HOTINT – A C++ ENVIRONMENT FOR THE SIMULATION OF MULTIBODY DYNAMICS SYSTEMS AND FINITE ELEMENTS

Johannes Gerstmayr
Linz Center of Mechatronics,
Altenbergerstraße 69, 4040 Linz, Austria
e-mail: johannes.gerstmayr@lcm.at

Keywords: Multibody System Dynamics, Simulation Software, HOTINT, Time Integration, Finite Elements, C++.

Abstract. A multibody dynamics system simulation code HOTINT is presented. The software has been developed for research purpose during the past ten years and has some consistently different features as compared to other commercial and research software. The simulation software originates from a pure time integration code that was able to solve differential algebraic equations of motion. Five years ago, a multibody system kernel has been attached to the time integration code and a 3D visualization engine has been developed. At the current stage the software is able to solve dynamic or static problem consisting of a general system of objects. The objects are represented by classical first or second order differential equations, algebraic equations and inequalities, which all of them can be nonlinear. The general kernel is not only able to manage the equations, but also to handle data of the objects for direct editing and storage in a file, as well as graphical representation of the system and export of resulting quantities of the system. The solver contains specific solvers for open and closed loop multibody systems, all of them based on redundant multibody formulations. The solver is especially adapted to second order differential equations and does not intend to factorize the mass matrix of the system in any time step. The paper will show some details on the hierarchy of the code and introduce into some concepts, which might be interesting for other developers. It will also be shown what is necessary to add a new object, such as a mechanical body or a constraint condition. While the code is not available open source, a part of the software is available as a freeware and can be downloaded as well as some examples that have been solved with the software.
1 INTRODUCTION

The C++ code HOTINT [5] has been originally developed in order to simulate multibody dynamics systems for academic research purposes. The basic libraries of the code originate from the solution of static nonlinear finite element problems which have been developed within a diploma thesis in 1997 [1]. In the following six years, the code has been extended to solve differential algebraic equations (DAEs). At that time, implicit Runge-Kutta (IRK) schemes with an arbitrary number of stages were implemented. The name of HOTINT thus originates from High Order Time INTegration. The IRK-tableaus were defined in an external text file. Special attention has been paid to the solution of nonlinear, differential-algebraic and discontinuous dynamical systems.

As a special feature, the time integration core can distinguish first order, second order and algebraic equations. The second order equations are given by a non-constant mass matrix and a right-hand side. The mass matrix needs to be supplied by a function but the Jacobian can be computed fully automatically. For the treatment of discontinuous problems, an additional iteration can be added which attempts to resolve the discontinuities outside the Newton iteration for the underlying nonlinear problem.

The time integration kernel has been extended in 2004 by means of a multibody library, which supplies the equations of motion for a large number of common objects in multibody dynamics simulation. The multibody library consists of planar and spatial rigid bodies, flexible beams and plates and standard solid finite elements, like triangles, tetrahedrals and hexahedrals. The library also includes joints, actuators, springs as well as more advanced objects such as contact, friction and control.

The simulation code has ever tempted to be researcher friendly. This means that all kinds of results are easily and instantaneous accessible during computation. As an important feature in dynamic simulation, the actual (displaced or deformed) configuration can be always visualized during computation and results that are available from the beginning up to the actual time step are available in an external file. Any errors in the model or bad convergence of the solver can be easily discovered at the beginning of or during the simulation and, in many cases, it is not necessary to wait for the completion of the computation job, but to interrupt and restart with a modified model after a short time. Any kind of result quantities, such as displacements, velocities, forces, stresses, strains, etc. can be measured point-wise with so-called sensors and directly viewed in or easily exported to MATLAB. Alternatively, overlaid field plots are possible in deformable bodies, e.g. the distribution of strain in a solid finite element.

The executable version of HOTINT is available as a freeware from the homepage [5], however, containing only a limited number of the features implemented in HOTINT. Currently, HOTINT is applied in many research investigations and in the solution of several industrial problems an currently has about 150,000 lines of code. In the present paper, we show some basic features and implementation details of the research code HOTINT. The paper should help other researchers in the implementation of similar algorithms.

2 STRUCTURE OF THE CODE

The structure of the code HOTINT is organized according to a very common C++ object-oriented style. The code is divided into two large parts, see Figure 1. The first part (red) is the Windows or user-interface. This part excessively uses windows functions and is therefore not available on Unix-platforms. In order to make the code portable, this part of the code is maintained in a separate library with only a small interface to the multibody kernel. The multibody
kernel and the solvers are in another library and do not use other features than standard C++ functions.

The Windows interface is connected to the following components:

- **Graphics Display** – an OpenGL 3D rendering engine, which is compatible with any nowadays graphics card. This allows pre-processing, post-processing as well as the visualization of the actual solution during computation. As a main feature in HOTINT, the solution can be visualized during computation, because the visualization is performed in a separate task, fully decoupled from the solution procedure. For research purposes, this is very advantageous, because it allows detecting errors in the simulation (e.g., wrong parameters or wrong discretization) at an early stage of the computation. It is not necessary to run the simulation till the end and to start post-processing afterwards.

- **User Input/Output** – The user can interact with the multibody system by adding, deleting and editing the objects of the multibody system. A large effort has been put to the simplification of this task regarding the implementation. Every editable object in the multibody system has only two access functions, which are used for all editing commands as well as for load from file / save to file procedures. Thus, it is easy for the researcher to add or extend an element in HOTINT.

