MBDyn Input File Format
Version develop

Pierangelo Masarati

Dipartimento di Ingegneria Aerospaziale
Politecnico di Milano

Automatically generated October 4, 2025

Contents

1 Overview **16**
- 1.1 Execution . . . 16
  - 1.1.1 Passing the Input . . . 16
  - 1.1.2 Passing the Output File Name . . . 16
- 1.2 Input File Structure . . . 17
  - 1.2.1 Syntax Highlighting . . . 18
- 1.3 Output . . . 19

2 General **20**
- 2.1 Types . . . 20
  - 2.1.1 Keywords . . . 20
  - 2.1.2 Strings . . . 20
  - 2.1.3 Numeric Values . . . 21
- 2.2 Variables . . . 22
  - 2.2.1 Namespaces . . . 26
  - 2.2.2 Plugin Variables . . . 29
- 2.3 Higher-Order Math Structures . . . 33
  - 2.3.1 3×1 Vector . . . 33
  - 2.3.2 6×1 Vector . . . 33
  - 2.3.3 3×3 Matrix . . . 33
  - 2.3.4 (3×3) Orientation Matrix . . . 35
  - 2.3.5 6×6 Matrix . . . 38
  - 2.3.6 $6 \times N$ Matrix . . . 38
  - 2.3.7 $N \times 6$ Matrix . . . 39
  - 2.3.8 $N \times N$ Matrix . . . 39
- 2.4 Input Related Cards . . . 39
  - 2.4.1 Constitutive Law . . . 39
  - 2.4.2 C81 Data . . . 40
  - 2.4.3 Drive Caller . . . 42
  - 2.4.4 Hydraulic fluid . . . 42
  - 2.4.5 Include . . . 43
  - 2.4.6 Module Load . . . 43
  - 2.4.7 Print symbol table . . . 44
  - 2.4.8 Reference . . . 44
  - 2.4.9 Remark . . . 48
  - 2.4.10 Set . . . 49
  - 2.4.11 Setenv . . . 49
  - 2.4.12 Template Drive Caller . . . 50

2.5 Node Degrees of Freedom . . . 50
2.6 Drives and Drive Callers . . . 51
2.6.1 Array drive . . . 51
2.6.2 Bistop drive . . . 53
2.6.3 Closest next drive . . . 53
2.6.4 Const(ant) drive . . . 53
2.6.5 Cosine drive . . . 53
2.6.6 Cubic drive . . . 54
2.6.7 Direct drive . . . 55
2.6.8 Discrete filter drive . . . 55
2.6.9 Dof drive . . . 56
2.6.10 Double ramp drive . . . 56
2.6.11 Double step drive . . . 57
2.6.12 Drive drive . . . 57
2.6.13 Element drive . . . 58
2.6.14 Exponential drive . . . 59
2.6.15 File drive . . . 59
2.6.16 Fourier series drive . . . 59
2.6.17 Frequency sweep drive . . . 60
2.6.18 GiNaC . . . 61
2.6.19 Linear drive . . . 61
2.6.20 Meter drive . . . 61
2.6.21 Mult drive . . . 61
2.6.22 Node drive . . . 62
2.6.23 Null drive . . . 62
2.6.24 Parabolic drive . . . 62
2.6.25 Periodic drive . . . 62
2.6.26 Piecewise linear drive . . . 63
2.6.27 Postponed drive . . . 63
2.6.28 Ramp drive . . . 63
2.6.29 Random drive . . . 64
2.6.30 Sample and hold drive . . . 64
2.6.31 Scalar function drive . . . 64
2.6.32 Sine drive . . . 65
2.6.33 Step drive . . . 66
2.6.34 Step5 drive . . . 66
2.6.35 String drive . . . 67
2.6.36 Tanh drive . . . 68
2.6.37 Time drive . . . 68
2.6.38 Timestep drive . . . 68
2.6.39 Unit drive . . . 68
2.6.40 Deprecated drive callers . . . 68
2.6.41 Hints . . . 69
2.6.42 Template Drive . . . 69
2.7 Scalar functions . . . 72
2.8 Friction . . . 78
2.8.1 1D Friction models . . . 78
2.8.2 2D Friction models . . . 79
2.8.3 Shape functions . . . 80

