Version 1.2.41

December 23, 2013

Contents

1 General Information **5**

2 HOTINT Developers Manual **11**
- 2.1 General Information . . . 12
  - 2.1.1 Program Structure . . . 12
  - 2.1.2 Entry Points - Where to start from? . . . 17
  - 2.1.3 Description of the multibody/mechatronic system in HOTINT . . . 19
  - 2.1.4 Algorithmic structure of the integrator . . . 23
  - 2.1.5 Dense and sparse matrices . . . 25
  - 2.1.6 Integration Window . . . 26
  - 2.1.7 Vectors, Matrices, Nonlinear Solver . . . 26
  - 2.1.8 Graphical commands (optional): . . . 28
  - 2.1.9 Autogenerated parts of HOTINT . . . 29
- 2.2 The Multibody System Kernel . . . 34
  - 2.2.1 Several main points have been focused in the multibody kernel: . . . 34
  - 2.2.2 Main structure of the multibody kernel . . . 34
  - 2.2.3 Multibody system: mbs.h . . . 35
- 2.3 Sensors . . . 37
  - 2.3.1 Sensors and Measuring . . . 37
  - 2.3.2 Deflection Sensor . . . 42
- 2.4 Loads . . . 44
  - 2.4.1 Loads . . . 44
  - 2.4.2 Add momentum to finite element . . . 44
- 2.5 Elements . . . 45
  - 2.5.1 Base class Element: element.h . . . 46
  - 2.5.2 Bodies . . . 67
  - 2.5.3 The base class `Constraint` . . . 67
  - 2.5.4 Rigid3D . . . 71
  - 2.5.5 Geometric Elements . . . 74
  - 2.5.6 Finite elements . . . 74
  - 2.5.7 CMSElement + GCMSElement . . . 78
  - 2.5.8 Control elements (time continuous and discrete elements) . . . 83
- 2.6 Import and export of data and interfaces to other software . . . 84
  - 2.6.1 FEMesh . . . 84
  - 2.6.2 Generate cylinders, rings and discs . . . 92
- 2.7 Models . . . 95
  - 2.7.1 How to add objects to the mbs . . . 95
  - 2.7.2 Possibilities for error handling . . . 97

3 HOTINT Guidelines **99**
- 3.1 How to get HOTINT running from the Source Code . . . 100
  - 3.1.1 Requirements . . . 100
  - 3.1.2 Unpack HOTINT . . . 100
  - 3.1.3 When using Visual Studio 2008 . . . 100
  - 3.1.4 When using Visual Studio 2010 . . . 100
- 3.2 Coding Conventions . . . 101
  - 3.2.1 Comments . . . 101
  - 3.2.2 How to structure new HOTINT code? . . . 105
  - 3.2.3 Efficiency . . . 108
- 3.3 If problems appear . . . 110
  - 3.3.1 If problems appear in HOTINT . . . 110
  - 3.3.2 Solutions for problems, which might be caused by yourself (programming errors): . . . 110
  - 3.3.3 Tips for debugging . . . 110
  - 3.3.4 Known frequent problems . . . 111
- 3.4 How to create an installer . . . 113
  - 3.4.1 How to create an installer and set up your system for the first time in VisualStudio . . . 113
  - 3.4.2 How to create an installer for a specific project . . . 119
- 3.5 How to include Intel MKL . . . 120
- 3.6 How to include Pardiso (Solver) . . . 121
- 3.7 How to include BLAS/LAPACK . . . 122
- 3.8 Useful Visual Studio settings . . . 123
  - 3.8.1 Save and Build before start . . . 123
  - 3.8.2 Expansion of class instances (variables) in the watch window . . . 124
- 3.9 AutoComplete and Highlight HOTINT syntax in notepad++ . . . 124
- 3.10 How to add your Element to the Script Language . . . 125
  - 3.10.1 Changes in the h-file of your element . . . 125
  - 3.10.2 Use the EDC-converter: . . . 128
  - 3.10.3 Test your element: . . . 128
  - 3.10.4 error handling . . . 128
  - 3.10.5 The results . . . 128
- 3.11 How to add Sensors/Nodes/Materials/BeamProperties/ to the Script Language . 129
  - 3.11.1 Test your element: . . . 129
- 3.12 How to start a HOTINT computation from command line or Matlab . . . 130

Bibliography **131**

Chapter 1

General Information

Introduction

Development history and background information

The code HOTINT has been initiated by Johannes Gerstmayr in 1997 and, until now, gone over the following steps:

- solution methods and basic linear algebra routines for static solver (diploma thesis of the main developer, 1997)
- addition of time integration methods for the accurate solution of large-scale flexible and discontinuous multibody systems (up to 2004)
- integration with graphical interface in 2003 (with Yury Vetyukov)
- implementation of various structural finite elements, such as flexible beam and plate elements based on the absolute nodal coordinate formulation
- implementation of the floating frame of reference concept, as well as the component mode synthesis
- HOTINT made available to and further developed by Linz Center of Mechatronics (since 2007)
- HOTINT made available to and further developed by Austrian Center of Competence in Mechatronics (from 2008 to 2013)
- User version of HOTINT V1.1 available as freeware (2013)
- A open source version of HOTINT is available (end of 2013)

Current State of HOTINT

HOTINT mainly consists of the multibody kernel, the solver and linear algebra kernel, and the graphics and user interface, and currently comprises several hundred thousand lines of code. It has been particularly developed for the use of arbitrary classes of fully implicit Runge Kutta (IRK) methods. The IRK-tableaus can be defined in an external text-file and are given for several methods for 1 to 10 stages. The code makes advantage of the very high order reached through the use of fully implicit methods, which makes it especially then fast, when higher

accuracy is needed.
In the current version, the K-form of IRK-equations has been implemented for the fast integration of 2^{nd} order (mechanical) systems. Instead of trying to invert the mass matrix, which leads to large terms in the case of symbolic inversion, or instead of trying to add the system as a constraint equation (this has been done by some people who implemented their system into existing codes), you can now provide the mass matrix and the right hand side separately and the solver only solves one large system, but does not need the accelerations to be written explicitly as function of the remaining unknowns.

Summarizing, advanced methods from flexible multibody dynamics cover

- the efficient geometrical description for moving rigid bodies and bodies with superimposed small deformation,
- the application of special finite element methods, which are well suited for simulating large deformations of structural elements,
- high-order implicit time-integration schemes, in order to enforce stability for the numerical solution,
- a sophisticated treatment of algebraic equations for the arbitrary coupling of bodies, and for the incorporation of certain (boundary) conditions,
- and finally the reduction of the system size by a component mode synthesis (CMS).

General Information

Chief developer

Johannes Gerstmayr

Further developers

Larissa Aigner, Markus Dibold, Alexander Dorninger, Peter Gruber, Alexander Humer, Rafael Ludwig, Karin Nachbagauer, Astrid Pechstein, Daniel Reischl, Martin Saxinger, Markus Schörgenhumer, Michael Stangl, Yury Vetyukov

Contact

support@hotint.org

Linz Center of Mechatronics GmbH
Altenbergerstr. 69, 4040 Linz, AUSTRIA
http://www.lcm.at

Thanks

The help and support from the contributors of the Institute of Technical Mechanics and Institute of Numerical Mathematics at the Johannes Kepler University of Linz is greatly appreciated.

I would like to acknowledge the important grant of the FWF (”Fond zur Förderung Wissenschaftlicher Forschung” - the Austrian National Science Fund) within the project P15195-N03 and the APART project of the Austrian Academy of Sciences.

Parts of this software have been developed in the project ”Nachhaltig ressourcenschonende elektrische Antriebe durch höchste Energie- und Material-Effizienz” (sustainable and resource saving electrical drives through high energy and material efficiency) which is part of the European Union program ”Regionale Wettbewerbsfähigkeit OÖ 2007-2013 (Regio 13)” sponsored by the European Regional Development Fund (ERDF) and the Province of Upper Austria.

Parts of this software have been developed with the support of the Comet K2 Austrian Center of Competence in Mechatronics (ACCM).

Link

`http://www.hotint.org`

Copyright and licence

HotInt 1.0
======

Copyright (c) 2012 Johannes Gerstmayr, Linz Center of Mechatronics GmbH, Austrian Center of Competence in Mechatronics GmbH, Institute of Technical Mechanics at the Johannes Kepler Universitaet Linz, Austria. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.

- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ”AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This program contains LAPACK 3.3.0 and SuperLU 4.3, covered under the following licenses:

LAPACK 3.3.0
======

Copyright (c) 1992-2011 The University of Tennessee and The University of Tennessee Research Foundation. All rights reserved.
Copyright (c) 2000-2011 The University of California Berkeley. All rights reserved.
Copyright (c) 2006-2011 The University of Colorado Denver. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.

- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ”AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

SuperLU 4.3
=======

Copyright (c) 2003, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from U.S. Dept. of Energy)

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

(1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
(2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
(3) Neither the name of Lawrence Berkeley National Laboratory, U.S. Dept. of Energy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ”AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAM-

AGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Chapter 2

HOTINT Developers Manual

2.1 General Information

2.1.1 Program Structure

2.1.1.1 The basic modules of HOTINT

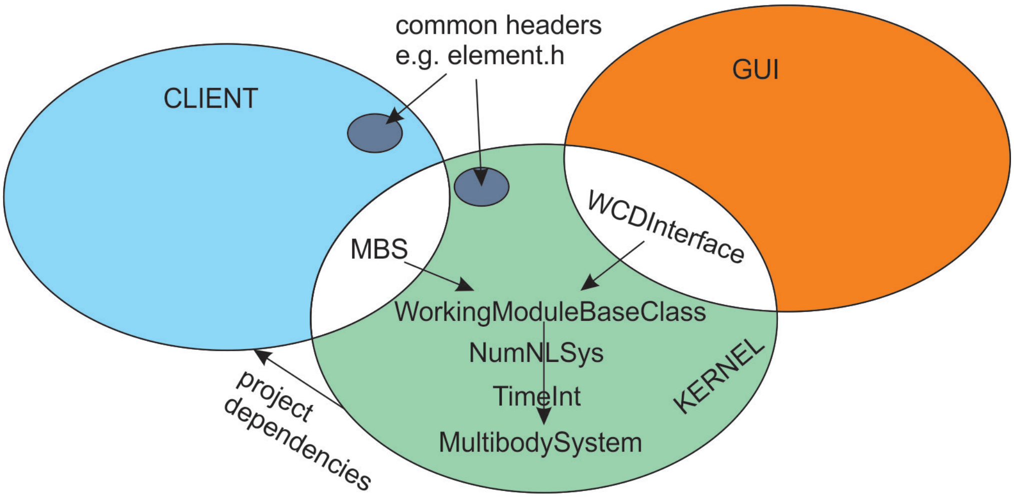


Figure 2.1: Interfaces between the kernel, the client and the graphical user interface

HOTINT is composed of the following 3 main modules:

1. kernel module leading to WorkingModule.dll
2. client module leading to MBSElementsAndModels.dll
3. GUI module (by now only MS Windows is supported) leading to hotint.exe

Additionally there exist common services (collected in the static libraries UtilityLib, Parser and SuperLU), which are linked statically to both WorkingModule and MBSElementsAndModels.

2.1.1.1.1 Kernel - WorkingModule.dll In the kernel, the multibody system is handled. Here the system matrices are assembled and stored, and the system equations are solved. The communication with the GUI module is also done by the kernel.
The module consists of the parts (projects in visual studio)

- WorkingModule
  - WCDInterface (in WinCompDriverInterface.h): contains the interface of the module (start/end computation, SetElement, ...)
  - WorkingModuleBaseClass: the base class for the computation, derived from WCDInterface
- MBSKernelLib
  - NumNLSolver
  - NumNLSys - Numerical nonlinear solver - derived from MBS and WorkingModuleBaseClass
  - TimeInt - derived from NumNLSys

  - MultiBodySystem - derived from TimeInt
  - Eigenmodes solver
- Parser
- SuperLU
- UtilityLib
  - Mathematical routines (linear algebra)
  - String manipulation functions
  - IO routines

2.1.1.1.2 Client - MBSElementsAndModels.dll In the client, the special information and code of the objects is provided. Each object (element, load, material, ...) adds some terms to the system equations and provides other informations (e.g. for drawing) for the multibody system.
The module consists of the (projects in visual studio):

- MBSElementsAndModels (core of the client module)
  - MBSModelLibrary, described in more detail below
  - MBSObjectFactory, for creation of objects from the kernel, described in more detail below
  - includes all interfaces needed for elements and models (in the folder MBS_Interface)
- ElementsLib (collection of all elements/objects available for defining models) contains
  - all elements (rigids, beams, CMS, ...)
  - all connectors (joints, control, ...)
  - IntegrationRule.h, for defining custom integration rules for specific elements
- ModelsLib
  - cpp-models defined in own files (my_model.cpp), which shall be included in all_models.h of this project
  - source files may be attached to this project, but shall be excluded from build!
- ServiceObjectsLib (collection of all service objects (objects apart from elements and connectors) available for defining models)
  - GeomElements
  - Material
  - Loads
  - Nodes
  - Sensors
  - Simulink2HotintConversion
  - TCP/IP-Routines

- FEMesh
    - FEMesh
    - FEMesh_aux
    - FEMesh_generator
- Parser
- SuperLU
- UtilityLib
    - Mathematical routines (linear algebra)
    - String manipulation functions
    - IO routines

2.1.1.1.3 GUI module (hotint.exe) This module contains the HOTINT GUI.

- WCDriver
    - CWCDriver3DApp (in WCDriver3D.cpp): First class that will be created when you start HOTINT.
    - CWCDriver3DDlg (in WCDriver3DDlg.cpp): Creates the GUI and manages the user input (e.g. if a button is clicked).
    - WCDriver3D.rc: Resource file in which all dialogs can be found. Here one can add or modify a dialog or a dialog element.
    - every dialog

2.1.1.1.4 Common headers Collected in the project folder MBSElementsAndModels/MBS_Interface are common headers, which are needed in more than one particular module.

NumSolverInterface

- GetNewtonIts()
- NumDiffepsi()
- UseSparseSolver()
- SymmetricJacobian()

HOTINTOptions defined in the (autogenerated) file options_class_auto.h with the subclasses

- LoggingOptionsClass
- GeneralOptionsClass
- ViewingOptionsClass
- PostProcOptionsClass

- GraphicsOptionsClass
- PlotToolOptionsClass
- PreProcOptionsClass

SolverSettings defined in the (autogenerated) file solversettings_auto.h

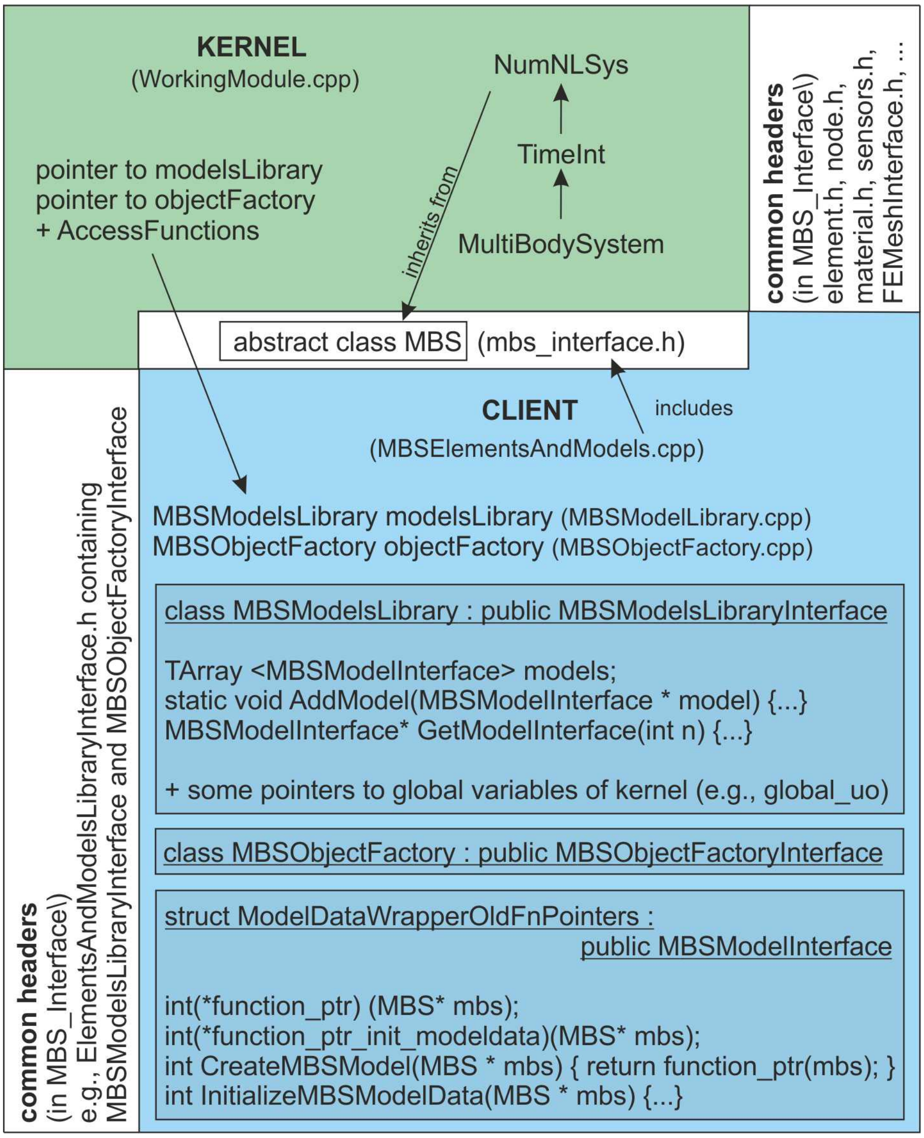
RenderContext Interface to OpenGL.

ControlWindowContext Interface for drawing in the control window.

StepSettings

2.1.1.2 Interaction between client and kernel module

In Fig. 2.2 the interaction between client and kernel module is outlined.



```
MBSModelsLibrary modelsLibrary (MBSModelLibrary.cpp)
MBSObjectFactory objectFactory (MBSObjectFactory.cpp)

class MBSModelsLibrary : public MBSModelsLibraryInterface

TArray <MBSModelInterface> models;
static void AddModel(MBSModelInterface * model) {...}
MBSModelInterface* GetModelInterface(int n) {...}

+ some pointers to global variables of kernel (e.g., global_uo)

class MBSObjectFactory : public MBSObjectFactoryInterface

struct ModelDataWrapperOldFnPointers :
                                      public MBSModelInterface

int(*function_ptr) (MBS* mbs);
int(*function_ptr_init_modeldata)(MBS* mbs);
int CreateMBSModel(MBS * mbs) { return function_ptr(mbs); }
int InitializeMBSModelData(MBS * mbs) {...}
```

Figure 2.2: Interaction between kernel (multi-body-system) and client (elements and models)

2.1.1.2.1 Models Library The models library contains all models available for being exicuted by the user via the GUI.

How source code models are added to the models library

- A new source code model is defined in form of a Generate- and a Init-function (in a cpp-file in `ModelsLib\` which shall be included in `ModelsLib\all_models.h`).
- Call of `ModelFunctionAutoregistration(*pFn,*pFnInit)` creates an instance

  ```
  ModelDataWrapperOldFnPointers model(..., (*pFn), (*pFnInit),...);
  ```

  and adds it to the model library: `modelLibrary.AddModel(model);`

How script models are added to the models library

How models from the models library are called by the kernel In the kernel, i.e., in `MultiBodySystem::Initialize` in `initialize_system.cpp`, the Generate-function of a selected (ith) cpp-model is finally called:

```
GetModelsLibrary()->GetModelInterface(i)->CreateMBSModel(this);
```

where `*this` is an instance of `class MultiBodySystem`.

2.1.1.2.2 Object Factory

2.1.1.2.3 MBS The abstract `class MBS` provides access on multibody system functionality (defined in the kernel module) from within the client module. While functions are declared on both sides (since `mbs_interface.h` is included in client and kernel module), their definitions are solely part of the kernel module (in the inherited `class MultiBodySystem`).
Most of the functions of `class MBS` are declared in sub-structures:

- MBSSolutionAccessInterface: GetXact, GetDrawValue,...
- MBSObjectsAccessInterface: AddElement, GetElement,...
- MBS3DDrawingInterface: MyDrawLine, SetColor,...
- MBS2DDrawingInterface: AddDrawComponent_Line,..
- MBSOptionsInterface: GetIOption, GetOptions, EDC2SolverOptions,...
- MBSUOInterface: UO(int message_level)
  - struct UserOutputInterface is defined in seperate file useroutputinterface.h
- MBSModelDataInterface: ReadModelData, GetModelDataContainer,..
- MBSParserInterface: ExpressionToDouble, File2EDC, ComputeInertia,...

while some of the functions are contained in `class MBS` itself:

- GetTime, GetDrawTime,...

- Assemble, ClearSystem,...

- some functions due to ancf..2d (GetDrawResolution, FEColor,...)

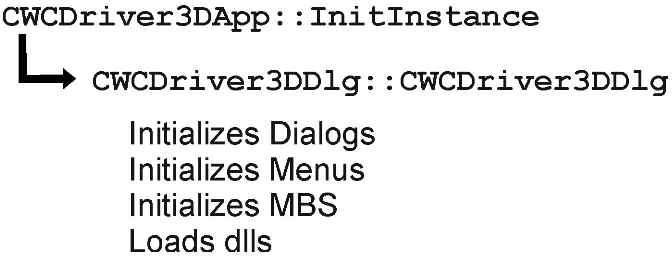
- some functions due to contact2D/3D and control (GetStepRecommendation, ForceJacobianRecomputation,...)

2.1.2 Entry Points - Where to start from?

In HOTINT there are several entry points, that is functions that get called from outside of HOTINT.
It is important to know which function is called from which entrypoint, since the different entrypoints may be called from different threads.

2.1.2.1 `CWCDriver3DApp::InitInstance()`

InitInstance will create a `CWCDriver3DDlg` which will draw all windows and which will initialize everything.



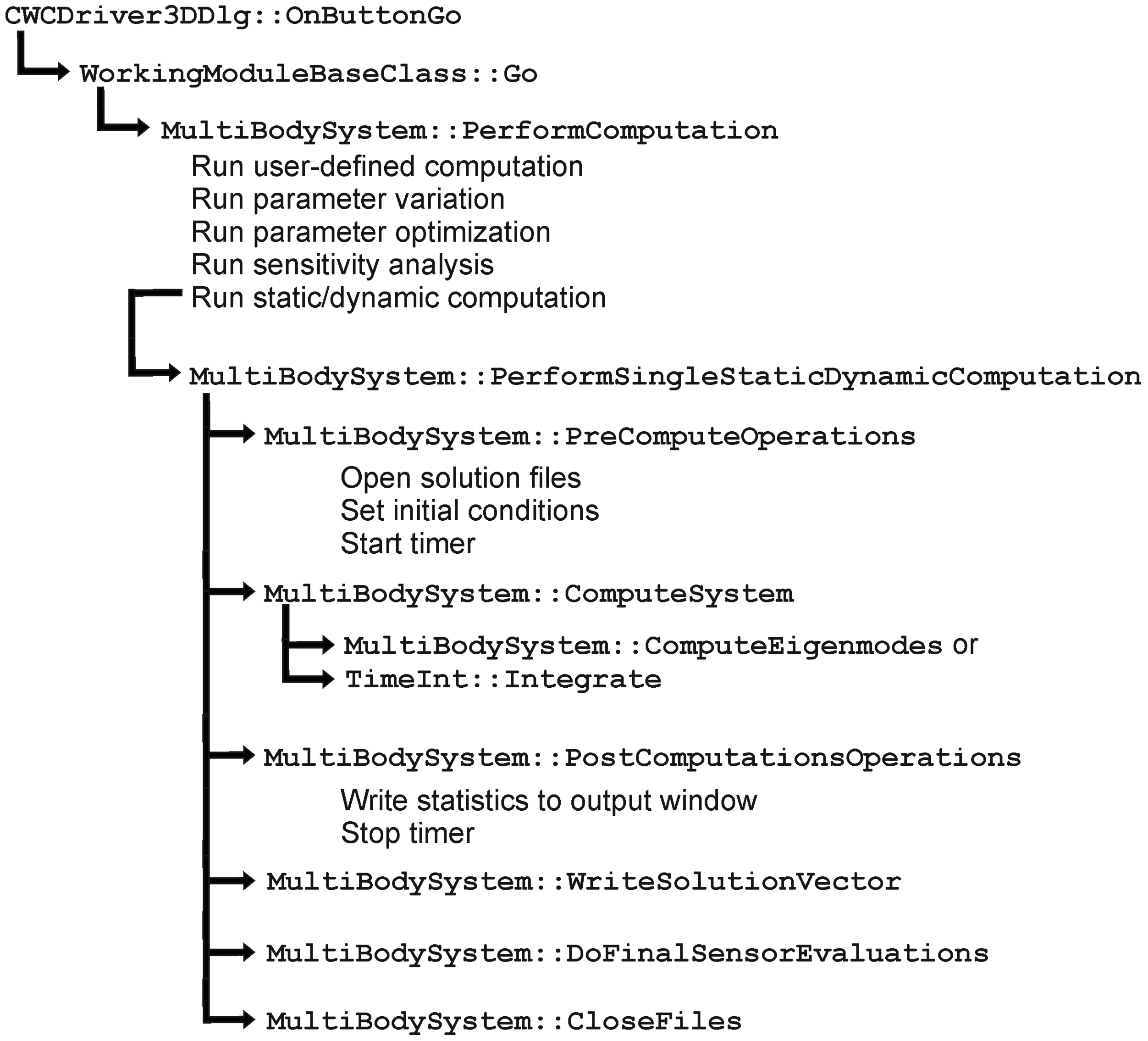
CWCDriver3DApp::InitInstance

CWCDriver3DDlg::CWCDriver3DDlg

Initializes Dialogs
Initializes Menus
Initializes MBS
Loads dlls

2.1.2.2 `WCDriver3DDlg::OnButtonGo`

Will be called when the Start button is clicked.
Attention when stopping a (user defined) calculation (intentionally or if an error occures): All files that have been opened have to be closed.

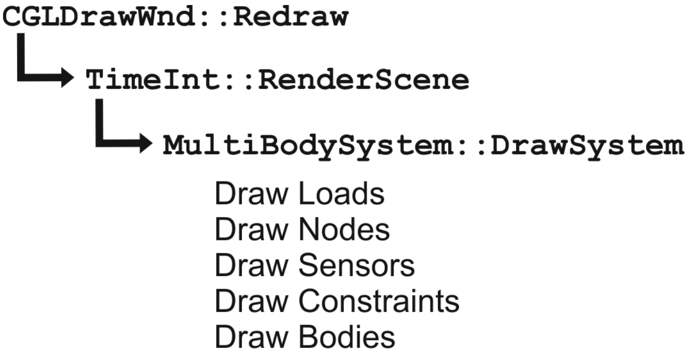

CWCDriver3DDlg::OnButtonGo
WorkingModuleBaseClass::Go
MultiBodySystem::PerformComputation
Run user-defined computation
Run parameter variation
Run parameter optimization
Run sensitivity analysis
Run static/dynamic computation
MultiBodySystem::PerformSingleStaticDynamicComputation
MultiBodySystem::PreComputeOperations
Open solution files
Set initial conditions
Start timer
MultiBodySystem::ComputeSystem
MultiBodySystem::ComputeEigenmodes or
TimeInt::Integrate
MultiBodySystem::PostComputationsOperations
Write statistics to output window
Stop timer
MultiBodySystem::WriteSolutionVector
MultiBodySystem::DoFinalSensorEvaluations
MultiBodySystem::CloseFiles

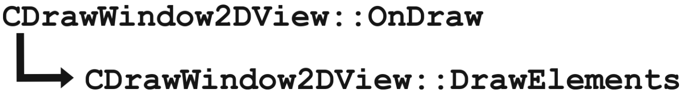
2.1.2.3 `CGLDrawWnd::Redraw()`

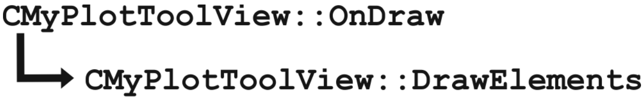
Will be called from Windows. This function calls then `RenderScene` which will call `MultiBodySystem::DrawSystem`, which then calls the `DrawElement()` function of every element in the model. The order in which the elements are drawn is

1. Loads
2. Nodes
3. Sensors
4. Constraints
5. Bodies

The order in which the elements are drawn is crucial if transparency is set to on.
The drawing routines are thus called from another entrypoint than the calculation routines. Hence it is important that the drawing routines and the calculation routines are independent from each other, and do not use the same variables. The drawing of the IOWindow and the Plottool Window works very similar.







2.1.3 Description of the multibody/mechatronic system in HOTINT

The idea is that you define your problem by deriving a class from the time integration class (which needs to be able to call the functions of your differential equations). While the program works almost automatically, you need to provide some functions which tell the program what it should do with your equations.

2.1.3.1 Commands used for description of differential algebraic system:

Parameters for the following Functions are all the same: `void EvalF(const Vector& x, Vector& f, double t);`
Here, `x` is a `Vector` of the length of your state vector, which **must** be of the form $\mathtt{x} = [\mathbf{x}_u\ \mathbf{x}_v\ \mathbf{x}_x\ \mathbf{x}_z]$, where $\mathbf{x}_u$ and $\mathbf{x}_v$ are position and velocity coordinates of the second order differential equations, $\mathbf{x}_x$ stands for variables of the first order differential equ. and $\mathbf{x}_z$ represents algebraic variables. The initial conditions and the evaluation of the below listet functions must be always sorted according to the vector `x`. For the development of a model, I recommend to first define the vector of your degrees of freedom, e.g. if you have the system (this example includes 2nd-order, 1st-order and algebraic equations but has **no** physical meaning!!!)

$$\begin{aligned} M_1\ddot{x}_1 &= -K_1x_1 - C_1\lambda F_1 \cdot x_3 \\ M_2\ddot{x}_2 &= -K_2x_2 + C_2\lambda F_2 \\ \dot{x}_3 &= R_I \cdot x_1 + R_P \cdot (\dot{x}_1) \\ x_1^2 + x_2^2 - L &= 0 \end{aligned} \tag{2.1}$$

Then your vector shall be sorted as

$$\mathtt{x} = [\mathbf{x}_1\ \mathbf{x}_2\ \dot{x}_1\ \dot{x}_2\ x_3\ \lambda] \tag{2.2}$$

what means, that $\mathbf{x}_1$ and $\mathbf{x}_2$ are your 2nd order position variables, $\mathbf{x}_u$, $\dot{x}_1$ and $\dot{x}_2$ are the 2nd order velocity variables, $\mathbf{x}_v$, x_3 is the first order variable, $\mathbf{x}_x$ and λ is the algebraic variable, $\mathbf{x}_z$, because there exists no differential equation for it! The description for the useage of `Vector` follows in Section 2.1.7.

- `void EvalF(...)` ... Evaluate first-order differential equations: $\dot{\mathbf{x}} = \mathbf{F}_1(\mathbf{x}, t)$; for every first order equation, the value of the evaluation of $\mathbf{F}_1$ must be returned in `Vector& f`;
- `void EvalM(...)` ... Evaluate mass matrix of second order differential equations: $\mathbf{M}\ddot{\mathbf{x}} = \mathbf{F}_2(\mathbf{x}, t)$
- `void EvalF2(...)`... Evaluate rhs of second order differential equations (external, elastic and gyroscopic forces, damping
- `void EvalG(...)`... Evaluate algebraic equations: $\mathbf{G}(\mathbf{x}, t) = 0$

2.1.3.2 Necessary Procedures which must be provided:

- `double GetFirstOrderSize() const return` *first order size*;
  ... provide the size (number of) first order equations
- `double GetSecondOrderSize() const return` *second order size*;
  ... provide the size (number of) second order equations
- `double GetImplicitSize() const return` *implicit size*;
  ... provide the size (number of) implicit or algebraic equations

2.1.3.3 Optional Commands and Parameters:

- `virtual double GetError() const;`
  ...This function should be adapted to the needs of your problem. This function shall compute the function (e.g. a norm) of your degrees of freedom, where you want to apply the error control (only adaptive step size). If you are e.g. only interested in certain quantities like the deformation of a certain point, but you are not interested e.g. in the velocities of another point (which might cause very small time integration steps), you should just return the quantities which should undergo the error control. Furthermore, if you would like to use a norm of all degrees of freedom, you should normalize your quantities as much as possible, otherwise a very small deformation will not be penalized, while a large Lagrange-multiplicator will be penalized extremely high.
- `void WriteSol();`
  ...This function is called at every output request; you can store your solution at this point for the desired degrees of freedom your own format.
- `double PostNewtonStep(double t);` and `void PostprocessingStep();`
  ...If you want to solve a problem with severe nonlinearities (or even jumps) like in contact or friction, you can use this procedures in order to switch between two states (e.g. between free motion and contact, or stick and slip, switch between plastic and elastic behaviour, ...). The function `NonlinStep` is called after the step is computed. You can e.g. then decide, if for example in contact penetration occurs and you have to switch from free motion to contact (you need to introduce switching variables). You have to return the error (which could be e.g. penetration depth) of the nonlinear functions. If the error is large than the `nonlinerror`, the integration step will be repeated taking into account the changes you made. If you just want to switch your configuration without repeating, just return 0. If you have no nonlinearities, simply return 0. The function `FixNonlinStep` will be called after the timestep has been integrated (with nonlinearity) and the nonlinear error is within the given tolerance.

Then you can reset e.g. your switching variables or you can store your plastic variables as new initial plastic variables.

- `DiscontinuousAccuracy()` and `MaxDiscontinuousIt()`:
  The maximum number of iterations for the `NonlinStep` is `MaxDiscontinuousIt()` and the desired accuracy is `DiscontinuousAccuracy()`. This is meant that you for example model a contact problem, and you return the error of the contact step, e.g. penetration or negative contact force. Allowing small errors for these parameters makes it possible for a problem with many discontinuities to converge without making the timestep extremly small.

