ESCUELA POLITÉCNICA SUPERIOR

UNIVERSITY OF LA CORUÑA

EFFICIENT IMPLEMENTATIONS AND CO–SIMULATION TECHNIQUES IN MULTIBODY SYSTEM DYNAMICS

A thesis submitted for the degree of
Doctor Ingeniero Industrial

FRANCISCO JAVIER GONZÁLEZ VARELA

Ferrol, March 2010

ESCUELA POLITÉCNICA SUPERIOR

UNIVERSITY OF LA CORUÑA

EFFICIENT IMPLEMENTATIONS AND CO–SIMULATION TECHNIQUES IN MULTIBODY SYSTEM DYNAMICS

A thesis submitted for the degree of
Doctor Ingeniero Industrial

Francisco Javier González Varela

Advisor: Javier Cuadrado Aranda
Co–Advisor: Manuel Jesús González Castro

Ferrol, March 2010

A mi familia

ACKNOWLEDGEMENTS

After four years and a half of work, one gets the impression that, to be fair, the list of names in this acknowledgements section should take up more space than the rest of the chapters. I must mention here the FPU grant from the Spanish Ministry of Education, which has allowed me to carry out the research of this thesis, but of course there is much more to acknowledge.

First, I wish to thank my advisors, Javier and Lolo, for the attention, the confidence and the appreciation for well finished work they have transmitted to me. Together with them, I also want to thank my colleagues at the Laboratory of Mechanical Engineering of the University of La Coruña: Alberto, Amelia, Dani, José Antonio, Miguel, Roland and Urbano. I owe them a lot, both in the professional and the personal scope, and they deserve all my gratitude. Thank you all!

I am very grateful to Professors Aki Mikkola and Kurt Anderson, as well as to the people in their work teams, for the fruitful and gratifying research stays I have done in Lappeenranta and Troy.

I cannot forget all the people that, one way or another, have supported me all this time, even when I have not been fully aware of it. And last, I would like to thank God and my family, specially my parents Antonio and María Jesús and my sister Inés, for their tireless patience, and because they have always been by my side when I have needed them.

AGRADECIMIENTOS

Al volver la vista atrás después de cuatro años y medio de trabajo, uno tiene la impresión de que, para ser justos, la lista de agradecimientos de esta tesis debería ocupar más espacio que los demás capítulos juntos. Es obligado hacer referencia aquí a la beca FPU del Ministerio de Educación, que me ha permitido llevar a cabo la investigación de esta tesis. Pero, claro, hay mucho más que añadir.

En primer lugar, quiero agradecer a mis directores de tesis, Javier y Lolo, la atención que me han dedicado y la confianza y el aprecio por el trabajo bien hecho que me han transmitido. Este agradecimiento se extiende también a mis compañeros del Laboratorio de Ingeniería Mecánica de la Universidad de La Coruña –Alberto, Amelia, Dani, José Antonio, Miguel, Roland y Urbano– que se han ganado a pulso un reconocimiento muy especial por mi parte; he recibido mucho de vosotros en el plano profesional y en el personal. ¡Gracias a todos!

A los profesores Aki Mikkola y Kurt Anderson, así como a las personas de sus equipos de trabajo, les debo que mis estancias de investigación en Lappeenranta y Troy hayan sido tan fructíferas y gratificantes.

Además, no puedo olvidarme de todos los que, de un modo u otro, me han apoyado durante todo este tiempo, incluso cuando yo mismo no haya sido muy consciente de ello. Y para concluir, quiero dar las gracias a Dios y a mi familia, sobre todo a mis padres Antonio y María Jesús y a mi hermana Inés, por su paciencia inagotable y porque siempre han estado ahí cuando los he necesitado.

A todos, una y mil veces más, ¡gracias!

ABSTRACT

Current research in simulation of multibody systems (MBS) dynamics is focused on two main objectives: the increase of the computational efficiency of the software that carries out the simulations, and the diversification of the problems this software is able to tackle, sometimes through the inclusion in the calculations of non purely mechanical phenomena. This work deals with these two objectives, studying the effect of source code implementation in software performance, as well as the different communication methods with external software that can contribute the interaction of the MBS code with elements of a non mechanical nature.

The first Chapter of this thesis contains a brief introduction to the present state of the art of MBS simulation software. It introduces the research lines this thesis forms part of and outlines its main objectives.

The second Chapter describes the software architecture for MBS simulation that has been developed for this research. The C++ language has been used for its implementation, according to the object–oriented programming paradigm. This Chapter also enumerates the programming tools employed in the process and draws general conclusions about the development of MBS programs.

The third and fourth Chapters introduce efficient implementation techniques in the field of linear algebra operations, and in the parallelization of the code, respectively. The obtained improvements in performance have been quantified, and the range of application of each technique has been delimited.

The fifth and sixth Chapters deal with the communication of the developed software with external simulation software packages. A comparative study between the different ways in which the coupling can be performed has been carried out and, besides, the impact on the efficiency and accuracy of the use of multirate co–simulation techniques has been assessed. A generic interface for multirate integration has been designed to link the MBS software with MATLAB/Simulink, a mathematical and block diagram package very popular in the multibody community.

Finally, the seventh Chapter summarizes the conclusions of the present work, and proposes future research lines that can be derived from it.

RESUMEN

La investigación actual en simulación dinámica de sistemas multicuerpo (MBS) gira en torno a dos objetivos principales: el incremento de la eficiencia computacional del software que lleva a cabo las simulaciones y la diversificación de las tareas que este es capaz de realizar, a veces mediante la inclusión en los cálculos de fenómenos no puramente mecánicos. Este trabajo aborda ambos objetivos, estudiando el efecto de la implementación del código fuente en el rendimiento del software, así como las diferentes estrategias de comunicación con programas externos que puedan aportar a la simulación multicuerpo la interacción con elementos de naturaleza no mecánica.

El primer capítulo de esta tesis consiste en una breve introducción al estado actual del software para simulación de sistemas multicuerpo. En él se muestran también las líneas de investigación en las que se enmarca el proyecto y se señalan sus objetivos principales.

El segundo capítulo describe la arquitectura del software para la simulación de sistemas multicuerpo que se ha creado en esta investigación. Para su implementación se ha utilizado el lenguaje C++, dentro del paradigma de programación orientada a objetos. En este capítulo se enumeran también las herramientas de programación utilizadas en el proceso y se obtienen conclusiones de validez general para la generación de programas multicuerpo.

Los capítulos tercero y cuarto presentan técnicas de implementación eficiente de las operaciones de álgebra lineal y en la paralelización del código, respectivamente. Se han cuantificado las mejoras en el tiempo de ejecución obtenidas y se han delimitado los campos de aplicación de cada estrategia.

En los capítulos quinto y sexto se aborda la comunicación del software desarrollado con otros programas de simulación externos. Se ha realizado un estudio comparativo de los diversos modos posibles en que se puede realizar esta unión y, además, se ha evaluado el impacto del empleo de técnicas de cosimulación *multirate* sobre la eficiencia y la precisión de los cálculos. Se ha diseñado para ello una interfaz genérica entre el software MBS y MATLAB/Simulink, una aplicación matemática y de diagramas de bloques de gran aceptación entre la comunidad de investigación en sistemas multicuerpo.

Por último, en el capítulo séptimo se resumen las conclusiones del presente trabajo y se proponen líneas de investigación futuras que pueden derivarse de él.

Contents

Chapter 1

Introduction

The simulation of multibody systems (MBS) is a very active field of Mechanics, intensely evolved since the late 1960s thanks to the improvements in computing hardware and software. The simulation of multibody systems enables the prediction of the kinematic and dynamic behaviour of a mechanical system from its physical definition, avoiding the need for building a real prototype of the mechanism, thus shortening the development process of the product and reducing costs. Due to the considerable computational effort required to formulate and solve the equations that rule the motion of mechanical systems when large rotations are present, the simulation of even relatively small problems was considered impractical until the appearance of digital computing. Since then, the advances in architecture of computers and software engineering have converted the task of simulating multibody systems in everyday work for many research groups, and MBS simulation is now common in product development in industry. On the other hand, this work has become heavily dependent on the quality and features of the available software.

The complexity of the models the researchers in multibody systems have to deal with is continuously growing, as the degree of detail in simulations required by industrial applications increases. Nowadays, mechanical engineers need more and more realistic simulations, which leads to highly demanding requirements for the multibody simulation software. In the last fifteen years, the range of research topics has notably widened, and new phenomena are now considered and included in simulation models, such as flexibility, contact, impacts and interaction with non–mechanical components (Schiehlen, 2007). These features add a higher degree of realism to the simulation, but they usually represent also an extra computational burden that the software has to tackle. Presently, the lines of research in the field of multibody systems can be grouped into two main categories: the development of methods to add new functionality to conventional MBS software in a reliable and precise way; and the design of new formalisms to improve the efficiency of the simulations, sometimes aiming at the exacting goal of performing the simulation in real time.

As it can be deduced from the previous paragraphs, software efficiency is a crucial component in multibody research. A poor implementation can seriously hinder the

performance of a carefully designed formalism, spoiling its potential advantages and preventing it from being used in non–trivial applications. Conversely, a rational use of programming techniques reduces the computational effort required to solve a particular problem, smoothing the path for the addition of new features or the fulfilment of tougher requisites in the simulation.

1.1 Motivation

The spectacular progress experienced by MBS research during the last years has given rise to a considerable number of software packages for the simulation of multibody systems (McPhee, 2008). Several commercial, off–the–shelf packages are available, such as Simpack (SIMPACK AG, 2009), MSC.Adams (MSC.Software Corporation, 2009), SAMCEF mecano (SAMTECH, 2009) or RecurDyn (Function Bay, Inc., 2009). These programs are efficient and versatile, and they undergo periodical enhancement. Additionally, some software tools, originally not designed as multibody programs, are incorporating this functionality through specific complements and modules. Examples of the stated are the SimMechanics library for MATLAB's block diagram tool Simulink (The Mathworks, Inc., 2009) and the applications for the analysis of mechanisms that are nowadays common in many CAD/CAE packages, among others.

While these programs are specially useful for simulating particular problems, of great complexity sometimes, they are not suitable as a platform for testing new formalisms or developing extra functionalities. Moreover, the lack of a neutral data format for the exchange of files between packages from different vendors frequently makes it impossible to use data from a package in another one. Thus it is that many academic research teams have written, and keep on writing, their own codes in order to assess the validity and efficiency of their dynamic formulations and integrators. Many researchers have also developed code to deal with the new functionalities required by multibody simulations on the basis of their in–house codes. Some of them have reached a high degree of maturity and are being used in real–life, industrial applications (Anderson et al., 2007; Dipartamento di Ingegneria Aerospaziale, Politecnico di Milano, 2009). However, although research in the multibody community has been traditionally focused in the definition of new efficient formalisms, very few studies about the impact of implementation details on performance have been carried out. Currently, there is a lack of available information about this subject in literature, so multibody research groups are compelled to draw their own conclusions based on subjective experience. This leads to a multiplication of efforts and increases the risk of using inadequate solutions in practice.

As a consequence of the stated, it is necessary to correctly evaluate the impact of code implementations on the reliability and efficiency of the developed MBS simulation software, as well as the possibilities the state of the art provides with regard to the enhancement of the features this software can offer. This thesis continues the work the Laboratory of Mechanical Engineering of the University of La Coruña has been carrying out on this field during the last years. This task was initiated with the work

of González (2005), who proposed some steps towards the efficient development of MBS software:

- The use of a neutral and extendable data format for the modelling of multibody systems, which can be used for the exchange of models between users of different simulation programs, both academic and commercial (González et al., 2007).

- The building of a benchmark to evaluate the efficiency of the different existing simulation methods, made up of a collection of mechanism simulation problems and a routine to obtain comparable parameters of efficiency in different simulation environments (González et al., 2006, 2010).

- The elaboration of a software architecture prototype for the development of optimized MBS simulation codes.

The third item in the list has been addressed during the elaboration of the present thesis, as the development of a software architecture provides an adequate environment for testing different implementations and for the identification of the problems that commonly arise during the writing of MBS simulation codes. The software itself, however, is not the main objective of this work, but an intermediate tool to study the effect of different implementations and find out general guidelines for the optimization of MBS software. The different ways of performing linear algebra routines, the possibilities the parallelization of code offers and the communication with external software constitute the main focuses of attention of the thesis.

1.2 Objectives

The objectives of this thesis can be summarized in the following ones:

- To assess the effect of code implementation on the overall performance of multibody software and to identify commonly used patterns of code in MBS programs that can be systematically enhanced using simple coding techniques. The obtained improvements must be measured and used to find out guidelines that can help the researchers of the multibody community to efficiently implement MBS simulation codes, in particular when the improvement can be performed without affecting substantially to the structure of the software.

- To evaluate the ways in which MBS software developers can add extra functionality to their codes via communication with external packages and libraries. This objective includes the review of the currently available coupling techniques between software tools and their comparison, and also the development of synchronization strategies between different codes, when needed. A special attention must be paid to block diagram simulators, because of their versatility and wide acceptance in the research community.

- To start the development of a software architecture for the simulation of multibody systems, in which the previous objectives can be studied. This architecture must be open and modular, in order to allow the extension of its features through the addition of new modules or the streamlining of the existing ones. The software must also serve to test and develop new MBS formulations and integrators.

The results obtained during the realization of this work are intended to yield useful information for the multibody community on the subject of efficient and versatile software implementations.

1.3 Structure

This thesis is structured into the following Chapters:

- Chapter 1: Introduction.
- Chapter 2: Design of a Software Architecture for MBS Simulation.
- Chapter 3: Linear Algebra Implementation.
- Chapter 4: Parallelization.
- Chapter 5: Integration with MATLAB/Simulink.
- Chapter 6: Multirate Co–simulation Methods.
- Chapter 7: Conclusions.

Chapter 1 briefly describes the currently active lines of research in the field of MBS dynamics and highlights the importance of the software in multibody research. The motivation and the objectives of this work are pointed out, and the layout of the thesis is presented.

Chapter 2 deals with the simulation software that has been developed for the achievement of the goals of this work. In the light of the software requirements, a programming language, work methodology and software structure have been chosen. The resulting layout and the features of the code are then described and general conclusions about the development of multibody software are exposed.

Chapter 3 discusses the impact of the use of different linear algebra implementations on the overall efficiency of MBS simulation codes. A benchmark problem is set up, and different configurations of the software, relative to matrix storage and basic algebraic routines, are tested and compared. Guidelines for the selection of the most convenient configuration, as a function of the problem nature (number of variables and number of non–zeros in matrices), are established at the end of this Chapter.

Chapter 4 assesses the suitability of improving the performance of the simulation through the use of *non–intrusive* parallelization techniques. Multi–threaded linear solvers and OpenMP directives have been applied to the software, and conclusions about their performance have been drawn.

Chapters 5 and 6 deal with the addition of functionality to multibody codes through communication with external software. Several ways of exchanging data during runtime between the MBS software and a general purpose mathematical package (MATLAB/Simulink) are described and compared. In Chapter 6, an interface for multirate co–simulation is described and tested.

And, finally, Chapter 7 summarizes the conclusions and results of this work and points out presently open lines of future research.

Chapter 2

Design of a Software Architecture for MBS Simulation

In order to achieve the goals identified in the Introduction to this thesis, a new software for the simulation of multibody systems has had to be developed. Although many MBS simulation software packages exist, some of them open–source, the design of a new one from scratch seemed convenient for several reasons. First, existing MBS codes have gone through long development and optimization processes, of years of duration in many cases. During these processes, decisions have been taken about the storage format to be used, the way in which interfaces are defined and many other implementation details. The knowledge background on which these decisions were taken is frequently not accessible to users, even in the case the code is available for downloading. Second, existing software is hardly ever prepared for the easy replacement of components at a low programming level in the code, such as the storage format or the way in which basic algebraic operations are performed. This makes difficult the test of alternative code implementations. Consequently, a new software for the simulation of MBS dynamics has been designed and used in this research.

2.1 Analysis of software requirements

The developed MBS software must comply with the following requirements:

- It must be *modular*, in order to simplify the substitution or modification of any of its components without carrying out significant modifications in its main structure. As the MBS software is designed to test different alternative implementations of the same components, its structure must be flexible enough to permit the replacement of the formulation, the integrator or the numeric methods for the solution of systems of equations without substantially affecting the

rest of the elements of the program, preferably in an easy to revert way.

- It must be *open*, to allow the addition of new functionality through the use of sub–programs or modules written in any common programming language (Fortran, C or C++), as well as the communication with other simulation and calculus tools, such as CAD programs or block diagram software.

- It must be *collaborative*, so many researchers can develop the software simultaneously and coordinately, even when they belong to geographically scattered research teams.

- Finally, it is desirable that the software is *platform–independent*, so it can be compiled and used under different operating systems and computer architectures.

The first two features are necessary for assessing the effect of different code implementations and testing techniques of communication with external software packages. They will also allow the future development of the project, because they make possible to add new modules and interfaces to external programs. This growth will be considerably facilitated by the fact that the software is built in a collaborative way, since the sharing of information and code among groups will not be hindered by constraints on the dissemination of knowledge. The fourth condition also makes easier the use of the software, as it is not conditioned by the availability of a determined compiler or operating system; moreover, it will allow the generalization of the obtained results, as their validity will not be confined to a particular configuration of the computer system in which they were obtained.

2.1.1 Programming language

The modularity the software requires can be obtained by designing the architecture of the code according to the *object–oriented paradigm*. Object–oriented languages provide the following features, all of which positively help the software modularity:

- *Inheritance*. The hierarchical structure of the software can be constructed on the basis of deriving new classes (*derived classes*) from existing ones (*base classes*). It is possible to create abstract classes that define the basic structure of the main components of the software, and to instantiate particular implementations of these components later, by adding specific functionality to the original base class. For example, a base class *Integrator* can be created, defining the basic functionality of the integrators and the methods they must declare, and then several instances of integrators derived from it, implementing the actual integration routines. *Multiple inheritance* can also be used to combine the features of two existing base classes.

- *Encapsulation*. The use of correctly designed base classes leads to the definition of interfaces, which control the interaction of the components of the software.

The use of interfaces makes the communication between components independent of the particular inner operations of each of them, and allows their non–intrusive substitution. It also facilitates the modification of the code inside of a particular component (e.g. the way in which an integrator performs the iterations to achieve convergence) without affecting the structure of the rest of the program.

- *Polymorphism*. Polymorphism allows the same function to be executed on different object types. For example, a function for adding matrices can receive two arguments representing dense matrices, but the same function can be defined to operate on sparse matrices. This simplifies the encapsulation of the code through the definition of standard interfaces.

- *Reusability*. The three enunciated characteristics of object–oriented languages result on an increment of the reusability of the code: classes and functions can be used in different applications without substantial modifications.

The object–oriented paradigm has been successfully applied to MBS dynamics in several works. Kecskeméthy and Hiller (1995) used the approach to simulate vehicle dynamics in a modular way; Han and Seo (2004) employed it in the generation of the equations of motion; and recent applications of the method in biomechanics have been developed by Tändl et al. (2009). The features of object–oriented languages have led many researchers to use them to implement their MBS software. This is the case of POEMS (Anderson et al., 2007), a modular multibody software that aims to work as a repository of efficient implementations of MBS algorithms, today a part of the LAMMPS molecular dynamics simulator (Sandia National Laboratories, 2009). MBDyn (Dipartamento di Ingegneria Aerospaziale, Politecnico di Milano, 2009) is another case of modular and open MBS software, with additional support for aeroelastic, hydraulic, electric and control problems. As a last example, SimTK (Simbios project, 2009) core module relies on the same principles of abstraction and modularity to build a highly–efficient multibody software, aimed at the simulation of biomechanical systems.

All the applications mentioned in the previous paragraph have been coded in C++; in the present work, C++ has been selected as programming language too. Its object–oriented features are missing in Fortran and C languages, commonly used in the multibody community, but more focused on procedural programming (Cary et al., 1997). Moreover, regarding efficiency, a software package written in C++ can make use of libraries coded in these more efficient languages in a relatively straightforward way. Thus, the slight penalty in performance due to data abstraction in C++ codes (under 5%, with respect to plain C) can be overcome via the use of external highly efficient routines, written in C or Fortran, when necessary. Additionally, C++ includes other helpful characteristics such as the possibility of using *templates* and the standardized management of exceptions.

An additional reason for selecting C++ as programming language is the considerable amount of available documentation and developing tools of good quality, many of

them freely available in the Net. The design of the software can make use of standardized solutions and design patterns, such as those described by Meyers (1999, 2000) and Alexandrescu (2001). The C++ Standard Library (STL) provides a set of common classes and interfaces that greatly extend the core C++ language (Josuttis, 1999), and other popular libraries have been recently proposed for standardization (Dawes et al., 2009). By reusing these elements, the software developer can accelerate the writing of the code with robust, already tested solutions.

The fact that the software is intended to be platform–independent must be borne in mind during the design process of the code, avoiding implementations of C++ specific of a certain compiler or operating system, at least in the case of the main components defined in the core module, to keep the code portability. In the present work, the software has been designed and tested to run under both Windows and Linux systems.

2.1.2 Methodology and development tools

The final requirements of the software in a research project are frequently difficult to foresee when the programming of the code begins; in many cases, they are discovered as the work progresses. This fact could be seen as an inconvenience; in this work, however, it has been useful in order to evaluate the actual degree of modularity of the software and the ability of the chosen programming solutions to be adapted to new necessities. For this reason, many characteristics of the MBS software have been added as new requirements were raised, instead of planning every detail completely from the start of the project. This flexible methodology has revealed itself to be very effective, even when new features of considerable entity have been incorporated to the code, as it was the case of the addition of support for sparse storage to the original code for dense matrices, or the introduction of a new module for the automatic generation of equations.

Regarding the development and auxiliary tools that have been used in the present project, the following ones are worthy to mention because of their critical impact on code performance and maintenance:

- *Compilers*, responsible for building libraries and executables from the source code.
- *Build–process managers*, which control the creation of the projects that rule the compilation of the code.
- *Version control systems*, for keeping track of the changes in the code, especially when several developers work on the same project.
- *Documentation systems*, which automate the generation of the reference documents.

The tools required by a truly open and collaborative software must be easy to obtain, so the lack of a license or a highly expensive retail price are not an obstacle for building the program. For this reason, open source tools, accessible through the Net, have been preferred for this study.

Compilers

As the tool that converts the C++ source code into libraries and executables, the compiler is a key component in the building of an MBS simulation software. Most compilers can carry out their own particular optimizations in the code, so they are also a factor to consider when measuring the performance of a program. Moreover, some specific techniques, such as the parallelization via OpenMP described in Chapter 4, are only supported by certain compilers. The selection of a compiler is also a function of the operating system and the computer architecture. The use of Windows– and UNIX–based compilers helps checking the portability of the code: in this research, this has been done with Microsoft Visual Studio (Microsoft, 2009) and GCC (Free Software Foundation, 2009).

Profiling tools are a complement to the compiler, particularly useful to detect bottlenecks that slow down the execution of the code, and also those parts of the software most adequate for optimization. Their characteristics vary greatly, and not many of them, with an suitable quality, are open source or freely distributed. Valgrind (Valgrind developers, 2009) is an exception and has been used in this research.

```
# Additional include directories
include_directories ( ${MBSLAB_HOME} )

# Files that form the executable
set ( Double_Pendulum_FILES
./Double_Pendulum.cpp
./Double_Pendulum.hpp
./DP_driver.cpp )

# Create executable from files
add_executable ( Double_Pendulum
${Double_Pendulum_FILES} )

# Link executable to core library
target_link_libraries
( Double_Pendulum mbscore )
```

Figure 2.1: *Example of CMake directives for the generation of an executable*

Build–process managers

The diversity of compilers that can be used for building the software has given rise to an additional problem, unexpected when this work was started. Every compiler requires a project file or a series of *makefiles* to regulate its work, deciding which files are to be compiled, and which options must be used during the process. The maintenance of these project files can become a cumbersome, annoying task when the number of supported compilers in the software project grows. Build–process manager systems solve this problem, automatically building the project files or *makefiles* from scripts

that are independent from the operating system and the compiler. CMake (Martin and Hoffman, 2007) has been selected for this work; an example of its directives for ruling the compilation of the code is displayed in Figure 2.1. As the figure shows, CMake directives must be coded in its own language, but the small effort the programmer must make to learn it is clearly compensated by the advantages of having just one set of script files, valid for guiding the compilation of the code in a considerable number of common operating systems and compilers.

Version control systems

The use of version control systems is recommended when developing software in a collaborative way, to synchronize the additions and modifications in the code. Even when the development is not collaborative, these systems can still be used to keep track of the changes in the code, test modifications and revert them when needed. Subversion (SVN) (Tigris.org, 2009) has been used as it has replaced the traditional Concurrent Versions System (CVS) as version control standard. As this work was being developed, new control systems have arisen, which avoid requiring a central repository of source code through the use of distributed architecture. This means that each developer has a local copy of the entire history of the software. Examples of distributed systems of control version are Mercurial (Selenic Consulting, 2010) and Bazaar (Canonical Ltd., 2010); currently they can be found in a mature state and used for the development of complex code projects.

Documentation systems

Finally, the use of a semi–automatic documentation system simplifies the task of writing reference files for each C++ element created in the project, though a minimum amount of work in this field must always be done by developers. The software in this thesis is documented with Doxygen (van Heesch, 2009). This tool parses special comment lines placed in the C++ source files of the software and in text files, describing the way in which classes, functions and other components work. It is configurable and its portability allows documenting the code with the same commentaries, independently of the platform where the software is developed. The comments inserted in the code are used by Doxygen to generate documents in HTML or TEX format that can be later improved by the programmers with their explanatory notes.

2.2 Description and general structure

The basic core of the developed multibody software is a general purpose program aimed at the simulation of generic multibody mechanisms going through large rotations and highly non–linear equations. For a multibody system, defined by a set of generalized coordinates $\mathbf{q}$, the nonlinear equations of motion can be written as a sys-

tem of Differential Algebraic Equations (DAE) as follows:

$$\begin{aligned} \mathbf{M}\ddot{\mathbf{q}} + \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\lambda} &= \mathbf{Q} \\ \mathbf{\Phi} &= \mathbf{0} \end{aligned} \tag{2.1}$$

where $\mathbf{M}$ is the mass matrix of the system, $\ddot{\mathbf{q}}$ is the vector of second derivatives of the generalized coordinates (accelerations), $\mathbf{\Phi}$ is the vector of constraint equations of the system, $\mathbf{\Phi_q}$ is the Jacobian matrix of the constraint equations with respect to the generalized coordinates, $\boldsymbol{\lambda}$ is the vector of Lagrange multipliers, and $\mathbf{Q}$ is the vector of generalized forces that applies to the generalized coordinates. Equations (2.1) cannot be directly managed by most integrators, so they must be converted into Ordinary Differential Equations (ODE) through the use of a convenient dynamic formulation, as it will be explained in Section 2.3.

The simulation software is designed to be able to manage different types of coordinates. In this work, however, natural coordinates, global and dependent (García de Jalón and Bayo, 1994), are used for modelling the systems. Natural coordinates describe the position of the elements of the system by means of basic points and unit vectors associated with the bodies of the system. For this reason, there is no need for the use of rotation parameters, such as Euler angles, to describe the rotation of the bodies.

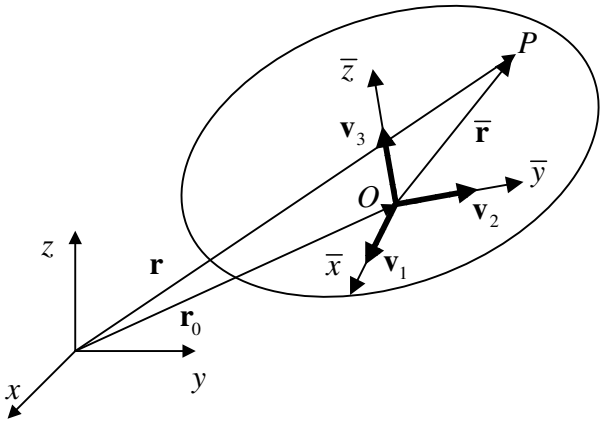

Figure 2.2: *Generic rigid body, parameterized with natural coordinates*

A description in natural coordinates of a generic rigid body can be seen in Figure 2.2; the use of natural coordinates for flexible bodies is described in detail by Cuadrado et al. (1996). The global position of an arbitrary particle of the rigid body P can be expressed by its position vector $\mathbf{r}$. If the unit vectors $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$, which do not need to be co–lineal with the local axes of the body ($\bar{x}$, $\bar{y}$ and $\bar{z}$), form a base of the local frame of reference, the position of the particle P can be expressed as a linear

combination of these vectors, in the following way:

$$\mathbf{r} = \mathbf{r_0} + \bar{\mathbf{r}} = \mathbf{r_0} + \alpha\mathbf{v_1} + \beta\mathbf{v_2} + \gamma\mathbf{v_3} \tag{2.2}$$

where $\mathbf{r_0}$ is the position vector of the origin of the local frame of reference of the body, $\bar{\mathbf{r}}$ is the position vector of particle P in the local frame of reference, and α, β and γ are constant coefficients of linear combination. Using a proper selection of the basic points and unit vectors of the body, the mass matrix $\mathbf{M}$ remains constant during large rotations. This unique feature of the natural coordinates simplifies the equations of motion since inertial forces that depend quadratically on velocities do not appear in them.

Finally, constraint equations representing kinematic joints between bodies must be added in order to complete the modelling of the system. These equations are defined in the constraints vector, $\mathbf{\Phi}$.

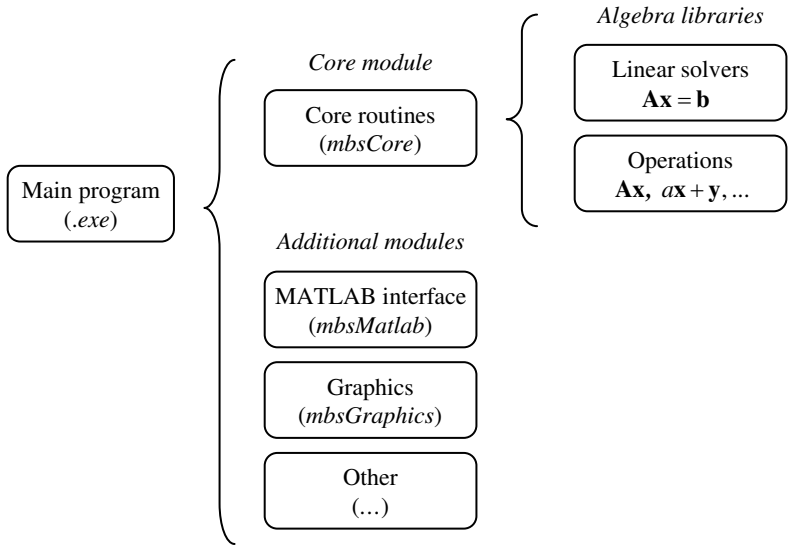

Figure 2.3: *General layout of MBS simulation software*

The main structure of the MBS simulation software is depicted in Figure 2.3. The software is designed as a modular series of libraries, each of which is responsible for a particular task in the simulation. The main library is designed as the *core module* and it is described in detail in Section 2.3; it contains the numeric integration routines, the multibody dynamic formulations for converting the DAE's in Equation 2.1 into a set of ODE's, and the basic components for building the models of the multibody systems to be simulated. The core module, in turn, invokes basic algebraic functionality, such as routines for the solution of linear systems and matrix–vector operations, from a set of *algebra libraries*. These libraries are third–party software, and their selection and tuning are described in Chapters 3 and 4.

In addition to the fundamental multibody facilities contained in the core module,

the functionality of the software can be further expanded through the definition and implementation of *additional modules*. Thus, libraries can be defined that add communication with external software (as it is the case of the MATLAB interface described in Chapter 5), manage the graphic representation of the results or perform pre– or post–processing of the obtained data. These additional modules, in turn, can use the functions implemented in the core module, in case they need them.

Finally, a *driver* or *main* program that defines the problem to be solved, manages the execution of the simulation and controls the calls to the multibody library functions must be defined. Several examples have been built, tested and added to the software as *demos*. A brief review of them is included in Section 2.5.

2.3 Core module

The core module of the MBS software has been designed following the same principles that inspire the structure of the whole architecture. It is a modular component, allowing the easy replacement of its parts, even with external, third–party programs and libraries. A scheme of the structure of the core module can be seen in Figure 2.4.

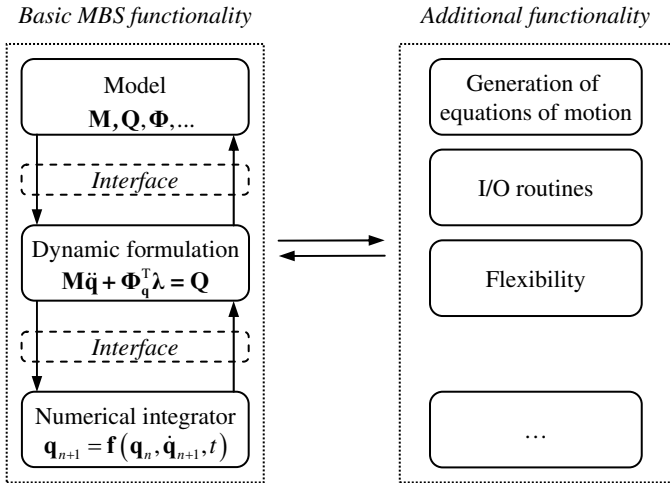

Figure 2.4: *Structure of the core module of the MBS software*

The basic MBS functionality of the module is provided by three core elements: a model of the multibody system to be simulated, a dynamic formulation for converting the equations of motion of the system (the DAE's in Equation 2.1) into ODE's, and a numerical integrator that obtains the value of the generalized coordinates in the next time–step from already known values of accelerations, velocities and positions. As a wide variety of numerical integrators and multibody formulations exists, not to mention the practically infinite collection of multibody problems that can be defined, template patterns for the definition of these elements have been created. In this way, C++ abstract base classes for the definition of models, formulations and integrators

have been written; the particular instances of each of them have been derived from these base classes through public inheritance, implementing the methods defined in their prototypes and thus satisfying their pre–defined interface.

The storage containers for matrices and vectors in this module have been obtained from the uBLAS library (Walter. et al., 2009), a C++ template class library, part of the Boost libraries, instead of using plain C++ vectors or arrays.

2.3.1 Implementation of models

The *model* component in the structure of the MBS software is the part of the code that is responsible for the evaluation of the dynamic terms of the problem, such as the mass matrix $\mathbf{M}$, the constraints vector $\mathbf{\Phi}$ or the generalized forces vector $\mathbf{Q}$. Obviously, the expression of these terms is closely related to the structure of the physical model to be simulated so, in practice, the expression of dynamic terms must be particularized for each problem. For this reason, only the prototype that each particular implementation of the model must comply with is defined in the core module. For natural coordinates, this is done in the abstract base class *GlobalModel*. Every global model instance must be derived from this class and implement every virtual method declared in its prototype. The model instance will also be in charge of storing the values of the generalized coordinates of the system $\mathbf{q}$ and their derivatives, $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$.

