:

1. DAS management thread, called `DapControl`. The thread has the main goal to manage the data acquisition processor and to transfer the data available at all time.

2. *Function call* section (on the left), that consists in three function: `derived`, `compute_par`, `brk_press`. The goals of these function are to process the most recent data and to send the values to the simulator.

The most important advantage of using a thread to manage the DAS is that it is possible to execute it in parallel respect the main program. So independently by the simulator running, the thread executes the operations programmed. The function call section, instead, is executed only when the simulator need data. It can be considered like a *read operation* of the simulator.

In detail, a simulation works as follows. In the *code main section*, first it is initialized the DAS management thread and subsequently the simulator that however will run effectively only after a user explicit command. The thread, at the contrary, runs immediately. First of all, it opens the communication pipes presented in § 4.3.2. Through the input pipe, the configuration script shown in § 4.4 is sent to the DAS and the sampling operations starts (these operation are executed only once). After that, thanks to the function explained in § 4.3.4, the thread takes all blocks available from the output DAS pipe and puts all the data transferred in the first buffer, called `bufferIn`. In this way, the output pipe is always kept empty. The `bufferIn` buffer consists in some blocks, so its *length* is equal to the maximum number of blocks which can be transferred. This number must be large enough, cause every transfer operation, all the data sampled must be transferred. One block, in turn, consists in: one value of the brake pressure, one value of the throttle angle, one value of the steering wheel angle. Moreover it is inaccessible by the simulator. Once the data are available in the first buffer, the *most recent data*, which is the last block entered, is stored to another buffer: `bufferOut`. This buffer has length *one*, or in other words, it consists only in one block which is always overwritten at every loop and that is accessible by the application. Being the number of blocks different at every loop, a check on the bytes transferred is implemented to recognize the correct number of blocks and so the last one. The data transfer operations are executed every time is possible, continuously, as fast is possible, in an infinite loop. As shown in § 4.3.4 one loop takes 0.43 ms.

On the other side, the simulator, every time step (i.e. 5 ms), needs input data to execute the simulation. It calls the functions presented in § 4.2 which in turn read the most recent data block from `bufferOut`, process the data, and send the data to the simulator. At this point, the simulator has all it needs. In the same time the thread is still running and the most recent data is continuously update.

Despite the parallel execution of the simulation and the data management, the two process are not truly independent from each other. In fact, it should be remembered that they are executed on the same PC processor. Obviously, they influence each other in term of performances. This aspect, unfortunately, is absolutely random and unquantifiable, especially when a real-time OS is not used. Certainly, there could be improvements with a real-time OS, but the best could be a real parallel execution exploiting two different CPUs: one dedicated to the simulation and one dedicated to the data management.

In the previous section, it has been stated that the most recent data is the last block entered in the buffer `bufferIn`. To recognize it, a check on the bytes transferred is implemented: every loop the number of bytes transferred is stored, and through Eq. (4.20)

