The Pinocchio C++ library – A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives

Justin Carpentier, Guilhem Saurel, Gabriele Buondonno, Joseph Mirabel, Florent Lamiraux, Olivier Stasse, Nicolas Mansard

The Pinocchio C++ library

A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives

Justin Carpentier, Guilhem Saurel, Gabriele Buondonno, Joseph Mirabel, Florent Lamiraux, Olivier Stasse and Nicolas Mansard

Abstract— **We introduce Pinocchio, an open-source software framework that implements rigid body dynamics algorithms and their analytical derivatives. Pinocchio does not only include standard algorithms employed in robotics (e.g. forward and inverse dynamics) but provides additional features essential for the control, the planning and the simulation of robots. In this paper, we describe these features and detail the programming patterns and design which make Pinocchio efficient. We also offer a short tutorial for easy handling of the framework.**

I. Introduction

Rigid Body Dynamics is a very useful tool in robotics. Although the theory dates back to the 18th century [1], current algorithms have been revisited recently [2]. They allow to compute in an efficient way both the inverse and forward dynamics of rigid body systems. These two functions are indeed fundamental for both the control and the simulation of robotic systems. Being able to compute them in a fast and accurate manner is of paramount importance in order to correctly plan and control the motion of complex systems such as humanoid robots or quadruped robots.

Starting from the late 80's, the first implementations were mostly based on code-generation: a meta-program first generates some source code dedicated to a specific robot model given in input. The source code is then compiled into an object code where model parameters are hard coded. Many open and closed source frameworks follow this philosophy: SD/Fast [3], ROBOTRAN [4], HuMAnS [5], Symoro [6], METAPOD [7], RobCoGen [8] just to name a few. This approach has the advantages of allowing simplifications of math expressions therefore avoiding unnecessary memory allocations or use of temporary variables. But at the same time, this approach suffers from a lack of flexibility: the code must be regenerated from scratch if any slight modification of the robot model occurs.

As more computational resources have been made available on desktop computers, another paradigm has emerged which consists in generating once for all a compiled library able to load at runtime a description file of the robot model (kinematic chain, mass distribution, etc.). Many recent frameworks implement this paradigm: OpenHRP [9], RBDL [10], Drake [11], Bullet [12], SymBody [13], DART [14], RigidBodyDynamics.jl [15], etc. This second approach is more versatile, allowing for instance to modify at runtime the dynamical properties of the model or to populate the model with new features. But it implies a loss in computational efficiency, even-though some of these frameworks show execution times close to code generation [7], [10].

In this paper, we introduce a new rigid body dynamics framework called Pinocchio. Unlike all other existing frameworks, Pinocchio follows the two aforementioned paradigms all at once. Pinocchio is a dynamic library able to load at runtime any robot model. The efficiency is then comparable to other dynamic frameworks [7], [10] and nearly matches code-generation frameworks. It is able to generate robot-specific source code (also at runtime). In that case, it outperforms any other existing frameworks.

In Section II, we give a global overview of the framework. The details of the code implementation are provided in Section III. An introductory tutorial to get started with Pinocchio is given in Section IV. The performances of Pinocchio against similar frameworks are reported in Section V. In Section VI, a summary of projects which are based or make use of Pinocchio is provided. Finally, Section VII concludes the paper and draws the future roadmap of Pinocchio.

II. Main features of Pinocchio

Pinocchio has been written in C++ for efficiency reasons and uses the Eigen library [16] for linear algebra routines. It comes with Python bindings for easy code prototyping. We describe here the main features implemented in Pinocchio.

A. Spatial algebra

Spatial algebra [2] is a mathematical notation commonly employed in rigid body dynamics to represent and manipulate physical quantities such as velocities, accelerations and forces. Pinocchio is based on this mathematical notation. Dedicated classes are provided to represent coordinate transformations in the 3D Euclidean space (named SE3), spatial motion vectors (Motion), spatial force vectors (Force), and spatial inertias (Inertia). Along with the available methods, this endows Pinocchio with an efficient software library for spatial algebra calculations.

