Combining control design tools — from modeling to implementation

Christian Ridderström, Johan Ingvast and Jan Wikander
( chr@md.kth.se, ingvast@md.kth.se and jan@md.kth.se )

Dept. of Machine Design
Royal Institute of Technology
100 44 Stockholm, Sweden

Abstract

Good tools are needed in order to develop robotic systems efficiently. Today, in addition to CAD/CAM, there are tools for model derivation, control design and implementation. There are also tools for exporting models to a control design environment, as well as from control design to implementation (rapid prototyping tools). It is however, still difficult to combine these tools, especially when working with large systems (i.e. a lot of signals and parameters). We have therefore combined, interfaced and augmented some of these tools into a method that bridges the gaps between automatic model derivation and control implementation. Analytically derived functions (from Maple) are used for control design, simulation, visualization, evaluation (MATLAB) and implementation (Real-Time Workshop and xPC Target). The method and tools are illustrated by application to a four-legged robot in both simulation and reality. Generality is also exemplified by applying the method to a simulated SCARA robot.

1 Introduction

When developing robots there is a need for analysis, modeling and control design tools. Figure 1 illustrates a process, where analysis precedes modeling and design. The next step in the process is to simulate and perform experiments, which produce data that need to be visualized and evaluated. To aid us with this process, we would like to have tools that help us with tasks such as model derivation, simulation, evaluation/visualization and control implementation.

This is especially important during the development of complex robots, such as the four-legged walking robot Warp1 [11] we are working with (figure 4). Here, simulations are important to evaluate performance of the robot and controller without risking damage. However, the models are complex and the large amount of states, signals and parameters are difficult to handle. For instance, it is desirable to numerically specify parameters only once, as well as refer to the parameters in a similar way in different tools.

Multibody systems can be simulated with graphical tools such as ADAMS, DADS and Envision that use numeric methods. However, there are also tools that produce analytic expressions for analysis and/or (numeric) integration such as the Sophia language [6] for Maple, Mathematica and Macsyma. Another analytic tool is described by Murray et al. [7] for Mathematica. In contrast to the CAD tools above, these analytic tools use textual input to describe the model. However, Hardell [3] has worked on using CAD models as input to Sophia. An advantage with analytic methods is the possibility to derive expressions representing forward kinematics and linearized models. These can then be used to design and test controllers in simulation using tools such as MATLAB/Simulink and MATRIXx. Finally, to automatically implement and test the controller, there are rapid prototyping tools such as dSPACE, WinCon, xPC Target and OPAL-RT. There are thus a lot of tools that aid the designer, but so far only the rapid prototyping tools that have been well integrated. Furthermore, with all of these tools it is tedious to evaluate and and keep track of data from complex systems.

To automate and speed up the design process, we have combined and interfaced several of the above tools. Specif-

1 Just the rigid body model of Warp1 consists of 36 differential equations and about 170 parameters.
ically, modeling is first simplified by using a computer algebra system to automatically derive an analytic rigid body model of the robot together with other useful expressions (forward kinematics, Jacobians etc). The designer then uses these to perform additional modeling, control design and simulations. Finally, rapid prototyping tools are used to automatically implement the controller. The evaluation is aided by using the derived expressions. Additionally, specific methods are used to handle the problem of a large amount of signals and parameters. This includes: ensuring that the correct numeric parameter values are used for the correct function; and extracting the correct sub-set of signals from a large set of signals, within the controller and when evaluating data.

The tools and methods are described in section 2. They have been tested and applied to the walking robot Warp1 (figure 4), as well as to a simulated SCARA robot. Warp1 is used as example when different types of models (section 3) and results (section 4) are discussed. Finally there is a general discussion of the tools and methods.

2 Development tools and methods

Figure 2 illustrates the tools used in the method and figure 3 illustrates how they have been combined. The computer algebra system Maple is used with an implementation of the Sophia language [6] and the macro package Exmex [5] to derive and export analytical models and expressions for multibody systems. These models and expressions are used to build and simulate controllers in MATLAB, an integrated environment that combines numeric computation with graphics and a high-level programming language. Specifically, the graphical block diagram interface of Simulink (a MATLAB toolbox) is used. It provides a simulation and prototyping environment for modeling, simulating and analyzing dynamic systems. The Real-Time Workshop (RTW, another MATLAB toolbox), is then used to automatically convert the controller block diagrams into C-code. Finally, a real-time kernel and I/O-drivers from the xPC Target (xPC) toolbox2 are used with the output from RTW to automatically build and download the controller to the target computer. The target computer is a standard PC, that executes the controller code. Furthermore, xPC supplies an interface to perform experiments and sample data directly through MATLAB/Simulink.