- **User Interface Options** – There are many options that can be changed in HOTINT. The main options are regarding
  - Viewing options: Change size of graphics window, animation speed, redrawing during computation, viewing angle, etc.
  - OpenGL drawing options: Setup general light, reflection and transparency options for the OpenGL visualization
  - Finite element drawing options: Options for drawing the field output of finite elements, like displacement field, stress field, etc. Allowing a contour plot with upper and lower or automatic limits and shrinking of the elements. The drawing resolution of structural and solid finite elements can be changed

Figure 1: Multibody system core and windows interface.
Rigid body / Joint options: Change how rigid bodies, joints and sensors are visualized in the multibody system: draw outlines, transparent, show body/joint/sensor numbers, show loads, etc.

Computation Settings: Generally setup the options for the computation of the multibody system: start time, end time, step size; accuracy parameters for Newton solver; setup static solver parameters; setup solution files and output directories

- Data manager – There exists a special tool for management of the time-dependent solution of the multibody system. The solution of the multibody simulation can not only be visualized and animated, but also exported as a compressed solution file or as a solution at specific time points. The animation can be exported as a series of .jpg pictures which then can be transformed to an animation with freeware tools.

### 2.1 Structure of the multibody kernel

The multibody kernel is used to put elements together and to prepare the equations of motion, which are given by each body and load, for time integration or for static solution. Thus the multibody kernel only consists of several lists of objects:

- **Element list**: A list of bodies and connectors that are simulated in the multibody system
- **Node list**: A list of nodes, which are used to define degrees of freedom of finite elements in the multibody system
- **Sensor list**: A list of sensors for measuring quantities in the multibody system
- **Geometrical objects list**: A list of geometrical objects that are drawn together with the multibody system, which allow a illustrative graphical representation of the multibody system

The tasks of the multibody kernel are

- Collecting the equations of motion of all attached elements to the solvers
- Starting or aborting the simulation
- Providing settings for simulation and for user interaction
- Collecting graphical objects to be drawn by the 3D visualization engine
- Providing an interface for reading or writing object data (elements, sensors, etc.)
- Reading or writing data files for the multibody system

There is lot of implementation of more than 100 general settings and solver parameters, as well as for the graphical representation which is not content of this paper. Concerning the preparation of the equations of motion for the solvers, the following procedures might be interesting to the reader. There is a general assembling procedure, which mainly assigns global coordinate indices to each element. The global coordinates are often degrees of freedom of the elements. The assembly routine has the following structure:

- Assign coordinate indices to nodes
- Assign coordinate indices to elements
- Link elements and constraints
- Link elements and loads
- Compute optimal ordering of global coordinates for solver
At the end of the assignment of the coordinate indices, which is performed element-wise, the total number of coordinates is known and a global solution vector can be built. It should be mentioned that for second order differential equations, there exist two coordinate indices, one for position level coordinates and one for velocity level coordinates. The linkage of elements, constraints and loads provides a double-linking, such that an element knows its constraints and a constraint knows its elements. The computation of the optimal ordering of the global coordinates for the solver is a simple algorithm which is described in the solver section.

The main objects in the multibody kernel are given by

- Elements (computable objects)
- Nodes
- Loads
- Sensors
- Geometrical objects

Each of these objects have a specific task as described in the following paragraphs.

### 2.2 Elements (computable objects)

An element, or in general a computable object is a part of the multibody system, which represents a set of algebraic or differential equations (e.g. a rigid body), or which contributes terms to the equations of motion of the multibody system (e.g. a spring attached to two bodies). There are two main classes of elements, see Figure 2. The first and main class of objects consists of 2D and 3D bodies, which have a certain volume, mass, a position and some physical parameters. As a common feature, there exist equations of motion of a body, which are not coupled to the equations of motion of other bodies. The second class of objects represents connectors. Connectors provide coupling conditions between bodies, which are usually denoted constraints. Classical connectors are joints like revolute or spherical joints, or spring-dampers attached to two bodies. Thus, a connector does not need its own mass, volume, position or physical parameters and may couple a certain number of bodies.

#### 2.1.1 Bodies

In general, it is not necessary to distinguish between planar and spatial bodies. The only difference lies in the different access functions, which means that 2D bodies have special 2D access functions e.g. for position or velocity.

As a main feature of the code HOTINT it is based on a redundant multibody formulation. This means that every body has its own degrees of freedom and is unconstrained. After assembling the bodies with the constraints, the bodies are constrained to certain motion or deformation. Thus, the implementation of 2D and 3D point mass or rigid body elements is comparatively simple. For example, a 2D rigid body only has a diagonal mass matrix and no gyroscopic terms, which leads to an extremely fast evaluation of the system components. Hence, for such elements, most of the CPU time is spent for the application of constraint forces, evaluation of constraint conditions and solution of the nonlinear system of equations in each time step.

Concerning finite elements in the code HOTINT, nodes are introduced, which provide the degrees of freedom for finite elements. This avoids the application of constraints between
each finite element at the nodes. Nodes only virtually exist in the multibody code, because after assembly, the finite elements know their respective nodal degrees of freedom and afterwards do not need nodal information any more.

Another special feature of flexible multibody systems is the floating frame of reference concept. Linear finite elements (e.g. linear beam element “Beam2D”) are utilized and combined with a moving, co-rotating reference frame. The equations of motion, including the mass then need to be modified as compared to the elements without reference frame. The reference frame itself obtains certain degrees of freedom, such as position of the reference point and rotation. Currently, only planar floating frame of reference elements are implemented.

Figure 2: Element tree with typical elements in the multibody system core.