The methods defined in the prototype of base class *GlobalModel*, besides the *constructor* and *destructor* always required by C++, can be classified in three main groups:

- Methods that evaluate and return the dynamic terms of the model. Most global dynamic formulations require the same terms to be provided to them: mass matrix $\mathbf{M}$; generalized forces vector $\mathbf{Q}$; constraints vector $\mathbf{\Phi}$ with its derivatives with respect to time, in case they exist, $\mathbf{\Phi}_t$ and $\dot{\mathbf{\Phi}}_t$; the Jacobian matrix of the constraints vector with respect to the generalized coordinates $\mathbf{\Phi}_\mathbf{q}$ and its time derivative $\dot{\mathbf{\Phi}}_\mathbf{q}$; and, finally, stiffness and damping matrices $\mathbf{K}$ and $\mathbf{C}$, in case they are needed.

- Methods that provide access to the vectors that contain the generalized coordinates of the system and their time derivatives, privately stored by the *GlobalModel* class.

- Methods that return information about the nature of the model: its size, the number of constraint equations it has and other features, such as whether the mass matrix and the vector of forces are constant or whether the stiffness and damping matrices are required. This information is very important to accelerate the execution of the code, avoiding unnecessary evaluations of the dynamic terms and the allocation of excessive memory for the storage entities.

Once the base class *GlobalModel* has been defined, models of mechanical systems can be manually implemented by simply deriving a new model class by public inheritance and writing the code for each method defined in the prototype. Although

this can be sometimes convenient for efficiency reasons, it becomes a cumbersome and prone to errors task for medium–size and large systems. This is why an additional inner module has been created to automatically generate the dynamic terms of the system from the inertial properties of the bodies it is made up of, the forces that act on the bodies and the physical constraints that link them. This module is described in Section 2.3.4.

2.3.2 Implementation of dynamic formulations

Contrary to what happened with model instances, dynamic formulations can be reused without modifications in different simulations, so a library of formulations can be constructed inside the core module. This library can be later enlarged as new formalisms are coded.

The *dynamic formulation* component of the core module includes the matrix and vector containers for the storage of the dynamic terms of the system. These matrices and vectors are passed by reference to the model, which evaluates and then returns them, using the standardized methods defined in the prototype of abstract class *GlobalModel*. The use of this interface makes models and dynamic formulations easily replaceable.

The formulation is also in charge of enabling the numeric integration of the equations of motion given by Equation 2.1. The equations of motion are DAEs, which must be converted into ODEs for their numerical integration. The abstract base classes for dynamic formulations can be categorized into two groups, namely first or second order, depending on the order in which the formulation returns the ODEs representing the equations of motion. The expression of the equations of motion as second order ODEs leads to a system of the form

$$\mathbf{L}_{O2}\ddot{\mathbf{q}} = \mathbf{R}_{O2} \tag{2.3}$$

whereas the first order ODEs take the following form

$$\mathbf{L}_{O1}\dot{\mathbf{y}} = \mathbf{R}_{O1} \tag{2.4}$$

where $\dot{\mathbf{y}}$ stands for a vector containing the velocities and accelerations of the system, $\dot{\mathbf{y}} = \{\dot{\mathbf{q}}, \ddot{\mathbf{q}}\}^{\mathrm{T}}$, and $\mathbf{L}$ and $\mathbf{R}$ are the leading matrix and right–hand–side vector of each system, respectively. As some integrators are only able to manage first order ODEs, an additional intermediate class *Ode2ToOde1* that manages second order ODEs as if they were first order ones has to be defined.

A second division can be made depending on the way in which the integrator receives the equations the formulation generates. If the integrator requires the formulation to yield the time derivatives of the variables of the system, this is, $\ddot{\mathbf{q}}$ or $\dot{\mathbf{y}}$, then the formulation will be denoted as *explicit*. The *eval* method of these classes, declared in the prototype of the abstract ODE classes, must return the above–mentioned vectors of time derivatives. However, some integrators receive as arguments the leading matrix $\mathbf{L}$ and the residual vector $\mathbf{R}$ of the system instead; in this case, two methods

are required to communicate with the integrator, one for returning the leading matrix of the system (*evalTangentMatrix*), and another one for the right–hand–side vector (*evalResidual*), and the formulation will be labelled as *implicit*. It should be noted that the difference between *implicit* and *explicit* formulations has nothing to do with their inner algorithms, but it is related to the way in which they submit their results to the integrator.

2.3.3 Implementation of integrators

The upper level in the numeric core module corresponds to the *integrator*. The integrator receives the terms calculated by the dynamic formulation and evaluates from them the value of the generalized coordinates of the system, $\mathbf{q}$ and their first time derivatives $\dot{\mathbf{q}}$ in the next time–step. These values are then transferred to the dynamic formulation and the model, and the process can be restarted at the following integration step, or at the next iteration if convergence has not been attained yet, in the case of implicit integrators.

From the point of view of the definition of the base classes, there is no difference between explicit and implicit integrators. All of them receive the output of their correspondent formulation and update the values of the position and velocity vectors after the integration step has been taken. Every numerical integrator implemented in the library derives from an abstract base class, *DynSolver*. The prototype of the class describes the functions for controlling the execution of the integration, triggering the start of the calculations and setting the initial and final time of the motion. The advance of a time–step in the integration is done through a call to the *step* method. All these methods must be invoked by the driver program, so the integrator constitutes the truly external interface of the core module.

In some cases, the degree of interaction between the dynamic formulation and the integrator is so high that the equations of motion cannot be separated from the integrator. This is the case, for example, of the index–3 augmented Lagrangian formulation with projection of velocities and accelerations, described by Cuadrado et al. (2001) and used in Sections 3.2.2 and 4.2.1. In such cases, the use of C++ multiple inheritance enables the creation of classes that behave, at the same time, as integrators and dynamic formulations. The *dynamic formulation* and *integrator* components can be therefore merged into a single block, with no side effects on the standard interfaces nor on the behaviour of the core module.

2.3.4 Additional functionality

The described basic functionality of the core module is intended to be expanded through the addition of *inner modules*. Ideally, these modules must be added without modifying the main structure of the basic part of the core module. The purpose of the inner modules is to improve the performance of the basic core, or to add new functionality to it, so their removal would not prevent the software from running elemental simulations. The features comprised in them range from simple input–output routines,

which allow the pouring of simulation data into storage files in a standard format, to the addition of flexibility to the models, or the automatic generation of the equations of motion of the system. Some of the modules that have been implemented are briefly described in the following paragraphs.

I/O routines

The definition of routines for allowing reading from and writing to files enables the use of the simulation software in combination with pre– and post–processing external tools. To this end, matrices and vectors must be converted from the storage format data the software uses (in this case, uBLAS matrix and vector containers) to some standard storage format. Read and write routines for conventional ASCII storage and Matrix Market (NIST, 2007) formats have been written. This way, results can be shared with other simulation codes and applications, and simulation results can be stored after execution.

Interface to external linear solvers

The routines for the solution of linear equation systems in the form $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ play a key role in the efficiency of the code. The nature of this work requires to evaluate the performance of several linear solvers, so modularity is necessary again here for the easy replacement of the components. The uBLAS library incorporates its own linear solvers, compatible with uBLAS storage formats, but their efficiency has been found to be low. An interface for the use of efficient, third–party software linear solvers has been written, so that external solvers, coded in Fortran or C/C++, can be used. The particular implementation details of the different external solver libraries imply that a new interface must be implemented for each of them.

Automatic generation of the equations of motion

The manual coding of the methods that build the dynamic terms of the model can become an awkward task when the number of variables and constraint equations of the system increases. Non–trivial simulation models can easily reach hundreds of variables, with a similar number of constraints. Even when working with relatively small systems, the automatic generation of the equations of motion can save time and prevent mistakes from arising. At the same time, in some cases the user may want to manually write or edit the expression of the dynamic terms, so both ways of creating them must be available in the software. Moreover, different strategies to assemble the equations of motion can be compared, in order to select the most effective and to estimate the overhead this task adds to the execution of the code.

In order to allow the automatic generation of the equations, classes for the definition of the variables, bodies, constraints and forces that are part of the system have been defined. These classes are responsible for the assembly and evaluation of the dynamic terms of the system. A new class, *ConstraintsModel*, has been defined, which inherits from the abstract class *GlobalModel*, implements the code for the methods the

latter declares and contains storage elements for the components of the mechanism. Instead of writing the code for the evaluation of the matrices and vectors of the dynamic terms, the user can now define the list of components of the model, the joints that link them, and the forces that affect the motion.

The final level of this module is the automatic generation of the variables, bodies, constraints and forces from the data defined in a text file describing the geometry and properties of the simulated mechanism. The module is currently being developed in order to reach this goal.

2.4 Additional modules

The additional modules in Figure 2.3 have the purpose of adding extra non–multibody functionality to the core module of the software. The use of these modules is optional, so the user can decide not to build them if they are not necessary in a particular case. Furthermore, users can write and add their own additional modules if they need them. In order to maintain compatibility, the additional modules must use the multibody routines defined in the header file of the core module, *mbscore.h*, and the storage formats used in the main core; alternatively, they can use a different format using the proper translation routines, although this strategy would penalize the performance of the code.

The *mbsMatlab* library for communication with MATLAB/Simulink, described in Chapters 5 and 6 of this thesis, is an example of additional module. More modules for the communication with similar packages could be written in a similar way. Other extra additional modules can be written for providing a graphical representation of the simulation, acting as an intermediary between the core module and a graphics library such as OpenSceneGraph (OSG Community, 2009), or for creating a Graphic User Interface (GUI) through the use of QT (Nokia, 2009) or wxWidgets (The wxWidgets team, 2009) libraries.

2.5 Examples of multibody problems

The validation of the MBS simulation software is carried out through the solution of simple benchmark examples, such as those described by González et al. (2006). In particular, the L–loop four–bar linkage mechanism has been intensively used in this work, as it can be seen in Sections 3.2.1, 4.2.1 and 5.4. Other simple mechanical systems, such as slider–cranks and pendulums, have been programmed and solved, too. In order to simulate the motion of these mechanical systems, a reduced set of well–known formulations and integrators has been coded and added to the core module of the software. This set can be easily enlarged with new components, as long as their implementations fit the prototypes described by the abstract base classes for formulations and integrators.

2.5.1 Examples of dynamic formulations

Among the many multibody formalisms available today, three global dynamic formulations have been initially selected for solving the above–mentioned test problems. All of them are simple to implement and can be easily compared in terms of efficiency.

The first formalism is a penalty formulation, proposed by Bayo et al. (1988), which modifies the generic equations of motion of the system, given by Equation 2.1, by substituting the unknown value of Lagrange multipliers ($\boldsymbol{\lambda}$) with a value proportional to the violation of the constraints vector ($\boldsymbol{\Phi}$):

$$\boldsymbol{\lambda} = \alpha \left(\ddot{\boldsymbol{\Phi}} + 2\xi\omega\dot{\boldsymbol{\Phi}} + \omega^2\boldsymbol{\Phi} \right) \tag{2.5}$$

where ξ and ω are Baumgarte's stabilization parameters and α is the penalty factor, a scalar whose value is usually taken as 10^7 times the largest term of the mass matrix. Replacing this expression in Equation 2.1, together with the adequate forms of the derivatives of the constraints vector, yields

$$\left(\mathbf{M} + \alpha\boldsymbol{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\Phi}_{\mathbf{q}}\right)\ddot{\mathbf{q}} = \mathbf{Q} - \alpha\boldsymbol{\Phi}_{\mathbf{q}}^{\mathrm{T}}\left(\dot{\boldsymbol{\Phi}}_{\mathbf{q}}\dot{\mathbf{q}} + 2\xi\omega\dot{\boldsymbol{\Phi}} + \omega^2\boldsymbol{\Phi} + \dot{\boldsymbol{\Phi}}_t\right) \tag{2.6}$$

and the value of the accelerations of the system ($\ddot{\mathbf{q}}$) can be obtained from this expression, provided the dynamic terms of the system are known.

The second implemented formulation is an augmented Lagrangian one, in the form described by García de Jalón and Bayo (1994):

$$\left(\mathbf{M} + \alpha\boldsymbol{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\Phi}_{\mathbf{q}}\right)\ddot{\mathbf{q}} = \mathbf{Q} - \boldsymbol{\Phi}_{\mathbf{q}}^{\mathrm{T}}\left[\boldsymbol{\lambda}_i + \alpha\left(\dot{\boldsymbol{\Phi}}_{\mathbf{q}}\dot{\mathbf{q}} + 2\xi\omega\dot{\boldsymbol{\Phi}} + \omega^2\boldsymbol{\Phi} + \dot{\boldsymbol{\Phi}}_t\right)\right] \tag{2.7}$$

that evaluates the Lagrange multipliers of the system via the following iterative process:

$$\boldsymbol{\lambda}_{i+1} = \boldsymbol{\lambda}_i + \alpha\left(\ddot{\boldsymbol{\Phi}} + 2\xi\omega\dot{\boldsymbol{\Phi}} + \omega^2\boldsymbol{\Phi}\right) \tag{2.8}$$

where i represents the iteration number. The value of the Lagrange multipliers is substituted in Equation 2.7 after each iteration, obtaining a new value of the accelerations and leading thus to an improved value of the multipliers. In practice, no more than three iterations are enough to achieve a good convergence.

Finally, preliminary tests showed the augmented Lagrangian formulation of index–3 with projections to be the most efficient one, and therefore it has been used in the subsequent work of this thesis. This formalism incorporates a numerical integrator, the well–known trapezoidal rule, in the implementation of its own algorithm; moreover, the different stages it is made up of are closely related to the way in which the optimization techniques introduced in Chapters 3 and 4 are implemented. For this reason, its structure will be described in more detail in these Chapters, under Sections 3.2.2 and 4.2.1.

2.5.2 Examples of integrators

Two integrators, one explicit and another one implicit, have been used in the initial tests of the MBS software. The first one is the explicit Runge–Kutta formula of second order. In the present work, this integrator has been implemented to manage first order ODEs, according to the expression:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2}\left(\dot{\mathbf{y}}_1 + \dot{\mathbf{y}}_2\right) \tag{2.9}$$

where

$$\begin{aligned} \dot{\mathbf{y}}_1 &= \mathbf{f}\left(\mathbf{y}_n, t\right) \\ \dot{\mathbf{y}}_2 &= \mathbf{f}\left(\mathbf{y}_n + \dot{\mathbf{y}}_1 \Delta t,\ t + \Delta t\right) \end{aligned} \tag{2.10}$$

with $\mathbf{f}\left(\mathbf{y}, t\right)$ representing the evaluation of the time derivatives of the positions and velocities of the system for a certain value of these variables and the time. The integration time–step from instant n to $n+1$ is noted as Δt. This integrator is not unconditionally stable, so its use must be constrained to systems without high stiffness. A detailed description of the algorithm can be found in books of numerical analysis (Shampine, 1994).

The other method of numerical integration is the well–known Newmark family of formulae (Newmark, 1959), an easy to implement, implicit, single–step integrator with good stability properties. Its second order form is the following one:

$$\begin{aligned} \mathbf{q}_{n+1} &= \mathbf{q}_n + \Delta t \dot{\mathbf{q}}_n + \frac{\Delta t^2}{2}\left[\left(1-2\beta\right)\ddot{\mathbf{q}}_n + 2\beta\ddot{\mathbf{q}}_{n+1}\right] \\ \dot{\mathbf{q}}_{n+1} &= \dot{\mathbf{q}}_n + \Delta t\left[\left(1-\gamma\right)\ddot{\mathbf{q}}_n + \gamma\ddot{\mathbf{q}}_{n+1}\right] \end{aligned} \tag{2.11}$$

where β and γ are scalar parameters. The trapezoidal rule is a particular case of this method in which $\beta = 1/4$ and $\gamma = 1/2$; this is equivalent to assuming that the value of the accelerations is constant during the time interval $[t_n, t_{n+1}]$ and equal to $\left(\ddot{\mathbf{q}}_n + \ddot{\mathbf{q}}_{n+1}\right)/2$. This implicit algorithm can be used in a predictor–corrector fashion, with fixed point iteration, although it is commonly introduced in the equations of motion of the system, as it is the case in the above–mentioned augmented Lagrangian formulation of index–3, and solved through the Newton–Raphson iteration.

2.6 Conclusions

The design of a software architecture for the simulation of multibody systems is a complex task, where many alternatives, sometimes mutually exclusive, must be considered and compared before making a choice. The difficulty of the job increases when it has to be carried out from scratch, due to the practical impossibility of predicting every requirement of the software during its useful life beforehand. From this point of view, flexibility and modularity arise as evident design goals; however, these targets must be achieved, as far as possible, without hindering the efficient execution of

the multibody algorithms, which would render the whole software useless for many practical applications.

The software architecture developed during the carrying out of this thesis has been designed to simultaneously meet these two apparently contradictory aims. Efficient computational routines for MBS simulation have been implemented in a core module, enabling the fast replacement of components without affecting the main structure of the code. Several examples of models, dynamic formulations and integrators have been implemented and tested to verify the modularity and efficiency of the architecture. The layout of the whole software has been built on the basis of this core module, with additional modules linked to the main one through standardized interfaces. Thus, a virtually infinite number of additional features can be added to the basic multibody functionality, enabling the software to meet new additional conditions.

Several conclusions about the design and building of a MBS software have been extracted and constitute valuable guidelines for the developers of multibody codes:

- The object–oriented approach is the most adequate for the programming of software for the simulation of multibody systems. Among the many different languages under this paradigm, C++ shows the best trade–off between the flexibility provided by its object–oriented features, such as inheritance and templates, and efficiency.

- The most computationally expensive parts of the code, such as dynamic solvers and matrix calculations, can be implemented in efficient procedural languages and then linked to the C++ main architecture with little effort. Optimized versions of matrix routines and linear solvers exist and are freely available in Internet, and can be used to deal with these segments of code in an effective and convenient way.

- There is a wide variety of freely available tools for the development of C++ source code. The use of these auxiliary tools is highly recommended, specially of build–process managers and version control systems. These tools greatly contribute to the flexibility of the software and reduce the required workload to synchronize the development of the code among several programmers. They also allow the same source code to be built in different computer environments.

These guidelines have been applied during the elaboration of the MBS software in this work, the one which has been used as a basis to carry out the research described in the following Chapters.

Chapter 3

Linear Algebra Implementation

This Chapter compares the efficiency of multibody system (MBS) dynamic simulation codes that rely on different implementations of linear algebra operations. The dynamics of an L–loop four–bar mechanism has been solved with an index–3 augmented Lagrangian formulation combined with the trapezoidal rule as numerical integrator. Different implementations for this method, both dense and sparse, have been developed, using a number of linear algebra software libraries (including sparse linear equation solvers) and optimized sparse matrix computation strategies. Numerical experiments have been performed in order to measure their performance, as a function of problem size and matrix filling. Results show that optimal implementations can increase the simulation efficiency in a factor of 2–3, compared with the starting classical implementations, and in some topics they disagree with widespread beliefs in MBS dynamics. Finally, advices are provided to select the implementation which delivers the best performance for a certain MBS dynamic simulation.

3.1 Introduction

Dynamic simulation of multibody systems (MBS) is of great interest for the dynamics of machinery, road and rail vehicle design, robotics and biomechanics. Computer simulations performed by MBS simulation tools lead to more reliable, optimized designs and significant reductions in cost and time of the product development cycle. The computational efficiency of these tools is a key issue for two reasons. First, there are some applications, like hardware–in–the–loop settings or human–in–the–loop devices, which cannot be developed unless MBS simulation is performed in real time. And second, when MBS simulation is used in virtual prototyping, faster simulations allow the design engineer to perform what–if analyses and optimizations in shorter times, increasing productivity and interaction with the model. Therefore, computational efficiency is an active area of research in MBS, and it holds a relevant position in MBS–related scientific conferences and journals.

A great variety of methods to improve simulation speed have been proposed during the last years, e.g. Cuadrado et al. (1997), Bae et al. (2000) and Anderson and Critch-

ley (2003), among others. Most of these methods base their efficiency improvements on the development of new dynamic formulations. However, although implementation aspects can also play a key factor in the performance of numerical simulations, their effect on multibody system dynamics has not been studied in detail. Some recent contributions have investigated the possibilities of parallel implementations (Anderson et al., 2007), but comprehensive comparisons about serial implementations in MBS dynamics have not been published yet.

Multibody dynamics codes make an intensive use of linear algebra operations. This is especially true in $O\left(n^3\right)$ formulations, where n is the number of bodies in the multibody system, as it is the case of many global formulations, which use a relatively large number of coordinates and constraint equations to define the problem, leading to the need of solving a large system of linear equations. These formulations spend around 80% of the CPU–time in matrix computations. Other formalisms have been developed that lead to $O\left(n\right)$ algorithms, reducing the size of the system of equations to be solved, but at the cost of considerably increasing the number of required matrix computations. Moreover, if flexible bodies are considered, the percentage of simulation time inverted in matrix operations can become even higher.

As a result, the implementation of linear algebra operations is critical to the efficiency of MBS dynamic simulations. These operations can be grouped into two categories: (a) operations between scalars, vectors and matrices, and (b) solution of linear systems of equations; two additional orthogonal categories can be established based on the data storage format: dense storage or sparse storage. Many efficient implementations for these routines have been made freely available in the last decade. Their performance has been compared in previous works, both in an application–independent context such as Gupta (2002), Gould et al. (2007) and Whaley et al. (2001) and under the perspective of a particular application like Finite Element Analysis (Turek et al., 2001) or Computational Chemistry (Yu and Yu, 2002). But, as it will be explained in this Chapter, these studies do not fit the particular features of MBS dynamics, and therefore their conclusions cannot be extrapolated to this field.

The goal of this Chapter is to compare the efficiency of different implementations of linear algebra operations, and study their effect in the context of MBS dynamic simulation. Results will provide guidelines about which numerical libraries and implementation techniques are more convenient in each case. This information will be very helpful to researchers developing high–performance or real–time multibody simulation codes.

The remainder of the Chapter is organized as follows: Section 3.2 describes the test problem and the dynamic formulation used in the numerical experiments to compare the efficiency of different implementations; Sections 3.3 and 3.4 present efficient implementations for dense and sparse linear algebra, respectively; Section 3.5 compares the results obtained in Sections 3.3 and 3.4 and extrapolates them to other dynamic formulations; finally, Section 3.6 provides conclusions, advices for efficient implementations and areas of future work.

3.2 Benchmark setup

In order to study the effect of linear algebra implementations in MBS dynamic simulations, a test problem has been solved with a particular dynamic formulation using different software implementations. A starting implementation will also be described, since its efficiency will serve as a reference to measure performance improvements.

3.2.1 Test problem

The selected test problem is a 2D, one degree–of–freedom assembly of four–bar linkages with L loops, composed by thin rods of 1 m length with a uniformly distributed mass of 1 kg, moving under gravity effects. Initially, the system is in the position shown in Figure 3.1, and the velocity of the x–coordinate of point B_0 is +1 m/s. The simulation time is 20 s. This mechanism has been previously used as a benchmark problem for multibody system dynamics (Anderson and Critchley, 2003; González et al., 2006).

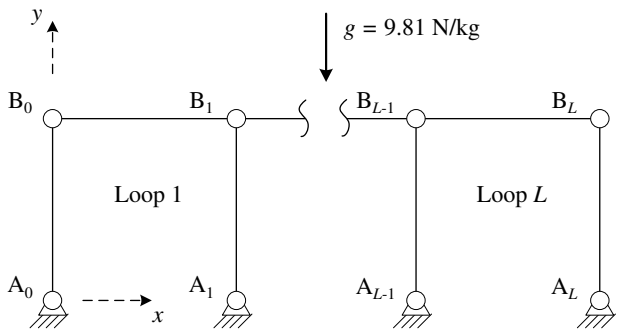

Figure 3.1: *L–loop four–bar linkage*

3.2.2 Dynamic formulation

The L–loop four–bar mechanism has been modelled using planar natural coordinates, global and dependent (García de Jalón and Bayo, 1994), leading to $2L + 2$ variables (the x and y coordinates of the B points), and $2L + 1$ constraints, associated with the constant length condition of the rods. The equations of motion of the whole multibody system are given by the well–known index–3 augmented Lagrangian formulation in the form:

$$\begin{aligned} &\mathbf{M}\ddot{\mathbf{q}} + \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\alpha\mathbf{\Phi} + \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\lambda}^{*} = \mathbf{Q} \\ &\boldsymbol{\lambda}_{i+1}^{*} = \boldsymbol{\lambda}_{i}^{*} + \alpha\mathbf{\Phi}_{i+1}; \quad i = 0, 1, 2, \ldots \end{aligned} \tag{3.1}$$

where $\mathbf{M}$ is the mass matrix (constant for the proposed test problem), $\ddot{\mathbf{q}}$ are the accelerations, $\mathbf{\Phi}_{\mathbf{q}}$ the Jacobian matrix of the constraint equations, α the penalty factor, $\mathbf{\Phi}$

the constraints vector, $\boldsymbol{\lambda}^*$ the Lagrange multipliers vector and $\mathbf{Q}$ the vector of applied and velocity dependent inertia forces. The Lagrange multipliers for each time–step are obtained from an iterative process, where the value of $\boldsymbol{\lambda}_0^*$ is equal to the $\boldsymbol{\lambda}^*$ obtained in the previous time–step.

As integration scheme, the implicit single–step trapezoidal rule has been adopted. The corresponding difference equations in velocities and accelerations are:

$$\begin{aligned}
\dot{\mathbf{q}}_{n+1} &= \frac{2}{\Delta t}\mathbf{q}_{n+1} + \hat{\dot{\mathbf{q}}}_n; \quad \hat{\dot{\mathbf{q}}}_n = -\left(\frac{2}{\Delta t}\mathbf{q}_n + \dot{\mathbf{q}}_n\right) \\
\ddot{\mathbf{q}}_{n+1} &= \frac{4}{\Delta t^2}\mathbf{q}_{n+1} + \hat{\ddot{\mathbf{q}}}_n; \quad \hat{\ddot{\mathbf{q}}}_n = -\left(\frac{4}{\Delta t^2}\mathbf{q}_n + \frac{4}{\Delta t}\dot{\mathbf{q}}_n + \ddot{\mathbf{q}}_n\right)
\end{aligned} \tag{3.2}$$

Dynamic equilibrium can be established at time–step $n+1$ by introducing the difference equations (3.2) into the equations of motion (3.1), leading to a nonlinear algebraic system of equations with the dependent positions as unknowns:

$$\begin{aligned}
&\mathbf{f}(\mathbf{q}) = \mathbf{0} = \\
&= \mathbf{M}\mathbf{q}_{n+1} + \frac{\Delta t^2}{4}\boldsymbol{\Phi}_{\mathbf{q}_{n+1}}^{\mathrm{T}}\left(\alpha\boldsymbol{\Phi}_{n+1} + \boldsymbol{\lambda}_{n+1}\right) - \frac{\Delta t^2}{4}\mathbf{Q}_{n+1} + \frac{\Delta t^2}{4}\mathbf{M}\hat{\ddot{\mathbf{q}}}_n
\end{aligned} \tag{3.3}$$

Such system, whose size is the number of variables in the model, is solved through the Newton–Raphson iteration

$$\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right]_i \Delta\mathbf{q}_{i+1} = -\left[\mathbf{f}(\mathbf{q})\right]_i \tag{3.4}$$

using the approximate tangent matrix (symmetric and positive definite)

$$\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right] \cong \mathbf{M} + \frac{\Delta t}{2}\mathbf{C} + \frac{\Delta t^2}{4}\left(\boldsymbol{\Phi}_{\mathbf{q}}^{\mathrm{T}}\alpha\boldsymbol{\Phi}_{\mathbf{q}} + \mathbf{K}\right) \tag{3.5}$$

where $\mathbf{C}$ and $\mathbf{K}$ represent the contribution of the damping and elastic forces of the system (which are zero for the test problem). Once convergence is attained into the time–step, the obtained positions $\mathbf{q}_{n+1}$ satisfy the equations of motion (3.1) and the constraint conditions $\boldsymbol{\Phi} = \mathbf{0}$, but the corresponding sets of velocities $\dot{\mathbf{q}}^*$ and accelerations $\ddot{\mathbf{q}}^*$ may not satisfy $\dot{\boldsymbol{\Phi}} = \mathbf{0}$ and $\ddot{\boldsymbol{\Phi}} = \mathbf{0}$. To achieve this, cleaned velocities $\dot{\mathbf{q}}$ and accelerations $\ddot{\mathbf{q}}$ are obtained by means of mass–damping–stiffness orthogonal projections, reusing the factorization of the tangent matrix:

$$\begin{aligned}
\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right]\dot{\mathbf{q}} &= \left[\mathbf{M} + \frac{\Delta t}{2}\mathbf{C} + \frac{\Delta t^2}{4}\mathbf{K}\right]\dot{\mathbf{q}}^* - \frac{\Delta t^2}{4}\boldsymbol{\Phi}_{\mathbf{q}}^{\mathrm{T}}\alpha\boldsymbol{\Phi}_t \\
\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right]\ddot{\mathbf{q}} &= \left[\mathbf{M} + \frac{\Delta t}{2}\mathbf{C} + \frac{\Delta t^2}{4}\mathbf{K}\right]\ddot{\mathbf{q}}^* - \frac{\Delta t^2}{4}\boldsymbol{\Phi}_{\mathbf{q}}^{\mathrm{T}}\alpha\left(\dot{\boldsymbol{\Phi}}_{\mathbf{q}}\dot{\mathbf{q}} + \dot{\boldsymbol{\Phi}}_t\right)
\end{aligned} \tag{3.6}$$

This method, described in detail by Cuadrado et al. (2000), has proved to be a robust and efficient global formulation (Cuadrado et al., 2001, 2004a). All the subsequent numerical experiments have been performed using as time–step a value of

$\Delta t = 1.25 \cdot 10^{-3}$ s and a penalty factor $\alpha = 10^8$.

3.2.3 Starting implementation

In the starting implementation, the simulation algorithm was implemented using Fortran 90 and the Compaq Visual Fortran compiler. Two versions were developed:

- A dense matrix storage version, using Fortran 90 matrix manipulation capabilities and the linear equation solver released with this compiler (IMSL Fortran Library, from Visual Numerics).
- A sparse matrix storage version, using the MA27 sparse linear equation solver from the Harwell Subroutine Library.

These two implementations, typical in the multibody community, have been tuned and improved by our group during the last years, and they have proved to be faster than commercial codes (Cuadrado et al., 2001, 2004a). Their efficiency has served as a reference to measure the performance improvements achieved with the new implementations proposed in this work.

Table 3.1: *Percentage of the total CPU–time required by each algorithm phase in the starting implementation for typical problem sizes: dense version in small problems (10 loops, 22 variables) and sparse version in medium–size problems (40 loops, 82 variables)*

Stage	Dense	Sparse
Evaluation of residual and tangent matrix, Eqs. (3.1), (3.5)	48%	15%
Evaluation of right–hand–side in projections, Eq. (3.6)	4%	13%
Factorizations and back–substitutions, Eqs. (3.4), (3.6)	44%	51%
Other	4%	21%

Table 3.1 shows the results of a CPU usage profiling in our starting implementation, for both dense and sparse versions, applied to representative problem sizes. As stated in the introduction to this Chapter, matrix computations consume most of the CPU–time.

In order to test alternative implementations, the MBS simulation software described in Chapter 2 has been used. Numerical experiments have been performed on an AMD Athlon64 CPU. After testing different operating systems and compilers, results show that their effect on the performance is an order of magnitude lower than the effect of linear algebra implementations. Final CPU–times have been measured using the GNU gcc compiler and the Linux O.S., without loss of generality.

3.3 Efficient dense matrix implementations

Global formulations applied to reduced rigid models (e.g. an industrial robot), or recursive and semi–recursive formulations applied to medium–size rigid models (e.g. a

complete road vehicle), lead to algorithms that operate with small–size matrices of dimensions smaller than 50×50. In these cases, dense linear algebra is frequently used in MBS dynamics, since it is *supposed* to provide equal or higher performance than sparse implementations. Achieving real time in the simulation of these small problems can be a challenge in hardware–in–the–loop settings (e.g. advanced Electronic Stability Control systems for automobiles), due to the low computing power of embedded microprocessors, the small time–steps required for hardware synchronization and the added control logic.

A straightforward way to increase the performance of dense matrix computations is using an efficient implementation of BLAS (Basic Linear Algebra Subprograms). BLAS (NIST, 2009) is a standardized interface that defines routines to perform low level operations between scalars, dense vectors and dense matrices. A Fortran 77 reference implementation is available, and more efficient implementations have been developed by hardware vendors and researchers.

These optimized BLAS versions exploit hardware features of modern computer architectures to get the best computational efficiency. In addition to the reference Fortran 77 implementation, three optimized BLAS implementations have been tested:

- ATLAS (Automatically Tuned Linear Algebra Software), which employs empirical techniques to generate an optimal implementation for any hardware architecture (Whaley et al., 2001);
- GotoBLAS, based on optimized assembler kernels, hand–written for the most popular hardware architectures (Goto, 2009); and
- ACML, developed by the microprocessor manufacturer AMD for its CPU's (AMD, 2009). Other hardware vendors also provide their own implementations (such as MKL from Intel and SCSL from SGI).

Dynamic simulations can also make a profit of these optimized BLAS implementations in the solution of dense linear equation systems, provided the LAPACK library is used (NETLIB, 2009), since its linear equation solvers are based on low–level BLAS operations. In addition to the reference LAPACK implementation, written in Fortran 77, some optimized BLAS implementations like ATLAS and ACML supply their own optimized versions of the LAPACK linear solvers.

The proposed test problem, with a number of loops L ranging from 1 to 20 (i.e. number of variables N ranging from 4 to 42), was solved using different BLAS and LAPACK implementations to perform all matrix computations. Since the tangent matrix in the proposed dynamic formulation is symmetric and positive definite (SPD), only the lower triangular part of the matrix is computed; the LAPACK routines DPOTRF and DPOTRS have been used as linear equation solver. Performance results are shown in Figure 3.2, where the legend text is encoded in the form "BLAS implementation + LAPACK implementation" (except for the starting implementation), and the combinations are ordered by increasing efficiency.