The use of the tools and methods will be now described in detail.

2.1 Analytical derivation

The Sophia language is based on Kane’s method [4] and is used to derive the kinematic and dynamic equations of general multibody systems. Closed kinematic chains are handled using velocity constraints to reduce the number of dynamic differential equations. Vector objects are easy to work with, since they contain both components and a coordinate system reference. This means that once the user has specified how different coordinate systems are related, vectors can be added with a simple operation, as illustrated below.

```plaintext
#Define coordinate system relationship
chainSimpRot([[A,B,3,alpha]]):
  r1 := A &ev [a,0,0]: #Define vector r1
  r2 := B &ev [b,0,0]: #Define vector r2
  r1 &++ r2; #Add the vectors
```

As an example, it takes about 30 lines of Sophia code to derive the equations for a SCARA robot consisting of three rigid bodies, connected by one linear joint and two revolute joints (figure 4).

Exporting models and expressions The Exmex package is used to export expressions and models as C-files in Simulink’s S-function format; therefore automatically usable with RTW and xPC. Parameters of the models and

---

2 The xPC toolbox was known as RealLink/32 before it was acquired by Mathworks.
expressions are given for two reasons as arguments to the S-function in Simulink:

- To keep the S-functions as general as possible.
- To be able to define all numeric parameter values in one place, i.e. MATLAB.

To handle systems with a large amount of parameters, we created an encapsulation to Exmex that automatically generates one auxiliary function (MATLAB .m-file) for each S-function. The function returns an argument vector with the required numerical values in the correct order. Numeric parameter values are obtained by letting each name correspond to a field of a structure argument supplied to the auxiliary function (or even simpler, as a variable in MATLAB’s base workspace). Note that this makes it easy to use the same names in both Maple and MATLAB.

### 2.2 Model and control assembly

The controller and models are built in Simulink. An analytically derived model or expression is included by adding a standard *S-function block* and specifying the name of the S-function and its corresponding auxiliary function in that block. In addition, the S-functions must be compiled before they can be used in simulation.

The robot hardware block in the controller block diagram is an instance from a library, either a simulation library or an implementation library. This is important since it allows switching between simulation and implementation by just changing the library search path. Similarly, it makes it easy to use robot models of different complexity with the same controller. However, this requires the structures of the libraries to be identical.

For a complex system with several sub-systems that are structurally identical, it also speeds up the design by letting them be instances of the same reference system from a library. However, this also causes problems with keeping track of all the signals from the different sub-systems.

The signals are therefore combined into one or several virtual busses, from which the desired signals later are automatically selected based on their name. It turned out that the standard Simulink blocks could not be used for this purpose and thus new functions were created that recursively search the virtual bus to find the desired components.

If the control system is intended for distribution, it is also helpful to create busses similar to the expected real communication busses. Then it is possible to distribute the control system by first breaking the busses with communication blocks representing for instance a CAN-bus and then implement the different parts of the controller on different target computers.

### 2.3 Visualization and evaluation

Simple 3D objects such as lines and plane surfaces are animated in Simulink to visualize the robot and environment (figure 4). Another tool, Envision/IGRIP\(^3\), has been used for better graphics and more complicated environments. A drawback with this tool is that the robot model has to be manually entered by the user. The Simulink animation on the other hand, is based on data from functions derived in Maple that produce point positions (e.g. joint positions) that lines/surfaces are drawn between. This

\(^3\)Envision/IGRIP is actually an integrated environment for robot design, simulation and off-line programming, but was in this context only used for visualization and environment modeling.
means that changing limb kinematics, or even the number of limbs, is automatically reflected.

Similarly, other functions that are useful for visualization and evaluation can also be derived for different number of legs and joints, automatically. Some examples are expressions that draw lines animating ground forces, joint torques and trunk velocities.