2.1.2 Connectors

Connectors are elements, which have the possibility to link elements or to add constraints to elements. As mentioned before, constraints are the most important ingredient in a redundant multibody formulation. For certain problems, the evaluation of constraint forces consumes most of the simulation time and should be therefore treated carefully.

Classical joints, see Figure 2, can be included in a quite simple form into the multibody system software HOTINT. There are two necessary functions to be given. One function, which specifies the residual (which may be a vector) for the constraint equations using a given solution vector. The second function needs to provide information on how Lagrange multipliers, which mostly represent constraint forces or constraint torques, are applied to a body. In order to simplify this process, only the general action of constraint forces to flexible bodies needs to be given, which is based on access functions provided by each body. For more details, see Section 3.6 on necessary implementations for constraints.

In addition to classical joints, special connectors are available. A useful constraint condition is a generalized coordinate constraint (CoordConstraint). In a coordinate constraint, a selected generalized coordinate of a body is constraint to be zero, to obtain a certain value, or to follow a certain function in time (PrescribedCoordConstraint). Such a constraint can be utilized to e.g. fix the x-component of the reference position of a rigid body to
the ground. Another important connector is a spring-damper-actuator. This type of connector has no constraint equations, but only adds forces to bodies, similar to a constraint. A special version of an angular spring-damper-actuator exists as well. Similar to a spring-damper-actuator, a hydraulic actuator exists, however the force of the hydraulic actuator is not proportional to the distance of two reference points, but the hydraulic actuator also includes the pressure-build-up equations. These equations are of first order and are incorporated to the multibody system. In order to include contact and friction into the multibody system, contact elements are available. Contact is implemented in a very general form, using a contact container, in which all objects that should be considered to undergo contact are put in. Each object undergoing contact must have a certain master-slave contact geometry based on line segments, points and circles in 2D or triangles, points and spheres in 3D. The geometry is used to build dynamically varying search trees, which provide efficient access to a geometrical object at each place. Upon contact of two bodies, one body is designated to be master and the other becomes the slave. The contact strategies include a Lagrange and a penalty contact formulation as well as an adaptive step size reduction during impact and release of contact. The automated formulation requires few user interaction to establish contact in multibody systems, however, not all possible combinations of contact geometries and contact formulations are yet available, especially for the 3D case.

In order to study controlled multibody systems, e.g. a classical mechatronical system, it is advantageous to have certain functionalities to include simple control algorithms in the code. At the actual stage, HOTINT includes so-called control elements, which have functionalities that are similar as in simulation codes such as Simulink or SciLab. A control element is a connector element with defined input and output. The input can be the output of another control element, or the output of a sensor. The output can be used to be visualized or to be written into a file, or it can be utilized as a force which is applied to a body. Control elements can be simple elements such as constant value, the time or linear transformations. Control elements can also be general mathematical objects (polynomials, transcendental functions, piecewise-linear functions, etc.), s-functions that are given in the Laplace domain, or general linear differential equations. With the combination of such functions, simple control algorithms can be implemented and added to the multibody system.

2.3 Nodes for Direct Connection of Finite Elements

Sometimes it is more efficient to connect elements without the application of constraints. E.g. in the case of nodal finite elements it is advantageous if the connected elements share nodal coordinates. Therefore, it is possible in HOTINT to define nodes, which can be afterwards used to assign nodal coordinates to the elements.

2.4 The Concept of Loads

Loads are used to add forces by means of terms at the right hand side of the equations of motion. Loads are directly linked to bodies and they do not have own generalized coordinates (unknowns). However, loads can depend on the body coordinates or body deformation (e.g. in the case of pressure). Loads are divided into two parts. The first part defines the way of action and the size of the load, such as a point load or torque, distributed or body load. The second part defines the time-dependency of the load, such as constant, polynomial or sinusoidal change in time.
2.5 Sensors for Measuring

Sensors are used to measure certain quantities of the multibody system at the current state of the computation. The output of a sensor is usually written to output files at certain time steps. The solution file contains the output of all sensors, each sensor in a row, versus the time (first row). Apart from output and controllers, sensors do not influence the computation.

While generalized coordinate sensors can be used to measure the coordinates of any element (e.g. of a constraint), the position, angle, distance and deflection sensors can only be applied to elements of the type body. Note that local second order differential variables of a body contain position level coordinates and velocity level coordinates. Sensors can not have own generalized coordinates (unknowns).

2.6 Geometric Elements for Bodies with Complex Geometry

Geometric elements are used to represent a realistic shape of complex bodies in the multibody simulation. Usually, a geometric element is either used to define objects in the background or it is attached to a (rigid) body.

Geometric elements can be either defined with geometric primitives or by triangular meshes. The only influence to the computation by geometric elements is present by the automatic computation of mass, volume and inertia from the geometric elements. Usually, the complexity of geometric elements does not influence the computational time (CPU time), except for the drawing and loading/saving of multibody models.

3 NECESSARY IMPLEMENTATIONS FOR BODIES AND CONSTRAINTS IN THE MULTIBODY SYSTEM

In the following, it is shown what is necessary to implement in order to include a new body to the multibody system software. This shall show that the software is especially intended for research purposes, as it facilitates the implementation of equations of motion significantly.