Results in Figure 3.2 clearly show the advantage of using BLAS and LAPACK, which speed up the simulation in a factor between 2 and 5, depending on the problem

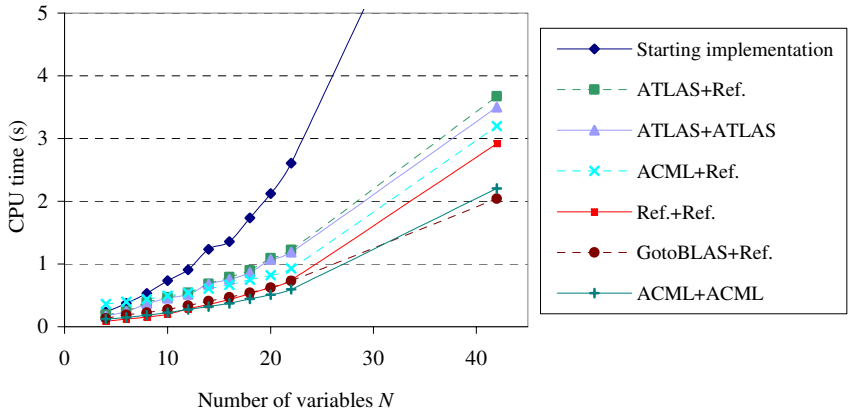

Figure 3.2: *Performance of different dense BLAS and LAPACK implementations*

size, compared with our previous starting implementation. The low performance of the ATLAS implementation, compared to the BLAS reference implementation, can be explained by its high sensitiveness to the development environment (e.g. compiler version) and its current unstable state (it is under strong development). The vendor implementation (ACML) and GotoBLAS deliver the best results except for very small problems (up to 10 variables). The implementation named "Ref.+Ref." delivers the best performance for very small problems, and 70–80% of the performance of the best implementations for medium–size problems (3 times more efficient than our starting implementation); in addition, it has a very good portability (it is written in plain Fortran 77) and usability: the installation process is straightforward, which is not always true for other implementations.

Since some MBS dynamic formulations lead to a non–symmetric tangent matrix (Dopico et al., 2006), the same numerical experiment has been executed using general algorithms (not SPD–specific) to compute all matrix operations; CPU–times are about 15% higher, but the efficiency ranking of Figure 3.2 is maintained.

3.4 Efficient sparse matrix implementations

In MBS dynamics, sparse matrix techniques are used in global formulations applied to medium– or big–size rigid models; as an example, a model in natural coordinates of an automobile leads to matrices of dimension about 200×200 (Cuadrado et al., 2004a). If flexible bodies are considered, the matrix size increases, making sparse techniques profitable even if recursive or semi–recursive formulations are used: a model in relative coordinates of the same automobile, with some of its bodies characterized as flexible elements (described by component mode synthesis), leads to matrices of dimension about 100×100. In any case, MBS models hardly ever lead to matrix sizes bigger than 1000×1000, significantly smaller than the typical sizes in other applica-

tions, like Finite Element Analysis (FEA) or Computational Fluid Dynamics (CFD).

Regarding the sparsity, the proposed test problem and MBS dynamic formulation lead to a tangent matrix of size $2L+2$ and $12L+4$ structural non–zeros. For matrices of size 50×50, 100×100 and 500×500, the corresponding number of non–zeros is 12%, 6% and 1%. These are representative values for MBS simulations, and they are considerably higher than typical values in other applications that require sparse matrix technology (FEA, CFD).

Hence, MBS dynamics has two characteristics which make its sparse matrix computations different from other applications:

- Matrix computations are very repetitive, and the sparse patterns usually remain constant during the simulation. Therefore, symbolical pre–processing can be applied to almost all matrix expressions at the beginning of the simulation, in order to accelerate the numerical evaluations during the simulation.

- The involved sparse matrices are relatively small and dense, compared with the typical values in sparse matrix technology.

3.4.1 Optimized sparse matrix computations

Several numerical libraries are available nowadays to support sparse matrix computations: MTL, MV++, Blitz++, SparseKIT, etc. For our new implementations, we have chosen uBLAS, a C++ template class library that provides BLAS functionality for sparse matrices (Walter. et al., 2009). Its design and implementation unify mathematical notation via operator overloading and efficient code generation via expression templates. Even though, the performance of some matrix operations can be further improved if some special algorithms are used. Results of CPU usage profiling (similar to Table 3.1) have led to the optimization of the following three operations.

The first optimized operation is the rank–k update of symmetric matrix, $\mathbf{\Phi_q}^{\mathrm{T}} \alpha \mathbf{\Phi_q}$, computed in Equation (3.5). Since the sparse structure of the Jacobian matrix $\mathbf{\Phi_q}$ is constant, a symbolic analysis is performed in order to pre–calculate the sparse pattern of the resultant matrix and to create a data structure that holds the operations needed to evaluate it during the simulation. In our starting sparse implementation, a similar approach was taken, but the Jacobian matrix was stored as dense, to simplify the operations at the cost of a higher memory usage.

The second optimized operation is the matrix addition computed in Equation (3.5). Our starting sparse implementation used the Harwell MA27 routine as linear equation solver, which requires the sparse matrix to be stored in coordinate format (Figure 3.3), and allows duplicated entries in the matrix structure. Therefore, the matrix addition is not actually computed, since the different terms are appended as duplicated entries in the tangent matrix. Our new implementation uses the compressed column storage format (Figure 3.3), since it is required by the sparse linear equation solvers tested in the following Section. This format, also known as the Harwell–Boeing sparse matrix format, is quite common in direct sparse linear equation solvers. Every value stored in the value data array *val* of the matrix is placed in its proper location in the pattern with

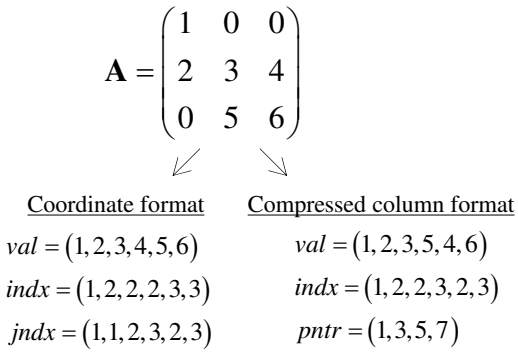

Figure 3.3: *Storage formats used in sparse implementations*

the use of an *indx* array, which assigns to each value the index of the row to which it belongs, and a *pntr* array, which stores the indices of the elements in the *val* array where a new column starts. With this storage, matrix additions require complex data traversing that slows down the performance.

The following approach was taken in order to optimize the operation:

$$\mathbf{B} = t_1\mathbf{A}_1 + t_2\mathbf{A}_2 \tag{3.7}$$

In the pre–processing stage, the sparse pattern of $\mathbf{B}$ is calculated as the union of $\mathbf{A}_1$ and $\mathbf{A}_2$ sparse patterns, and the resulting pattern is added to $\mathbf{A}_1$ and $\mathbf{A}_2$. In this way, $\mathbf{A}_1$, $\mathbf{A}_2$ and $\mathbf{B}$ share the same sparse pattern (same *indx* and *pntr* arrays in the compressed column storage format shown in Figure 3.3), and therefore, the matrix addition can be computed as a vector addition of the *val* arrays:

$$\mathbf{val_B} = t_1\mathbf{val}_{\mathbf{A}_1} + t_2\mathbf{val}_{\mathbf{A}_2} \tag{3.8}$$

This technique increases the number of non–zeros (*NNZ*) of the addend matrices. In the proposed MBS dynamic formulation, the *NNZ* of the mass matrix $\mathbf{M}$ is increased in a 10% approximately, which slows down the matrix–vector multiplications needed in the right terms of Equations (3.4) and (3.6). However, the simulation timings show that this slowdown is negligible compared with the gains derived from the fast matrix addition.

Finally, the third optimized operation concerns sparse matrix access. The write operation $\mathbf{A}(i, j) = a_{ij}$, straightforward in dense storage, needs additional position lookup when the compressed column storage is used. In the proposed formulation, the update of the Jacobian matrix $\mathbf{\Phi_q}$ in each iteration takes 10–15% of the CPU–time. The involved operations are rather simple, and most of this time is spent in matrix access. In order to optimize this procedure, a pre–processing stage evaluates the Jacobian matrix and registers the order in which entries $\mathbf{\Phi_q}(i, j)$ are written in the *val* array of the compressed column format, creating a vector that holds indices to

these positions, in the same order of evaluation. Later, in the simulation stage, access to the Jacobian matrix is performed using this index vector, without the need to map (i, j) indices to memory addresses for each writing operation.

Table 3.2: *Efficiency of the optimized sparse matrix operations*

	CPU–time (ms)		
Sparse operation	Not Optimized	Optimized	Ratio
1) Rank–k update of symmetric matrix	2525.2	9.4	269
2) Matrix addition	140.9	1.9	74
3) Jacobian matrix evaluation	11.6	3.8	3

Table 3.2 summarizes the performance gains delivered by the proposed optimizations, compared with the performance delivered by the uBLAS default algorithms (which are similar to other generic sparse matrix libraries). The numerical experiment used the matrix terms derived from an L–loop four–bar mechanism with $L = 40$ loops, which leads to a tangent matrix of size 82×82. Results show the importance of optimizing rank–k updates and matrix additions, since the performance delivered by off–the–shelf sparse matrix libraries is not satisfactory for these repetitive operations.

3.4.2 Evaluation of sparse linear equation solvers

Data in Table 3.1 shows that, in our starting sparse implementation, about 50% of the total CPU–time is spent in tangent matrix factorizations and back–substitutions (Equations (3.4) and (3.6)). Thus, the main performance improvements in MBS dynamic simulation can be achieved by using a more efficient sparse linear solver. During the last decade, sparse solvers have significantly improved the state of the art of the solution of general sparse linear equation systems, and more than 30 sparse solver libraries are freely available in the World Wide Web (Dongarra, 2009).

The efficiency of sparse solvers varies greatly depending on parameters like the matrix size, structure and number of non–zeros. In addition, solving a sparse linear equation system usually involves three stages: pre–processing (ordering, symbolic factorization), numerical factorization and back substitution; some solvers are very fast in the first stage, while others perform better in the second or third stage. The performance of sparse solvers has been compared in previous works, e.g. Gupta (2002) and Scott and Hu (2007), but the conditions of these studies (in particular, matrix sizes and percentage of non–zeros) do not fit the above–mentioned particular features of MBS dynamics, and therefore their conclusions cannot be extrapolated to this field. As a result, it is almost impossible to determine, without numerical experiments, which sparse solver will deliver the best performance in an MBS dynamic simulation.

Given the large number of existing sparse solvers, a selection process is required in order to narrow the scope. Solvers for shared memory or distributed memory parallel machines have been discarded, since the small matrix sizes in MBS real–time

dynamics (almost fit in the CPU cache memory) makes them unprofitable. The same argument applies to iterative solvers and out–of–core solvers, designed for very big linear equation systems. From the remaining solvers, those that performed best in previous comparative studies have been selected:

- CHOLMOD, a symmetric positive definite solver (Chen et al., 2008);
- KLU, a solver specifically designed for circuit simulation matrices (Davis and Natarajan, 2010);
- SuperLU (serial version), an unsymmetric general purpose solver (Demmel et al., 1999a);
- Umfpack, an unsymmetric multifrontal solver (Davis, 2004); and
- WSMP, a symmetric indefinite solver (Gupta et al., 1998).

Despite the coefficient matrix is symmetric positive definite in the proposed dynamic formulation, we have included in the numerical experiments some general, non–symmetric solvers (KLU, SuperLU, Umfpack), since other dynamic formulations lead to a non–symmetric coefficient matrix (Dopico et al., 2006). In these cases, the whole coefficient matrix (upper and lower parts) is computed, while with symmetric solvers only half matrix is used in the formulation equations. Each solver supports its own set of reordering strategies; all of them have been tested to select the best one in each simulation. In addition, all the optimizations described in the previous Section were applied to our new sparse implementation.

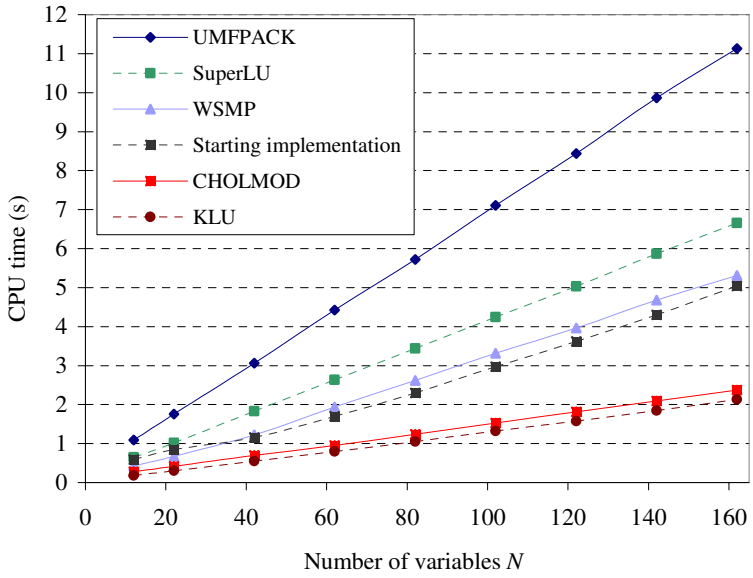

Figure 3.4: *Performance of different sparse linear equation solvers as a function of the problem size*

The proposed test problem, with a number of loops L ranging from 10 to 500 (i.e. number of variables N ranging from 22 to 1002), was solved using different sparse solvers. Performance results are shown in Figure 3.4 for a number of variables up to 160, since the trends are preserved for higher number of variables. The legend text shows the name of the sparse solvers, ordered by increasing efficiency.

Surprisingly, KLU is the fastest solver, despite being a general solver that does not exploit the symmetric positive definite condition of the coefficient matrix; in addition, it has been designed for circuit simulation problems, which lead to very sparse matrices, the opposite case of MBS dynamics. However, these results have been obtained by using the KLU *refactor* routine for numerical factorizations, which reuses the pivoting strategy generated in the pre–processing stage. In multibody problems where the elements of the tangent matrix of Equation (3.5) may significantly change their relative values during the simulation (e.g. due to violent impacts), the initial pivoting strategy may become invalid and the *refactor* routine would probably accumulate high numerical errors. To avoid this, the KLU solver can recalculate the pivoting strategy in each numerical factorization, but this method increases the CPU–times in a 50%. On the other hand, CHOLMOD, a symmetric positive definite solver, performs at 85% of KLU, despite recalculating the pivoting strategy in each numerical factorization. Our best new sparse implementations (using KLU or CHOLMOD) perform faster than our starting implementation, in a factor from 2 (small problems) to 3 (large problems of 1000 variables).

3.4.3 Effect of dense BLAS implementation

Some sparse solvers rely on the dense BLAS routines, described in Section 3.3, to improve the computation of some basic linear algebra operations they internally treat as dense, increasing thus their performance. In addition, some sparse matrix operations (e.g. the optimized matrix addition described in Section 3.4.1) are actually computed as dense vector operations using BLAS routines. Results shown in Figure 3.4 have been generated using the reference BLAS implementation. The same numerical experiment has been executed using the faster, optimized GotoBLAS and ACML implementations, and CPU–times have decreased only in a 2% – 3%. Hence, the reference BLAS implementation is recommended for MBS dynamics in sparse implementations, since it provides the best compromise between performance and usability.

3.5 Sparse vs. dense implementations

As previously stated, dense linear algebra is frequently used in MBS dynamics for small problems (dimension of the coefficient matrix lower than 50), since it is supposed to provide higher performance than sparse implementations. Our starting sparse implementation, which already employs some of the optimizations described in Section 3.4, disagrees with this assumption, and this fact is reinforced with the performance of the new optimized implementations: sparse versions perform always faster than dense versions even for small problems, in a factor which ranges from 1.5 (prob-

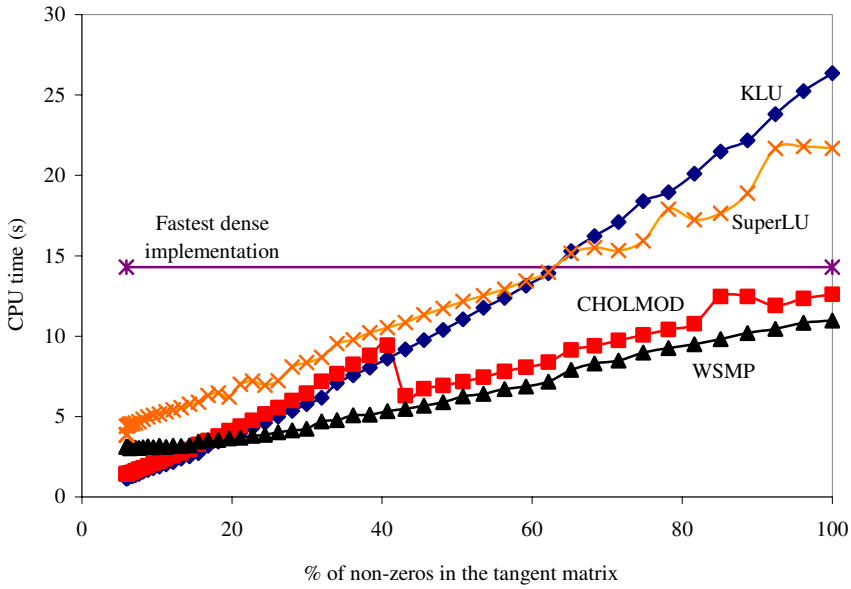

Figure 3.5: *Performance of different sparse linear equation solvers as a function of tangent matrix filling, for a problem size of 100 variables*

lems of 10 variables) to 5 (problems of 50 variables).

However, this conclusion has been obtained for the proposed test problem and dynamic formulation, and it could be argued that it cannot be generalized to other situations that lead to a coefficient matrix with a higher percentage of non–zeros, as in the case of highly constrained mechanisms or recursive formulations. The objection could be made to the efficiency ranking shown in Figure 3.4. In order to get insight about this subject, the numerical experiments used to generate Figure 3.4 were repeated, but in this case artificial non–zeros were introduced in the mass matrix **M**, in order to generate a tangent matrix with a variable percentage of non–zeros. Figure 3.5 shows the CPU–times for a mechanism of 48 loops (100 variables), as a function of matrix filling. Results show that two sparse implementations, based on the CHOLMOD and WSMP sparse solvers, are always faster than the best dense implementation, even with 100% of non–zeros in the tangent matrix. This surprising result can be explained by the fact that the percentage of non–zeros is always under 100% in the Jacobian matrix, hence optimized sparse implementations achieve significant time savings in Jacobian operations, in comparison with dense implementations.

Results for other problem sizes are synthesized in Figure 3.6: the different regions represent the points (problem size, matrix filling) where each implementation delivers the best performance. For most MBS problems and dynamic formulations, a sparse implementation based on the KLU solver will be the front runner. However, recursive formulations (which result in a higher matrix filling) with a symmetric tangent matrix will benefit from a sparse implementation based on the WSMP solver.

Figure 3.6 has been obtained by using the KLU *refactor* routine for numerical fac-

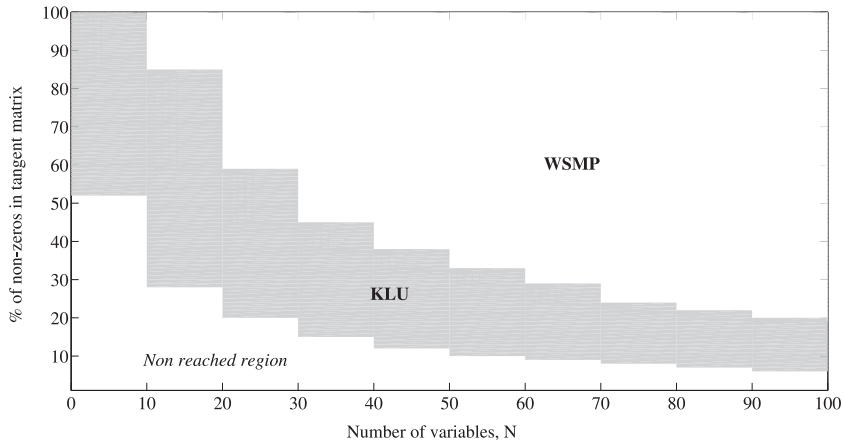

Figure 3.6: *Best implementation, as a function of problem size and percentage of non–zeros in the tangent matrix*

torizations. As explained in Section 3.4.2, this may cause trouble in problems where the entries of the tangent matrix change their relative values significantly during the simulation. If the *refactor* routine is not used, Figure 3.7 is obtained. In this case KLU is replaced by CHOLMOD, WSMP increases its influence area, and the dense implementation based on LAPACK emerges for very small problems (less than 10 variables), but with a very small advantage. Conversely, two exceptions can be mentioned:

- For dynamic formulations with symmetric indefinite tangent matrices, WSMP would be the front runner for almost all the situations, since CHOLMOD does not support them.
- For dynamic formulations with unsymmetric tangent matrices, KLU would be again the front runner for almost all the situations (even if the *refactor* routine is avoided), since WSMP does not support them.

3.6 Conclusions

Regarding the implementation aspects of MBS dynamic simulations, the following conclusions can be established:

- Efficient linear algebra implementations can speed up the efficiency in a factor of 2–3, compared with traditional implementations.
- The proposed optimizations based on symbolic pre–processing of the sparse matrix computations can deliver huge speedups, since off–the–shelf sparse matrix libraries do not take advantage of the constant sparse pattern of operations during the dynamic simulation.

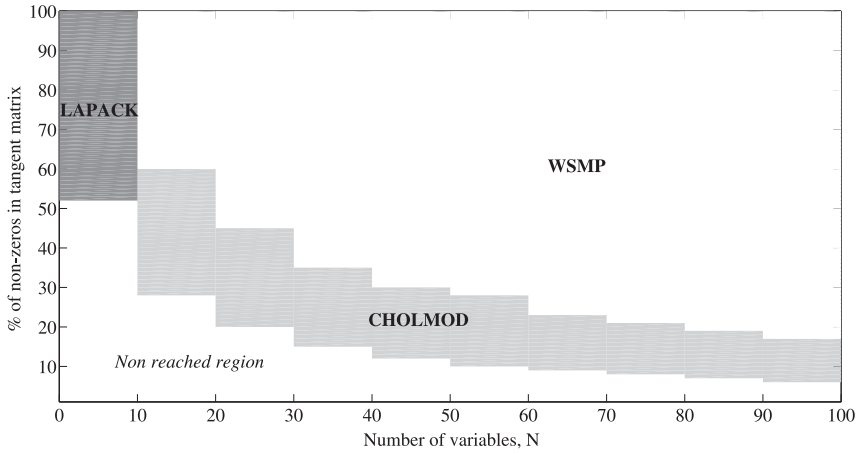

Figure 3.7: *Best implementation, as a function of problem size and percentage of non–zeros in the tangent matrix (*refactor *routine of KLU is not used)*

- Optimized sparse implementations are recommended since they perform better than optimized dense implementations, even for small–size problems or relatively dense matrices. This disagrees with the widespread belief in MBS dynamics.
- Concerning sparse linear equation solvers, it has been found that KLU, an unfamiliar solver designed for circuit simulation, performs very well with many of the linear equation systems resulting from MBS dynamics. In addition, it was found that the reference BLAS implementation provides the best compromise between performance and usability for sparse implementations.

Table 3.3: *Decision rules for selecting the best sparse solver for MBS dynamics, based on matrix type, size and number of non–zeros*

Type of tangent matrix	$N \times$ (% of non–zeros -10)	
	>900	<900
Symmetric positive definite	KLU (smooth problems) CHOLMOD (rough problems)	WSMP
Symmetric	KLU	WSMP
Unsymmetric	KLU	KLU

The results from numerical experiments are summarized in Table 3.3, which provides a simple decision rule to select the best linear equation solver for MBS dynamics, based on matrix type, size and percentage of non–zeros. Efficient implementations of global MBS dynamic formulations can be easily achieved, provided the above recommendations are followed. All the recommended software libraries are freely avail-

able, and the proposed optimization techniques are not bound to any programming language.

As a consequence of the above–mentioned conclusions, the limit for problem size where global formulations perform better than recursive or semi–recursive formulations, established in the order of 50 absolute variables (Cuadrado et al., 2004b, 2008), should be revised. This limit might get higher if the proposed optimized sparse implementations are used, since their effects on the efficiency are higher in global formulations than in recursive or semi–recursive formulations. In addition, further work must be carried out in order to determine if the proposed recommendations are still valid for other formulations, since all the numerical experiments have been performed using a particular global formulation.

Chapter 4

Parallelization

This Chapter evaluates two non–intrusive parallelization techniques for multibody system dynamics: parallel sparse linear equation solvers and OpenMP. Both techniques can be applied to existing simulation software with minimal changes in the code structure; this is a major advantage over MPI (Message Passing Interface), the standard parallelization method in multibody dynamics. Both techniques have been applied to parallelize a starting sequential implementation of a global index–3 augmented Lagrangian formulation combined with the trapezoidal rule as numerical integrator, in order to solve the forward dynamics of a variable–loop four–bar mechanism. Numerical experiments have been performed to measure the efficiency as a function of problem size and matrix filling. Results show that the best parallel solver (Pardiso) performs better than the best sequential solver (CHOLMOD) for multibody problems of large and medium sizes leading to matrix fillings above 10 non–zeros per variable. OpenMP also proved to be advantageous even for problems of small sizes. Both techniques delivered speedups above 70% of the maximum theoretical values for a wide range of multibody problems.

4.1 Introduction

Computational efficiency of numerical simulations is a key issue in multibody system (MBS) dynamics. When MBS dynamics is used in Computer Aided Design and Engineering, faster simulations allow the design engineer to perform what–if analyses and optimizations in shorter times, increasing productivity and interaction with the model. Moreover, some applications like hardware–in–the–loop settings or human–in–the–loop devices cannot be developed unless MBS forward dynamic simulations are performed in real–time. Hence, computational efficiency is a very active area of research in multibody systems dynamics.

Parallel computing is one of the approaches to increase the computational efficiency of MBS dynamic simulations. The first parallel MBS algorithm was proposed by Kasahara et al. in 1987; since then, a variety of formulations and simulation algorithms have been developed to exploit parallel computing architectures in MBS

dynamics (Anderson et al. (2007), Anderson and Oghbaei (2005), Critchley and Anderson (2003), Critchley and Anderson (2004), Cuadrado et al. (2000), Eichberger et al. (1994), Fisette and Péterkenne (1998), among others). Some of these algorithms apply parallelization directly at the level of equations of motion, which are formulated in a form that facilitates the concurrent evaluation of their different terms, see e.g. Bae et al. (1988) and Avello et al. (1993); most of these algorithms are based on recursive or semi–recursive formulations. Other algorithms apply substructuring techniques to partition the multibody system in disjoint subdomains, which are solved concurrently taking into account the interconnection constrains, see e.g. Mukherjee et al. (2005) and Quaranta et al. (2002). With regard to the implementation, the Message Passing Interface (MPI) (Argonne National Laboratory, 2009) has become the de facto standard for the parallelization of multibody dynamic simulation codes, e.g. Anderson et al. (2007), Anderson and Duan (1999) and Quaranta et al. (2002). MPI is a message–passing application programmer interface that provides functionality to enable communication and synchronization between a set of processes which run concurrently. Due to its language independence, high performance, scalability and good portability through completely different parallel architectures (from shared–memory processors to computer clusters), it has been broadly accepted in the field of parallel MBS dynamics.

The aforementioned parallel methods for MBS dynamics could be described as *intrusive* parallelization, since they introduce major modifications both in formulations and implementations. Formulations are specifically designed to obtain highly parallelizable numerical computations, and most importantly, parallel MPI–based implementations enforce a particular MPI–oriented code design: the programmer must explicitly divide tasks in processes and insert message–passing operations for data transfer and synchronization. As a result, the structure of an MPI–based parallel code is usually quite different from its sequential counterpart. These parallelization methods have been proved to attain very good results in terms of efficiency and scalability in the context of MBS dynamics, as demonstrated e.g. by Anderson et al. (2007) and Quaranta et al. (2002). However, their intrusive character makes them quite difficult to apply to existing sequential MBS dynamic simulation codes. Many of these sequential packages, developed by academia, still have a great value as research tools and they are successfully used in ongoing industrial applications. Due to their internal complexity and design dependencies with third–party software, parallelization of these MBS packages by intrusive methods like MPI would be very time–consuming and error–prone. For that reason, most of them remain as sequential codes which cannot take advantage of today's almost ubiquitous availability of parallel computing architectures, present even in low–cost laptop computers. This limitation will be accentuated in the future, since trends indicate that performance of single processors is close to reaching its limit and that multi–core processors are the preferred technology to increase computing power in the next decade (Gorder, 2007).

The goal of this Chapter is to investigate alternative *non–intrusive* parallelization methods for MBS dynamics, which do not require major modifications in existing formulations and implementations. Although their scalability may be inferior when

compared to intrusive methods, such non–intrusive methods could be easily applied to parallelize the above–mentioned legacy sequential MBS simulation packages, and they may also reduce the effort required to develop some kinds of new parallel formulations and implementations. This Chapter deals with two non–intrusive parallelization methods for MBS dynamics:

- the use of parallel sparse linear equation solvers; and
- the OpenMP parallel programming model.

Linear equation solvers represent an opportunity for non–intrusive parallelization since the solution of linear equation systems is a key process in many MBS dynamic simulation codes. This linear algebra operation is present in almost all simulation methods except some types of fully recursive formulations (García de Jalón and Bayo, 1994), although its weight in the total computation time of the simulation depends on the type of problem and formulation. Global formulations, which use a high number of coordinates and constraint equations to define the position of the multibody system, lead to comparatively big sparse linear equation systems whose solution usually consumes around 30–60% of the total CPU–time in a dynamic simulation. Recursive and semi–recursive formulations lead to smaller and more compact linear equation systems, and therefore their weight is reduced to less than 30% of the total CPU–time; however, if flexible bodies are considered, matrix sizes increase and the solution of linear equation systems also takes a significant percentage of the CPU–time, even for recursive formulations. As a result, the performance of the linear equation solver is critical to the efficiency of most MBS dynamic simulations. The replacement of a sequential solver by a parallel solver is considered a non–intrusive parallelization technique because it only requires minor changes in the code, provided that both solvers use similar sparse matrix storage formats. Many parallel linear equation solvers have been developed in the last years, but they are not considered to be appropriate for MBS dynamics due to the small matrix sizes involved in this field of computational mechanics. Comparative studies about their performance have been published by Gupta (2002, 2007), Davis et al. (2003) and Tracy et al. (2007); however, the test problems used in these studies do not fit the particular features of MBS dynamics, specially in regard to matrix sizes (in MBS dynamics, typical sizes are at least two orders of magnitude smaller than in Finite Element Analysis or Computational Fluid Dynamics), and therefore their conclusions cannot be extrapolated since parallel solvers will perform very differently under these circumstances. The first contribution of this Chapter is the evaluation of the efficiency and suitability of parallel sparse linear equation solvers in the context of multibody system dynamics, a subject that has not been investigated yet.

The second non–intrusive parallelization method explored in this Chapter is the OpenMP parallel programming model (OpenMP Architecture Review Board, 2008). OpenMP is a standard application programming interface to support multi–threaded parallel programming. It is scalable and portable like MPI, but it has two important differences. First, OpenMP is only targeted at shared–memory multiprocessor architectures, while MPI supports both shared– and distributed–memory architectures. How-

ever, this OpenMP limitation is not a severe disadvantage in the field of MBS forward dynamics: due to the characteristics of the problem, concurrent tasks running a parallelized simulation must exchange data several times per integration step (usually in the order of milliseconds), causing a high communication overhead compared with other applications. As a consequence, gains obtained from concurrent computation can be easily outweighed by the high communication overhead in distributed–memory architectures like PC clusters (Quaranta et al., 2002). Conversely, the low communication overhead of shared–memory architectures, supported by OpenMP, makes them more appropriate to run parallel MBS simulations. Another advantage of shared–memory architectures is the availability of low–cost commodity hardware with 2 or 4 CPU cores, like Intel Core 2 Quad and AMD Phenom X4. The second core difference between OpenMP and MPI concerns with the programming model: OpenMP is based on a multi–threaded model simpler to use than the MPI's multi–process model. This key difference delivers important advantages when OpenMP is applied to parallelize a sequential code (Chapman et al., 2007):

- the initial design can be maintained and only minor changes in the code are required;
- data transfer and task synchronization are handled transparently by OpenMP; and
- parallelization can be applied incrementally.

These three advantages make OpenMP a non–intrusive parallelization method when compared to MPI. On the other hand, Krawezik and Cappello (2006) demonstrated that OpenMP cannot achieve the same performance as MPI for some types of numerical problems and code designs, hence its pros and cons in a particular domain shall be evaluated before claiming it as a better technique than MPI. Despite its potential advantages, studies about the efficiency of OpenMP in the context of MBS dynamics have not been published yet, and this subject will be the second contribution of this Chapter.

The rest of the Chapter is organized as follows: Section 4.2 describes the numerical experiments used to evaluate the efficiency and applicability of the two proposed non–intrusive parallelization methods: test problem, dynamic formulation, and parallelization procedures applied to a starting sequential implementation. Section 4.3 presents and analyzes the results of numerical experiments. Finally, Section 4.4 extracts conclusions and suggests future work.

4.2 Methods

In order to study the efficiency and applicability of the two proposed non–intrusive parallelization methods, a test problem has been solved with a given dynamic formulation. This formulation has been initially implemented in a sequential simulation code, which has been parallelized by means of parallel linear equation solvers and OpenMP.

This test setup represents a worst–case scenario for parallelization in terms of problem, dynamic formulation and implementation, as it will be explained in the following subsections. With this approach, the obtained performance results will represent a lower limit when the non–intrusive parallelization methods investigated in this Chapter are applied to legacy MBS simulation codes.

4.2.1 Test problem and dynamic formulation

The selected test problem in this Chapter is the same that was chosen for the evaluation of linear equation solvers and described in Chapter 3: the 2D L–loop four–bar linkage. This model is shown in Figure 4.1.