- `SaveState` and `RestoreState`
  ...If you make use of the `NonlinStep` and you add switching variables, you have to store this values (into a copy of variables) or restore them in the Restore-function. E.g. if you use a contact variable, which indicates you, whether you are in contact or free motion state, say `iscontact`, you need to introduce the variables `iscontact` and `iscontact_copy` in your `TestInt` class; Then you modify your functions to `SaveState() {iscontact_copy = iscontact;}` and `RestorState() {iscontact = iscontact_copy;}`. These functions are called, if a timestep is repeated during step changes and error estimation. The function is not needed for constant stepsize.

- **Newton method**: There are several parameters for specifying the behaviour and accuracy of the Newton method. Changing them or adapting them to a specific problem might help to improve convergence:

  - `NLS_ModifiedNewton()=...`: Set to 0 or 1 in order to activate the modified Newton method. Generally, a problem should always converge for the modified Newton method, because it changes to the full Newton method as soon as it starts to fail for the modified method. However, you might with to analyse your problem, or it might be just better for some problems to immediately try the full Newton method.
  - `NLS_RelativeAccuracy()=...`: Sets the relative accuracy ($rtol$) of your Newton method. The Newton method iterates until it fulfills $R_i(F) < rtol \cdot R_0(F)$ where $R_i(F)$ is the the actual residual and $R_0(F)$ is the the initial residual when starting the method. For some problems the desired accuracy might not be reachable (default: 10^{-10}), therefore you should change this value, if the method does not converge!
  - `NLS_NumDiffepsi()=...`: This is the parameter for the numerical differentiation in the computation of the Jacobian. You should normally set this between 10^{-6} and 10^{-10}. The smaller this value gets, the more accurate the differentiation becomes, but the higher the rounding error becomes (which has its extremum at 10^{-17} where you get probably 100% error. If you use larger values (10^{-6} to 10^{-4}), you more and more smooth your Jacobian, which means that small perturbations in your sysetm are not taken into account in the Jacobian. You should keep in mind that parameter does not have influence to the accuracy of the solution you get from the Newton method. It only influences the convergence speed, which might be sometimes faster, if you smooth a little, or sometimes better if you try to compute your Jacobian more accurately.
  - `NLS_MaxModNewtonSteps()=...`: This sets the maximum number of modified Newton steps. At the moment. If the number of modified Newton steps is reached without convergence or the contractivity becomes very bad, the Newton solver restarts solving the nonlinear problem with once computing a Jacobian and iterating until `MaxRestartNewtonSteps()`. When still having no convergence, it will try the full Newton method with in total `MaxFullNewtonSteps()`.

  - `NLS_SymmetricJacobian()=...`: Set to 1, if you would like to compute the Jacobian by a symmetric difference quotient, or to 0, if you want to solve it with a right-side difference quotient. The symmetric difference quotient is approximately 2 times slower, but might be better for certain problems.

- **Output**: you can activate or deactivate the call of the function write sol by setting the parameter `writeresults`. The actual state of the system can be gained from `GetSolVector()` (at the end of the step), the actual computed time comes with `GetTime()` and the actual stepsize was `GetStepSize()`. Additional functions are the actual computational time `GetCompTime()`, the number of time steps `GetTIit()`, the number of computed Jacobians `GetJacCount()`, the number of step size changes `GetStepChanges()`.

- **Data Storage**: you can activate or deactivate the data storage function by setting the parameter `storedata`. The data can be only stored when it is painted, so be sure to set the parameter `withgraphics` to 2. The parameters for `storedata` are
  - -2: store at every time step
  - -1: store at every max-timestep
  - 0: never store
  - $+x.x$: store every $x.x$ time; note that it is only stored if drawn.

- **Drawing**: The parameter `withgraphics` lets you influence how often your model is redrawn. This might help, to increase the speed of your simulation. The Values are:
  - 0: never draw
  - 1: draw every 25 milliseconds (of real time)
  - 2: draw every frame (use this for data storage)
  - 3: draw every 100 milliseconds (of real time)
  - 4: draw every 10 steps

2.1.3.4 Start of integration

- `void PerformComputation();` ... This Procedure is called when you press the `Start` button. You have to call the function `TIInit()`, which initializes the time-integrator. This is not done automatically, because you might wish to run your simulation for several times and maybe do not wish to initialize it all the time. It initializes integration parameters with default values. After calling `TIInit()`, you should set your specific integration parameters (method, accuracy, time-span) and start the integration → call `Integrate(...)`.

- `void Initialize();` ... Set initial conditions with `SetStartVector(const Vector& x0)`, and initialize special parameters for integration or numerical solver. This function is furthermore called prior to the computation, such that your model can be drawn with initial conditions before pressing the start button. Everything should be set in Initialize() so that the DrawScene function can be performed after the startup of the program.

- `virtual int Integrate(mystr name, int maxstages, double endtime, double maxstepsize, int adaptivestepsize, double initstepsize, double minstepsize, double relaccuracy, double absaccuracy);` ... Start the integration of your specific system, parameters are:

  - `name`: name of method (Gauss, RadauIIA, RadauIA or method defined by you).

  - `maxstages`: number of stages used for integration.

  - `endtime`: time-span which you would like to solve

  - `adaptivestepsize`: set to 1 if you want to adaptively integrate; if set to 0, the `maxstepsize` is used as a constant step size without error control for the time integration; all further parameters only concern adaptive stepsize integration

  - `initstepsize`: initial stepsize for adaptive stepsize integration

  - `minstepsize`: minimum of allowed stepsize

  - `relaccuracy`: relative accuracy for error control (not active now)

  - `absaccuracy`: absolute accuracy for error control. Note that this accuracy is for each time step and it is checked due to comparison with two smaller timesteps. For standard mechanical problems which are not chaotic (e.g. double pendulum) or contact or generally discontinuous problems the solution of the final timestep might be much less accurate than the specified tolerance for every timestep. The error for the final timestep is found by comparison of two computations with different tolerances.

Note that it is important that the file `tableau.txt` which contains the IRK-coefficients must be in the same directory as your program file (in the example it is `Release/WCDriver3D.exe`).

2.1.4 Algorithmic structure of the integrator

- This Section describes, what the Integrator does. The algorithms should not be changed and might be fully reorganized in future versions! For a description of the solution procedure, see the paper of Gerstmayr and Stangl, APM [3], found on the web-page of HOTINT (see above).

- The algorithms are located in: `timeint.cpp, timeint.h`

- `Integrate`: Integrate a certain time-interval

  - call function `Initialize` in order to set initial conditions. The initial conditions of the algebraic variables (Lagrange parameters) do not influence the solution, they might only influence your drawing of the system in the first step (e.g. if you draw forces).
  - choose method and stages
  - set all integration parameters
  - Convert initial conditions to integration variables
  - initialize all graphical and drawing activities during integration

- do until endtime is reached: Call `FullAdaptiveStep` (for adaptive step size) or `FullStep` (for constant step size)
- `FullAdaptiveStep`:
  - choose stepsize (Check for next desired output-time position, at max-step size or endtime).
  - compute one nonlinear and discontiuous step: `NonlinSubStep`; store values; reset;
  - compute two nonlinear and discontiuous half steps: 2 × `NonlinSubStep`;
  - estimate error from two solutions
  - if error is within tolerance, eventually suggest bigger step size for next step and call finish step: `FinishStep`;
  - if error is not within tolerance, suggest smaller stepsize (computed from error, actual step size and desired tolerance), repeat step.
- `FullStep`: constant stepsize without error checking
  - compute one nonlinear and discontiuous step: `NonlinSubStep`;
  - finish step: `FinishStep`;
- Do one nonlinear and discontiuous step: `NonlinSubStep`
  - Do one IRK step: `GeneralIStep`
  - Correct nonlinearities or discontinuities: `NonlinStep`
  - if `DiscontinuousAccuracy()` not fullfilled by `NonlinStep`, go to `GeneralIStep` again.
  - if converged, call `FixNonlinStep` and return.
  - if not converged, tell `FullAdaptiveStep` that timestep shall be repeated with smaller step size.
- `GeneralIStep`: Performs a general IRK with the chosen method and number of stages, defined by tableau specified in the file "`tableau.txt`". Note, that you can basically write your own `tableau.txt` file, but you should know that the solver always considers the method to be implicit and that it will not speed up the solution if you e.g. include DIRK or SDIRK methods. The function currently does not work with Lobatto methods or any method where $c_1 = 0$ (this includes explicit methods). The function `GeneralIStep` does the following operations:
  - Set the initial conditions for the solution of the nonlinear IRK-equations for one timestep, based on the results last time step. Two formulations of the nonlinear equations are possible: The so-called g-form and the K-form, where the K-form is default and perform significantely better than the implemented version in the g-form, especially for second order systems. The Newton solver is initialized and `NLSolve` is called to solve the IRK equations. The Newton solver uses needs a `Jacobian` and the nonlinear IRK-function, which is given by `NLF`.
  - The results for the stages are used to compute the full step and set the initial conditions for the next step.

- `NLF`: Nonlinear IRK-equations. This function is called for the computation of the residual of the nonlinear IRK-equations and for the computation of the full `Jacobian`. It puts together in a quite complicated way the IRK-equations out of the given equations of motion. The operations are performed such that no new Vectors are generated and copying is kept at a minimum.

- `Jacobian`: There are two ways to compute the Jacobian: Either it is computed by numerical differentiation of the function `NLF`, which is quite expensive. Otherwise, an approximate Jacobian can be computed, which uses the same Jacobian for every stage of the Runge-Kutta method, as described in the Radau5 code of Hairer and Wanner. However, it furthermore makes it possible to neglect some parts in the second order system which should improve the speed for very complicated mass matrices. The approximate Jacobian is built on the Jacobian of the Mass matrix `JacobianM`, of the right hand side of the second order system `JacobianF2`, the Jacobian of the first order system `JacobianF`, and the Jacobian of the Constraint equations `JacobianG`.

- variable order strategy: not included in this version

2.1.5 Dense and sparse matrices

The time integration can be performed in two modes, the mode with full matrices and a sparse mode. The mode using full matrices is slower, but more general and it can be used with the full Newton method in every step. The mode using sparse matrices can be much faster for larger systems, however, it only approximates the Jacobian for the Newton iterations.

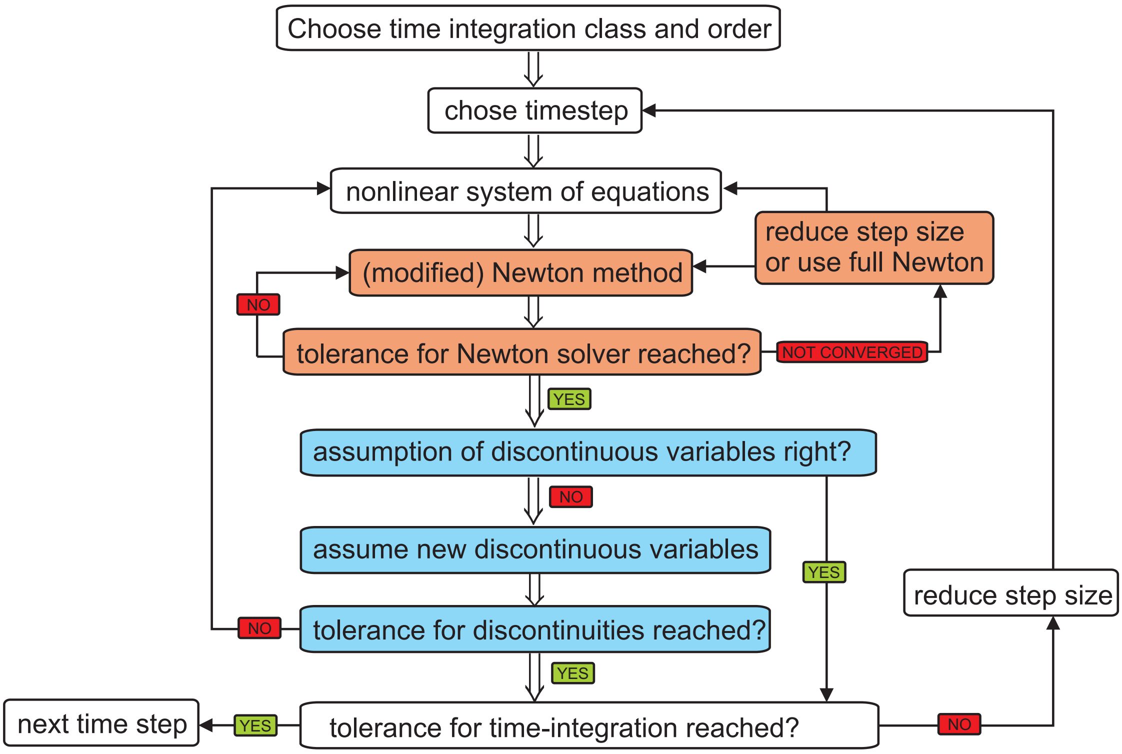

Choose time integration class and order
chose timestep
nonlinear system of equations
reduce step size
or use full Newton
(modified) Newton method
NO
tolerance for Newton solver reached?
NOT CONVERGED
YES
assumption of discontinuous variables right?
NO
assume new discontinuous variables
YES
reduce step size
NO
tolerance for discontinuities reached?
YES
next time step
YES
tolerance for time-integration reached?
NO

Figure 2.3: Structure of the dynamic solver in HOTINT.

2.1.6 Integration Window

Once you started your integration with the graphical window, you mainly have two parts. To the left you can see the text which you output during your simulation with the `uo` operator. On the right you can see the drawing window.
You can rotate, move or zoom by means of the right mouse button, respectively the left mouse button, respectively the mouse wheel and moving around. Alternatively it is possible to use only the right key and press the shift or ctrl key for zooming respectively moving.
Pressing the Start button will execute the Procedure `PerformComputation`. After that, the Symbol changes to `Stop` which will be active until your simulation has been finished, which means that the command `pCFB->FinishedComputation();` has been called.
The `Data manager` lets you store the results of your simulation, replay a simulation or save to or load from the harddisk, special functions need to be provided for the use of this function.
You can store all your graphical settings by means of `save configuration`.
With `Read text` you can read the text in a separate window which is not updated.
Next, there are some buttons concerning the 3D-view of the system. On the right side, there is a button for saving frames (for animations). If you want to record a video follow the steps presented in the HOTINT Guidelines.

2.1.7 Vectors, Matrices, Nonlinear Solver

Almost all linear algebra and nonlinear solver classes, functions, etc. are placed in the files `femath.cpp` and `femath.h` (finite-element-mathematics). There are 3DVectors, general Vectors and Matrices including (Gaussian-based solvers) and a nonlinear solver. You should not touch these files, because they might be improved or changed in future implementations. You might need some of the functions or classes for developing your model and for drawing.
The nonlinear solver computes the full inverse of the Jacobian and needs at least N^3 operations, if N is the number of unknowns. This might be a little annoying for very large systems ($N > 500$, therefore it will be replaced by a LU-decomposition in the future and sparse matrices are considered to be implemented.

- `Vector`: class for a arbitrary length vector; The vector allocates its memory by itself. This operation is slow on most computers, so be sure that Vectors are not generated in each step of your integration. If you need Vectors or Matrices, just place them in the class variables so that they are generated only once. Default and copy Contructor, assign operator exist.
    - `Vector(int n)` gives you a vector of length n, initialized with zeros
    - `operator(int i)` or `operator[int i]`: This operators, such as `v(i);` give you access to the i-th entry of your vector, either by value or reference. The indexes for `Vector` of length n are from 1 to n for both operators () and [].
    - `Norm()`, `MaxNorm()`, `MinNorm()`: give you the quadratic norm ($\mathbf{v}^T\mathbf{v}$), Min or Max norm of the vector.
    - `operator +, -, =, ==, +=, -=, *=, *`: Vector and Vector-Matrix operators are defined according to linear algebra standards.
    - `GetLen`, `Length` : give you the length n of the vector.
    - `SetAll(double x)`: Sets all values of the vector to x.

  - `SetLen(int i)`: Change the length of a vector to length i.
  - `FillWithZeros()`: Sets all element to zero.

- `Matrix`: class for a arbitrary size full matrix; The matrix allocates its memory by itself. This operation is slow on most computers, so be sure that matrices are not generated in each step of your integration. If you need Vectors or Matrices, just place them in the class variables so that they are generated only once. Default, copy constructor and assign operator exist.
  - `Matrix(int rows, int cols)`: generate Matrix with # rows *rows* and # columns *cols*. Entries filled with zero.
  - `operator(int row, int col)`: Access to Element row *row*, column *col* of your matrix. Indexing gos from $1 \ldots rows$ and $1 \ldots cols$.
  - `operator +, -, =, ==, +=, -=, *=, *`: Vector and Vector-Matrix operators are defined according to linear algebra standards.
  - `GetTp()`: returns a new Matrix which is the transposed original one.
  - `TpYs()`: transposes the Matrix
  - `Det()`: Returns the determinate of the Matrix, works only for diagonal matrices, size $<= 3$.
  - `FillWithZeros()`: Sets all element to zero.
  - `MaxNorm()`, `MinNorm()`: Minimum and Maximum norm.

- `Vector3D`: This is a Vector with only 3 components. The components are stored on the stack, therefore creating and deleting of a Vector3D is fast. This class has access functions similar to `Vector`.
  - `Vector3D(double x, double y, double z`: Create a 3D-Vector with components x, y and z.
  - `operator(int i)`: The () operator such as `v(i)` gives you access to the i-th entry of your vector, either by value or reference. The range for a `Vector3D` of length n with operator() is from 1 to n.
  - `operator[int i]`: The [] operator such as `v[i]` gives you access to the $(i+1)$-th entry of your vector, either by value or reference. The range for a `Vector3D` of length n with operator[] is from 0 to $n-1$.
  - `Norm()`, `MaxNorm()`, `MinNorm()`: give you the quadratic norm $(\mathbf{v}^T\mathbf{v})$, Minimum or Maximum norm of the vector.
  - `operator +, -, =, ==, +=, -=, *=, *`: Vector and Vector-Matrix operators are defined according to linear algebra standards.
  - `GetLen`, `Length` : give you the length (number of entries) n of the vector, which is always 3.

- `X()`, `Y()`, `Z()`: Access to the 3 components of the vector, useful for graphics.
- `Normalize()`: divides all components by the geometrical length (`Norm()`) of the vector.
- `Cross(Vector v2)`: Computes the cross-product of the this-vector with vector v2.

2.1.8 Graphical commands (optional):

HOTINT provides some graphical commands, which are directly included into `timeint.cpp` and implemented in `ti_misc.cpp`. This graphical commands will help you to set up any multibody system very easily and verify results or make impressive animations. The graphics is based on OpenGL and draws everything in 3D. However, some functions help you, to draw 2D or beam-type models very easily!

- `void DrawSystem();`: Here, you should place your drawing functions. This function is called according to the updates of the system, the time integration and your settings
- Always use `GetDrawVector()` to get the actual solution vector for drawing. This vector is provided from the graphical driver and can give you the solution which you have once stored in a file. `GetDrawTime()` should be used as the actual time for drawing.
- There are some helpful functions (based on OpenGL) implemented in HOTINT:
  - `virtual void MyDrawLineH(const Vector3D& p1, const Vector3D&p2, const Vector3D& vy2, double t, double h, int drawouterface=1)`:
    draws a line which is considered to lie in the xy-plane. `p1` and `p2` are the endpoints of the line (the z-coordinate must be equal for both points). The vector `vy2` should indicate the direction of the normal of the face at point `p2`. The parameters `t` and `h` indicate the depth and height of the line. The reason for `vy2` is, that you might want to draw a line which is put together from several segments (as in a deformed beam). Then you set `drawoutersurface` to 0 and define `vy2` as the direction of the right-hand (following) line-segment. Then your 3D drawing of a beam or circle will be more continuous than otherwise.
  - `virtual void MyDrawLine(const Vector3D& p1, const Vector3D&p2, double t, double h, int drawouterface=1)`: draws a line. works like `MyDrawLineH`, but `vy2` is defined as `p2-p1`.
  - `void DrawHex(const Vector3D& p1,const Vector3D& p2,const Vector3D& p3, const Vector3D& p4, const Vector3D& p5,const Vector3D& p6,const Vector3D& p7,const Vector3D& p8, int drawouterfaces=1)`:
    draws a hexahedron (cube) with endpoints `p1 - p8`.
  - `DrawZyl(const Vector3D& pz1,const Vector3D& pz2, double rz, int tile=8)`:
    Draws a zylinder with axis-end points `pz1` and `pz2`, radius `rz` and the tiling (over the circumference) is `tile`. Use tile=20 or above for smooth graphics, or 8 and less for slow graphics processors.
  - `DrawSphere(const Vector3D& p1, double r, int tile=8, double fill=1)`:
    Draws a shpere, midpoint `p1`, radius `r`, tiling `tile`. The fill parameter indicates, how much is painted from the sphere: 1 ... full sphere, 0.5 ... half-sphere, etc.
  - `SetColor(const Vector3D& col)`: Set the RGB-values of the actual drawing color. Predefined colors are:

    - `colred, colred1, colblue, colblue1, colblue2, colgreen, colbrown, colgrey 1, colgrey2, colgrey3, colgrey4`

  For more drawing functions see `ti_misc.cpp`.

- There are some additional commands provided by the graphical driver WCDriver3D:

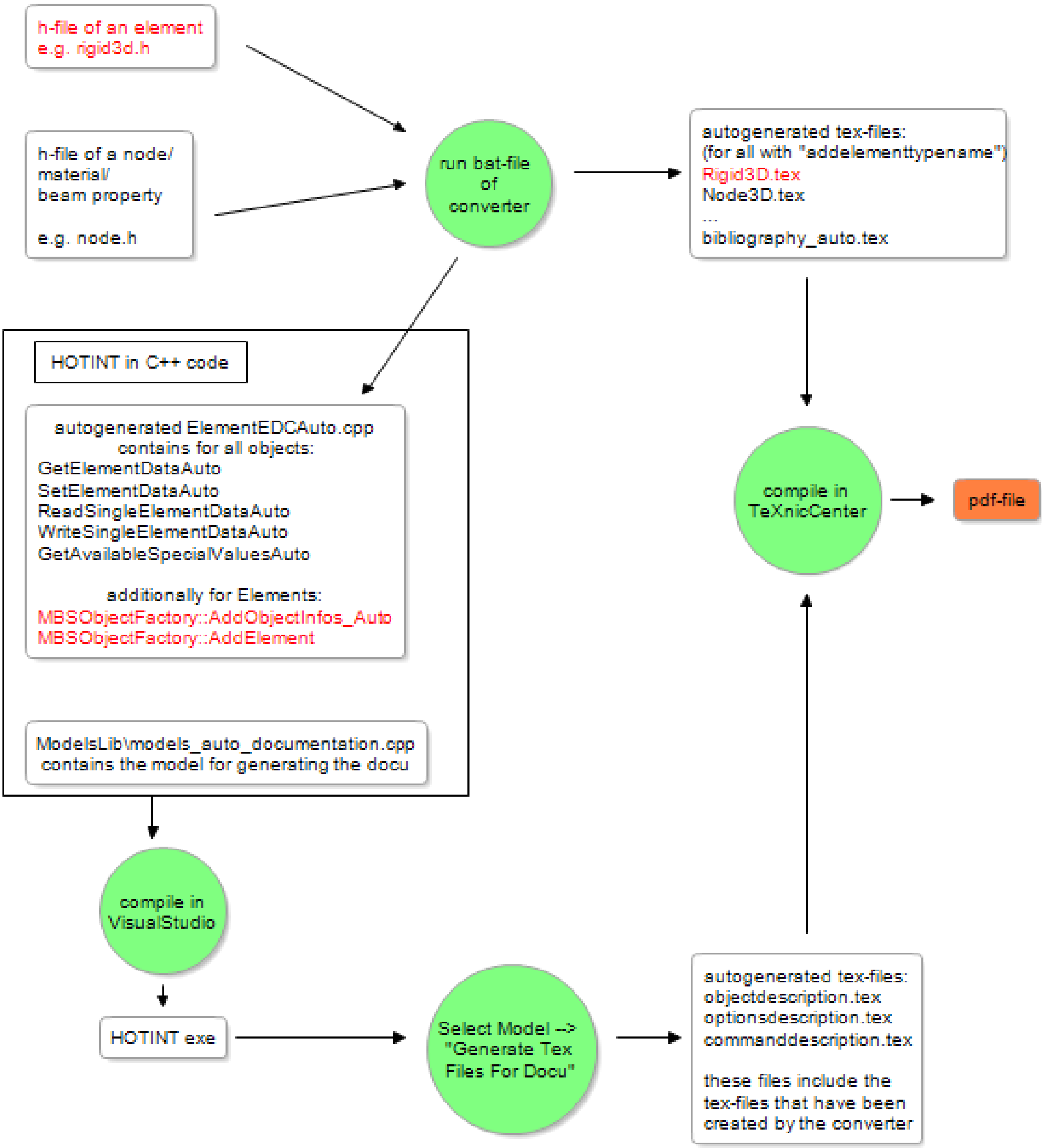
  - `GetRC()->PrintTextStruct(int nLineNo, int nXPos, const char * text)`:
    prints a Text `text` at the `nLineNo`th line of your 3D-window, `nXPos` defines the alignment (-1 ... left, 0 ... center, +1 ... right alignment). The function `RenderContext* GetRC()` returns the interface structure of the graphical driver.

2.1.9 Autogenerated parts of HOTINT

Several parts of HOTINT are generated automatically by the use of tools. These autogenerated parts should not be changed directly, since the changes would be overwritten the next time the tools for autogeneration are started. In the following these concepts are described in detail:

1. tex-files for the documentation of each object
2. reference manual of the documentation itself
3. GetElementDataAuto and SetElementDataAuto
4. ReadSingleElementDataAuto and WriteSingleElementDataAuto
5. GetAvailableSpecialValuesAuto
6. MBSObjectFactory::AddElement
7. MBSObjectFactory::AddObjectInfos_Auto()

All of the information, that is used to autogenerate code in HOTINT is stored in the h-files of the objects. Figure 2.1.9 gives a short overview about the flow of information and the autogenerated files.


h-file of an element
e.g. rigid3d.h
h-file of a node/ material/ beam property
e.g. node.h
run bat-file of converter
autogenerated tex-files:
(for all with "addelementtypename")
Rigid3D.tex
Node3D.tex
...
bibliography_auto.tex
HOTINT in C++ code
autogenerated ElementEDCAuto.cpp contains for all objects:
GetElementDataAuto
SetElementDataAuto
ReadSingleElementDataAuto
WriteSingleElementDataAuto
GetAvailableSpecialValuesAuto
additionally for Elements:
MBSObjectFactory::AddObjectInfos_Auto
MBSObjectFactory::AddElement
ModelsLib\models_auto_documentation.cpp
contains the model for generating the docu
compile in TeXnicCenter
pdf-file
compile in VisualStudio
HOTINT exe
Select Model --> "Generate Tex Files For Docu"
autogenerated tex-files:
objectdescription.tex
optionsdescription.tex
commanddescription.tex
these files include the tex-files that have been created by the converter

2.1.9.1 tex-files for the documentation of each object

By running the converter (bat-file in EDC_onverter \release) a seperate tex-file for each object is generated. The files are stored in "documentation\EDCauto_documentation\classDescriptions". Additionally the file "bibliography.tex" is generated in this step.
The converter uses the information specified in section 3.10.

2.1.9.2 reference manual of the documentation itself

The reference manual of the (user) documentation is generated automatically by loading the model "ModelsLib\models_auto_documentation.cpp". This model generates the following files:

1. objectdescription.tex
2. optionsdescription.tex
3. commanddescription.tex

To generate these files, each object is added to the mbs with the default constructor and then the ElementData is obtained. In a next step, for elements, a list of possible constraints is generated.
The file objectdescription.tex additionally includes the autogenerated tex-files of each object.

2.1.9.3 GetElementDataAuto and SetElementDataAuto

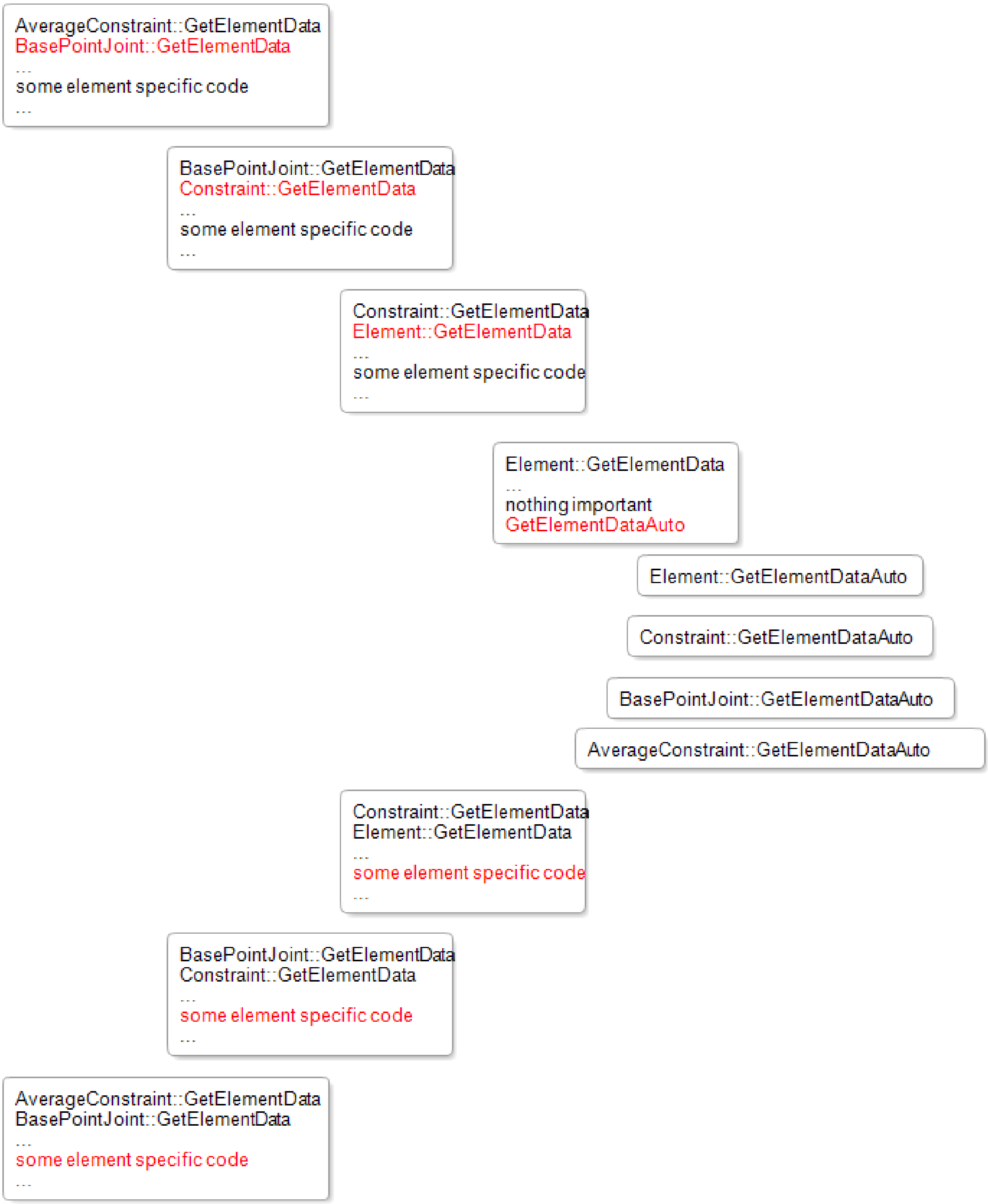
By running the converter (bat-file in EDC_converter\release) the file "ElementsLib\ElementEDCauto.cpp" is generated. In this file, beside others, the functions

- GetElementDataAuto
- SetElementDataAuto

are generated.
This functions get the data of the user (hid-file or GUI) and set the variables of the object (SetElementData), or vice versa (GetElementData).
These autogenerated functions for each object are just a part of the following larger concept:
The function GetElementData (without Auto!) calls the function GetElementData of the parent class, until the cascade reaches Element::GetElementData.
In Element::GetElementData the first time the function GetElementDataAuto is called. This autogenerated functions again first calls the parent class and then execute the object specific code.
The order of the code is therefore:

1. autogenerated function of the parent class(es)
2. autogenerated function of the object itself
3. the rest of the function GetElementData (without Auto!) of the parent class(es)
4. the rest of the function GetElementData (without Auto!) of the object itself

For SetElementDataAuto, the concept and order is exactly the same!
Compare figure 2.1.9.3


AverageConstraint::GetElementData
BasePointJoint::GetElementData
...
some element specific code
...
BasePointJoint::GetElementData
Constraint::GetElementData
...
some element specific code
...
Constraint::GetElementData
Element::GetElementData
...
some element specific code
...
Element::GetElementData
...
nothing important
GetElementDataAuto
Element::GetElementDataAuto
Constraint::GetElementDataAuto
BasePointJoint::GetElementDataAuto
AverageConstraint::GetElementDataAuto
Constraint::GetElementData
Element::GetElementData
...
some element specific code
...
BasePointJoint::GetElementData
Constraint::GetElementData
...
some element specific code
...
AverageConstraint::GetElementData
BasePointJoint::GetElementData
...
some element specific code
...

2.1.9.4 MBSObjectFactory::AddElement

By running the converter (bat-file in EDC_converter\release) the file "ElementsLib\ElementEDCauto.cpp" is generated.
In this file, the function MBSObjectFactory::AddElement provides the functionality to add an element to the mbs.
This is used for the script-language and the GUI.

2.1.9.5 MBSObjectFactory::AddObjectInfos_Auto()

By running the converter (bat-file in EDC_converter\release) the file "ElementsLib\ElementEDCauto.cpp" is generated.
In this file, the function MBSObjectFactory::AddObjectInfos_Auto() adds some basic information about the element to the object factory.
This may be used for some online help in the future.

2.2 The Multibody System Kernel

A general framework for systems consisting of first and second order differential equations and algebraic equations has been set up. The present code is based on a redundant coordinate formulation for the modeling of the motion and deformation of bodies. This means that e.g. every rigid body has its own six degrees of freedom (DOF), no matter how this body is constrained by other bodies or even if it is fixed to the ground. The main reason for this formulation is the simple extensibility of the code regarding the development of new elements, constraints, forces, etc. . The numerical efficiency is gained by adapted solvers for the sparse structure of the system equations, which leads to a similar effort as in recursive and minimal coordinate approaches.