The large amount of signals can also result in a lot of logged data (either from simulation or experiments). If each signal is logged separately, the user has to keep track of a lot of variables. On the other hand, if the data is just lumped together, it becomes tedious and error prone for the user to keep track of what component of the data vector that corresponds to what signal. We therefore expanded the functionality that allows us to select the desired signal within a combined Simulink bus, to do the same from the command line. For instance, the commands

```matlab
simData = expdata('trot', 0.01, [-10 0]);
expData = expdata(target, 0.01, [-10 0]);
```

acquires the latest ten seconds of output data at a sample rate of 0.01 seconds from a simulation (first line) and a real experiment (second line). The argument ‘trot’ refers to a Simulink model, whereas target refers to a xPC-object representing the target computer that has just tested the same control model against the real robot. Different signals are then easy to access as members of the `simData` or `expData`, for instance as

```matlab
time = expData.t;
```

### 2.4 Control implementation and hardware

RTW is used together with xPC to automatically generate the controller C-code and build it. Note that control expressions generated from the analytical model and used in the simulation are automatically included. In fact, we use the same Simulink model file for both simulation and implementation.

The resulting controller C-code, with a small real-time kernel, is then ready for execution on the target computer. The control program uses xPC’s I/O libraries to communicate with I/O-cards in the target PC. Using communication cards (for CAN-busses etc) it becomes possible to work with distributed systems. In that case, only encapsulation blocks in Simulink have to be created that convert between Simulink signals and the bus communication protocol.

It is not necessary to use the xPC toolbox with this method. There are also other toolboxes (Real-Time Windows etc) and products (dSPACE etc) that would work because they all use the S-function format as an interface.

### 2.5 Experiments

Experiments are performed using xPC and Simulink’s external simulation mode, i.e. without leaving the Simulink environment. This allows us to create a crude graphical “User Interface”. Since the experiments are performed from Simulink, logging and retrieving data to MATLAB is easy.

### 3 Models

This section describes the different models used for Warp1. The complete system has models of the robot (i.e. its rigid bodies), environment, actuators, sensors and control implementation. As seen from the rigid body model, the ground and the actuators apply external forces and torques. The joint friction, limit stops and compliance was included in the actuator model.

#### 3.1 Rigid body mechanical model

The rigid body model of Warp1 assumes that the trunk, thighs, shanks and feet are rigid bodies, connected by revolute joints. In the hip there is both an abduction/adduction and flexion/extension joint and in the knee there is a flexion/extension joint.

The script that derives the rigid body model is specialized for Warp1. To speed up the derivation, it is assumed that the legs have identical kinematics, but not identical parameters such as link lengths and inertias. However, the script is not limited to Warp1 and can actually generate the rigid body model for a machine with \( L \) limbs, where each limb has 1–4 joints. This assumes limb kinematics similar to that of Warp1, but it is only necessary to change two or three rows, to use another sequence of rotational joints. Inserting linear joints is also easy.

This script is in principle very similar to the script used for a SCARA robot. It is however about ten times as large, partly due to the need to handle a variable (and therefore larger) structure, and partly to be user friendly by using “human like” names such as hip, thigh, knee and shank for different parts of the limbs.

#### 3.2 Environment model

Terrain geometry is described using plane surfaces. We assume that the ground applies forces only on the supporting feet, no torques. This is motivated by a small footpad radius (about 2 cm).

The ground force is calculated as a function of penetration depth and velocity. A linear spring/damper model is easy to use, but can result in discontinuous impact forces,
due to damping and nonzero impact velocities. For the force perpendicular to the surface, we therefore use a modified version of a spring/damper model [8], when the foot penetrates the ground

\[
F_z = - k z^n - d z \dot{z} \cdot \text{Heaviside}(\dot{z})
\]

where \( k \) and \( d \) are the stiffness and damping coefficients, and \( z \) is the penetration depth. For the \( x \) and \( y \)-components, a smoothed viscous friction with a maximum limit based on the vertical force is used as in [1]

\[
F_x = -\frac{2\gamma F_z}{\pi} \arctan\left( \frac{d\pi}{2\gamma F_z} \dot{x} \right)
\]

where \( \gamma \) is the friction coefficient and \( \dot{x} \) is the velocity in the \( x \)-direction. \( F_y \) is calculated similarly.

### 3.3 Actuator model

Each joint is actuated by one DC motor. The hip and knee flexion/extension motors are attached to one link and their rotor shafts connect to the next link via a harmonic drive gear, a wire gear and a rubber torsion spring. The configuration is similar for the hip abduction/adduction joint, except that there is no rubber spring.