The authors are with the Gepetto Team, Robotics Department, Laboratoire d'Analyse et d'Architecture des Systèmes, CNRS and the Université de Toulouse, Toulouse, France.

Corresponding author: Justin Carpentier (justin.carpentier@laas.fr)

B. Model and data

A fundamental paradigm of Pinocchio is the strict separation between *model* and *data*. By *model*, we mean the physical description of the robot, including kinematic and possibly inertial parameters defining its structure. This information is held by a dedicated class which, once created, is never modified by the algorithms of Pinocchio. By *data*, we mean all values which are the result of a computation. *Data* vary according to the joint configuration, velocity, etc... of the system. It contains for instance the velocity and the acceleration of each link. It also stores intermediate computations and final results of the algorithms in order to prevent memory allocation. With this splitting, all the algorithms in Pinocchio follow the signature:

```
algorithm(model, data, arg1, arg2, ...)
```

where `arg1, arg2, ...` are the arguments of the function (e.g. configuration or velocity). Keeping model and data separated reduces memory footprint when performing several different tasks on the same robot, notably when this involves parallel computation. Each process can employ its own data object, while sharing the same model object. The fact that a model object never changes within an algorithm of Pinocchio enhances the predictability of the code.

A model can be created using the C++ API or loaded from an external file, which can be either URDF, Lua (following the RBDL standard) or Python.

C. Supported kinematic models

Within a model, a robot is represented as a kinematic tree, containing a collection of all the joints, information about their connectivity, and, optionally, the inertial quantities associated to each link. In Pinocchio a joint can have one or several degrees of freedom, and it belongs to one of the following categories: **Revolute** joints, rotating around a fixed axis, either one of X, Y, Z or a custom one; **Prismatic** joints, translating along any fixed axis, as in the revolute case; **Spherical** joints, free rotations in the 3D space; **Translation** joint, for free translations in the 3D space; **Planar** joints, for free movements in the 2D space; **Free-floating** joints, for free movements in the 3D space. Planar and free-floating joints are meant to be employed as the basis of kinematic tree of mobile robots (humanoids, automated vehicles, or objects in manipulation planning). More complex joints can be created as a collection of ordinary ones through the concept of **Composite** joint.

D. Dealing with Lie group geometry

Each type of joints is characterized by its own specific configuration and tangent spaces. For instance, the configuration and tangent spaces of a revolute joint are both the real axis line $\mathbb{R}$, while for a Spherical joint the configuration space corresponds to the set of rotation matrices of dimension 3 and its tangent space is the space of 3-dimensional real vectors $\mathbb{R}^3$. Some configuration spaces might not behave as a vector space, but have to be endowed with the corresponding integration (exp) and differentiation (log) operators. Pinocchio implements all these specific integration and differentiation operators.

E. Geometric models

Aside the kinematic model, Pinocchio defines a geometric model, i.e. the volums attached to the kinematic tree. This model can be used for displaying the robot and computing quantities associated to collisions. Alike the kinematic model, the fixed quantities (placement and shape of the volums) are stored in a *GeometricModel* object, while buffers and quantities used by associated algorithms are defined in a GeometricData object. The volumes are represented using the FCL library [17][1]. Bodies of the robot are attached to each joint, while obstacles of the environment are defined in the world frame. Collision and distance algorithms for the kinematic trees are implemented, based on FCL methods.

F. Main algorithms

The implementation of the basic algorithms, including all those listed in this section, is recursive. The recursive formulation allows the software to avoid repeated computations and to exploit the sparsity induced by the kinematic tree. For the dynamics algorithms, we largely drew inspiration from [2], with slight improvements.

a) ***Forward kinematics:*** Pinocchio implements direct kinematic computations up to the second order. When a robot configuration is given, a forward pass is performed to compute the spatial placements of each joint and to store them as coordinate transformations. If the velocity is given, it also computes the spatial velocities of each joint (expressed in local frame), and similarly for accelerations.

b) ***Kinematic Jacobian:*** the spatial Jacobian of each joint can be easily computed with a single forward pass, either expressed locally or in the world frame.