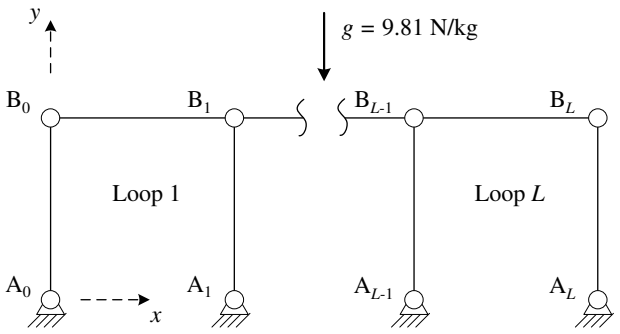

Figure 4.1: *L–loop four–bar linkage*

The dynamic simulation is performed by means of the index–3 augmented Lagrangian formulation,

$$\begin{aligned} &\mathbf{M}\ddot{\mathbf{q}} + \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\alpha\mathbf{\Phi} + \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\lambda}^{*} = \mathbf{Q} \\ &\boldsymbol{\lambda}_{i+1}^{*} = \boldsymbol{\lambda}_{i}^{*} + \alpha\mathbf{\Phi}_{i+1}; \quad i = 0, 1, 2, ... \end{aligned} \tag{4.1}$$

with the trapezoidal rule as integrator

$$\begin{aligned} \dot{\mathbf{q}}_{n+1} &= \frac{2}{\Delta t}\mathbf{q}_{n+1} + \hat{\dot{\mathbf{q}}}_{n}; \quad \hat{\dot{\mathbf{q}}}_{n} = -\left(\frac{2}{\Delta t}\mathbf{q}_{n} + \dot{\mathbf{q}}_{n}\right) \\ \ddot{\mathbf{q}}_{n+1} &= \frac{4}{\Delta t^{2}}\mathbf{q}_{n+1} + \hat{\ddot{\mathbf{q}}}_{n}; \quad \hat{\ddot{\mathbf{q}}}_{n} = -\left(\frac{4}{\Delta t^{2}}\mathbf{q}_{n} + \frac{4}{\Delta t}\dot{\mathbf{q}}_{n} + \ddot{\mathbf{q}}_{n}\right) \end{aligned} \tag{4.2}$$

The dynamic formulation and the integrator were introduced in Section 3.2. However, as the tested parallelization techniques are closely related to the identification of the parallelizable computation steps, they are briefly described here again. Introducing Equations (4.2) in Equations (4.1), yields the following non–linear system of equations

$$\mathbf{f}(\mathbf{q}) = \mathbf{M}\mathbf{q}_{n+1} + \frac{\Delta t^{2}}{4}\left[\mathbf{\Phi}_{\mathbf{q}_{n+1}}^{\mathrm{T}}\left(\alpha\mathbf{\Phi}_{n+1} + \boldsymbol{\lambda}_{n+1}\right) - \mathbf{Q}_{n+1} + \mathbf{M}\hat{\ddot{\mathbf{q}}}_{n}\right] = \mathbf{0} \tag{4.3}$$

which is solved through the Newton–Raphson iteration

$$\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right]_i \Delta \mathbf{q}_{i+1} = -[\mathbf{f}(\mathbf{q})]_i \tag{4.4}$$

using the approximate tangent matrix (symmetric and positive–definite)

$$\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right] \cong \mathbf{M} + \frac{\Delta t}{2}\mathbf{C} + \frac{\Delta t^2}{4}\left(\mathbf{\Phi}_\mathbf{q}^{\mathrm{T}} \alpha \mathbf{\Phi}_\mathbf{q} + \mathbf{K}\right) \tag{4.5}$$

Finally, as the corresponding sets of $\dot{\mathbf{q}}^*$ and accelerations $\ddot{\mathbf{q}}^*$ may not satisfy $\dot{\mathbf{\Phi}} = \mathbf{0}$ and $\ddot{\mathbf{\Phi}} = \mathbf{0}$, they must be projected, reusing the factorization of the tangent matrix:

$$\begin{aligned}
\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right] \dot{\mathbf{q}} &= \left[\mathbf{M} + \frac{\Delta t}{2}\mathbf{C} + \frac{\Delta t^2}{4}\mathbf{K}\right] \dot{\mathbf{q}}^* - \frac{\Delta t^2}{4}\mathbf{\Phi}_\mathbf{q}^{\mathrm{T}} \alpha \mathbf{\Phi}_t \\
\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right] \ddot{\mathbf{q}} &= \left[\mathbf{M} + \frac{\Delta t}{2}\mathbf{C} + \frac{\Delta t^2}{4}\mathbf{K}\right] \ddot{\mathbf{q}}^* - \frac{\Delta t^2}{4}\mathbf{\Phi}_\mathbf{q}^{\mathrm{T}} \alpha \left(\dot{\mathbf{\Phi}}_\mathbf{q} \dot{\mathbf{q}} + \dot{\mathbf{\Phi}}_t\right)
\end{aligned} \tag{4.6}$$

This global method has been designed for sequential computation and it is not as suitable for parallelization as recursive and semi–recursive formulations. For that reason, it nearly represents a worst–case scenario for parallelization with regard to dynamic formulations. The numerical experiments have been performed using as a time–step $\Delta t = 10^{-3}$ s and a penalty factor $\alpha = 10^8$; the simulation time for each of them has been, again, 20 s.

The number of loops in the test mechanism can be adjusted to generate problems of different sizes. In MBS dynamics, global formulations generate several hundreds of variables when applied to automotive or railway vehicles made up of rigid bodies. Recursive formulations lead to problems of smaller size, but when body flexibility needs to be considered, the number of variables increases even with this kind of formulations. If flexible bodies are described by component mode synthesis, as explained by Ambrósio and Gonçalves (2001) and Lugrís et al. (2007), multibody models of automobile or railway vehicles can exceed 1000 variables. If non–linear elastic or plastic behaviour is considered, the number of variables in the problem is augmented by the degrees of freedom of the finite element discretization of the flexible bodies, see e.g. García Orden and Goicolea (2000) and Sugiyama and Shabana (2004); under these circumstances, multibody models in industrial applications may reach 10^4 variables. This number can be considered the upper–level limit in the field of conventional multibody dynamics, at least during the next decade, with the exception of some specific applications such as the simulation of molecular dynamics (Mukherjee et al., 2008). For that reason, the numerical experiments will be performed using a number of variables N that ranges from 100 to 8000 (generated by a number of loops L from 49 to 3999).

4.2.2 Initial sequential implementation

The initial implementation of the dynamic formulation has been heavily optimized for sequential execution by using efficient BLAS implementations for dense linear algebra, symbolic pre–processing of sparse matrix computations, fast access to sparse storage formats and state–of–the–art sequential lineal equation solvers, as described in Chapter 3. These optimizations reduced CPU–times by a factor of 3 compared with more traditional implementations of the same dynamic formulation. On the other side, such a highly optimized sequential code makes it difficult to gain advantage from parallelization: since computations are performed at higher FLOPS (Floating Point Operations per Second) rates and in shorter times, the relative weight of the communication overhead associated with parallelization becomes higher; in addition, some optimization techniques make fine–grain parallelization unable to be applied to certain code sections, as it will be explained later. Again, the described initial implementation represents a nearly worst–case scenario for parallelization. Indeed, the parallelization of this code by means of MPI would be very cumbersome.

Table 4.1: *Performance analysis of the initial sequential implementation for problems of N variables*

			% of elapsed time	
Task	Description	Eq.	$N = 1000$	$N = 8000$
1	Update of variables	–	4.1	4.0
2	Evaluate dynamic terms	(4.1 and 4.5)	9.3	9.8
3	Evaluate tangent matrix	(4.5)	11.8	11.8
4	Evaluate residual vector	(4.1)	7.6	7.6
5	Factorize tangent matrix	(4.4)	36.8	36.7
6	Back–substitution	(4.4)	5.9	5.8
7	Project velocities	(4.6)	9.4	9.3
8	Project accelerations	(4.6)	12.3	12.2
9	Other	–	2.8	2.8
Total elapsed time (s)			10.0	102.4

Table 4.1 summarizes the results of a performance analysis of the initial sequential formulation for tests problems of 1000 and 8000 variables. Both cases show very similar profiling results, since the use of symbolic pre–processing of sparse computations through all the code leads to nearly $O(n)$ tasks in spite of using a dynamic formulation usually classified as $O(n^3)$. This performance analysis will be used to guide the parallelization described in the next subsections.

4.2.3 Parallelization with multi–threaded linear equation solvers

Table 4.1 shows that around 54% of the CPU–time is consumed by the solution of linear equation systems: matrix factorization (task 5, close to 37%) and back–substitutions (task 6 and part of tasks 7 and 8). This high contribution is caused by

the simplicity of the dynamic terms in the proposed test problem (task 2); in problems with time–consuming force, constraint and Jacobian evaluations, task 2 can achieve higher percentages of runtime and reduce the contribution of linear equation systems. Nevertheless, this operation is a significant bottleneck in most MBS dynamic simulations and represents an important opportunity for non–intrusive parallelization.

In the previous Chapter, the efficiency of different dense and sparse sequential linear equation solvers in the simulation of MBS dynamics was measured; the number of variables N in that study ranged from 10 to 1000. Results demonstrated that current state–of–the–art sparse implementations outperform dense implementations even for very small problems (e.g., 20 variables), contradicting a widespread conviction in MBS dynamics. Three sequential solvers were found to be the most efficient ones, as a function of the type of multibody problem and dynamic formulation, and trends in that previous Chapter indicate that they are also the most efficient solvers for $N > 1000$:

- CHOLMOD, a symmetric positive definite solver;
- KLU, an unsymmetric solver specially designed for circuit simulation; and
- WSMP (sequential version), a symmetric indefinite or unsymmetric solver.

In this Chapter, these three top–performing sequential solvers will be compared against parallel solvers. Given the large number of existing parallel sparse solvers, a selection process has been applied to narrow the scope: iterative solvers have been discarded, since they have a high communication overhead during each iteration, so they work efficiently only for very large problems out of the scope of MBS dynamics (Saad, 2000); the same argument applies to out–of–core solvers. From the remaining parallel linear equation solvers, three of them which have demonstrated good performance for matrix sizes close to those found in MBS dynamics (Gupta, 2002, 2007) have been selected to evaluate their performance in this field:

- SuperLU (multi–threaded version), a non–symmetric solver (Demmel et al., 1999b);
- Pardiso, an either symmetric or unsymmetric, positive definite or indefinite solver (Schenk et al., 2001); and
- WSMP (multi–threaded version).

The efficiency of a linear equation solver depends on three factors: matrix size, sparsity pattern and matrix filling. In this study, the effect of matrix size has been analyzed by solving the test problem with a number of variables N ranging from 100 to 8000. The effect of the sparsity pattern has been greatly diminished by reordering the tangent matrix: each of the six benchmarked solvers supports different reordering strategies, usually computed by third–party numerical libraries like METIS (Karypis and Kumar, 1998) and AMD and its variants (Amestoy et al., 1996), among others; all of them have been tested in the symbolic pre–processing stage of the simulation, to select the best one for each simulation case. For that reason, the results obtained for the proposed test

Table 4.2: *Typical matrix filling ratios in multibody dynamics (N = number of variables, NNZ = number of non–zeros)*

Type of problem and dynamic formulation	NNZ/N
Rigid bodies – Global formulations	<10
Rigid bodies – Recursive formulations Flexible bodies – Component mode synthesis	10 – 30
Flexible bodies – Finite element mesh	30 – 100

problem will be still valid for other multibody problems leading to different sparsity patterns.

With regard to matrix filling, the described formulation applied to the test problem of L loops leads to a tangent matrix in Equation (4.5) of size $N = 2L + 2$ with $12L + 4$ structural non–zeros. A meaningful matrix filling indicator can be computed as the ratio between the number of non–zeros (NNZ) and N. In this case, $NNZ/N \approx 6$ is a typical value for global formulations applied to problems involving rigid bodies. Nevertheless, other dynamic formulations and multibody problems may lead to higher filling ratios, as depicted in Table 4.2. Problems involving rigid bodies lead to higher fillings if recursive or semi–recursive formulations are used (Cuadrado et al., 2004a); the same filling range applies if the problem involves flexible bodies and they are described by component mode synthesis (Ambrósio and Gonçalves, 2001; Cuadrado et al., 2001). Finally, if flexibility is described by introducing the degrees of freedom of the finite element discretization in the multibody problem (García Orden and Goicolea, 2000; Sugiyama and Shabana, 2004), the filling of the finite element mass and stiffness matrix dominates the tangent matrix; in these cases, matrix filling ranges from 30 to 100, depending on the type of finite element (beam, shell, brick).

It is expected that the performance gains from parallel solvers increase as the NNZ/N ratio increases, due to the higher workload of the numerical factorization. The proposed test problem and dynamic formulation represent a worst–case scenario for parallel solvers because they lead to a very small NNZ/N ratio; in these circumstances, parallel solvers could perform worse than sequential solvers. Conversely, more complex and realistic problems lead to higher NNZ/N ratios, as described in the previous paragraph. In order to study the effect of parallel solvers in such cases, a variable number of artificial non–zeros will be added in the sparsity pattern of the original tangent matrix to increase its NNZ/N ratio.

Only minor changes were required in the initial sequential implementation to incorporate the three proposed parallel solvers, because they use the same storage format of the three above–mentioned sequential solvers already supported by the simulation code (Compressed Column Storage format or CCS). For solvers used in symmetric mode (CHOLMOD, WSMP, Pardiso), only the upper or lower triangular part of the tangent matrix is computed in Equation (4.5), depending on the requirements of each solver; for non–symmetric solvers (KLU, SuperLU), the whole matrix is evaluated and factorized.

Some benchmarks for linear equation solvers (Gupta, 2002, 2007) only measure factorization and solve (forward triangularization and back substitution) times. In this work, the total elapsed time for a multibody dynamic simulation was measured, since this procedure takes into account other important attributes like precision (more precise solvers will need less iterations in Equation (4.4)) and memory footprint (its effect on the behaviour of CPU–cache can affect the overall performance of the simulation).

4.2.4 Parallelization with OpenMP

OpenMP (OpenMP Architecture Review Board, 2008) is a standard application programming interface (API) to support multi–threaded parallel programming in shared–memory architectures. It provides a set of *directives* that can be added to a sequential program in Fortran, C, or C++ to describe, with minimal modifications in the code, how the work is to be distributed among multiple threads that run in parallel. A good description of OpenMP is provided by Chapman et al. (2007).

```
// Calls 2 functions in parallel
void example1()
{
     #pragma omp parallel sections

     #pragma omp section
     function1();
     #pragma omp section
     function2();
}

// '1'-norm of a vector in parallel
double example2(double v[], int n)
{
     double sum = 0;
     #pragma omp for reduction(+:sum)
     for (int i=0; i<n; i++) {
          sum = sum + v[i];
     }
     return sum;
}
```

Figure 4.2: *Example of OpenMP directives for parallelization*

Figure 4.2 shows a couple of examples of parallelization with OpenMP: the first one calls two code sections in parallel, while the second one splits a *for* loop into several non–overlapping fragments to traverse them in parallel and accumulate the results. These directives are understood by OpenMP compilers, which deal with the burden of working out the communication and synchronization details of the parallel program. The directives look like comments to regular, non OpenMP–aware compilers, which will generate sequential code. In this way, the same source code can be used in both sequential and parallel versions; this feature can simplify the maintenance of

MBS simulation codes that are used to run simulations in both desktop PCs (suitable for parallel execution) and embedded microprocessors (which only support sequential execution) like automotive Electronic Control Units (ECU's).

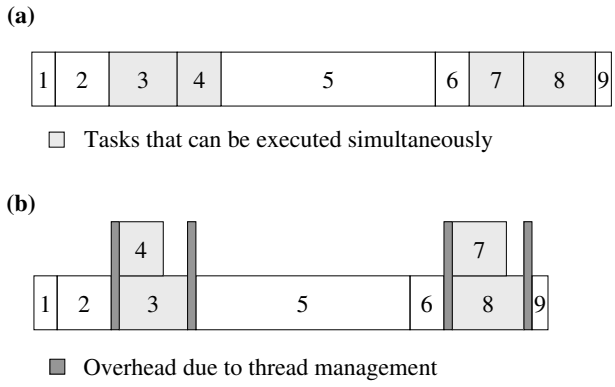

Figure 4.3: *Distribution of computational load in* **(a)** *the initial sequential version and* **(b)** *the proposed parallel version*

Coarse–grain parallelization, in which large program regions are executed concurrently, can be easily achieved with OpenMP. An analysis of the profiling results in Table 4.1 and the sequence of calculations in Equations (4.1) to (4.6) evidences that two pairs of tasks (3–4 and 7–8) can be executed concurrently, as shown in Figure 4.3. On the other hand, tasks 1 and 2 cannot be scheduled in parallel because the second one requires the values previously computed by the first. In addition, some of the optimizations implemented in the initial sequential version make not possible to apply fine–grain parallelization. For example, the Jacobian evaluation, which represents around 80% of task 2, has been optimized for fast writing operations to matrix data stored in CCS format. This optimization reduced the evaluation time by a factor of 3 but it requires a sequential traversing of the involved *for* loop, which cannot be split like in Figure 4.2.

Figure 4.3 and the details given in the previous paragraph confirm that the proposed test problem and dynamic formulation represent a worst–case scenario for the parallelization with OpenMP, since most of the tasks must be executed sequentially. In contrast, other simulation setups (e.g. recursive) can be used to generate algorithms where most of the time–consuming tasks can be parallelized, although this also depends on the structure of the multibody system.

4.2.5 Test environment and implementation details

Simulations have been run in a desktop PC with a dual–core Intel Core Duo E6300 CPU (1.86 GHz clock speed in each core, 64 Kb L1 cache, 2 Mb L2 cache) and 1 Gb of RAM, running Linux OS kernel 2.6.24 in 64 bit mode. Two compiler toolchains

have been used: the GNU Compiler Collection (gcc version 4.3) and the Intel C/C++ Compiler (icc version 10.1); both of them support OpenMP.

A parallel computer with only two CPUs has been used because the tested dynamic formulation, heavily oriented to sequential execution, will deliver poor scalability since the fraction of parallelizable code is relatively small. The goal of this Chapter is to test whether the proposed non–intrusive parallelization methods can increase the efficiency of MBS dynamic simulations; if they can, the scalability of the speedups will greatly depend on the multibody problem and dynamic formulation.

4.3 Results and discussion

The following subsections present numerical results for the two above–mentioned non–intrusive parallelization methods.

4.3.1 Multi–threaded linear equation solvers

Figure 4.4 shows the elapsed times for dynamic simulations with a number of variables N ranging from 100 to 4000 and three representative values of the matrix filling ratio according to Table 4.2 (NNZ/N = 6, 20, 50). Sequential single–threaded (st) solvers are represented in dashed lines, while parallel multi–threaded (mt) solvers are represented in solid lines. Elapsed times for N in the range 4000–8000 follow the trends indicated on the right side of the figures, so they have not been represented. Figure 4.4a evidences that parallel solvers are not competitive for problems with low filling ratios: in these circumstances, KLU (unsymmetric solver) and CHOLMOD (symmetric positive definite solver) perform better than any other. The efficiency of KLU is outstanding in this case, taking into account that, due to its unsymmetric nature, the whole tangent matrix is evaluated and factorized during the simulation. The explanation for this excellent behaviour is that KLU is a sparse LU factorization algorithm specially designed for its use in circuit simulation problems, which have a typical filling ratio of 7–8; however, this feature is also an important penalty for filling ratios above 10. For medium (Figure 4.4b) and high (Figure 4.4c) filling ratios, Pardiso emerges as the best solver for problems of large size. For medium size problems, CHOLMOD continues to be the most efficient solver under these circumstances.

In order to gain insight into the most favourable conditions for each solver, numerical experiments similar to those represented in Figure 4.4 have been run with a matrix filling ratio within a range from 6 to 100. Results are synthesized in Figure 4.5, which represents the regions where each solver delivers the best performance, as a function of the number of variables N and the filling ratio NNZ/N. The solid line draws up the boundary between the parallel and the sequential solvers, and the dashed lines draw up the boundary between different sequential or parallel solvers. This figure serves as a decision tool to identify which solver is best suited for a particular multibody simulation. Figure 4.5 shows that, contrary to general beliefs, parallel solvers can increase simulation efficiency for a wide range of problems in MBS dynamics. Pardiso dominates the region of parallel solvers, since the multi–threaded version of WSMP is only

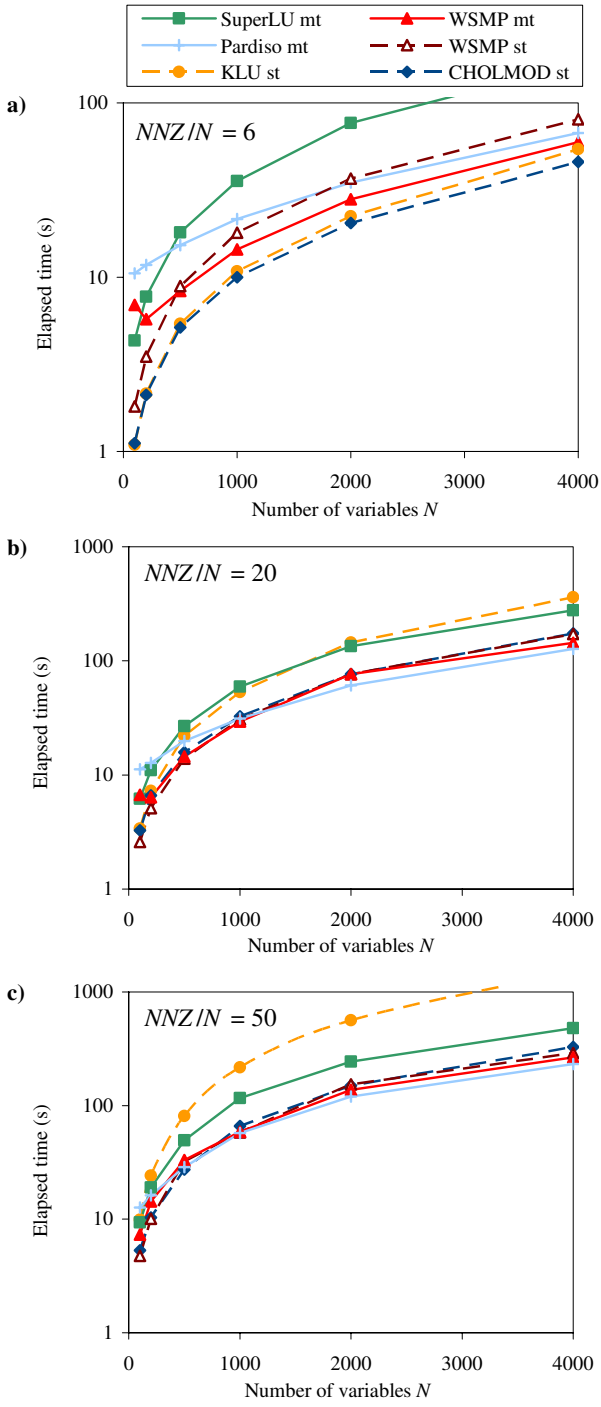

Figure 4.4: *Performance of linear equation solvers as a function of problem size and matrix filling*

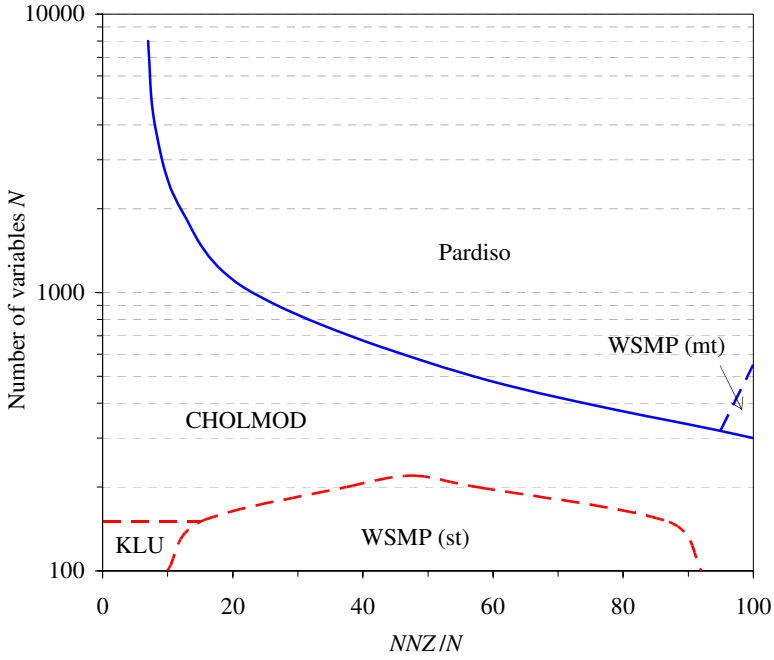

Figure 4.5: *Best solver, as a function of problem size and matrix filling*

better in a small, non–representative region. On the other hand, CHOLMOD dominates the region of sequential solvers, while KLU and single–threaded WSMP are only competitive for small problems under 200 variables; these last results fully agree with the recommendations given in Chapter 3 for problems under 1000 variables.

Since Pardiso has been demonstrated to perform better than sequential solvers for many multibody problems, it is important to quantify the speedups that it can deliver. Figure 4.6 represents the speedups achieved by Pardiso, as a function of the filling ratio NNZ/N and the number of variables N; the speedup S is relative to the best sequential solver in each point of the figure:

$$S = \frac{elapsed\ time_{sequential}}{elapsed\ time_{parallel}} \tag{4.7}$$

Table 4.3 shows the maximum speedup that can be achieved by a parallel solver in the tested implementation, for three typical values of the filling ratio; the values have been obtained from profiling results and Amdahl's law: for a program with a parallel fraction f running on P processors, the maximum speedup is:

$$S\,(P)_{\max} = \frac{1}{f/P + 1 - f} \tag{4.8}$$

Table 4.3: *Maximum speedup for 2 processors due to parallelization of the linear equation solver in the tested implementation, as a function of the matrix filling ratio* NNZ/N

NNZ/N	CPU–time in factorizations and back–substitutions	Max. speedup S
6	52%	1.35
20	69%	1.53
50	68%	1.52

The information given in Figure 4.6 and Table 4.3 is important in order to correctly interpret the results in Figure 4.5. While Pardiso performs better for $N < 1000$ in a significant region of Figure 4.5, the delivered speedups are very small compared with the best sequential solver (CHOLMOD), specially for $NNZ/N > 50$. Pardiso only delivers significant speedups for $N > 1000$, and it achieves the maximum performance for NNZ/N in the range from 10 to 30. In some cases, the speedups exceed 70% of the maximum values predicted by Amdahl's law in Table 4.3.

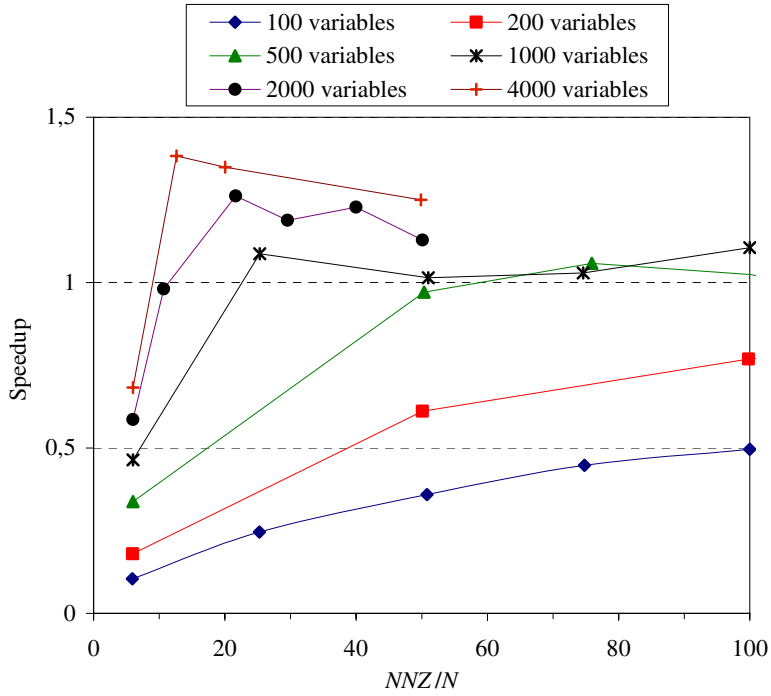

Figure 4.6: *Speedup of Pardiso with respect to the best sequential solver*

With regard to the effect of the compiler toolchain on the simulation efficiency, it has been observed that the two tested toolchains (GNU and Intel) can increase or decrease the elapsed times for the tested solvers by a factor up to 34%, depending on

matrix size and filling ratio. However, in the conditions where each solver performs better (according to Figure 4.5) the effect of the compiler is diminished, as shown in Table 4.4. In general, icc gives slightly better results than gcc, specially for Pardiso.

Table 4.4: *Effect of compiler toolchain on the efficiency of linear equation solvers in the region where each solver performs best*

Linear equation solver	Best compiler	Min. gain	Max. gain
Pardiso	icc	7%	18 %
CHOLMOD	icc	1%	8 %
WSMP (st)	icc/gcc	-2%	2 %
KLU	icc	-1%	7 %

4.3.2 OpenMP

Figure 4.7 shows the elapsed times for dynamic simulations with the OpenMP parallel version of the code, for a number of variables ranging from 100 to 8000 and a filling ratio $NNZ/N \approx 6$ (no artificial non–zeros were added to the tangent matrix). The simulations have been run using CHOLMOD as linear equation solver. Since most of the burden of OpenMP parallelization is carried out by the compiler, results for both compiler toolchains (GNU and Intel) have been represented. Taking into account the profiling data in Table 4.1, the task schedule shown in Figure 4.3b can deliver a maximum speedup of 1.20. Results indicate that the compiler has a significant effect on the performance of the OpenMP parallel version. Intel OpenMP implementation, with a lower communication overhead, delivers speedups greater than one even for small problems of 100 variables, and it achieves the optimum conditions for around 500 variables. The GNU implementation needs more than 200 variables to become advantageous, and delivers the maximum values for 2000 variables; however, the speedups of GNU are higher, reaching the 95% of the maximum theoretical value (1.20).

Figure 4.7 also shows that OpenMP speedups start to fall for $N > 2000$. This fact does not agree with the normal behaviour of parallel programs: both the communication overhead due to parallelization and the maximum speedup do not depend on N and should be constant (the overhead of thread creation and destruction depends only on the number of threads, and Table 4.1 demonstrates that the relative elapsed times of the parallelized tasks do not depend on N). Therefore, the maximum speedup $S_{max} = 1.20$ should be a horizontal asymptote for the curve $S(N)$, as it happens in MPI parallel codes (Anderson and Duan, 1999). This weird behaviour may be produced by adverse effects in the cache memory, because tasks scheduled in parallel in Figure 4.3b share part of the data: both tasks 3 and 4 operate with the mass matrix, Jacobian matrix and constraint vector, and both tasks 7 and 8 operate with the tangent matrix factorization and other common terms. Since each CPU has its own private cache, common data terms must be transferred twice from memory to cache, and for large problem sizes the memory bandwidth becomes a bottleneck. This phenomenon is

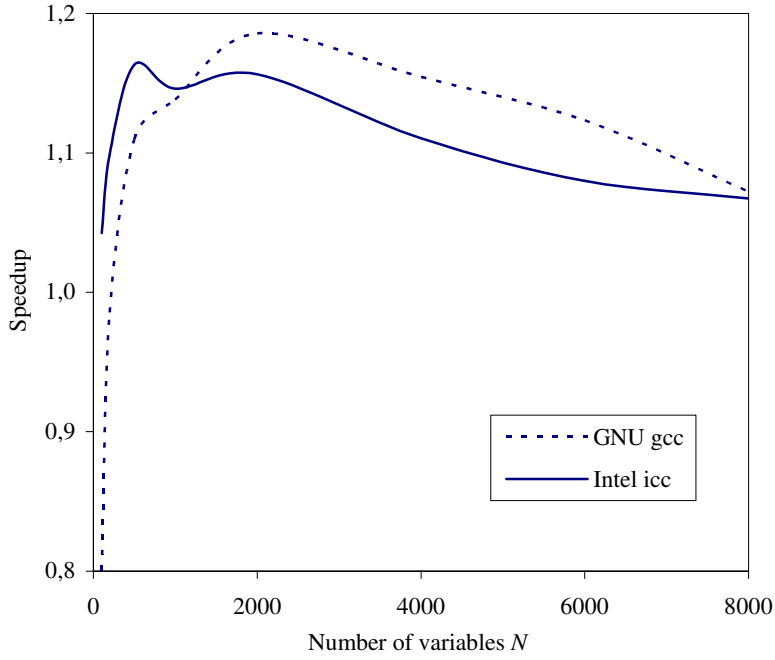

Figure 4.7: *Speedup of the OpenMP parallel implementation*

not frequent in MPI parallel implementations, since MPI processes operate in private, unshared data. Adverse effects of cache can be diminished with a proper allocation and distribution of data, as explained in Chapman et al. (2007). However, these techniques can enforce major changes in existing sequential MBS simulation codes, and their effect highly depends on the computer architecture and the compiler toolchain; therefore, they cannot be considered as non–intrusive or easy to implement.

Nevertheless, results demonstrate that OpenMP is advantageous even for small problems and that it can deliver speedups above 80% of the maximum theoretical value for a wide range of problem sizes (from 500 to 3500 variables), provided the appropriate compiler toolchain is selected. Given the simplicity of its application to sequential codes, it is a valuable tool for non–intrusive parallelization of existing MBS simulation packages.

The attainable absolute speedups depend on the problem and the simulation characteristics. As described in Section 4.2.4, the test setup used in this work represents a worst–case scenario for OpenMP parallelization, and therefore the absolute performance gains are small (around 15%). However, the results from this study suggest that OpenMP can deliver higher absolute speedups under more realistic multibody system simulations. For example, Lugrís et al. (2007) describe two formulations for flexible multibody dynamics that spend up to 82% of the elapsed time in computing matrix terms associated with flexible bodies; since these matrix terms are evaluated body by

body, when several flexible bodies are present the parallelization of these tasks with OpenMP would be straightforward, and absolute speedups above 2 could be easily achieved on a quad–core processor. Problems with very time–consuming force evaluations can also achieve high improvements due to OpenMP parallelization. For example, multibody simulations involving collisions between bodies must perform collision detection at every time–step in order to evaluate contact forces. Body geometries are usually described by complex and dense polygonal meshes, and collision detection algorithms must evaluate distances between a large number of polygons; in many cases, this task can be divided into several subtasks which can be easily parallelized with OpenMP.

4.4 Conclusions

In the present Chapter, two non–intrusive parallelization techniques, parallel linear equation solvers and OpenMP, have been used to parallelize a starting sequential implementation of an MBS dynamic simulation software, in order to investigate their efficiency and suitability in the field of multibody dynamics. Both techniques are usually considered not appropriate for MBS dynamics due to the small sizes of matrix computations involved in this field.

Regarding the efficiency and suitability of parallel sparse linear equation solvers, the following conclusions can be established:

- Parallel solvers are advantageous for two types of multibody problems: (a) problems with more than 2000 variables leading to matrix filling ratios NNZ/N from 10 to 30 (the case for rigid multibody problems with recursive formulations or flexible multibody body problems modelled by component mode synthesis), and (b) problems with more than 2000 variables leading to matrix filling ratios NNZ/N above 30 (the case for flexible multibody body problems solved by introducing the finite element discretization in the formulation). Out of these two regions, sequential solvers (specially CHOLMOD) are more efficient.