The equations of motion for a body are given in general form as

\[ M(q, \dot{q}, y, z)\ddot{q} = f_2(q, \dot{q}, y, z) \]

\[ \dot{y} = f_1(q, \dot{q}, y, z) \]

\[ \theta = g(q, \dot{q}, y, z) \]  

(1)

Here, \( M \) denotes the mass matrix, \( q \) are second order variables, \( y \) are first order variables, and \( z \) are algebraic variables. Note that the mass matrix also can depend on the velocity degrees of freedom, which is usually not the case. The right hand side of the second order differential equations is split into the following parts

\[ f_2(q, \dot{q}, y, z) = -D\dot{q} - Kq - \left( \frac{\partial g}{\partial q} \right)^T\lambda - f_{nl} - f_{ext} \]

(2)

in which \( D \) denotes the damping matrix, \( K \) denotes the stiffness matrix, \( \lambda \) are Lagrange multipliers, \( f_{nl} \) are nonlinear forces resulting e.g. from nonlinear elastic problems or gyroscopic terms, and \( f_{ext} \) are external forces. Note that external forces due to loads and constraint forces due to joints are automatically added to the body’s equations by the multibody kernel. In Eq. (2) constraint forces of the body itself need to be taken into account. Such internal constraints
of a body could result from a redundant parameterization by using e.g. four Euler parameters for the representation of rigid body rotations.

In addition to the second order equations, first order equations can be added, see the second line of Eq. (1), which usually result e.g. from coupling to fluid problems or from hydraulic or electric systems. The first order equations need to be written in explicit form. In the third line of Eq. (1), constraint equations can be prescribed representing the internal constraints of the body.

In order to include standard second order equations in a body, the following functions need to be prescribed (exemplarily given for a 3D rigid body):

- Specify the number of equations and the number of unknowns for the body (usually identical), e.g.:

```cpp
int Rigid3D::SOS() const {return 7;}
```

- Compute mass matrix $M$

```cpp
void Rigid3D::EvalM(Matrix &m, double t)
{
    //m is already initialized with zeros!
    m(1,1) = mass; m(2,2) = mass; m(3,3) = mass;

    //beta0 .. beta3 contains actual Euler Parameters
    Matrix Gbar;
    Gbar.Set34(-2*beta1, 2*beta0, 2*beta3,-2*beta2,
               -2*beta2,-2*beta3, 2*beta0, 2*beta1,
               -2*beta3, 2*beta2,-2*beta1, 2*beta0);
    Matrix mtheta = (Gbar.GetTp()*Iphi)*Gbar;

    for (int i=1; i<=4; i++)
        for (int j=1; j<=4; j++)
            m(3+i,3+j) = mtheta(i,j);
}
```

- Compute right-hand side $f_2$ including constraint forces ($\lambda$) due to the constraint for the Euler parameters:

```cpp
void Rigid3D::EvalF2(Vector &f, double t)
{
    //compute angular velocity vector
    Vector omegabar = Gbar*betap;
    Vector temp = omegabar.Cross(Iphi*omegabar);

    Mult(Gbar.GetTp(),temp,betap);
    //compute quadratic velocity vector:
    Vector Qv = Gbar.GetTp()*temp;

    f(4) += -Qv(1) - lambda*beta(1);
    f(5) += -Qv(2) - lambda*beta(2);
    f(6) += -Qv(3) - lambda*beta(3);
    f(7) += -Qv(4) - lambda*beta(4);
}
```
• The stiffness matrix is automatically computed for each body by default, however, it can also be explicitly given as a function.

The latter specification would be already sufficient to compute the body’s equations of motion and to store the solution of each coordinates in an output file. However, in order to make the element generally available in the multibody system the following parts should be additionally provided:

• Initialization (at several stages: during setup of element; before linking all elements; before start of computation; before and after each time step)
• Access functions for editing or loading/saving of data of the body
• Access functions for sensors and drawing
• Access functions for attaching loads and constraints
• Functions for graphical representation of the body

3.1 Initialization procedures

Usually, there is a difference in the representation of the body by user input quantities and the quantities that are stored for the body. Regarding the rotations, the body inputs initial Euler angles and the angular velocity vector. However, internally rotations are represented by means of Euler parameters and its time derivatives. Such and other transformations, e.g. the calculation of the mass moments of inertia needs to be done during initialization. Regarding finite elements, the element coordinates depend on nodes. Thus, several initialization procedures can only be done after assembly of the multibody system. Therefore, the initialization is necessary after assembly as well. For efficiency purposes, it is also necessary to perform some tasks at the beginning of a time step and after a time step has been finished.

3.2 Access functions for editing

There exist very convenient access functions for editing an object in the multibody system. In the C++ class object hierarchy, such access functions are implemented hierarchically, which means that only tasks need to be done, which are not done already by parent class members. In order to make editing of an object easy and extendable, each quantity of an object has the following properties:

• Data type: Boolean, integer, double, vector, matrix, text
• Limits: maximum and minimum values the data can obtain
• Name: a string, which is not used within the object for another data element
• Optional flag: specifies, if this data object is mandatory or optional for the element specification
• Array size flag: specifies, if the size of vector or matrix elements are fixed or variable
• Editable flag: specifies, whether this data element is editable or if it is just viewable (e.g. the object type can only be viewed but not edited)

During editing or loading/saving, the system automatically verifies for each data object, if violations of the definitions of each data components occur, or if e.g. a data component is missing. In the graphical user interface, the system automatically sets up a dialog box showing all editable components of the object, see Figure 3.
All data of an object is put into an element data container. This data container is used to transfer all relevant data between the user interface and the multibody system. As an example, the element data access functions for a point mass object are shown below:

Example for filling in the element data container by the point mass element:

```cpp
void Mass3D::GetElementData(ElementDataContainer& edc)
{
    ElementData ed; //temporary element data

    //data for computation
    ed.SetDouble(mass, "Mass"); edc.Add(ed);
    ed.SetDouble(radius, "Radius"); edc.Add(ed);
    ed.SetVector3D(x_init, "Initial_position"); edc.Add(ed);
    ed.SetVector3D(v_init, "Initial_velocity"); edc.Add(ed);

    //data for drawing and additional data
    ed.SetText(element_name, "Element_name"); edc.Add(ed);
    ed.SetInt(drawres, "Draw_resolution", 1, 1000); edc.Add(ed);
    ed.SetVector3D(color, "Color"); edc.Add(ed);
    ...
}
```