2.2.1 Several main points have been focused in the multibody kernel:

- The application of implicit time integration algorithms shall be efficient
- The code shall be capable of structural and solid finite elements
- The code shall be extendable and open for new elements (e.g. non-mechanical, variable mass, variable topology, etc.)

Some things you should know:

- Dimensions: dimensions are chosen by user, but should use standard international units: kg/m/s.
- Numbering: All lists, arrays or other ordering numbers start with 1 if not specified differently. Some of the arrays and vectors have two possibilities: index from 1 to n with operator(), and index from 0 to $n-1$ with operator[].
- Elements: Bodies and connectors are elements. If you search for bodies or connectors (e.g. for editing) in the HOTINT program, you should look for elements.

2.2.2 Main structure of the multibody kernel

There were some main points to be fulfilled with the present multibody kernel:

- The formulation shall be easily accessible and maintainable via C++ functions
- The formulation shall be easily accessible and maintainable via the Windows user interface.

In the current implementation there is one base multibody system object which contains all information about the system. On top of this structure, there is a dynamic and static solver class, i.e. an implicit time integration method and an incremental nonlinear solver. The solver requires the multibody system to provide residuals and derivatives of the differential and algebraic equations based on assumed values.

2.2.2.1 Solution vector

The multibody system and solver always have two solution vectors. One containing either the initial vector or the actual solution (this is the solution vector) and another one that is used for the graphics drawing which is called drawing solution vector. The latter vector is utilized to independently draw the solution of a certain computed time instantly during the computation (e.g. if the computation lasts very long or is of indefinite length). The solution vector is split into a "position level" (not necessarily a real position) and "velocity level" part for the case of the second order differential variables. Assume that there are n second order differential equation variables, then the solution vector will contain first n position level coordinates and after that n velocity level coordinates. The local coordinates of a body (e.g. accessible via the sensor) are ordered in a similar way. The local second order differential variables of a body contain first m position level coordinates and after that another m velocity level coordinates.

2.2.2.2 Structure of the multibody system

The multibody system consists of the following components:

- Elements
- Nodes
- Loads
- Sensors
- Geometric elements

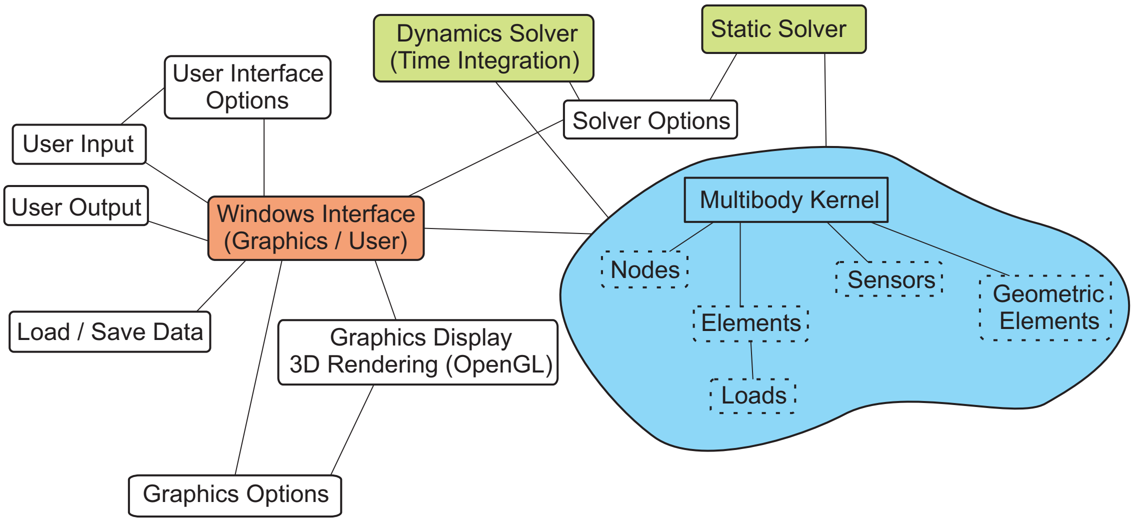


Figure 2.4: Multibody system core and windows interface.

2.2.3 Multibody system: mbs.h

The class `MBS` is based on the time integration HOTINT. `MBS` provides all information about equations, nonlinearities and the structure, which HOTINT needs. `MBS` contains basically only objects of the type `Element`. An `Element` must provide the amount of virtual work to each generalized coordinate, the number of unknowns (can be zero) for algebraic, first and second order

differential equations, as well as information about nonlinearities and about the dependency on generalized coordinates (GC). Elements are added to the system `MBS`, and after finishing, the call `Assemble()` will provide global GC to the local Element GC. Forces which act onto the element are only known to the element itself. Constraints are Elements as well, but provide only algebraic constraints. For the computation of the Jacobian, it is very important, that every element (including contraints) knows its dependency on the global degrees of freedom.
For USERS: the MBS class members are not intended to be changed, except for the `Generate....()` functions. They could be changed by deriving a `class MyMBS` from `class MBS` and overriding or extending some member functions. The functionality of the multibody code should be mainly extended by adding new elements to the system, which are derived from the `class Element`.

2.3 Sensors

2.3.1 Sensors and Measuring

Sensors are used to measure certain quantities of the multibody system at the cur-rent state of the computation. The output of a sensor is usually written to output files at certain time steps (See Computation Settings dialog). The solution file `'sol.txt'` 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 `local_DOF_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 first $[1 \ldots m]$ position level coordinates and another $[m+1 \ldots 2m]$ velocity level coordinates. Sensors can not have own generalized coordinates (unknowns).

The following flags define the type and properties of the sensor. Only certain (natural) combinations of the sensor are possible:

- `TSempty`: no sensor
- `TSElement`: Element sensor, usually measures properties of an element at a certain position given in poslist
- `TSNode`: Nodal sensor, measures properties of nodes (displacement, position, velocity, etc.)
- `TSPos`: Measure position quantities
- `TSVel`: Measure velocity quantities
- `TSAngle`: Measure angles
- `TSX`: Measure x-component of a quantity
- `TSY`: Measure y-component of a quantity
- `TSZ`: Measure z-component of a quantity
- `TSDist`: Measure the distance of two points
- `TSDeflection`: Measure the deflection of a line represented by 3 points and one normal vector to the deflection
- `TSDOF`: Measure a degree of freedom
- `TSplanar`: The sensor is planar
- `TSAuxElem`: The sensor measure quantities of auxiliary elements
- `TSAccelDOF`: Measure the acceleration of degrees of freedom
- `TSMultSensor`: The sensor is composed of several sensors
- `TSLocalAxis`: Measure quantities with respect to co-rotated body axes

Examples for possible sensor flags:

- `TSElement+TSPos+TSX`: Measure the x-component of a position within an element
- `TSNode+TSVel+TSY`: Measure the y-component of a velocity of an element
- `TSElement+TSAngle+TSX+TSLocalAxis`: Measure the rotation at a point of an element with respect to the local x-axis
- `TSElement+TSDeflection`: Measure the deflection of defined by three positions of one or several elements and one direction defining the normal to the deflection plane
- `TSElement+TSPos+TSplanar+TSX`: Measure the x-component of the position within a planar element
- `TSDOF`: Measure a certain degree of freedom of an element

For the case of Multiple Sensors, a sensor returns values which follow from an algebraic manipulation of the results of several sensors:

- `TSMAverage`: Average of all sensors
- `TSMSum`: Sum of all sensors
- `TSMMult`: Multiply all results of sensors
- `TSMDiv`: Return sensor1/(sensor2*sensor3*...)
- `TSMNorm`: Return quadratic norm of all sensors
- `TSMDifference`: Return sensor1-(sensor2+sensor3+...)

2.3.1.1 Class MBS Sensor

- `MBSSensor(): elementlist(), nodelist(), poslist(), poslist2D(), actvalues() :`
  Standard constructor

- `MBSSensor(MBS* mbsi, TMBSSensor typeI, int element, int node): elementlist(), nodelist(), poslist(), poslist2D(), actvalues():`
  Constructor for sensor of `typeI`, with one element or one node; `node` can also represent some component (e.g. the specific type of DOF in a `TSDOF` sensor)

- `MBSSensor(MBS* mbsi, TMBSSensor typeI, int element, Vector3D& pos): elementlist(), nodelist(), poslist(), poslist2D(), actvalues():`
  Constructor for sensor of `typeI`, with one element and one position

- `MBSSensor(MBS* mbsi, TMBSSensor typeI, int element, Vector3D& pos1, Vector3D& pos2): elementlist(), nodelist(), poslist(), poslist2D(), actvalues():`
  Constructor for sensor of `typeI`, with one element and two positions

- `MBSSensor(MBS* mbsi, TMBSSensor typeI, TArray<int>& elements, TArray<int>& nodes): elementlist(), nodelist(), poslist(), poslist2D(), actvalues():`
  Constructor for sensor of `typeI`, with list of elements and list of nodes

- `MBSSensor(MBS* mbsi, TMBSSensor typeI, TArray<int>& elements, TArray<Vector3D>& positions): elementlist(), nodelist(), poslist(), poslist2D(), actvalues():`
  Constructor for sensor of `typeI`, with list of elements and list of positions

- `MBSSensor(MBS* mbsi, TMBSSensor typeI, int element, Vector2D& pos): elementlist(), nodelist(), poslist(), poslist2D(), actvalues():`
  Constructor for sensor of `typeI`, with one element and one 2D position

- `MBSSensor(MBS* mbsi, TMBSSensor typeI, TArray<int>& elements, TArray<Vector2D>& positions): elementlist(), nodelist(), poslist(), poslist2D(), actvalues():`
  Constructor for sensor of `typeI`, with list of elements and list of 2D positions

- `MBSSensor(const MBSSensor& s):`
  Copy constructor

- `MBSSensor& operator=(const MBSSensor& s) :`
  Operator =

- `virtual MBSSensor* GetCopy():`
  Return a pointer to a copy of `*this`

- `virtual void CopyFrom(const MBSSensor& s):`
  Copy all data from sensor `s`

- `virtual ~MBSSensor():`
  Destructor

- `virtual void Init():`
  Initialize Sensor

- `virtual int CheckConsistency(mystr& errorstr):`
  Check consistency of sensor (e.g. if elements and nodes exist, etc.)

- `virtual void GetElementData(ElementDataContainer& edc):`
  Write element data of sensor into `edc`

- `virtual int SetElementData(const ElementDataContainer& edc):`
  Read element data of sensor from `edc`

- `virtual mystr GetTypeName() const:`
  Return type name of sensor

- `virtual void SetSensorName(const mystr& name) :`
  Assign a name to the sensor

- `virtual mystr GetSensorName() const:`
  Return the name of the sensor

- `virtual void SetSensorNumber(int num):`
  Set the internal number of the sensor

- `virtual int GetSensorNumber() const :`
  Return the internal number of the sensor

- `virtual int NElements() const`:
  Return the number of elements attached to the sensor
- `virtual int GetElNum(int i) const`:
  Return the element number of element `i`
- `virtual int& GetElNum(int i)`:
  Return a reference to the element number of element `i`
- `virtual void SetDrawingDim const`:
  Set the drawing dimensions of the sensor
- `virtual const Vector3D& GetDrawingDim(Vector3D size_detail_dummy)`:
  Return the drawing dimensions of the sensor
- `virtual void SetPrecision(int prec) :`
  Set the output precision of the sensor in digits
- `virtual int GetPrecision() const`:
  Return the output precision of the sensor in digits
- `virtual void SetFactor(double fact)`:
  Set a factor which is applied to the result of the sensor
- `virtual double GetFactor() const`:
  Return the factor which is applied to the result of the sensor
- `virtual void SetOffset(double off)`:
  Set an offset which is applied to the result of the sensor
- `virtual double GetOffset() const`:
  Return the offset which is applied to the result of the sensor
- `virtual void SetVisible(int vis)`:
  Set this flag to 1 in case the sensor should be visible; otherwise 0
- `virtual int GetVisible() const`:
  Returns the actual status of the visible flag
- `virtual void SetWriteResults(int writeresultsI = 1)`:
  Set this flag to 1 if the output of the sensor should be written:
  - `writeresults = 0`: no results written
  - `writeresults = 1`: results are written into the general solution file
  - `writeresults = 2`: results are written the specific solution file of the sensor
  - `writeresults = 3`: results are written into both files
- `virtual int GetWriteResults() const`:
  Return the actual status of the writeresults flag
- `virtual void SetOutputFile(ofstream* file)`:
  Assign the output stream which represents the specific output file of the sensor

- `virtual ofstream* GetOutputFile()`:
  Return output file stream

- `virtual const MBS* GetMBS() const`:
  Return const pointer to MBS system

- `virtual MBS* GetMBS()`:
  Return pointer to MBS system

- `virtual double GetValue() const`:
  Return the value, actually measured by the sensor

- `virtual void Draw()`:
  Draw the actual configuration of the sensor

2.3.1.2 Data in class MBSSensor

- `TMBSSensor type`:
  Type of the sensor

- `mystr sensorname`:
  String for the sensor

- `TArray<int> elementlist`:
  List of elements that correspond to the sensor

- `TArray<int> nodelist`:
  List of nodes that correspond to the sensor

- `TArray<Vector3D> poslist`:
  List of positions according to the elements of the sensor

- `TArray<Vector2D> poslist2D`:
  List of 2D positions according to the elements of the sensor

- `int precision`:
  Output precision of the sensor values

- `ofstream* fileout; //only set if writeresults==2 or 3`:
  Output file stream of the sensor

- `int sensornumber`:
  Number of the sensor in the multibody system

- `double factor`:
  Factor which is applied to the output values of the sensor; output_value = value*factor+offset

- `double offset`:
  Offset which is applied to the output values of the sensor; output_value = value*factor+offset

- `int writeresults`:
  Flag, which defines if output is written, see `SetWriteResults()`

- `int visible`:
  Flag for visibility of sensor, see `SetVisible()`

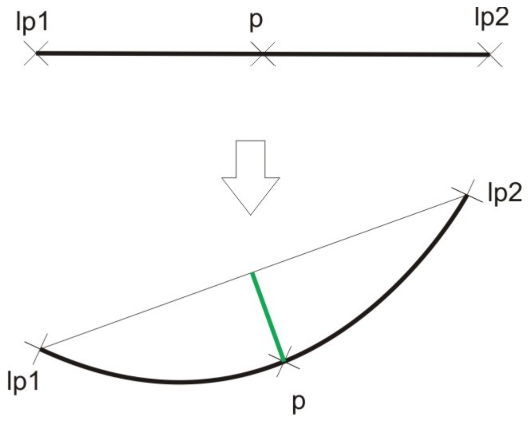


Figure 2.5: Deflection Sensor

- `Vector3D draw_dim`:
  Drawing dimensions of the sensor: `draw_dim.X()` = radius of sphere of sensor, `draw_dim.Y()` = resolution (tiling) of sphere

- `TArray<double> actvalues`:
  Temporary storage of actual values of sensor

2.3.2 Deflection Sensor

The deflection sensor measures the distance of a point p to a line. The line is defined by two points lp1 and lp2.
With this sensor it is possible to measure the deflection of a beam for example.
Additionally it is necessary to provide a vector n perpendicular to the plane (p,lp1,lp2) to define the sign of the output. The vector n is just used for this aspect and does not influence the absolute value of the output.

2.3.2.1 tested call for CMS-Element:

```
MBSSensor s_defl(mbs, (TMBSSensor)(TSElement+ TSNode+ TSDeflection), elemlist,...   ...
nodelist, poslist);
```

elemlist: 3 times the same (int) element number of the cms-element
nodelist: **global** node numbers of lp1, lp2, p
poslist: vector to define the sign of the output

2.3.2.2 tested calls for HexahedralGeneric:

```
MBSSensor s_defl(mbs, (TMBSSensor)(TSElement+TSDeflection), elemlist, poslist);
```

elemlist: element numbers of lp1, lp2, p
poslist: 4 entries:
- coordinates of the 3 points
- vector to define the sign of the output

```
MBSSensor s_defl(mbs, (TMBSSensor)(TSElement+ TSNode+ TSDeflection), elemlist,...   nodelist,
poslist);
```

elemlist: element numbers of lp1, lp2, p
nodelist **local** node numbers of lp1, lp2, p

poslist: vector to define the sign of the output

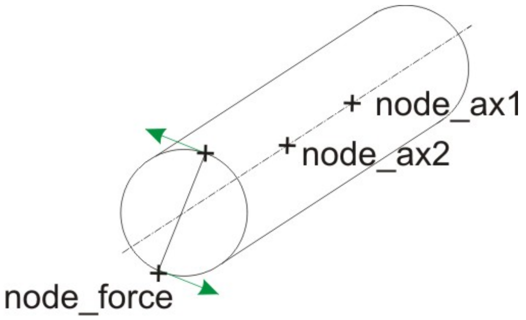


Figure 2.6: 2 rotating forces to apply momentum.

2.4 Loads

2.4.1 Loads

Loads are used to add forces at the right hand side of the second order differential equations that describe the dynamics of a body. 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). The loads can have a time-dependency which is evaluated in every step of the com-putation. Loads can only be applied to bodies that provide according information of the work of external linear, angular or integrated loads.

2.4.2 Add momentum to finite element

If you want to apply a momentum to a body modelled with finite elements, in general it is not possible to apply the momentum to the finite element directly. In this section some possibilities are presented, how to apply a moment to FE.

2.4.2.1 Attach small rigid body

If you attach a (small) rigid body to the body modelled with finite elements, then you can apply the momentum to the rigid body. The rigid body will then drive the finite elements.

- generate FEMesh and add points + elements to mbs
- generate Rigid3D and add to mbs
- constrain FE to Rigid3D, with SphericalJoint
- use `MBSLoad::SetMomentVector3D` to apply momentum to Rigid3D

2.4.2.2 SetRotatingForce

- use `MBSLoad::SetRotatingForce` to apply rotating forces directly to FE
- (at least) 2 rotating (nodal) forces are applied to apply a resulting momentum
- 2 nodes are necessary to define the axis
- up to now just tested for CMS and not for FE!

2.5 Elements

2.5.1 Base class Element: element.h

Elements are separated into connections (constraints, including springs), 2D bodies and 3D bodies. Bodies (`Body3D`, `Body2D`) are then devided into rigid bodies and flexible bodies.

In order to distinguish member functions which use computation coordinates (`GetXact`) and drawing coordinates (`GetDrawValue`), it is assumed in general that all functions are implemented for computation coordinates, except for functions which have explicitly a `Draw` in the name or a `D` at the end of the name. Furthermore, it is assumed in general that functions for position, velocity, etc. are implemented for the spatial (3D) case. If not, these functions have the specifier `2D` at the end of the name. As example:

- `virtual Vector3D GetRefPos() const`: Get reference position for **3D** case, computation coordinates

- `virtual Vector2D GetRefPos2D() const`: Get reference position for **2D** case, computation coordinates

- `virtual Vector3D GetRefPosD() const`: Get reference position for **3D** case, **drawing** coordinates

- `virtual Vector2D GetRefPos2DD() const`: Get reference position for **2D** case, **drawing** coordinates

2.5.1.1 General class members for access and modification in class Element; functions with * can/should be changed in a derived element (many of these functions can take default values)

- `Element()`:
  Empty constructor

- `Element(MBS* mbsi)`: *
  Empty constructor, with initialization of MBS pointer

- `Element(const Element& e)`:
  Copy constructor

- `Element& operator=(const Element& e)`:
  Copy operator

- `virtual Element* GetCopy()`:
  Standard function which generates a copy of the element

- `virtual ~Element()`:
  Destructor: all data which has been allocated with new within the element needs to be deleted (except GetCopy)

- `virtual void CopyFrom(const Element& e)`: *
  Copy data from the element `e`: only the function `CopyFrom` needs to be implemented in elements derived from `class Element`, in order to provide the copy function for any element. All other functions (Copy constructor, Copy operator, and `GetCopy`) use `CopyFrom`!

- `virtual void ElementDefaultConstructorInitialization()`:
  Assigns default values to the element variables. This function is called by the constructors.

- `virtual void Initialize()`: *
  This function is called each time before the computation is started with new initial conditions. Use this function e.g. in order to precompute some vectors or matrices. The initial conditions are already known at this stage.

- `virtual const MBS* GetMBS() const`:
  Get the const pointer to the MBS object

- `virtual MBS* GetMBS()`:
  Get the pointer to the MBS object

- `virtual UserOutputInterface& UO()`:
  This is the access to the Windows interface for text output in the console window. `UO()` can be used as stream operator in order to make outputs, e.g.: `UO() << "Number of elements = " << NE() << "\n";`

- `virtual int MaxIndex() const`:
  This function returns the maximum index as given in MBS

- `virtual double GetConstraintDrift(double t) const`:
  Returns the maximum absolute constraint drift of all constraints connected to this element

- `virtual int GetOwnNum() const`:
  Returns the element number as stored in MBS

- `virtual void SetOwnNum(int i)`:
  Set the element number according to MBS (automatically done when adding an element to MBS)

- `virtual void SetElementName(const char* name)`:
  Set an individual name for the element

- `virtual const mystr& GetElementName() const`:
  Get the const element name string

- `virtual mystr& GetElementName()`:
  Get the non-const element name string

- `virtual const char* GetElementSpec() const`: *
  Get the element specification as a string (Body, Constraint, etc.)

- `virtual void SetType(TMBSElement te)`:
  Set a specific type for the element (`TElement`, `TConstraint`, `TBody`, `TFFRF`, `TCMS`, `TCMSflag`, `TController`)

- `virtual int IsType(TMBSElement te) const`:
  Check if element is of type 'te'

- `virtual TMBSElement GetType() const`:
  Get the element type

- `virtual void AddType(TMBSElement te)`:
  Add a type flag 'te' to the element (with boolean OR)

- `virtual int IsRigid() const`:
  Check if element is a rigid body

- `virtual int IsFiniteElement() const`:
  Check if element is a finite element

- `virtual int PerformNodeCheck() const`:
  Check whether all node numbers of the element are consistent

- `virtual int CheckConsistency(mystr& errorstr)`: *
  Check for general consistency of the element (node numbers, constraint numbers, etc.)

- `virtual void GetElementData(ElementDataContainer& edc)`:
  Communication function for the Windows Interface (retrieves all element data)

- `virtual int SetElementData(const ElementDataContainer& edc)`:
  Communication function for the Windows Interface (provides all element data)

- `virtual void GetElementDataAuto(ElementDataContainer& edc)`:
  Communication function for the Windows Interface (retrieves all element data); function is generated automatically by the EDC-converter (in ElementEDCauto.cpp)

- `virtual int SetElementDataAuto(const ElementDataContainer& edc)`:
  Communication function for the Windows Interface (provides all element data); function is generated automatically by the EDC-converter (in ElementEDCauto.cpp)

- `virtual int ReadSingleElementData(ReadWriteElementDataVariableType& RWdata)`:
  Retrieve value of element variable/function named RWdata.variable_name which specified components

- `virtual int WriteSingleElementData(const ReadWriteElementDataVariableType& RWdata)`:
  Write value of element variable/function named RWdata.variable_name which specified components

- `virtual int GetAvailableSpecialValues(TArrayDynamic<ReadWriteElementDataVariableType>& available_variables)`:
  Adds the available special values that can be retrieved from this element to the array available_variables

- `virtual int ReadSingleElementDataAuto(ReadWriteElementDataVariableType& RWdata)`:
  Autogenerated version of `ReadSingleElementData` (implemented in ElementEDCauto.cpp)

- `virtual int WriteSingleElementDataAuto(const ReadWriteElementDataVariableType& RWdata)`:
  Autogenerated version of `WriteSingleElementData` (implemented in ElementEDCauto.cpp)

- `virtual int GetAvailableSpecialValuesAuto(TArrayDynamic<ReadWriteElementDataVariableType>& available_variables)`:
  Autogenerated version of `GetAvailableSpecialValue` (implemented in ElementEDCauto.cpp)

- `virtual void GetAvailableFieldVariables(TArray<FieldVariableDescriptor>& variables)`:
  Adds to the list `variables` all field variables that can be provided by the element (e.g. position, velocity, ...)

- `virtual double GetFieldVariableValue(const FieldVariableDescriptor& fvd, const Vector3D& local_position, bool flagD)`:
  Returns the value of the field variable specified in `fvd` at the elements local position `local_position`. If the flag `flagD` is true, then the value is used for drawing. The list of all available elements can be retrieved by `GetAvailableFieldVariables`.

- `virtual double GetFieldVariableValue(const FieldVariableDescriptor& fvd, const Vector2D& local_position, bool flagD)`:
  See 3D version

- `virtual double GetFieldVariableValue(const FieldVariableDescriptor& fvd, const double& local_position, bool flagD)`:
  1D version, see the 3D version

- `virtual double GetFieldVariableValue(const FieldVariableDescriptor& fvd, int local_node_numb`
  `bool flagD)`:
  Returns the value of the field variable specified in `fvd` at position of the node with local number `local_node_number`. If the flag `flagD` is true, then the value is used for drawing.

- `virtual int SOS() const`: *
  Second Order Size provided by the element. This is the size of mass and stiffness matrices of the element, or the size of the residual vector

- `virtual int SOSowned() const`: *
  Second Order Size of new equations and variables provided by the element which are owned (caused) by one element (not borrowed e.g. from another element or node)

- `virtual int ES() const`: *
  Explicit (=first order) Size of the equations provided by the element. This is equal to the number of first order variables of the element

- `virtual int IS() const`: *
  Implicit (=algebraic) Size of the equations provided by the element. This is equal to the number of algebraic variables of the element

- `virtual int SS() const`:
  System Size of the element = 2`SOS`+`ES`+`IS`

- `virtual int DataS() const`:
  number of data variables (e.g. plastic strains), for which there are no separate equations

- `virtual int IS_RS() const`:
  Basically this is the implicit size, however, the components may be resorted (for band width optimization)

- `virtual int SOSowned_RS() const`:
  Basically this is `SOSowned`, however, some components may not be resorted (for band width optimization)

- `virtual int FlexDOF() const`:
  The number of flexible degrees of freedom (for finite elements or CMS elements)

- `virtual void StartTimeStep()`: *
  This function is called at the beginning of the time step (e.g. in order to set some variables, read input, etc.)

- `virtual void EndTimeStep()`: *
  This function is called at the very end of the time step (e.g. in order to set something for the next time step)

- `virtual int Dim() const`: *
  Set to 2 for planar elements, and 3 for spatial elements

- `virtual int IsRigid() const`: *
  Equals one if the body is rigid, otherwise zero

- `virtual const double& GetXact(int i) const`:
  Returns the const global coordinate with index `i` of MBS

- `virtual double& GetXact(int i)`:
  Returns the non-const global coordinate with index `i` of MBS

- `virtual void SetInitConditions(const Vector& x0)`:
  Sets the vector of initial conditions of the element. `x0` must coincide with the length of the element's initial conditions

- `virtual const Vector& GetXInit() const`:
  Returns the vector of initial conditions

- `virtual const double& GetXInit(int i) const`:
  Returns the i^{th} initial condition

- `virtual void SetDataInit(const Vector& data_initI)`:
  This function sets the initial data variables

- `virtual const Vector& GetDataInit() const`:
  Returns the initial data of the element

- `virtual double GetOutput(double t, int i=1) const`:
  Returns the output of an element (e.g. of an IOElement)

- `virtual void Update()`:
  Compute things which should be actualized after every change of the MBS state vector `xact`, not used so far

- `virtual int ElementBandwidth() const`:
  Return the element bandwidth (the standard value is equal to `SS`)

- `virtual int LTG(int iloc) const`:
  Returns the $\underline{\mathrm{l}}$ocal-$\underline{\mathrm{t}}$o-$\underline{\mathrm{g}}$lobal coordinate index mapping. The local index `i` of the element coordinate i has the global index `LTG(i)`

- `virtual int& LTG(int iloc)`:
  Returns a non-const reference to local-to-global index mapping (`LTG`)

- `virtual int LTGdata(int iloc) const` Returns the local to global index mapping of the data variables. The local index `i` of the element data variable i has the global index `LTG(i)`

- `virtual int& LTGdata(int iloc)` Returns a reference to the local to global index mapping of the data variables. The local index `i` of the element data variable i has the global index `LTG(i)`

- `virtual const TArray<int>& GetLTGArray() const`:
  Returns the whole array to the local-to-global (`LTG`) index mapping of the element, see `LTG`

- `virtual const TArray<int>& GetLTGdataArray() const` Returns the whole array to the local-to-global (`LTGdata`) index mapping of the elements data, see `LTG`

- `virtual void AddLTG(int gi)`:
  Add an additional local-to-global (`LTG`) index to the element

- `virtual void AddLTGdata(int gi)`:
  Add an additional local-to-global (`LTGdata`) data index to the element

- `virtual int LTGlength() const`:
  Returns the length of the array of local-to-global (`LTG`) index mappings

- `virtual void LTGreset()`:
  Resets the array of local-to-global (`LTG`) index mappings

- `virtual void LTGdataReset()`:
  Resets the array of local-to-global (`LTGdata`) data index mappings

- `virtual void LinkLoads()`:
  Assign element numbers to loads

- `virtual void LinkToElements()`:
  For special finite elements, the local-to-global (`LTG`) index mappings can be done by hand; usually done automatically by MBS

- `virtual void BuildDependencies()`:
  Set dependencies; This defines the coordinates which influence the system matrices and the residual vector of the element; usually done automatically!

- `virtual int IsDependent(int i) const`:
  Returns 1, if the element is depening on the global DOF `i`, otherwise zero

- `virtual const double XG_dc(int iloc, TComputeDrawInitFlag flag) const`:
  Const access to the system coordinate of actual state; `flag` can be `TCD_compute` to get the actual computation value (same as `XG(iloc)`), `TCD_draw` for the drawing value of the variable (same as `XGD(iloc)`), `TCD_reference_configuration` for the value of the coordinate in the reference configuration and `TCD_initial_values` for the coordinates initial value (same as `GetXInit(iloc)`)

- `virtual const double& XG(int iloc) const`:
  Const access to the actual value of the local coordinate number `iloc`

- `virtual double& XG(int iloc)`:
  Non-const access to the actual value of the local coordinate number `iloc`
- `virtual const double& XGP(int iloc) const`:
  Const access to the actual value of the time derivative of the local coordinate number `iloc`
- `virtual double& XGP(int iloc)`:
  Non-const access to the actual value of the time derivative of the local coordinate number `iloc`
- `virtual double XGPP(int iloc) const`:
  Returns the second time derivative of the local coordinate number `iloc`
- `virtual const double& XGG(int iglob) const`:
  Const access to the actual value of the time derivative of the global coordinate number `iglob`
- `virtual double& XGG(int iglob)`:
  Non-const access to the actual value of the time derivative of the global coordinate number `iglob`
- `virtual const double& XData(int iloc) const`:
  Const access to the actual value of the data variable with local index `iloc`
- `virtual double& XData(int iloc)`:
  Non-const access to the actual value of the data variable with local index `iloc`
- `virtual const double& XGD(int iloc) const`:
  Const access to the drawing value of the local coordinate number `iloc`
- `virtual double& XGD(int iloc)`:
  Non-const access to the drawing value of the local coordinate number `iloc`
- `virtual const double& XDataD(int iloc) const`:
  Const access to the drawing value of the data variable with local index `iloc`
- `virtual double& XDataD(int iloc)`:
  Non-const access to the drawing values value of the data variable with local index `iloc`
- `virtual const double& XGPD(int iloc) const`:
  Const access to the drawing value of the time derivative of the local coordinate number `iloc`
- `virtual const double& GetDrawValue(int iloc) const`:
  Const access to the drawing value of the GLOBAL coordinate number `iloc`
- `virtual TKinematicsAccessFunctions GetKinematicsAccessFunctions(int mode = 1) const`:
  Returns the kinematic functions (e.g. position, velocity, rotation matrix) that are provided by the element.
- `virtual void PreAssemble()`:
  This function can be implemented by elements, which wish to perform some actions just before the system is assembled
- `virtual const int& GetMaterialNum() const`:
  Const access to the number of the material used by the element

- `virtual int& GetMaterialNum()`:
  Non-const access to the number of the material used by the element

- `virtual void SetMaterialNum(int mnum)`:
  Set the material for the element to the material with number `mnum`

- `virtual void AddMaterial(const Material& m)`:
  Adds the material to the (global) list of materials and makes it the material used by the element

- `virtual const Material& GetMaterial() const`:
  Const access to the material used by the element

- `virtual Material& GetMaterial()`:
  Non-const access to the material used by the element

2.5.1.2 Main functions for computation of the element:

- `virtual void PrecomputeEvalFunctions()`:
  Precomputations for `EvalF`, `EvalF2`, `EvalM`, and also for `EvalG` and `AddElementCqTLambda` in case of constraints

- `virtual void EvalF(Vector& f, double t)`: *
  This function is needed for the modeling of first-order differential equations on element level. The vector `f` in function `EvalF` returns the right-hand-side of the differential equation

  $$\dot{\mathbf{x}} = \mathbf{f}(\mathtt{xact}, t) \tag{2.3}$$

  The actual coordinates are stored in the vector `xact`. The size of the vector **f** corresponds to the value given in the function `ES()`, the components of **f** correspond to local indices. The local values of `xact` can be accessed with `XG()`, `XGP()`, etc.

- `virtual void EvalG(Vector& f, double t)`: *
  This function is needed for the modeling of algebraic equations on element level. The vector `f` in function `EvalG` returns the left-hand-side of the differential equation

  $$\mathbf{g}(\mathtt{xact}, t) = 0 \tag{2.4}$$

  The actual coordinates are stored in the vector `xact`. The size of the vector **g** corresponds to the value given in the function `IS()`, the components of **g** correspond to local indices. The local values of `xact` can be accessed with `XG()`, `XGP()`, etc.

- `virtual void EvalM(Matrix& m, double t)`: *
  This function is needed for the modeling of second order differential equations on element level. The matrix `m` in function `EvalM` returns the matrix **M** of the differential equation

  $$\mathbf{M}\ddot{\mathbf{u}} = \mathbf{f}_2(\mathtt{xact}, t) \tag{2.5}$$

  The actual coordinates are stored in the vector `xact`. The size of the matrix **M** corresponds to `SOS()`×`SOS()`, the components of **M** correspond to local (element) indices. The local values of `xact` can be accessed with `XG()`, `XGP()`, etc.