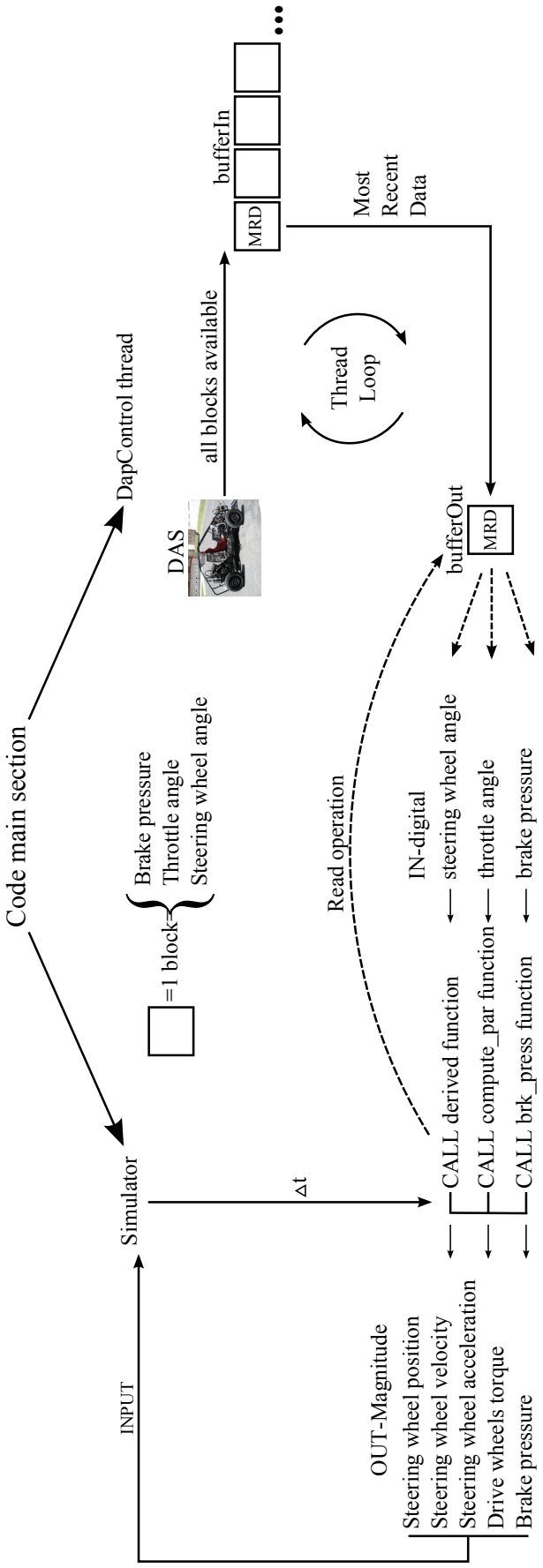

Fig. 4.18: Working detailed scheme of the application developed

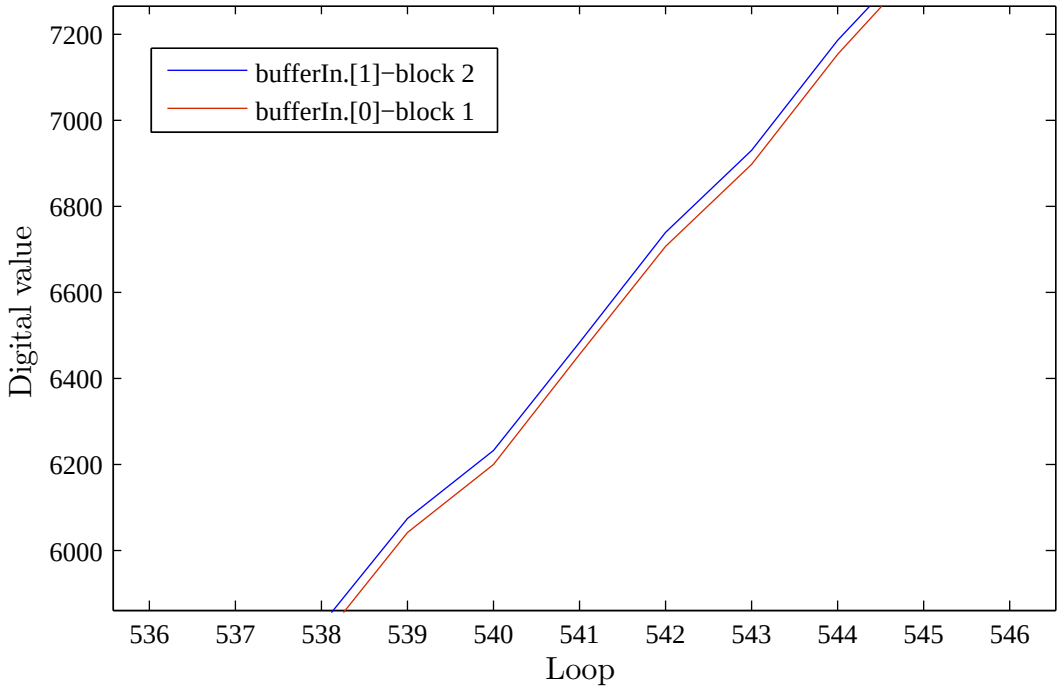

Fig. 4.19: Values assumed by the first and last block stored in bufferIn

is calculated the number of block transferred nb, where 8 byte is the size of one block.

$$nb = \frac{\text{bytes transferred}}{8\ byte} \tag{4.20}$$

The most recent data is located in the last block, which has an index that is $i = nb - 1$. This is simple justified keeping in mind some basic rules of `C/C++` programming: the buffer, which is a mono-dimensional *array* or vector, consists in *elements*. These elements, or blocks as are called in the present paper, are identified by a progressive index. The first element has not an index value of 1, but an index value of 0. For this reason, the index can assume the value within 0 and $D-1$, where D is the vector length (Bellini and Guidi, 2009). At the contrary, none rules assure that the most recent data is stored precisely in the last block, cause only the order in the communication pipes is certain. For this reason, a simple check has been done: turn the steering wheel in the right hand implies a registered digital value increasing; during the execution of a slow operation of turning, maintaining the right hand direction, all the blocks of `bufferIn` has been stored in a text file with the correct index. Considering only the cases of when two blocks are transferred, Fig. 4.19 plots the values of the first and the second block. It is evident that the last block transferred, which has the index number 1, has always a value bigger then the first. As consequence the last block is the most recent data at the moment of the transfers data operation and the order of the pipe is kept. The last check is to verify that the block stored in the `bufferOut` matches to the last block of `bufferIn`, and so to the most recent data. Figure 4.20 shows the subtraction of the most recent data from the value stored in `bufferOut`: the result is always 0, the values match perfectly during the entire maneuver. At the end, a numerical extract is reported.

bufferIn.[0]	bufferIn.[1]=MRD	bufferOut.[0]
93947	93992	93992
94175	94221	94221
94497	94543	94543
94869	94916	94916
95152	95199	95199
95532	95580	95580
95869	95918	95918
96209	96258	96258
96600	96649	96649
96894	96943	96943
97188	97237	97237
97533	97583	97583
97883	97933	97933
98234	98284	98284
98536	98586	98586
98888	98938	98938
99239	99290	99290
99645	99696	99696
...	...	...