- Pardiso is the most efficient parallel solver in the above–mentioned conditions among the three tested parallel linear equation solvers (SuperLU, Pardiso and WSMP).

- The speedups delivered by Pardiso in the above–mentioned conditions exceed 70% of the maximum theoretical value predicted by Amdahl's law for matrix filling ratios in the range from 10 to 30. Beyond that point, speedups fall gradually. In addition, the speedups become higher as the problems increase their size.

Regarding the efficiency and suitability of the non–intrusive OpenMP parallel programming model, the following conclusions can be established:

- The parallelization of several tasks of an existing sequential dynamic simulation software was very easy to implement with OpenMP.

- The OpenMP parallel version proved to be advantageous even for small problems of 100 variables, and the speedups exceeded 80% of the maximum theoretical value predicted by Amdahl's law for problem sizes in the range from 500 to 3500 variables.
- Beyond a certain problem size (2000 variables), the speedups fall gradually. This abnormal behaviour could be caused by adverse effects in the CPU's cache memories.
- The compiler toolchain has a significant effect on the efficiency of OpenMP: Intel icc performs better for problems of less than 1000 variables, while GNU gcc performs better for larger problems.

Despite the fact that both parallelization techniques cannot deliver high absolute speedups due to their non–intrusive character, their application is straightforward and therefore they are very appropriate to achieve partial parallelization of existing sequential multibody simulation codes with minimal effort. In addition, the good performance and ease of use of OpenMP suggest that it could be a substitute of MPI in the development and implementation of new formulations specially targeted to parallel execution; this topic represents an open line for future work.

Chapter 5

Integration with MATLAB/Simulink

Simulation of complex mechatronic systems like an automobile, involving mechanical components as well as actuators and active electronic control devices, can be accomplished by combining tools that deal with the simulation of the different subsystems. In this sense, it is often desirable to couple a multibody simulation software (for the mechanical simulation) with external numerical computing environments and block diagram simulators (for the modelling and simulation of non–mechanical components).

In this Chapter, the in–house developed C++ MBS simulation software described in Chapter 2 has been coupled with the commercial tools MATLAB and Simulink, and different coupling techniques have been identified, implemented and tested in order to assess their computational performance. Two categories of coupling techniques have been investigated: those in which only one tool performs the integration (*function evaluation*) and those in which each tool uses its own integrator (*co–simulation*). Furthermore, the efficiency of the described coupling methods has been compared to that of equivalent monolithic models, and indications are provided to implement them in other simulation environments.

Results show that the use of state–of–the–art coupling techniques can reduce simulation times in one or two orders of magnitude with respect to standard techniques. Finally, advices are provided to select the coupling method best suited to a particular application, as a function of its efficiency and implementation effort.

5.1 Introduction

Machines, in general, consist of several different subsystems such as mechanical components and actuators as well as control systems. These subsystems represent engineering disciplines that are coupled together and the overall performance of the machine is defined by the operation of each individual subsystem as well as by the inter-

actions of subsystems. For this reason, the traditional design procedure where mechanical components, actuators and control methods are considered separately is not able to produce optimum solutions. The multibody system (MBS) simulation approach meets the challenge and can be used in the design process of a machine that consists of different subsystems. It is noteworthy, however, that complex non–mechanical components such as control loops and actuators often fall beyond the scope of traditional multibody codes.

As the industry requirements increase, the demanded degree of realism in the simulation of multidisciplinary systems is continuously growing, so the engineer needs to bear in mind different phenomena simultaneously when simulating a system. When evaluating the behaviour of an automobile, for example, not only an accurate representation of its mechanical elements is needed, but also of the electronic control systems (like ABS or traction control), the hydraulic components or the thermodynamics of its engine. The realistic simulation of such multidisciplinary system, as required, for instance, by Human/Hardware–in–the–Loop (HiL) devices, must handle each different subsystem in an efficient way.

Several ways of dealing with multidisciplinary systems can be found in the literature, as mentioned by Valášek (2008). Two main approaches can be distinguished: communication between different simulation tools, and uniform modelling. Uniform or monolithic modelling is based on representing all the subsystems of a multi–domain problem in the same language (Samin et al., 2007). Specialized software and languages exist for this purpose, such as ACSL (The AEgis Technologies Group, Inc., 2009), VHDL–AMS (IEEE P1076.1 Working Group, 2009), and Modelica (Modelica Association, 2009), that manage simultaneously the equations of the entire system. Another way of performing uniform modelling is based on the use of general mathematical software for defining and solving the equations of the system. Recently, this task has been simplified by the development of specific–domain modules in block diagram software, such as SimMechanics and SimHydraulics for MATLAB/Simulink (The Mathworks, Inc., 2009). Coupling of tools, on the other hand, is based on the combination of specialized tools for modelling each subdomain. These tools are interfaced during execution time in order to emulate the real interaction between physical subsystems. As stated by Kübler and Schiehlen (2000), this is the optimal approach for the simulation of multidisciplinary systems. It allows the selection of optimized settings for the simulation of each subsystem, such as the integration time–step, the numerical solver and other particular details. Furthermore, in many cases, these specialized tools have been developed during years by researchers, leading to robust and efficient software and wide collections of tested examples and toolboxes.

Coupling strategies can be further categorized into two main approaches, depending on how the integration is performed. The name *co–simulation* is usually reserved for those cases in which each simulation tool incorporates its own integrator. In this work, when the integration is performed only in one tool that requests values from the others, the name *function evaluation* will be used.

Commercial multibody packages have been incorporating multiphysics capabilities during the last years and many of them, for example SIMPACK (SIMPACK AG,

2009), offer a wide range of coupling possibilities to external software tools, as well as add–on modules with non–multibody functionality. When the multibody software has been developed by a non–commercial research group, as in the case of academia, and coupling capabilities need to be added to it, the programmer must often choose between several available implementation techniques. Currently, it is nontrivial to make this decision, as the research about the suitability of the different coupling techniques for particular applications has been overlooked. In particular, there is a lack of information about the amount of effort the implementation of a coupling strategy takes and, more importantly, the efficiency of a specific technique when compared to other strategies applicable to the same problem. A study of the impact on performance of different co–simulation time–steps and processor configurations, in a simulation involving SIMPACK and MATLAB has been carried out by Vaculín et al. (2004) for a truck model. However, the evaluation of the computational efficiency of different coupling techniques, and a comparison with the performance of equivalent monolithic models, when possible, has not been performed yet. To this end, test models must be selected and built up, and simulations performed in order to measure the overhead the coupling techniques give rise to.

A closely related open field of research in the simulation of multidisciplinary systems is the use of multirate integration schemes, which improves the numerical efficiency during the simulation of interacting subsystems with very different time scales. Multirate algorithms have been developed (Oberschelp and Vöcking, 2004; Shome et al., 2004), while, however, the implementation of these techniques in the communication between software packages, specially when block diagram software is involved, is still in progress. It is noteworthy that the numerical performance of multirate algorithms depends greatly on the co–simulation strategy selected for solving the problem. The understanding of the limitations imposed by block diagram software packages, and the definition of a convenient interface between them and other simulation tools is the first step in the implementation of the general scheme for multirate co–simulation that is shown in Chapter 6

In this Chapter, coupling techniques with external simulation tools have been used for widening the capabilities of the existing MBS software, through the addition of functionality with numerical computation environments (such as MATLAB, Scilab (INRIA, 2009b), Mathematica (Wolfram Research, 2009) or MATRIXx (National Instruments, 2009)) and block diagram simulators (Simulink, Scicos (INRIA, 2009a) or SystemBuild (National Instruments, 2009)). To this end, coupling possibilities between the MBS software developed in this thesis and MATLAB/Simulink are examined in detail. MATLAB has been selected for this work because of its wide acceptance in the research community, derived from its versatility and easiness of programming. A practical way of performing the coupling in real cases has been implemented for each technique. It is important to note that the coupling techniques introduced in this study are not limited to a specific mathematical package, but they can also be applied to other similar tools, as similar communication capabilities are available in them. Finally, a generic co–simulation interface, which manages the communication between MBS software and the block diagram package Simulink, has been created and im-

plemented. This interface is intended to allow multirate co–simulation, with different synchronization methods, between simulation tools.

This Chapter is organized as follows: Section 5.2 gives a general review of the existing techniques for communicating a multibody package with external simulation tools. In Sections 5.3 and 5.4, these techniques are implemented in the MBS software and a general software tool for numeric computations. Introduced computational strategies are utilized in two example problems. Finally, the conclusions of the work are summarized.

5.2 Coupling techniques

The expansion of the multibody software via communication with external simulation tools can be performed in several ways, which can be categorized as data files exchange, function evaluation and co–simulation approaches. The most straightfor-

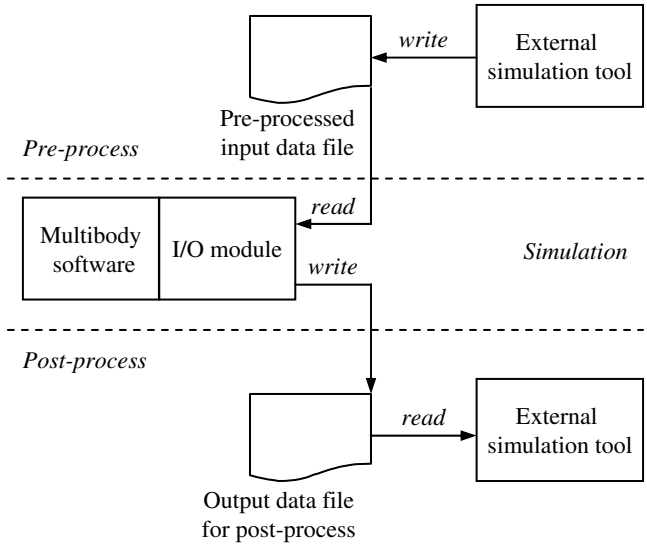

Figure 5.1: *Data file input–output configuration*

ward and easy to implement way of sharing data between two different simulation environments is the use of importing and exporting of data files. As the computational cost of read/write operations is high, this technique should not be applied during runtime. For this reason, files exchange approach should be reserved for pre– and post–processing operations, where computational efficiency is not a key factor. A scheme of this method is described in Figure 5.1. In the MBS simulation field, a large variety of tasks can be managed with files exchange approach adding to the multibody software the functionality of an external processing tool. The off–line realistic graphic representation of results and the pre–processing of complex dynamical terms when these are remaining constant during simulation are examples of this approach. The software

requirements for the use of this strategy are the existence of a common data format, understandable by the involved packages, and the availability of input–output routines for handling the data files in each program.

An alternative to data files exchange, more adequate for runtime, are *function evaluations* from one simulation tool to another. In this work, the name function evaluation is reserved for those communications in which only one of the software tools is actually performing a numerical integration, while the other one returns values on request, from the output values passed by the integrator tool. This configuration can be achieved through code exporting (via joint compiling, together with the integrator tool, or pre–compiled as a library) or by direct communication between processes. Application fields of the function evaluation strategy would be complex force evaluations during runtime, table look–up and other processes in which numerical integration is not present.

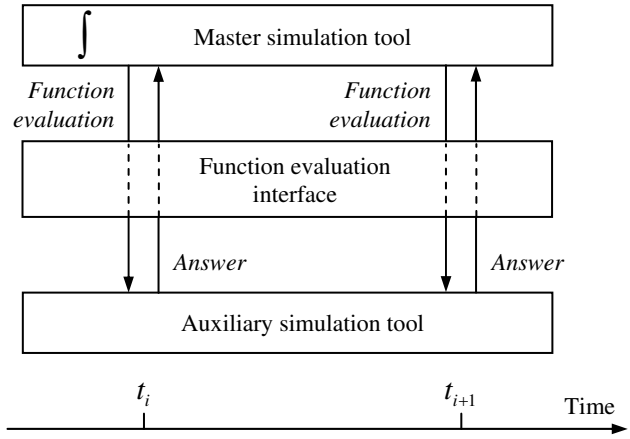

Figure 5.2: *Generic function evaluation configuration*

The implementation of this technique requires the development of an interface between the software tools to allow the main process to use the functionality of the auxiliary software and receive the return data conveniently. Data formats in different tools are often incompatible, so translation routines may be necessary for the correct transmission of information. A simplified depiction of this technique can be seen in Figure 5.2. The block representing the auxiliary software tool at the bottom of the figure can be a standalone process, if direct communication between processes is used, a library or even exported source code, that has been previously compiled together with the source code of the driver program. The availability of these methods is determined by the nature of the external tool, as it may allow or not communication with external processes (for example, via TCP/IP) or the access to inner functions in case of it is compiled as a library.

Finally, a *co–simulation* approach in the strict sense can be developed, in which two simulation tools, each of them with its own states and integrator, share data at defined synchronization points (Arnold, 2008). Again, code export or direct commu-

nication between processes can be used to implement this configuration. In the case of a multibody simulation tool, state–space equations can be represented by

$$\begin{cases} \dot{\mathbf{x}}_m(t) = \mathbf{f}_m(\mathbf{x}_m(t), \mathbf{u}_m(t)) \\ \mathbf{y}_m(t) = \mathbf{g}_m(\mathbf{x}_m(t)) \end{cases} \tag{5.1}$$

where $\mathbf{x}_m$ are the states of the multibody system, $\mathbf{u}_m$ the inputs to the system and $\mathbf{y}_m$ the system outputs. An analogue expression can be used for the equations of the external simulation tool

$$\begin{cases} \dot{\mathbf{x}}_e(t) = \mathbf{f}_e(\mathbf{x}_e(t), \mathbf{u}_e(t)) \\ \mathbf{y}_e(t) = \mathbf{g}_e(\mathbf{x}_e(t)) \end{cases} \tag{5.2}$$

being the inputs of a system the outputs of the other

$$\begin{cases} \mathbf{u}_e(t) = \mathbf{y}_m(t) \\ \mathbf{u}_m(t) = \mathbf{y}_e(t) \end{cases} \tag{5.3}$$

Nowadays, state–of–the–art commercial software performs co–simulation at constant time–steps, with the same integration stepsizes in every subsystem, although research is being carried out to introduce multirate methods in co–simulation environments (Busch et al., 2007). Even with constant and identical time–steps in each subsystem, the evaluation of the inputs for each subsystem, given by Equation (5.3), at synchronization point t_i can be performed in several ways. A frequent strategy is assuming that the inputs of each subsystem can be considered constant during each time–step $[t_i, t_{i+1}]$, which leads to

$$\begin{cases} \mathbf{u}_e(t) = \mathbf{u}_e(t_i) = \mathbf{y}_m(t_i) \\ \mathbf{u}_m(t) = \mathbf{u}_m(t_i) = \mathbf{y}_e(t_i) \end{cases} \tag{5.4}$$

This approach, known as constant extrapolation, has been followed in this Chapter, as the detailed testing of different interpolation degrees and multirate techniques falls beyond its scope, and it will be tackled in Chapter 6. Direct co–simulation, in which co–simulated variables are exchanged once in each integration step, and then each subsystem proceeds its own integration independently, has been used. As it was the case in the function evaluation strategy, co–simulation can be implemented on the basis of intercommunication between processes, or through code export. Again, translation routines between data storage formats will likely be necessary. The synchronization of integrators and the exchange of data can be managed by a co–simulation interface, which can be implemented in one of the communicating software tools. A scheme of this composition is shown in Figure 5.3.

In order to test the described coupling techniques, the MBS software developed in this thesis has been linked to MATLAB/Simulink. MATLAB is a a numerical computing environment that provides state–of–the–art algorithms for a wide range of applications (optimization, control, data acquisition and analysis). MATLAB's add–in

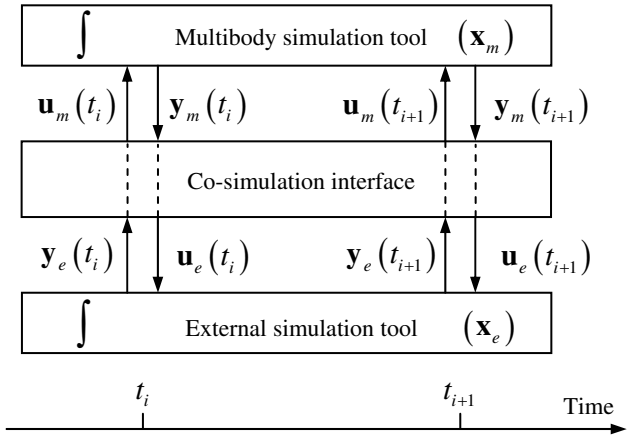

Figure 5.3: *Generic co–simulation configuration*

Simulink can be considered as the de facto standard for model–based design of control systems. This software package includes a library with a wide variety of components and it allows the user to create new elements in a straightforward manner. It is important to note that MATLAB/Simulink code has to be interpreted during runtime, which leads to a considerable increase in simulation time and inefficient execution. This fact rules out the software for demanding applications, for example real–time simulation. Communication between the MBS software and MATLAB/Simulink programs, representing control loops, actuators and other external components, can provide an additional functionality that is missing in the original multibody software.

The techniques described in the following can be applied to other software tools different from MATLAB/Simulink, for example MATRIXx/SystemBuild or the free software Scilab/Scicos. In general, communication between processes can often be achieved if the software supports the use of inter–process communication (IPC), like sockets. The use of external code can be performed through calls to dynamically linked libraries, with their corresponding import libraries and header files, if necessary.

5.3 Function evaluation

A runtime call to MATLAB functions from the multibody software would be desirable in order to evaluate complex force functions or to access look–up tables. Additionally, MATLAB can also be used as a test environment for the definition of new implementations for formulations or models. These could be written in MATLAB's easy–to–use M language, and called from the multibody software as library functions in order to test their correctness before performing their final implementation in an efficient language such as C or Fortran. This would make possible the definition and testing of new models even for users without advanced programming skills.

In this research, three alternative implementation approaches for the function eval-

uation method have been tested: MATLAB Engine, MATLAB Compiler and a MEX API of functions. A dynamic simulation of a double–pendulum has been selected as test example for the above–mentioned implementation approaches: the multibody software carries out the numerical integration and MATLAB is used to evaluate the equations of motion at each time–step. This simple example has been chosen, as there is no practical increase of complexity derived from applying the function evaluation technique to more involved problems.

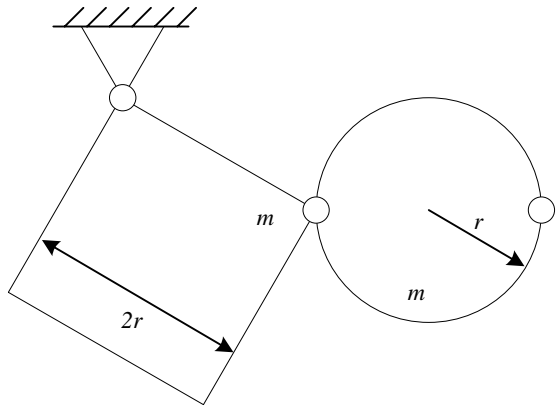

Figure 5.4: *Double pendulum*

The double pendulum is shown in Figure 5.4. The integration of the motion is performed by the MBS software, making use of the augmented Lagrangian formulation of index–3, already described in Sections 3.2.2 and 4.2.1. In this study, the mass (m) and radius (r) parameters have been set to 1 kg and 1 m. The code for the updating of the dynamic terms of the system, including the mass matrix $\mathbf{M}$, the constraints vector $\mathbf{\Phi}$, the Jacobian matrix of the constraints vector $\mathbf{\Phi_q}$ and the generalized forces vector $\mathbf{Q}$, is written in *.m* files and accessed from the MBS simulation software through function evaluation methods. This is equivalent to replace the C++ *model* component in the core module, described in Section 2.3, with an M language counterpart; in this way, the integrators and formulations of the MBS software can be applied to easy–to–code MATLAB models. A similar approach could be taken in order to test dynamic formulations or numerical integrators written in MATLAB, replacing the corresponding component of the core modulus while avoiding the need for translating them to C++. Other application of function evaluation is the invocation of specific MATLAB functionality, such as involved matrix operations.

The methods for implementing the function evaluation described in this Section can be applied to similar numerical software, different from MATLAB, making use of alternative communication facilities. For example, Scilab provides the *intersci* program, which allows calling C and Fortran routines from Scilab, and the calling routines defined in *CallScilab.h*, which make Scilab work as a calculus engine.

5.3.1 MATLAB Engine

The MATLAB Engine library is a set of routines that allows calling MATLAB functionality directly from external C/C++ and Fortran programs. The Engine is a way of intercommunicating running processes such that a MATLAB command window must be open, waiting for receiving the commands sent by the external program and executing them. As the Engine uses its own data structure, *mxArray*, to exchange information with the caller program, several translation functions have to be defined in order to manage the data type and make it compatible with the data types used in the multibody program. Once this problem has been solved, MATLAB functions can be called from the C++ code of the multibody tool. It should be noted that the Engine receives its commands as a string of characters which must be parsed, resulting in the deceleration of the code execution.

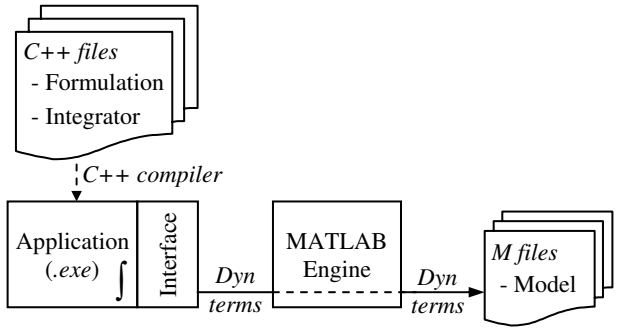

Figure 5.5: *Function evaluation configuration with MATLAB Engine*

The function evaluation configuration through the Engine is represented in Figure 5.5. The MBS software acts as a master tool, integrating the positions of the double pendulum, while the evaluation of dynamic terms is performed, through the Engine, via calls to the *.m* files that code the model.

5.3.2 MATLAB Compiler

Function evaluation has also been achieved through code export, with the use of MATLAB Compiler, transforming *.m* code files into dynamically linked libraries (*.dll*, *.so*). The libraries are then loaded by the multibody software during runtime, thus allowing the invocation of functions. As the Engine does, the Compiler uses its own storage data type, *mwArray*, and translation routines between the MBS code and the compiled MATLAB code must be written. The C/C++ library generated by the Compiler only contains wrappers for the MATLAB routines, and hence it still depends on MATLAB libraries to carry out the computations on runtime.

The use of the Compiler on the *.m* files removes the need for the use of the Engine, as shown in Figure 5.6, replacing the process communication with the export of the pre–compiled code. The evaluation of dynamic terms is directly called from the main application while the library that wraps the routines coded in *.m* files still needs to

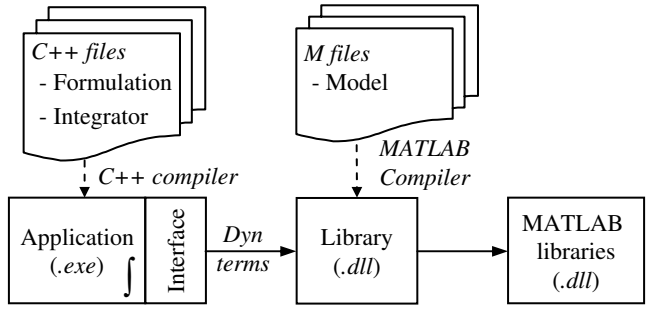

Figure 5.6: *Function evaluation configuration with MATLAB Compiler*

invoke additional MATLAB functions.

5.3.3 MEX functions

A third way of communicating both tools is the definition of an application programming interface (API), which allows calling from MATLAB the functions that are defined and implemented in the multibody package. This way, MATLAB acts as driver tool, starting the integration performed by the MBS software. The API consists of a series of MEX functions that manage the data types defined by MATLAB and make the convenient translation to those types the C++ program uses and vice versa.

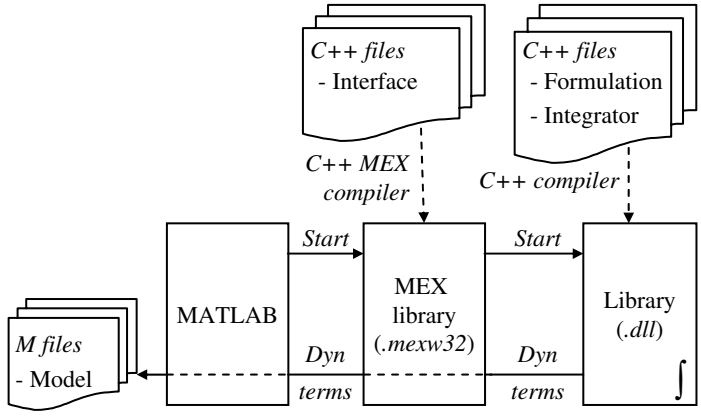

Figure 5.7: *Function evaluation configuration with a MEX API of functions*

Figure 5.7 shows the layout of the function evaluation through the use of a MEX function. Under this configuration, the interface routines are separated from the MBS software and compiled into a library that manages the communication between MATLAB and the MBS software, compiled as a dynamic library. The MBS code calls the model *.m* files for the evaluation of the dynamic terms of the model through this MEX function and this one, in time, through MATLAB.

5.3.4 Results

Two simulations of 10 seconds have been performed using a penalty factor of $\alpha = 10^8$ and constant integration time–steps of 10^{-3} s and 10^{-2} s, respectively. The MBS software is configured to use dense LAPACK routines *gtrf* and *gtrs* as linear solver, which have been proved to be efficient for small–size problems and allow an easy conversion of the storage format from MATLAB.

The elapsed times in calculations, on an AMD Athlon 64 3000+, at 1.81 GHz with 1.00 GB of RAM, are summarized in Table 5.1. As the initial conditions of the motion, the expression of the dynamic terms and the formalism used to integrate the motion are the same in every implementation, output results (positions, velocities and accelerations during the motion) are identical for each time–step, independently of the method used for providing the dynamic terms. The ratios defined in the table refer to the elapsed time of the correspondent function evaluation implementation when compared to standalone C++ MBS code (without the use of MATLAB). The number of iterations is the number of times the iterative solution of the system in Equation (3.4) has been performed. The evaluation of dynamic terms takes place within the Newton–Raphson iteration loop. This, together with the fact that the use of function evaluation methods slows down the execution of the code, makes negligible the amount of computational time elapsed out of the iterative loop. In the standalone C++ implementation, however, the code out of the loop takes around 20% of the time, and this explains the variations that appear in the ratios when using different time–steps.

Table 5.1: *Elapsed times in a 10 s dynamic simulation of the double–pendulum*

Function evaluation	$\Delta t = 10^{-3}$ s		$\Delta t = 10^{-2}$ s	
	Elapsed time (s)	Ratio	Elapsed time (s)	Ratio
Standalone MBS code	$5.02 \cdot 10^{-2}$	1	$8.40 \cdot 10^{-3}$	1
MATLAB Engine	18.12	361.0	3.32	395.2
MATLAB Compiler	5.56	110.8	1.07	127.4
MEX API of functions	0.64	12.7	0.12	14.3
Number of solver iterations	10,000		1,840	

As it was expected, the use of function evaluations in external simulation tools slows down the execution of the program. The MATLAB Engine approach is very easy to implement, but it also delivers very poor efficiency: it has been estimated that the parsing of a single empty function evaluation takes around 0.25 ms. Therefore, the use of MATLAB Engine should be discouraged when function evaluations in the auxiliary tool are repetitive (for example, several times in each integration step).

MATLAB Compiler is usually claimed to be the fastest coupling technique, since it removes the need of parsing string instructions as function calls are performed directly on routines stored in dynamically linked libraries. Even so, the generated C code is still two orders of magnitude slower than standalone C++ MBS code. The overhead of the MATLAB Compiler approach comes from the need of converting data structures

between the MBS software and MATLAB routines. This approach has an additional drawback: if the MATLAB code is modified, it must be compiled and linked again, and this process slows down the code development.

The implementation of the function evaluations as MEX API of functions, shown in Figure 5.7, has yielded the best performance. This approach, nevertheless, requires a high development effort due to the need for building a MATLAB compliant C interface for each function in the multibody package. It is surprising that the implementation of the MBS code as a MEX API leads to an almost 8 times faster execution time when compared to MATLAB Compiler. This may be related to the way in which MATLAB functionality is invoked from the compiled library in the latter case. Another advantage of the MEX API of functions is that the MATLAB code stays in *.m* files, and therefore it allows fast development iterations because it can be modified and tested again without going through a compilation and linking process (as in the case of the previous approach based on MATLAB Compiler).

5.4 Co–simulation

Under the co–simulation approach, the MBS simulation tool has been connected to MATLAB's add–on Simulink, a block diagram simulation tool. With this configuration, two integrators are coupled in the simulation process: the MBS integrator contained in the multibody software and the general purpose integrator in Simulink. In

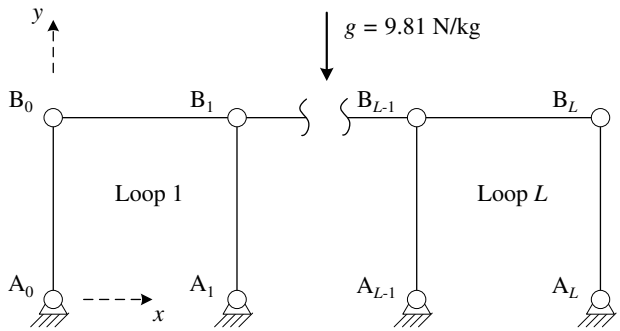

Figure 5.8: *L–loop four–bar linkage*

order to test the co–simulation, a multiphysics model composed of an engine and a mechanical system is simulated. Each subsystem is modelled and integrated in a different environment. The engine model has been obtained from Simulink library of example models and is based on results published by Crossley and Cook (1991). It describes the thermodynamic simulation of a four–cylinder spark ignition internal combustion engine. The multibody system moved by the engine is a planar assembly of four-bar linkages with L loops, composed by thin rods of 1 m length with a uniformly distributed mass of 1 kg, moving under gravity effects. Initially, the system is in the position shown in Figure 5.8 and the velocity of the x–coordinate of point B_0

is +30 m/s. This mechanism has been previously used as a benchmark problem for multibody system dynamics (Anderson and Critchley, 2003; González et al., 2006). It has been selected for this work because it allows testing the effect of variations in the problem size without modifying the structure of the model, just by adding more loops to the mechanism.

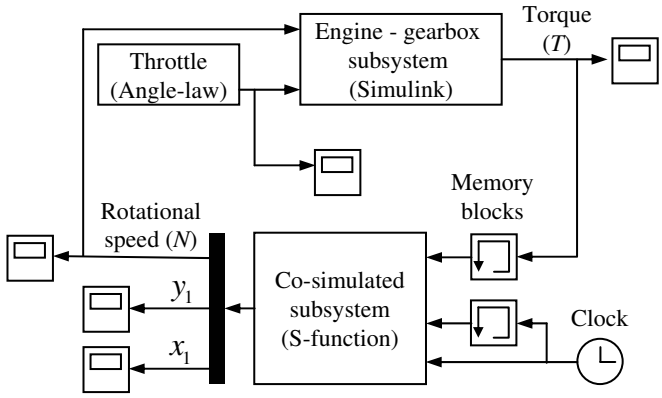

Figure 5.9: *Simplified Simulink model for co–simulation, implemented with an* S–function

The engine provides a motor torque T to the linkage through a gearbox, which is also modelled in Simulink. A constant rotational damping is considered to act on the mechanism, of value 3.18 Ns/rad. Both damping and motor torque are assumed to be applied on the first bar of the mechanism ($A_0 - B_0$). The angular speed of this bar (N) is returned to the engine model as input value, together with the x and y positions of the first point of the linkage (x_1, y_1), for graphical output. The throttle angle of the engine is guided through a pre–defined angle–law. The resulting Simulink model can be seen in Figure 5.9. The use of *memory* blocks is motivated by the need of avoiding algebraic loops; a concise explanation on this particular is provided in Section 6.3.1.

In this research, three implementation approaches for the co–simulation have been tested: network connection, Simulink as master, and MBS software as master.

5.4.1 Network connection

Data exchange between two processes running simultaneously can be carried out using a TCP/IP network connection through standard sockets. To this end, the MBS software is modified in order to make it work as a server socket. Accordingly, a user–defined block (*S–function*) is added to the Simulink model to act as a client socket. In other similar block simulation packages, the role of the *S–function* block can be performed by an equivalent component, such as the *UserCode* block in SystemBuild (National Instruments, 2009) and the *C* or *Fortran* block in Scicos (INRIA, 2009a). Thus, the interface is split into two parts, one in the block diagram environment and one in the MBS software. Both parts of the interface are responsible for the translation of the

storage formats and for the adequate exchange of information at each time–step, so they must be correctly coordinated. Moreover, the communication sequence between both subsystems has to be separately coded in each environment, adding an extra burden to the task of keeping the synchronization of the integrators.

5.4.2 Simulink as master

An alternative to network communication is the code export approach, in which the MBS code can be compiled as a dynamically linked library (*.dll* or *.so*) and directly called from an *S–function* block inside Simulink. In this case, the *S–function* includes all the code of the interface between the MBS code and the Simulink model, and must manage the required exchange of data and format conversions between both environments. In this configuration, Simulink becomes the driver software, starting and ending the simulation and calling the MBS software through the interface block each time a return value is needed by the Simulink integrator.

5.4.3 MBS software as master

Another possibility, following the opposite approach to that used in the previous Subsection, is using the MATLAB product Real–Time Workshop (RTW) (The Mathworks, Inc., 2009) to translate the Simulink model into fast standalone C code, which can be called from the MBS code. In this case, the Simulink model is converted into a dynamically linked library (*.dll*) through the use of RTW functionality; this can be done with little modifications to the Simulink model used in the previous Subsection. With this configuration, the MBS code starts and manages the co–simulation process, invoking the functions compiled in the *.dll* in each integration time–step in order to obtain the value of the torque the engine supplies to the linkage.

This approach has a drawback when compared with the previous techniques: if the Simulink model is modified, it must be translated into C code and compiled again; this is a complex and delicate procedure, which slows down the iterations in code development. The process is detailed in the Appendix of this work.

In both cases (Simulink as master and MBS software as master) two integrators are acting simultaneously and, for this reason, a careful coordination between them is required. Simulink behaviour is, in many aspects, beyond the control of the user, so the co–simulation interface has to be specifically defined to fit Simulink.