c) ***Inverse dynamics:*** the Recursive Newton-Euler Algorithm (RNEA) [18] computes the inverse dynamics: given a desired robot configuration, velocity and acceleration, the torques required to execute this motion are computed and stored. The algorithm first performs a forward pass (equivalent to second-order kinematics). It then performs a backward pass, computing the wrenches transmitted along the structure and extracting the joint torques needed to obtain the computed link motions. With the appropriate inputs, this algorithm can also be employed to compute specific terms of the dynamic model, such as the gravity effects.

d) ***Joint space inertia matrix:*** the Composite Rigid Body Algorithm (CRBA) [19] is employed to compute the joint space inertia matrix of the robot. We have implemented some slight modifications of the original algorithm that improve the computational efficiency.

e) ***Forward dynamics:*** the Articulated Body Algorithm (ABA) [20] computes the unconstrained forward dynamics: given a robot configuration, velocity, torque and external forces, the resulting joint accelerations are computed.

[1]Pinocchio indeed uses a fork of FCL 0.3.1 version.

*f) **Additional algorithms:*** beside the algorithms above, other methods are provided, most notably for constrained forward dynamics, impulse dynamics, inverse of the joint space inertia [21] and centroidal dynamics.

G. Analytical derivatives

Beside proposing standard forward and inverse dynamics algorithms, Pinocchio also provides efficient implementations of their analytical derivatives [22]. These derivatives are for instance of primary importance in the context of whole-body trajectory optimization or more largely, for numerical optimal control. To the best of our knowledge, Pinocchio is the first rigid body framework which implements this feature natively.

H. Automatic differentiation and source code generation

In addition to analytical derivatives, Pinocchio supports automatic differentiation. This is made possible through the full *scalar* templatization of the whole C++ code and the use of any external library that does automatic differentiation: ADOL-C [23], CasADi [24], CppAD [25] and others. It is important to keep in mind that these automatic derivatives are often much slower than the analytical ones.

Another unique but central feature of Pinocchio is its ability to generate code both at compile time and at runtime. This is achieved by using another external toolbox called CppADCodeGen[2] built on top of CppAD [25]. From any function using Pinocchio, CppADCodeGen is able to generate on the fly its code in various languages: C, Latex, etc. and to make some simplifications of the math expressions. Thanks to this procedure, a code tailored for a specific robot model can be generated and used externally to Pinocchio.

III. What Makes Pinocchio Fast

In this section, we detail the programming paradigms that we have implemented in Pinocchio, to make the framework both efficient and versatile.

A. Handling the sparsity

Each joint by definition constrains the motion between two bodies to be restricted to some particular directions of movement. This particularly means that each joint can be endowed with its own specific operators and state representations, in order to achieve minimal memory print and number of computations. For instance, the joint Revolute transformation is represented by a single scalar value, the rotation around its axis, while the joint Spherical transformation is encoded as a rotation matrix. A similar observation can be made for the other spatial quantities that characterize the joint state, such as spatial velocities or accelerations, joint constraint, etc. Hence, each joint in Pinocchio is endowed with its own sparse description of the spatial quantities. In combination with an overloading of the spatial operators, this allows us to adequately exploit the sparsity inherent to each joint at the computational level.

[2]https://github.com/joaoleal/CppADCodeGen

B. Static polymorphism

The concept of polymorphism then enables us to adequately exploit the sparsity induced by the joints. Pinocchio classes make extensive use of inheritance and polymorphism. For instance, all different joint models are implemented as subclasses of `JointModelBase`, which defines the common API for all the joints. Methods and data structures are then specialized in each joint model class.

In Pinocchio, we chose to implement this behavior through *static polymorphism*, in contrast with *dynamic* polymorphism, the traditional way of implementing polymorphism in C++, by means of virtual methods. In dynamic polymorphism, when a method is called, the object class is deduced at runtime, and the appropriate method is then executed. This has the main drawback of breaking the prediction mechanisms of modern CPUs. In static polymorphism, instead, the appropriate function is selected directly at compile time. The adoption of this paradigm improves efficiency in many ways. In the first place, the double redirection which is typical of dynamic polymorphism is avoided, as well as the runtime class deduction. In the second place, since class information is known at compile time, the compiler is allowed to optimize the code to make it more efficient.