- `virtual void EvalF2(Vector& f, double t):` *
  This function is needed for the modeling of second order differential equations on element level. The vector `f` in function `EvalF2` returns the right-hand-side of the differential equation

  $$\mathbf{M}\ddot{\mathbf{u}} = \mathbf{f}_2(\texttt{xact}, t) \tag{2.6}$$

  The actual coordinates are stored in the vector `xact`. The size of the vector $\mathbf{f}_2$ corresponds to `SOS()`, the components of $\mathbf{f}_2$ correspond to local (element) indices. The local values of `xact` can be accessed with `XG()`, `XGP()`, etc.

- `virtual void EvalMinvF2(Vector& f, double t):`

- `virtual void JacobianF2(double t, Matrix& m, IVector& colref):` *
  This function provides the element jacobian for second order differential equations,

  $$\mathbf{J}_{F2} = \frac{\partial \mathbf{f}_2}{\mathbf{x}_{loc}} \tag{2.7}$$

  where $\mathbf{x}_{loc}$ represent the local (element) coordinates. In classical finite elements, the Jacobian $\mathbf{J}_{F2}$ represents the tangential element stiffness matrix.

- `virtual void JacobianG(double t, Matrix& m, IVector& colref):` *
  This function provides the element jacobian for algebraic equations,

  $$\mathbf{J}_{G} = \frac{\partial \mathbf{g}}{\mathbf{x}_{loc}} \tag{2.8}$$

  where $\mathbf{x}_{loc}$ represent the local (element) coordinates. In classical constraint formulation, the Jacobian $\mathbf{J}_G$ represents the constrain jacobian, usually denoted by $d\mathbf{C}/d\mathbf{q}$.

- `virtual double PostNewtonStep(double t):` *
  This function is suited for nonlinear (discontinuous) iterations. This function is called after each computation of a time step. This function checks for events such as switch of contact behavior or friction, elastic-plastic material behavior, switching of springs, etc. The usage of this function instead of a switching directly within the nonlinear equations of the element greatly improves the Newton convergence of the problem.
  The function returns an error estimate. If the error estimate is smaller than `SolverSettings.discontinu`
  the step is accepted

- `virtual void PostprocessingStep():` *
  After some nonlinear (discontinuous) iterations,

- `virtual double GetError():`
  This function shall return a certain characteristic norm of the state of the element which is used for the estimation of the error in the time integration. The default value is the sum of squares of position and velocity degrees of freedom.

- `virtual void SetGlobalInitConditions(Vector& x_glob):`
  This function is called such that every element writes its initial condition into the global vector of initial conditions. This is done automatically.

- `virtual void SetGlobalInitData(Vector& data_glob):`
  This function writes the it's initial data into the global vector `data_glob`

- `virtual double GetKineticEnergy():`
  Compute the actual kinetic energy of the element; only for special applications, otherwise not needed

- `virtual double GetPotentialEnergy():`
  Compute the actual potential (stored) energy of the element; only for special applications, otherwise not needed

- `virtual const double& Rho() const:`
  Const access to the density of the element

- `virtual double& Rho():`
  Non-const access to the density of the element

- `virtual const double& Em() const:`
  Const access to the elastic modulus of the element

- `virtual double& Em():`
  Non-const access to the elastic modulus of the element

- `virtual const double& Nu() const:`
  Const access to the Poisson's ratio of the element

- `virtual double& Nu():`
  Non-const access to the Poisson's ratio of the element

- `virtual double GetMass() const:`
  Return the mass of the element. Not every element may use the density (e.g. constraints and actuators). Usually, it is assumed that the mass m, densitiy ρ and volume V follow the relation $m = \rho V$.

- `virtual double GetVolume() const:`
  Return the volume of the element. Usually the volume is computed from mass and density.

- `virtual double GetMassDamping() const:`
  Return the factor for mass-proportional damping.

- `virtual void SetMassDamping(double d):`
  Set the factor for mass-proportional damping

- `virtual int FastStiffnessMatrix() const:`
  Switch flag: Equals 1 if the function `StiffnessMatrix(Matrix& m)` is implemented, otherwise zero

- `virtual void StiffnessMatrix(Matrix& m):`
  Fill in the actual tangential stiffness matrix of the element, fill in `SOS()` × `SOS()` components, the matrix `m` might be larger

- `virtual void GyroscopicMatrix(SparseMatrix& gy)const:`
  Returns the gyroscopic matrix (for rotordynamic elements)

- `virtual int TransformJacApply() const:`
  For ACF formulation, usually not needed

- `virtual void ApplyTransform(const Vector& v, Vector& Av, int mode):`
  For ACF formulation, usually not needed

- `virtual void SetUseSparseM(int i=1):`
  Switch flag: Set to 1, if the function `AddMSparse()` for the sparse computation of the mass matrix is implemented

- `virtual void SetUseSparseK(int i=1):`
  Switch flag: Set to 1, if the function `AddKSparse()` for the sparse computation the stiffness matrix is implemented

- `virtual int UseSparseM() const:`
  Switch flag: Returns 1, if the function `AddMSparse()` for the sparse computation of the mass matrix is implemented

- `virtual int UseSparseK() const:`
  Switch flag: Returns 1, if the function `AddKSparse()` for the sparse computation of the stiffness matrix is implemented

- `virtual void AddMSparse(SparseMatrix& m, double t):`
  Function which adds sparse matrix to system mass matrix, must be implemented if `UseSparseM()==1`

- `virtual void AddKSparse(SparseMatrix& m, double t):`
  Function which adds sparse tangential stiffness matrix to system stiffness matrix, must be implemented if `UseSparseK()==1`

- `virtual void AddDSparse(SparseMatrix& m, double t):`
  Function which adds sparse damping matrix to system damping matrix, should be implemented if `UseSparseMK()==1`

- `virtual void ComputationFinished():`
  Function is called when computation is finished (e.g. in order to free memory, write results, close connections, etc.)

- `virtual int GetNOutputs() const:`
  Returns the number of outputs of the element

- `virtual void SetOutputName(int output_nr, mystr& str):`
  Sets the name of the output with index `output_nr`

- `virtual void InsertConNode(int list_idx, int input_nr, Vector2D& node):`
  Inserts a connection node to the element at the position `node`; only used for IOElements

- `virtual void DeleteConNode(int list_idx):`
  Deletes a connection node; only used for IOElements

- co virtual void MoveConNode2D(int list_idx, double delta_x, double delta_y):
  Move a connection node (in the 2D Window); only used for IOElements

- co virtual void MoveElement(double delta_x, double delta_y, double delta_z):
  Move an element; only used for IOElements

2.5.1.3 Access functions for position, velocity, rotation, etc.

The following functions are used for elements of type `Body2D` or `Body3D` only.

- `virtual Matrix3D GetRotMatrix() const`:
  Returns the rotation matrix of the whole body (for rigid bodies)

- `virtual Matrix3D GetRotMatrixP() const`:
  Returns the time derivative of the rotation matrix of the whole body (for rigid bodies)

- `virtual Matrix3D GetRotMatrix(const Vector3D& ploc) const`:
  Returns the rotation matrix of the at a certain local position `ploc`

- `virtual Matrix3D GetRotMatrixD() const`:
  Returns the rotation matrix of the whole body (for rigid bodies) computed from drawing coordinates

- `virtual Matrix3D GetRotMatrixInit()`:
  Returns the rotation matrix of the whole body (for rigid bodies) at the initial state.

- `virtual Vector3D GetRefPos() const`: $*$
  Return actual position of the reference point for 3D case with computation coordinates

- `virtual Vector3D GetRefPosInit() const`: $*$
  Return initial position of the reference point for 3D case with computation coordinates

- `virtual Vector3D GetRefVel() const`: $*$
  Return actual velocity of reference point for 3D case with computation coordinates

- `virtual Vector3D GetRefConfPos(const Vector3D& p_loc) const`:
  Return initial position of the local point `p_loc`

- `virtual Vector3D GetPos(const Vector3D& ploc) const`: $*$
  Return actual position of a local point `ploc` for 3D case with computation coordinates

- `virtual Vector3D GetVel(const Vector3D& ploc) const`: $*$
  Return actual velocity of a local point `ploc` for 3D case with computation coordinates

- `virtual Vector3D GetDisplacement(const Vector3D& ploc) const`:
  Return actual displacement of the local point `ploc` from its initial position

- `virtual Vector3D GetAcceleration(const Vector3D& ploc) const`:
  Return (global) acceleration of the local point `ploc`

- `virtual Vector3D GetRefPosD() const`: $*$
  Return position of the reference point for 3D case with drawing coordinates

- `virtual Vector3D GetRefVelD() const`: $*$
  Return velocity of the reference point for 3D case with drawing coordinates

- `virtual Vector3D GetPosD(const Vector3D& ploc) const`: $*$
  Return position of a local point `ploc` for 3D case with drawing coordinates

- `virtual Vector3D GetVelD(const Vector3D& ploc) const`: $*$
  Return velocity of a local point `ploc` for 3D case with drawing coordinates

- `virtual Vector3D GetDisplacementD(const Vector3D& ploc) const`:
  Return the displacement of the local point `ploc` from its initial position with drawing coordinates

- `virtual Vector2D GetRefPos2D() const`: *
  See 3D case

- `virtual Vector2D GetRefPosInit2D() const`:
  See 3D case

- `virtual Vector2D GetRefVel2D() const`: *
  See 3D case

- `virtual Vector2D GetPos2D(const Vector2D& ploc) const`: *
  See 3D case

- `virtual Vector2D GetVel2D(const Vector2D& ploc) const`: *
  See 3D case

- `virtual Vector2D GetDisplacement2D(const Vector2D& ploc) const`: *
  See 3D case

- `virtual Vector2D GetRefPos2DD() const`: *
  See 3D case

- `virtual Vector2D GetRefVel2DD() const`: *
  See 3D case

- `virtual Vector2D GetPos2DD(const Vector2D& ploc) const`: *
  See 3D case

- `virtual Vector2D GetVel2DD(const Vector2D& ploc) const`: *
  See 3D case

- `virtual Vector2D GetDisplacement2DD(const Vector2D& ploc) const`: *
  See 3D case

- `virtual Vector3D GetAngularMomentum()` Returns the angular momentum of the body at the origin of the reference coordinates

- `virtual Vector3D GetAngularMomentum(const Vector3D& p_ref) const` Returns the angular momentum of the body at the reference position `p_ref`

- `virtual Vector3D GetDOFPosD(int idof) const`:
  Returns the drawing position of the idof$^{\text{th}}$ degree of freedom

- `virtual Vector3D GetDOFDirD(int idof) const`:
  Returns the drawing direction of action of the idof$^{\text{th}}$ degree of freedom

2.5.1.4 Access functions for nodes and nodal positions

The following functions are especially important for solid finite elements. Nodes are stored globally in the MBS system, which contain references to global coordinates (DOF) in the multibody system. The finite elements usually only store local to global node number reference lists.

- `virtual int NNodes() const`:
  Return the number of nodes of the element

- `virtual const int& NodeNum(int i) const`:
  Return const reference to global node number with local index `i`

- `virtual int& NodeNum(int i)`:
  Return non-const reference to global node number with local index `i`

- `virtual const Node& GetNode(int i) const`:
  Return const reference to Node with local index `i`

- `virtual Vector3D GetNodeLocPos(int i) const`:
  Return local nodal position in element coordinates of Node with local index `i`

- `virtual Vector3D GetNodePos(int i) const`:
  Return actual (global) nodal position of Node with local index `i`

- `virtual Vector3D GetNodePosD(int i) const`:
  Return drawing (global) nodal position of Node with local index `i`

- `virtual Vector3D GetNodeVel(int i) const`:
  Return actual (global) nodal velocity of Node with local index `i`

- `virtual Vector3D GetNodeVelD(int i) const`:
  Return drawing (global) nodal velocity of Node with local index `i`

- `virtual Vector2D GetNodeLocPos2D(int i) const`:
  Return actual (global) nodal position of 2D Node with local index `i`

- `virtual Vector2D GetNodePos2D(int i) const`:
  Return local nodal position in element coordinates of 2D Node with local index `i`

- `virtual Vector2D GetNodePos2DD(int i) const`:
  Return drawing (global) nodal position of 2D Node with local index `i`

- `virtual Vector2D GetNodeVel2D(int i) const`:
  Return actual (global) nodal velocity of 2D Node with local index `i`

2.5.1.5 Loads and constraints

- `virtual void GetH(Matrix& H)`: *
  Compute the integral over the element of the derivative of the displacement **u** or position **p** with respect to the element coordinates **q** and the element jacobian **J**:

$$\mathbf{H} = \int_{V_{elem}} \frac{\partial \mathbf{u}}{\partial \mathbf{q}} \det(\mathbf{J})\, dV_{elem} \tag{2.9}$$

For finite elements with shape matrix $\mathbf{S}$, and displacements $\mathbf{u} = \mathbf{S}\mathbf{q}$, it simply reads

$$\mathbf{H} = \int_{V_{elem}} \mathbf{S} \det(\mathbf{J})\, dV_{elem} \tag{2.10}$$

- `virtual void AddLoad(const MBSLoad& li)`:
  Add a load of type `MBSload` to element

- `virtual int NLoads() const`:
  Return the number of loads of the element

- `virtual const MBSLoad& GetLoad(int i) const`:
  Return a const reference to a specific load of the element

- `virtual MBSLoad& GetLoad(int i)`:
  Return a non-const reference to a specific load of the element

- `virtual void DeleteLoad(int i)`:
  Delete a load of the element

- `virtual int GetLoadNr(int i)`:
  Return the global node number of the local node with index `i`

- `virtual TArray<int> GetLoadNrs()`:
  Return a list of all global load numbers acting on this element

- `virtual void SetLoadNrs(TArray<int> load_nrs)`:
  Set the list of loads acting on the element

- `virtual void AddConstraint(Constraint* c, int lind)`:
  Add a constraint to the element; This function is called from the system when assembling the MBS system. Do not call this function by hand!

- `virtual void RemoveConstraints()`:
  Remove a constraint from the element. This function is used by the *Windows Interface* to remove constraints

- `virtual int NC() const`:
  Return the number of constraints

- `virtual Constraint* GetConstraint(int i) const`:
  Return a pointer to the constraint with local index i

- `virtual void ExternalSetRefCoord(double val)`:
  Assigns an external input value to an element data

- `virtual void GetDuxDq(Vector& dudq)`: *
  Return the derivative of the x-component of the displacement $\mathbf{u}$ with respect to the element coordinates $\mathbf{q}$, `GetDuxDq` $= \dfrac{\partial u_x}{\partial \mathbf{q}}$

- `virtual void GetDuyDq(Vector& dudq)`: *
  Return the derivative of the y-component of the displacement $\mathbf{u}$ with respect to the element coordinates $\mathbf{q}$, `GetDuyDq` $= \dfrac{\partial u_y}{\partial \mathbf{q}}$

- `virtual void GetDuzDq(Vector& dudq):` *
  Return the derivative of the z-component of the displacement $\mathbf{u}$ with respect to the element coordinates $\mathbf{q}$, `GetDuzDq` $= \dfrac{\partial u_z}{\partial \mathbf{q}}$

- `virtual void GetDrotxDq(Vector& dudq):`
  not used!

- `virtual void GetDrotyDq(Vector& dudq):`
  not used!

- `virtual void GetDrotzDq(Vector& dudq):`
  not used!

- `virtual void GetIntDuDq(Matrix& dudq):` *
  This is equal to `virtual void GetH(Matrix& H`!!!

- `virtual void GetIntRhoDuDq(Matrix& rhodudq):`
  This is equal to `GetIntDuDq(Matrix& dudq)` times `Rho()`

- `virtual void GetIntDkappaDq2D(Vector& dudq):`
  Returns $\int \frac{\partial \kappa}{\partial q}$ (needed for ANCFCable2D)

- `virtual void GetIntDuDqFCentrifugal(Matrix& dudq, const Vector3D& omega, const Vector3D& r0):`
  Returns the centrifugal load for flexible bodies

- `virtual void AddSurfacePressure(Vector& f, double pressure, int dir):`
  For ANCFplate elements only, to add surface pressure; Do not use

- `virtual int AddSensor(int sensornum):`
  Adds a sensor to the element

- `virtual int NSensors() const :`
  Returns the number of sensors attached to the element

- `virtual int GetSensorNum(int i) const:`
  Returns the global sensor number of the sensor attached to the element with index `i`

- `virtual void SetSensorNum(int i, int sensnum):`
  Sets the sensor with global index `sensnum` to the local index `i`

- `virtual const Sensor& GetSensor(int i) const:`
  Const access to the sensor with local index `i`

- `virtual Sensor& GetSensor(int i):`
  Non-const access to the sensor with local index `i`

- `virtual double GetSpecialSensorValue(int nr, double time) const:` *
  Deprecated function! Use ReadSingleElementData instead

- `virtual void AddElement(int en):`
  Add the element with global index `en` to the list of depending elements

- `virtual int NE() const`:
  Returns the number of depending elements

- `virtual int GetElnum(int i) const`:
  Returns the global element number of the dependent element with index `i`

- `virtual void SetElnum(int i, int elnum)`:
  Sets in the list of dependent elements the element with global element number `elemnum` to index `i`

- `virtual const Element& GetElem(int i) const`:
  Const access to the dependent element with local index `i`

- `virtual Element& GetElem(int i)`:
  Non-const access to the dependent element with local index `i`

- `virtual void GetDirectFeedThroughElements(TArray<int>& elnums) const`:
  Add all elements which are depending on this element by direct feed through the list `elnums`

- `virtual int RespondToKey(int key)`:
  Gets called if a key was pressed. Should return 1 if the element reacted to the key and 0 otherwise.

- `virtual void Increase()`:
  Increases a counter in the element (used by IOResponseElement)

- `virtual void Decrease()`:
  Decreases a counter in the element (used by IOResponseElement)

- `virtual void GetGlobalConstrainedDOFs(TArray<int>& dofs) const`:
  For constrained elements this function adds the global indices of constrained degrees of freedoms to the list `dofs`; this function is used by the eigenmodes solver

2.5.1.6 Drawing functions and geometric elements

- `virtual void DrawElementPreProc()`:
  Gets called before the drawing routines are started; gets also called if the element is not drawn (e.g. because an alternative shape is used)

- `virtual void DrawElement()`: *
  Draws the element

- `virtual void DrawElement2D()`:
  Draws the element in the 2D window (IOBlocks window)

- `virtual void DrawElementLocalFrame()` Draws the elements local frame

- `virtual void DrawElementVelocityVector()`:
  Draws a vector from the element visualizing its current velocity

- `virtual const int& DrawElementFlag() const` Const access to the flag that indicates if the element should be drawn

- `virtual int& DrawElementFlag()` Non-const access to the flag that indicates if the element should be drawn

- `virtual const Vector3D& GetSize() const`:
Return the (characteristic) size of the element. Every element has a characteristic size. This may be the size of the bounding box or for beams the length, width and height.

- `virtual void SetAltShape(int truefalse)`:
Set 0, the body is drawn according to its default shape (e.g. a cube for rigid bodies); set 1, the body is drawn with the given `GeomElements`

- `virtual int GetAltShape() const`:
Returns 0, the body is drawn according to its default shape (e.g. a cube for rigid bodies); returns 1, the body is drawn with the given `GeomElements`

- `virtual int NGeomElements()`:
Returns the number of `GeomElements`

- `virtual int AddGeomElement(int i)`:
Add (in fact link) an existing `GeomElement` to the element; if a `GeomElement` is added to an element, the local coordinates of the element are used for drawing the `GeomElement`

- `virtual const int& GetGeomElementNum(int i) const`:
Return the const reference to the global number of the `GeomElement` with local index `i`

- `virtual int& GetGeomElementNum(int i)`:
Return the non-const reference to the global number of the `GeomElement` with local index `i`

- `virtual void DeleteGeomElement(int i)`:
Delete a `GeomElement` from the element

- `virtual GeomElement* GetGeomElement(int i)`:
Return the pointer to the `GeomElement` with local index `i`

- `virtual int Add(const GeomElement& de)`:
Add a new `GeomElement` to the element. The `GeomElement` is added to the MBS system and the index of this `GeomElement` is stored in the element

- `virtual Box3D GetBoundingBox() const`:
Return the bounding box of the element (for the actual configuration), taking into account the `altshape` value. The bounding box represents the maximum and minimum x, y and z coordinates of the element, given in global coordinates of the MBS system

- `virtual Box3D GetBoundingBoxD() const`:
Return the bounding box of the element (for the drawing configuration)

- `virtual Box3D GetElementBox() const`: *
Return the bounding box of the element with default shape (for the actual configuration)

- `virtual Box3D GetElementBoxD() const`: *
Return the bounding box of the element with default shape (for the actual configuration)

- `virtual Box2D GetBoundingBox2D() const`:
See 3D function

- `virtual Box2D GetBoundingBox2DD() const`:
  See 3D function

- `virtual Box2D GetElementBox2D() const`:
  See 3D function

- `virtual Box2D GetElementBox2DD() const`:
  See 3D function

- `virtual void DrawElementAdd()`:
  Draw elements with alternative shape. Called by system, don't use!

- `virtual const Vector3D& GetCol() const`:
  Return the color of the element

- `virtual Vector3D& GetCol()`:
  Non-const access to the color of the element

- `virtual Vector3D GetSurfaceNormalD(int dir)`:
  Get normal to a specified surface in order to plot surface pressure

2.5.1.7 Class members for FFRF elements

- `virtual void GetI1(Vector& I1)`:
  Integral needed for FFRF formulation, see [8]

- `virtual double GetIkl(int k, int l)`:
  Integral needed for FFRF formulation, see [8]

- `virtual void GetIbarkl(int k, int l, Vector& I1)`:
  Integral needed for FFRF formulation, see [8]

- `virtual void GetSbar(Matrix& Sbar)`:
  Integral needed for FFRF formulation, see [8]

- `virtual void GetSbarkl(int k, int l, Matrix& Sbar)`:
  Integral needed for FFRF formulation, see [8]

- `virtual void EvalMff(Matrix& m, double t)`:
  Evaluate the part of the mass matrix for flexible DOF, needed for FFRF formulation, see [8]

- `virtual int NCMSNodes() const`:
  needed for FFRF formulation, see [8]

2.5.1.8 Data of class Element (protected)

- `mutable MBS* mbs`:
  Pointer to MBS system

- `Vector x_init`:
  Vector of element initial coordinates

- `Vector data_init`:
  Vector of element initial data variables values

- `mystr elementname`:
  Name of the element

- `TArray<int> ltg`:
  Array which stores the local-to-global mapping of coordinate indices

- `TArray<int> ltgdata`:
  Array which stores the local-to-global mapping of data vector indices

- `int elnum`:
  Element number as stored in the MBS system

- `TMBSElement type`:
  Type of the element (`TElement, TConstraint, TBody, TFFRF, TCMS, TCMSflag, TController`)

- `TArray<int> loads`:
  Array of indices of loads that are applied to the element

- `TArray<Constraint*> constraints`:
  Array of constraints, that are linked to the element (list generated by `Assemble()` procedure in MBS)

- `TArray<Constraint*> constraints_nodouble`:
  Special array of constraints, where there are no double constraint indices. This list resolves some problems which occur with contact constraints, where a contact element can constrain an element at several different positions. This list does not contain any two equal element numbers as compared to `constraints`

- `TArray<int> constraintindices`:
  Array that stores the local index of this element in the local element list of the constraint

- `TArray<char> dependencies`:
  Array of dependencies: the length of this array is equal to the system size (`SS()`) of the MBS system. If the element is depending on a specific global coordinate `i`, then `dependencies(i)==(char)1`, otherwise 0

- `TArray<int> sensors`:
  Array of indices of sensors that are attached to the element

- `TArray<int> elements`:
  Array if indices of elements depending on this element (mainly for constraints)

- `int materialnum`:
  Material number which contains the main material properties

- `double rho`:
  Stores the density of the element

- `double mass`:
  Stores the mass of the element

- `double damping_m`:
  Stores the mass proportional damping of the element

- `Vector3D col`:
  Stores the color of the element (RGB)
- `TArray<int> drawelements`:
  Array which stores the global `GeomElement` numbers for the element
- `int altshape`:
  Equal to 1, if the element is represented by `GeomElements`, otherwise 0
- `int draw_element`:
  Flag to draw element (1 - draw element, 0 - don't draw element)

2.5.2 Bodies

Bodies are specific elements which represent rigid or deformable mechanical bodies, with a certain mass and extension. Forces and constraints can be applied to bodies, and the position of a specific local point of the body is defined by means of the displacement of the body reference and the deformation or orientation of the body. A body might have no extension (point mass: `Mass3D`). Finite elements are nodal based elements, where the position and velocity of a point on the body depends on the interpolation of the (global) nodal displacement and velocity. Bodies must at least provide information about position of the center of mass and its rotation (except `Mass3D`). All informations must be available for computation time and for drawing time (additional letter "D").

2.5.3 The base class `Constraint`

2.5.3.1 Data

- `TArray<int> elements`:
  Element numbers (refer to elements stored in MBS), which are attached to the constraint

- `Vector3D draw_dim`:
  Drawing dimensions of the constraint; depends on the type of the constraint

2.5.3.2 Methods

Constraints must provide the residual of the algebraic equation (and possible its first derivatives to be applicable to an index 2 reduction), as well as the terms for the "*virtual work of constraints*" (`virtual void AddElementCqTLambda(int locelemind, Vector& f)`). Here, the element with the local element index of the constraint `locelemind` asks for the element-specific $\mathbf{C}_q^T\boldsymbol{\lambda}$ term, which will be added to the vector `f`, which is related to the local element generalized coordinates.

- `virtual void EvalG(Vector& f, double t)`: *
  This function is needed for the modeling of algebraic equations on element level. The vector `f` in function `EvalG` returns the left-hand-side of the differential equation

  $$\mathbf{g}(\mathtt{xact}, t) = 0 \tag{2.11}$$

  The actual coordinates are stored in the vector `xact`. The size of the vector $\mathbf{g}$ corresponds to the value given in the function `IS()`, the components of $\mathbf{g}$ correspond to local indices. The local values of `xact` can be accessed with `XG()`, `XGP()`, etc. The function `EvalG` usually differs for a certain maximum index of the integrator: If the integrator is capable of index 3, the constraints are formulated on the position level, while if the integrator is capable of index 2, the constraints are given on the velocity level. The function sometimes switches between an implementation of a ground joint (only one element) or a joint which connects two elements

- `virtual void JacobianG(double t, Matrix& m, IVector& colref)`: *
  This function provides the element jacobian for algebraic equations,

  $$\mathbf{J}_G = \frac{\partial \mathbf{g}}{\mathbf{x}_{loc}} \tag{2.12}$$

  where $\mathbf{x}_{loc}$ represent the local (element) coordinates. In classical constraint formulation, the Jacobian $\mathbf{J}_G$ represents the constrain jacobian, usually denoted by $d\mathbf{C}/d\mathbf{q}$.

- `virtual void AddElementCqTLambda(int locelemind, Vector& f)`: *
  This function adds the vector
  $$\mathbf{f} = -\frac{\partial \mathbf{C}}{\mathbf{q}}\boldsymbol{\lambda} \tag{2.13}$$
  to the right hand side of the differential equations. This term represents the constraint Jacobian times the Lagrange multipliers; all terms are computed on the element level.

- `virtual double PostNewtonStep(double t)`: *
  This function is suited for nonlinear (discontinuous) iterations. This function is called after each Newton method at a time / load step. This function checks for events such as switch of contact behavior or friction, elastic-plastic material behavior, switching of springs, etc. The usage of this function instead of a switching directly within the nonlinear equations of the element greatly improves the Newton convergence of the problem.
  The function returns an error estimate. If the sum (over all elements of the multibody system) of those error estimates is larger than the parameter `SolverSettings.discontinuousaccuracy`, then the time / load step is restarted.

- `virtual void FixNonlinStep()`: *
  After some nonlinear (discontinuous) iterations,

- `virtual int IsGroundJoint() const`:
  Return 1, if the element is a ground joint

- `virtual void AddElement(int en)`:
  Add an element to the constraint; usually done by the assembly procedure

- `virtual const Element& GetElem(int i) const`:
  Return the const reference to an element that is attached to the constraint

- `virtual int NE_nodouble() const`:
  Number of elements which are attached to the constraint, counting every element only once (e.g. if an element is attached at several points in the case of contact)

- `virtual const Element& GetElem_nodouble(int i) const`:
  Return the const reference to an element that is attached to the constraint, using the list with no double entries

- `virtual Element& GetElem_nodouble(int i)`:
  Return the non-const reference to an element that is attached to the constraint, using the list with no double entries

- `virtual const Body2D& GetBody2D(int i) const`:
  Return the const reference to an element which is converted into a `Body2D`

- `virtual Body2D& GetBody2D(int i)`:
  Return the non-const reference to an element which is converted into a `Body2D`

- `virtual const Body3D& GetBody3D(int i) const`:
  Return the const reference to an element which is converted into a `Body3D`

- `virtual Body3D& GetBody3D(int i)`:
  Return the non-const reference to an element which is converted into a `Body3D`

2.5.3.3 Concepts for creating a new specialized Constraint

```
#ifndef __MY_CONSTRAINT
#define __MY_CONSTRAINT

class MyConstraint : public Constraint
{
public:
  MyConstraint(MBS* mbs) : Constraint(mbs)
  {
    ElementDefaultConstructorInitialization();
  }

  MyConstraint(const MyConstraint& e) : Constraint(e.mbs)
  {
    CopyFrom(e);
  }

  virtual void ElementDefaultConstructorInitialization()
  {
    // just set some legal values
  }

  virtual int CheckConsistency(mystr& errorstr)
  {
    int rv = Constraint::CheckConsistency(errorstr);
    if (rv) return rv;

    ...  /*specify error scenarios*/

    return rv;  //rv==0 : OK, rv==1 : can't compute, rv==2 : can't draw and compute
  }

  virtual void CopyFrom(const MyConstraint& rhs)
  {
    Constraint::CopyFrom(rhs);
    /*copy own members*/
  }

  virtual Element* GetCopy()
  {
    Element* ec = new MyConstraint(*this);
    return ec;
  }

  virtual const char* GetElementSpec() const
  {
    return "MyConstraint";
  }

  virtual int IS() const  /*number of constraint equations*/
  {
    int implicit_size;
    return implicit_size;
  }

  void SetMyConstraint(/*some relevant constraint parameters*/)
  {
    // set members of this constraint
    // define, if penalty formulation shall be used
    SetPenaltyFormulation(/*0|1*/);

    //collect elements for which this constraint takes effect
    for (...)
    {
      AddElement(global_element_index);
    }
  }

  virtual void EvalG(Vector &C, double t)
  /*evaluation of C(q) for constraint equations C(q)=0, where q are generalized
    coordinates of the unconstrained multibody system, and C is vector of size IS()*/
  {
    if (UsePenaltyFormulation()) return;   // in this case only EvalF2() is relevant

    if (MaxIndex()==3)
    // determines kind of index reduction for constraints in time integration scheme
    // (maxindex=3 --> position constraints, maxindex=2 --> velocity constraints)
    {
      C = ...;
    }
    else
    {
      C = ...;
    }
  }

  virtual void AddElementCqTLambda(double t, int locelemind, Vector& f)
  /*evaluation of d/dq c(q)^T lambda (according to Lagrangian constraint formalism).
    you probably have to call dposdq or similar of each element on which this constraint acts.*/
  {
    if (UsePenaltyFormulation()) return;   /*in this case only EvalF2() is relevant*/

    f += ...;    // add dCdqTLambda  /*sign often depends on local element index (see int locelemind)*/
  }

  virtual Vector3D GetRefPosD() const
  {
    return GetBody3D(1).GetPosD(Vector3D(0.));   /*or similar*/
  }

  virtual void DrawElement()
  {
    Constraint::DrawElement();
    ...
  }

protected:
  /*members*/
};

#endif    //__MY_CONSTRAINT
```

2.5.3.4 Connecting multi-point constraints with elements

Multi point constraints, e.g. `AverageConstraint or contact!`, require an enhanceded communication with multiple elements.

- `class AverageConstraint : public BasePointJoint`

- `BasePointJoint` has `TArray<int> elements_nodouble`

- `BasePointJoint::GetElem_nodouble(int i)` returns `GetMBS()->GetElement(elements_nodouble(i))`

- `AverageConstraint::LinkToElements()` tells elements, which constraint acts on them. The cases Lagrange and Penalty are distinguished, and implemented in the routines `LinkToElementsLagrange` and `LinkToElementsPenalty`

- `AverageConstraint::LinkToElementsLagrange()` does

```
for (int i = 1; i <= NE_nodouble(); i++)
{
  GetElem_nodouble(i).AddConstraint(this, i);
}
```

- `Element` has `IVector constraints` and `IVector constraintindices`

- `Element::AddConstraints(Constraint* c, int lind)` does

```
...
constraints.Add(c);
constraintindices.Add(lind);
...
```

- `Element::EvalF2(Vector& f, double t)` does

```
...
for (int i=1; i <= constraints.Length(); i++)
{
  constraints(i)->AddElementCqTLambda(t, constraintindices(i), f);
}
...
```

2.5.4 Rigid3D

The following description provides some information on the implementation of the `Rigid3D` element, which represents a general 3D rigid body. The element is derived from `Body3D`, which includes some general functions that are necessary in all rigid or flexible spatial bodies.
Euler parameters are usually denoted with `beta`. Time derivatives usually have an additional "`p`". Some important functions for the transformation between Euler angles and Euler parameters are implemented in `linalg3D.h`.