2.9 Shapes . . . 81
2.10 Constitutive Laws . . . 83
2.10.1 Linear elastic, linear elastic isotropic . . . 83
2.10.2 Linear elastic generic . . . 85
2.10.3 Linear elastic generic axial torsion coupling . . . 85
2.10.4 Cubic elastic generic . . . 85
2.10.5 Inverse square elastic . . . 86
2.10.6 Log elastic . . . 86
2.10.7 Linear elastic bistop . . . 86
2.10.8 Double linear elastic . . . 86
2.10.9 Isotropic hardening elastic . . . 87
2.10.10 Scalar function elastic, scalar function elastic isotropic . . . 87
2.10.11 Scalar function elastic orthotropic . . . 87
2.10.12 Linear viscous, linear viscous isotropic . . . 88
2.10.13 Linear viscous generic . . . 88
2.10.14 Linear viscoelastic, linear viscoelastic isotropic . . . 88
2.10.15 Linear viscoelastic generic . . . 88
2.10.16 Linear time variant viscoelastic generic . . . 89
2.10.17 Linear viscoelastic generic axial torsion coupling . . . 89
2.10.18 Cubic viscoelastic generic . . . 89
2.10.19 Double linear viscoelastic . . . 90
2.10.20 Turbulent viscoelastic . . . 90
2.10.21 Linear viscoelastic bistop . . . 90
2.10.22 GRAALL damper . . . 91
2.10.23 shock absorber . . . 91
2.10.24 symbolic elastic . . . 92
2.10.25 symbolic viscous . . . 92
2.10.26 symbolic viscoelastic . . . 93
2.10.27 ann elastic . . . 93
2.10.28 ann viscoelastic . . . 93
2.10.29 nlsf viscoelastic . . . 94
2.10.30 nlp viscoelastic . . . 96
2.10.31 Hookean linear elastic isotropic . . . 97
2.10.32 Neo-Hookean . . . 98
2.10.33 Mooney-Rivlin . . . 99
2.10.34 Bilinear isotropic hardening . . . 101
2.10.35 Linear viscoelastic Maxwell . . . 101
2.10.36 MFront code generator . . . 103
2.10.37 mfront small strain/mfront finite strain . . . 103
2.10.38 array . . . 105
2.10.39 axial wrapper . . . 105
2.10.40 bistop . . . 105
2.10.41 drive caller wrapper . . . 106
2.10.42 invariant angular . . . 106
2.11 Hydraulic fluid . . . 107
2.11.1 Incompressible . . . 107
2.11.2 Linearly compressible . . . 107
2.11.3 Linearly compressible, with thermal dependency . . . 108
2.11.4 Super (linearly compressible, with thermal dependency) . . . 108

2.11.5 Exponential compressible fluid, with saturation . . . 108
2.12 Authentication Methods . . . 109
2.12.1 Note on security and confidentiality . . . 109
2.12.2 No authentication . . . 109
2.12.3 Password . . . 109
2.12.4 PAM (Pluggable Authentication Modules) . . . 110
2.12.5 SASL (Simple Authentication and Security Layer) . . . 110
2.13 Miscellaneous . . . 110

3 Data **114**
3.1 Problem . . . 114

4 Problems **115**
4.1 Initial-Value Problem . . . 115
4.1.1 General Data . . . 115
4.1.2 Derivatives Data . . . 139
4.1.3 Dummy Steps Data . . . 140
4.1.4 Output Data . . . 140
4.1.5 Real-Time Execution . . . 141
4.2 Other Problems . . . 143

5 Control Data **144**
5.1 Enable support for automatic differentiation . . . 144
5.2 Assembly-Related Cards . . . 144
5.2.1 Skip Initial Joint Assembly . . . 144
5.2.2 Initial Assembly of Deformable and Force Elements . . . 146
5.2.3 Use . . . 146
5.2.4 Initial Stiffness . . . 146
5.2.5 Omega Rotates . . . 146
5.2.6 Tolerance . . . 146
5.2.7 Max Iterations . . . 147
5.3 General-Purpose Cards . . . 147
5.3.1 Title . . . 147
5.3.2 Print . . . 147
5.3.3 Make Restart File . . . 148
5.3.4 Load restart file . . . 148
5.3.5 Select Timeout . . . 148
5.3.6 Default Output . . . 148
5.3.7 Output units . . . 150
5.3.8 Output Frequency . . . 150
5.3.9 Output Meter . . . 150
5.3.10 Output Precision . . . 151
5.3.11 Output Results . . . 151
5.3.12 Default Orientation . . . 151
5.3.13 Default Aerodynamic Output . . . 152
5.3.14 Default Beam Output . . . 153
5.3.15 Default Scale . . . 153
5.3.16 Finite Difference Jacobian Meter . . . 154
5.3.17 Model . . . 155
5.3.18 Rigid Body Kinematics . . . 155

5.3.19 Loadable path . . . 155
5.4 Model Counter Cards . . . 156
5.4.1 Nodes . . . 156
5.4.2 Drivers . . . 156
5.4.3 Elements . . . 156

6 Nodes **158**
6.1 Abstract Node . . . 159
6.2 Electric Node . . . 159
6.3 Hydraulic Node . . . 159
6.4 Parameter Node . . . 160
6.5 Structural Node . . . 160
6.5.1 Static Node . . . 161
6.5.2 Dynamic Node . . . 161
6.5.3 Modal Node . . . 161
6.5.4 Syntax . . . 161
6.5.5 Dummy Node . . . 162
6.6 Thermal Node . . . 167
6.7 Miscellaneous . . . 167
6.7.1 Output . . . 167

7 Drivers **168**
7.1 File Drivers . . . 168
7.1.1 Fixed Step . . . 168
7.1.2 Variable Step . . . 170
7.1.3 Socket . . . 170
7.1.4 RTAI Mailbox . . . 171
7.1.5 Stream . . . 171