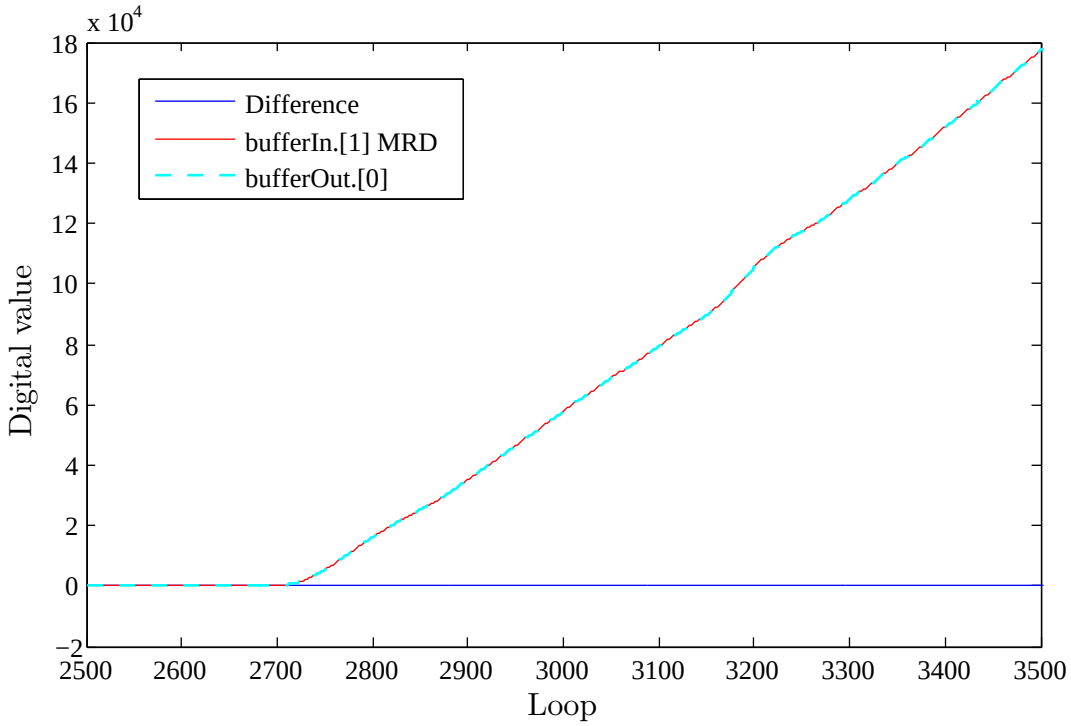

Fig. 4.20: Correspondence between the most recent data and the value stored in bufferOut

Chapter 5

Conclusions

5.1 Goals achieved, possible applications and future developments

All the aims, goals reached, and improvements available of the application developed, are already reported in every sections of the present work. However, for greater clarity, as final comment, all of them are summarized and further discussed below.

First of all, the main objective of the present work is reached: the execution of the simulation on-board, in real-time, employing the signal from the digital acquisition system. The performances reached are better than expected, and this is absolutely a step forward from an industrial application point of view. The application developed permits now to simulate in real-time, on-board, the behaviour of the car. In this way, the system is able to prevent an event of unstable handling and takes decision in real-time. Moreover, the small number of sensors utilized to monitor the prototype involves a small investment. A futuristic application regards the automation of land vehicles without the driver interaction. In fact, monitoring the car through a data acquisition system and elaborating the data through the simulator, in case of sudden changes of the environment, appropriate actions and/or decisions can be taken. So the car will be provided with a better automatic and realistic driving. The car industries, moreover, can be interested on the present project cause it could be useful to test the performances of a real car, driving the virtual prototype on a virtual track, with the engine off. It is evident the enormous saving of money. At least, thanks to the communication interface developed and to the on-board simulation execution, it is possible trough a state observers technique to fix potential simulation errors and to obtain more informations about the vehicle handling than the number of magnitudes monitored. So instead of using real sensors to support some common safety car systems, it is possible to obtain every kind of desired data directly from the real-time simulation. This innovation involves a considerable saving of money and car development time.
The main objective is achieved thanks to some small goals, the most important are:

- Asynchronous data management is reached, creating a thread for the data transfer operations, as shown in § 4.5, which runs in parallel to the simulator.