5.4.4 Results

The simulation time in the previously described test example is 30 s. A penalty factor of $\alpha = 10^{10}$ and a constant integration time–step of 10^{-3} s have been used. Direct co–simulation with the same integration time–step size in both subsystems has been used. The values of the exchanged variables have been taken as constant from one time–step to the next one (constant interpolation). The MBS software is configured to use KLU (Davis and Natarajan, 2010) routines for solving the linear systems the simulation requires. The co–simulation coupling has been implemented with the three

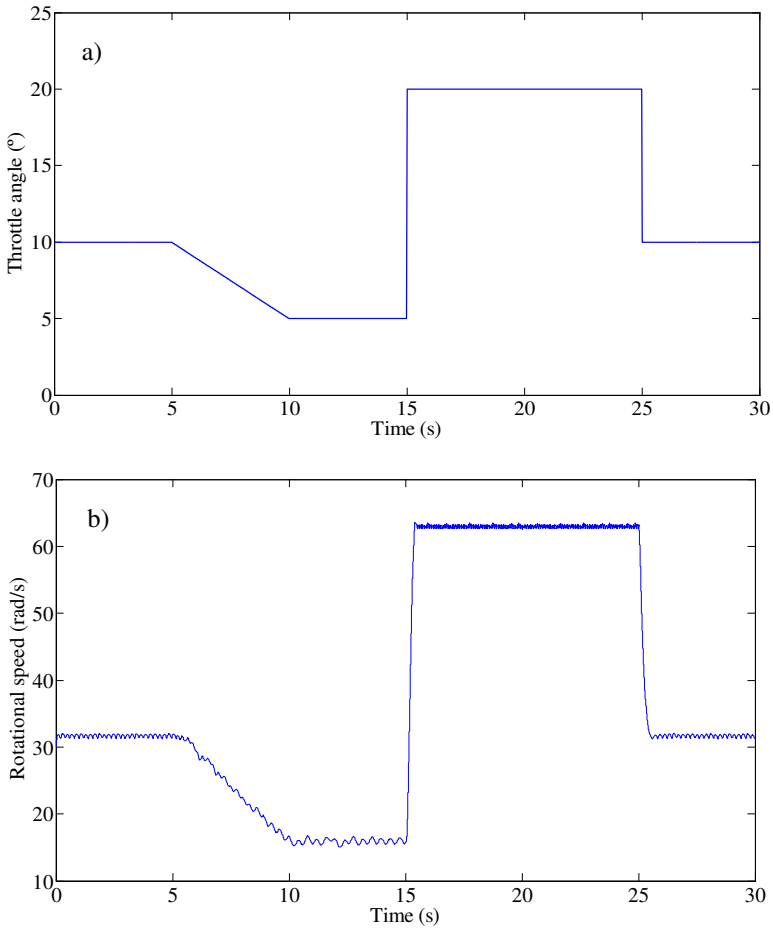

Figure 5.10: *Throttle angle (a) and rotational speed of the mechanism (b) for a 30 s simulation of a 1–loop linkage*

different approaches described above. Results of the simulation can be seen in Figure 5.10, for a 1–loop linkage. The angle-law of the engine throttle is pictured at the top of the figure. The rotational speed of the mechanism, depicted below, shows that the linkage follows the input given by the pedal angle, with the limitations imposed by its rotational inertia and damping. Results do not show significant variations between the three tested coupling techniques. The performance of the described techniques has been tested against a model of the whole system (engine and four–bar linkage) entirely built in Simulink. The four–bar linkage has been modelled with the SimMechanics library (The Mathworks, Inc., 2009), a Simulink add–on for modelling and simulation of rigid multibody systems. In a second stage, the computational efficiency of this model has been further improved via the RTW package, translating the whole model into a standalone C executable. Simulink *ode1* integrator has been used in these simulations, since it is the fastest available integrator and it provides enough precision for

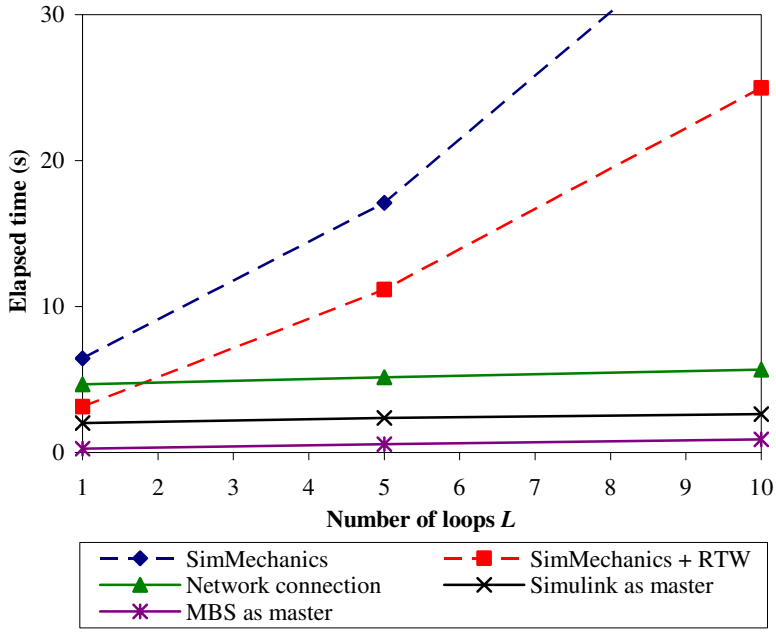

Figure 5.11: *Elapsed times for a 30 s simulation of the L–loop linkage powered by the engine, with different simulation techniques*

the test problem. A comparison of the elapsed times for a 30 s simulation can be seen in Figure 5.11. The monolithic approaches are represented with dashed lines, as they are not properly co–simulation, and labelled as *SimMechanics* for the pure Simulink model, and *SimMechanics + RTW* for the model translated into C code via RTW. The co–simulation methods are designed as *Simulink as master*, for the implementation where Simulink calls MBS code compiled as a library; *Network connection*, when the communication is performed via sockets between simultaneously running processes and *MBS as master*, when the MBS code calls Simulink routines from the *.dll* library compiled with RTW.

Table 5.2: *Elapsed times in a 30 s dynamic simulation of an L–loop four–bar linkage powered by the engine. N stands for the number of variables of the mechanical system*

Co–simulation method	$L = 5$ ($N = 13$)		$L = 10$ ($N = 23$)	
	Elapsed time (s)	Ratio	Elapsed time (s)	Ratio
MBS software as master	0.58	1	0.90	1
Simulink as master	2.37	4.1	2.62	2.9
Network connection	5.15	8.9	5.67	6.3
SimMechanics + RTW	11.17	19.3	24.97	27.7
SimMechanics	17.11	29.5	38.88	43.2

Results are summarized in Table 5.2, where the ratios of elapsed time with respect to the fastest method are also included. They show that the elapsed time for the Simulink model, as expected, grows fast when the number of variables of the problem increases. This is valid even in the case of using a very simple integrator as *ode1*. The use of RTW mitigates this problem and reduces the calculation time between a 30% and a 50%. However, the use of co–simulation techniques leads to even lower computation times, as they permit taking advantage of the highly optimized routines of the MBS code, reducing thus the time needed for calculating the mechanic subsystem of the problem. It can be seen that the *Simulink as master* implementation is somewhat faster than the *Network connection* method, as the overhead derived from socket communications is not present. The *MBS as master* yields the best results, as expected, because the intercommunication takes place, in this case, between an executable and a library both of them coded in an efficient language (C/C++).

It should be noted, however, that the *MBS as master* implementation is significantly more complex than the *Network connection* or *Simulink as master* implementations, and it forces to follow a complex translation process every time the Simulink model is modified, as explained in Section 5.4.3. For these reasons, the *Network connection* or *Simulink as master* co–simulation approaches are better suited for developing and fine–tuning Simulink models, while the *MBS as master* technique is appropriate for production code and real–time applications.

Trends indicate that co–simulation will achieve greater differences with respect to models fully implemented in Simulink as the number of variables of the problem increases. In fact, real–time simulation (less than 30 s of computations) has been achieved with the described co–simulation techniques for multibody models up to 300 variables. This upper limit would allow the efficient real–time simulation of many industrial, non–academic multidisciplinary systems.

5.5 Conclusions

In this Chapter, several implementation methods for coupling MBS simulation software with block diagram simulators and numerical computing environments have been tested. The methods have been tested in a software environment where the C/C++ MBS code developed in this thesis is coupled with MATLAB/Simulink, a quite common setup in the modelling and simulation of complex mechatronic systems. The investigated coupling techniques have been divided into two categories: *function evaluation* and *co–simulation*.

Regarding the implementation methods for function evaluation in MATLAB, the following conclusions can be established:

- The MATLAB Engine approach is the easiest to implement but also the slowest one. The use of MATLAB Compiler reduces the simulation times to a 30% of the time consumed by MATLAB Engine, but at the cost of slowing down the code development iterations. Both approaches are two orders or magnitude slower than standalone MBS code.

- The MEX API of functions is the fastest approach, being only one order of magnitude slower than standalone MBS code. The implementation effort is higher that in other methods, but not overwhelming, and therefore it is recommended as the best approach for function evaluation when simulation efficiency is needed.

Regarding to the implementation methods for co–simulation with Simulink, the following conclusions can be established:

- Co–simulation methods are approximately one order of magnitude faster than simulations based on monolithic models developed in Simulink, even if tools like Real–Time Workshop are used.

- The method labelled *Simulink as master* provides the best trade–off between simulation efficiency and ease of implementation and code development, and therefore it is recommended for developing and fine–tuning models for co–simulation setups.

- The method labelled *MBS software as master* is the fastest approach (several times faster than *Simulink as master*, depending on the relative complexity between the block diagram and multibody models), but its implementation is more complex and requires the translation of Simulink models into C code, a step that slows down the development iterations. Therefore, this method is recommended for production code and real–time applications.

The described coupling techniques can also be implemented with minor changes in other numerical computing environments and block diagram simulators different from MATLAB/Simulink, for example SystemBuild or Scilab/Scicos. However, the efficiency of the different tested methods highly depends on the internal data structures and algorithms of software, and therefore their relative efficiency could be different.

Chapter 6

Multirate Co–simulation Methods

As it was shown in the previous Chapter, dynamic simulation of complex mechatronic systems can be carried out in an efficient and modular way making use of weakly coupled co–simulation setups. When using this approach, multirate methods are often needed to improve the efficiency, since the physical components of the system usually have different frequencies and time scales. However, most multirate methods have been designed for strongly coupled setups, and their application in weakly coupled co–simulation is not straightforward due to the limitations enforced by the commercial simulation tools used in mechatronics design.

This Chapter describes a weakly coupled multirate method intended to be a generic multirate interface between block diagram software and multibody dynamics simulators, arranged in a co–simulation setup. The use of the interface is first demonstrated on a simple, purely mechanical system with known analytical solution and variable frequency ratio (*FR*) of the coupled subsystems. Several synchronization schemes (*fastest–first* and *slowest–first*) and interpolation/extrapolation methods (polynomials of different orders and smoothing) have been implemented and tested. Next, the effect of the interface on the accuracy and the efficiency of the calculations is assessed making use of a co–simulation setting that links an MBS model of a kart to a thermal engine modelled in Simulink.

The use of the multirate interface simplifies the tuning process of the co–simulation parameters, necessary to find values for them which are adequate to the particular properties of the simulated problem. Results show that weakly coupled multirate methods can achieve considerable reductions in the execution times of the simulations without degrading the numerical solution of the problem.

6.1 Introduction

Modern complex mechatronic systems are made up of multi–domain components of different nature. An automobile is a very representative example of these kinds of systems, involving mechanical components (chassis, suspensions, steering mechanism, powertrain), active control devices (Anti–lock Braking System, Electronic Stability Control, traction control), hydraulic devices (brake circuit) and power sources (internal combustion engine or electric motors). Due to the increasing demand of quality and performance, the traditional design approach based on a sequential design of the components can no longer be applied to such systems: engineers need to model and simulate the dynamic response of the whole system, taking into account the simultaneous interaction phenomena between components.

The modelling of complex mechatronic systems can be accomplished via two different strategies: strongly coupled and weakly coupled. On one hand, the strongly coupled strategy assembles the dynamic equations of each subsystem into a monolithic set of equations, which can be numerically integrated in a single environment. On the other hand, the weakly coupled strategy does not assemble the equations: their numerical integration is performed in parallel by several interconnected environments that exchange information during the integration process, working in a co–simulation configuration. Reviews about both strategies are provided by Samin et al. (2007) and Arnold (2008).

The weakly coupled strategy has important advantages over the strongly coupled one: specialized modelling and simulation tools, familiar to experts in the corresponding field, can be applied to each component. In addition, component models can be modified with minor impact on other components, which results in a better modularity of the whole model. For example, control and hydraulic devices are usually modelled and simulated in general–purpose block diagram simulators like Matlab/Simulink (The Mathworks, Inc., 2009), MATRIXx/SystemBuild (National Instruments, 2009) or the free open source tool Scilab/Scicos (INRIA, 2009a,b). Conversely, the behaviour of complex mechanical components is better modelled and simulated in specialized tools for multibody system dynamics like MSC.Adams (MSC.Software Corporation, 2009), Simpack (SIMPACK AG, 2009) or Recurdyn (Function Bay, Inc., 2009); these tools also provide interfaces to the aforementioned block diagram simulators, which simplify the setting of weakly coupled simulations. Representative examples of these kinds of co–simulation setups are given by Liao and Du (2001) and Vaculín et al. (2004), where the authors combine a multibody system simulation package (ADAMS and Simpack, respectively) with a block diagram simulator (Simulink) to model a full vehicle equipped with electronic control devices. Similar setups for the co–simulation of mechatronic systems are described by Mikkola (2001) and Teppo et al. (2001).

One important feature of complex mechatronic systems, derived from their multi–domain nature, is the presence of different time scales, which results in notably different dynamic response characteristics in terms of frequencies. For example, mechanical components have slow frequency responses compared to fast electronic compo-

nents. The computational efficiency of dynamic simulations of complex mechatronic systems is quite important, because these models are often used in optimization processes (where each function evaluation involves a complete dynamic simulation) or hardware–in–the–loop settings (where the dynamic simulation must be run in real–time). In order to make the numerical integration of the dynamic equations of the whole system as efficient as possible, each component should be integrated with a stepsize adapted to its time scale. This procedure is known as multirate integration.

Research on multirate integration methods for ordinary differential equations has been carried out since the late 1970s (Gear and Wells, 1984). The basic idea is to employ two, or more, time–grids: a coarse one for the slow components, and a refined one for the fast components; the coupled terms in the slow and fast equation sets are estimated by means of extrapolation or interpolation methods. Many contributions to this subject have been proposed, including advanced techniques like dynamic partitioning of equations with automatic identification of fast and slow components during the integration (Engstler and Lubich, 1997), self–adjusting multirate time stepping strategies (Savcenco et al., 2007) and stability analysis of the proposed methods (Verhoeven et al., 2007).

The application of existing multirate integration methods to mechatronic models obtained by the strongly coupled strategy is straightforward, since they are precisely designed to work on a monolithic set of equations with full control on the integration process. However, if the mechatronic system is modelled according to the weakly coupled strategy, these multirate integration methods cannot be applied directly due to their particular features:

- They introduce modifications in the integration schemes, something that is not possible in commercial off–the–shelf modelling and simulation tools used for weakly coupled co–simulation. For example, the aforementioned block diagram simulators and multibody system simulation packages offer their own set of integration schemes that cannot be modified.

- They assume that the coarse and refined time–grids are equidistant and synchronized, which means that the large stepsize H is a multiple of the small stepsize h. This condition cannot be guaranteed in weakly coupled co–simulations if one or more subsystems are integrated with a variable time–step integrator, since the stepsize control algorithms of the different commercial simulation environments cannot be synchronized.

- They mitigate the unstable behaviour caused by the explicit extrapolation of some equation terms by introducing implicit schemes, which involve some kind of iterative process. Again, off–the–shelf simulation tools like block diagram simulators do not allow this kind of iteration with other simulation tools.

Due to these impediments, commercial state–of–the–art simulation environments used in mechatronics industry do not provide yet tools to enable multirate integration when they are used in weakly coupled co–simulation setups. Two examples of this situation are provided: the first one is veDYNA (Tesis DYNAware, 2009), a real–time

vehicle dynamics simulation environment very popular in the automotive industry, which is based on Matlab/Simulink. veDYNA works as an external simulation tool embedded in Simulink, and provides a library of mechanical elements to model any kind of automobile. Non–mechanical elements, like electronic and control devices, are modelled in Simulink as usual, exchanging input and output data with the mechanical model. veDYNA uses an internal semi–implicit fixed–step Euler integration scheme to solve the equations of motion of the vehicle, and requires that the Simulink integration be performed with the *ode1* integrator (explicit fixed–step Euler's method) in order to properly synchronize both integrations. This requirement is a strong drawback, since Simulink's *ode1* integration scheme is not suited at all in many situations. Another example of the limitations of currently available simulation environments is SIMAT (SIMPACK AG, 2009), the interface provided by the multibody simulation software Simpack to perform co–simulation with Simulink. SIMAT works as a Simulink block that exchanges data between the Simpack model and the Simulink model during the integration. However, its current implementation only allows fixed stepsize integrators with the same time–step in both simulation environments. The same constraint applies to other packages for multibody system simulation, like ADAMS, which provide interfaces for performing co–simulation with block diagram simulators: none of them supports multirate integration.

Research is being carried out to introduce multirate methods in weakly coupled co–simulation environments, principally in those which combine a general–purpose block diagram simulator with external specialized simulation tools, a common setup in the industry. Busch et al. (2007) have tested several approaches to improve the aforementioned Simpack's SIMAT interface, making it able to support variable stepsizes in both sides of the co–simulation environment; in a similar way, Oberschelp and Vöcking (2004) have investigated the behaviour of some multirate techniques in weakly coupled co–simulations. However, these works apply multirate methods to solve a particular mechatronic model, and therefore their conclusions cannot be generalized nor extrapolated to other cases.

The main goal of this Chapter is to gain insight into the behaviour and performance of multirate methods in weakly coupled co–simulation environments. To achieve this, an interface including an algorithm to implement a general multirate method (i.e. not constrained to synchronized time–grids or to a particular integration scheme), able to couple block diagram simulators with external simulation tools, like multibody simulation packages, has been developed. The proposed algorithm can be configured to work in different modes and to use different interpolation and extrapolation methods. Its use is demonstrated in a very simple example, which clearly shows the need for adjusting the interpolation method and the co–simulation strategy as a function of the nature of the mechanical system. The interface is later applied to a more complex example to evaluate the effect of multirate techniques on the efficiency and accuracy of industrial–like multi–domain simulations.

The remainder of this Chapter is organized as follows: Section 6.2 describes the multirate co–simulation interface created in this research and outlines the coupling strategy it uses. The test of this interface through the use of a simple, purely mechani-

cal example with known analytical solution is detailed in Section 6.3. In these two Sections, several techniques for increasing the accuracy of the simulation are described and the convenience of their use is discussed in the light of the obtained results. In Section 6.4 the interface is used again, this time in the co–simulation of a non–trivial application, in which the multibody model of a kart is coupled to a Simulink block diagram representing a thermal engine. This example has been used to measure the impact of multirate techniques on the time required to compute the simulation and the precision of the results. Finally, in Section 6.5, the conclusions of this Chapter are summarized and some lines for future research are discussed.

6.2 Multirate co–simulation interface

In order to attain the goals of this Chapter, a new multirate interface has been designed and implemented, which allows using a weakly coupled co–simulation scheme that combines a general–purpose block diagram package with a multibody simulation software. This configuration is very common in the design and development of mechatronic systems. Simulink has been selected as block diagram simulator, since it is a well–known tool in this field. However, the building blocks and modelling procedures employed in Simulink are also available in other block diagram simulators like SystemBuild and Scicos, and therefore the co–simulation techniques presented in this section are not particular to Simulink and can be implemented in other tools in a straightforward way.

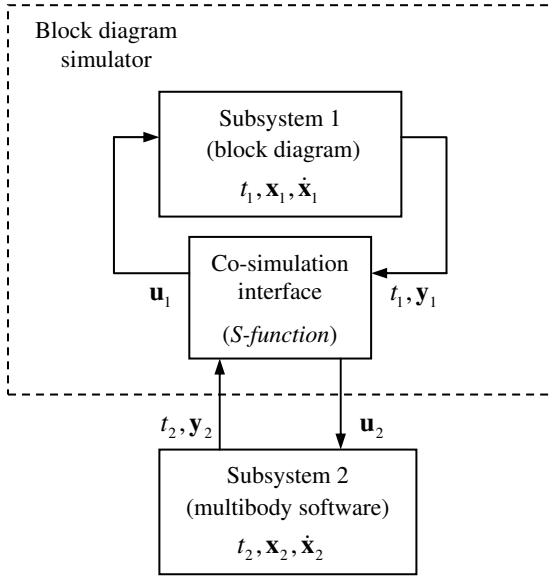

Figure 6.1: *Use of the multirate co–simulation interface*

The generic use of the interface is shown in the block diagram model depicted in Figure 6.1. The dynamics of the subsystem integrated by the block diagram package is modelled in the upper part of the figure. The states and the outputs of this subsystem are represented by $\mathbf{x}_1$ and $\mathbf{y}_1$, respectively, while t_1 stands for the time inside the block diagram software. The multibody software, in the lower part of the figure, tackles the numerical integration of the second subsystem, which has its own states, outputs and time $\mathbf{x}_2$, $\mathbf{y}_2$ and t_2. The time–steps of the subsystems are denoted by h_1 and h_2; as the mechanical components in mechatronic devices are usually slower than the rest of the system (electronic devices and control elements, for example) it will be assumed in the following that the block diagram software manages the fastest subsystem in the model, while the external multibody software integrates the slowest one. This condition is equivalent to state that $h_1 < h_2$. The co–simulation interface is responsible for obtaining the inputs to each subsystem ($\mathbf{u}_1$ and $\mathbf{u}_2$) from the outputs supplied by both programs (which can include, but not necessarily, the states of the subsystem and their derivatives) and synchronizing the different time schemes of the subsystems. This interface is embedded in the block diagram simulator, in a block of type *S–function* in Simulink, *UserCode* in SystemBuild or *C/Fortran* block in Scicos, according to the *Simulink as master* strategy described in Section 5.4.2. The design and behaviour of this block will be described in the following paragraphs.

6.2.1 Coupling strategy for multirate integration

As explained in the Introduction to this Chapter, the simulation environments used in weakly coupled co–simulations implement their own set of integration schemes that cannot be modified. Therefore, the co–simulation interface must implement a coupling scheme that enables a multirate integration of different subsystems independently of the integration schemes and time–steps that apply to each of them. In the proposed coupling scheme, the block diagram simulator (Simulink, in this case) plays the role of *master* integrator, since it is responsible for starting and stopping the numerical simulation. On the other hand, the external simulator acts as *slave* integrator, working on request.

Without loss of generality, it will be assumed that the block diagram simulator uses the well–known fourth–order Runge–Kutta formula, which is known as *ode4* in Simulink:

$$
\begin{aligned}
\mathbf{x}_1^{i+1} &= \mathbf{x}_1^i + h_1 \sum_{j=1}^{4} b_j \mathbf{K}_j \\
\mathbf{K}_1 &= \mathbf{f}\left(t_1^i,\ \mathbf{x}_1^i\right) \\
\mathbf{K}_2 &= \mathbf{f}\left(t_1^i + h_1/2,\ \mathbf{x}_1^i + h_1\mathbf{K}_1/2\right) \\
\mathbf{K}_3 &= \mathbf{f}\left(t_1^i + h_1/2,\ \mathbf{x}_1^i + h_1\mathbf{K}_2/2\right) \\
\mathbf{K}_4 &= \mathbf{f}\left(t_1^i + h_1,\ \mathbf{x}_1^i + h_1\mathbf{K}_3\right)
\end{aligned}
\tag{6.1}
$$

In order to advance a time–step from t_1^i to t_1^{i+1}, the block diagram simulator needs to evaluate all blocks in the model four times, one for each term $\mathbf{K}_j$. The first evaluation is performed at $(t_1^i, \mathbf{x}_1^i)$, using the states ($\mathbf{x}_1$ in this case) computed in the previous time–step. In block diagram terminology, this evaluation is known as *major time–step*, while the next evaluations (corresponding to $\mathbf{K}_2$, $\mathbf{K}_3$ and $\mathbf{K}_4$) are known as *minor time–steps*.

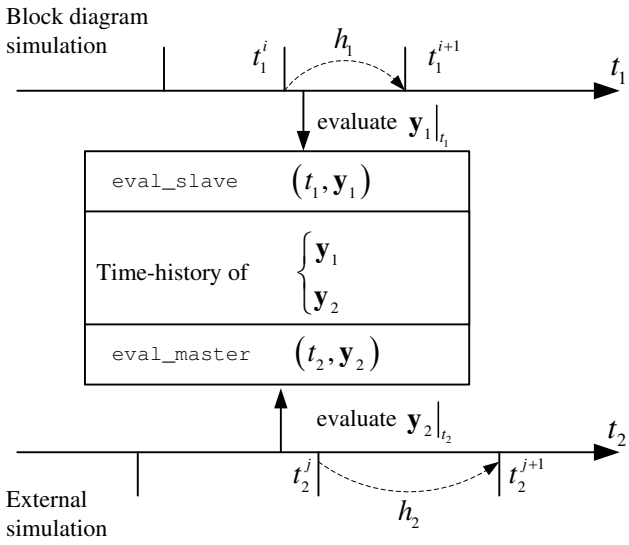

Figure 6.2: *Working diagram for the* co–simulation interface *block*

The *co–simulation interface* block in Figure 6.1 manages the evaluation of the dynamic response of the second subsystem at the times required by the block diagram simulator. It contains a set of functions and data structures responsible for synchronizing the numerical integrations in the block diagram software and the external simulator. The structure and behaviour of this block are represented in Figure 6.2. When the *co–simulation interface* block is evaluated at a given time, it calls its *eval_slave* function in order to get the inputs it needs. The algorithm of this function is represented in pseudo–code in Table 6.1 and will be described in the next paragraphs.

- In step 1, if the evaluation is performed in a *major time–step* (block diagram simulators provide routines to determine this condition), the input time t_1 and outputs $\mathbf{y}_1$ in the block diagram are appended to a dataset that holds the time–history of these values. As it has been mentioned above, these outputs may include or not the states of the block diagram and their derivatives. Data at *minor time–step* evaluations are not stored because they do not correspond to integration points in the timeline.

- Step 2 determines whether the external simulator should move ahead in the numerical integration of its subsystem. Two criteria are available to take this

Table 6.1: eval_slave *function algorithm, in pseudo–code*

1) if t_1^i is a *major time–step*
store $t_1^i, \mathbf{y}_1^i$
$n = 0$
2a): if (*slowest–first*) then
while $\left(t_2^{j+n} < t_1^i\right)$
advance integration step in external simulator
store results $\left(t_2^{j+n}, \mathbf{y}_2^{j+n}\right)$; $n = n + 1$
end
2b): if (*fastest–first*) then
while $\left(t_2^{j+n} + h_2 < t_1^i\right)$
advance integration step in external simulator
store results $\left(t_2^{j+n}, \mathbf{y}_2^{j+n}\right)$; $n = n + 1$
end
3) Interpolate or extrapolate $\mathbf{u}_1$ at t_1^i

decision (steps 2a and 2b), depending on the selected synchronization scheme: *slowest–first* and *fastest–first* (Gear and Wells, 1984). In the *slowest–first*, represented in step 2a, the numerical integration of the slowest subsystem is always ahead of the fastest one. Therefore, when the *co–simulation interface* block is evaluated at $t_1 > t_2$, it calls the external simulator to move ahead in its numerical integration a certain number of time–steps (represented by counter variable n) until $t_1 < t_2$. After each time–step of the external simulator, the time and the outputs of the slow subsystem, t_2 and $\mathbf{y}_2$, are appended to a dataset that holds the time–history of these values. In this process, the integration scheme of the external simulator will need the values of its inputs $\mathbf{u}_2$ at particular instants; these values are interpolated or extrapolated from the time–history of outputs of the fast subsystem $\mathbf{y}_1$ at *major time–steps* (stored in step 1) by the *eval_master* function. The *fastest–first* scheme represented in step 2b is very similar, but the numerical integration of the slowest subsystem is always one time–step behind the fastest subsystem.

- Finally, in step 3, the values of the inputs to the fast subsystem $\mathbf{u}_1$, at time t_1, requested by the block diagram simulator are interpolated or extrapolated from the time–history of the outputs of the slow subsystem $\mathbf{y}_2$, stored in step 2.

The interpolation or extrapolation of states in the *eval_slave* and *eval_master* functions is performed using order P polynomials. The user can select the value of P from 0 to 4. The polynomials are built with $P + 1$ time–steps t^P, ..., t^0, selected as follows: t^P is the time–step closest to the evaluation time t that satisfies $t^P > t$ (if there is

any time–step ahead of t), and t^{P-1}, ..., t^0 are the previous time–steps stored in the time–history.

The functions and data structures of the *co–simulation interface* have been implemented as a C/C++ library, independent of the external simulator and the number of exchanged variables. The external simulator only needs to provide two functions: a function to move ahead a time–step in the numerical integration and to return the resulting time and outputs, and a user routine to connect the *eval_master* function. Most dynamic simulation tools can satisfy these requirements.

6.2.2 Smoothing techniques

For models with very different time scales in their subsystems, interpolation and extrapolation techniques may fail to give correct results in weak coupled multirate co–simulation. Oberschelp and Vöcking (2004) described a smoothing technique to overcome this problem; a similar strategy has been tested in this work. Smoothing is expected to improve the global precision of the simulation, avoiding the need of raising the number of integration time–steps per cycle, or using higher order integrators, which would noticeably increase the elapsed time in computations.

When using smoothing in this work, the interpolation or extrapolation strategies described above are replaced by an averaging (arithmetic mean) of the values of the fast subsystem during the last time–step of the slow one. This averaging is performed on the basis of a *fastest–first* method, with the integration of the fast subsystem being performed in advance with respect to the slow one. When the slow subsystem needs to evaluate its states at time t_2^n, it requests the necessary inputs $\mathbf{u}_2^n$ through a call to the *eval_master* function. The value of these inputs is determined by averaging the buffered values of the outputs of the fast subsystem $\mathbf{y}_1$ in the time–history from time t_2^{n-1} to t_2^n. The averaged value is returned by the *eval_master* function, and considered constant during the integration of the whole time–step of the slow subsystem.

It should be noted that the use of extrapolation techniques is still required during the calls to the *eval_slave* function, for the computation of the inputs of the fast subsystem at the times required by the block diagram simulator.

6.3 Test problem

A test problem involving two subsystems with fast and slow dynamic responses will be solved by coupling a block diagram model in Simulink (to integrate the fast subsystem, 1) with an external multibody model (to integrate the slow subsystem, 2) through the multirate interface already introduced. The parameters of the test problem will be adjusted to generate a range of co–simulation situations, which will be used to test different coupling strategies in terms of precision.

The double–mass triple–spring system shown in Figure 6.3 has been selected as test problem. It is made up of two subsystems represented by masses m_1 and m_2, which are coupled by the spring k_2. This simple, two degree–of–freedom system presents the advantage of having a known analytical solution for its dynamic response,

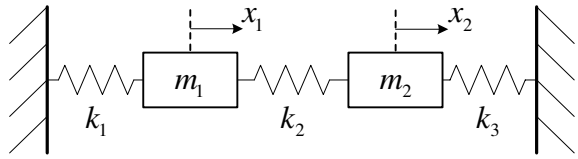

Figure 6.3: *Test problem*

which can be used as a reference in order to measure the accuracy of the coupled multirate numerical integration carried out by any co–simulation scheme. The dynamics of the test problem is governed by Equation (6.2):

$$\begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix} \begin{Bmatrix} \ddot{x_1} \\ \ddot{x_2} \end{Bmatrix} + \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 + k_3 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} \tag{6.2}$$

where x_1 and x_2 measure the horizontal displacement of the masses from their equilibrium position. Equation (6.2) is a simple second order differential equation whose analytical solution is given by

$$\begin{aligned} x_1(t) &= C_{11} \cdot \cos(\omega_1 t) + C_{12} \cdot \sin(\omega_1 t) + C_{13} \cdot \cos(\omega_2 t) + C_{14} \cdot \sin(\omega_2 t) \\ x_2(t) &= C_{21} \cdot \cos(\omega_1 t) + C_{22} \cdot \sin(\omega_1 t) + C_{23} \cdot \cos(\omega_2 t) + C_{24} \cdot \sin(\omega_2 t) \end{aligned} \tag{6.3}$$

where ω_1 and ω_2 are the natural frequencies of the two vibration modes of the system, and the terms C_{ij} are constants that define the amplitude of the vibration. In order to simplify the problem, sinus terms in Equation (6.3) are removed by setting the initial velocities to zero:

$$\begin{aligned} x_1(t) &= C_{11} \cdot \cos(\omega_1 t) + C_{13} \cdot \cos(\omega_2 t) \\ x_2(t) &= C_{21} \cdot \cos(\omega_1 t) + C_{23} \cdot \cos(\omega_2 t) \end{aligned} \tag{6.4}$$

The dynamic response shown in Equation (6.4) is a function of six independent parameters (ω_1, ω_2, C_{11}, C_{13}, C_{21}, and C_{23}). For the purposes of this study, two of them are set to the values in Equation (6.5),

$$\begin{aligned} \omega_1 &= 1 \text{ Hz} \\ C_{11} &= 1 \text{ m} \end{aligned} \tag{6.5}$$

and the rest are presented in a more suitable form making use of the ratios defined in Equation (6.6):

$$\begin{aligned} FR &= \omega_1/\omega_2 \\ AR_{12} &= C_{11}/C_{23} \\ AR_1 &= C_{11}/C_{13} \\ AR_2 &= C_{23}/C_{21} \end{aligned} \tag{6.6}$$

From here on, frequencies ω_1 and ω_2 will be identified respectively with the primary frequencies of masses m_1 (fast subsystem) and m_2 (slow subsystem), assuming $\omega_1 > \omega_2$, $|C_{13}| > |C_{11}|$ and $|C_{23}| > |C_{21}|$. The ratios defined in Equation (6.6) are interpreted as follows:

- The *frequency ratio FR* measures how fast the fast subsystem m_1 is, compared with the slow subsystem m_2.
- The *amplitude ratio* AR_{12} compares the primary amplitudes of both subsystems (C_{11} for m_1 and C_{23} for m_2).
- The *amplitude ratios* AR_1 and AR_2 measure how much the dynamic response of each subsystem is affected by the other subsystem.