- `Rigid3D(MBS* mbsi):Body3D(mbsi)`:
  Basic constructor

- `Rigid3D(const Rigid3D& e):Body3D(e.mbs)`:
  Copy constructor

- `Rigid3D(MBS* mbsi, const Vector& x0, const Vector& phi0, double rhoi, double Vi, const Vector3D& Ip, const Vector3D& si, const Vector3D& coli)`:
  Constructor, initialization with density, volume, moments of inertias; product moments of inertia are set zero

- `Rigid3D(MBS* mbsi, const Vector& x0, const Vector& phi0, double rhoi, double Vi, const Matrix3D& Ip, const Vector3D& si, const Vector3D& coli)`:
  Constructor, initialization with density, volume, moments and product moments of inertias

- `Rigid3D(MBS* mbsi, const Vector& x0, const Vector& phi0, double rhoi, const Vector3D& si, const Vector3D& coli)`:
  Constructor, initialization with density and size of a cubic body (`size.X()` × `size.Y()` × `size.Z()`)

- `void ComputeInitialConditions(const Vector3D& xp, const Vector3D& vp, const Vector3D& phi, const Vector3D& phip, Vector& xinit)`:
  Compute initial conditions from initial position `xp`, velocity `vp`, Euler angles `phi`, and angular velocities `phip` (everything in global coordinates)

- `virtual const Matrix3D& GetRotInertia() const return Iphi;`:
  Return the inertia tensor

- `virtual Vector3D GetAngularVel(const Vector3D& p_loc) const`:
  Return the angular velocity vector, `p_loc` is unused

- `virtual Matrix3D GetG() const`:
  Return the **G**-matrix as defined in the book of Shabana [8]

- `virtual Matrix3D GetGT() const`:
  Return the transpose of the **G**-matrix as defined in [8]

- `virtual Matrix3D GetGbar() const`:
  Return the $\bar{\mathbf{G}}$-matrix as defined in [8]

- `virtual Matrix3D GetGbarT() const`:
  Return the transpose of the $\bar{\mathbf{G}}$-matrix as defined in [8]

- `virtual Matrix3D GetGbarp() const`:
  Return the time derivative of the **G**-matrix as defined in [8]

- `virtual void AddEPCqTterms(Vector& f)`:
  Add the vector
  $$\frac{\partial C}{\partial \mathbf{q}}, \quad \text{where} \quad C(\mathbf{q}) = \sum_{i=0}^{3} \theta_i^2 - 1 = 0 \tag{2.14}$$
  which represents the derivative of the constraint $C(\mathbf{q})$ of the Euler parameters θ_i with respect to the element parameters $\mathbf{q}$

- `virtual Matrix3D GetRotMatrix() const`:
  Return the actual rotation matrix for the body

- `virtual Matrix3D GetRotMatrixP() const`:
  Return the time derivative of the actual rotation matrix for the body

- `virtual Matrix3D GetRotMatrixD() const`:
  Return the rotation matrix of the body for the drawing configuration

- `virtual Matrix3D GetRotMatrixPD() const`:
  Return time derivative of the rotation matrix of the body for the drawing configuration

- `virtual Matrix3D ComputeRotMatrix(const double& beta0, const double& beta1, const double& beta2, const double& beta3) const`:
  Compute a rotation matrix from the four Euler parameters `beta0` . . . `beta3`

- `virtual Matrix3D ComputeRotMatrixP(const double& beta0, const double& beta1, const double& beta2, const double& beta3, const double& betap0, const double& betap1, const double& betap2, const double& betap3) const`:
  Compute the time derivative of the rotation matrix from the four Euler parameters `beta0` . . . `beta3` and its time derivatives `betap0` . . . `betap3`

- `virtual void ApplyDprefdq(Vector& f, const Vector3D& x)`:
  Compute the derivative of the reference point $\mathbf{p}_{ref}$ with respect to the element coordinates $\mathbf{q}$ times a vector $\mathbf{x}$
  $$\mathbf{f} = \frac{\partial \mathbf{p}_{ref}}{\partial \mathbf{q}} \mathbf{x} \tag{2.15}$$

- `virtual void ApplyDrotrefdq(Vector& f, const Vector3D& x)`:
  Compute the derivative of the reference rotation matrix $\mathbf{A}_{ref}$ with respect to the element coordinates $\mathbf{q}$ times a vector $\mathbf{x}$
  $$\mathbf{f} = \frac{\partial \mathbf{A}_{ref}}{\mathbf{q}} \mathbf{x} \tag{2.16}$$

- `virtual void GetDuxDq(Vector& dudq)`:
  Derivative of displacement $\mathbf{u}_x$ with respect to the element coordinates $\mathbf{q}$: `dudq` $= \frac{\partial \mathbf{u}_x}{\partial \mathbf{q}}$

- `virtual void GetDuyDq(Vector& dudq)`:
  Derivative of displacement $\mathbf{u}_y$ with respect to the element coordinates $\mathbf{q}$: `dudq` $= \frac{\partial \mathbf{u}_y}{\partial \mathbf{q}}$

- `virtual void GetDuzDq(Vector& dudq)`:
  Derivative of displacement $\mathbf{u}_y$ with respect to the element coordinates $\mathbf{q}$: `dudq` $= \frac{\partial \mathbf{u}_z}{\partial \mathbf{q}}$

- `virtual void GetH3T(const Vector3D& vloc, Matrix3D& d)`:
  Compute
  $$\mathbf{d} = \mathbf{G}^T \mathrm{skew}(\mathbf{A}\mathbf{v}_{loc}) \tag{2.17}$$
  where $\mathbf{A}$ is the rotation matrix of the body, $\mathbf{v}_{loc}$ is a local vector of the body, and skew is the operator which maps a vector to a skew-symmetric matrix

- `virtual void GetdRotvdqT(const Vector3D& vloc, const Vector3D& ploc, Matrix& d)`:
  Compute the derivative of the rotated vector $\mathbf{A}\mathbf{v}$ of a local point $\mathbf{p}_{loc}$ of the body with respect to the element coordinates $\mathbf{q}$
  $$\mathbf{d} = \frac{\partial (\mathbf{A}\mathbf{v})}{\partial \mathbf{q}} \tag{2.18}$$

- `virtual void GetdPosdqT(const Vector3D& ploc, Matrix& d)`:
  Compute the derivative of the global position of a local point $\mathbf{p}_{loc}$ of the body with respect to the element coordinates $\mathbf{q}$
  $$\mathbf{d} = \frac{\partial \mathbf{p}}{\partial \mathbf{q}} \tag{2.19}$$

- `virtual void GetdAngVeldqpT(const Vector3D& ploc, Matrix& d)`:
  Compute the derivative of the angular velocity vector of a local point $\mathbf{p}_{loc}$ of the body with respect to the element coordinates $\mathbf{q}$
  $$\mathbf{d} = \frac{\partial \boldsymbol{\omega}}{\partial \mathbf{q}} \tag{2.20}$$

- `virtual void GetdPosdx(const Vector3D& ploc, Vector3D& dpdx)`:
  Compute the derivative of the global position of a local point $\mathbf{p}_{loc}$ of the body with respect to the local x-coordinate; Used for beam elements
  $$\texttt{dpdx} = \frac{\partial \mathbf{p}}{\partial x} \tag{2.21}$$

2.5.4.1 Data in class Rigid3D

- `Matrix3D Iphi`:
  Matrix containing moments and products of inertia

- `Vector betap`:
  temporary vector for time-derivative of Euler parameters

2.5.5 Geometric Elements

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 (see the Section about GeomMesh). The only influence to the computation by GeomElements is present by the automatic computation of mass, volume and inertia from the GeomElements. Usually, the complexity of GeomElements does not influence the computational time (CPU time), except for the drawing and loading/saving of multibody models. In the case of big GeomMesh models, it is recommended that the redrawing time is set to a high value, e.g. set the redrawing to every 20 seconds.

2.5.6 Finite elements

A distinguished feature of the present software is that traditional rigid multibody models can be augmented by flexible bodies by means of finite element method.

The base class for the 2D/3D finite elements is `FiniteElementGeneric`. It is a template class, which takes as an argument the `BodyXD` class of the corresponding dimensionality. The functionality of `FiniteElementGeneric` provides a common platform for all 2D/3D finite elements. Many of the virtual functions are overwritten in the derived classes. Two important points in the functionality of this class are that it manages

- the array of indices of the nodes, which belong to the element,

- the integration rules, used for the integration of various entities over the volume (or area) of the element (see section 2.5.6.3).

2.5.6.1 Finite elements in three dimensions: general description

The basic class for the three-dimensional finite elements is declared as
`class FiniteElement3D : public FiniteElementGeneric<Body3D>`.
This class has the following basic functionality:

- deals with the faces and the topology of a 3D element;
- computes entries in the stiffness and mass matrices;
- computes right-hand side in the equations of motion;
- performs post-processing functions (computes e.g. stresses);
- provides drawing functions for three-dimensional presentation of the geometry of the element and for the distribution of values for visual post-processing;

The class `FiniteElement3D` is abstract: it cannot be used by itself, as the particular element type, shape functions etc. are concretized in the derived classes.
From `FiniteElement3D` inherits the class `FiniteElement3DFFRF`, which provides interaction with other frameworks, in which the elements are unified into a Component Mode Synthesis (CMS) or Floating Frame of Reference Formulation (FFRF).
The particular finite element types are derived either from `FiniteElement3D` or from `FiniteElement3DFFRF` with the help of templates:

- `Tetrahedral`, `TetrahedralFFRF` – for 4-node and 8-node tetrahedral elements and
- `Hexahedral`, `HexahedralFFRF` – for 8-node and 20-node “brick” elements.

The latter two classes define the topology of a particular element (via the helper class `Face`), its shape functions with their necessary derivatives. The sets of Gauss points for different integration rules, which should be used for the computation of mass and stiffness matrices, are defined in the source file `IntegrationRule.cpp`, see section 2.5.6.3.
There exist different scenarios of usage of finite elements:

- The elements can be created either “by hand”, or in the framework of a finite element mesh, implemented in the class `FEMesh3D` (which is derived from `FEMesh`).
- The elements can be used
  - on their own, which is equivalent to the Absolute Nodal Coordinate Formulation (ANCF);
  - in the framework of FFRF (information, required for the computation, is provided by a helper class `FFRFData`);
  - in the framework of CMS; in this latter case the elements do not appear themselves as a part of a multibody system, but rather they are included in some other “super-element” `CMSElement`.

Elements require nodes to be present in the multibody system, which are encapsulated by the class `Node`; possible constraints of the nodal degrees of freedom can be achieved by using the class `NodalConstraint`.

2.5.6.2 ANCF finite elements

Using finite elements in the framework of Absolute Nodal Coordinate Formulation requires the following steps:

- Nodes need to be created at their initial positions and added to the storage in the multibody system by calling `MBS::AddNode()`.
- Materials need to be created and added to the multibody system.
- Elements are created with references to the indices of the created nodes. The function `void Hexahedral::SetHexahedral(int bodyindi, const TArray<int>& nodelist, int material_num, const Vector3D& coli, int CMSelementi)` should be used in the case of a hexahedral element with the following parameters:
  - `bodyind` – index of the current solid;
  - `nodelist` – indices of the nodes, returned by `MBS::AddNode()`, in the right order; 8 or 20 indices are required by a hexahedral element, 4 or 8 by a tetrahedral;
  - `material_num` – index, returned by `MBS::AddMaterial()`;
  - `coli` – color;
  - `CMSelementi` should be omitted in the case of an ANCF element.

2.5.6.3 Integration rules

All finite element classes require appropriate quadrature rules, with which various fields are integrated over the volume of the element. The currently utilized concept of centralized handling of integration rules is based on the following key points:

- Each finite element may require various integration rules with different integration orders and/or other properties (integration rules for the stiffness matrix and internal forces, for the mass matrix, for the external forces (H-matrix), etc.).
- An integration rule is defined by a list of integration points with their coordinates, weights and possibly some additional data (matrices of derivatives of the shape functions, which may be pre-computed for the integration points to speed-up the computation); see the class `IntegrationRule` in `IntegrationRule.h/cpp`.
- For the sake of the optimal memory usage, the required integration rules and the corresponding infrastructure are stored and managed by a centralized library, encapsulated in `IntegrationRulesLibrary`. An instance of the object is a member of `MBS`.
- A particular integration rule is identified by the following set of parameters, which is in the following called settings (see `IntegrationRule::IntegrationRuleSettings`):
  - Type of the finite element (see `TFiniteElementType`).
  - Interpolation order of the finite element (which is neither the order of the integration rule nor the number of points in each direction).
  - Type of the integrated entity (stiffness matrix, mass matrix, H-matrix, see `IntegratedValueType`).
  - The type of geometric nonlinearity, used in the particular simulation.

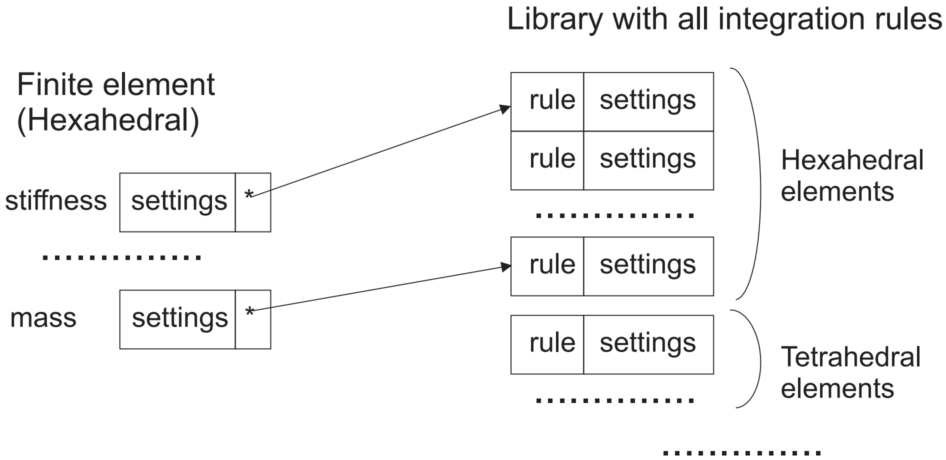


Figure 2.7: Access to the relevant integration rules for a finite element

  - Possible additional flags, which may have a meaning of the reduced integration rule, locking compensation, etc.

- "Linkage" of the finite elements with the relevant integration rules from the library (see Fig. 2.7) is done during the assembling of the model (see `FiniteElementGeneric::PreAssemble()`). The standard implementation in `FiniteElementGeneric` includes pointers to three integration rules, which are initialized in `FiniteElementGeneric::PreAssemble()` during the evaluation of `MBS::Assemble()`. The consequence is that some functions should not be called during the model creation, as e.g. `Element::DataS()` may at this stage be undefined for elements with plastic variables.

- When an element requires a particular integration rule, then a rule with a matching set of settings is sought in the library (see `IntegrationRulesLibrary::GetIntegrationRule()`). If the rule is not found, the library calls the element back, asking to create the rule. It means that the elements are responsible for the initialization of the relevant integration rules: each finite element must provide an implementation of the interface `IntegrationRule::IntegrationRuleProvider`.

- Actual usage of integration rules in the form of sequential processing in a `for`-loop is convenient with the help of iterators, see `IntegrationPointsIterator`.

- Often particular information, which is element specific, needs to be associated with each integration point. Two solutions are then possible.

  - If the data needs to be stored in `XData` (e.g. plastic strains or other variables, which vary with the time as the solution progresses), then the association needs to be performed by the indices of the points (`IntegrationPointsIterator::Index()`).
  - Gradients, inversed jacobians, etc., which are just pre-computed, can be stored in an additional array of objects of a particular type, see `FiniteElementGeneric::IntegrationPointStiffne` The mapping from integration points to the corresponding array entries is again performed with the help of the index. Several definitions of the class `IntPointsStiffnessIterator`, which are specific for particular finite element types, help accessing the local data while the integration points are processed.

2.5.6.4 Field variables: post-processing and contour plotting

Contour plotting of various field variables like stress components, displacements, etc. during post-processing is especially important for the finite elements. This is achieved by means of a unified mechanism, the central point of which is the structure `FieldVariableDescriptor`. Another benefit of this unification is that all the values, which are available for post-processing, can be attached to a sensor.
The points are the following.

- The definition of a variable, which is to be painted, is presented in the form of the structure `FieldVariableDescriptor`. The source code is documented in detail, here are the most important facts:
  - Each variable has its type. There is a list of standard types defined (`FieldVariableType`), to each type corresponds a text string (`FieldVariableDescriptor::GetTextualIdentifier()`). There exists a possibility to define a non-standard (problem specific) variable.
  - A variable may have one or two component indices; setting the first index to `FVCI_magnitude`, we obtain the norm of a vector or a tensor.
  - For all standard variable types their dimensionality can be obtained.
- Each element, which should be able to expose values of the field variables for post-processing, should reimplement the following functions:
  - `GetAvailableFieldVariables` will be called at the end of `MBS::Assemble()`. Each element, which is able to compute some field values, should provide a list of of the available variables. Adding a variable type with the whole range of component indices can be easily done with the helper function `AddTypeIntoArray`, defined in `FieldVariableDescriptor`. The set of added variables may depend on the state of the element (e.g. inelastic strains should be added only when the selected material is actually inelastic). All collected lists will be merged into a single list in `MBS`, which will then populate combo-boxes in the user interface dialog "FE options".
  - `GetFieldVariableValue` will be used to actually compute the value of a variable of a given point of the element (there are 2D and 3D versions of the function; `flagD` indicates whether the function is called from a drawing routing `DrawElement` or from a sensor). The helper function `FieldVariableDescriptor::GetComponent` simplifies the computation.

2.5.7 CMSElement + GCMSElement

In this section the CMSElement and the GCMSElement are described. Although the theory of this two approaches differs, the usage in HOTINT is very similar.
Flags that are set in HOTINT:

- TCMSflag: set if the Element is a BaseCMSElement
- TCMS: set if the Element is a finite element belonging to a BaseCMSElement (i.e. CMS *OR* GCMS)
- TGCMS: set if the Element is a finite element belonging to a GCMSElement

2.5.7.1 Theory of CMS

The position vector of a point P is given by

$$\mathbf{r}_P^{(i)} = \mathbf{r}_R^{(i)} + \mathbf{A}^{(i)}(\bar{\mathbf{x}}^{(i)} + \bar{\mathbf{u}}_F^{(i)}). \tag{2.22}$$

The degrees of freedom for the position vector are collected in the coordinate vector $\mathbf{q}^{(i)}$. In the floating frame of reference formulation (FFRF) as introduced e.g. by Shabana [8], the coordinate vector $\mathbf{q}^{(i)}$ is split into three parts, [1]

$$\mathbf{q}^{(i)} = \left[(\mathbf{q}_R^{(i)})^T \quad (\mathbf{q}_\theta^{(i)})^T \quad (\mathbf{q}_F^{(i)})^T\right]^T. \tag{2.23}$$

The last part $\mathbf{q}_F^{(i)}$ contains the flexible degrees of freedom, which are assumed to correspond to some underlying finite element discretization. The flexible part of displacement is linked to the flexible degrees of freedom $\mathbf{q}_F^{(i)}$ via the shape function matrix,

$$\bar{\mathbf{u}}_F^{(i)} = \bar{\mathbf{N}}^{(i)}\mathbf{q}_F^{(i)}. \tag{2.24}$$

In the floating frame formulation, the equations of motion can be derived from Lagrange's equations and lead to [8]

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{K}\mathbf{q} + \mathbf{C}_{\mathbf{q}}^T\lambda = \mathbf{Q}_e + \mathbf{Q}_{\mathbf{v}}. \tag{2.25}$$

In flexible multibody systems, the component mode synthesis (CMS) is a standard approach to reduce the number of flexible degrees of freedom by taking into account only the most important deformation modes.

Craig and Bampton [9] proposed a sophisticated set of modal shape functions, which included dynamic modes gained from eigen-analysis as well as static modes obeying displacement boundary conditions. A reduced set of coordinates $\mathbf{q}_F^{CMS}$ is linked to the original flexible coordinates $\mathbf{q}_F$ via the transformation matrix $\mathbf{\Phi}_{mod}$ by

$$\mathbf{q}_F = \mathbf{\Phi}_{mod}\mathbf{q}_F^{CMS}. \tag{2.26}$$

The modal transformation matrix or also modal reduction matrix $\mathbf{\Phi}_{mod}$ contains the chosen deformation modes column-wise, where the number of modes N_{CMS} is much smaller than the original number of flexible coordinates N_F. The implemenation in HOTINT uses the free-free Eigenmodes of the finite element model as deformation modes. If you want to use additional or different modes (static modes, boundary nodes, eigenmodes for a constrained structure,...) see the following section.

The equations of motion (2.25) obtained for the FFRF can be transformed to a reduced setting in a rather straightforward manner. The constant flexible sub-blocks $\mathbf{M}_{FF}$ and $\mathbf{K}_{FF}$ of mass and stiffness matrix are transformed to reduced matrices $\mathbf{M}_{FF}^{CMS}$ and $\mathbf{K}_{FF}^{CMS}$ by application of the transformation matrix $\mathbf{\Phi}_{mod}$. Depending on the chosen mode shapes, they are diagonal or fully populated.

After the reduction is done in a preprocessing step, the reduced matrices are used to compute the actual components of mass matrix and quadratic velocity vector in the FFRF-CMS setting. Note that the computational complexity is then independent of the size of the underlying finite element model.

Basic steps:

---

[1] Concerning the implementation, the order of the degrees of freedom in the vector $\mathbf{q}$ is different as stated in Eq. 2.23. The implemented order is: $\mathbf{q}_F$, $\mathbf{q}_R$, $\mathbf{q}_\theta$, which is important e.g. if you want to constrain a particular degree of freedom.

- create/load mesh
- compute/load eigenmodes
- compute $\mathbf{\Phi}_{mod}$, $\mathbf{M}_{FF}^{CMS}$, $\mathbf{K}_{FF}^{CMS}$ and reduced inertia tensors

degrees of freedom: RigidDOF + DOF for modes
e.g. for 2 modes and Bryant angles: $6 + 2 = 8$

2.5.7.2 using the CMSElement

Using the CMSElement in HOTINT requires following steps:

- create FEMesh3D
  - load nodes + elements from files or generate it by using Generate-functions
  - Transform the Mesh if necessary
  - AddMaterialsToMBS
- initialize the CMSElement
  - `CMSElement⟨FFRFRIGID⟩ cmsinit1(mbs)`
  - depending on the typdef of FFRFRIGID you can choose between Rigid3D and Rigid3DKardan
  - inital position, rotation, etc.: `cmsinit1.SetCMSElement(...`
  - damping: `cmsinit1.SetDiagonalInternalDamping(damp)`
- add loads to CMSElement, e.g.: `cmsinit1.AddLoad(loadmoment)`
- set options for eigenmode computation
  - `cmsinit1.SetSolverParameters(...`
  - `cmsinit1.SetEigenmodesFromFile(EigenmodeFile1);`
- add CMSElement to mbs: `int elnum_cms1 = mbs→AddElement(&cmsinit1);`
- set reference to CMSElement
  - `CMSElement⟨FFFRFRIGID⟩* cmsel1 = (CMSElement⟨FFRFRIGID⟩*)`
    `(&(mbs→GetElement)(elnum_cms1));`
  - necessary to access the CMSElement after adding (a copy of the element) to mesh
- add mesh to CMSElement: `cmsel1→AddFEMesh(mesh3d_1);`
  - nodes and elements of mesh are not added to mbs!
  - in mbs there is just the reference frame element
- add constraints
- do modal analysis: `cmsel1→DoModalAnalysis(fixednodes);`

- assemble the model: `mbs→Assemble();`

Some more informations concerning SetCMSElement:

```
SetGCMSElement(const Vector3D& p, const Vector3D& v, Vector3D phi, Vector3D phip,
const int nimodesi, const Vector3D& sizei, const Vector3D& coli)
```

- p: initial position of the reference point (origin), in global xyz-coordinates
- v: initial velocity of the reference point (origin), in global xyz-coordinates
- phi: initial rotation of the reference frame, reference point is origin, and parameters are zxz
- phip: initial angular velocity of the reference frame, reference point is origin, and parameters are global xyz-coordinates (=omega Vector)

If Bryant (Kardan) angles are chosen:

```
SetCMSElementKardan(const Vector3D& p, const Vector3D& v, Vector3D phi,
Vector3D phip, const int nimodesi, const Vector3D& sizei, const Vector3D& coli,
Rigid3DKardan::RotationsSequence rs)
```

- p: initial position of the reference point (origin), in global xyz-coordinates
- v: initial velocity of the reference point (origin), in global xyz-coordinates
- phi: initial rotation of the reference frame, reference point is origin, and parameters are zxz
- phip: initial angular velocity of the reference frame, reference point is origin, and parameters are global xyz-coordinates (=omega Vector)
- RotationsSequence: just used during computation, the input format of the initial conditions is not influenced by this option.

2.5.7.3 how to constrain a CMSElement

Many of the constraints in HOTINT are based on local coordinates and can therefore not be used for a CMSElement. In this section successfully tested constraints are presented:

- NodalConstraintCMS: It is recommended to use this constraint if possible. Up to now just constraints of the type element-ground are realized.
- NodalConstraint: This is the base class of NodalConstraintCMS and some of the constructors also work with CMSElement. With this constraint it is possible to implement kinematic pairs of the type element-element. Only use constructors, which are using node-numbers and not local coordinates!
- CoordConstraint: This constraint is usefull to fix the reference frame w.r.t. the global coordinate system. Be aware of the fact, that the order of the d.o.f.s is differet than stated in Eq. 2.23.

2.5.7.4 theory of GCMS

The modal reduction can also be conducted for the absolute coordinate formulation, which leads to a ACF-based CMS or *generalized component mode synthesis* (GCMS) method [10]. For a given set of local modal shape functions, modally reduced basis functions with respect to the global frame are developed. These shape functions represent rigid body translation, rotation and small co-rotated flexible deformation, and are collected in the shape function matrix $\mathbf{N}^{GCMS}(\mathbf{x})$. Corresponding degrees of freedom $\mathbf{q}^{GCMS}$ directly define the global displacement $\mathbf{u}$ via a linear relationship

$$\mathbf{u}(\mathbf{x}) = \mathbf{N}^{GCMS}(\mathbf{x})\mathbf{q}^{GCMS}. \tag{2.27}$$

As in the FFRF-based CMS, the degree of freedom vector $\mathbf{q}^{GCMS}$ consists of three parts corresponding to translation, rotation and flexible deformation

$$\mathbf{q}^{GCMS} = [(\mathbf{q}_t^{GCMS})^T \quad (\mathbf{q}_r^{GCMS})^T \quad (\mathbf{q}_f^{GCMS})^T]^T \tag{2.28}$$

with a total number of coordinates $N_{GCMS} = 12 + 9 \cdot N_M$.

In contrast to the FFRF-based CMS, in the GCMS each degree of freedom corresponds directly to one basis function, as the linear relationship (2.27) indicates.

In computations, it is necessary to evaluate the rigid body part of displacement $\mathbf{u}_r + \mathbf{u}_t$ and the flexible part $\mathbf{u}_f$ separately. Note that these parts do not correspond to the degrees of freedom $\mathbf{q}_t^{GCMS}$, $\mathbf{q}_r^{GCMS}$ and $\mathbf{q}_f^{GCMS}$ directly, since the rotation degrees of freedom $\mathbf{q}_r$ also contain stretch and shear deformations.

degrees of freedom: DOF for Modes but no seperate RigidDOF
e.g. for 2 modes and Bryant angles: $12 + 9 \cdot 2 = 30$

2.5.7.5 using the GCMSElement

In general, the usage of the GCMSElement (generalized CMSElement) in HOTINT requires the same steps as described before for the CMSElement. There is just one difference during initialization:

- create FEMesh3D
- initialize the GCMSElement
    - `GCMSElement⟨FFRFRIGID⟩ gcmsinit1(mbs)`
    - inital position, rotation, etc.: `gcmsinit1.SetGCMSElement(...`
    - set 3 reference nodes: `gcmsinit1.RefNode1() = node_nr;`
- ...

The three reference nodes define a coordinate system and should therefore be chosen according to the following rules:

- not all nodes on one line !

- RefNode1 and RefNode2 as far away from each other as possible

- line(RefNode1, RefNode2) and line(RefNode1, RefNode3) should be perpendicular

Some more informations concerning SetGCMSElement:

```
SetGCMSElement(const Vector3D& p, const Vector3D& v, Vector3D phi,
Vector3D phip, const int nimodesi, const Vector3D& sizei, const Vector3D& coli)
```

- p: initial position of origin, in global xyz-coordinates
- v: initial velocity of RefNode1, in global xyz-coordinates
- phi: initial rotation of the reference frame, reference point is RefNode1, and parameters are zxz
- phip: initial angular velocity of the reference frame, reference point is RefNode1, and parameters are global xyz-coordinates (=omega Vector)

If you prefer to define the initial conditions for a reference point P instead for RefNode1, there is following possibility. It is not necessary that P is a nodal position!

```
SetGCMSElement(const Vector3D& p, const Vector3D& v_refP, Vector3D phi,
Vector3D phip_refP, const Vector3D& ref_node1TOref_pos, const int nimodesi,
const Vector3D& sizei, const Vector3D& coli)
```

- p: initial position of origin, in global xyz-coordinates
- v: initial velocity of reference point P, in global xyz-coordinates
- phi: initial rotation of the reference frame, reference point is RefNode1, and parameters are zxz
- phip: initial angular velocity of the reference frame, reference point is reference point P, and parameters are global xyz-coordinates (=omega Vector)
- ref_node1TOref_pos: vector from RefNode1 to reference point P

2.5.8 Control elements (time continuous and discrete elements)

InputOutputElements are used for modelling of directed graphs (like controller circuits). Their inputs are Sensors or other InputOutputElements. In order to test the functionality of the system, the outputs can be watched with Sensors. In the class InputOutputElement, the fundamental code of such a graph objects are defined. The time continuous elements are derived from this class. In order to simulate time discrete behaviour, the class InputOutputElementDiscrete is derived from InputOutputElement and contains some functioniality for handling the constant time step size. The discrete elements like discrete z-Transfer Functions or discrete controllers should be derived from this class.

2.6 Import and export of data and interfaces to other software

2.6.1 FEMesh

The main purposes of the FEMesh class are

- provide routines to import meshes from external sources (NetGen, Ansys,...)
- create simple meshes from building bliocks
- manipulation of nodes and elements
- application of constraints and loads
- forward the mesh to the MBS

2.6.1.1 data content of the FEMesh

The FEMesh class contains data considering

- geometry on finite element level (Nodes, FEElements)
- geomety on macroscopic scale (bodies, areas, material sections,...)
- boundary conditions (loads, constraints)
- materials

Explanation of used terms:

Mesh: An ensemble of nodes and elements

Node: Defines the position of a vertex or intermediate point of a finite element. It is possible that two nodes occupy the same position if they differ in domain number.

Element: A finite element of the mesh, defined primarily by its Nodes. Various types of basic elements are implemented in the FEMesh class. Functionalities of the Element in the FEMesh class include returning its faces and .

Face: A side of an single finite element, stored as a list of nodenumbers. e.g. a hexahedral element has 6 faces with 4 nodes each, while a triangle element has 3 faces with 2 nodes each.

Surfaceelement: a side of an single finite element, stored as a FEElement. e.g. a hexahedral element has 6 quad elements. All outer elements of an assembly can be computed by the mesh

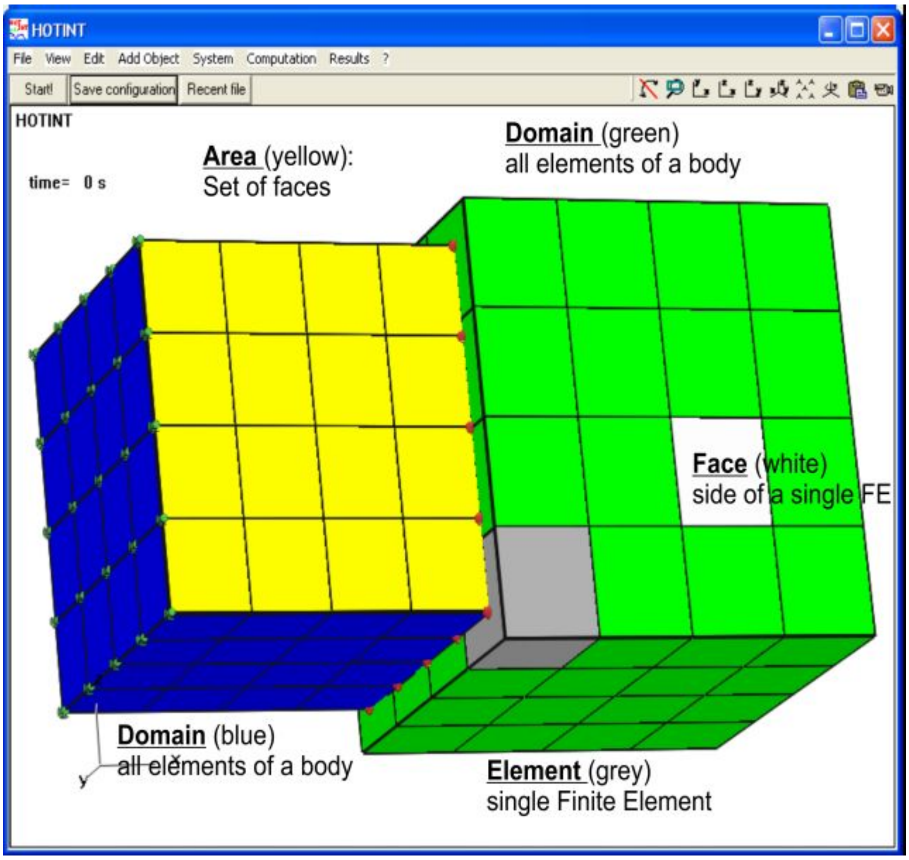


Figure 2.8: example for an FEMesh, components highlighted

Surfaceparents: list of element numbers and side numbers to map surface elements to elements

Area: Set of Faces. Areas may be defined at any time and are stored. The block generation functions of FEMesh can automatically create areas for the sides of the block or base and shell areas of a cylinder. It is also possible to compute areas where material changes (within a domain).

Domain: A single component of the assembly. The mesh of a domain is always continuous. provided at runtime

Material section: Elements with the same material - regardless of their domain. provided at runtime

Load: A Load boundary condition defined on the FEMesh. There are several different types of loads available.

Constraint: A Constratint bondary condition defined on the FEMesh. There ars several different types of constraints available.

2.6.1.2 Import functions

One major purpose of the FEMesh class is to import meshes from external programs. For some programs custom export routines were written as well.

For the import routines a file format is defined. The files themselves can be created "‘manually"’ or from export routines in the external programs.