- Thanks to the `DapBufferGetEx` function, presented in § 4.3.4, the data transfers operation reaches very good result: a time for the transfer of 0.43 ms, which is less than the time step imposed and moreover less than one millisecond. This guarantees a time-faithful data.

- The communication pipes are always kept empty, avoiding the overloading problem.

- The most recent and time-faithful data is made available every transfer operation. This is proved and shown in § 4.5, where the buffers system is explained.

- The functions presented in § 4.2 process the data before to send the values to the simulator. Especially, regarding the steering wheel, through a numerical differentiator method the derivatives are calculated and smoothed, obtaining a very good result with a small number of past values considered.

Despite the considerations above, and the good results reached, numerous improvements are available:

- The employment of a real-time operating system (RTOS), e.g. LynxOS, OSE, QNX, RTlinx. This solution will give more reliability on the test results, cause RTOSs are more appropriate to achieve the aims of a real-time application. The key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application task; the variability is *jitter*[1]. An RTOS, moreover, has an advanced algorithm for scheduling, a minimal interrupt latency and minimal thread switching latency. Compared to a classic OS, the advantages and the benefits are several, both on the performances of the application and of the simulator.

- The application developed deserializes the two processes, i.e. the simulation and the data management, but they are performed on the same CPU. This approach does not permit to make the best use of the parallel tasks programming. So the execution of the processes on different CPUs must be surely investigated.

- In § 4.2.1, in order to estimate the drive wheels torque, an engine model is adopted. However this solution is appropriate only in an early development phase: first of all, the gearbox must be taken into account in the current model, and moreover a more faithful model must be developed. Another solution is to monitor directly the drive wheels torque through an Hall-effect sensor. In this manner, however, the only way to drive the virtual prototype is to drive the real car.

- In § 4.2.2, a numerical method is exploited to obtain the first and the second derivatives of the steering wheel position. This approach introduced unavoidable errors already presented. A more better measurements can be obtained thanks to a simple processing command, i.e. `CTRATE`, which can be included in the configuration file presented in § 4.4. This command can give a direct measurement of the rotation speed processing the signal of the position, without the need of another sensor. On the other hand, this option, that deserves to be investigated, involves changes in the entire application. The command was not considered at first cause the good performances of the data transfer operations were unknown. By this logic, it was thought to limit the DAS operations as much as possible, relying instead on the calculation speed of the PC. At the conclusion of the present project, the positive results give the possibility to trust on the data transfer operation and to exploit the DAS to obtain at least the velocity of the steering wheel.

- The sampling period can be reduced. Smaller is the sampling period, more the data is faithful in term of time, cause the delay between the simulator call and the sampling instant is reduced.

Lastly, another improvement is available, and regards the simulator. In truth, the dynamic model of the simulator is not complete, missing the model of the tyres behaviour. Once the model will be introduced and since the application developed

[1] Jitter is the undesired deviation from true periodicity of an assumed periodic signal in electronics and telecommunications, often in relation to a reference clock source.

does not introduced important delay, the number of the iteration will decrease. The real-time simulation could be reached in full.

Appendix A

Changes introduced in main.cpp

```
...
...
//Changes introduced by Pasquale Gallo in order to interface
//                                                DAS-simulator;
//March-September 2012
//
#include <Windows.h>      //create thread
#include <iostream>
#include <fstream>
#include "DapControl.h"   //header dapControl
#include <stdio.h>
...
...
//bufferOutDap as global variable.
//The most recent value from Dap is stored here.
typedef struct My_buf
{
  short int brake;
  short int thr;
  long int stw;
} Type;
#define BUFFER_OUT_DIM 1
Type bufferOutDap[BUFFER_OUT_DIM];
...
...
//Thread variables.
DWORD ID;
HANDLE threadDap;
//call thread DapControl.
//The thread will running parallel to simulator.
threadDap=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)
                       DapControl,&bufferOutDap,0,&ID);
printf("Create Thread surpassed\n");
...
...
```

Changes introduced in guiado.f90

```
...
...
!Changes introduced by Pasquale Gallo in order to interface
!                                                DAS-simulator;
!March-September 2012
!Fortran calls to C function "interface"

interface
  subroutine compute_par(a, b, c) bind (C, name='compute_par')
     implicit none
     real(8):: a, b, c
  end subroutine
  subroutine brkPress(a) bind (C, name='brkPress')
     implicit none
     real(8):: a
  end subroutine
  subroutine derived(a, b, c) bind (C, name='derived')
     implicit none
     real(8):: a, b, c
  end subroutine
  subroutine time_c(a) bind (C, name='time_c')
    implicit none
    real(8)::a
  end subroutine
end interface
!!Reading data from DAP
!!Fortran calls to C function
  call time_c (time)
  call compute_par (ap(30),ap(40),torque)
  call brkPress (brk_press)
  call derived (str_pos,str_vel,str_ace)
...
...
```