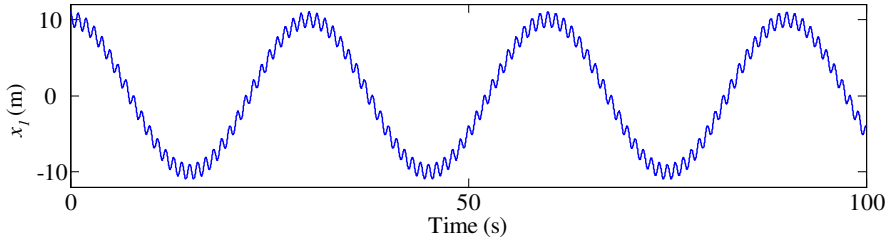

Figure 6.4: *Dynamic response of* x_1

Numerical experiments performed in Section 6.3.2 will use different sets of values for the ratios defined in Equation (6.6), in order to reproduce diverse co–simulation situations. As example, Figure 6.4 shows the dynamic response of x_1 for $FR = 30$, $AR_{12} = 0.1$, $AR_1 = 0.1$ and $AR_2 = -1000$.

After solving the dynamics of the problem in analytical form, the final step is finding the physical parameters that define the system (m_1, m_2, k_1, k_2, k_3) and the initial conditions x_1 ($t = 0$) and x_2 ($t = 0$) as a function of the response parameters defined in Equations (6.5) and (6.6). The resulting expressions will allow adjusting the physical parameters of the test problem in order to generate any desired dynamic response in its two subsystems. Note that the five aforementioned physical parameters can be scaled by the same factor without changing the dynamic response of the system, and therefore one of them must be fixed in advance. The selection of k_2 as fixed parameter greatly simplifies the mathematical manipulations:

$$k_2 = 1 \text{ N/m} \tag{6.7}$$

and the remaining physical parameters can be obtained from the eigenvalue equation:

$$\left(\mathbf{K} - \omega^2 \mathbf{M}\right) \mathbf{A} = \mathbf{0} \tag{6.8}$$

where $\mathbf{A}$ is the matrix of modal amplitudes of the system, $\mathbf{K}$ and $\mathbf{M}$ are the stiffness and mass matrices shown in Equation (6.2) and ω stands for the natural frequencies

of the system. The characteristic polynomial of the eigenvalue equation leads to a biquadratic equation in ω:

$$\begin{aligned}&\omega^4 m_1 m_2 - \omega^2 \left(m_2 \left(k_1 + k_2\right) + m_1 \left(k_2 + k_3\right)\right) + \\ &+ \left(k_1 + k_2\right)\left(k_2 + k_3\right) - k_2^2 = 0\end{aligned} \tag{6.9}$$

which can be analytically solved, giving two equations of the form

$$\omega = f\left(m_1, m_2, k_1, k_2, k_3\right) \tag{6.10}$$

Two more equations can be obtained by substituting the solution given in Equation (6.4) in the equations of motion given by Equation (6.2); as each mode of vibration must satisfy the equations of motion, they lead to:

$$\begin{aligned}\frac{C_{11}}{C_{21}} &= \frac{k_2}{k_1 + k_2 - \omega_1^2 m_1} = \frac{k_2 + k_3 - \omega_1^2 m_2}{k_2} \\ \frac{C_{13}}{C_{23}} &= \frac{k_2}{k_1 + k_2 - \omega_2^2 m_1} = \frac{k_2 + k_3 - \omega_2^2 m_2}{k_2}\end{aligned} \tag{6.11}$$

In this point, the intermediate parameters a and b are defined to simplify the expression of the following equations:

$$\begin{aligned}a &= C_{11}/C_{21} \\ b &= C_{13}/C_{23}\end{aligned} \tag{6.12}$$

so the solution of the set of four equations formed by Equations (6.10) and (6.11) can be expressed in the following way:

$$m_1 = \frac{a - b}{ab\left(\omega_1^2 - \omega_2^2\right)} k_2 \tag{6.13}$$

$$m_2 = \frac{ab\left(b - a\right)}{ab\left(\omega_1^2 - \omega_2^2\right)} k_2 \tag{6.14}$$

$$k_1 = \frac{a\left(1 - b\right)\omega_1^2 + b\left(1 - a\right)\omega_2^2}{ab\left(\omega_1^2 - \omega_2^2\right)} k_2 \tag{6.15}$$

$$k_3 = \frac{ab\left(\left(b - 1\right)\omega_1^2 - \left(1 - a\right)\omega_2^2\right)}{ab\left(\omega_1^2 - \omega_2^2\right)} k_2 \tag{6.16}$$

Finally, the initial positions of the masses can be easily obtained from the solution of Equations (6.4) at time $t = 0$, substituting in them the values of the parameters

defined in Equations (6.5) and (6.6):

$$\begin{aligned} x_1(0) &= C_{11} + C_{13} = C_{11} \cdot \frac{1 + C_{11}/C_{13}}{C_{11}/C_{13}} = \frac{1 + AR_1}{AR_1} \\ x_2(0) &= C_{21} + C_{23} = C_{11} \cdot \frac{1 + C_{23}/C_{21}}{(C_{11}/C_{23})\,(C_{23}/C_{21})} = \frac{1 + AR_2}{AR_{12} \cdot AR_2} \end{aligned} \tag{6.17}$$

Equations (6.13) to (6.17) provide the values of the physical parameters and initial conditions that generate the desired dynamic response of the test problem, described by the parameters in Equation (6.6). The range of validity of these expressions is limited by the fact that the physical parameters (m_1, m_2, k_1, k_2, k_3) must be positive. This constraint restricts the values of the parameters defined in Equation (6.6) within the following limits:

$$|AR_{12}| > \frac{FR^2 - 1}{\left|\frac{FR^2}{AR_1} - AR_2\right|} \tag{6.18}$$

$$|AR_{12}| < \left|\frac{FR^2 - \frac{1}{AR_1 \cdot AR_2}}{FR^2 - 1} \cdot AR_1\right| \tag{6.19}$$

$$\begin{aligned} FR < 1 \Rightarrow & \begin{cases} AR_1 \cdot AR_2 < 0 \\ AR_{12}/AR_1 < 0 \\ AR_{12} \cdot AR_2 > 0 \end{cases} \\ FR > 1 \Rightarrow & \begin{cases} AR_1 \cdot AR_2 < 0 \\ AR_{12}/AR_1 > 0 \\ AR_{12} \cdot AR_2 < 0 \end{cases} \end{aligned} \tag{6.20}$$

The computing model using for the co–simulation of this test problem is shown in Figure 6.5. In this Simulink model, the acceleration of the fast subsystem goes through a double integration to obtain its position. This process is performed by Simulink integrator blocks. The dynamics of the slow subsystem (m_2) is evaluated in the external multibody simulation package and the communication is managed by the co–simulation interface as described in Section 6.2.1.

6.3.1 Algebraic loops

Block diagram simulators allow creating algebraic loops in the model by connecting the output of a block to its input via direct feedthrough blocks (i.e. no differentiation or integration blocks). Algebraic loops are a convenient way to model certain problems, but they require an iterative solution at each time–step in the numerical integration. As a result, they drastically increase simulation times, which is usually unacceptable for weakly coupled co–simulation of mechatronic systems. Several techniques can be

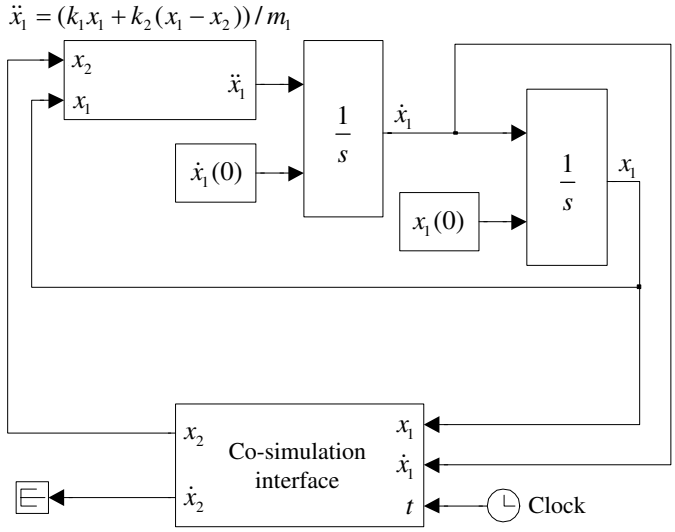

Figure 6.5: *Simulink model of the test problem*

used to avoid algebraic loops: *delay* and *memory* blocks, which delay the value of a variable one time–step, are examples. It is very convenient to test the proposed multirate method with this modelling technique, since it is often present in block diagram simulations.

In the model shown in Figure 6.5, spring forces acting on m_2 are evaluated inside the external simulator. When these forces are transferred to the block diagram simulator, an algebraic loop appears, as shown in Figure 6.6: the input force F to the *co–simulation interface* block is connected to its output x_2 through the direct feedthrough block *Springs*. The algebraic loop is broken by placing *memory* blocks in the force and time signals before entering the *co–simulation interface* block. This model will also be used to test the proposed multirate method.

6.3.2 Numerical experiments and error measurement

Preliminary investigations confirmed that the behaviour of the multirate simulation of the test problem is mostly affected by the frequency ratio *FR* while the other ratios defined in Equation (6.6) do not have a significant impact. Therefore, the test problem described in Section 6.3 has been adjusted with $AR_1 = 0.1$, $AR_2 = -1000$ and $AR_{12} = 0.1$; see Figure 6.4 for an example of the dynamic response of x_1. A sweep of frequency ratios *FR* is performed in order to evaluate how this parameter affects the co–simulation process.

In the block diagram simulator (Simulink), the *ode4* integrator is used, while the multibody simulator uses the trapezoidal rule. Stepsizes h_1 and h_2 have been adjusted to perform 100 time–steps per cycle in each simulator. These time–steps are small enough to keep integration errors very low in both subsystems, and therefore the er-

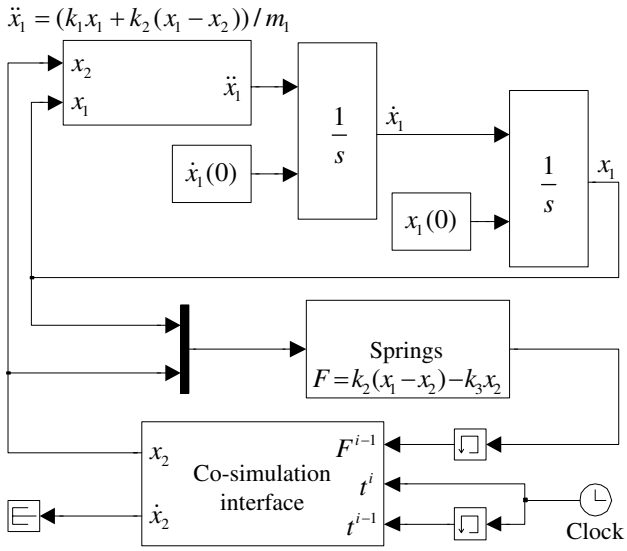

Figure 6.6: *Simulink model with memory blocks to break algebraic loops*

ror in the numerical solution will be mainly caused by the multirate co–simulation scheme. Each numerical experiment consists on a simulation of 100 cycles of the fastest frequency ω_1, which corresponds to $100/FR$ cycles of the slowest frequency ω_2.

The dynamic response obtained from the weakly coupled co–simulation is compared with the analytical solution of the motion given in Equation (6.4). The error in the numerical simulation is measured in two ways: position error and energy error. Position error is given by Equation (6.21):

$$\Delta x = \frac{FR}{N}\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{x_i - x_i^{exact}}{x_{rms}}\right)^2} \tag{6.21}$$

where x_i is the value of the position at time t_i obtained in the numerical simulation, x_i^{exact} is the position at the same time obtained from the analytical solution in Equation (6.4), and n is the number of points of time in the time–history of the solution ($n = 10,000$). To obtain a relative error, the absolute error in position is divided by the quadratic mean in the simulation (x_{rms}) instead of x_i^{exact} to avoid singularities when the analytical solution takes values close to zero. $N = 100$ is the number of simulated cycles of the fast subsystem, and the factor FR/N is introduced to correct the accumulation of errors when a high number of cycles of the slow subsystem is present. In this way, errors obtained from Equation (6.21) are comparable through numerical experiments with different FR ratios. If the test problem is fully modelled and solved in Simulink (without co–simulation) with the *ode4* integrator and the smallest stepsize, the position error given by Equation (6.21) is in the order of 10^{-8}, which

corresponds to an almost exact solution. Position errors below 10% still correspond to a good numerical solution, very similar to the analytical solution at first glance.

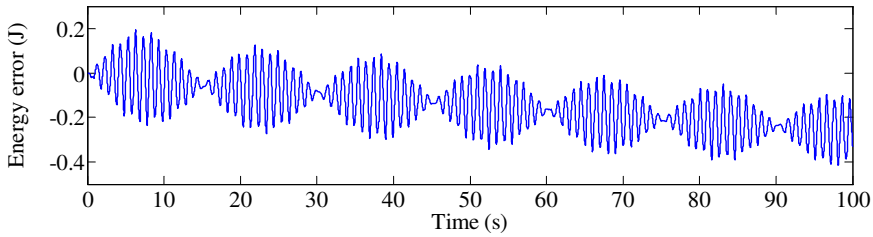

Figure 6.7: *Time–history of the energy error in the numeric simulation (FR = 30, cubic interpolation)*

However, Equation (6.21) gives high position errors when the numerical solution presents a small delay compared to the analytical solution, even when the phase difference is very small and the numerical solution can be still considered good. Therefore, this position error can mislead about the precision in certain situations. To overcome this limitation, an additional measurement of the energy error can be used, as the system is fully conservative. Thus, the energy error is defined as

$$\Delta E = \frac{FR}{N}\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{E_i - E_0}{E_0}\right)^2} \tag{6.22}$$

being E_0 the initial value of the energy of the system (that should be constant during the simulation), and E_i the energy at time t_i obtained in the numerical simulation. The oscillations that have been observed in the energy history of the system (see Figure 6.7) justify the use of a norm–2 error instead of a simple comparison between the initial and final energy levels of the system.

It has been observed that some numerical simulations lead to low energy errors despite the position time–history is obviously incorrect: the numerical integration conserves the system energy but gives a wrong solution after a few cycles. Therefore, both errors (position and energy) should be considered to determine the precision and correctness of the obtained numerical solutions.

6.3.3 Results and discussion

Both *fastest–first* and *slowest–first* schemes have been tested. In the following, they will be referred to as *FF* and *SF*, respectively. In addition, the interpolation orders used in *eval_master* and *eval_slave* functions can be different and one of the following: zero (constant value, designed as *O*0), linear (*O*1), quadratic (*O*2), cubic (*O*3) and fourth order (*O*4). The position error for x_1 and the energy error, defined in Equations (6.21) and (6.22), have been measured for each interpolation method for a span of *FR* ranging from 1.5 to 100. Results can be seen in Figure 6.8

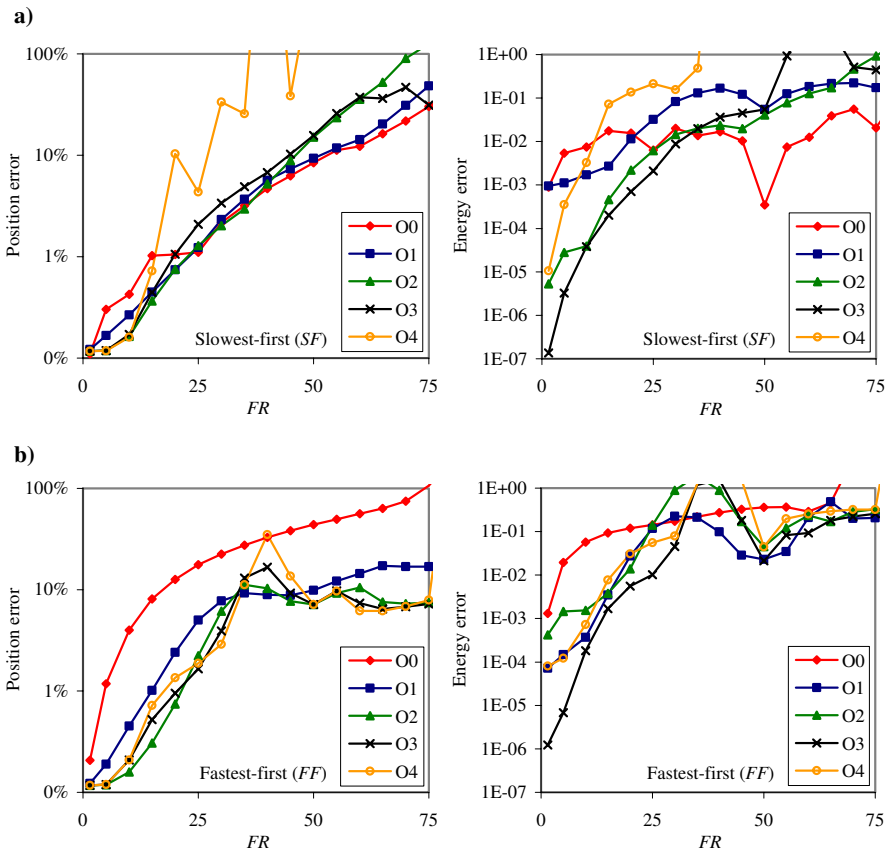

Figure 6.8: *Position error in x_1 (left) and energy error (right) for different interpolation polynomial orders as a function of FR, for* slowest–first *(a) and* fastest–first *(b) schemes*

The first conclusion that can be drawn from the performed simulations is that it is not possible to find an optimal general purpose co–simulation method, even for such a simple test problem as the one described in Section 6.3.

For $FR \leq 25$, *slowest–first* (*SF*) integration combined with cubic interpolation ($O3$) shows the best performance, attaining good position and energy error levels. The use of higher order interpolation polynomials suffers from instabilities, which results in the losing of the reference solution, and therefore has not helped the reduction of errors. *Fastest–first* (*FF*) techniques, on the other hand, attain very low error levels in the integration of the position of x_2, as it was expected, because the integration of the slow subsystem is performed on the basis of already evaluated values of x_1; however, this improvement is made at the cost of worsening the energy levels and the shape of the time–history of x_1.

For $25 \leq FR \leq 50$, *SF* integration without interpolation ($O0$) seems to be the most suitable strategy. The use of *FF* strategies in this range of frequency ratios leads to a

numerical instability that translates into the amplification of the oscillations in x_1, and can be visualized in Figure 6.8 as a peak in the error graphics around $FR = 40$.

For $FR > 50$, the position errors with *SF* strategies are always over 10% and they follow an upwards trend; among them, the use of no interpolation $O0$ gives the best results in position and energy. On the other hand, *FF* techniques seem to stabilize the position error in this region under 10% with reasonable levels of energy errors, at least with $O2$ and higher interpolation orders. However, the analysis of the position history shows that this is a consequence of the attenuation of the fast oscillations of the first subsystem, m_1. In fact, when *FR* grows to values of 80 and higher, the inverse effect takes place and the oscillations are amplified, leading to great errors in position and energy. In both cases, amplification and attenuation, the results cannot be considered valid, even when low error levels in both position and energy are attained.

Two consequences can be inferred from the exposed:

- The errors defined in Equations (6.21) and (6.22), and used as indicators of the correctness of the solution, are not enough for determining the suitability of a co–simulation method for solving every particular problem.

- The use of *FF* strategies can lead to the rising of numerical instabilities, resulting in amplified oscillations in the solution of the problem or, on the contrary, in the filtering of small oscillations, with the loss of the contribution of the fast frequency ω_1 to the solution.

For values of $FR \geq 90$, even *SF* with $O0$ configuration is affected by a sudden growth of the errors and every interpolation order fails completely to follow the analytic reference solution.

The use of smoothing techniques can contribute to the reduction of the error for relatively high values of *FR*, increasing the ability of the simulation to track the reference solution. In order to attain acceptable results, the polynomial fitting interpolation methods for the evaluation of the states of the slow subsystem can be substituted with least squares approximations. This can help to *filter* the variations in velocities that arise when the difference between the time–steps grows.

A comparison of the co–simulation results for $FR = 90$ with and without smoothing can be seen in Figure 6.9. The co–simulated output for variable x_1 is compared to the analytical solution of the motion (thin continuous line). In the upper image no interpolation ($O0$) has been used; in the central graphic, $O3$ interpolation has been used in *eval_master* and *eval_slave* functions, together with *FF* strategy. The lower image shows the better accuracy obtained using the smoothing technique with $O3$ interpolation in *eval_slave* function. However, it must be noted that smoothing is subject to the same filtering or amplifying problems that *fastest–first* implementations suffer from. As a consequence, smoothing has only shown an acceptable performance for certain combinations of *FR* and the interpolation (or approximation) order used for the slow subsystem.

Regarding the equivalent model with an algebraic loop, depicted in Figure 6.6, the obtained results have been practically equivalent to those of the original model

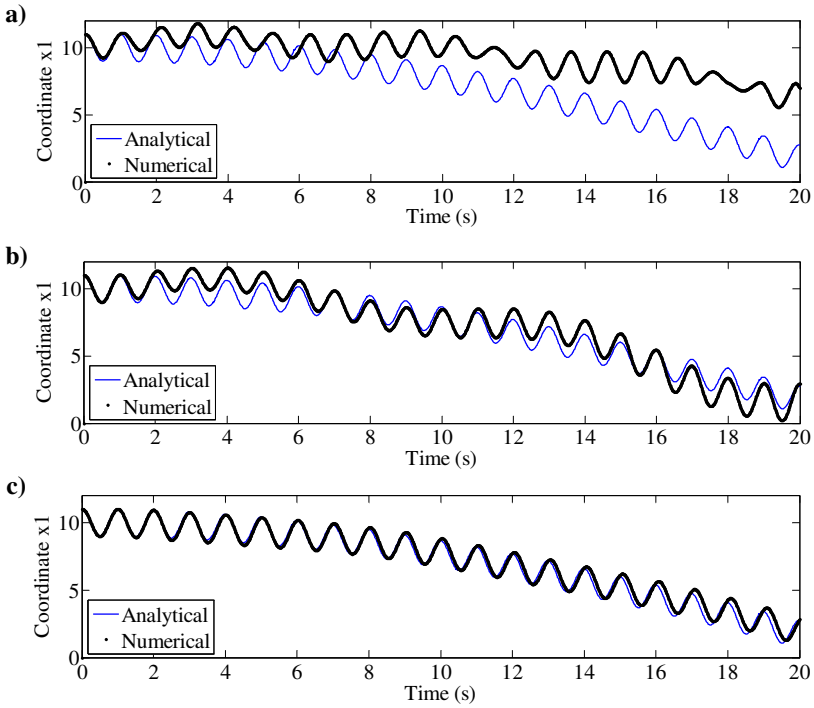

Figure 6.9: *Response to 20 simulation cycles of the fast subsystem for FR* $= 90$*: (a)* slowest–first *with* O0*, (b)* fastest–first *with* O3*, (c) smoothing with* O3

of Figure 6.5. The use of *memory* blocks has yielded a better performance than the equivalent model with *delay* blocks.

In most simulations, it has been observed that the accumulated error grows as the simulation time increases. This is not expected to happen in complex multiphysics systems for two reasons. First, real systems use to have dissipative elements like dampers that soften the effect of small vibrations. In the second place, most co–simulated systems include control elements, oriented to reference tracking, which make the whole system less sensitive to error accumulation.

6.4 Application to a multiphysics problem

The multirate interface and co–simulation methods described in the previous Sections have been applied to the solution of the dynamics of a vehicle, in this case a kart. This multiphysics model is divided into two subsystems: a multibody model of the mechanical components of the vehicle, including the steering column, tyres and suspensions, and a thermodynamic model of a four–cylinder spark ignition engine.

The model of the mechanical components of the kart can be seen in Figure 6.10; the figure represents only half of the model, the actual one includes the suspension of the four wheels and the whole chassis. The number of variables of the multibody

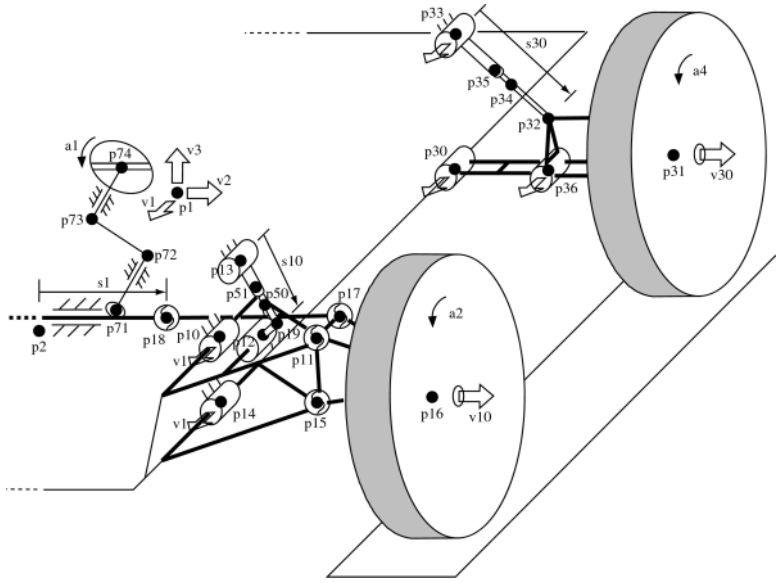

Figure 6.10: *Multibody model of the vehicle used in simulations*

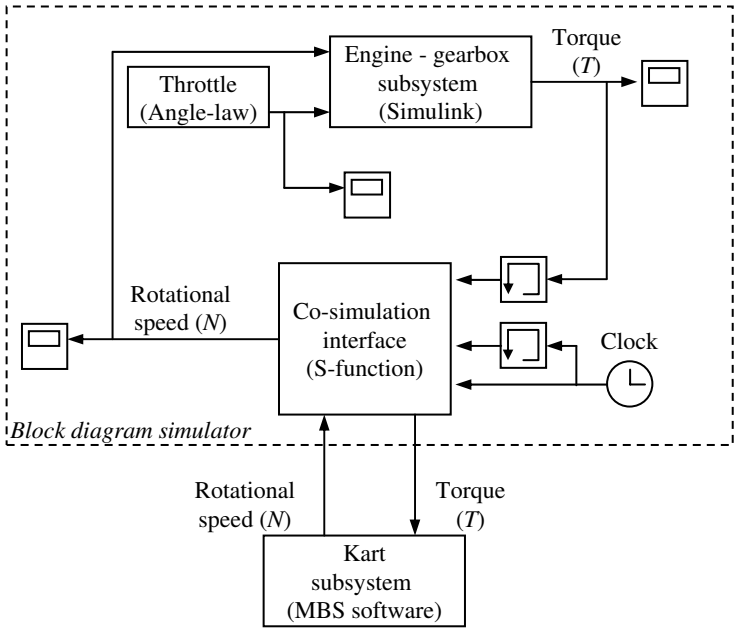

Figure 6.11: *Joint model of the engine and the kart*

system is 163, and the motion is integrated making use of the well–known index–3 augmented Lagrangian formulation with projection of velocities and accelerations. This formalism uses the trapezoidal rule as numerical integrator, it has been described

by Cuadrado et al. (2000) and a brief overview of its equations can be found in Section 3.2.2. The multibody code is implemented in Fortran and its configuration is detailed by Naya et al. (2007).

The engine is modelled in Simulink, following the description given by Crossley and Cook (1991), using conventional diagram blocks and adding an automatic gearbox to link it to the transmission. The block diagram model that corresponds to this system is shown in Figure 6.11. The upper part of the graphic represents the Simulink model of the engine and the gearbox, which also includes the *co–simulation interface*, described in Section 6.2. *Memory* blocks are used to avoid the closing of an algebraic loop. The code for the simulation of the mechanical components of the kart is compiled as a library and invoked from the co–simulation interface. The model undergoes a maneuver in which the angle of the throttle varies following the law depicted in Figure 6.12. The pitch angle of the vehicle (ψ) is taken as control variable, to check if the setting behaves in an adequate way. This variable is closely related to the acceleration of the vehicle.

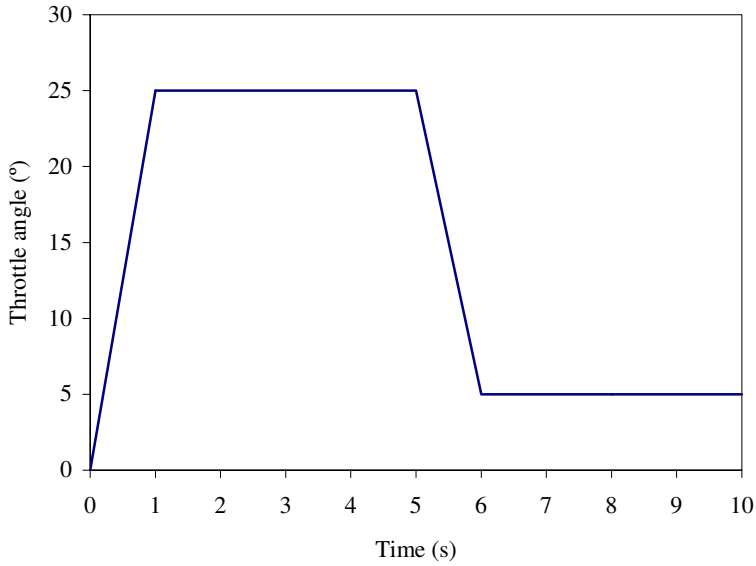

Figure 6.12: *Throttle angle during simulations*

The Simulink part of the model is integrated with *ode4*, and the nature of the system it models requires using a time–step of $h_1 = 10^{-4}$ s. The multibody subsystem can be integrated with trapezoidal rule with time–steps as big as $h_2 = 10^{-2}$ s without significant errors. A direct co–simulation scheme with the same time–step in both subsystems would be forced to use the smallest one to keep numerical accuracy in the fast component, leading to a considerable increase in the total computation time. In the performed simulations, the time–step in Simulink has been kept constant, and the time–step of the multibody subsystem has been varied from $h_2 = 10^{-4}$ s to $h_2 =$

10^{-2} s in order to measure the effect of using multirate integration on the accuracy and efficiency of the simulation. The case in which both subsystems are integrated with the same time–step $h_1 = h_2 = 10^{-4}$ s and constant interpolation $O0$ is taken as the reference solution; the pitch angle in this case is shown in Figure 6.13. The shape of the pitch angle curve in this graphic agrees with the angle law for the throttle depicted in Figure 6.12. The sudden drops in the angle between seconds 4 and 5 and at second 8 correspond to the moments when the gear of the vehicle is changed by the automatic gearbox.

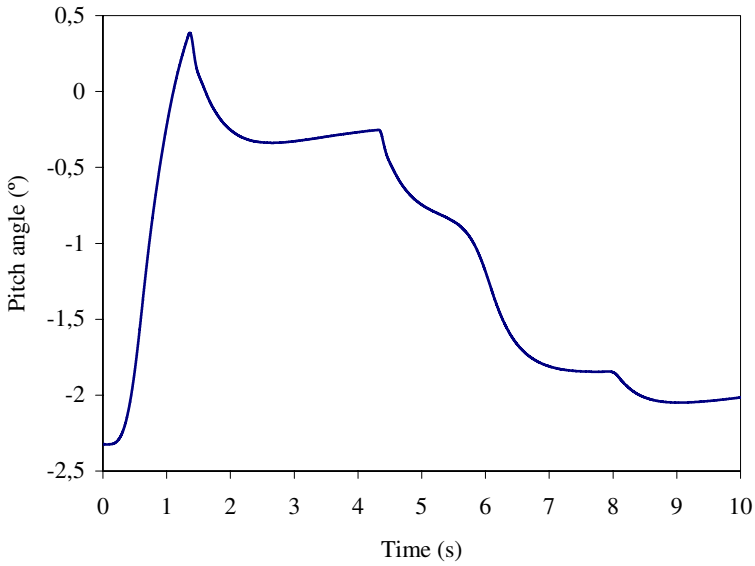

Figure 6.13: *Pitch angle ψ in reference case*

The total computing time of the 10 s simulation under the reference conditions $h_1 = h_2 = 10^{-4}$ s exceeds 150 s. The use of multirate co–simulation is expected to reduce the total computing time; however, it is also reasonable to expect divergences to occur in the results with respect to the reference solution. In order to measure the impact of multirate simulation in the elapsed time in computations and the deviations from the reference value, simulations at different values of *FR* have been carried out. It must be noted that the meaning of *FR*, for complex multiphysics problems like the one here discussed, does not correspond to the ratio between the natural frequencies of the subsystems (which may not be easy to identify), but it must be substituted by the relation between the time–steps used to integrate them. For this first set of simulations, constant interpolation ($O0$) and *slowest–first* strategy have been used. Besides the computation time, the maximum deviation in pitch angle with respect to the reference case during the motion has been measured.

The results summarized in Table 6.2 show a dramatic reduction in computing time as the value of *FR* increases. Regarding to the differences in the pitch angle ψ, these

Table 6.2: *Elapsed time in simulations and maximum difference in pitch angle ($\Delta\psi$) with respect to the reference case, for different values of FR, with* SF *and* O*0 interpolation in both subsystems*

	$FR = 1$ (ref.)	$FR = 5$	$FR = 10$	$FR = 50$	$FR = 100$
Elapsed time (s)	158.4	44.8	30.4	19.0	17.1
$\Delta\psi$ (°)	0	0.0031	0.0055	0.0252	0.0398

are never higher than 0.04° in absolute value, for a variable that oscillates between -2.5° and 0.5°. This means that direct co–simulation, with the use of *O*0 polynomials, is able to simulate the system without significant deviations in the results, with values of *FR* up to 100. The plots of the pitch angle for the different values of *FR* overlap Figure 6.13, so they are indistinguishable in practice. A graphical representation of the deviation of the control variable with respect to that of the reference case has been chosen instead, and it can be seen in Figure 6.14 for $FR = 100$. The sudden variations of the measured deviation make the results in this graphic look like a solid region, but a line is actually represented.

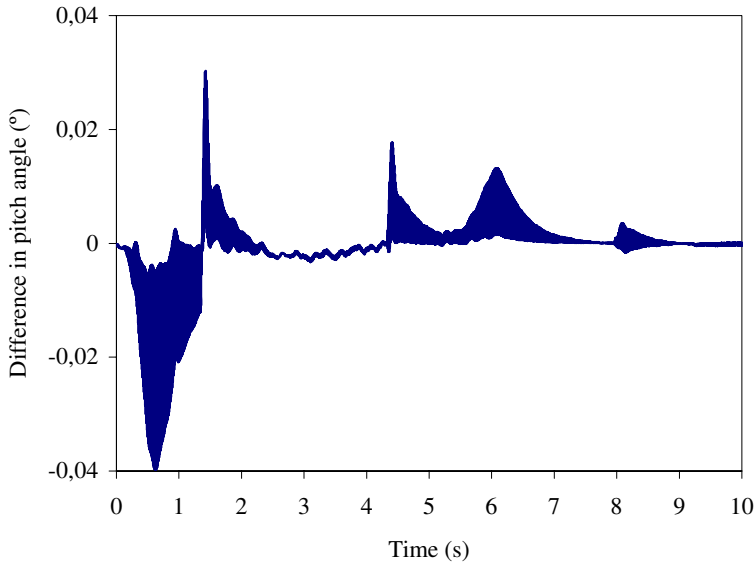

Figure 6.14: *Difference in pitch angle ($\Delta\psi$) with respect to the reference case, with* $FR = 100$, SF *and* O*0 in both subsystems*

Figure 6.14 shows another relevant feature of the behaviour of the co–simulated system: the divergences in pitch angle increase when sudden variations of the variable happen, but the error gets close to zero when the angle varies slowly. This stable behaviour of the whole system agrees with the conclusions stated in Section 6.3.3.