8 Elements **175**
8.1 Aerodynamic Elements . . . 176
8.1.1 Aerodynamic Body/Aerodynamic Beam2/3 . . . 176
8.1.2 Aeromodal Element . . . 181
8.1.3 Aircraft Instruments . . . 182
8.1.4 Air Properties . . . 185
8.1.5 Generic Aerodynamic Force . . . 189
8.1.6 Induced velocity . . . 193
8.1.7 Rotor . . . 196
8.2 Automatic structural . . . 196
8.3 Beam Elements . . . 197
8.3.1 Beam Section Constitutive Law . . . 198
8.3.2 Three-node beam element . . . 208
8.3.3 Two-node beam element . . . 214
8.3.4 Output . . . 216
8.3.5 Notes . . . 217
8.4 Body . . . 217
8.5 Bulk Elements . . . 219
8.5.1 Stiffness spring . . . 219
8.6 Couple . . . 220
8.7 Electric Elements . . . 220

8.7.1 Accelerometer . . . 220
8.7.2 Discrete control . . . 221
8.7.3 Displacement . . . 225
8.7.4 Motor . . . 225
8.8 Force . . . 226
8.8.1 Output . . . 227
8.8.2 Abstract force . . . 227
8.8.3 Abstract reaction force . . . 227
8.8.4 Structural force . . . 228
8.8.5 Structural internal force . . . 230
8.8.6 Structural couple . . . 231
8.8.7 Structural internal couple . . . 232
8.8.8 Modal . . . 233
8.8.9 External forces . . . 234
8.8.10 External Structural . . . 237
8.8.11 External structural mapping . . . 239
8.8.12 External modal . . . 243
8.8.13 External modal mapping . . . 246
8.9 Genel Element . . . 249
8.9.1 General Purpose Elements . . . 249
8.9.2 Special Rotorcraft GENEL Elements . . . 255
8.10 Gravity Element . . . 257
8.11 Hydraulic Element . . . 257
8.11.1 Accumulator . . . 258
8.11.2 Actuator . . . 259
8.11.3 Control Valve . . . 260
8.11.4 Dynamic Control Valve . . . 260
8.11.5 Dynamic Pipe . . . 261
8.11.6 Flow Valve . . . 261
8.11.7 Imposed Flow . . . 262
8.11.8 Imposed Pressure . . . 262
8.11.9 Minor Loss . . . 262
8.11.10 Orifice . . . 263
8.11.11 Pipe . . . 263
8.11.12 Pressure Flow Control Valve . . . 264
8.11.13 Pressure Valve . . . 264
8.11.14 Tank . . . 264
8.11.15 Three Way Minor Loss . . . 264
8.12 Joint Element . . . 265
8.12.1 Angular acceleration . . . 265
8.12.2 Angular velocity . . . 266
8.12.3 Axial rotation . . . 266
8.12.4 Beam slider . . . 267
8.12.5 Brake . . . 269
8.12.6 Cardano hinge . . . 269
8.12.7 Cardano pin . . . 270
8.12.8 Cardano rotation . . . 270
8.12.9 Clamp . . . 271
8.12.10 Coincidence . . . 271

8.12.11 Deformable Axial Joint . . . 272
8.12.12 Deformable displacement hinge . . . 273
8.12.13 Deformable displacement joint . . . 273
8.12.14 Deformable Hinge . . . 274
8.12.15 Deformable joint . . . 276
8.12.16 Distance . . . 278
8.12.17 Distance with offset . . . 279
8.12.18 Drive displacement . . . 279
8.12.19 Drive displacement pin . . . 280
8.12.20 Drive hinge . . . 281
8.12.21 Gimbal hinge . . . 282
8.12.22 Gimbal rotation . . . 282
8.12.23 Imposed displacement . . . 283
8.12.24 Imposed displacement pin . . . 283
8.12.25 In line . . . 284
8.12.26 In plane . . . 285
8.12.27 Invariant deformable displacement joint . . . 285
8.12.28 Invariant deformable hinge . . . 285
8.12.29 Linear acceleration . . . 286
8.12.30 Linear velocity . . . 286
8.12.31 Modal . . . 286
8.12.32 Offset displacement joint . . . 291
8.12.33 Plane displacement . . . 292
8.12.34 Plane displacement pin . . . 293
8.12.35 Plane hinge . . . 293
8.12.36 Plane pin . . . 293
8.12.37 Prismatic . . . 293
8.12.38 Revolute hinge . . . 294
8.12.39 Revolute pin . . . 296
8.12.40 Revolute rotation . . . 297
8.12.41 Rigid body displacement joint . . . 298
8.12.42 Rod . . . 300
8.12.43 Rod with offset . . . 303
8.12.44 Screw joint . . . 303
8.12.45 Spherical hinge . . . 304
8.12.46 Spherical pin . . . 305
8.12.47 Total equation . . . 306
8.12.48 Total joint . . . 306
8.12.49 Total pin joint . . . 309
8.12.50 Total reaction . . . 313
8.12.51 Universal hinge . . . 313
8.12.52 Universal pin . . . 313
8.12.53 Universal rotation . . . 314
8.12.54 Viscous body . . . 314
8.12.55 Lower Pairs . . . 314
8.13 Joint Regularization . . . 314
8.13.1 Tikhonov . . . 315
8.14 Output Elements . . . 315
8.14.1 Stream output . . . 315