Note that there are additional components in the original implementation of the point mass object. Finally, the dialog box looks like:

![Edit dialog automatically generated for point mass.](#)

Storing the element data of a point mass “Mass3D” gives the following text in the saved MBS-file:

```
Element: Mass3D
{
    Element_name= Tip Mass3D
    RGB_color= [0.2, 0.5, 0.5]
    Use_alternative_shape= 0
    Geom_elements= []
    Mass= 1
    Radius= 0.1
    Initial_position= [0.25, 0.25, 0]
    Initial_velocity= [0, 0, 0]
    Draw_resolution= 16
}
```
3.3 Access functions for sensors and drawing

For a user-friendly multibody dynamics simulation software, the post-processing capabilities are inevitable. Access functions for sensors and drawing are split into certain categories:

- 2D and 3D functions
- Functions for the access to the actually computed configuration or access to the visualized configuration

In the following, we only exemplarily show 3D functions for the computed configuration. These functions include:

- Position and velocity of reference point (e.g. of the center of gravity)
- Position and velocity at a local position at the (rigid/flexible) body
- Rotation and angular velocity of reference point (e.g. of the center of gravity)
- Rotation and angular velocity at a local position
- Beam-related resultant forces
- Stress / strain components, inelastic part of strain (for elasto-plasticity)
- Direction of actuation of each coordinate (for drawing of nodal loads)

These quantities can be given with respect to body local coordinates or with respect to global coordinates. The choice of the coordinate system is defined by an appropriate flag.

Some of the access functions, however, do not make sense in every element, such as the computation of stress in a rigid body. However, in the object oriented programming style, in such cases a standard function exists, which returns e.g. zero stress in a rigid body per default.

3.4 Access functions for attaching loads and constraints

For the possibility to add constraints or loads onto bodies, special access functions are necessary. The following example of the virtual work of a point load \( F \) shows the necessary terms needed,

\[
\delta W_F = F \delta \mathbf{u} = F \left( \frac{\partial \mathbf{u}}{\partial \mathbf{q}} \right)^T \delta \mathbf{q}
\]  

(3)

The applied load \( F \), which can also represent a Lagrange multiplier in a constraint, is multiplied by the derivative of the displacements \( \mathbf{u} \) with respect to the element coordinates \( \mathbf{q} \). Thus, the matrix \( \frac{\partial \mathbf{u}}{\partial \mathbf{q}} \) must be implemented for each body. In general, the following types of vector-valued and matrix-valued functions are necessary, given also exemplarily in C-notation

Position: \( \left( \frac{\partial \mathbf{u}(\xi)}{\partial \mathbf{q}} \right)^T \) ... GetDPosDqT(Vector3D xi, Matrix& dudq),
Rotation: \[ \left( \frac{\partial \mathbf{A}(\mathbf{\xi}) \mathbf{v}}{\partial \mathbf{q}} \right) \] ... GetDAvDqT(Vector3D xi, Matrix& dudq)

Body load: \[ \int \frac{\partial \mathbf{u}(\mathbf{\xi})}{\partial \mathbf{q}} d\xi_1 d\xi_2 d\xi_3 \ldots \text{GetIntDuDq}(	ext{Matrix}& \text{dudq}) \] (4)

in which \( \mathbf{A} \) represents the transformation of a vector \( \mathbf{v} \) from local element (body) coordinates into global coordinates. The integral term is needed e.g. for body loads. In the case of nodal finite elements, it might be more efficient to especially implement these functions at the nodal positions. The implementation of Eq. (4) simplifies the combination of constraints with each body type considerably.

### 3.5 Functions for graphical representation of the body

Concerning the post-processing, the software HOTINT includes a unique online visualization tool, which allows interactive visualization of the actual simulation. The advantage for a developer lies in the fact that the results can be verified during simulation and it is not necessary to wait for the end of the simulation in order to visualize the results.

The visualization builds up on two types of access function. One type is the same as for sensors, which provides information about the position, velocity, rotation, stress, etc. with respect to a local position of the body, for details see Section 3.3. The second type of access function is a drawing function, called “DrawElement”, which provides the necessary 3D visualization information for the object under consideration. Usually, a body is visualized by means of a surface representation. In addition to that, a scalar field can be projected onto the surface, e.g. a displacement or stress component, which uses a color bar in order indicate the size of the scalar field quantity.

For automatic scaling, the object also needs to provide information about its extension, meaning that a bounding box needs to be specified. This bounding box is also necessary for contact computation.