Table 6.3: *Maximum difference in pitch angle ($\Delta\psi$) with respect to the reference case for FR = 100. Only representative interpolation strategies are represented*

	SF	*FF*	*FF*	*FF*	*FF*
Simulink interpolation	*O*0	*O*0	*O*0	*O*0	*O*0
MBS interpolation	*O*0	*O*0	*O*1	*O*2	*O*3
$\Delta\psi$ (°)	0.0398	0.0385	0.0078	0.0086	0.0303

As it was shown in the previous Section, it is not possible to determine beforehand whether the use of higher order interpolation polynomials or other co–simulation techniques will enhance the obtained results. More simulations have been performed to gain insight into this subject; the most relevant ones are summarized in Table 6.3 for $FR = 100$. The elapsed time is not shown, as there are not significant differences between the methods. Other configurations have been tested (alternative combinations of orders in interpolation polynomials and smoothing techniques) but their use has not improved the precision of the simulation. As it can be drawn from the table, there is no gain in rising the order of the polynomials beyond one, as the linear case yields the best results.

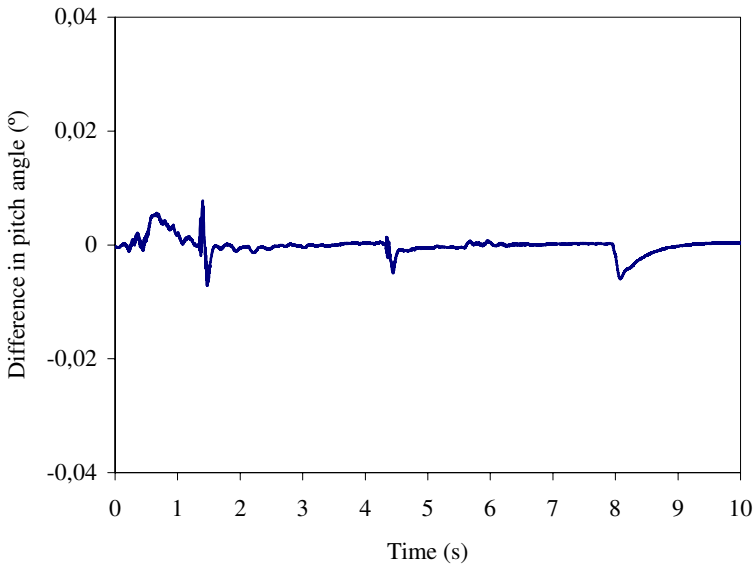

Figure 6.15: *Difference in pitch angle ($\Delta\psi$) with respect to the reference case, with FR = 100,* FF, O*0 interpolation in Simulink and* O*1 interpolation in MBS*

The time–history of the pitch angle in the case that performs best in Table 6.3 is represented in Figure 6.15. The comparison of this graphic to the one in Figure 6.14 highlights the benefits of using the *fastest–first* configuration and linear polynomials for the interpolation of the data from the MBS software in this particular case.

6.5 Conclusions

In this Chapter, the effect of multirate techniques in the efficiency and accuracy of weakly coupled co–simulation settings has been assessed. To this end, a general multirate co–simulation interface for coupling block diagram simulators and external tools has been built, and a synchronization algorithm has been designed, in order to coordinate the exchange of information between both software packages. The way in which the interface operates is based on interpolation and extrapolation of inputs and outputs between simulators using polynomial approximations, and two synchronization schemes are available: *slowest–first* and *fastest–first*. The interface avoids techniques which are not available in block diagram simulators (iteration or modifications in the integration schemes) and overcomes the limitations of the current commercial coupling solutions, since it can be used with non–synchronized, variable–step multirate integration time–grids. This interface allows the user to select different co–simulation settings, such as the order of the interpolation polynomials, and incorporates additional techniques to improve the behaviour of the simulation under certain conditions.

The algorithm has been implemented in C/C++ and tested in the co–simulation of the dynamics of a simple, purely mechanical system by coupling the well–known simulation tool Simulink with the multibody dynamics simulator developed in this thesis. The accuracy of the method was tested against the frequency ratio FR, which is equivalent to the ratio between the time–step sizes used in the two coupled simulators. The first test battery of the designed interface has revealed that the adjustment of the co–simulation settings is strongly dependent of the physical characteristics of the simulated subsystems. As a consequence, the co–simulation parameters must be adapted as a function of the particular features of the problem, and a general configuration, valid for any situation, cannot be found. In some cases, the use of smoothing is required in order to find a stable solution to the problem.

Next, the interface has been applied to the co–simulation of the multibody model of a real kart, simulated in a Fortran MBS code, powered by a thermal engine modelled in Simulink. Results show that the use of multirate techniques has been able to reduce the computation time required by the simulation in one order of magnitude, within a reasonable margin of error. In this case, the use of first order interpolation polynomials ($O1$) has contributed to alleviate the deviations of the motion with respect to the reference solution. The example is very representative of the co–simulation of complex mechatronic systems, where the dynamic simulation of the mechanical components in a multibody software consumes around 60% – 90% of the CPU–time, while the remaining time is consumed by the block diagram simulator. In these circumstances, increasing the stepsize in the multibody dynamics simulator by a factor of 50 can reduce the time needed to complete the simulations in a factor ranging from 2.4 to 8.5. The described multirate interface represents a significant improvement over current off–the–shelf commercial coupling solutions, which enforce equal time–steps ($FR = 1$) in both sides of the co–simulation.

Currently, two lines of future research can be pointed out. First, a general numerical indicator is desirable, in order to measure in a practical and easy way the deviation

of a solution with respect to a reference. And second, a way of determining the optimal co–simulation strategy before running the simulation would be very helpful, as it would remove the need of performing several trials to adjust the interface to the particular conditions of the simulated system.

Chapter 7

Conclusions

7.1 Conclusions

The main goal of the present thesis is the evaluation of different techniques for the optimization of multibody simulation codes. This work intends to contribute to the two main currently open lines of research of the multibody community: the reduction of the elapsed time in computations, and the addition of extra functionality beyond the purely mechanical simulation.

The development of a generic and modular software architecture for MBS simulation has been addressed in Chapter 2. While the designing and implementation of a simulation package is a complex task, subject to a considerably high number of design variables, it has been possible to point out some recommendations of general validity, specially regarding the modularity of the project. Modularity is a key feature of this kind of software, and the object–oriented paradigm is the best suited to achieve it. Following this approach, an operational platform for the simulation of mechanical systems has been built in C++, in which the optimization strategies proposed in this work have been tested. Its modular nature makes the software capable of incorporating new functionality, so the writing of the code can never be considered finished. At the moment, the module for the automatic generation of the equations of motion of the system is under development.

Chapter 3 discusses the effect of the implementation of linear algebra routines in the performance of the software. It has been found that the use of efficient libraries of basic routines for matrix computations (BLAS for dense storage, and equivalent routines for sparse matrices) and linear solvers (LAPACK, CHOLMOD, KLU) can speed up the execution of the code in a factor of 2–3, without negative side effects on the portability of the code. Decision rules for selecting the most adequate solver as a function of the size and number of non–zeros of the leading matrix of the system have also been provided.

The use of non–intrusive parallelization methods is the main subject of Chapter 4. Parallel linear equation solvers and OpenMP have been selected in this work; their use has been preferred to that of more efficient but intricate parallelization protocols,

such as MPI. These techniques have been successfully applied to existing software with minimal modifications in the code. Results have shown that parallel solvers and OpenMP can be applied to a wide range of problems in multibody dynamics, obtaining speedups over 70% of the theoretical maximum values according to Amdahl's law.

Finally, Chapters 5 and 6 deal with the coupling of the multibody architecture to external software packages, thus enlarging the capabilities of the basic MBS program. First, the different alternatives when communicating the MBS simulation software with MATLAB/Simulink have been considered; two main coupling categories (*function evaluation* and proper *co–simulation*) have been identified, and a comparison of the different strategies under each of them, in terms of efficiency, has been carried out. In a second stage, a multirate co–simulation interface between multibody software and block diagram simulators has been built. This interface is able to coordinate the integrators in each software tool, even if they use different time–grids; it also features different interpolation and synchronization strategies to manage the execution of the simulation. The communication techniques and the interface have been used to evaluate the possibilities of software coupling in demanding applications such as real–time settings. The obtained results demonstrate that co–simulated models can outperform their monolithic counterparts under certain circumstances and be used in non–academic, real applications.

7.2 Future research

The research developed in this work falls within the active projects of the Laboratory of Mechanical Engineering of the University of La Coruña, and it is the continuation of the predating efforts of its researchers. The conclusions obtained in this thesis have been added to the know–how of the group, and therefore they will be used in the future development of efficient code for multibody simulation. Also, the software architecture described in this work will serve as a platform for the implementation and testing of new components.

The code optimizations presented in Chapters 3 and 4 can be introduced in implementations of dynamic formulations different from the ones tested in this work; for example, they are good candidates for the enhancement of recursive and semi–recursive formulations. An assessment of the obtained improvements in performance could be obtained, leading to a comparison with respect to the results found in this thesis with global formulations.

Finally, the co–simulation strategies described in Chapters 5 and 6 can be applied in a direct way to the simulation of complex multiphysics systems, thanks to the multirate interface described in Section 6.2. Thus, the fast and effective simulation of non–trivial problems with a non purely mechanical nature is greatly simplified. As it was pointed out in these Chapters, a way to find out the optimal co–simulation strategy for each problem beforehand, based on the characteristics of the involved subsystems, constitutes a desirable research goal that would save time and efforts in the adjustment of the co–simulation setting. In this research, a general purpose indicator of the quality of the co–simulation technique would represent a valuable intermediate objective.

Appendix

This Appendix details the procedure that has been followed in this work to compile a Simulink model into a dynamically linked library (a *.dll* file, under Windows operating system) through the use of MATLAB's Real–Time Workshop.

Real–Time Workshop (RTW) is a complementary module of MATLAB/Simulink that generates standalone C code from block diagram models. The resulting code can be later compiled into an executable program with a conventional C/C++ compiler. The performance of this compiled code is much higher than that of the original Simulink model, allowing thus its use in highly demanding applications, such as real–time settings. Sometimes, however, it would be desirable that the C code was turned into a dynamically linked library in order to allow calling the routines it contains from a different program. This can be the case, for example, of the Simulink models that represent controllers or actuators for mechanical systems. It is also the same situation described in Section 5.4.3 as *MBS as master*.

Currently, the creation of dynamically linked libraries from the C code generated by RTW is not straightforward, and a series of operations must be carried out in order to achieve this goal. Some instructions can be found in MATLAB's website (The Mathworks, Inc., 2009) but, even so, the complete task is not intuitive and is prone to errors. A short description of the steps that should be taken to build the library is provided below. The problem solved in Section 5.4 will serve as example.

Translation of the Simulink model to C via RTW

In the example solved in Section 5.4, a Simulink model of an engine is coupled to an MBS model of an L–loop four–bar linkage. This simulation is performed under the Windows operating system, using Microsoft Visual Studio as C/C++ compiler; RTW settings are function of the operating system and the compiler, and so are the final output files of the process which is being described. Nonetheless, the steps which must be taken are very similar in every case. When the process is complete, the coupling of both subsystems is the one depicted in Figure A.1. The MBS software simulating the motion of the linkage acts as the driver program, requesting values from the C library generated via RTW, compiled as a *.dll* file, which simulates the thermal engine. It should be noted that the coupling scheme corresponds to a co–simulation one, as every subsystem (namely, the executable and the library) includes its own integrator.

The Simulink model from which the C code is created must be fine–tuned and

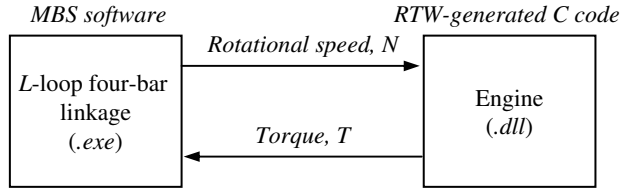

Figure A.1: *General layout of* MBS as master *coupling scheme under Windows OS*

tested before its compilation as a library. If the model in the *.dll* library has to be changed after the compilation has taken place, the need of repeating all the steps in the process arises, in order to take the changes in the model to the code of the library. For this reason, it seems very convenient to test the model previously to its compilation, with one of the two configurations described in Sections 5.4.1 (*Network connection*) and 5.4.2 (*Simulink as master*). These coupling techniques make possible to test the Simulink model in real interaction with the external software, while modifying easily its parameters and its configuration to attain the desired behaviour of the whole system.

When the functioning of the assembled system is correct, RTW can be invoked from Simulink in order to generate the C equivalent of the model. Two files are necessary in this step: a *system target file* (*.tlc*) and a *template makefile* (*.tmf*). Both of them depend on the compiler that is going to be used. In this case, for Visual Studio, *grt.tlc* and *grt_msvc.tmf* are used. The execution of RTW yields a series of C files, which encapsulate the functionality of the model, and one or more *makefiles* or project files, to manage the compilation of the code.

Edition of generated files

The execution of the *makefiles* or project files as they are created by RTW would lead to the compilation of an executable application; they have to be edited to yield a library.

The first step in this stage is finding the *main* process in the C files. In the current case, this is contained in file *grt_main.c*. This file must be modified, removing its *main* function and dividing its functionality among newly created equivalent functions that can be called by the MBS software. These are three C functions that return a pointer to a *char* type, namely the following ones:

- *char* initiate()*. This function initializes the global memory and the Simulink model. It must be called once, at the beginning of the simulation.
- *char* getOutput(int nInputs, double* inputs, int nOutputs, double* outputs)*. This function is the main communication gateway between the executable and the library, and it must be called in every time–step. It receives a number of input arguments *nInputs* and returns a number of outputs *nOutputs*. These arguments are pointed to by the pointers *inputs* and *outputs*. The code of this function must be edited in order to assign the outputs of the Simulink model

to the corresponding element of the *outputs* array; the same operation must be done for the inputs. If a co–simulation interface is used, this interface will be responsible for synchronizing the calls to this function and the execution of the MBS integration.

- *char* performCleanup()*. It executes the termination routines and shuts down the model. It must be called once, at the end of the simulation.

The names of these three functions are not standard and they may vary from one model to another. A header file with their prototypes (*.h*) and a module definition file (*.def*) with their names have to be created to allow calling the functions from outside of the dynamically linked library.

The *makefiles* or the project files (in this case, the Visual Studio project) must also undergo several changes, in order to fit the generation of a library, instead of an executable. The type of the output file must be *.dll* instead of *.exe*, the linking must be made compatible with that of the MBS executable, and the module definition file must be included and used for creating an import library (*.lib*), which will be necessary for accessing the library from the MBS executable. These steps would be different with other platforms and compilers. For example, dynamically linked libraries are replaced by shared libraries (*.so*) under UNIX and there is no need for using module definition files during their creation. The great number of possible combinations of platforms and compilers makes impossible the detailed enumeration of all these particularities in this Appendix.

Compilation and linking to the executable

Once the project is ready, the compilation of the C code generated by RTW can be performed. The final output of the compilation process, in the case of using Visual Studio, is made up of three files:

- the dynamically linked library itself (a *.dll* file). This file includes the compiled code that executes the functionality contained in the Simulink model;
- an import library (a *.lib* file), to enable the linking of the library; and
- the header file (*.h*) with the prototypes of the functions.

The MBS executable must be linked to the *.dll* library, adding the import library to the linker parameters and including the header file in its code. After these changes have been made, the MBS program can be compiled, yielding the executable that constitutes the final result of this process.

Bibliography

A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison–Wesley, 2001.

J. A. C. Ambrósio and J. P. C. Gonçalves. Complex flexible multibody systems with application to vehicle dynamics. *Multibody System Dynamics*, 6(2):163–182, 2001.

AMD. AMD Core Math Library (ACML), 2009. `http://developer.amd.com/cpu/libraries/acml/Pages/default.aspx`.

P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

K. S. Anderson and J. H. Critchley. Improved 'order–n' performance algorithm for the simulation of constrained multi–rigid–body dynamic systems. *Multibody System Dynamics*, 9(2):185–212, 2003.

K. S. Anderson and S. Duan. A hybrid parallelizable low–order algorithm for dynamics of multi–rigid–body systems: Part I, chain systems. *Mathematical and Computer Modelling*, 30(9-10):193–215, 1999.

K. S. Anderson and M. Oghbaei. A state–time formulation for dynamic systems simulation using massively parallel computing resources. *Nonlinear Dynamics*, 39(3): 305–318, 2005.

K. S. Anderson, R. Mukherjee, J. Critchley, J. Ziegler, and S. Lipton. POEMS: Parallelizable open–source efficient multibody software. *Engineering with Computers*, 23(1):11–23, 2007.

Argonne National Laboratory. The Message Passing Interface (MPI) standard, 2009. `http://www-unix.mcs.anl.gov/mpi/`.

M. Arnold. Numerical Methods for Simulation in Applied Dynamics. *Simulation Techniques for Applied Dynamics*, M. Arnold and W. Schiehlen, (eds.), pp. 191–246. Springer, 2008.

A. Avello, J. M. Jiménez, E. Bayo, and J. García de Jalón. A simple and highly parallelizable method for real–time dynamic simulation based on velocity transformations. *Computer Methods in Applied Mechanics and Engineering*, 107(3):313–339, 1993.

D. S. Bae, J. G. Kuhl, and E. J. Haug. A recursive formulation for constrained mechanical system dynamics. 3. Parallel processor implementation. *Mechanics of Structures and Machines*, 16(2):249–269, 1988.

D. S. Bae, J. K. Lee, H. J. Cho, and H. Yae. An explicit integration method for realtime simulation of multibody vehicle models. *Computer Methods in Applied Mechanics and Engineering*, 187(1-2):337–350, 2000.

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

M. Busch, M. Arnold, A. Heckmann, and S. Dronka. Interfacing SIMPACK to Modelica/Dymola for multi–domain vehicle system simulations. *SIMPACK News*, 11 (2):1–3, 2007.

Canonical Ltd. Bazaar, 2010.
`http://bazaar.canonical.com/en/`.

J. R. Cary, S. G. Shasharina, J. C. Cummings, J. V. W. Reynders, and P. J. Hinker. Comparison of C++ and Fortran 90 for object–oriented scientific programming. *Computer Physics Communications*, 105(1):20–36, 1997.

B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 887: CHOLMOD, supernodal sparse cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software*, 35(3):art. 22, 2008.

J. H. Critchley and K. S. Anderson. On parallel methods of multibody dynamics. In *Proceedings of the ASME Design Engineering Technical Conference*, pp. 133–142, Chicago, IL, 2003.

J. H. Critchley and K. S. Anderson. A parallel logarithmic order algorithm for general multibody system dynamics. *Multibody System Dynamics*, 12(1):75–93, 2004.

P. R. Crossley and J. A. Cook. A nonlinear engine model for drivetrain system development. In *International Conference on Control (Control 91)*, pp. 921–925, Edinburgh, 1991.

J. Cuadrado, J. Cardenal, and J. García de Jalón. Flexible mechanisms through natural coordinates and component synthesis: An approach fully compatible with the rigid case. *International Journal for Numerical Methods in Engineering*, 39(20):3535–3551, 1996.

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

J. Cuadrado, J. Cardenal, P. Morer, and E. Bayo. Intelligent simulation of multibody dynamics: Space–state and descriptor methods in sequential and parallel computing environments. *Multibody System Dynamics*, 4(1):55–73, 2000.

J. Cuadrado, R. Gutiérrez, M. Naya, and P. Morer. 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, 2001.

J. Cuadrado, D. Dopico, M. González, and M. A. Naya. A combined penalty and recursive real–time formulation for multibody dynamics. *Journal of Mechanical Design*, 126(4):602–608, 2004a.

J. Cuadrado, D. Dopico, M. A. Naya, and M. González. Penalty, semi–recursive and hybrid methods for MBS real–time dynamics in the context of structural integrators. *Multibody System Dynamics*, 12(2):95–185, 2004b.

J. Cuadrado, D. Dopico, M. A. Naya, and M. González. Real–Time Multibody Dynamics and Applications. *Simulation Techniques for Applied Dynamics*, M. Arnold and W. Schiehlen, (eds.), pp. 247–311. Springer, 2008.

R. Davis, B. Henz, and D. Shires. Performance evaluation of parallel sparse linear equation solvers for positive definite systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 1172–1178, Las Vegas, NV, 2003.

T. A. Davis. Algorithm 832: UMFPACK V4.3 – An unsymmetric–pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.

T. A. Davis and E. P. Natarajan. Algorithm 8xx: KLU, a direct sparse solver for circuit simulation problems, 2010.
`http://www.cise.ufl.edu/~davis/techreports/KLU/KLU.pdf`.

B. Dawes, D. Abrahams, and R. Rivera. Boost C++ libraries, 2009.
`http://www.boost.org/`.

J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999a.

J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, 1999b.

Dipartamento di Ingegneria Aerospaziale, Politecnico di Milano. MBDyn – MultiBody Dynamics Software, 2009.
`http://www.aero.polimi.it/mbdyn/`.

J. Dongarra. Freely available software for linear algebra on the web, 2009.
`http://www.netlib.org/utk/people/JackDongarra/la-sw.html`.

D. Dopico, U. Lugrís, M. González, and J. Cuadrado. Two implementations of IRK integrators for real–time multibody dynamics. *International Journal for Numerical Methods in Engineering*, 65(12):2091–2111, 2006.

A. Eichberger, C. Führer, and R. Schwertassek. The benefits of parallel multibody simulation. *International Journal for Numerical Methods in Engineering*, 37(9): 1557–1572, 1994.

C. Engstler and C. Lubich. Multirate extrapolation methods for differential equations with different time scales. *Computing*, 58(2):173–185, 1997.

P. Fisette and J. M. Péterkenne. Contribution to parallel and vector computation in multibody dynamics. *Parallel Computing*, 24(5-6):717–728, 1998.

Free Software Foundation. GCC, the GNU Compiler Collection, 2009.
`http://gcc.gnu.org/`.

Function Bay, Inc. RecurDyn, 2009.
`http://www.functionbay.co.kr/`.

J. García de Jalón and E. Bayo. *Kinematic and Dynamic Simulation of Multibody Systems – The Real–Time Challenge*. Springer–Verlag, 1994.

J. C. García Orden and J. M. Goicolea. Conserving properties in constrained dynamics of flexible multibody systems. *Multibody System Dynamics*, 4(2-3):225–244, 2000.

C. W. Gear and D. R. Wells. Multirate linear multistep methods. *BIT Numerical Mathematics*, 24(4):484–502, 1984.

M. González. *A Collaborative Environment for Flexible Development of MBS Software*. PhD thesis, University of La Coruña, 2005.

M. González, D. Dopico, U. Lugrís, and J. Cuadrado. A benchmarking system for MBS simulation software: Problem standardization and performance measurement. *Multibody System Dynamics*, 16(2):179–190, 2006.

M. González, F. González, A. Luaces, and J. Cuadrado. Interoperability and neutral data formats in multibody system simulation. *Multibody System Dynamics*, 18(1): 59–72, 2007.

M. González, F. González, A. Luaces, and J. Cuadrado. A collaborative benchmarking framework for multibody system dynamics. *Engineering with Computers*, 26:1–9, 2010.

P. F. Gorder. Multicore processors for science and engineering. *Computing in Science & Engineering*, 9(2):3–7, 2007.

K. Goto. GotoBLAS, 2009.
`http://www.tacc.utexas.edu/resources/software/`.

N. I. M. Gould, J. A. Scott, and Y. Hu. A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Transactions on Mathematical Software*, 33(2):art. 10, 2007.

A. Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Transactions on Mathematical Software*, 28(3):301–324, 2002.

A. Gupta. A shared– and distributed–memory parallel general sparse direct solver. *Applicable Algebra in Engineering, Communications and Computing*, 18(3):263–277, 2007.

A. Gupta, M. Joshi, and V. Kumar. WSSMP: A High–Performance Serial and Parallel Symmetric Sparse Linear Solver. *Applied Parallel Computing – Large Scale Scientific and Industrial Problems*, B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, (eds.), pp. 182–194. Springer, 1998.

H. S. Han and J. H. Seo. Design of a multi–body dynamics analysis program using the object–oriented concept. *Advances in Engineering Software*, 35(2):95–103, 2004.

IEEE P1076.1 Working Group. VHDL–AMS, 2009.
`http://www.eda.org/vhdl-ams/`.

INRIA. Scicos: Block diagram modeler/simulator, 2009a.
`http://www.scicos.org/`.

INRIA. Scilab: The open source platform for numerical computation, 2009b.
`http://www.scilab.org/`.

N. M. Josuttis. *The C++ Standard Library – A Tutorial and Reference*. Addison–Wesley, 1999.

G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20(1):359–392, 1998.

H. Kasahara, H. Fujii, and M. Iwata. Parallel processing of robot motion simulation. In *Proceedings IFAC 10th World Conference*, pp. 329–336, Munich, 1987.

A. Kecskeméthy and M. Hiller. Object–oriented programming techniques in vehicle dynamics simulation. *Mathematics and Computers in Simulation*, 39(5–6):549–558, 1995.

G. Krawezik and F. Cappello. Performance comparison of MPI and OpenMP on shared memory multiprocessors. *Concurrency and Computation: Practice and Experience*, 18(1):29–61, 2006.

R. Kübler and W. Schiehlen. Modular simulation in multibody system dynamics. *Multibody System Dynamics*, 4(2-3):107–127, 2000.

Y. G. Liao and H. I. Du. Cosimulation of multi–body–based vehicle dynamics and an electric power steering control system. *Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi–body Dynamics*, 215(3):141–151, 2001.

U. Lugrís, M. A. Naya, F. González, and J. Cuadrado. Performance and application criteria of two fast formulations for flexible multibody dynamics. *Mechanics Based Design of Structures and Machines*, 35(4):381–404, 2007.

K. Martin and B. Hoffman. *Mastering CMake*. Kitware, Inc., 4th edition, 2007.

J. McPhee. Multibody system dynamics: Research activities, 2008. `http://real.uwaterloo.ca/~mbody/`.

S. Meyers. *More effective C++*. Addison–Wesley, 1999.

S. Meyers. *Effective C++*. Addison–Wesley, 2nd edition, 2000.

Microsoft. Visual Studio 2008 – Express Edition, 2009. `http://www.microsoft.com/express/vc/`.

A. Mikkola. Utilization of coupled simulation in the fatigue loads prediction of a hydraulically driven log crane. In *Proceedings of the ASME 2001 Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, paper VIB-21360, Pittsburgh, PA, 2001.

Modelica Association. Modelica, Modeling of Complex Physical Systems, 2009. `http://www.modelica.org/`.

MSC.Software Corporation. ADAMS, 2009. `http://www.mscsoftware.com/`.

R. M. Mukherjee, K. S. Anderson, and J. Ziegler. Multigranular molecular dynamics simulations of polymer melts using multibody algorithms. In *Proceedings of the ASME 2005 Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pp. 2111–2120, Long Beach, CA, 2005.

R. M. Mukherjee, P. S. Crozier, S. J. Plimpton, and K. S. Anderson. Substructured molecular dynamics using multibody dynamics algorithms. *International Journal of Non-linear Mechanics*, 43(10):1040–1055, 2008.

National Instruments. MATRIXx Software Suite, 2009. `http://www.ni.com/matrixx/what_is_matrixx.htm`.

M. A. Naya, D. Dopico, J. A. Pérez, and J. Cuadrado. Real–time multi–body formulation for virtual–reality–based design and evaluation of automobile controllers. *Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi–body Dynamics*, 221(2):261–276, 2007.

NETLIB. LAPACK, Linear Algebra PACKage, 2009.
`http://www.netlib.org/lapack/`.

N. M. Newmark. A method of computation for structural dynamics. *Journal of the Engineering Mechanics Division, ASCE*, 85(EM3):67–94, 1959.

NIST. BLAS, Basic Linear Algebra Subprograms, 2009.
`http://www.netlib.org/blas/`.

NIST. Matrix Market, 2007.
`http://math.nist.gov/MatrixMarket/`.

Nokia. Qt – A cross–platform application and UI framework, 2009.
`http://qt.nokia.com/`.

O. Oberschelp and H. Vöcking. Multirate simulation of mechatronic systems. In *Proceedings of the IEEE International Conference on Mechatronics 2004*, pp. 404–409, Istanbul, 2004.

OpenMP Architecture Review Board. OpenMP, 2008.
`http://openmp.org`.

OSG Community. Open Scene Graph, 2009.
`http://www.openscenegraph.org/projects/osg`.

G. Quaranta, P. Masarati, and P. Mantegazza. Multibody analysis of controlled aeroelastic systems on parallel computers. *Multibody System Dynamics*, 8(1):71–102, 2002.

Y. Saad. *Iterative methods for sparse linear systems*. SIAM, Philadelphia, PA, 2nd edition, 2000.

J. C. Samin, O. Brüls, J. F. Collard, L. Sass, and P. Fisette. Multiphysics modeling and optimization of mechatronic multibody systems. *Multibody System Dynamics*, 18(3):345–373, 2007.

SAMTECH. SAMCEF Mecano, 2009.
`http://www.samcef.com/products/product.asp?idP=92&prod=11`.

Sandia National Laboratories. LAMMPS Molecular Dynamics Simulator, 2009.
`http://lammps.sandia.gov/`.

V. Savcenco, W. Hundsdorfer, and J. G. Verwer. A multirate time stepping strategy for stiff ordinary differential equations. *BIT Numerical Mathematics*, 47(1):137–155, 2007.

O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker. PARDISO: A high–performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer Systems*, 18(1):69–78, 2001.

W. Schiehlen. Research trends in multibody system dynamics. *Multibody System Dynamics*, 18(1):3–13, 2007.

J. A. Scott and Y. Hu. Experiences of sparse direct symmetric solvers. *ACM Transactions on Mathematical Software*, 33(3):art. 18, 2007.

Selenic Consulting. Mercurial, 2010.
`http://mercurial.selenic.com/`.

L. F. Shampine. *Numerical Solution of Ordinary Differential Equations*. Chapman & Hall, 1994.

S. S. Shome, E. J. Haug, and L. O. Jay. Dual–rate integration using partitioned Runge–Kutta methods for mechanical systems with interacting subsystems. *Mechanics Based Design of Structures and Machines*, 32:3(3):253–282, 2004.

Simbios project. SimTK, 2009.
`https://simtk.org/xml/index.xml`.

SIMPACK AG. SIMPACK, 2009.
`http://www.simpack.de/`.

H. Sugiyama and A. A. Shabana. Application of plasticity theory and absolute nodal coordinate formulation to flexible multibody system dynamics. *Journal of Mechanical Design*, 126(3):478–487, 2004.

M. Tändl, T. Stark, N. E. Erol, F. Löer, and A. Kecskeméthy. An object–oriented approach to simulating human gait motion based on motion tracking. *International Journal of Applied Mathematics and Computer Science*, 19(3):469–483, 2009.

J. Teppo, A. Rouvinen, A. Mikkola, P. Kurronen, P. Salminen, and O. Pyrhönen. Coupled Simulation of Electrically Driven Machine System. *Bath Workshop on Power Transmission and Motion Control (PTMC 2001)*, C. R. Burrows and K. A. Edge, (eds.), pp. 103–116. Professional Engineering Publishing, 2001.

Tesis DYNAware. veDYNA, 2009.
`http://www.tesis.de/en/index.php?page=544`.

The AEgis Technologies Group, Inc. ACSLX, 2009.
`http://www.acslsim.com/`.

The Mathworks, Inc. MATLAB, 2009.
`http://www.mathworks.com/`.

The wxWidgets team. wxWidgets – Cross–platform GUI library, 2009.
`http://www.wxwidgets.org/`.

Tigris.org. Subversion, 2009.
`http://subversion.tigris.org/`.

F. T. Tracy, T. C. Oppe, and S. Gavali. Testing parallel linear iterative solvers for finite element groundwater flow problems. In *Department of Defense - Proceedings of the Users Group Conference 2007; High Performance Computing Modernization Program: A Bridge to Future Defense*, pp. 474–481, Pittsburgh, PA, 2007.

S. Turek, C. Becker, and A. Runge. The FEAST indices – Realistic evaluation of modern software components and processor technologies. *Computers and Mathematics with Applications*, 41(10-11):1431–1464, 2001.

O. Vaculín, W. R. Krüger, and M. Valášek. Overview of coupling of multibody and control engineering tools. *Vehicle System Dynamics*, 41(5):415–429, 2004.

M. Valášek. Modeling, Simulation and Control of Mechatronical Systems. *Simulation Techniques for Applied Dynamics*, M. Arnold and W. Schiehlen, (eds.), pp. 75–140. Springer, 2008.

Valgrind developers. Valgrind, 2009.
`http://valgrind.org/`.

D. van Heesch. Doxygen, 2009.
`http://www.stack.nl/~dimitri/doxygen/`.

A. Verhoeven, E. J. W. Ter Maten, R. M. M. Mattheij, and B. Tasić. Stability analysis of the BDF slowest–first multirate methods. *International Journal of Computer Mathematics*, 84(6):895–923, 2007.

J. Walter., M. Koch, and G. Winkler. UBLAS, 2009.
`http://www.boost.org/libs/numeric/`.

R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

Wolfram Research. Mathematica, 2009.
`http://www.wolfram.com/`.

J. S. K. Yu and C. H. Yu. Recent advances in PC–Linux systems for electronic structure computations by optimized compilers and numerical libraries. *Journal of Chemical Information and Computer Sciences*, 42(3):673–681, 2002.

Publications

The research carried out in this thesis has yielded the following publications:

- M. González, F. González, D. Dopico and A. Luaces. On the effect of linear algebra implementations in real–time multibody system dynamics. *Computational Mechanics*, 41(4):607–615. 2008.
- F. González, A. Luaces, U. Lugrís and M. González. Non–intrusive parallelization of multibody system dynamic simulations. *Computational Mechanics*, 44 (4):493–504. 2009.
- F. González, M. González and A. Mikkola. Efficient coupling of multibody software with numerical computing environments and block diagram simulators. Submitted to *Multibody System Dynamics* in December, 2009.
- F. González, M. A. Naya, A. Luaces and M. González. On the effect of multirate co–simulation techniques in the efficiency and accuracy of multibody system dynamics. Submitted to *Multibody System Dynamics* in March, 2010.