LIB3D_MEC–GINAC, A LIBRARY FOR SYMBOLIC MULTIBODY DYNAMICS

Javier Ros*, Luis Arrondo*, Javier Gil*, and Xabier Iriarte*

*ACME Research Group
Public University of Navarre, Campus Arrosadia s/n, 31006 Pamplona, Navarre, Spain
e-mail: jros@unavarra.es, luisarrondo@hotmail.com, jgil@citean.es, xabier.iriarte@unavarra.es
http://www.imac.unavarra.es/

Keywords: Multibody, Symbolic, GiNaC, 3D_MEC, Atomization, Trigonometrical Simplification.

Abstract. A general purpose symbolical C++ Open Source GPL–ed (Ref. [2]) library for multibody dynamics is presented. The library offers the user a language that closely resembles that of mechanics: vectors, tensors, points, bases, frames, matrixes,... and a swiss army knife of dynamic operators are available. So, the pose of multibody dynamics equations is greatly simplified. There is not any limitation regarding usable coordinate types or multibody formalism implementation. Modern features such as atomization of expressions are available. C++ object oriented features allow the extension and particularization of the library. The library has been developed as a modern replacement of 3D_MEC’s (Ref. [3]) own kernel, although it can easily be used standalone, as is demonstrated in this article.
1 INTRODUCTION

Algebra offers to the scientist a natural language to express the equations of mechanics. Symbolic algebra makes available this language in computers.

It has been used to deal with the pose of the equations of multibody systems for long. Today there are a lot of well known multibody packages relying internally in a symbolic kernel (Ref. [7], [5], [9], [8],...) Such multibody packages use their own symbolic libraries while some others rely on commercial symbolic packages.

3D_MEC (Ref. [3]) is such a package, one that implements its own symbolic kernel. It has been used for more than 15 years in order to teach mechanics. Users of 3D_MEC interact with the application in a programmatic way. They define the equations of the mechanical system on 3D_MEC’s own language, so no particular formalism is implemented. Today 3D_MEC lacks some desirable features of modern symbolic algebra based multibody packages (Ref. [5]) mainly because the difficulties related to the implementation of a powerful symbolic kernel.

GiNaC (Ref. [1]) is a C++ Open Source GPL–ed (Ref. [2]) symbolic algebra library that has proved its maturity in a different number of contexts. In fact its authors claim that its development was triggered because they were unable to deal with very large expressions related to their research on theoretical physics using existing commercial packages. Previous tests performed using GiNaC to serve as the underlying kernel of a symbolic package for multibody mechanics have been satisfactory (Ref. [4]), and demonstrate that GiNaC implements all the primitives needed to implement a multibody symbolical kernel with modern features.

In this work a C++ library for symbolic multibody dynamics based on GiNaC, lib3D_MEC–GiNaC, is presented. This library allows the user to symbolically pose the equations of general problems in multibody dynamics in a very efficient way using the programmatic features of C++. It transparently manages atomization of expressions (Ref. [5]), trigonometrical simplifications (to some extent), standard matrix algebra, and 3D euclidean space common mechanical manipulations: point, projection bases and reference frame definitions, plus operators that transparently give base change matrixes, angular velocities and accelerations, total time derivative of vectors, velocities and accelerations of points,...

2 THE LIBRARY IDEA IN AN EXAMPLE

The simplest way to get a quick idea of the library capabilities is to see how the dynamical equations of a very simple mechanical system can be obtained. Fig. 1 represents the mechanism to be analyzed: Block (yellow) slides with coordinate $x$ (red horizontal arrow) and pendulum (blue) rotates with $\theta$ (green horizontal arrow). There is no friction. The equations are obtained using the Newton–Euler Formalism.

```cpp
#include "Matrix.h"
#include "Vector3D.h"
#include "Tensor3D.h"
#include "System.h"

#include <ginac/ginac.h>
using namespace GiNaC;
```

Figure 1: System drawing.
int main()
{

    //System definition
    System sys;
    atomize=0;

    //Coordinate definition
    symbol x = *sys.new_Coordinate("x");
    symbol theta = *sys.new_Coordinate("theta");

    //Kinematical parameter definition
    symbol d = *sys.new_Parameter("d");
    symbol l = *sys.new_Parameter("l");
    symbol r = *sys.new_Parameter("r");

    //Define Base
    sys.new_Base("BPendulum","xyz",0,1,0, -theta);

    //Define Vectors
    Vector3D OGb = *sys.new_Vector3D("OGb",x,0,d/2,"xyz");
    Vector3D OA = *sys.new_Vector3D("OA",x,0,0,"xyz");
    Vector3D GbGp=*sys.new_Vector3D("GbGp",0,0, -l,"BPendulum");