- **text file format**

The data is divided into several blocks representing the data structure of the FEMesh class. Each data block starts with an identifier in [brackets] in the first line. The second line always contains the number of datalines that follow and additional parameters if necessary. From line 3 of each block the actual data begins, usually with a constant number of colums in each line. A single data block can not be spread over different text files but each block can be placed in different file.

Nodes `LoadNodes`
contains a list of nodes, the nodes are automatically added with domainnumber 1.
identifier: [NODES]
parameters: number of nodes
data line: node number |t x-position | y-position | z-position

Elements `LoadElements`
contains a list of elements, the number of nodes per elements must be known.
identifier: [ELEMENTS]
parameters: number of elements | number of columns reserved for nodes

example:
[NODES]
3

1	0.0	0.0	0.0
2	0.0	1.0	0.0
3	1.0	0.0	0.0

data line: element number | element dimension | element order | used nodes | node1 | ... | nodeN | material number

The routines for Nodes and Elements have very rigid format restrictions. In principle the parsing of a single dataline can be branched to use alternate formats.

Materials `LoadMaterials`
contains a list of materials, uses 2 data lines per material. This allows, in principle, to import all kinds of non-elastic materials with the same routine.
identifier: [MATERIALS]
parameters: number of materials
data line 1: number of material parameters | identifier property 1 | ... | identifier property N
data line 2: value property 1 | ... | value property N

implemented identifiers as of December 23, 2013: [density], [youngs_modulus], [poisson_ratio], [drawing_color].

Loads `LoadLoads_STA`
contains a list of loads and constraints to apply to the model, will be split into separate lists with more
identifier: [LOADS]
parameters: number of loads | identifier units
data line: identifier load type | load target | load vector

implemented identifiers as of December 23, 2013: [AreaSurfaceLoad], [AreaConstraint], [NodalConstraint], [BodyLoad], [FaceLoad], [FaceConstraint], [AreaContact]

- **Netgen**

Files saved with the **NETGEN neutral file format** can be loaded with `LoadNetgenMesh3D`.

2.6.1.3 Export functions

- **export funcitons in Hotint**

The mesh can be exported in a plain text format with the function `ExportAsText`.

- **export funcitons in Ansys**

An Ansys mesh can be exported into three separate text files with the macros ”‘NODES.mac”’, ”‘ELEMENTS.mac”’ and ”‘MATERIALS.mac”’. Each of these files contains one block of data, it would also be possible to have all blocks in only one file.

2.6.1.4 group selection in FEMesh - Filter functions and selections

The integral parts of the FEMesh are Nodes, Elements and with less significance Faces. For manipulations of the mesh filter functions are provided.

Filters considering position naturally work on the Nodes. Filters considering material or domain work on Elements. A mappping between Nodes and Elements is stored in the FEMesh class such that for a set of nodes a corresponding set of Elements can be computed quickly and vice versa.

Faces as such are list of nodes with a built-in mapping to the corresponding finite element. For performance reasons it makes sense to store the faces rather than compute them every time they are needed.

Filter functions are normally called with an array for the return values as first parameter and an array defining a subset as optional last parameter.

filter functions - geometry: These functions work with a cell-search method to exclude part of the nodes from search if possible. The Searchtree must be up to date, this can be done by calling the FEMesh function `InitializeNodeSearchTree()` ().

GetNodesInBox: all nodes inside a box, box edges parallel to axis

GetNodesOnPlane: all nodes on a plane (plane defined by plane equation ($\vec{(n)}$,c))

GetNodeAtPos: single node at given position - use subset to filter for a special domain.

GetNodesOnCircle: all nodes on a circle (circle defined by center point, symmetry axis and radius)

GetNodesMinRespVect: all nodes with a minumum distance to a plane (plane defined by plane equation ($\vec{(n)}$,c))

filter functions - element properties: These functions basically run through the elementlist and check the property.

GetElementsWithMaterial: all elements with given material number == material section

GetElementsOfBodies: all elements with given domain number == domain

filter functions - stored mappings: For more complicated filter a quick mapping between from Nodes to Elements is stored in the FEMesh class as array of arrays. The function `ComputeNodesToElements()` builds the list `nodes_to_elements` that contains for each node a list of elements this node is part of. In the opposite direction each element has a list of nodes anyway.

filter functions - composition With the basic filter operations from above, the Faces and Areas as third entities, and set operations on the arrays it is possible to create more specific filter functions by composition. For example to get all nodes of body 3 that are on outer faces and on finite elements with material type 2 the correct procedure is:

1. elements of body 3
2. nodes of these elements
3. list of outer faces
4. nodes on outer faces
5. elements with material 2
6. nodes of these elements
7. Intersection of the 3 nodes arrays

available filter functions:

- dummy

selections

2.6.1.5 Loads in FEMesh

The FEMesh class allows to place Loads. There are several types of Loads available, they are stored in an array of FEMesh_Load (or derived class) objects. This strategy allows to expand the list of implemented loads at any time.

commom properties of all loads: (data members of base class FEMesh_Load)

type: type specifier. Derived classes automatically set the type.

element: number of the item the load is applied to. Item can be Finite Element, Area, Face, ... depending on the type of the Load

loadvector: 3D Vector defining the load strength and direction.

implemented types of loads + examples: (derived classes)

Face Load: applies a pressure force to a face by placing equal loads the nodes of that face.
`FEMesh_FaceLoad( 4, Vector3D( 0., 0., 1.));`
a load of $1N/mm^2$ in positive z-direction applied to face 4

Area Load: applies a pressure force to a set of faces, each face is weighted with its actual size. `FEMesh_AreaLoad( 1, Vector3D (3., 4., 0.));`
a load of a total $5N/mm^2$ applied to area 1

Body Load: applies a force on a finite element. can be set to *gravity load* with a flag.
`FEMesh_BodyLoad( 2, Vector3D (0., -9.81, 0.), 1);`
a gravity load with gravity constant of $9.81N$ in negative y-direction applied to body 2

2.6.1.6 Constraints in FEMesh

The FEMesh class also allows to place Constraints. Again, several types of Constraints are available, they are stored in an array of FEMesh_Constraint (or derived class) objects. This strategy allows to expand the list of implemented constraints at any time.

commom properties of all constraints: (data members of base class FEMesh_Constraint)

type: type specifier. Derived classes automatically set the type.

element: number of the item the constraint is applied to. Item can be Finite Element, Area, Face, ... depending on the type of the Constraint

axis: the axis that is constrained /niy allow multiple

penalty: flag if constraint uses penalty formulation

stiffness: spring stiffness for the constraint in penalty formulation

damping: damoing of the constraint in penalty formulation

implemented types of constaints + examples: (derived classes)

Nodal Constraint: constraint for a single node
`FEMesh_NodeConstraint( 482, 1, 0, Vector3D(0.), Vector3D(0.));`
lagrangian constraint for node 482 in x-direction

Face Constraint: constrtaint for a single face
`FEMesh_FaceConstraint( 3, 3, 1, Vector3D(1e10.), Vector3D(0.));`
penalty constraint for face 3 in z-direction with 1e10 spring stiffness

Area Constraint: constraint for a single area
`FEMesh_AreaConstraint( 1, 2, 1, Vector3D(1e12.), Vector3D(0.));`
penalty constraint for area 1 in y-direction with 1e12 spring stiffness

Area Contact: constrain all nodes of one area to another area (not necessarily to nodes there) with spherical joints
additional parameter: *target area* replaces axis as 2^{nd} parameter
`FEMesh_AreaContact( 17, 4, 1, Vector3D(1e10.), Vector3D(0.));`
penalty constraints for all nodes of area 17 to corresponding positions of area 4

The FEMesh provides some routines to quickly create meshes of basic geometrical objects.

2.6.1.7 Generate Blocks of finite elements

A block of hexahedral finite elements can be created in some variations. The Block is always alligned along the cartesean axes, rotations and translations can be applied *after* creation.

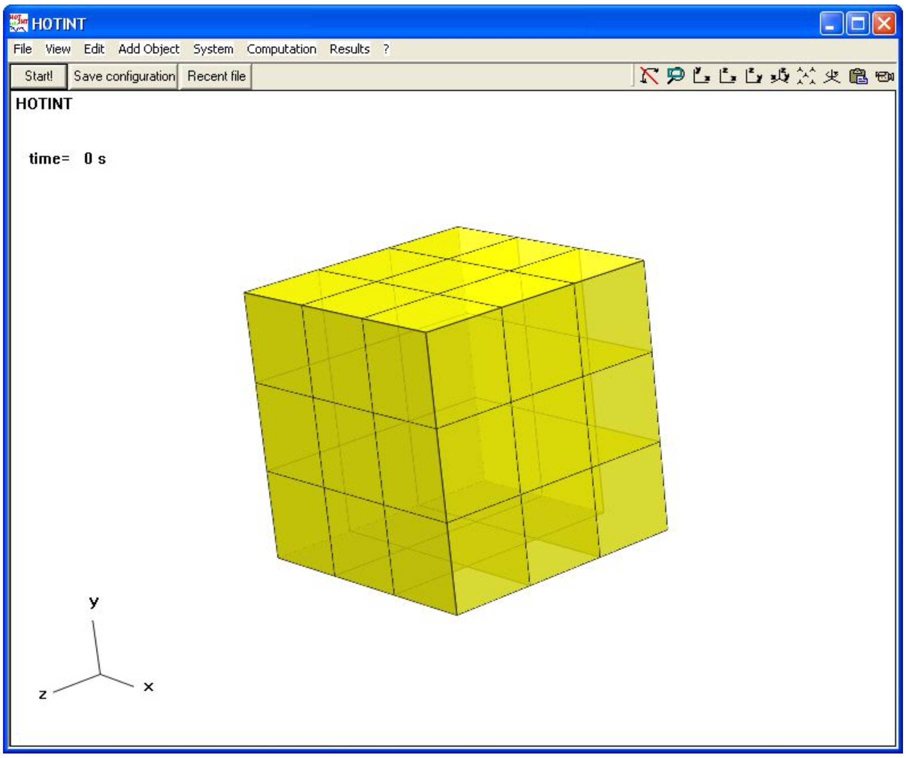


Figure 2.9: Meshed cube

The full function call is:

```
GenerateHexahedralBlock(
Box3D block, int3 divisions_xyz, int bodynr, int matnr, Vector3D color,
IVector& blockelements, IVector& blocknodes, TArray<IVector*>& blockfaces,
IVector& constraintfaces_constainttypes, IVector& constraintfaces_directions,
MBSLoad& bodyload, int bodyloadactive,
IVector& loadfaces_active, Vector& loadfaces_loadstrength,
int order)
```

The parameters of the function call are:

geometry and properties

Box3D block: size and position of the block

int3 divisions: divisions for the three main directions

int bodynr: domain number for the block (applied to all nodes created)

int matnr: material number forthe block (applied to all elements)

Vector3D color: color of the block (color of all elements)

return values

IVector& blockelements: returns numbers of all newly created elements

IVector& blocknodes: returns numbers of all nodes of the block

Array<IVector*>& blockfaces: returns the facenumbers of the 6 outer surfaces of the block

boundary conditions

IVector& constraintfaces_constainttypes: constraint types for the 6 outer surfaces of the block

IVector& constraintfaces_directions: directions that are constrained - for each outer surface individually

MBSLoad& bodyload: a body load to apply on all elements

int bodyloadactive: switch for the body loads

IVector& loadfaces_active: switches for 6 area loads perpendicular on the 6 outer surfaces

Vector& loadfaces_loadstrength: strengths for these 6 loads

simplified function calls:

- `GenerateHexahedralBlock(`
  `Box3D block, int3 divisions_xyz, int bodynr, int matnr, Vector3D color)`

- `GenerateHexahedralBlock(`
  `Box3D block, int3 divisions_xyz, int bodynr, int matnr, Vector3D color,`
  `IVector& blockelements, IVector& blocknodes, TArray<IVector*>& blockfaces)`

2.6.2 Generate cylinders, rings and discs

There are several options to generate the mesh of a cylinder, ring (cylinder with hole in the center) or disc (cylinder with small height). All the options, presented in this section, lead to a mesh that consists of hexahedral elements only. The axis of the generated object is equal to the z-axis, see Fig. 2.10.

2.6.2.1 GenerateDisc

This is the most general constructor in this section. The mesh of a cylindrical disc (with or without a hole in the center) is generated, see Fig. 2.11.

```
GenerateDisc(radius, radius_hole, height, divisions_angle, divisions_radial,
divisions_height, pos, bodynr, matnr, color, blockelements, blocknodes)
```

following options are available:

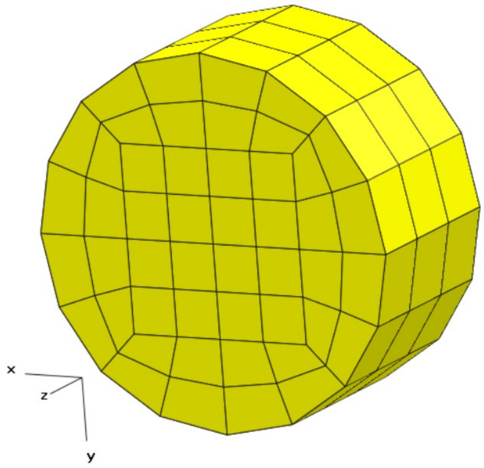


Figure 2.10: Mesh of a cylinder.

- geometric values are double values
- radius + height: radius and heigth of the cylindrical disc
- radius_hole: radius of the hole, if set to zero, than a disc without a hole is generated
- divisions are integer values, greater than or equal to 1
- divisions_angle: 45 degrees are divided into divisions_angle parts
- divisions_radial: the number of rings of which the disc is composed, the radius of the rings is growing arithmetic.
- divisions_height: the number of discs of which the cylinder is composed
- pos: center point of the area with the smallest z-value
- the others: see the general description of FEMesh

2.6.2.2 Other constructors

These constructors are used to implement `GenerateDisc` and can also be used directly:

```
GenerateCylinder(radius, small_radius, height, divisions_angle,
divisions_height, ...)
```

A cylinder is generated according to the first line of the left part of Fig. 2.11. small_radius defines the size of the square in the center of the circle.

```
GenerateHollowCylinder(radius, radius_hole, height, divisions_angle,
divisions_height, ...)
```

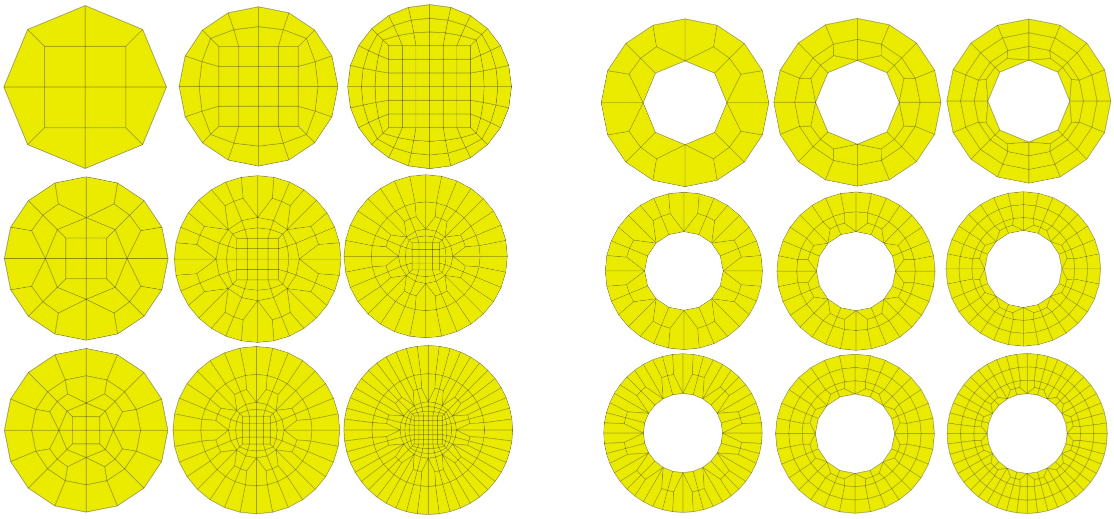


Figure 2.11: influence of divisions_angle (vertical) and divisions_radial (horizontal) on the mesh; left: radius_hole = 0; right: radius_hole = radius /2

A cylinder with a hole in the center is generated. The mesh is equal to the outer ring of the meshes presented in the last line of the left part of Fig. 2.11. The factor divisions_angle is equal for the inner and outer circle.

`GenerateHollowCylinderDoubleDiv(radius, radius_hole, height, divisions_angle, divisions_height, ...)`

A cylinder with a hole in the center is generated. The mesh is equal to the rings presented in the first column of the right part of Fig. 2.11. The factor divisions_angle of the outer circle is two times the factor of the inner circle.

2.6.2.3 correct procedure to add a FEMesh to MBS

1. Add Materials The Materials used in the finite elements **must** be in the FEMesh material list.
2. Add Nodes
3. Add Elements
4. Add Loads and Constraints

Figure 2.12: How to add an object (element, node,..) to the mbs

2.7 Models

This section describes

- how to add objects (elements,nodes,..) to the mbs and which functions are called when the mbs is assembled/initialized
- possibilities for error handling in the previous step

2.7.1 How to add objects to the mbs

In HOTINT there are at the moment 4 possibilities to add objects to the MBS:

1. Use the dialogs in the graphical user interface (GUI)
2. Load a mbs from a file with the button "Open MBS"
3. Define the objects with the constructors of the objects in the cpp-file
4. Use the function "ReadModelData" in the cpp-file

In figure 2.7.1 these options are shown.

2.7.1.1 Use the dialogs in the graphical user interface (GUI)

When the specific button (e.g. AddElement) is pressed, an object with default values is created. These default element data is presented to the user in a dialog. The data for this dialog is obtained from the object with the function `GetElementData`.

The user can edit the data in the GUI.

After pressing the button "OK", the data is written to the object with the function `SetElementData`.

Afterwards, the functions `Assemble` and `Initialize` are called directly from CWCDriver3DDlg.

The function `ModelChanged` is called at last, in order to update the graphical output.

2.7.1.2 Load a mbs from a file with the button "Open MBS"

When the button "Open MBS" is pressed, the user can search for a file.

The filename and directory are stored in the EDC GeneralOptions.ModelFile.hotint_input_data_filename.

Afterwards, the function `Initialize` is called directly from the CWCDriver3DDlg. `Initialize` calls the function `ReadModelData` and afterwards `Assemble` if a hotint_input_data_filename is specified.

The function `ModelChanged` is called at last directly from CWCDriver3DDlg, in order to update the graphical output.

2.7.1.3 Define the objects with the constructors of the objects in the cpp-file

It is possible to define the elements directly in the cpp-file of the model. These models can be selected with the button "Select Model". If the name of the model is already saved in the config-file, the same procedure is performed, as if the user would have chosen it.

The cpp-file is called from the function `Initialize` if no hotint_input_data_filename is defined.

After calling the constructor and additional optional Set-Functions, the object is added to the mbs e.g. with `AddElement(..)`.

After all objects are added, the mbs is assembled with the function `Assemble`, which is also called directly from the cpp-file of the model.

Initialization of the mbs and the objects is done directly in `Initialize`.

The function `ModelChanged` is called at last directly from CWCDriver3DDlg, in order to update the graphical output.

2.7.1.4 Use the function "ReadModelData" in the cpp-file

It is possible to combine the options described above: If you define your model in a cpp-file you can call at any position in the model the function `ReadModelData`.

If the file that is read with `ReadModelData` includes commands of the script language, these are processed imediately. If objects are added (e.g. with `AddElement`,`AddNode`,...), the following steps are performed:

- the object is created with default values
- the default values are replaced by the values defined in the EDC
- the object is added to the mbs

It is possible to add/edit/delete other objects of the mbs before and after `ReadModelData` is called.

The function `Assemble` has to be called in the cpp-file of the model.

2.7.2 Possibilities for error handling

In order to avoid errors due to inconsistent objects or mbs the following guidelines should be followed:

- Each object must have a constructor without any arguments except of `MBS*`
- In this constructor a function called ObjectDefaultConstructor should be called. This function sets all the default values.
- Use `Default_ParserObjectValues` as far as possible in the ObjectDefaultConstructor.
- Each object can have various additional SetFunctions to change the values set by the ObjectDefaultConstructor in a cpp-file. These Set-Functions are not available in script language or in the GUI!
- Script language and GUI are using `GetElementData` and `SetElementData` instead of the Set-Functions to change the element data.
- It is possible to implement additional functions that are called at the end of `SetElementData`, this also works if `SetElementDataAuto` is used.
- When `Assemble` is called, for all elements (including constraints!) the function `PreAssemble` is called, which can not be used for additional error handling, because it is a void function.
- After `PreAssemble` the ltg-lists and dependencies are built
- In `Initialize` (only available for elements and constraints) additional initialization can be done.
- After Initialization the object is drawn for the first time.
- `Initialize` is called again after the button "Start" is pressed
- The last function called before `ComputeSystem` is `CheckSystemConsistency` which is divided in
    - `CheckConsistencyLTG`
    - `CheckConsistencyElements` which calls `CheckConsistency`
    - `CheckConsistencySensors` which calls `IsConsistent`
    - `CheckConsistencyNodes` which calls `PerformNodeCheck`
    - `CheckConsistencyMaterials` which calls `CheckConsistency`

  If an error is detected here, the computation does not start.

Chapter 3

HOTINT Guidelines

Johannes Gerstmayr

December 23, 2013

3.1 How to get HOTINT running from the Source Code

3.1.1 Requirements

- Microsoft Visual Studio (preferred version 2008)

3.1.2 Unpack HOTINT

1. Unpack the zip file to a directory (preferred D:\cpp\).

3.1.3 When using Visual Studio 2008

3. Rename *WCDriver3D/WCDriver3D_mbs - Template 2008.sln* to *WCDriver3D/WCDriver3D_mbs.sln*

4. Start Visual Studio by double click on *WCDriver3D/WCDriver3D_mbs.sln*.

5. Rightclick the project WCDriver, click Eigenschaften, and make sure, the proper working directory path is set (see figure 3.1) for all configurations:

    - Befehl: `"..\hotint$(PlatformName)\$(ConfigurationName)\$(TargetFileName)"`
    - Arbeitsverzeichnis: `"..\hotint$(PlatformName)\$(ConfigurationName)"`

6. compile (debug & release). You should now be able to run HOTINT.

3.1.4 When using Visual Studio 2010

3. Rename *WCDriver3D/WCDriver3D_mbs - Template 2010.sln* to *WCDriver3D/WCDriver3D_mbs.sln*

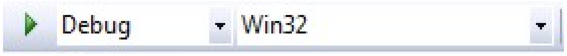
4. Start Visual Studio by double click on *WCDriver3D/WCDriver3D_mbs.sln*.

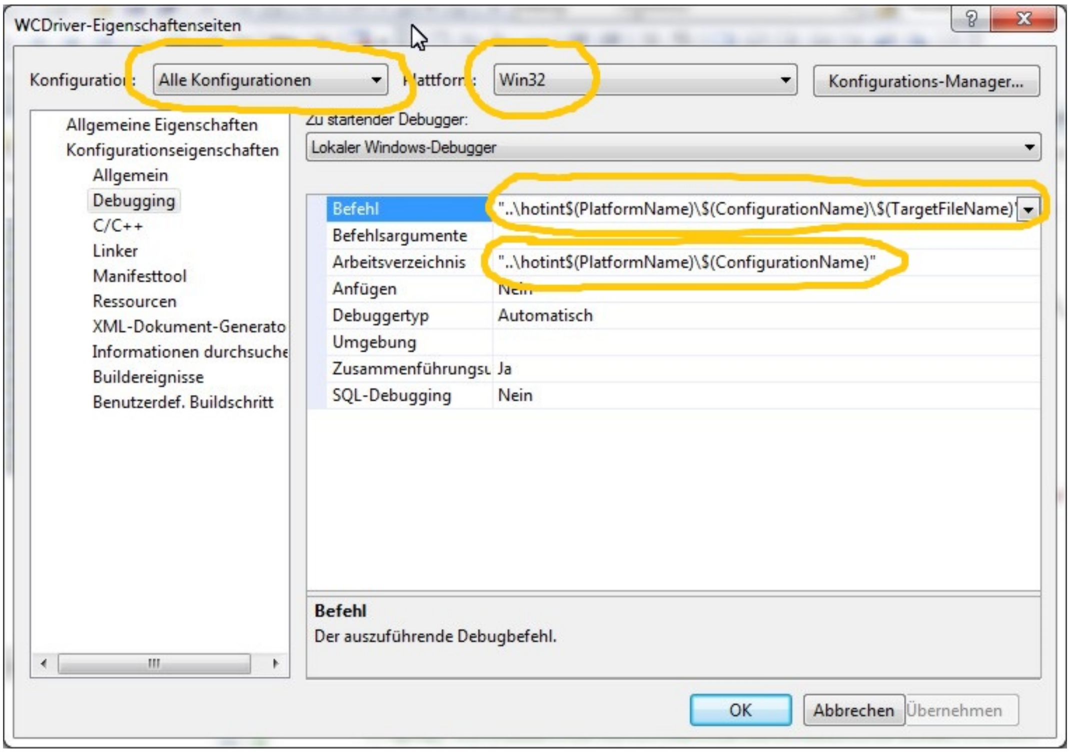
5. Rightclick the project WCDriver, click Eigenschaften, and make sure, the proper working directory path is set for all configurations:

    - Befehl: `..\HotInt$(Platform)\$(Configuration)\hotint.exe`
    - Arbeitsverzeichnis: `..\HotInt$(Platform)\$(Configuration)`

6. compile (debug & release). You should now be able to run HOTINT.

Please take care that the platform is set to *Win32*, since it is the only possible right now.





Konfiguration: Alle Konfigurationen

Plattform: Win32

Zu startender Debugger: Lokaler Windows-Debugger

Befehl	"..\hotint$(PlatformName)\$(ConfigurationName)\$(TargetFileName)"
Befehlsargumente	
Arbeitsverzeichnis	"..\hotint$(PlatformName)\$(ConfigurationName)"
Anfügen	Nein
Debuggertyp	Automatisch
Umgebung	
Zusammenführungsu	Ja
SQL-Debugging	Nein

Befehl
Der auszuführende Debugbefehl.

Figure 3.1: Set “Arbeitsverzeichnis” and “Befehl” for ALL CONFIGURATIONS in WCDriver

3.2 Coding Conventions

3.2.1 Comments

3.2.1.1 Use English Language throughout for comments

Everyone should have a chance to understand them.

3.2.1.2 Write detailed comments in header (.h) file

Description of the classes and functions. Here it is important to describe the input and output parameters and possible limitations etc., not the implementation details.

3.2.1.3 Write comments in implementation (.cpp) files

Here the implementation should be described. No need to repeat the information which is contained already in the header file. Here the implementation should be explained. It should be such that

1. The general idea of the code becomes obvious
2. the meaning of variables (“j”, etc.) becomes more obvious

3. parts of the code which are not easy to understand become more readable
4. the code becomes structured (use //++++++++++++++++++ and some small description of each part)

3.2.1.4 Commenting changes in common hotint code

Many developers are working at the program HOTINT. In order to keep an overview about the new features, the comment format for the changes in the common HOTINT-code is defined here.

Important or critical code changes (e.g.: changes of the solver code) are marked with an call sign

//$!FL YYYY[-MM-DD]:['['|']'] [text]

Less critical changes are marked with space sign instead of the call sign

//$ FL YYYY[-MM-DD]:['['|']'] [text]

If two developers want to write comments to the same code, the comments must be written one after another according to the described rules in one line or in a separate line.

Legend:

F...first name

L...last name

YYYY ... year

MM ... month

DD... day

[] means optional (is not part of comment)

| ... logical OR

'[' ...ASCII-sign: squared bracket open

']' ...ASCII-sign: squared bracket close

Examples:

//$ FL 2011-01-21:[start of comment example 1 with day

CODE

//$ FL 2011-01-21:] end of comment example 1 with day

...

//$ FL 2011-01:[start of comment example 2 without day

CODE

//$ FL 2011-01:] end of comment example 2 without day

...

//$ FL 2011-01: comment example 3 without day

...

//$ FL 2011-01-21: comment example 4 with day

//$!FL 2011-01-21: important code change 5 with day

//$!FL 2011: important code change 6 with year

Create Makro:

Following Makro in Visual Studio can be used to save time and make correct formatted comments.

1. Open *Visual Studio* .
2. Select *Extras* → *Makros*→ "TemporaryMacro aufzeichnen" and write an arbitrary comment e.g.:
   `//test`
3. Press red stop button or *Extras* → *Makros*→ "Aufzeichnung beenden"
4. Start *Extras* → *Makros*→ "Makro Explorer"
5. Right mouse click on "Temporary Makro"→ bearbeiten
6. Following code line
   ```
   DTE.ActiveDocument.Selection.Text = "//test"
   ```
   must be replaced with
   ```
   DTE.ActiveDocument.Selection.Text = "//$ FL " & Year(Now) & "-" & Month(Now) & "-"
   & Day(Now) & ":  "
   ```
7. "FL" means the first character of the first and last name (see legend)
8. Store macro and close "Microsoft Visual Studio Macros"

9. With “Strg+Shift+P” or *Extras* → *Makros*→ “TemporaryMacro ausführen”, the macro can be used.

Makro Code:

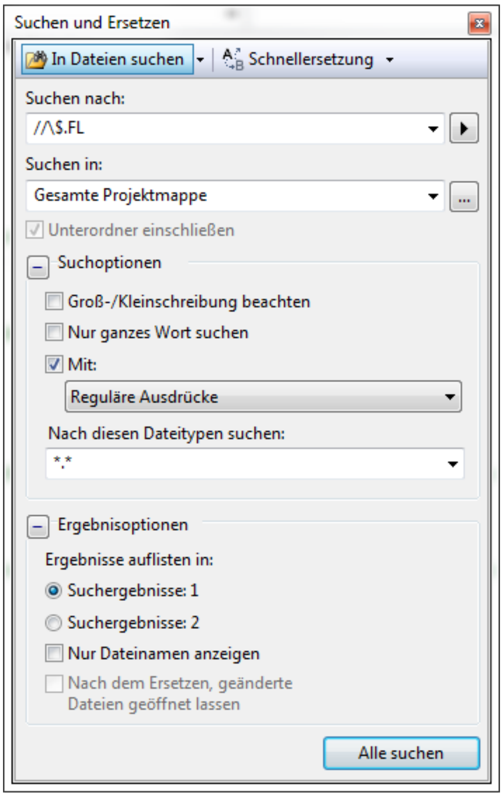
```
Option Strict Off
Option Explicit Off
Imports System
Imports EnvDTE
Imports EnvDTE80
Imports EnvDTE90
Imports System.Diagnostics

Public Module RecordingModule


   Sub TemporaryMacro()
         DTE.ActiveDocument.Selection.Text = "//$ FL " & Year(Now) & ↩
             ↪ "-" & Month(Now) & "-" & Day(Now) & ": "
   End Sub
End Module
```

Search with regular expression:

The comments can be found with regular expressions like in following figure:



Suchen nach: //\$.FL

Suchen in: Gesamte Projektmappe

☑ Unterordner einschließen

☐ Groß-/Kleinschreibung beachten

☐ Nur ganzes Wort suchen

☑ Mit: Reguläre Ausdrücke

Nach diesen Dateitypen suchen: *.*

Ergebnisse auflisten in:

● Suchergebnisse: 1

○ Suchergebnisse: 2

☐ Nur Dateinamen anzeigen

☐ Nach dem Ersetzen, geänderte Dateien geöffnet lassen

3.2.2 How to structure new HOTINT code?

A clear and uniform structure helps avoiding common errors.

3.2.2.1 Write & and * next to the data type

Write

```
int& ref = x;
```

instead of

```
int &ref = x;
```

and

```
int* ptr = &x;
```

instead of

```
int *ptr = &x;
```

Attention: do not write

```
int* a,b;
```

since that would mean that a is a pointer to an integer and b is an integer.

3.2.2.2 Declare each variable separately

If

- a variable is defined in the declaration (e.g.) or
- a pointer or reference is declared

always declare each variable separately in a new line.

So

```
int i, j, k;
```

or

```
Matrix mat1, mat2;
```

is ok, but not

```
int i=3, j=2, k=5;
```

or

```
Matrix* mat1, * mat2;
```

Write instead

```
int i=3;
int j=2;
int k=5;
```

respectively

```
Matrix* mat1;
Matrix* mat2;
```

3.2.2.3 Use braces

Write `if (i<1) {t=1;}` instead of `if (i<1) t=1;` since lines like `if (i<1) t=1; s=3;` may lead to errors which are hard to detect. If-commands with condition and body in one line should only be used for very short commands. In case of doubt always use seperate lines.

3.2.2.4 Write braces into separate lines

Use

```
for(int i=1; i<=10; i++)
{
  for(int j=1; j<=10; j++)
  {
    test = test+1;
  }
}
```

and not

```
for(int i=1; i<=10; i++){
  for(int j=1; j<=10; j++){
    test = test+1;
  }
}
```

or

```
for(int i=1; i<=10; i++)
  for(int j=1; j<=10; j++)
    test = test+1;
```

3.2.2.5 Do not use uncommon constructs

Code like

```
int sign=1;
if (elemind == 2) {sign = -1}
```

or

```
int sign;
if (elemind == 1)
{
  sign = 1
}
else
{//elemind == 2
  sign = -1
}
```

is much more common and understood by more people than

```
int sign = elemind == 2 ? -1 : 1;
```

3.2.2.6 Use a tab-space of 2 spaces

This ensures that the indentation is the same for everybody. However for new code please take care to make tabs when tabs are needed and spaces are needed. Then it does not matter how many spaces a tab-space corresponds.

3.2.2.7 Use spaces

For the usage of spaces exist a small set of rules.