8.14.2 RTAI output . . . 319
8.14.3 Stream motion output . . . 319
8.15 Plate Elements . . . 319
8.15.1 Shell4 . . . 319
8.15.2 Membrane4 . . . 322
8.16 Solid Elements . . . 323
8.16.1 Scope . . . 323
8.16.2 Element input format . . . 323
8.16.3 Constitutive law types to be used for solid elements . . . 324
8.16.4 Displacement based elements versus displacement/pressure elements (u/p-c) . . . 325
8.16.5 Dynamic versus static elements . . . 325
8.16.6 Gravity and rigid body kinematics . . . 325
8.16.7 Material properties and constitutive laws . . . 325
8.16.8 Output . . . 327
8.16.9 Example: . . . 328
8.16.10 Pre- and post-processing . . . 329
8.17 Surface Loads . . . 333
8.17.1 Pressure Loads . . . 333
8.17.2 Surface Traction's . . . 334
8.18 Thermal Elements . . . 337
8.18.1 Capacitance . . . 337
8.18.2 Resistance . . . 337
8.18.3 Source . . . 338
8.19 User-Defined Elements . . . 338
8.19.1 Loadable Element . . . 338
8.19.2 User-Defined Element . . . 339
8.19.3 General Discussion on Run-Time Loadable Modules . . . 340
8.20 Miscellaneous . . . 341
8.20.1 Bind . . . 341
8.20.2 Driven Element . . . 343
8.20.3 Inertia . . . 345
8.20.4 Output . . . 348

A Modal Element Data **349**
A.1 FEM File Format . . . 349
A.1.1 Usage . . . 354
A.1.2 Example: Dynamic Model . . . 354
A.1.3 Example: Static Model . . . 356
A.2 Code Aster Procedure . . . 357
A.3 NASTRAN Procedure . . . 359
A.4 Procedure for mboct-fem-pkg . . . 360
A.5 Procedures for Other FEA Software . . . 361

B Modules **362**
B.1 Element Modules . . . 362
B.1.1 Module-aerodyn . . . 362
B.1.2 Module-asynchronous_machine . . . 362
B.1.3 Module-cyclocopter . . . 363
B.1.4 Module-fab-electric . . . 364
B.1.5 Module-fab-motion . . . 367

B.1.6 Module-fab-sbearings . . . 367
B.1.7 Module-hfelem . . . 368
B.1.8 Module-hydrodynamic_plain_bearing_with_offset . . . 374
B.1.9 Module-hydrodynamic plain bearing2 . . . 375
B.1.10 Module-imu . . . 382
B.1.11 Module-mds . . . 383
B.1.12 Module-MFtire . . . 383
B.1.13 Module-nonsmooth-node . . . 384
B.1.14 Module-template2 . . . 385
B.1.15 Module-wheel2 . . . 385
B.1.16 Module-wheel4 . . . 385
B.1.17 Module-MFtire . . . 390
B.2 Constitutive Law Modules . . . 390
B.2.1 Module-constlaw . . . 390
B.2.2 Module-constlaw-f90 . . . 390
B.2.3 Module-constlaw-f95 . . . 391
B.2.4 Module-cont-contact . . . 391
B.2.5 Module-damper-graall . . . 392
B.2.6 Module-damper-hydraulic . . . 392
B.2.7 Module-muscles . . . 393
B.3 Drive Caller Modules . . . 396
B.3.1 Module-drive . . . 396
B.3.2 Module-randdrive . . . 396
B.4 Template Drive Caller Modules . . . 396
B.4.1 Module-eu2phi . . . 396
B.5 Driver Modules . . . 397
B.5.1 Module-HID . . . 397
B.6 Scalar Function Modules . . . 397
B.6.1 Module-scalarfunc . . . 397
B.7 Miscellaneous Modules . . . 398
B.7.1 Module-chrono-interface . . . 398
B.7.2 Module-flightgear . . . 400
B.7.3 Module-octave . . . 401
B.7.4 Module-tclpgin . . . 404
B.7.5 Module-template . . . 405
B.7.6 Module-udunits . . . 405

C NetCDF Output Format **406**
C.1 NetCDF Output . . . 406
C.1.1 Base Level . . . 407
C.1.2 Run Level . . . 407
C.1.3 Node Level . . . 407
C.1.4 Element Level . . . 409
C.1.5 Eigenanalysis . . . 419
C.2 Accessing the Database . . . 422
C.2.1 Octave . . . 423

D Results Visualization **424**
D.1 EasyAnim . . . 424
D.2 Output Manipulation . . . 425
D.2.1 Output in a Relative Reference Frame . . . 425