Static polymorphism is implemented through the so-called Curiously Recurring Template Pattern (CRTP), which is at the core of our framework. This is also the design pattern employed in the Eigen library, greatly contributing to its performances and versatility. We now introduce the concept of CRTP with a simple example which depicts the architecture of the joint classes in Pinocchio. According to this design pattern, the joint model base class `JointModelBase` that defines the common methods and attributes for all the joints, is templated by its child class:

```
template<typename Derived>
struct JointModelBase<Derived> {
  void calc(q,v)
  {
    ⟨call calc method of Derived class⟩
    static_cast<Derived*>(this)->calc();
  }
};
```

Then all the joint model classes inherit this base class, as follows:

```
struct JointModelRevolute
       : JointModelBase<JointModelRevolute> {
  void calc(q,v)
  { ⟨do specific computations⟩ }
};
```

In this way, `JointModelBase` is employed to define the prototypes of all joint model-specific operators, which are then implemented in the child classes. In turn, this allows developers to write generic code which works for all subclasses, by simply writing templated functions. For instance, all algorithms in Pinocchio are written as a sequence of steps to execute over the joint models contained in the `model`. A single step may be implemented as

```
template<typename Derived>
void step(JointModelBase<Derived> joint, arg1, ...)
{
  joint.calc(arg1,arg2); // Calling Derived::calc
  ...
}
```

where function `step` calls the method `calc` defined in `JointModelBase<Derived>` which directly redirects to the method `calc` of the Derived class performing the computations. Since the value of `Derived` is known at compile time, modern compilers are able to remove this level of indirection and directly call the method in the Derived class. The same happens with all other similarly implemented methods employed within `step`. A different version of `step` is then compiled for each joint class, each statically linked to the joint-specific methods and avoiding the use of dynamic redirection.

This enhanced performance comes with a loss of flexibility. From a conceptual point of view, when using this coding paradigm, two different derived classes do not inherit from the same base class. For instance, `JointModelBase<JointModelRevolute>` is a base class of `JointModelRevolute`, while `JointModelBase<JointModelPlanar>` is a base class of `JointModelPlanar`. This means that `JointModelRevolute` and `JointModelPlanar` cannot be cast to a same parent class, as usually done with dynamic polymorphism. Therefore, it is neither possible to create a vector of `JointModelBase` objects, nor possible to have joints whose type is unknown at compile time.

To recover the flexibility allowed by dynamic polymorphism, we resort to the concept of *variant*. A variant is a class which can be used to represent any type of a finite predetermined set of types. Most importantly, a variant keeps runtime information about the represented type. In Pinocchio, we define a variant called `JointModelVariant`, able to represent any joint. Joints are stored within a Pinocchio model as a collection of variant objects. In this way, when an algorithm is executed, the appropriate steps are selected at runtime, depending on the type of each joint. However, as explained above, each individual step is fully optimized for the specific joint type at hand.

Thanks to the combination of CRTP and of the variant paradigm, the code flexibility is recovered, while for each algorithm the overall runtime overhead due to class redirection is reduced to the minimum.

IV. Getting Started with Pinocchio

Pinocchio is currently supported for most Linux distributions and also Mac OS X, with plans to release Windows versions soon. The project is fully open-source under the LGPL-3 license and is currently hosted on the following GitHub repository

```
github.com/stack-of-tasks/pinocchio
```

A. Documentation and tutorials

Documentation for Pinocchio is available within the package, as well as on the GitHub project page; the Python interface is fully documented, while the C++ documentation is still work in progress. Benchmark and unit tests are all available within the package. Tutorials for Pinocchio can be found in a dedicated GitHub repository

```
github.com/stack-of-tasks/pinocchio-tutorials
```

The tutorials are mostly in Python and are especially suited as a support to educators in robotics classes.

B. Installation

Pinocchio releases are managed by robotpkg, a package manager tailored for robotics software, available on most Unix and BSD platforms. Details on the installation procedure can be found on the GitHub project page. Here, we only provide a sketch.