Appendix B

bufferOutDap.h

```
1  //Developed by Pasquale Gallo, March-September 2012;
2  //include this header to read from bufferOutDap
3  //bufferOutDap defined in main.cpp

4  #ifndef __BUFFEROUTDAP_H__
5  #define __BUFFEROUTDAP_H__
6  typedef struct My_buf
7  {
8       short int brake;
9       short int thr;
10      long int stw;
11 } Type;
12 #define BUFFER_OUT_DIM 1
13 extern Type bufferOutDap[BUFFER_OUT_DIM];;
14 #endif
```

DapControl.h

```
1  //Developed by Pasquale Gallo, March-September 2012;
2  //Dap thread header

3  #ifndef __DAPCONTROL_H__
4  #define __DAPCONTROL_H__
5  #include "windows.h"
6  extern "C"
7  {
8  DWORD WINAPI DapControl (LPVOID lpParameter);
9  }
10 #endif
```

Appendix C

DapControl.cpp thread

```
1  //Developed by Pasquale Gallo, March-September 2012

2  //DapControl Thread.
3  //This Thread is called from main.cpp.
4  //The thread runs parallel to simulator.
5  //DapControl continuously reads from bufferIn
6  //The most recent value is sent to bufferOut.
7  //bufferOut is a pointer to bufferOutDap.
8  //bufferOut has BUFFER_DIM and can store a lot of values in order
9  //to avoid the complete $binout filling.
10 //bufferOutDap has BUFFER_OUT_DIM 1 and stores only the most
11 //recent value of bufferOut.
12 //All the operations are executed as fast as possible but:
13 //with variable TIME, it is possible to do a cycle-pause.
14 //
15 #include <stdio.h>
16 #include <Windows.h>
17 #undef UNICODE
18 #include <dapio32.h>
19 #include "DapControl.h"
20 #define BUFFER_DIM 50000       //number of vector "My_buf" which is
21                                //contained in buffer.
22 #define BUFFER_OUT_DIM 1       //number of vector "My_buf" which is
23                                //contained in buffer.
24 const short int TIME=2;        //time to "Sleep" {ms) for Data Send
25 const short int dim_one_block=8//dim of one block=8 bytes
26 const short int TimeMaxWait=5;//Milliseconds maximum to wait for
27                                //first new data, no data=return
28 const short int TimeMaxTot=10; //Milliseconds maximum before abort
29                                //operation, 0=wait indefinitely.
30 //delivered data function
31 int DeliveredData;
32 typedef struct My_buf
33 {
34      short int brake;
```

```
35      short int thr;
36      long int stw;
37  } Type;
38  //function declaration
39  int DeliverData(Type *,Type *,int,int);
40  void ReceiveAvailableData(HDAP hDapBinGet,TDapBufferGetEx *BufControl,
41                                                          Type *,Type *);
42  DWORD WINAPI DapControl (LPVOID lpParameter)
43  {
44  //bufferOut points to lpParameter; lpParameter points to bufferOutDap.
45  struct My_buf*bufferOut=(struct My_buf*)lpParameter;
46  Type bufferIn[BUFFER_DIM];
47  //Get: send to PC, set OUT port, option READ; Put: reverse Get
48  HDAP hdapBinGet, hdapSysPut;
49  BOOL isConfigured;
50  // Open communication handles
51  //PC->DAQ
52  hdapSysPut = DapHandleOpen("\\\\.\\dap0\\$SysIn", DAPOPEN_WRITE);
53  if (hdapSysPut==0)
54   printf("Error opening DAP text input handle hdapSysPut\n");
55   else printf("Input handle hdapSysPut is opened\n");
56  //DAQ ->PC
57  hdapBinGet = DapHandleOpen("\\\\.\\dap0\\$BinOut", DAPOPEN_READ);
58  if (hdapBinGet==0)
59   printf("Error opening DAP binary data output handle hdapBinGet\n");
60   else printf("Output handle hdapBinGet is opened\n");
61  DapLinePut(hdapSysPut,"RESET");
62  DapInputFlush(hdapBinGet);
63  //load .DAP file
64  isConfigured = DapConfig(hdapSysPut, "con_dap.DAP");
65  if (isConfigured){
66  //TDapBufferGetEx control structure
67   TDapBufferGetEx BufControl;
68   DapStructPrepare(&BufControl,sizeof(TDapBufferGetEx) );
69  //At least 3 samples
70   BufControl.iBytesGetMin = sizeof(Type);
71  //As many as " " of these groups
72   BufControl.iBytesGetMax = BUFFER_DIM*sizeof(Type);
73  //Always groups of 3 samples
74   BufControl.iBytesMultiple = sizeof(Type);
75   BufControl.dwTimeWait = TimeMaxWait;
76   BufControl.dwTimeOut = TimeMaxTot;
77   printf("\nConfiguration and Buffer structure are ok\n");
78   ReceiveAvailableData(hdapBinGet,&BufControl,bufferIn,bufferOut);
79  }
80  else printf("error load configuration Dap file\n");
81  }
```

```
// Call this to fetch and deliver any new data that arrive.
// Report the number of new values.
void ReceiveAvailableData(HDAP hDapBinGet,
        TDapBufferGetEx *BufControl,Type*bufferIn,Type *bufferOut)
{
  int total = 0;
  int received;
  int bytes;
  while (1)
  {
   bytes = DapBufferGetEx(hDapBinGet,BufControl,(void *)bufferIn);
   received = (bytes / sizeof(Type));
   DeliveredData = DeliverData(bufferIn,bufferOut,received,bytes);
//total+ = received;     //total number of values
  }
  return;
}
//DeliverData Function
int DeliverData (Type *bufferIn,Type *bufferOut,int received,int bytes)
{
  int nblock=0
  int valuesent = 0;
  if (bytes>0 && received<BUFFER_DIM)
  {
    nblock=(bytes/dim_one_block)-1
    bufferOut[0].brake = bufferIn[nblock].brake;
    bufferOut[0].thr = bufferIn[nblock].thr;
    bufferOut[0].stw = bufferIn[nblock].stw;
    valuesent = received;
    //Sleep (TIME);//Uncomment this line in order to do a cycle-pause.
  }
  else
     if (bytes<=0)
        return (valuesent);
  else
     if (received==BUFFER_DIM)
     {
        printf ("\nValue received: %dl\n  ",received);
        printf ("BUFFER_DIM is reached! Increase BUFFER_DIM
         or you will not ""read"" the Most Recent Value\n");
     }
return (valuesent);
}
```