E Log File Format **427**
E.1 Generic Format . . . 427
E.2 Model Description Entries . . . 427
E.2.1 acceleration, velocity . . . 427
E.2.2 aero0 . . . 427
E.2.3 aero2 . . . 428
E.2.4 aero3 . . . 428
E.2.5 axial rotation . . . 428
E.2.6 beam2 . . . 428
E.2.7 beam3 . . . 429
E.2.8 body . . . 429
E.2.9 beam slider . . . 429
E.2.10 brake . . . 429
E.2.11 clamp . . . 430
E.2.12 deformable joints . . . 430
E.2.13 distance . . . 430
E.2.14 drive displacement . . . 430
E.2.15 drive displacement pin . . . 431
E.2.16 gimbal rotation . . . 431
E.2.17 inline . . . 431
E.2.18 inplane . . . 431
E.2.19 prismatic . . . 432
E.2.20 relative frame structural node . . . 432
E.2.21 revolute hinge . . . 432
E.2.22 revolute rotation . . . 432
E.2.23 rod . . . 432
E.2.24 spherical hinge . . . 433
E.2.25 spherical pin . . . 433
E.2.26 structural forces . . . 433
E.2.27 structural node . . . 433
E.2.28 total joint . . . 434
E.2.29 total pin joint . . . 434
E.2.30 universal hinge . . . 435
E.2.31 universal pin . . . 435
E.2.32 universal rotation . . . 435
E.3 Analysis Description Entries . . . 435
E.3.1 inertia . . . 435
E.3.2 output frequency . . . 436
E.3.3 struct node dofs . . . 436
E.3.4 struct node momentum dofs . . . 436
E.3.5 struct node labels . . . 436
E.4 Output Elements . . . 436
E.4.1 RTAI stream output . . . 436
E.4.2 INET socket stream output . . . 437
E.4.3 local socket stream output . . . 437

E.4.4 output element content 438
E.5 File drivers 438
E.5.1 INET socket stream file driver 438

F Frequently Asked Questions **439**
F.1 Input 439
F.1.1 What is the exact syntax of element X? 439
F.1.2 What element should I use to model Y? 439
F.2 Output 440
F.2.1 How can I reduce the amount of output? 440
F.3 Execution Debugging 441
F.3.1 How can I find out why the iterative solution of a nonlinear problem does not converge? 441

List of Figures

2.1 Predefined constants in math parser . . . 25
2.2 Construction of 3×3 orientation matrix from two non-parallel vectors. . . . 35
2.3 3D screw thread sketch. . . . 81

8.1 Airfoil geometry . . . 179
8.2 Constitutive coefficients grouping (S: shear, A: axial) . . . 199
8.3 Beam section . . . 205
8.4 Geometry of the three-node beam element. . . . 208
8.5 Geometry of the two-node beam element. . . . 214
8.6 NASTRAN `CONM2` card . . . 218
8.7 Discrete control layout. . . . 222
8.8 Cardano hinge. . . . 270
8.9 Inline joint. . . . 285
8.10 Revolute hinge. . . . 295
8.11 Spherical hinge. . . . 305
8.12 Shell: definitions . . . 320
8.13 Node order . . . 331
8.14 Node order: hexahedron27 . . . 332
8.15 Cook's membrane: deformed shape and VON MISES stress rendered using Gmsh . . . 332
8.16 Node order: surface loads . . . 336

List of Tables

2.1 Built-in types in math parser . . . 21
2.2 Predefined variables and constants in math parser . . . 23
2.3 Built-in mathematical operators in math parser (from higher to lower precedence) . . . 26
2.4 Built-in mathematical functions in math parser . . . 27
2.5 Built-in cast and evaluation functions in math parser . . . 28
2.6 Functions in the `model` namespace . . . 30
2.7 Functions in the `model` namespace (contd.) . . . 31
2.8 Drive callers . . . 52
2.9 Constitutive laws dimensionality . . . 84

4.1 Linear solvers properties . . . 138
4.2 Linear solvers memory usage . . . 138
4.3 Linear solvers pivot handling . . . 138

5.1 Elements with support for automatic differentiation (AD) . . . 145
5.2 Predefined units systems . . . 150

8.1 Constitutive laws to be used for each solid element type . . . 324
8.2 Finite Element Types using a pure displacement based formulation . . . 327
8.3 Finite Element Types using a displacement/pressure formulation . . . 328
8.4 Finite Element Types using the deformation gradient and the First Piola-Kirchhoff stress tensor . . . 328
8.5 Finite Element Types for surface loads . . . 335

B.1 Constants hard-coded in the `erf` muscle model $f_1(x)$ function. . . . 394

Introduction

This document describes the format of the input data for MBDyn, the Multibody Dynamics analysis suite. It should be considered a syntax more than a format, since the rules of the input allow a lot of freedom in the file format.

As the title states, this manual describes the syntax of the input. The reader should not expect to *learn* how to model systems from this text; it is rather a reference manual where the user can find the exact and detailed description of the syntax of some entity, where it is assumed that the user already knows that entity exactly does what is required to model that system or solve that problem.

To get on the right track, one may have a look at the *Frequently Asked Questions* (Appendix F), a steadily growing chapter that collects the most common issues arising from users. They might help pointing to the right statement for a given purpose, but in any case nothing can replace the judgment of the user about what entity exactly fits some need. The only document that may help users in growing this type of judgment is the *tutorials* book, available from the website. For more specific questions, one should try the

`mbdyn-users@mbdyn.org`

mailing list (browse the archives, just in case, then post a question; posting is restricted to subscribers; subscription is free).