On Ubuntu, the software binaries of the packages managed by robotpkg are directly available through the robotpkg apt repository. After adding the repository to the list of available sources, Pinocchio, its Python bindings and all the required dependencies can be simply installed with

```
apt-get install robotpkg-py27-pinocchio
```

The tutorials include details about the installation procedure, as well as a script which automatically performs all the installation steps.

On Mac OS X, installation of Pinocchio through the Homebrew package manager is supported. It is sufficient to issue the following commands

```
brew tap gepetto/homebrew-gepetto
brew install pinocchio
```

Another option is to install the framework directly from the source code, which can be downloaded from the official GitHub repository of Pinocchio.

C. Small example: running RNEA on the ATLAS robot

In the following, we demostrate how Pinocchio can be used in Python to compute the inverse dynamics of the humanoid robot Atlas on random input values.

```
import pinocchio as se3, numpy as np
root  = se3.JointModelFreeFlyer()
model = se3.buildModelFromUrdf('atlas.urdf',root)
data  = model.createData()

qmax = np.matrix(np.full([model.nq, 1], np.pi))
q    = se3.randomConfiguration(model,-qmax,qmax)
v    = np.matrix(np.random.rand(model.nv,1))
a    = np.matrix(np.random.rand(model.nv,1))
tau  = se3.rnea(model,data,q,v,a)
```

V. Results

A. Setup

The performances of Pinocchio are compared with four other libraries: RBDL [10], Orocos KDL [26], METAPOD [7] and RigidBodyDynamics.jl [15]. The 3 main algorithms are tested: RNEA, CRBA and ABA, over 8

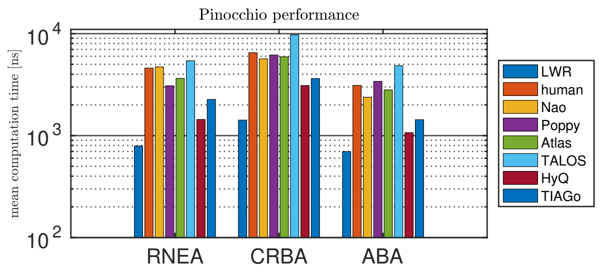

Fig. 1. Performance of Pinocchio on different robots and algorithms

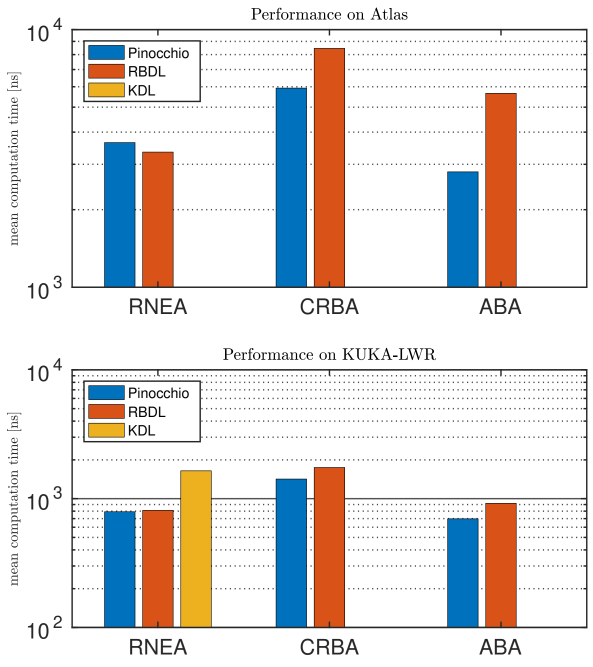

Fig. 2. Comparison of Pinocchio versus KDL and RBDL using manipulator (top) and humanoid (bottom) models.

different robot models[3]. Depending on the library, only some functionality can be tested. ABA is not implemented by RigidBodyDynamics.jl and METAPOD. KDL provides an earlier version of ABA but no CRBA and only support kinematic chain and no tree. Each test case is run 100 000 times with randomized input values on three modern CPU and the average is plotted.