### 3.6 Necessary Implementation for Connectors in the Multibody System

Bodies and connectors are both objects in the multibody system. However, there are conceptual differences between bodies and connectors in their dependency and coupling of degrees of freedom. In general, a body leads to second order differential equations and it has a certain amount of degrees of freedom. The body is usually unconstrained and thus moves freely in space. A connector, typically a joint, is not represented by means of differential equations, but it applies constraints to other objects. Classical connectors therefore have the following properties:

- Specific number of constraints
- Constraint equations
- Lagrange parameters according to the constraint equations
- Add force terms \( \left( \frac{\partial \mathbf{C}}{\partial \mathbf{q}} \right) \lambda \) to other bodies
- Graphical representation of the connector
- Index of the constraint equations

These properties are explained in the following using the example of a revolute joint.
The most important component of a connector is the set of algebraic constraint equations. The constraint equations need to be provided in a specific form, according to a requested index of the system. Some solvers of dynamical equations of motion are limited to index 2 DAEs, which means that position constraints need to be transformed to velocity constraints. The resulting system of equations is sensitive to drift off effects, which is compensated by small time steps. In the current implementation, several stabilization methods are available to certain bodies, however, the most general way to overcome the drift off which only occurs in the case of large time steps and long term simulations, is to use an index 3 solver, which is any solver based on RadauIIA methods with two or more stages. For the case of static problems, the position constraint equations need to be given in index 3 form. Thus, most conventional constraint equations are implemented as position and velocity constraint equations. In the case of the revolute joint, the constraint equation procedure is given in the following. The revolute joint has five local vectors that are given in local body coordinates of the first and the second body. The vectors $l_{r1}$, $l_{n2}$, and $l_{t2}$ are unit vectors. The residual of the five constraint equations for the revolute joint are written in the vector $f$:

```c++
void RevoluteJoint::EvalG(Vector& f, double t)
{
    //given local vectors:
    //lp1 ... local position vector in body 1
    //lp2 ... local position vector in body 2
    //lr1 ... local axis of rotation in body 1
    //ln2 ... local vector n normal to rotation axis in body 2
    //lt2 ... local second vector t normal to rotation axis in body 2

    if (Index()==3)
    {
        //compute relative vector between joint position in body 1 and 2:
        Vector3D v = GetBody3D(1).GetPos(lp1) - GetBody3D(2).GetPos(lp2);

        //compute local to global transformation for both bodies
        Matrix3D A1=GetBody3D(1).GetRotMatrix(lp1);
        Matrix3D A2=GetBody3D(2).GetRotMatrix(lp2);
        Vector rot = A1*lr1; //compute global rotation axis

        //return residuals of five index 3 constraint equations:
        f(1) = v(1);
        f(2) = v(2);
        f(3) = v(3);
        f(4) = rot*(A2*ln2);
        f(5) = rot*(A2*lt2);
    }
    else if (Index()==2)
    {
        //compute relative velocity between joint velocity in body 1 and 2:
        Vector3D v = GetBody3D(1).GetVel(lp1) - GetBody3D(2).GetVel(lp2);

        //compute local to global transformation for both bodies
        Matrix3D A1=GetBody3D(1).GetRotMatrix(lp1);
        Matrix3D A2=GetBody3D(2).GetRotMatrix(lp2);
        Matrix3D A1p=GetBody3D(1).GetRotMatrixP(lp1);
        Matrix3D A2p=GetBody3D(2).GetRotMatrixP(lp2);
        Vector3D rot = A1*lr1; //compute global rotation axis

        //compute time derivative of transformation
        Vector3D rot = A1p*lr1; //compute global rotation axis
    }
}
```
Vector3D rotp = A1p*lr1; //compute time derivative of rotation axis
//return residuals of five index 2 constraint equations:
  f(1) = v(1);
  f(2) = v(2);
  f(3) = v(3);
  f(4) = rot*(A2p*ln2)+rotp*(A2*ln2);
  f(5) = rot*(A2p*lt2)+rotp*(A2*lt2);
}
);

The constraint condition can only be fulfilled, if respective constraint forces, in the form of Lagrange multipliers, are applied to the constrained bodies. The constraint forces are added with the help of another function:

//add the generalized constraint forces to vector f
//locelemind is the local index of the constraint
//locelemind==1 ==> first body, locelemind==2 ==> second body
void RevoluteJoint::AddElementCqTLambda(int locelemind, Vector& f)
{
  //f has the size of the second order equations of the constrained body
  hmat.SetSize(f.Length(),5); //constraint matrix CqT
  double sign = 1;
  if (locelemind==2) sign = -1;
 getBody3D(locelemind).GetDPosDqT(loccords(locelemind),dpdq);
  hmat.SetSubmatrix(dpdq,1,1, sign);

  if (locelemind==1)
  {
    //compute global vectors n and t,
    Vector3D vn2_glob = GetBody3D(2).GetRotMatrix(lp2)*ln2;
    Vector3D vt2_glob = GetBody3D(2).GetRotMatrix(lp2)*lt2;

    //compute derivative of lr1 w.r.t. element DOF at position lp1
    GetBody3D(1).GetDAvDqT(lr1,lp1,dpdq);
    Mult(dpdq,vn2_glob,hvec);
    hmat.SetColVec(hvec,4);
    Mult(dpdq,vt2_glob,hvec);
    hmat.SetColVec(hvec,5);
  }
  else
  {
    //compute global rotation axis
    Vector3D vrglob = GetBody3D(1).GetRotMatrix(lp1)*lr1;

    //compute derivative of ln2 w.r.t. element DOF at position lp2
    GetBody3D(2).GetDAvDqT(ln2,lp2,dpdq);
    Mult(dpdq,vrglob,hvec);
    hmat.SetColVec(hvec,4);

    //compute derivative of lt2 w.r.t. element DOF at position lp2
    GetBody3D(2).GetDAvDqT(lt2,lp2,dpdq);
    Mult(dpdq,vrglob,hvec);
    hmat.SetColVec(hvec,5);
  }
}
//Vector lambda(5) contains Lagrange multipliers for revolute joint
for (int i=1; i <= f.Length(); i++)
    for (int j=1; j <= 5; j++)
        f(i) -= hmat(i,j)*lambda(j);
}

It is clear that the constraint procedures are more complicated for a redundant multibody dynamics formulation as compared to a minimum coordinates approach. However, once a joint is implemented and verified, it can be applied to every kind of rigid or flexible body without further restrictions.