Conventions on the Notation

Throughout the manual, a (sometimes naïve) Backus-Naur-like syntax description is used. Extensions are made, such as to put optional arguments in square brackets '`[]`', mutually exclusive arguments in curly brackets '`{}`', separated by the operator '`|`' (the logical "OR"). Non-terminal symbols are enclosed in angle brackets '`<>`', while terminal symbols are written in normal types. The association of a non-terminal with terminal or non-terminal symbols is performed by means of the operator '`::=`'. When required, the type of a non-terminal symbol is enforced in a "C" programming language casting style, by prepending a (usually self-explanatory) type enclosed in plain brackets '`()`'.

Remarks

The input of the program MBDyn is subject to continuous changes at a fast pace, since the code is under development. This documentation attempts to give some insight into the logic that drove its implementation.

While most of the software resulted from a careful design, some portions are "as they are" only because they were made in a hurry, or because no better way was at hand at the moment they were made. The input format partially reflects the development of the software. Whenever changes in the input format and in the syntax of existing parts are required, an effort will be attempted to make it as much backward

compatible as possible, or at least reliable diagnostics and error checking will be put in place that warns for possible erroneous inputs related to a change in the input format.

At present the input of MBDyn already warns for many possible errors and makes as many cross-checks as possible on the consistency of the data. Nearly every warning and error issue shows the line of the input file where the error occurred.

This document will be subjected to changes, so be sure you always have a release that is aligned with the version of the code you're running.

For any question or problem, to fix typos, bugs, for comments and suggestions, please contact the users' mailing list

```
mbdyn-users@mbdyn.org
```

Please note that you need to subscribe to be allowed to post to the list. Posting is public, so please only post information not subjected to distribution restrictions.

Mailing list subscription information is available on the web site

```
https://www.mbdyn.org/Mailing.html
```

As an alternative, you can directly contact the Developers' Team:

Pierangelo Masarati,
MBDyn Development Team
Dipartimento di Ingegneria Aerospaziale
Politecnico di Milano
via La Masa 34, 20156 Milano, Italy
Fax: +39 02 2399 8334
E-mail: mbdyn@aero.polimi.it
Web: https://www.mbdyn.org/

Make sure you use the manual that refers to the version of MBDyn that you are using; from MBDyn 1.2 on, the input manual is marked with the same version number of the package.

The website `http://www.mbdyn.org/` may advertise different versions of the manual, including those related to past releases and a HEAD or -devel version, which refers to current development; this can be useful to check whether the current development may address problems you are having. Moreover, since the updating of the manual is not as prompt as required to keep pace with code releases, the -devel manual may describe features that are already in the latest release but not yet documented in that release's manual.

Chapter 1

Overview

This chapter gives a global overview of the structure of the input file.

1.1 Execution

MBDyn is a command-line tool. This means that in order to execute it, one needs to start the executable from a terminal, and, as a minimum, pass it some indications about the problem it must analyze.

Usually, this is done by preparing one or more input files that describe the model and the analysis that needs to be performed.

1.1.1 Passing the Input

The input file can be passed either using a specific switch (`-f`, preferred), or by listing multiple input file names on the command line. In the latter case, the analyses are executed in sequence. The format of the input file is the topic of most of this manual. If no input file is defined, MBDyn as a last resort tries to read from the standard input. As a consequence, the following commands are equivalent:

```
# assuming file "input" exists in the current working directory and is readable...
$ mbdyn -f input
$ mbdyn input
$ mbdyn < input
```

They are listed in order of preference.

If the input file name consists of a path, either absolute or relative, the file can reside anywhere in the file system. In this case, MBDyn changes its current working directory (see `chdir(2)` for details) to the directory where the input file resides. As a consequence, the path of any included file is seen as relative to the directory where the file containing the `include` directive is located (see Section 2.4.5 for details).

1.1.2 Passing the Output File Name

The output is stored in a family of files whose common name is passed through another switch (`-o`). Those files are formed by adding an extension specific to their contents to the output file name. If none is defined, the output file name defaults to `MBDyn`.

If the argument of the `-o` switch consists of a path, either absolute or relative, and a file name, the full path up to the file name excluded must exist and be write-accessible. The output files are stored in

the corresponding directory. Otherwise they are stored in the current working directory of the process at the time of its execution.

If the argument of the `-o` switch consists of exactly a directory, either absolute or relative, without any file name, the name of the input file is used, but the output files are placed in the directory indicated in the argument of the `-o` switch.

For example:

```
$ mbdyn -f input                    # output in "input.<extension>"
$ mbdyn -f input -o output          # output in "output.<extension>"
$ mbdyn -f input -o folder/output   # output in "folder/output.<extension>"
$ mbdyn -f input -o folder/         # output in "folder/input.<extension>"
$ mbdyn < input                     # output in "MBDyn.<extension>"
```

1.2 Input File Structure

The input is divided in blocks. This is a consequence of the fact that almost every module of the code needs data and each module is responsible for its data input. So it is natural to make each module read and interpret its data starting from the input phase.