- Use spaces in complicated expressions:

  ```
  complicated_expression = complicated_expression + 1;
  ```

  instead of

  ```
  complicated_expression=complicated_expression+1;
  ```

- Use spaces in parameter lists

  ```
  function(i ,j, k)
  ```

  instead of

  ```
  function(i,j,k)
  ```

- Use spaces in for-loops

```
for(int i=1; i<=2; i++)
```

and not

```
for(int i=1;i<=2;i++)
```

- Do not use spaces next to the scope operator. Write

  ```
  Matrix::get(x);
  ```

  and not

  ```
  Matrix :: get(x);
  ```

- Do not place a space between the function name and the parenthesis, but place one between the closing parenthesis and the const keyword.

  ```
  function(a, s, d, f) const
  ```

  and not

  ```
  function ( a, s, d, f)const
  ```

- The only exception are operators where a space should be between the operator name and the parenthesis, but not between the operator word and the operator sign.

  ```
  Matrix::operator+ (Matrix m1, Matrix m2);
  ```

  and not

  ```
  Matrix::operator+(Matrix m1, Matrix m2);
  ```

  or

  ```
  Matrix::operator + (Matrix m1, Matrix m2);
  ```

The general guideline is to make as much spaces as necessary to make the code as readable as possible. In case of doubt better insert an extra space.

3.2.3 Efficiency

3.2.3.1 Optimize the code only at parts where it is necessary

Functions that are invoked during every time step (e.g. EvalF2) should be highly effective, whereas the optimization of functions that are only called once (e.g. the initialization functions) do not need to be carried to extremes. For such functions it is better to keep the code short, readable and memory-saving.

3.2.3.2 Operations which include memory allocations: Vector, Matrix, TArray, mystring, and "<< outputs"

The allocation of Vector, Matrix, TArray and mystring invokes the new operator which is slow. Writing output with << will create a mystring and is therefore slow. So please avoid the creation of those objects during time critical sections. During initialization etc. there is no problem using them. Some hints about how to avoid Vector, Matrix, TArray and mystring:

- Use Matrix2D, Matrix3D, MatrixXD, Vector2D, Vector3D, ConstVector, and ConstMatrix instead. They get there memory from the stack. However too big ConstMatrix and ConstVector should be avoided since that could lead to stack overflow. The guide line is not to create ConstVector of dimension larger that 100 and analogously ConstMatrixes such that rows*columns is bigger than 100.

- Reuse matrices and vectors by using +=, *= and ApplySqrMat(M). So for example don't write

  ```
  vec1 = 2*vec1 + vec2;
  ```

  which would allocate 2 new vectors, write instead

  ```
  vec1 *= 2;
  vec1 += vec2;
  ```

  which does not allocate any vector. Don't forget to write a comment to keep the code readable.

- Define member variables which can be used as temporary matrices, vectors, etc. Take care that a temporary member variable is not used in two functions, to avoid errors that can be hardly detected. Static is no solution.

- If you want to return some debug message (in debug mode) and you need to concatenate mystring therefore, take care that this concatenation is actually done only when the string is needed (i.e. when being in debug mode).

3.2.3.3 Avoid "static"

The static keyword may cause errors that are difficult to find.

3.2.3.4 Try to reduce code

The shorter the code, the easier to understand and maintain.

3.3 If problems appear

3.3.1 If problems appear in HOTINT

- search in section 3.3.4 if a solution is already posted.
- write an email describing your problem to support@hotint.org

3.3.2 Solutions for problems, which might be caused by yourself (programming errors):

- use Debug version and search for origin of problem
- if code does not work in Debug version (bad errors), use UO().InstantMessage("step 1,2,3,...") in order to search where the problem might happen
- re-compile the whole code
- delete hotint_cfg.txt in the appropriate Release/Debug folder
- look at code which your recently changed
- try minimal version of your new code
- write an email describing your problem to support@hotint.org

3.3.3 Tips for debugging

3.3.3.1 'Meaningless' windows framework error

It occasionally occurres (especially using Windows 7) that you get a 'meaningless' framework error (e.g. mfc8d.dll!CWnd::IsDialogMessageA... which just tells you the Windows message 514 WM_LBUTTONUP, which is your mouse button) instead of an access violation error.
To detect the primary error do not choose your model after starting hotint, but insert your model into the according hotint_cfg.txt line and start debugging.

3.3.3.2 Expansion of class instances (variables) in the watch window

It is often useful to see important fields of frequently used variables in the watch-window of the debugger without expanding them and searching among other not so important fields. This can be achieved by adding simple instructions to the configuration file, which can be usually found at C:\Program Files (x86)\Microsoft Visual Studio 9\Common7\Packages\Debugger\autoexp.dat
The file can sometimes be protected for editing, so that the process will include the following stages:

1. Close Visual Studio.

2. Make a copy of autoexp.dat somewhere that you can edit it.

3. Open the file and in the very beginning of the section [AutoExpand] add the following lines

```
Vector3D = <vec[0]>,<vec[1]>,<vec[2]>
TArray<*> = l=<length>, data=<data[0]>,<data[1]>,<data[2]>,<data↩
   ↪ [3]>,<data[4]>,<data[5]>,<data[6]>,<data[7]>...
FieldVariableDescriptor = var=<variable_type>, c1=<↩
   ↪ component_index_1>, c2=<component_index_2>
```

4. Copy the file back to the Visual Studio directory.

5. Start debugging. Now in the watch window all variables of the types Vector3D, TArray and FieldVariableDescriptor should display something useful directly in the right column.

6. Add additional variable types, which you would like to have auto-expanded, with the fields you wish to see, to autoexp.dat

3.3.4 Known frequent problems

3.3.4.1 Can't find WorkingModule.dll

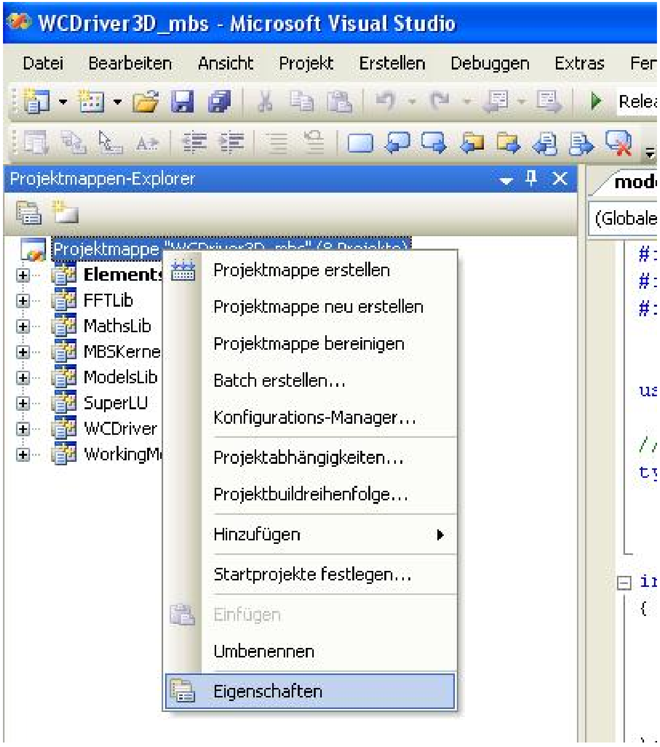
Occasionally occurres since, as a postbuild event, hotint and its dll's are moved to the directory HotIntWin32/release respectively HotIntWin32/debug

That's a problem where it is not yet sure, which of the followings steps is necessary, but the total of it serves as solution. If you encounter this problem please try the following order of solution steps and look after each step, if the problem is still there, to enhance this solution:

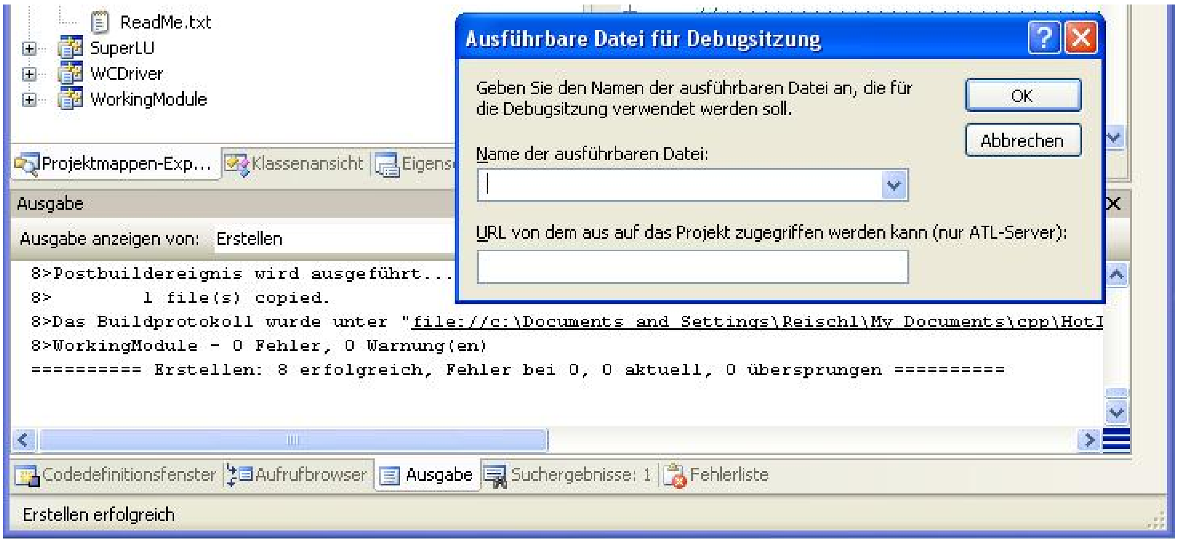
- look if the WorkindModule.dll is contained and up to date in the hotint/debug, hotint/release, hotint/release2 folder respectively.
- If yes try the folowing steps:
  - restart Visual Studio, delete hotint_cfg.txt (might not be necessary), newly recompile
  - restart Windows, delete hotint_cfg.txt (might not be necessary), newly recompile
- If no: recompile and check again (make sure you check the correct configuration). if still no: the project properties have to be checked (consult an experienced user).
- If specific dll's (e.g. from partner company) are not found by the Workingmodule.dll, the error message is also "Can't find ...". In this case, use the dependency walker to locate the missing dll's.

3.3.4.2 Visual Studio is asking for 'Ausführbare Datei für Debugsitzung'

This errors occured on a x64 system with windows xp. MS Visual Studio 2005 was installed and after the update (SP1) was done the following error appeared:



The update is somehow changing the settings of 'Startprojekt' which can be fixed in the following way:



Ausführbare Datei für Debugsitzung

Geben Sie den Namen der ausführbaren Datei an, die für die Debugsitzung verwendet werden soll.

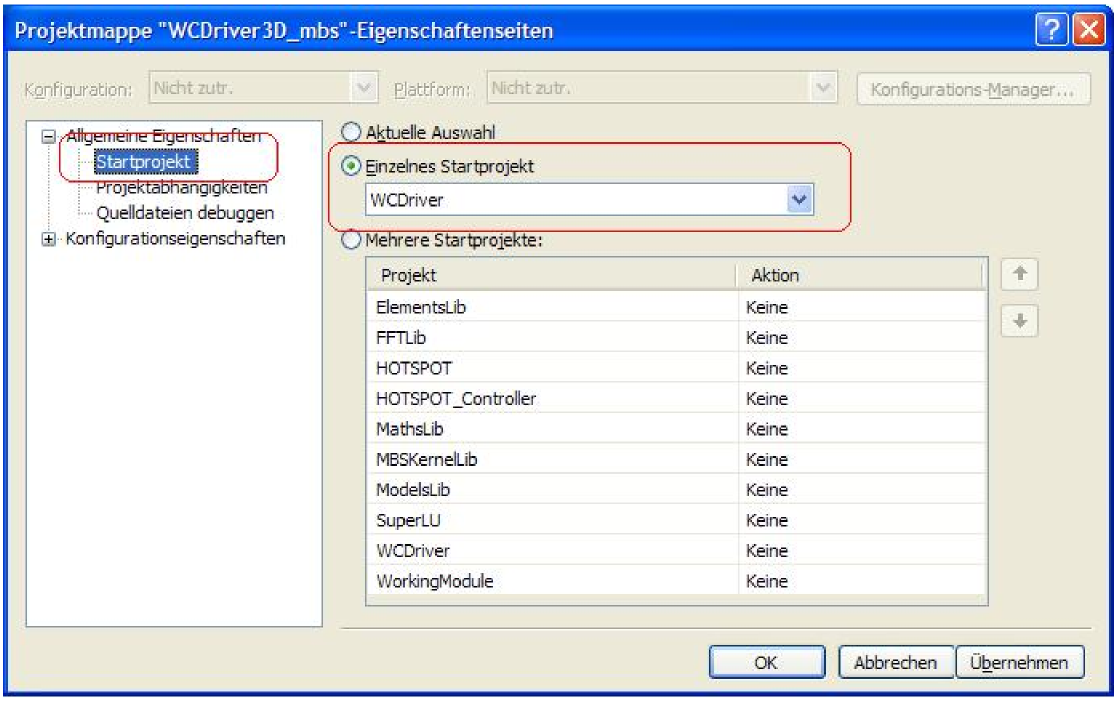
Name der ausführbaren Datei:

URL von dem aus auf das Projekt zugegriffen werden kann (nur ATL-Server):

Ausgabe anzeigen von: Erstellen

```
8>Postbuildereignis wird ausgeführt...
8>        1 file(s) copied.
8>Das Buildprotokoll wurde unter "file://c:\Documents and Settings\Reischl\My Documents\cpp\HotI
8>WorkingModule - 0 Fehler, 0 Warnung(en)
========== Erstellen: 8 erfolgreich, Fehler bei 0, 0 aktuell, 0 übersprungen ==========
```

Erstellen erfolgreich


Projektmappe "WCDriver3D_mbs"-Eigenschaftenseiten
Konfigurations-Manager...
Allgemeine Eigenschaften
Startprojekt
Projektabhängigkeiten
Quelldateien debuggen
Konfigurationseigenschaften
OK
Abbrechen
Übernehmen

Konfiguration: Nicht zutr.
Plattform: Nicht zutr.

○ Aktuelle Auswahl
● Einzelnes Startprojekt
WCDriver
○ Mehrere Startprojekte:

Projekt	Aktion
ElementsLib	Keine
FFTLib	Keine
HOTSPOT	Keine
HOTSPOT_Controller	Keine
MathsLib	Keine
MBSKernelLib	Keine
ModelsLib	Keine
SuperLU	Keine
WCDriver	Keine
WorkingModule	Keine

Afterwards you have to rebuild your project (Strg+Alt+F7)

3.4 How to create an installer

If you want to distribute your Project to a customer it is a good way to do this by the use of an installer.

The following guideline is tested for Microsoft Visual Studio 2005!

This section consists of 2 tutorials:

1. How to create an installer and set up your system for the first time in VisualStudio

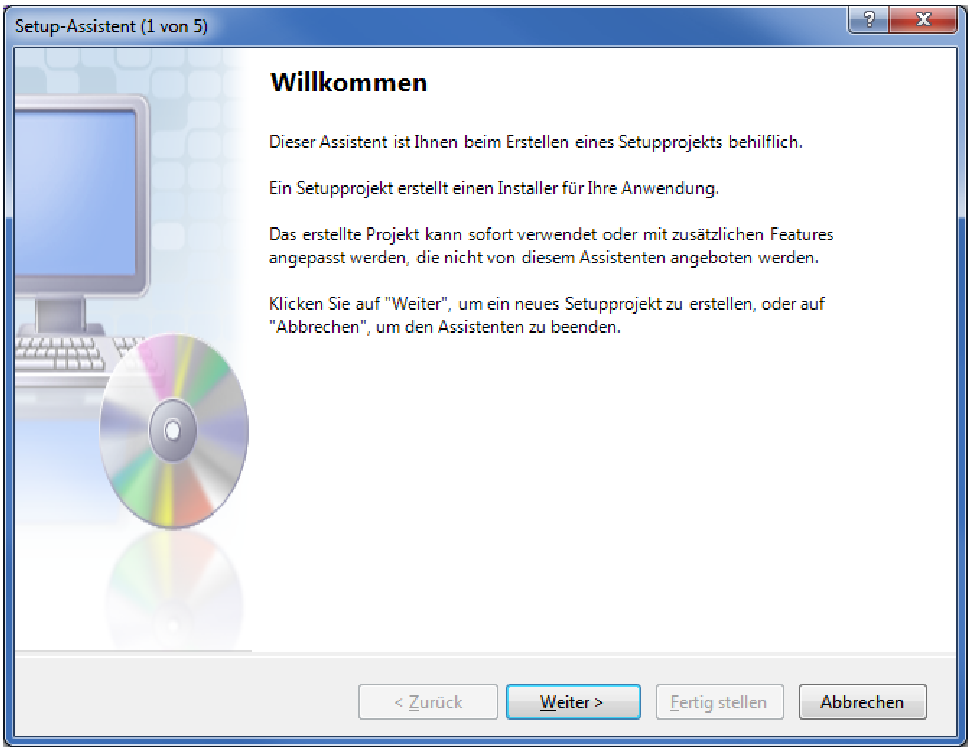
2. How to create an installer for a specific project

3.4.1 How to create an installer and set up your system for the first time in VisualStudio

1. Make sure, that your model compiles and runs correctly in Release Mode

2. Create a new Install-project (www.youtube.com/watch?v=d0h7CqC3Ltg)

   (a) Datei/Neu/Projekt: choose options like in the following figures (Fig. 3.2 - 3.5)



Willkommen

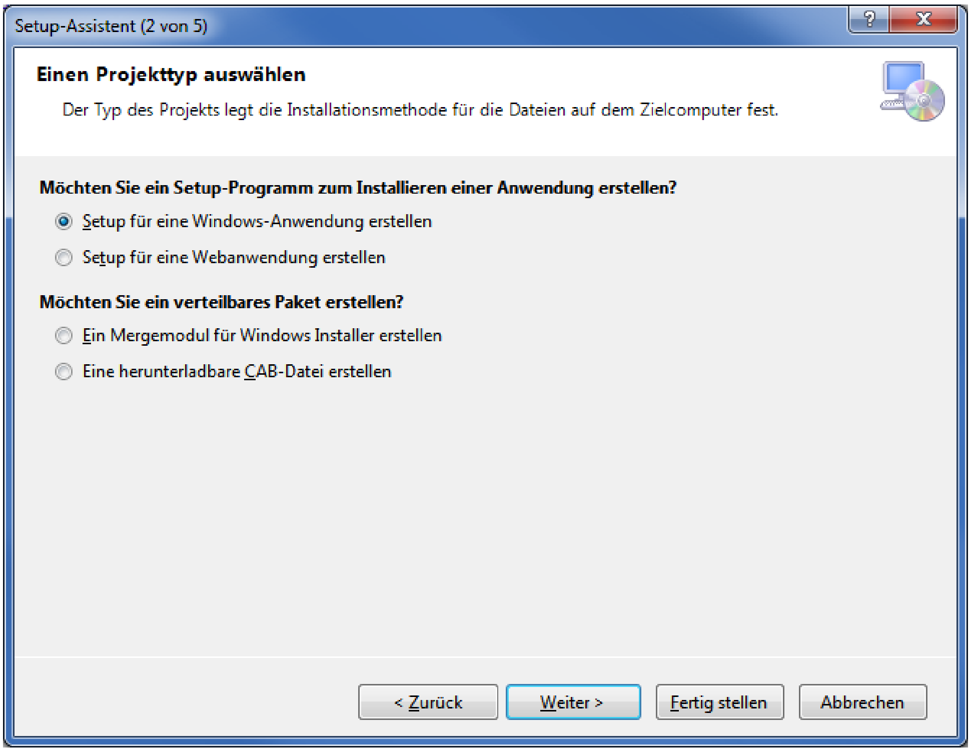
Dieser Assistent ist Ihnen beim Erstellen eines Setupprojekts behilflich.

Ein Setupprojekt erstellt einen Installer für Ihre Anwendung.

Das erstellte Projekt kann sofort verwendet oder mit zusätzlichen Features angepasst werden, die nicht von diesem Assistenten angeboten werden.

Klicken Sie auf "Weiter", um ein neues Setupprojekt zu erstellen, oder auf "Abbrechen", um den Assistenten zu beenden.

Figure 3.2: Start Wizard



Einen Projekttyp auswählen

Der Typ des Projekts legt die Installationsmethode für die Dateien auf dem Zielcomputer fest.

Möchten Sie ein Setup-Programm zum Installieren einer Anwendung erstellen?

● Setup für eine Windows-Anwendung erstellen

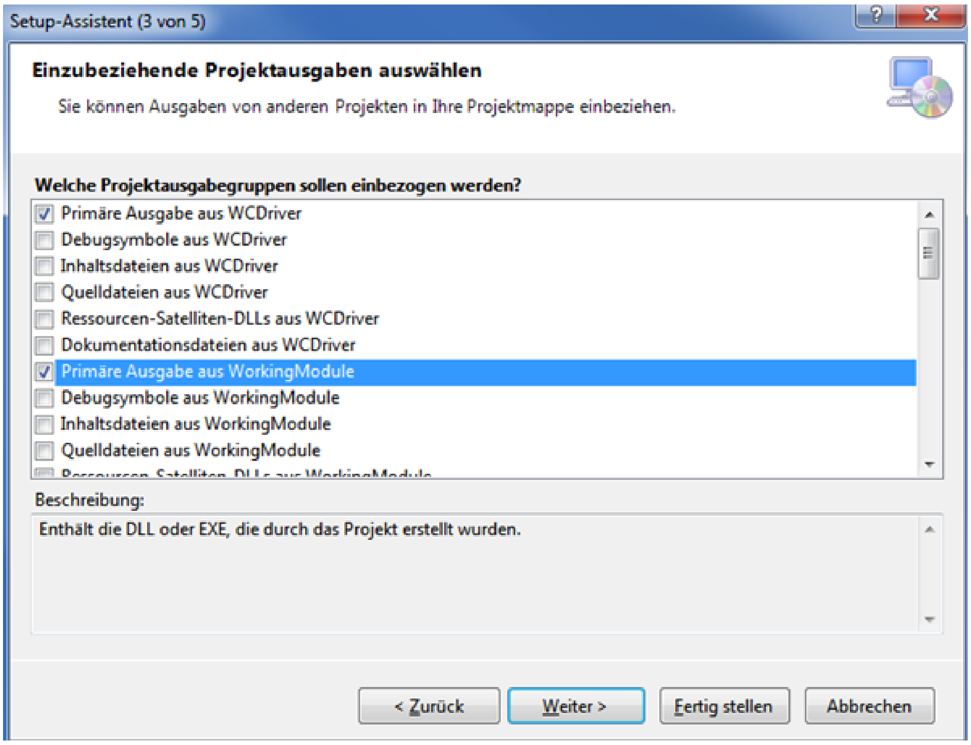
○ Setup für eine Webanwendung erstellen

Möchten Sie ein verteilbares Paket erstellen?

○ Ein Mergemodul für Windows Installer erstellen

○ Eine herunterladbare CAB-Datei erstellen

Figure 3.3: chose type of project



Einzubeziehende Projektausgaben auswählen

Sie können Ausgaben von anderen Projekten in Ihre Projektmappe einbeziehen.

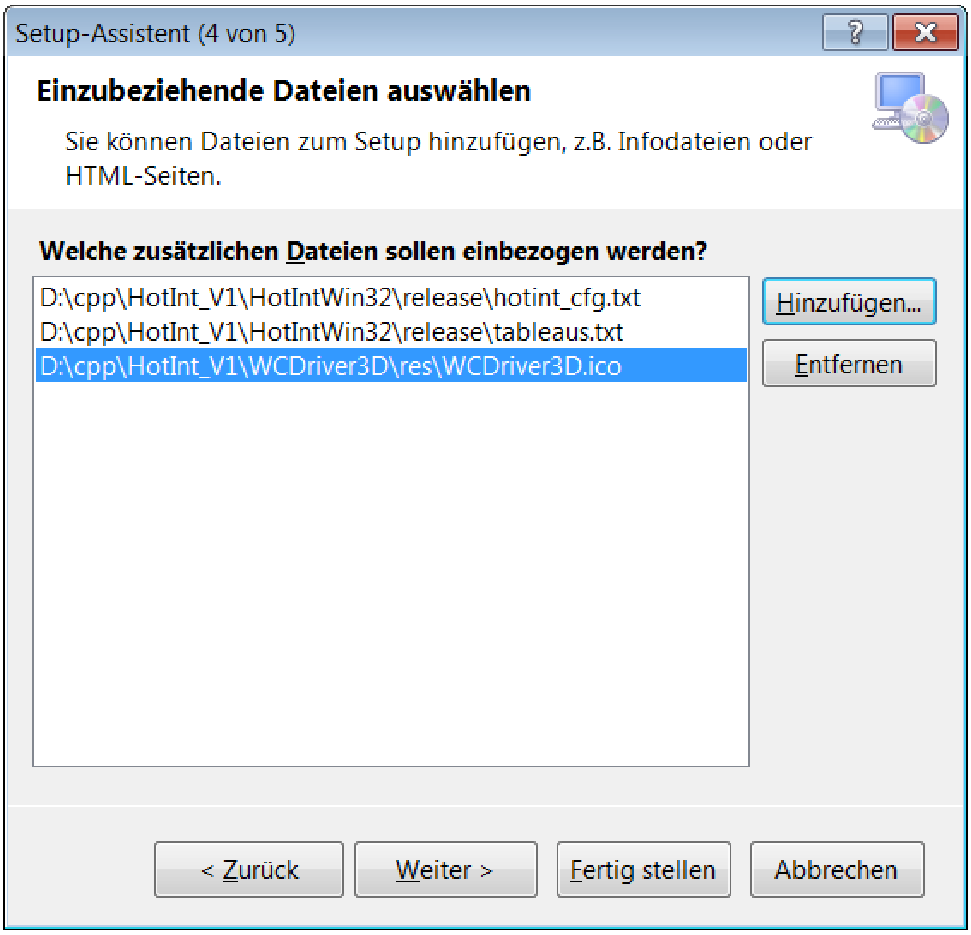
Welche Projektausgabegruppen sollen einbezogen werden?

- ☑ Primäre Ausgabe aus WCDriver
- ☐ Debugsymbole aus WCDriver
- ☐ Inhaltsdateien aus WCDriver
- ☐ Quelldateien aus WCDriver
- ☐ Ressourcen-Satelliten-DLLs aus WCDriver
- ☐ Dokumentationsdateien aus WCDriver
- ☑ Primäre Ausgabe aus WorkingModule
- ☐ Debugsymbole aus WorkingModule
- ☐ Inhaltsdateien aus WorkingModule
- ☐ Quelldateien aus WorkingModule
- ☐ Ressourcen-Satelliten-DLLs aus WorkingModule

Beschreibung:

Enthält die DLL oder EXE, die durch das Projekt erstellt wurden.

Figure 3.4: Also Check: ”Primäre Ausgabe aus MBSElementsAndModels”



Einzubeziehende Dateien auswählen

Sie können Dateien zum Setup hinzufügen, z.B. Infodateien oder HTML-Seiten.

Welche zusätzlichen Dateien sollen einbezogen werden?

D:\cpp\HotInt_V1\HotIntWin32\release\hotint_cfg.txt
D:\cpp\HotInt_V1\HotIntWin32\release\tableaus.txt
D:\cpp\HotInt_V1\WCDriver3D\res\WCDriver3D.ico

Figure 3.5: add additional files

(b) add all DLLs (only DLLs and in these folders but no sub-folders!) you are using (be sure to use the correct platform x86 or x64) to the same folder as the previous files

- VC: all DLLs in the (5 sub-) folders of C:\Program Files (x86)\Microsoft Visual Studio 8\VC\redist\x86
- Compiler (e.g. C:\Program Files (x86)\Intel\ComposerXE-2011\redist\ia32\compiler)
- If the folder C:\Program Files (x86)\Intel\ComposerXE-2011\is not found, search the file 'redist.txt' in the 'Intel'-folder. This text file contains the DLLs for your platform x86 or x64. Search and add those platform-specific DLLs *

(c) "Fertig stellen"

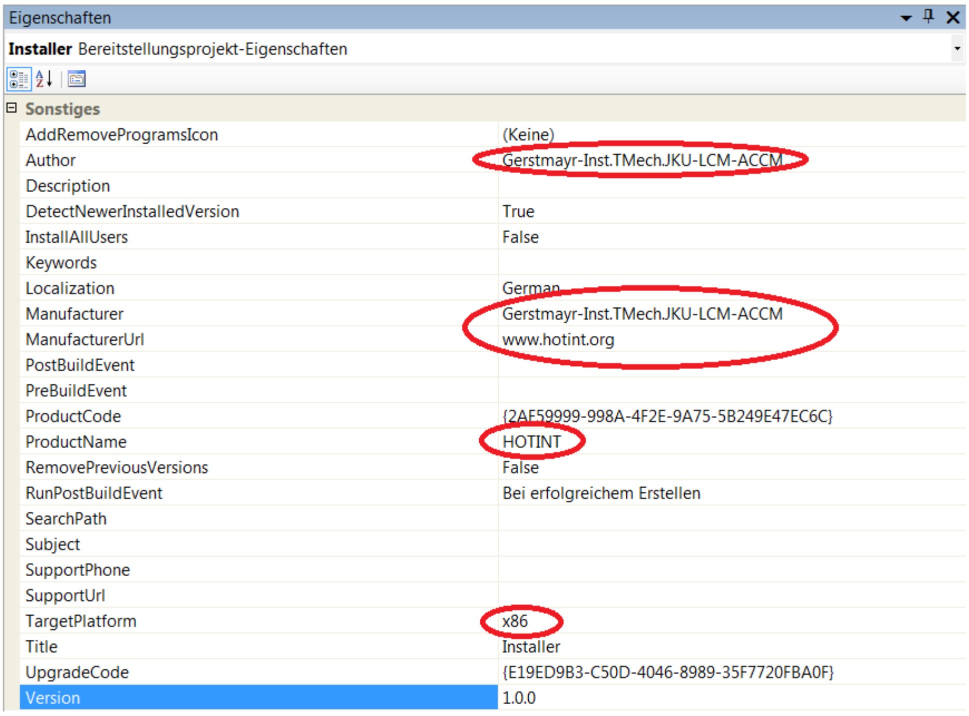
(d) If there are warnings like "the file xyz.dll is already added, do you want do add the other file?" always answer with "yes"

(e) Now there is a new Project "Installer"

3. In the folder "Gefundene Abhängigkeiten" you can see all dlls. Remove the following dlls (right mouse button and then "Ausschliessen"):

- opengl32.dll
- glu32.dll

4. Click on "Installer", than set the properties in the "Eigenschaftenfenster":



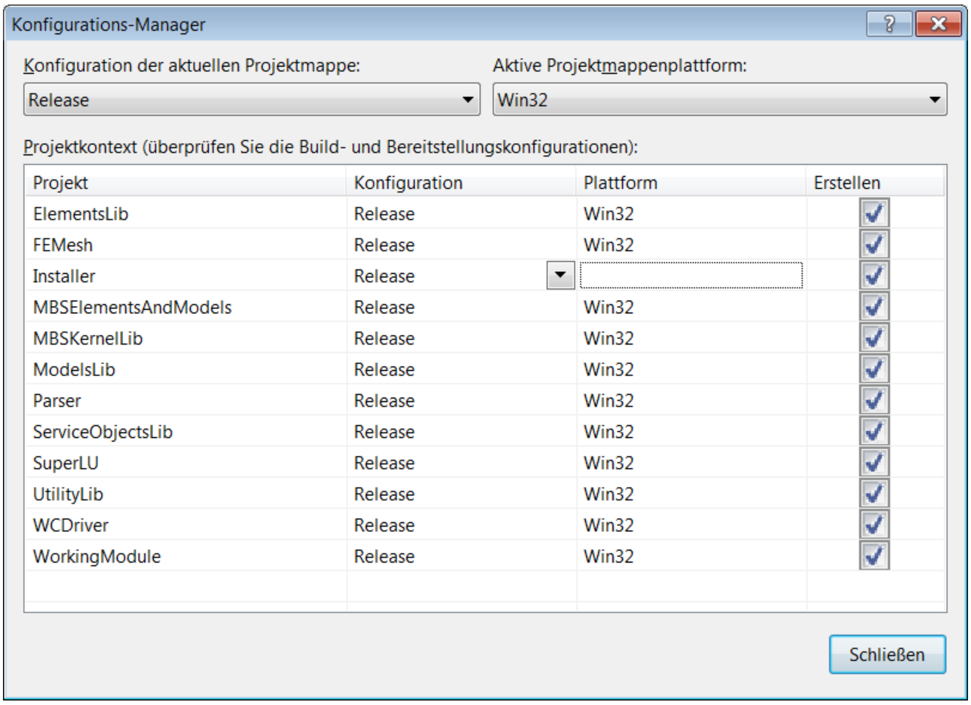
Sonstiges

AddRemoveProgramsIcon	(Keine)
Author	Gerstmayr-Inst.TMech.JKU-LCM-ACCM
Description	
DetectNewerInstalledVersion	True
InstallAllUsers	False
Keywords	
Localization	German
Manufacturer	Gerstmayr-Inst.TMech.JKU-LCM-ACCM
ManufacturerUrl	www.hotint.org
PostBuildEvent	
PreBuildEvent	
ProductCode	{2AF59999-998A-4F2E-9A75-5B249E47EC6C}
ProductName	HOTINT
RemovePreviousVersions	False
RunPostBuildEvent	Bei erfolgreichem Erstellen
SearchPath	
Subject	
SupportPhone	
SupportUrl	
TargetPlatform	x86
Title	Installer
UpgradeCode	{E19ED9B3-C50D-4046-8989-35F7720FBA0F}
Version	1.0.0

Figure 3.6: Also Change Localization to English!