3.7 Performance monitoring

In order to check performance of each function and subroutine, a performance monitoring is essential. While there are many compilers that can do this, it is convenient to see which parts of the multibody system need most time, such as the assembly of mass matrix, computation of residuals, computation of the Jacobian, factorization of the Jacobian, back-substitution of the residual into the factorized system, graphical processing, etc.

The main idea of the performance monitoring is, to see which parts of the simulation contribute most to the simulation time. Therefore, timer functions are attached to selected functions, which are stochastically evaluated during the simulation. After each simulation, the percentiles of the CPU-time for each measured function can be viewed in the output window. The continuous performance monitoring avoids speculations about more or less time consuming functions. Most code optimization has been made only at selected parts of the code. The performance monitoring consumes almost no computational time.

![Figure 4: Solution strategy in HOTINT.](image)

4 SOLUTION TECHNIQUES

The general solution strategy for differential algebraic equations is based on implicit time integration algorithms as depicted in Figure 4. Many different methods and options are avail-
able for the solution of the dynamic equations. In the end, each time step of the integration scheme leads to a set of nonlinear algebraic equations, which needs to be solved. The nonlinear solver is based on the Newton method. The Jacobian is either updated after each iteration (full Newton) or only after bad convergence of the nonlinear iterations (modified Newton). Additionally to the Newton method, a discontinuous iteration is performed for discontinuous variables, which include contact, plasticity or general jump conditions. As a new feature, the code can also perform a static simulation for the case that the system has no kinematical degrees of freedom.

The Newton solver automatically builds the residual and the Jacobian for each iteration on basis of the information provided by the single elements and the information about the connection of elements. The Jacobian matrix can be partially computed automatically or provided by special functions given in the elements. For the efficient solution of large scale systems, special band-solver techniques have been implemented for tree-like graphs of typical multibody systems. For dense finite element problems, the open-source solver SuperLU has been attached to HOTINT, which has been tested with systems that contain well beyond 100,000 unknowns.

4.1 Optimal Ordering of Global Coordinates Vector for Solver

For chain or tree-like multibody systems, it is known that the solution of one time step of an implicit time integration scheme can be obtained in a time of the order \(O(n)\), where \(n\) denotes the number of bodies in the multibody system. There are a large number of algorithms, which have been designed for that purpose and they are mostly based on redundant multibody formulations, initiated by Kane’s algorithm [6]. In the present code, a redundant formulation is used, however, for chain or tree-like multibody systems, the same performance can be achieved. The equations of motion of the redundant coordinate formulation lead to the following equations from an implicit time integration method

\[
\begin{pmatrix}
  aM + bK & C_q^T \\
  C_q & 0
\end{pmatrix}
\begin{pmatrix}
  \delta q \\
  \lambda
\end{pmatrix} =
\begin{pmatrix}
  F_F \\
  F_C
\end{pmatrix}
\]  

(5)

Here, \(M\) is the mass matrix, \(K\) is a stiffness-type matrix, \(a\) and \(b\) are constants of the time integration, \(C_q\) is the matrix of derivatives of the constraint equations with respect to the coordinates \(q\), \(\delta q\) is the increment for the time step, \(\lambda\) is the vector of Lagrange multipliers, \(F_F\) is the right hand side of the time integration for the second order equations and \(F_C\) is the right hand side for constraint equations. The latter two terms result from the incremental procedure. The system of equations (5) results in a sparse structure. Concerning the redundant coordinate formulation, the matrices \(M\) and \(K\) are block-diagonal. However, due to the matrix \(C_q\), the block-diagonal structure cannot be used for the solver. One way to obtain a fast solution of such a system is to use modern sparse solvers, such as SuperLU which is open source. However, such solvers include a large overhead and need to analyze the structure of the system of equations. Due to the simplicity of Eq. (5), it is easy to reorder the equations by reordering entries of \(C_q\) into the mass and stiffness terms. For that purpose, the constraints in the multibody system are analyzed by the following algorithm:

- First, it is assumed that the element numbers of the bodies are ordered in such a way, that in each chain or sub-chain, the elements are numbered consecutively
• There is a vector $u$ which has the length according to the number of elements, initialized with zeros
• There is a list $r$ of indices initialized with zero
• We go through all elements $i$ and
  o If element $i$ is not a constraint for which holds $u_i = 1$, add all coordinate indices of the element to the list $r$
  o If the element $i$ has constraints attached, take the first two constraints and for each of the two constraints $j$ do
    • if $u_j \neq 1$, if number of elements that the constraint constrains is less or equal two, and if the difference in the two element numbers is less than one, then add the constraint coordinate (Lagrange multiplier) indices to the list $r$ and set $u_j = 1$

After this basic algorithm, a list $r$ of resorted coordinate and Lagrange multiplier indices is available that can be used by the solver to resort equations such that a banded structure is achieved for all chain-systems. For a small number of loops or for elements with more than two constraints attached, there remain some entries in a new matrix $\bar{C}_q$. However, the system can be solved by splitting the matrix into four parts and building the Schur complement, which again leads to a order $O(n)$ for the algorithm if the number of loops is not proportional to the number of elements. The resulting equations read

$$\begin{pmatrix} A & \bar{C}_q^T \\ \bar{C}_q & 0 \end{pmatrix} \begin{pmatrix} \delta \bar{q} \\ \bar{\lambda} \end{pmatrix} = \begin{pmatrix} \bar{F}_F \\ \bar{F}_C \end{pmatrix}$$