    //Define Points
    Point * Gb = sys.new_Point("Gb","O",&OGb);
    Point * A = sys.new_Point("A","O",&OA);
    Point * Gp = sys.new_Point("Gp","Gb",&GbGp);

    //Define Frames
    Frame * Block = sys.new_Frame("Block","Gb","xyz");
    Frame * Pendulum=sys.new_Frame("Pendulum",Gp,"BPendulum");

    //Dynamical Parameter Definition
    symbol mblock = *sys.new_Parameter("mblock");
    symbol mpendulum = *sys.new_Parameter("mpendulum");
    symbol g = *sys.new_Parameter("g");
    symbol I1 = *sys.new_Parameter("I1");
    symbol I2 = *sys.new_Parameter("I2");
    symbol I3 = *sys.new_Parameter("I3");
    Tensor3D IpendulumGp = *sys.new_Tensor3D("IpendulumGp", I1,0,0,0,I2,0,0,0,I3,"BPendulum");

    //Joint Unknown Definition
    symbol Fgby = *sys.new_Joint_Unknown("Fgby");
    symbol Fgbz = *sys.new_Joint_Unknown("Fgbz");
    symbol Mgbx = *sys.new_Joint_Unknown("Mgbx");
    symbol Mgby = *sys.new_Joint_Unknown("Mgby");
    symbol Mgbz = *sys.new_Joint_Unknown("Mgbz");
    symbol Fbpx = *sys.new_Joint_Unknown("Fbpx");
symbol Fbp = *sys.new_Joint_Unknown("Fbp");
symbol Fb = *sys.new_Joint_Unknown("Fbp");
symbol Mbpx = *sys.new_Joint_Unknown("Mbpx");
symbol Mbpz = *sys.new_Joint_Unknown("Mbpz");

//Joint Torsor Definition
Vector3D F_GB = *sys.new_Vector3D("F_GB",0,Fgb,Fgbz,"xyz");
Vector3D M_GB_A = *sys.new_Vector3D("M_GB_A",Mgbx,Mgb,Mgbz,"xyz");
Vector3D F_BP = *sys.new_Vector3D("F_BP",Fbpx,Fbpy,Fbpz,"xyz");
Vector3D M_BP_Gb = *sys.new_Vector3D("M_BP_Gb",Mbpx,0,Mbpz,"xyz");

//Constitutive Forces and moments Definition
Vector3D Block_gravity = *sys.new_Vector3D("mblock*g",0,0,-mblock * g,"xyz");
Vector3D Pendulum_gravity = *sys.new_Vector3D("mpendulum*g",0,0,-mpendulum * g,"xyz");

//Define Velocity and Acceleration of Point Gb
Vector3D VabsGb = sys.Dt(OGb,"abs");
Vector3D AabsGb = sys.Dt(VabsGb,"abs");

//Define Acceleration of Point Gp
Vector3D Ogp = sys.PositionVector("O","Gp");
Vector3D VabsGp = sys.Dt(OGp,"abs");
Vector3D AabsGp = sys.Dt(VabsGp,"abs");

//Angular moment and Inertia Moment of pendulum
Vector3D OmegaPendulum = sys.AngularVelocity("xyz","BPendulum");
Vector3D H_Gp = IpendulumGp * OmegaPendulum;
Vector3D Iner_Moment_Pendulum_Gp = -sys.Dt(H_Gp,"xyz");

//Dynamic Equations
Matrix equation_1_3 = -mblock * AabsGb + Block_gravity + F_GB - F_BP;
Matrix equation_2_3 = ((sys.PositionVector("Gb","A") ^ F_GB) + M_GB_A) - M_BP_Gb;
Matrix equation_3_3 = -mpendulum * AabsGp + Pendulum_gravity + F_BP;
Matrix equation_4_3 = Iner_Moment_Pendulum_Gp + ((sys.PositionVector("Gp","Gb") ^ F_BP) + M_BP_Gb);

Matrix Dynamic_Equations(4,1,
equation_1_3,equation_2_3,equation_3_3,equation_4_3);