Different actuator models are used (trading simulation speed against accuracy), ranging from ideal torque sources up to and including the DC motor’s electrical and mechanical dynamics, viscous damping and linear spring/damping characteristics. Unmodeled (potential) problems include backlash, wire pulley dynamics and slippage as well as the motor’s temperature dependence. Experience has shown that the motor parameters can be quite temperature dependent.

The rotational acceleration of the link that the stator is attached to is not modeled and the spring model is used to map spring deformation to output torque. This makes it easy to add the actuator dynamics as extra states to the system, outside the rigid body model.

### 3.4 Sensor model

The robot has dual encoders to measure joint and rotor shaft angles. Motor current sensors are used to estimate torques. These sensors are usually not modeled. A combination of inclinometers, rate gyros and accelerometers mounted on the trunk are used to estimate orientation.

### 3.5 Control implementation model

The control system is distributed to two target PC:s that communicate using a CAN-bus. One target PC is connected to four intelligent I/O-nodes (ACN:s); one for each leg. Separate CAN-busses are used for the individual legs to achieve a high control frequency (500 Hz – 1000 Hz). This structure reduces complexity by allowing identical and simple CAN protocols, as well as identical ACN software, to be used on each leg. The second target PC is used similarly with a fifth I/O-node connected to the trunk attitude sensors.

In order to simulate a limited control frequency, we have added sampling delays in sensor sampling and actuator commands. Similarly, signals are delayed to reflect the fact that we do control over a CAN bus, but we do not model the corresponding jitter.

The next section will present some results from using these models with the previously described tools.

### 4 Results

A typical design process would now be as follows. First the robot model is derived analytically (using the automated script in this case, section 3.1) and then exported to Simulink, where the model is tested with local joint controllers. The next step is to test the joint controllers against the real robot, thereby verifying that both the method and

---

4The spring was added to reduce shocks and also to be able to increase power efficiency.

---

5Further testing can be done by analytically reducing the model and study special cases, or by using the automated script to reduce the number of limbs and degrees of freedom per limb.
the robot works properly. Then, the designer gets an idea for a controller (such as the one in section 4.1) and returns to the analytical environment to derive functions for the controller. These are also exported to Simulink where they are used to build a controller that is tested in simulation and then against the robot.

Table 1 illustrates the cost of evaluating some of these functions, after Maple has used intermediate variables to minimize the cost. The expressions \( B_T H P_1 \) and \( B_V H P_1 \) denote the vector components in a trunk oriented coordinated system, \( B \), of a foot’s position and velocity with respect to the corresponding hip. The same components evaluated in an inertial coordinate system, \( N \), are denoted \( N_T H P_1 \) and \( N_V H P_1 \). They are of course much more complex since the orientation of the trunk now matters. An even more complex example is \( N_T C M \), the robot’s centre of mass; it is useful for performance evaluation and analysis of walking robots. However, on the target PC (a Pentium 350 MHz), even this expression only takes about 60 \( \mu \)S to evaluate.

Finally, the field calculations represent the cost of doing one evaluation of the robot’s rigid body differential equations in order to solve the differential equations implicitly. It takes about 20 seconds to execute the script that derives these equations on a Sun Ultra/60. However, poorly chosen generalized coordinates and adding complicated constraints can increase this time significantly.

Several levels of details are possible in the simulation, from ideal actuators, to actuators with full electric and mechanic dynamics. Simulating one second of walking takes about four seconds on a Sun Ultra/60, when assuming “ideal actuators”. Including actuator dynamics (electric and mechanic) approximately doubles the required time, and calculating all visualization and evaluation expressions during simulation increases the required time with about 25%.

### 4.1 Simple controller

The controller\(^6\) described here is only intended to show our models and methods. It works however, and produces a surprisingly stable trot gait in both simulation and experiments.

A simple combined stiffness/damping controller is used for each leg, similar to for instance Virtual Actuator Control [9]. The joint torques for leg \( l \) are denoted \( \tau_l \).

\[
\tau_l = J_l^T \cdot \left( P_l \cdot \left( B_T H P_1 \cdot ref \right) - B_T H P_1 \right) + D_l \cdot \left( B_V H P_1 \cdot ref \right)
\]

where \( P_l \) and \( D_l \) are matrices with stiffness and damping coefficients, \( B_T H P_1 \) are the components of the vector from the leg’s hip to its foot in the trunk coordinate system, \( B \), and the corresponding velocity components are denoted \( B_V H P_1 \).