(6)

A bar indicates that the terms are resulting from the reordering of the generalized coordinates and the Lagrange multipliers. The mass and stiffness matrix is reordered into the matrix $A$. Note that the size of $\bar{\lambda}$ depends on the number of loops and branchings which is small, and the number of rows of $\bar{C}_q$ is therefore small as well. The solution of Eq. (6) is obtained by the Schur complement, which reads

$$\bar{\lambda} = Q^{-1} \left( \bar{F}_C - \bar{C}_q A^{-1} \bar{F}_F \right)$$

$$\delta \bar{q} = A^{-1} \left( \bar{F}_F - \bar{C}_q \bar{\lambda} \right)$$

with

$$Q = -\bar{C}_q A^{-1} \bar{C}_q^T$$

(7)

Here, the factorization of $A$ is cheap because of its block-diagonal structure. The factorization of the constraint matrix $Q$ is also cheap, because it has very small dimension.

### 4.2 Time Integration Without Factorization of the Mass Matrix

General implicit Runge Kutta methods are used to solve the DAEs, see Eq. (1), of the multibody system. The tableaus of various classes and different orders (up to 20) are available in text files. Typical Runge Kutta schemes, such as Gauss, LobattoIIIA, LobattoIIIC, and RadauIIIA are available. The low order representations of these schemes include classical integration schemes, such as midpoint rule, implicit Euler, implicit trapezoidal and others. The higher order versions of LobattoIIIC and RadauIIIA allow the direct solution of index 2 and even index 3 problems. Most times, an index 2 reduction of constraint conditions is used, because the drift is negligible and the condition of the system is less sensitive to the step size.
The Runge-Kutta methods are written in the so-called K-form. This means that the unknowns stage variables of the implicit methods are not position-based but velocity (and acceleration) based, similar as to classical Newmark scheme in the trapezoidal rule. Consider rewriting Eq. (1) into a system of first order DAEs. The unknowns of each stage are then chosen to be $\mathbf{v} = \mathbf{K}_{iv}, \mathbf{y} = \mathbf{K}_{iy}$ and $\mathbf{z}_i$. Then the nonlinear equations for every stage $i$ with step size $\tau$ are given by

$$
\begin{bmatrix}
(\mathbf{K}_{iq}) \\
\mathbf{M}_{iv} \mathbf{K}_{iv} \\
\mathbf{K}_{iv} \\
0
\end{bmatrix}
\begin{bmatrix}
(\mathbf{v}_i) \\
\mathbf{f}_2(\mathbf{q}_i, \mathbf{v}_i, \mathbf{y}_i, \mathbf{z}_i) \\
\mathbf{f}_1(\mathbf{q}_i, \mathbf{v}_i, \mathbf{y}_i, \mathbf{z}_i) \\
\mathbf{g}(\mathbf{q}_i, \mathbf{v}_i, \mathbf{y}_i, \mathbf{z}_i)
\end{bmatrix}
= 
\begin{bmatrix}
v_i \\
\mathbf{v}_0 + \tau \sum_{j=1}^{n} A_{ij} \mathbf{K}_{jq} \\
\mathbf{q}_0 + \tau \sum_{j=1}^{n} A_{ij} \mathbf{K}_{jq} \\
\mathbf{y}_0 + \tau \sum_{j=1}^{n} A_{ij} \mathbf{K}_{jq}
\end{bmatrix}
\quad \text{with}
\begin{bmatrix}
v_i \\
\mathbf{v}_0 + \tau \sum_{j=1}^{n} A_{ij} \mathbf{K}_{jq} \\
\mathbf{q}_0 + \tau \sum_{j=1}^{n} A_{ij} \mathbf{K}_{jq} \\
\mathbf{y}_0 + \tau \sum_{j=1}^{n} A_{ij} \mathbf{K}_{jq}
\end{bmatrix}
\quad (8)
$$

Note that $\mathbf{K}_{iq} = \mathbf{v}_i$ is eliminated from these equations, which are solved at every stage of the Runge Kutta scheme. For stability reasons in combination with algebraic equations or with contact-impact conditions, it is preferable to use Runge Kutta schemes that have a stage located at the end of the time step. This is the case for RadauIIA or LobattoIIIC schemes. The Jacobian is the computed for one stage of the Runge Kutta scheme and then used as a Jacobian for all stages. As can be seen from Eq. (8), it is not necessary to compute the factorization of the mass matrix for the residual or for the Jacobian, which is different for standard time integration methods, that are based on position and velocity unknowns.

5 CONCLUSIONS

Some details about the multibody system simulation software HOTINT have been presented. The software is composed of a Windows user interface, numerical solvers for static and dynamic problems and a multibody kernel, which allows the easy handling and extension of objects in the multibody system. Some of the features of the redundant multibody formulation have been shown, especially the implementation of bodies and joints. Furthermore, some information about the solution procedures has been given. The advantages of the solution procedures are based on a reordering of the equations of motion for tree-like multibody systems. A second advantage is the avoidance of factorization of the mass matrix in the implicit time integration methods. An extension of the present freeware version of the software with the ability to add user-defined elements is planned in the near future.

ACKNOWLEDGEMENTS

The support of the author by the K2 – Austrian Center of Competence in Mechatronics (ACCM) for being able to publish this paper is gratefully acknowledged. Significant contributions with respect to certain elements and the windows interface of the original version have been made by Dr. Michael Stangl, Dr. Yury Vetyukov, Mr. Markus Dibold and Rafael Ludwig, which are gratefully acknowledged.

REFERENCES