B. Results

Absolute performances are plotted in Fig. 1 for the 8 robot models on the 3 algorithms. Pinocchio requires about 1 μs for evaluating the dynamics on manipulator robots and about 3 μs on legged robots. Performances versus KDL and RBDL are reported in Fig. 2 for 2 representative models. Pinocchio RNEA is similar to RBDL but its ABA and CRBA outperforms RBDL.

[3] Tests are runned with the 7-DoF manipulator KUKA LWR 4+ [27], a simple 22-DoF legged humanoid model, the humanoids Nao (www.softbankrobotics.com/emea/en/robots/nao), Poppy (www.poppy-project.org), Atlas (www.bostondynamics.com/atlas) and TALOS [28], the quadruped HyQ [29], and the mobile manipulator TIAGo (www.tiago.pal-robotics.com)

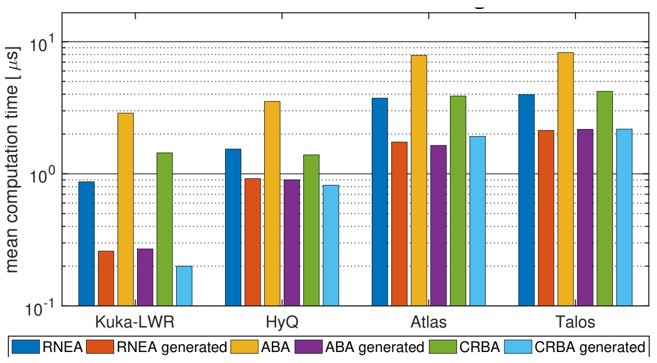

Fig. 3. Performances of code generated by Pinocchio (compared to the dynamic version of the same algorithms).

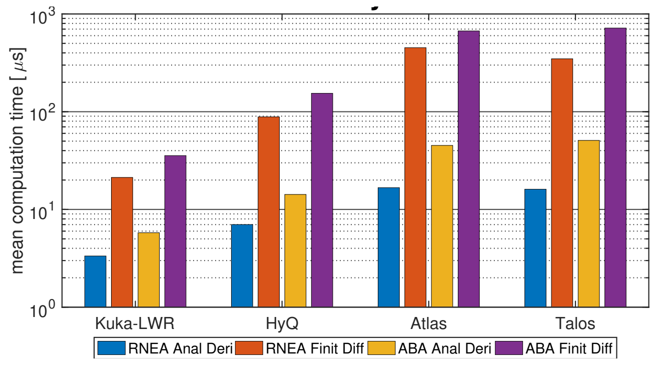

Fig. 4. Performances of the derivatives of RNEA and ABA, for 4 robot models, versus evaluation by finite differences.

These first set of results are obtained using the dynamic algorithms. We now compare these scores with the performances when dedicated source code is generated, in Fig. 3. For simple robot like the LWR, the code generation divides the computation cost by 8. For more complex robots like humanoid or quadruped, it divides the cost by 3. The difference of cost improvement is likely due to caching effects. As a result, it is possible to evaluate the dynamics of complex legged robots in slightly more than $1\mu s$.

Finally, Pinocchio also provides the implementation of the derivatives of RNEA and ABA among others. We report in Fig. 4 the performances of these algorithms when the model is loaded dynamically (it would also be possible to generate dedicated code for these algorithms).

VI. Framework Dissemination

Pinocchio was used in [31] and later in [32], [33] to generate the whole body motion of the HRP-2 robot. This experiment of climbing stairs with multiple contacts has been run on the real hardware about 100 times with a rate of success around 80 %. Thanks to its versatility, it is currently used on the humanoid robot Pyrene (TALOS-01) [28] to perform kinematic and dynamical tasks. It also has been used in [34] to perform 150 RNEA runs in less than 1 ms to implement a dynamical filter. This has been a key point in order to improve the capabilities of the reactive walking pattern generator. It is also used to perform manipulation planning in the Humanoid Path Planner (HPP) software [35] on multiple robots: UR5, PR-2, and Romeo. It has also been recently used inside the HPP framework to plan dynamically feasible contact sequences on HRP-2 [36].