Every statement (or 'card') follows the syntax:

```
<card> ::= <description> [ : <arglist> ] ;
```

`arglist` is a(n optional) list of arguments, that is driven by the `description` that identifies the card. The keyword can be made of multiple words separated by spaces or tabs; the extra spaces[1] are skipped, and the match is case insensitive. The arguments are usually separated by commas[2]. A semicolon ends the card.

Many cards allow extra arguments that may assume default values if not supplied by the user. Usually these arguments are at the end of the card and must follow some rules. A check for the existence of extra args is made, and they are usually read in a fixed sequence. Optional args are usually prefixed by a keyword.

When structured arguments like matrices, vectors, or drive callers are expected, they are parsed by dedicated functions. Anyway, the structured data always follows the rules of the general data. Few exceptions are present, and will be eliminated soon. Every data block is surrounded by the control statements `begin` and `end`:

```
begin : <block_name> ;
    # block content
end : <block_name> ;
```

The block sequence is:

```
begin : data ;
    # select a problem
    problem : <problem_name> ;
end : data ;

begin : <problem_name> ;
    # problem-specific data
```

[1] Anything that makes the C function `isspace()` return `true`

[2] A few exceptions require a colon to separate blocks of arguments; wherever it is feasible, those exceptions will be eliminated in future versions. Those cards will be marked as deprecated.

```
end : <problem_name> ;

begin : control data ;
    # model control data
end : control data ;

begin : nodes ;
    # nodes data
end : nodes ;

# note: optional, if requested in control data
begin : drivers ;
    # special drivers data
end : drivers ;

begin : elements ;
    # elements data
end : elements ;
```

The drivers block is optional.

The Schur solver allows an extra block after that of the elements:

```
# note: optional; only allowed when Schur solver is defined in problem_name
begin : parallel ;
    # parallel data
end : parallel ;
```

where data specific to the partitioning and the connectivity of the Schur solver is provided. This is yet undocumented, and will likely be described in a future chapter.

Chapter 2 describes the basic and aggregate data structures that concur in building up the cards. Chapter 3 describes problem selection options. Chapter 4 describes the cards dedicated to the specification of the parameters of the simulation and of the selected integration scheme. Chapter 5 describes the model control data. Chapter 6 describes the cards related to the input of nodes. Nodes come first because they are the elementary blocks of the connectivity graph of the model, so historically they had to be defined before any element could be introduced. Chapter 7 describes the cards related to special drivers. Chapter 8 describes the cards related to the input of elements.

1.2.1 Syntax Highlighting

To ease the creation of MBDyn file, several syntax highlighting plugins, for different text editors, can be used:

- **Vim** (`http://www.vim.org/`) and its GUI variants: syntax highlighting for MBDyn can be activated by adding the `<mbdyn-directory>/var/mbd.vim` file to the `~/.vim/syntax/` folder;
- **Geany** (`http://www.geany.org/`) : the way to add MBDyn syntax highlighting is given in Geany's wiki at `http://wiki.geany.org/config/mbdyn`.
- **Kate and KWrite** (`https://kate-editor.org/`): instructions to add MBDyn syntax support can be found here: `https://github.com/pietrodantuono/MBDynSyntaxHighlightingForKate`;
- **Atom** (`https://atom.io/`): the **language-mbdyn** package can be installed directly from the Atom `Preferences->Install` panel: it provides both MBDyn syntax highlighting and autocompletion snippets.

1.3 Output

The program outputs data to a set of files for each simulation. The contents of each file is related to the file extension, which is appended to the input file if no output file name is explicitly supplied.

If no input file is explicitly provided as well, and thus input is directly read from `stdin`, the output file name defaults to 'MBDyn'. Otherwise, unless the output file name is explicitly set, the name of the input file is used.

The contents of the output files are described within the description of the items (nodes or elements) that generate them. Only a general information file, with extension `.out`, is described here. The file contains general information about the simulation; it is not formatted.

The file contains occasional informational messages, prefixed by a hash mark ('#'). These messages should be intended as comments about the current status of the simulation. At some point, after initialization completes, the comment

```
# Step Time TStep NIter ResErr SolErr SolConv
```

appears, which illustrates the contents of the lines that will be written for each time step. The fields indicate:

1. `Step`: the time step number;
2. `Time`: the time at that step;
3. `TStep`: the time step at that step ($\texttt{Time}_k$ - $\texttt{Time}_{k-1}$);
4. `NIter`: the number of iterations required to converge;
5. `ResErr`: the error on the residual after convergence (0 if not computed);
6. `SolErr`: the error on the solution after convergence (0 if not computed);
7. `SolConv`: a boolean that indicates whether convergence was determined by the error criterion on the residual or on the solution (0 for residual, 1 for solution).

There is also a supplementary file, with `.log` extension, that may contain extra (logging) information. Its content, although very experimental and subjected to changes, is documented in Appendix E.

Experimental support for output using the NetCDF database format is available for selected items. See Appendix C for details.

Chapter 2

General

This chapter describes how data structures are read and how they participate, as building blocks, to the definition of specific cards. Consistency across the software and the input file has been a driving principle in designing the input of MBDyn. As such, the very same elementary data structures are present in very different contexts.