5. change properties of installproject

- go to 'Eigenschaften von Installproject' → Konfigurationsmanager → check 'Erstellen'



Konfiguration der aktuellen Projektmappe: Release

Aktive Projektmappenplattform: Win32

Projektkontext (überprüfen Sie die Build- und Bereitstellungskonfigurationen):

Projekt	Konfiguration	Plattform	Erstellen
ElementsLib	Release	Win32	☑
FEMesh	Release	Win32	☑
Installer	Release		☑
MBSElementsAndModels	Release	Win32	☑
MBSKernelLib	Release	Win32	☑
ModelsLib	Release	Win32	☑
Parser	Release	Win32	☑
ServiceObjectsLib	Release	Win32	☑
SuperLU	Release	Win32	☑
UtilityLib	Release	Win32	☑
WCDriver	Release	Win32	☑
WorkingModule	Release	Win32	☑

Figure 3.7: Konfigurationsmanager

6. Add folders and Files to Filesystem (this folders are installed on the destination computer)

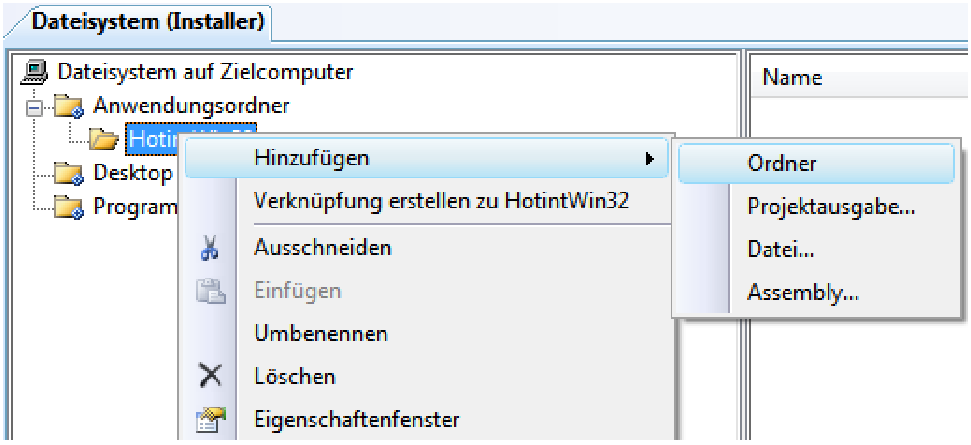


Figure 3.8: Filesystem

Create the following folders.

- documentation/examples
- output
- userdata
- HotIntWin32/Release

Do not forget to set ”AlwaysCreate” to ”true” in the properties

7. Remove [Manufacturer] from ”Anwendungsordner”-Property ”DefaultLocation” It should be: [ProgramFilesFolder]\[ProductName]

8. Move Files from "Anwendungsordner" to Release

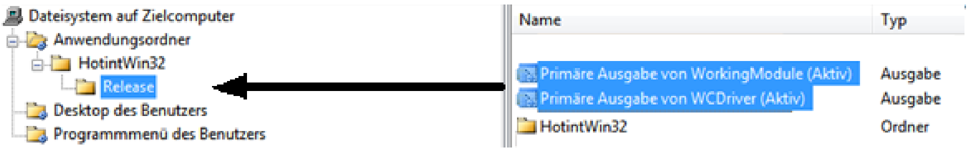

Dateisystem auf Zielcomputer
Anwendungsordner
HotintWin32
Release
Desktop des Benutzers
Programmmenü des Benutzers
Name
Typ
Primäre Ausgabe von WorkingModule (Aktiv)
Ausgabe
Primäre Ausgabe von WCDriver (Aktiv)
Ausgabe
HotintWin32
Ordner

Figure 3.9: Move files from the "Anwendungsorder" to the "HotIntWin32/Release"-folder

You have to move the following files:

- Primäre Ausgabe von WorkingModule
- Primäre Ausgabe von WCDriver
- Primäre Ausgabe von MBSElementsAndModels
- hotint_cfg.txt
- tableaus.txt

Add the dll-files of the folder HotintWin32\Release-Folder to the folder release

- blas_win32_MT.dll
- lapack_win32_MT.dll

9. Check license file
   - Add the rtf-file "HotInt_V1/licenses/License_Free.rtf" to Anwendungsordner/documentation
   - lapack_win32_MT.dll

10. Edit the GUI of the Installer:
    (a) Check dialog boxes of installer (modify if necessary with right mouse click on "Installer"-then Project/Ansicht/Benutzeroberfläche) The entries should be as follows: Willkommen (Copyright Warning and Welcome Text), Fertig, UpdateText. The entries have to be equal for "Installieren" and "Administratorinstallation".
    (b) Add the License Agreement:
      - right click on "Starten": Dialogfeld hinzufugen / Lizenzvertrag
      - Set the entry "LicenseFile" in the properties to the rtf-file
      - Move the "Lizenzvertrag" to the correct position (before "Installation bestätigen"!)
      - Make sure that this step is done for "Installieren" and "Administratorinstallation"

11. Compile the whole project (get a coffee)

12. The executable of the installer is in the folder Installer/Release

13. copy the folder with the installer

14. go to another computer and test your installer

15. **The installer is not finished yet!**
    You have to also follow the steps, which are described in the section 3.4.2. Otherwise important things like version numbers or license may be wrong!

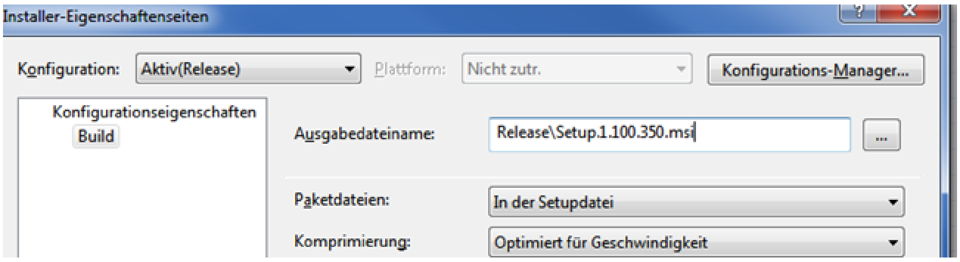
3.4.2 How to create an installer for a specific project

You need at least 15 minutes for this procedure!

1. Make sure, that you already have set up an installer before, otherwise follow the tutorial for creating an installer for the first time, see 3.4.1.
2. if it is for the homepage: create a new log entry, that a new release is ready
   this changes the version number in the file WorkingModule \ hotint_version.h
3. if it is for the homepage: create a new docu
   (a) include models_auto_documentation.cpp in all_models.h
   (b) Compile project
   (c) delete config-file
   (d) run model ”generate tex-files for docu”
   (e) start TeXniCenter and compile documentation/HID_USERs.tex
   (f) check the pdf (version, date,)
   (g) run the script generate_files_for_highlighting in the folder development \ autogeneration
4. Make sure, that your model compiles and runs correctly, and no other models are included, check all_models.h!
5. Check if the hotint_config.txt is correct. Especially make sure, that the flag ”GeneralOptions.Application.remove_experimental_menu_items” is true. Correct and save(!) the hotint_config.txt if necessary!
6. Check if menu has correct view (no experimental options or elements)
7. Add the Projekt ”Installer/Installer.vdproj” to your ”Projektmappe”
8. Add files and folders, that are necessary for your model to the ”Dateisystem(Installer)” (right mouse click on ”Installer”- Project / Ansicht /Dateisystem)
   (a) if it is for the homepage:
      i. add examples in documentation/examples
      ii. check if the hotint_version in the examples is set correct
      iii. rename HID_USERs.pdf to ”documentation.pdf” and add it to the folder documentation
      iv. add the 2 files
         - HOTINT.xml
         - hotint_highlight_notepad.xml

         to the folder documentation.
   (b) if it is for a company partner: add your specific files
9. Set correct version number

   (a) Set the name of the outputfile: SetupX.Y.Z.msi [1], see Figure 3.10 (right mouse click on "Installer"- Project/Eigenschaften to get this window)



Installer-Eigenschaftenseiten

Konfiguration: Aktiv(Release)

Plattform: Nicht zutr.

Ausgabedateiname: Release\Setup.1.100.350.msi

Paketdateien: In der Setupdatei

Komprimierung: Optimiert für Geschwindigkeit

Figure 3.10: Name of the installer file

   (b) Set the correct version number in the properties of the installer-project

10. Check dialog boxes of installer (modify if necessary with right mouse click on "Installer"-Project /Ansicht /Benutzeroberfläche)

11. Check license file!

    (a) If it is for the Release on www.hotint.org:
    Add the rtf-file " HotInt_V1/licenses/License_Free.rtf" to Anwendungsordner/documentation

    (b) If it is for a company partner:
    replace the rtf file e.g. by License_CompanyPartners.rtf

    (c) change the property settings in the dialog boxes of installer to the correct rtf-file

12. Check registry entries from setup (if they are needed)

13. Compile the whole project in Mode Release Win32

14. The executable of the installer is in the folder Installer/Release

15. Test your installer at another computer

3.5 How to include Intel MKL

Requirements:

- HOTINT
- MKL

First Steps:

---

[1] X Major update, Y Minor update, Z newest number of changes_log. X and Y are defined by Johannes Gerstmayr

1. Install Intel MKL
2. Run mklvars32.bat, if the variables are not set afterwards,
   open Systemsteuerung → System → Erweitert → Umgebungsvariablen
   and set the variables as defined in mklvars32.bat as new variables with the according file path. Check if variables are set via echo %variablename% in the command line.
3. Make sure *Projektmappen-Explorer* is active (*Ansicht > Projektmappen Explorer*)
4. Select *Extras > Optionen* and *Projekte und Projektmappen > VC++-Verzeichnisse*
5. From the *Verzeichnisse anzeigen für* list, select *Includedateien*
   Add the directory for the Intel MKL include files, that is "<mkl directory> \include"
6. From the *Verzeichnisse anzeigen für* list, select *Bibliotheksdateien*
   Add the directory for the Intel MKL include files, that is "<mkl directory> \ia32\lib"
7. From the *Verzeichnisse anzeigen für* list, select *Ausführbare Dateien*
   Add the directory for the Intel MKL include files, that is "<mkl directory> \ia32\bin"
8. Modify WorkingModule properties:
   Linker → Eingabe: Zusätzliche Abhängigkeiten: mkl_intel_c_dll.lib mkl_intel_thread_dll.lib libiomp5md.lib mkl_core_dll.lib
   C/C++ → Präprozessor: Präprozessordefinitionen: USE_MKL
   If you want to use Pardiso from MKL also set
   C/C++ → Präprozessor: Präprozessordefinitionen: USE_PARDISO
   NOTE: if you only use USE_PARDISO without the MKL, follow the "Getting Started with Pardiso using Hotint"-file and make sure to remove the *Zusätzliche Abhängigkeiten: mkl_intel_c_dll.lib mkl_intel_thread_dll.lib libiomp5md.lib mkl_core_dll.lib* which cause trouble otherwise.

3.6 How to include Pardiso (Solver)

Requirements:

- Internet
- HOTINT

First Steps:

1. Open http://www.pardiso-project.org and select "Download Software and License"
2. Choose between Academic and Commercial
3. Fill in your Data
   The license is specified for the computer you want to use it on, therefore you have to specify under Windows with hostname in capital letters (uppercase) and the user identification with the username.

4. Submit Form.
5. You will receive an email containing a download link where you can download the files for the use of Pardiso under Windows.
6. The email also contains your license key for the specified host and user.
   Save the key without surrounding whitespace in a file named “pardiso.lic”.
   The license is valid for one year.
7. Paste “libpardiso400_INTEL_IA32.lib” into arbitrary folder (e.g. “C:\ .. \cpp\pardiso\”)
8. Paste the files “libpardiso400_INTEL_IA32.dll”, “libiomp5md.dll” and “pardiso.lic” into the folders “../WCDriver3D/Release”, “../WCDriver3D/Release2” and “../WCDriver3D/Debug”.
9. Switch to or stay in configuration *Release.*
10. Modify WCDriver properties:
    Linker → Allgemein: Zusätzliche Bibliotheksverzeichnisse: e.g. “C:\ .. \cpp\pardiso\”
11. Modify WorkingModule properties:
    Linker → Allgemein: Zusätzliche Bibliotheksverzeichnisse: e.g. “C:\ .. \cpp\pardiso\”
    Linker → Eingabe: Zusätzliche Abhängigkeiten: libpardiso400_INTEL_IA32.lib
    C/C++ → Präprozessor: Präprozessordefinitionen: USE_PARDISO
12. Use Pardiso

Hints:

- Do not use these adjustments in the configuration *Debug.*
- If you want to use Pardiso also in the configuration Release2, repeat steps 10-12 within this configuration.
- If you want to stop using Pardiso, but may want to use it later on again, just change the *Präprozessordefinitionen* in Step 11 to noUSE_PARDISO. (It is a descriptive change, easily withdrawn.)

3.7 How to include BLAS/LAPACK

Requirements:

- HOTINT
- BLAS/LAPACK

First Steps:

1. If you are using the Intel MKL (WorkingModule → Eigenschaften: C/C++ → Präprozessordefinitionen USE_MKL), do nothing!

2. If you are NOT using the Intel MKL:
   Edit your system path: Systemsteuerung > System > Erweitert > Umgebungsvariablen > Systemvariablen:
   Path = <old_path>;<mylocalhotintdir> \WCDriver3D\extern_lib

3. Make sure *Projektmappen-Explorer* is active (*Ansicht* > *Projektmappen Explorer*)

4. Modify WorkingModule properties:
   Linker → Eingabe: Zusätzliche Abhängigkeiten: lapack_win32_MT.lib, blas_win32_MT.lib.
   Linker → Allgemein: Zusätzliche Bibiotheksverzeichnisse: ..\WCDriver3D\extern_lib

3.8 Useful Visual Studio settings

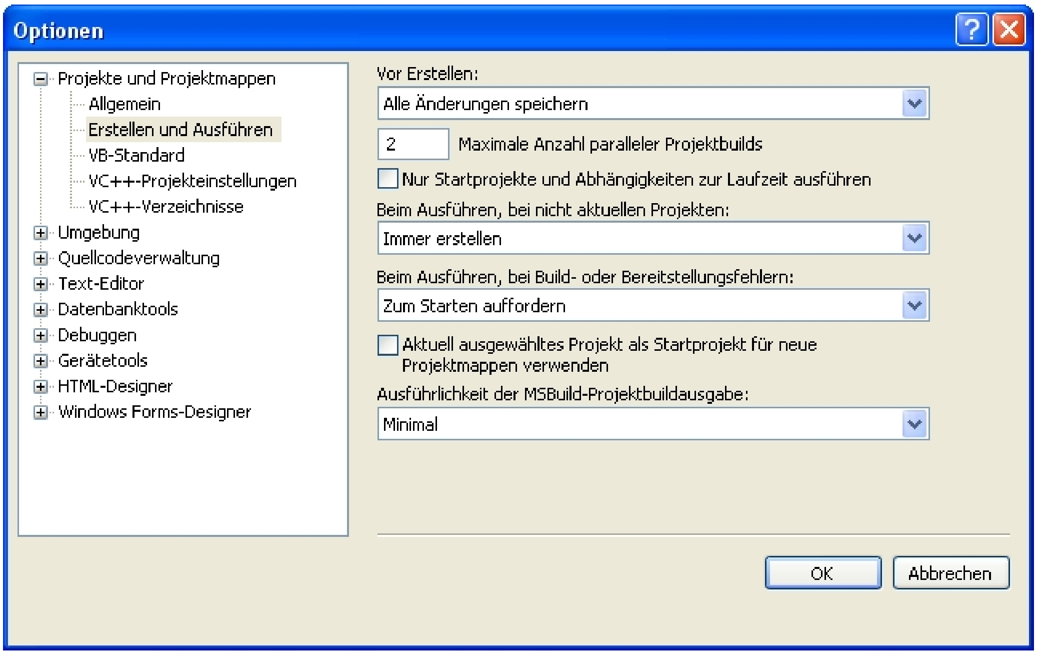
3.8.1 Save and Build before start

By default the combination Ctrl+F5 starts the program without debugging and without building the project.

If one wants that the project is saved and compiled before starting, one has to modify the settings of Visual Studio.

Go to Extras→Optionen→Projekte und Projektmappen→Erstellen und Ausführen

Change the option "Beim Ausführen, bei nicht aktuellen Projekten" to "Immer erstellen".



Vor Erstellen: Alle Änderungen speichern

2 Maximale Anzahl paralleler Projektbuilds

☐ Nur Startprojekte und Abhängigkeiten zur Laufzeit ausführen

Beim Ausführen, bei nicht aktuellen Projekten: Immer erstellen

Beim Ausführen, bei Build- oder Bereitstellungsfehlern: Zum Starten auffordern

☐ Aktuell ausgewähltes Projekt als Startprojekt für neue Projektmappen verwenden

Ausführlichkeit der MSBuild-Projektbuildausgabe: Minimal

3.8.2 Expansion of class instances (variables) in the watch window

It is often useful to see important fields of frequently used variables in the watch-window of the debugger without expanding them and searching among other not so important fields. This can be achieved by adding simple instructions to the configuration file, which can be usually found at C:\Program Files (x86)\Microsoft Visual Studio 8\Common7\Packages\Debugger\autoexp.dat The file can sometimes be protected for editing, so that the process will include the following stages:

1. Close Visual Studio.

2. Make a copy of autoexp.dat somewhere that you can edit it.

3. Open the file and in the very beginning of the section [AutoExpand] add the following lines

```
Vector3D = <vec[0]>,<vec[1]>,<vec[2]>
TArray<{*}> = l=<length>, data=<data[0]>, <data[1]>,
<data[2]>, <data[3]>, <data[4]>, <data[5]>, <data[6]>, <data↩
    ↪ [7]>...
FieldVariableDescriptor = var=<variable_type>, c1=<↩
    ↪ component_index_1>, c2=<component_index_2>
```

4. Copy the file back to the Visual Studio directory.

5. Start debugging. Now in the watch window all variables of the types Vector3D, TArray and FieldVariableDescriptor should display something useful directly in the right column.

6. Add additional variable types, which you would like to have auto-expanded, with the fields you wish to see, to autoexp.dat

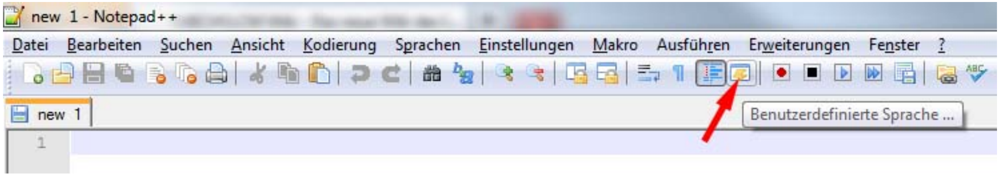
3.9 AutoComplete and Highlight HOTINT syntax in notepad+

- Go to the folder Hotint_V1\development\autogeneration.

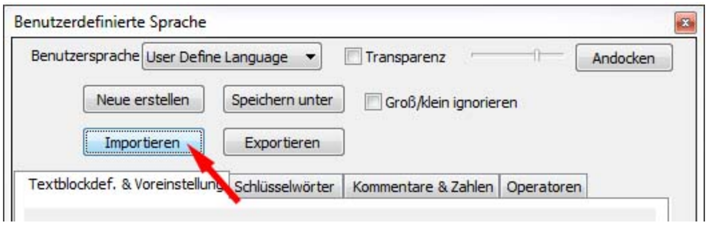
- Copy the file HOTINT.xml to the notepad folder 'plugins\APIs' (e.g. C:\Program Files (x86)\Notepad++\plugins\APIs)

- open NOTEPAD++

- click on icon User defined language


new 1 - Notepad++
Datei Bearbeiten Suchen Ansicht Kodierung Sprachen Einstellungen Makro Ausführen Erweiterungen Fenster ?
new 1
Benutzerdefinierte Sprache ...
1

- import file hotint_hightlight_notepad.xml (in the folder Hotint_V1\development\autogeneration)



- close NOTEPAD++

If you open a file with the extensions 'txt' or 'hid' with notepad++ there should be 2 new features now:

- highlighting of known keywords
- auto complete (ctrl + space) for known keywords

3.10 How to add your Element to the Script Language

Some parts of HOTINT are autogenerated, see 2.1.9, for more details. These parts are essential for the script language and the documentation!

If you have a special element or constraint and want to use it in script language, follow these steps:

The example is done with the class ANCFBeamShear3DLinear which is derived from ANCF-BeamShear3DGeneric.

author: D. Reischl

3.10.1 Changes in the h-file of your element

1. Make sure, that your element has a constructor in the form like: `ANCFBeamShear3DLinear (MBS* mbs)`
    - you must not use any other arguments in your constructor!
    - It is advised to use a function called `InitConstructor()` or `ElementDefaultConstructor()` to set the default values of your element in all the constructors of your element
2. Add the following code in your h-file in order to tell the edc-converter, that it should process your class: `class ANCFBeamShear3DLinear: public ANCFBeamShear3DGeneric //$EDC$[beginclass, classname=ANCFBeamShear3DLinear, parentclassname=ANCFBeamShear3DGeneric, addelementtype=TAEBody, addelementtypename=ANCFBeamShear3DLinear]`

- if your class is derived from a class, do something very similar with the parent class: `class ANCFBeamShear3DGeneric: public Body3D //$EDC$[beginclass, classname=ANCFBeamShear3DGeneric, parentclassname=Body3D]`
- note the difference, that only in the child class the tags `addelementtype`[2] and `addelementtypename`[3] are used
- `addelementtypename` MUST NOT contain any characters that cause problems in TeX, especially no ”_”

3. Add the documentation of your class here in the h-file. The following tags are available:

- all line breaks in the h-file are ignored
- If you want to force a line break in the docu, use the TeX-Command for line break \\
- You can use TeX code and therefore also formulas in all the tags except of ’texdescription’
- The syntax for all tags is, if nothing else is stated: `tag = "my description"`, e.g. `texdescription = "This is a wonderfull element"`
- If you want to put some part of your docu just in the developer docu and hide it in the user docu, use the TeX command `\developer{...}`. Do not use it with the tag ’texdescription’. It is just possible to use this command inside the quotes of a tag: `tag="\developer{...}"` is possible, but `\developer{tag="..."}` is not.
- `texdescription`: short description of the class. Do not use formulas, references or special TeX commands here!
- `texdescriptionDOF`: degrees of freedom, how many, which ones and in which order
- `texdescriptionNode`: are nodes necessary for the element, which ones, how many d.o.f.s
- `texdescriptionGeometry`: center of gravity, reference and initial configuration, local coordinate system
- `texdescriptionEquations`: especially for constraints it is convenient to write the equation(s)
- `texdescriptionLimitations`: only penalty formulation, just for small rotations, no shear
- `texdescriptionComments`: some additional comments. Only use this tag if it is absolutely necessary!
- `modus`: Are there different modi available? Use the syntax `modus="{name of the mode}{description` e.g. `modus="{element to ground}{Position2.element\_number AND Position2.node\_number have to be equal to 0}"`. You can use as many `modus`-tags as necessary. The modi are converted to a table, so do not use any commands or characters which are not allowed in a table in TeX, especially do not use line breaks \\. You can use \newline if necessary.
- `example`: a short example how to set up this element in script language. Just the filename without the path! The file has to be stored in documentation/ EDCauto_documentation/ docu_examples

---

[2] addelementtypename is the name that is visible to the user

[3] addelementtype is used to group the elements, e.g. 2D/3D, constraints/elements, a.s.o. The available types are listed in ElementsAndModelsLibraryInterface.h

- `figure`: you can use figures in all of the descriptions above. Figures are defined according to the syntax: `figure="filename,caption"` , e.g. `figure="beam3d, This is a nice caption"` You can add more than one figure. It is possible to reference to the figures with `\ref{AddElementTypeNamefigureX}` e.g. `\ref{LinearBeam3Dfigure2}`. Figures have to be stored in documentation/ EDCauto_documentation/ figures.
- `reference`: you can use references to the bibliography in all of the descriptions above. References are defined according to the syntax: `reference="citation"` , e.g. `reference="M. Mustermann, A new approach, 2012"` You can add more than one citation. It is possible to reference to the citations with `\cite{AddElementTypeNamereferenceX}` e.g. `\cite{Mass3Dreference1}`.

4. go to the end of your class in the h-file and add the code:
`//$EDC$[endclass, ANCFBeamShear3DLinear]` to tell the edc-converter where your class ends
    - if your class is derived from a class, do the same for the parent class
    `//$EDC$[endclass,ANCFBeamShear3DGeneric]`
5. Add the description of the variables. This information is used to auto-generate the GetElementDataAuto and SetElementDataAuto of your element. Go to the place in your h-file where the variables of your element are defined.
    (a) you have to use a separate line for each variable:
    `int n1,n2;` this is not possible
    `int n1;` this is correct
    `int n2;` this is correct
    (b) Add the description of the variables.
    `int n1; //$EDC$[varaccess, EDCvarname="nodenumber1", EDCfolder="Geometry", tooltiptext="global number of node 1 (left), node must already exist"]`
    (c) For more details see the file ElementEDCauto.cpp
    (d) If you want to have/change access to variables of (any) parent class, there are 4 ways:
        i. if the variable already has an EDC-Access and the name and tooltip text are correct for your derived element: do nothing! The variable will be available for your element.
        ii. if the variable does not have and EDC-Access yet, but you want to use it, add this code:
        `//EDC Vector q0 //$EDC$[varaccess, EDCvarname="node3initialposition`
        iii. if the variable has an EDC-Access, but you want to remove it, use the flag remove:
        `//EDC double dampingm; //$EDC$[varaccess,remove, EDCvarname= "massprop ..`
        iv. if the variable has an EDC-Access, but you want to change the name or tooltiptext: first add a line to remove the variable, than add a line to define the EDC-Access
6. add the following lines to your h-file:

```
virtual void GetElementDataAuto(ElementDataContainer& edc);
virtual int SetElementDataAuto(ElementDataContainer& edc);
virtual int ReadSingleElementDataAuto(ReadWriteElementDataVariableType& RWdata);
virtual int WriteSingleElementDataAuto(const ReadWriteElementDataVariableType& RWdata);
virtual int GetAvailableSpecialValuesAuto(TArrayDynamic
<ReadWriteElementDataVariableType>& available_variables);
```

7. save your h-file
8. do not compile!

3.10.2 Use the EDC-converter:

1. go to tools/EDC_converter/release
2. add the name of your h-file (and of parent classes) in filelist.txt
    - do not change the order of the first 6 filenames!
    - adjust the integer in this file
    - note, that the order of the elements in the docu corresponds to the order in this list. So please paste your file close to similar elements
3. run EDC_converter.bat - it is located in folder HotInt_V1/tools/EDC_converter/release
4. The converter has now generated the file ElementsLib/ElementEDCauto.cpp

3.10.3 Test your element:

1. compile your project
2. Check if the element is available in the GUI. Press button ”Add Element”
    - Check all the options in the window (are all default values correct, are all the parameters in the correct sub-trees,...)
    - press ”OK”
    - your element should be in the mbs now
3. press ”file/Save As” in the GUI to save your model to a file
4. you can see your element in script language now in this file

3.10.4 error handling

Implement the function CheckConsistency in your element, to avoid crashes

3.10.5 The results

your element is now available

- in the GUI
- in the script language

- in the docu
- for saving

3.11 How to add Sensors/Nodes/Materials/BeamProperties/ to the Script Language

All the steps in section 3.10, have to be done except the fact, that the flag "addelementtype" is not used.

There are only differences for the section 3.11.1: (this example is for adding a node)

3.11.1 Test your element:

1. Check that the function GetTypeName() returns the correct name. This name is used for the user and has to be used in all of the following steps!
   `virtual mystr GetTypeName() return "Node3DS1rot1";;`
2. Go to MBSElementsAndModels/MBSObjectFactory.cpp
   (a) create a new case in MBSObjectFactory::AddNode(int objectTypeId)
   (b) add a new line in MBSObjectFactory::MBSObjectFactory(void):
   `AddObjectInfo(OFCNode, "Node3DS1rot1", 0, "", "");`
   (c) In `AddObjectInfo` you can use flags instead of the 0. Up to now only TAENotInRelease is checked.
   (d) make sure that the order in a) and b) is equal
3. compile your project
4. Check if the element is available in the GUI. Press button "Add Element"
   - Check all the options in the window (are all default values correct, are all the parameters in the correct sub-trees,...)
   - press "OK"
   - your element should be in the mbs now
5. press "file/Save As" in the GUI to save your model to a file
6. you can see your element in script language now in this file

3.12 How to start a HOTINT computation from command line or Matlab

HOTINT can be started from a DOS-Window (or from MATLAB), with additional options-commands. The options are parsed after the HOTINT-config File is read. The order of reading the options is:

- HOTINT-config File
- ModelData-File
- additional options-commands

Each option-text may not include spaces! Instead of “ use \”, otherwise DOS ignores the “. These options will be used in the HOTINT-options if known, or put to the model_data if it is a not known keyword. At the moment, for starting hotint.exe from DOS/Matlab, the current directory MUST be the root directory of hotint.exe

Examples: starting from DOS-window:

-
  ```
  hotint.exe \"SolverOptions.Solution.output_filename=\"dir/my ↩
      ↪ nice file.txt\""
  ```

-
  ```
  hotint.exe GeneralOptions.Application.↩
      ↪ start_computation_automatically=1
  ```

-
  ```
  hotint.exe MyOptions.usemodeNr=2 hotint.exe SolverOptions.↩
      ↪ Solution.output_filename=\"dir/myfile.txt\"
  ```

example using MATLAB:

-
  ```
  dos('hotint.exe GeneralOptions.Application.↩
      ↪ start_computation_automatically=1 SolverOptions.Solution↩
      ↪ .output_filename=\"malatbfile1.txt\" &')
  ```

If the ”&” character is used, it is possible to run multiple computations parallel, for example (using Matlab):

-
  ```
  dos('hotint.exe GeneralOptions.Application.↩
      ↪ start_computation_automatically=1 SolverOptions.Solution↩
      ↪ .output_filename=\"malatbfile1_slowRotation.txt\" Rotor.↩
      ↪ rot_freq=10 &')
  ```

-
  ```
  dos(’hotint.exe GeneralOptions.Application.↩
      ↪ start_computation_automatically=1 SolverOptions.Solution↩
      ↪ .output_filename=\"malatbfile2_fastRotation.txt\" Rotor.↩
      ↪ rot_freq=1000 &’)
  ```

Breaking down a statement into a number of lines can result in a clearer programming style. Because you cannot continue an incomplete string to another line following work-around is recommended:

```
%% starting Hotint from Matlab

%% multiple computations parallel (all options in one line)
dos(’hotint.exe GeneralOptions.Application.↩
   ↪ start_computation_automatically=1 SolverOptions.Solution.↩
   ↪ output_filename=\"matlabfile1_slowRotation.txt\" Rotor.↩
   ↪ rot_freq=10 &’)
dos(’hotint.exe GeneralOptions.Application.↩
   ↪ start_computation_automatically=1 SolverOptions.Solution.↩
   ↪ output_filename=\"matlabfile2_fastRotation.txt\" Rotor.↩
   ↪ rot_freq=1000 &’)

%% multiple computations parallel (options in multiple lines)
command = [ ’dos(’ ’’’hotint.exe ’ ... % do not change this ↩
   ↪ line !!!
            ’GeneralOptions.Application.↩
               ↪ start_computation_automatically=1 ’ ... % make↩
               ↪  sure you end your strings with the blank-↩
               ↪ character!
             ’SolverOptions.Solution.output_filename=\"↩
                ↪ matlabfile1.txt\" ’...
             ’Rotor.rot_freq=10 ’...
             ’&’  ’’’)’] % do not change this line !!!
eval(command)

command = [ ’dos(’ ’’’hotint.exe ’ ...
             ’GeneralOptions.Application.↩
                ↪ start_computation_automatically=1 ’ ...
             ’SolverOptions.Solution.output_filename=\"↩
                ↪ matlabfile2.txt\" ’...
             ’Rotor.rot_freq=1000 ’...
       ’&’  ’’’)’]
eval(command)
```

Bibliography

[1] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM, Philadelphia, 1996.

[2] E. Eich-Soellner, C. Führer, *Numerical Methods in Multibody Dynamics*, Teubner, Stuttgart, 1998.

[3] J. Gerstmayr, M. Stangl, *High-Order Implicit Runge-Kutta Methods for Discontinuous Multibody Systems*, Proceedings of the APM 2004, St. Petersburg, Russia, submitted.

[4] J. Gerstmayr, J. Schöberl, *An Implicit Runge-Kutta Based Solver for 3-Dimensional Multibody Systems*, PAMM, Volume 3(1), 2003, pp. 154-155.

[5] E. Hairer and G. Wanner, *Stiff differential equations solved by Radau methods* or *the RADAU5-code*, available via WWW at `ftp://ftp.unige.ch/pub/doc/math/stiff/radau5.f` (1996)

[6] E. Hairer, (Nørsett) and G. Wanner, *Solving ordinary differential equations I (II)*, Springer Verlag Berlin Heidelberg, 1991.

[7] E. Hairer and Ch. Lubich, and M. Roche, *The numerical solution of differential-algebraic systems by Runge-Kutta methods*, Lecture Notes in Math. 1409, Springer–Verlag, (1989).

[8] A. Shabana *Dynamics of Multibody Systems*, Third Edition, Cambridge University Press, 2005.

[9] R. R. Craig Jr. and M. C. C. Bampton, *Coupling of substructures for dynamic analyses*, AIAA Journal, 6(7), pp. 13131319, 1968

[10] J. Gerstmayr and A. Pechstein, *A generalized component mode synthesis approach for multibody system dynamics leading to constant mass and stiffness matrices*, Proceedings of the ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2011, Washington, DC, USA, 2011. Paper No. DETC2011/MSNDC-47826, submitted.

[11] Masarati, P, *Direct eigenanalysis of constrained system dynamics*, Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi-body Dynamics, 2009.

[12] R. Ludwig and J. Gerstmayr, *Automatic Parameter Identification for Generic Robot Models*, Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi-body Dynamics, 2011.

[13] K. Nachbagauer, P. Gruber, J. Gerstmayr. Structural and Continuum Mechanics Approaches for a 3D Shear Deformable ANCF Beam Finite Element: Application to static and linearized dynamic examples. Journal for Computational and Nonlinear Dynamics, 8, 021004, DOI:10.1115/1.4006787, 2012.

[14] K. Nachbagauer. Development of shear and cross section deformable beam finite elements applied to large deformation and dynamics problems, Johannes Kepler University Linz, 2012.

[15] K. Nachbagauer, P. Gruber, Yu. Vetyukov, J. Gerstmayr. A spatial thin beam finite element based on the absolute nodal coordinate formulation without singularities. Proceedings of the ASME 2011 International Design Engineering Technical Conferences, Computers and Information in Engineering Conference IDETC/CIE 2011, Paper No. DETC2011/MSNDC-47732, Washington, DC, USA, 2011.

[16] P. Gruber, K. Nachbagauer, Yu. Vetyukov, J. Gerstmayr. A novel director-based Bernoulli-Euler beam finite element in absolute nodal coordinate formulation free of geometric singularities. Mechanical Science, 2013 (to appear).