Appendix D

derived.cpp function

```
1  //Developed by Pasquale Gallo, March-September 2012

2  //Derived function
3  //This function reads steering encoder value from bufferOutDap,
4  //and calculates stw1 and stw2.
5  //stw is converted in double type, steering angle,
6  //and saved to str_pos.
7  //stw1 (double) is saved in str_vel
8  //stw2 (double) is saved in str_ace
9  //(stwFdN=steering wheels first derivative of step N{x})
10 //stwN -> point x(n)
11 //stwN1 -> point x(n-1)
12 //stwN2 -> point x(n-2)
13 //stwN3 -> point x(n-3)
14 //stwn4 -> point x(n-4)
15 //Call:
16 //call derived (str_pos,str_vel,str_ace) in guiado.f90

17 #include "bufferOutDap.h"
18 extern "C" void derived( double *stw, double *stw1, double *stw2);
19 void derived( double *stw, double *stw1, double *stw2)
20 {
21 //mehthod: 4 points backward filtered
22 static double stwN=0,stwN1=0,stwN2=0;
23 static double stwN3=0,stwN4=0,stwN5=0,stw_double;
24 static double stwFdN=0,stwFdN1=0,stwFdN2=0;
25 static dobule stwFdN3=0,stwFdN4=0,stwFdN5=0;
26 static float time=0.005f;        //integration time
27 static int i = 0;
28 const double ENCODER_PULSE_ANGLE=0.18f;
29 //safe conversion type from long to double
30 stw_double=(double(bufferOutDap[0].stw)*-1.0);

31  *stw=(stw_double*ENCODER_PULSE_ANGLE);
32  stwN4=stwN3;
```

```
 stwN3=stwN2;
 stwN2=stwN1;
 stwN1=stwN;  //send x(n) to x(n-1)
 stwN=*stw;  //send stw's most recent value to x(n)
//first derivative variables
 stwFdN4=stwFdN3;
 stwFdN3=stwFdN2;
 stwFdN2=stwFdN1;  // send x(n-1) to x(n-2)
 stwFdN1=stwFdN;   //send x(n) to x(n-1)

//First 4 cycle (cycle needed to capture 5 points)
  if (i<4)
  {
  *stw1=0;
  *stw2=0;
   i++;
  }
  else     //all next cycle
  {
  *stw1=(stwN+2*stwN1-2*stwN3-stwN4)/(16.0*time);
   stwFdN=*stw1;//store stw first derivative most recent value
  *stw2=(stwFdN+2*stwFdN1-2*stwFdN3-stwFdN4)/(320*time);
  }
}
```