Matrix q = sys.Coordinates();
Matrix dq = sys.Velocities();
Matrix ddq = sys.Accelerations();
Matrix epsilon = sys.Joint_Unknowns();
Matrix M=Jacobian(Dynamic_Equations.transpose(),ddq);
Matrix $V$ = Jacobian (Dynamic_Equations.transpose(), epsilon);
Matrix $Q$ = Dynamic_Equations - (M * ddq + V * epsilon);

cout << Dynamic_Equations << endl;
}

Outputs:

\[-F_{bpx} - ddx \cdot m_{block};
  -F_{bpy} + F_{gby};
  F_{gbz} - g \cdot m_{block} - F_{bpz};
  1/2 \cdot d \cdot F_{gby} - M_{bpx} + M_{gbx};
  M_{gby};
  M_{gbz} - M_{bpz};
  -(ddx + \sin(-\theta) \cdot d \theta^2 \cdot l + dd \theta \cdot l \cdot \cos(-\theta)) \cdot m_{pendulum} + F_{bpx};
  F_{bpy};
  -g \cdot m_{pendulum} - (-\sin(-\theta) \cdot dd \theta \cdot l
  + d \theta^2 \cdot l \cdot \cos(-\theta)) \cdot m_{pendulum} + F_{bpz};
  M_{bpx} - F_{bpy} \cdot l \cdot \cos(-\theta);
  dd \theta \cdot I_2 - \sin(-\theta) \cdot l \cdot F_{bpz} + l \cdot F_{bpx} \cdot \cos(-\theta);
  M_{bpz} + \sin(-\theta) \cdot F_{bpy} \cdot l]\n
3 LIBRARY IMPLEMENTATION

The library is mainly based upon GiNaC’s expression class, class $\text{ex}$. GiNaC’s expressions are used in order to represent algebraic expressions made of integers, rationals, symbols, and functions, and common algebraic operators “+, -, *, /, ^”, although GiNaC’s expressions are more general than this. This class provides several key methods: partial derivative with respect to a symbol, automatic expression canonicalization and simplification (this does not include trigonometrical simplification), substitution of subexpressions (literal and algebraic), expansion, and much more.

Lib3D_MEC–GiNaC library implements a new matrix class, class $\text{Matrix}$. Matrix elements are symbolical expressions that become the body of the matrix algebra that is implemented using the operator overloading features of C++. When C++ does not provide the required operator to overload (i.e. transpose, partial derivative, jacobian) member functions are used instead.

Total time derivative of expressions and matrixes requires the definition of the time derivatives of variables.

To that end a system class, class $\text{System}$, is defined that provides containers for time symbol, and for symbols representing generalized coordinates (or state variables), and their derivatives (velocities and accelerations). Also containers for parameters and joint–unknowns are provided. Then time derivative of expressions and matrixes are implemented as methods of this class.

Based on matrix class, classes that represent vectors and tensors, class $\text{Vector3D}$ and class $\text{Tensor3D}$, are defined. They are essentially $3 \times 1$ and $3 \times 3$ matrixes with a reference to the base in which their components are meaningful. As with matrix class, operators are also
overloaded in these classes so the algebra of vectors and tensors can be used transparently. Member functions that return partial derivative of vectors and tensors are provided as well as time derivatives, $\partial_t$, with respect to a base.

So, the needed base class is defined composed of a reference to a father base, a rotation vector and an angle that give the orientation of a base relative to its father. In general a set of bases will resemble a tree shaped structure. Methods are provided that give base change matrix, angular velocity vector, and skew-symmetric angular velocity tensor with respect to the base's father.

A container for user defined bases is added to system class, and new methods are defined in system that return the base change matrix, angular velocity vector and skew-symmetric angular velocity tensor between arbitrary bases.

These methods are enough to be able to implement the above mentioned overloaded operators for vectors and tensors as will be further explained in section 3.1.

A point class, class Point, is defined composed of a reference to father point, and a vector that gives the position of the point relative to its father. In general a set of bases will resemble a tree shaped structure. Methods are provided that give the position vector of a point relative to the points father.

A container for user defined points is added to system class, and a new method is defined in system that returns the position vector between arbitrary points.

Finally a frame class, class Frame, composed of a reference to a point and to a base. Methods allowing points and bases as arguments are overloaded so that they can allow instead a Frame argument. When the original method parameter is a point, the frame is interpreted as frame's point in overloaded method. When the original method parameter is a base, the frame is interpreted as frame's base in overloaded method.

Finally new methods are added to system class to export expressions and matrixes using MatLab or C syntax.

With the provided classes and methods any multibody formalism (Newton-Euler, Virtual Power Principle, Lagrange,...), based in absolute coordinate, or in natural (Ref. [?]) or relative coordinate, including recursive type formulations, can be posed.