VII. Conclusions

We have introduced Pinocchio, a new, fast and flexible framework that implements rigid body dynamics algorithms and their analytical derivatives. Pinocchio demonstrates equivalent or even better performances than all other similar libraries when used dynamically. The gap is even increased with the support of code generation, where the timings outperforms the state of art.

As next milestones, we plan to integrate in Pinocchio both the model of transmissions (gear, pulley, tendons, etc.) and actuations (pneumatic muscles, biological muscles, electrical motors, etc.) in order, for instance, to take into account their physical effects in the dynamic equations of motion. This will make Pinocchio not only able to deal with robotic systems but also with the simulation of biological systems, that is of primary importance to understand for instance the foundations of anthropomorphic locomotion [37].

Acknowledgement

We first thank all the developers of Eigen, CppAD and CppADCodeGen who provide efficient, high-quality and open source frameworks. We also thank all the contributors who took part, directly or indirectly, to the development of the Pinocchio library.

References

[1] J.-L. Lagrange, *Mécanique analytique*. Académie Royale des Sciences, 1788.

[2] R. Featherstone, *Rigid Body Dynamics Algorithms*. Springer, 2008.

[3] M. G. Hollars, D. E. Rosenthal, and M. A. Sherman, "SD/FAST user's manual," 1991.

[4] N. Docquier, A. Poncelet, and P. Fisette, "Robotran: a powerful symbolic gnerator of multibody models," *Mechanical Sciences*, vol. 4, no. 1, pp. 199–219, 2013.

[5] P.-B. Wieber, F. Billet, L. Boissieux, and R. Pissard-Gibollet, "The HuMAnS toolbox, a homogenous framework for motion capture, analysis and simulation," in *International Symposium on the 3D Analysis of Human Movement*, 2006.

[6] W. Khalil, A. Vijayalingam, B. Khomutenko, I. Mukhanov, P. Lemoine, and G. Ecorchard, "Opensymoro: An open-source software package for symbolic modelling of robots," in *IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pp. 1206–1211, 2014.

[7] M. Naveau, J. Carpentier, S. Barthelemy, O. Stasse, and P. Souères, "METAPOD — Template META-PrOgramming applied to dynamics: CoP-CoM trajectories filtering," in *IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, 2014.

[8] M. Frigerio, J. Buchli, D. G. Caldwell, and C. Semini, "RobCoGen: a code generator for efficient kinematics and dynamics of articulated robots, based on Domain Specific Languages," *Journal of Software Engineering for Robotics (JOSER)*, vol. 7, no. 1, pp. 36–54, 2016.

[9] F. Kanehiro, H. Hirukawa, and S. Kajita, "Openhrp: Open architecture humanoid robotics platform," *The International Journal of Robotics Research*, vol. 23, no. 2, pp. 155–165, 2004.

[10] M. L. Felis, "RBDL: an efficient rigid-body dynamics library using recursive algorithms," *Autonomous Robots*, 2017.

[11] R. Tedrake and the Drake Development Team, "Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems," 2016.

[12] E. Coumans, "Bullet Physics Simulation," in *ACM SIGGRAPH 2015 Courses*, 2015.

[13] M. A. Sherman, A. Seth, and S. L. Delp, "Simbody: multibody dynamics for biomedical research," *Procedia Iutam*, vol. 2, pp. 241–261, 2011.

[14] J. Lee, M. X. Grey, S. Ha, T. Kunz, S. Jain, Y. Ye, S. S. Srinivasa, M. Stilman, and C. K. Liu, "Dart: Dynamic animation and robotics toolkit," *The Journal of Open Source Software*, vol. 3, no. 22, p. 500, 2018.

[15] T. Koolen and contributors, "RigidBodyDynamics.jl," 2016.

[16] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3," 2010.

[17] J. Pan, S. Chitta, and D. Manocha, "Fcl: A general purpose library for collision and proximity queries," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pp. 3859–3866, IEEE, 2012.

[18] J. Y. Luh, M. W. Walker, and R. P. Paul, "On-line computational scheme for mechanical manipulators," *Journal of Dynamic Systems, Measurement, and Control*, vol. 102, no. 2, pp. 69–76, 1980.