Appendix E

compute_par.cpp function

```
1  //Developed by Pasquale Gallo, March-September 2012

2  //Compute_par function
3  //This function computes the motor PAR
4  //input: ap(30) and ap(40) (from simulator),
5  //throttle encoder (from bufferOutDap)
6  //ap(30) and ap(40) are backward wheels angular velocity {rad/sec}
7  //call:
8  //call compute_par (ap(30),ap(40),torque) in guiado.f90

9  #include "bufferOutDap.h"
10 extern "C" void compute_par(double *ap30,double *ap40,double *par);
11 void compute_par(double *ap30,double *ap40,double *par)
12 {
13   short throttle_encoder=bufferOutDap[0].thr;
14   const short int ENC_THR_MAX=420;  //throttle encoder max value
15   const double PI_GRECO=3.14159265;
16   double ap_av=0;                   //ap avarage [rad/s]
17   const float rid=3.673f;           //
18   const float ro=1.475f;            //
19   double rpm;
20   double T,Tc;                      //
21   float f;                          // throttle displacement
22   double ap4,ap3;
23   ap4 = *ap40;
24   ap3 = *ap30;
25   ap_av = (ap3+ap4)/2;
26   rpm = (60*ap_av*rid*ro)/(2*PI_GRECO);
27   T = ((-9.9444E-10)*(rpm*rpm*rpm))-((3.2888E-6)*(rpm*rpm))+
28      +(0.046583*rpm)+53.7;
29   Tc = (-15E-3)*rpm;
30   f = float(throttle_encoder)/float(ENC_THR_MAX);
31   *par = -((T*f)+Tc*(1-f))*ro*rid;
32 }
```

Appendix F

brkPress.cpp function

```
1  //Developed by Pasquale Gallo, March-September 2012

2  //brkPress function
3  //This function reads brake encoder value from bufferOutDap (short)
4  //and sends it to guiado's brk_press (double) after bar conversion.
5  //Call:
6  //call brkPress(brk_press) in guiado.f90

7  #include "bufferOutDap.h"
8  #define D_MAX_VALUE 32767   //max digital value
9  #define MAX_BAR 40
10 extern "C" void brkPress(double *brk_press);
11 void brkPress (double *brk_press)
12 {
13   double brk_pressDouble;
14   if (bufferOutDap[0].brake>=0)
15   {
16    brk_pressDouble = double(bufferOutDap[0].brake);
17 //bar conversion
18    *brk_press = ((brk_pressDouble*MAX_BAR)/D_MAX_VALUE);
19   }
20 //if encoder value is <0 brk_press=0
21   else *brk_press = 0.00000;
22 }
```

time.cpp function

```
1  //Developed by Pasquale Gallo, March-September 2012

2  extern "C" void time_c(double *timeOut);
3  void time_c(double *timeOut)
4  {
5    static double time = 0.0;
6    *timeOut = time;
7    time = time+0.005; }
```

Bibliography

Alonso, J., F. Romero, R. Pàmies-Vilà, U. Lugrís, and J. Font-Llagunes (2012). A simple approach to estimate muscle forces and orthosis actuation in powered assisted walking of spinal cord-injured subjects. *Multibody System Dynamics 28*, 109–124. 10.1007/s11044-011-9284-5.

Arnold, M. and W. Schiehlen (2009). *Simulation Techniques For Applied Dynamics*, Volume 507 of *CISM Courses and lectures*. Springer.

Bae, D. S., J. Lee, H. Cho, and H. Yae (2000). An explicit integration method for realtime simulation of multibody vehicle models. *Computer Methods in Applied Mechanics and Engineering 187*(1), 337–350.

Baumgarte, J. (1972). Stabilization of constraints and integrals of motion in dynamical systems. *Computer Methods in Applied Mechanics and Engineering 1*(1), 1 – 16.

Bayo, E., J. García de Jalón, A. Avello, and J. Cuadrado (1991). An efficient computational method for real time multibody dynamic simulation in fully cartesian coordinates. *Computer Methods in Applied Mechanics and Engineering 92*(3), 377 – 395.

Bayo, E., J. García de Jalón, and M. A. Serna (1988). A modified lagrangian formulation for the dynamic analysis of constrained mechanical systems. *Computer Methods in Applied Mechanics and Engineering 71*(2), 183 – 195.

Bayo, E. and R. Ledesma (1996). Augmented lagrangian and mass-orthogonal projection methods for constrained multibody dynamics. *Nonlinear Dynamics 9*(1), 113–130.

Bellini, A. and A. Guidi (2009). *Linguaggio C*. McGraw-Hill.

Carbone, V., L. Primavera, and F. Stabile. Appunti di metodi numerici. http://www.fis.unical.it/astroplasmi/carbone.pdf. [Online, accessed March-2012].

Cossalter, V., M. D. Lio, and A. Doria (2006). *Meccanica Applicata alle Macchine*. Edizioni Progetto.

Cuadrado, J. (2012). Curso de multibody. Universidade da Coruna, On line material course.

Cuadrado, J., J. Cardenal, and E. Bayo (1997). Modeling and solution methods for efficient real-time simulation of multibody dynamics. *Multibody System Dynamics 1*(3), 259–280.

Cuadrado, J., D. Dopico, J. Perez, and R. Pastorino (2012). Automotive observers based on multibody models and the extended kalman filter. *Multibody System Dynamics 27*, 3–19. 10.1007/s11044-011-9251-1.

Cuadrado, J., R. Gutiérrez, M. A. Naya, and P. Morer (2001). A comparison in terms of accuracy and efficiency between a mbs dynamic formulation with stress analysis and a non-linear fea code. *International Journal for Numerical Methods in Engineering 51*(9), 1033–1052.