### 3.1 Vector and tensor operator implementation algorithm

Operators for vectors an tensors would have the same implementation that the ones for matrices if all vectors an tensors had associated the same base. When this is not the case the question reduces to transform vectors or tensors components from their own base to a common base. This is only necessary in case of binary operators. The needed base change matrixes are computed using the system's above mentioned methods. If the result is not a expression, then common base has to be assigned to the resulting vector or tensor.

The common base election is critical. A bad election will produce too long expressions that are plenty of trigonometrically simplificable expresions.

The most general base structure is going to have a tree like shape. For a couple of bases a common base that falls within the shortest possible path between them minimizes successive base transformations needed to transform vectors and tensors. Which base in the path remains an undetermined question that is addressed as follows:

- If bases are in different tree arms, common base gives the base the closest to the root of the base tree.

- If they are in the same tree arm a gravity modifier is provided. If gravity modifier is:
– “down gravity” common base gives the base closest to the root of the base tree
– “up gravity” common base gives the base closest to the leafs.

This gravity modifier allows to apply the most common type of base transformations used in multibody dynamics, including recursive formulations (Ref. [5]).

A similar algorithm is used when dealing with the determination of position vectors between arbitrary points in the tree of points.

### 3.2 Atomization

Atomization an deatomization of symbolic expressions (Ref. [5]), a technique that drastically reduces the number of operations needed to numerically evaluate the set of expressions required in the solution of different dynamical problems, has been implemented. Basically atoms are the simplest subexpressions that can be found in expressions. Expressions are substituted for existing atoms and then searched for new atoms recursively until no more atomizations can be performed. To numerically evaluate a set of expressions, only the set of atoms these expressions are dependent on has to be evaluated.

For example, for the listing of section 2 the following output is obtained if atomization is activated:

```
[atom10; atom11; atom12; atom15; Mgby; atom16; Fbpx; Fbpy; atom17; atom21; atom19; atom22]
```

Where atoms are something like

```
atom(1) = -g*mpendulum;
atom(2) = -sin(-theta);
atom(3) = atom2*l;
atom(4) = -l*cos(-theta);
atom(5) = x+atom3;
...
```

The algorithm used differs sightly from the one used in (Ref. [5]) although producing similar results. Basically:

- Every time a binary operation is performed, the result it is atomized.
- Symbols excluded from atomization can be defined. When atomizing a sub-expression, if a excluded symbol is found, that subexpression is not atomized. Then partial derivatives with respect to the excluded atoms can be evaluated.
Deatomization, is the opposite of atomization. It can be used to print out results in readable format. Also to perform derivatives of atomized expressions that involve symbols not excluded from atomization. Then the expression can be re-atomized.

![Figure 2: 3D_MEC.](image)

### 4 USING THE LIBRARY

As has been demonstrated in section 2, it is possible to pose dynamical equations of multibody systems defining a single class instance of the system class. Then, only the functionality provided by the library is available.

Programmatic features of C++ allow the user to derive child classes from the system class, adding new fields, and methods.

For example, a user may prefer to have LaTeX output for the generated mathematical expressions. Another user may prefer to define a solid class and a joint class and containers for it in a derived new system class, thus providing a higher level of abstraction for the pose of the multibody equations.

A strictly minimum functionality is implemented. For example no particular formalisms (Newton-Euler, Virtual Power, Lagrange,...) nor coordinate type (natural, absolute, relative), no particular algorithm implementation (Recursive $O(n^3)$, $O(n)$,...) is implemented (Ref. [?]). Nevertheless, it is possible to use the library to deal with the above particularizations, for instance by specializing its system class.

This simplicity is intended: we want the library to be as general as possible, so that it can serve as the base for the implementation of more specialized multibody implementations. We’ve tried to implement a maximum of functionality but without compromising generality.

The presented library will soon replace 3D_MEC’s (Fig. 2) old symbolic library. Then 3D_MEC’s will provide a new intuitive interface for the library, using its own language (notation aspects left apart, 3D_MEC’s language resembles the one used in the example of section 2).
5 CONCLUSIONS

A generic symbolic library for multibody dynamics has been presented. Directly provided features allow to symbolically pose the equations of multibody system dynamics in an abstract language that resembles that of mechanics. It provides advances features like atomization of expressions and best compromise base change strategies that minimize trigonometrical simplificability and length of expressions. C++ features allow the specialization of the library to suit particular purposes. Lib3D_MEC-GiNaC, has been developed as a replacement for the old kernel library that is used by the multibody application 3D_MEC, so as an alternative the functionality of the library will be soon available in a more interactive way using 3D_MEC’s own language.

Lib3D_MEC-GiNaC is GPL–ed Open Source, so you can use and modify it to fit your particular needs.

REFERENCES