[19] M. W. Walker and D. E. Orin, "Efficient dynamic computer simulation of robotic mechanisms," *Journal of Dynamic Systems, Measurement, and Control*, 1982.

[20] R. Featherstone, "The calculation of robot dynamics using articulated-body inertias," *The International Journal of Robotics Research*, 1983.

[21] J. Carpentier, "Analytical inverse of the joint space inertia matrix," tech. rep., Laboratoire d'Analyse et d'Architecture des Systèmes, 2018.

[22] J. Carpentier and N. Mansard, "Analytical derivatives of rigid body dynamics algorithms," in *Robotics: Science and Systems*, 2018.

[23] A. Griewank, D. Juedes, and J. Utke, "ADOL-C: a package for the automatic differentiation of algorithms written in C/C++," *ACM Transactions on Mathematical Software (TOMS)*, vol. 22, no. 2, pp. 131–167, 1996.

[24] J. Andersson, J. Åkesson, and M. Diehl, "CasADi: A symbolic package for automatic differentiation and optimal control," in *Recent advances in algorithmic differentiation*, pp. 297–307, Springer, 2012.

[25] B. Bell, "CppAD: a package for C++ algorithmic differentiation," 2005-2018.

[26] R. Smits, "KDL: Kinematics and Dynamics Library." `http://www.orocos.org/kdl`.

[27] R. Bischoff, J. Kurth, G. Schreiber, R. Koeppe, A. Albu-Schaeffer, A. Beyer, O. Eiberger, S. Haddadin, A. Stemmer, G. Grunwald, and G. Hirzinger, "The KUKA-DLR lightweight robot arm - a new reference platform for robotics research and manufacturing," in *41st International Symposium on Robotics*, June 2010.

[28] O. Stasse, T. Flayols, R. Budhiraja, K. Giraud-Esclasse, J. Carpentier, J. Mirabel, A. D. Prete, P. Souères, N. Mansard, F. Lamiraux, J. . Laumond, L. Marchionni, H. Tome, and F. Ferro, "TALOS: A new humanoid research platform targeted for industrial applications," in *Int. Conf. on Humanoid Robotics (Humanoids)*, 2017.

[29] C. Semini, N. G. Tsagarakis, E. Guglielmino, M. Focchi, F. Cannella, and D. G. Caldwell, "Design of HyQ – a hydraulically and electrically actuated quadruped robot," *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, vol. 225, no. 6, pp. 831–849, 2011.

[30] A. F. Vereshchagin, "Computer Simulation of the Dynamics of Complicated Mechanisms of Robot Manipulators," *Engineering Cybernetic*, no. 6, pp. 65–70, 1974.

[31] J. Carpentier, S. Tonneau, M. Naveau, O. Stasse, and N. Mansard, "A versatile and efficient pattern generator for generalized legged locomotion," in *IEEE/RAS Int. Conf. on Robotics and Automation (ICRA)*, 2016.

[32] J. Carpentier, R. Budhiraja, and N. Mansard, "Learning feasibility constraints for multi-contact locomotion of legged robots," in *Robotics: Science and Systems*, p. 9p, 2017.

[33] J. Carpentier and N. Mansard, "Multi-contact locomotion of legged robots," *IEEE Transactions on Robotics (In Press)*, 2018.

[34] M. Naveau, M. Kudruss, O. Stasse, C. Kirches, K. Mombaur, and P. Souères, "A reactive walking pattern generator based on nonlinear model predictive control," *IEEE/RAS Robotics and Automation Letters*, vol. 2, 2017.

[35] J. Mirabel, S. Tonneau, P. Fernbach, A. Sepp al a, M. Campana, N. Mansard, and F. Lamiraux, "HPP: A new software for constrained motion planning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016.

[36] P. Fernbach, S. Tonneau, and M. Taïx, "CROC: Convex Resolution Of Centroidal dynamics trajectories to provide a feasibility criterion for the multi contact planning problem," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2018.

[37] J. Carpentier, *Computational foundations of anthropomorphic locomotion*. PhD thesis, Université Toulouse 3 Paul Sabatier, 2017.