Flores, P., J. Ambrósio, J. C. P. Claro, and H. M. Lankarani (2008). *Kinematics and Dynamics of Multibody Systems with Imperfect Joints*, Volume 34. Springer.

Font-Llagunes, J. M., R. Pàmies-Vilà, J. Alonso, and U. Lugrís (2011). Simulation and design of an active orthosis for an incomplete spinal cord injured subject. *Procedia IUTAM 2*(0), 68 – 81. IUTAM Symposium on Human Body Dynamics.

García de Jalón, J. and E. Bayo (1994). *Kinematic and Dynamic Simulation of Multibody Systems*. Springer-Verlag.

García de Jalón, J., J. Unda, and A. Avello (1986). Natural coordinates for the computer analysis of multibody systems. *Computer Methods in Applied Mechanics and Engineering 56*(3), 309 – 327.

Guiggiani, M. (2007). *Dinamica del Veicolo*. CittáStudi Edizioni.

Haug, E. J. (1989). *Computer aided kinematics and dynamics of mechanical systems*, Volume 1: basic methods. Allyn and Bacon.

Heitel, H. and P. J. Heitel (1999). *C++ How to Program*. Prentice Hall.

Holoborodko, P. (2008). Smooth noise robust differentiators. http://www.holoborodko.com/pavel/numerical-methods/numerical-derivative/smooth-low-noise-differentiators/. [Online, accessed June-2012].

Hong, S., H. W. Kim, Y. S. Cho, H. J. Cho, J. H. Jung, and D. S. Bae (2011). Development of realtime simulator for multibody dynamics analysis of wheeled vehicle on soft soil. *Journal of Ocean Engineering and Technology 25*(6), 116–122.

Kane, T. R. and D. Levinson (1985). *Dynamics: theory and applications*. McGraw Hill.

Korkealaakso, P. (2009). *Real time simulation of mobile and industrial machines using the multibody simulation approach*. Ph. D. thesis, Lappeenranta University of Technology.

Lio, M. D. and R. Lot (1999). Analisi modale di sistemi multibody descritti in coordinate naturali. Technical report, XIV Congresso Nazionale dell'Associazione Italiana di Meccanica Teorica ed Apllicata.

Microstar Laboratories. *DAP 4200a Manual* (1.01 ed.). Microstar Laboratories. http://www.mstarlabs.com/docs/manuals/DAP4200A.PDF. [Online, accessed March-2012].

Microstar Laboratories. *DAPIO32 Reference Manual* (4.14 ed.). Microstar Laboratories. http://www.mstarlabs.com/docs/manuals/DAPIO32.PDF. [Online, accessed March-2012].

Microstar Laboratories. *DAPL 2000 Manual* (6.00 ed.). Microstar Laboratories. http://www.mstarlabs.com/docs/manuals/DAPL2000.PDF. [Online, accessed March-2012].

Microstar Laboratories. *Expansion Boards Manuals*. Microstar Laboratories. http://www.mstarlabs.com/docs/manuals.html. [Online, accessed April-2012].

Natalini, R. Introduzione ai Metodi Numerici alle Differenze Finite. http://www.dmmm.uniroma1.it/pubblicazioni/doc/phd_quaderni/04-02-nat.pdf. [Online, accessed April-2012].

Naya, M. A. (2007). *Aplicación de la dinámica multicuerpo en tiempo real a la simulación y el control de automóviles*. Ph. D. thesis, Universidade da Coruña.

Nikravesh, P. E. (1988). *Computer-aided analysis of mechanical system*. Prentice Hall.

Orden, J., J. Goicolea, and J. Cuadrado (2007). *Multibody dynamics: computational methods and applications*. Computational methods in applied sciences. Springer.

Pastorino, R. (2012, June). *Experimental validation of a multibody model for a vehicle prototype and its application to automotive state observers*. Ph. D. thesis, Universidade da Coruña.

Pastorino, R., M. Ángel Naya, A. Luaces, and J. Cuadrado (2010). X-by-wire vehicle prototype: automatic driving maneuver implemantation for real-time MBS model validation. In *Proceedings of the 515th EUROMECH Colloquium*.

Pastorino, R., M. A. Naya, J. A. Pèrez, and J. Cuadrado (2011). Geared PM coreless motor modelling for drivers force feedback in steer-by-wire systems. *Mechatronics 21*(6), 1043 – 1054.

Rauh, J. (2003). Virtual development of ride and handling characteristics for advanced passenger cars. *Vehicle System Dynamics 40*(1-3), 135–155.

Sanjurjo, E. M. (2011). Modelo multicuerpo de automóvil para su aplicación en técnicas de estimación de estados. Master's thesis, Universidade da Coruña.

Shabana, A. (1989). *Dynamics of multibody system*. Wiley.