The reference trajectory (position: \( B_T H P_1 \cdot ref \), velocity: \( B_V H P_1 \cdot ref \)) is elliptic, where diagonal legs are in phase and lateral legs are half a cycle out of phase, i.e. a trot gait. Figure 5 shows two experiments at different periods. In the first experiment (period: 1 s), tracking is good and there is little difference between real and simulated data. The small shift down to the left is caused by gravity. In the second experiment, tracking is worse because the actuator velocities saturate. There is also a larger discrepancy between the real and simulated data, suggesting parameter errors or unmodeled effects close to the actuators’ performance limits.

Note that since the expressions \( B_T H P_1 \) and \( B_V H P_1 \), and \( B_V H P_1 \)’s Jacobian, \( J_l \), are automatically generated, each leg controller should work even if the legs’ kinematics are changed in the analytical model. A mechanical designer could therefore easily test different ideas for leg kinematics (using an actual controller) with very little extra work. In practice, the real hardware would of course have to be changed similarly, and there are limits to how the kinematics could be changed. For instance, it is not clear what would happen if extra degrees of freedom were added, since this controller only uses feedback on the foot’s position (not its orientation).

---

\(^6\)A similar controller for a leg prototype has previously been used with satisfactory results on a treadmill [2].
5 Discussion

In this article we have combined and interfaced several tools to bridge the gap between automatic model derivation and control implementation. This was illustrated by simulation and application on our four-legged robot. The automatic model derivation is capable of producing analytic models for entire classes of robots (not just a specific robot type). Furthermore, the designer is aided by being able to analytically derive, generate and use various expressions.

The rigid body model derived in Maple is not only used for simulation, but also in the control design. Yet another use is debugging, since in Maple, the analytic model can be investigated by reducing the equations through linearization or by looking at special cases.

It is very useful for the designer to be able to easily derive expressions from the model. In our application, we automatically derived and exported expressions that were used in the controller and for performance evaluation, but we also see the possibility to use this for filter and mechanical design. The ability to easily generate and export expressions saves the designer a lot of time, compared to deriving them by hand and then manually implementing them in the controller or simulator. Additionally, changes in the robot’s structure are automatically reflected in other tools for purposes such as visualization, simulation and expressions for control design.

The generality is unfortunately not supported in full by all tools. For instance, Simulink’s graphical interface makes it difficult to design more generally applicable controllers. The problem of too many signals was handled by creating signal busses from which we used special methods to extract and use the desired components.

Another aspect is optimality with respect to speed and size. We have focused on doing the tools generic, not optimal. Since expressions are exported as C-code, optimization by Maple and the compiler(s) will affect simulation and implementation performance. Similarly, the performance of RTW and xPC’s real-time kernel will also affect the implementation performance, i.e. the maximum frequency with which we can execute the controller. However, with a target PC using a Pentium 350 MHz CPU, executing the controller from section 4.1 only takes about 140 μs.

One improvement to this chain of tools would be a closer connection to other CAD tools, in order to extract parameter values and perhaps also the robot structure. Another improvement would be adding specific support for standard control methods such as linear state feedback controllers.

In this paper, we worked with a four-limbed robot and a simulated SCARA robot, but we believe this method can easily be extended to other structures. We found the method to be useful and believe it has advantages over other tools for robot simulation, control and implementation. Furthermore, we are convinced that these tools and methods will be very valuable for future work, validating models and developing more advanced controllers.

To summarize the method, it uses Maple/Sophia to derive models and expressions for analysis, control design and simulation in MATLAB/Simulink. RTW and xPC are then used to implement the controller and perform experiments. Note that no low-level coding is necessary, once the tools have been integrated and combined. As a final example, we had an idea for a simple trunk attitude controller that distributes desired vertical leg forces based on the estimated attitude. Reusing parts from the simple controller previously described, it took less than 90 minutes to go from idea to simulation and perform the first experiments.

Acknowledgments

This work was supported by the Foundation for Strategic Research through its Centre of Autonomous Systems at the Royal Institute of Technology, Stockholm, Sweden.

References