2.1 Types

2.1.1 Keywords

In MBDyn's input a lot of so-called "keywords" appear literally in the input file. They are case-insensitive and may have an arbitrary amount of space in between. For example, '`control data`', '`Control Data`', and '`controldata`' are equivalent.

However, most of the keywords are context-dependent, so they are not illustrated altogether in a dedicated section, but rather presented in relation to the contexts they may appear in.

2.1.2 Strings

Literal strings are not used very much in MBDyn. However, they may be used in quite a few significant places, for example to indicate file names, so few details on their syntax are provided.

Strings are typically delimited by double quotes ('`"`'). When a string is expected, the parser looks for the opening double quotes and eats up all the white space before it. Then all characters are read as they are until the closing double quotes are encountered. The escape character is the backslash ('\'); it is used:

- to escape the escape character itself ('\');
- to escape any double quotes that are part of the string, and would otherwise be interpreted as the closure of the string;
- to break a string on multiple lines by placing it before the newline character ('\`n`'); in this case, the escape character and the newline are eaten up;
- to allow the use of non-printing characters, represented in the form '\`<hexpair>`', so that the hexadecimal representation of the non-printing char is converted into its integer equivalent in the 0–255 range.

Character sets different from ASCII are not supported.

Table 2.1: Built-in types in math parser

Name	Description
bool	Boolean number (promoted to integer, real, or string (0 or 1), whenever required)
integer	Integer number (promoted to real, or string, whenever required)
real	Real number (promoted to string whenever required)
string	Text string

2.1.3 Numeric Values

Every time a numeric value is expected, the result of evaluating a mathematical expression can be used, including variable declaration and assignment (variable names and values are kept in memory throughout the input phase and the simulation) and simple math functions. Limited math on strings is also supported. Named variables and non-named constants are strongly typed.

Four types are currently available: bool, integer, real, and string. They are listed in Table 2.1.

Bool

A bool can take the values 0 or 1. In the symbol table, the constants TRUE (= 1) and FALSE (= 0) are predefined. Casting from integer and real to bool is supported; zero casts to (bool)0 and non-zero casts to (bool)1.

Integer

An integer can take any integer value between the predefined constants INT_MIN and INT_MAX (which are machine-dependent; the C built-in type int is used). Integers are signed. Please note that any sequence of digit not terminated by a dot ('.') is interpreted as an integer. Thus, very large numbers, intended as real, might overflow. Use the exponential notation, instead, or terminate real-valued numbers with a dot.

Examples:

```
0
1
1234
-12
1000000000000000 # this likely overflows!
```

Real

A real can take any real value between the predefined constants REAL_MIN and REAL_MAX (which are machine-dependent; the C built-in type double is used). Any number that contains a dot ('.') is treated as real.

Examples:

```
0.
1.
1.234
-1.2
```

```
1000000000000000. # same as 1e15
1e15 # same as 1000000000000000.
```

String

A `string` is an arbitrary sequence of characters. Currently, only basic chars (i.e. 0 to 127) are guaranteed to work. Strings are enclosed in double quotes (‘"’).

Examples:

```
"0"
"1"
"hello"
```

Label

Labels are not actually types. They are treated separately because they play an important role in identifying several simulation entities. Currently, only unsigned `integer` values can be used as labels. Please note that unsigned integers are not treated specially by the mathematical parser, but negative integers will be rejected when used as labels. In the future, `string` values might be used as labels.

2.2 Variables

Variable declaration/definition:

```
<declaration> ::= [<declaration_modifier>] [<type_modifier>] <type> <name>
<definition> ::= [<declaration_modifier>] [<type_modifier>] <type> <name> = <value>

<declaration_modifier> ::= ifndef
<type_modifier> ::= const
<type> ::= { bool | integer | real | string }
<name> ::= [_[:alpha:]][_[:alnum:]]*
```

The `declaration_modifier` `ifndef` declares the variable only if it does not exist yet, otherwise it is ignored.

The `type_modifier` `const` requires the initialization by assigning a value, since the variable would remain uninitialized as `const` variables cannot be assigned a value later.

Examples:

```
set: integer N;
set: integer W = 10;
set: ifndef integer W = 11; # ignored, as W already exists
set: const integer M = 100;
set: string welcome_message = "hello";
set: const real x = -1.e-3;
set: real y;
set: bool B;
set: const real z; # error: const'ants must be initialized!
```

Table 2.2 lists the predefined variables; notice that those not defined as `const` are treated exactly as user-defined variables, so they can be reassigned.

Table 2.2: Predefined variables and constants in math parser

Name	Type	Value
`Time`	Real	Current simulation time
`TimeStep`	Real	Current simulation time step
`Step`	Integer	Current simulation step
`Var`	Real	Set by `dof`, `node`, or `element` drive callers with degree of freedom value, node or element private data value, respectively
`e`	Real	Neper's number
`pi`	Real	π
`FALSE`	Bool	'false'
`TRUE`	Bool	'true'
`INT_MAX`	Integer	Largest integer
`INT_MIN`	Integer	Smallest integer
`RAND_MAX`	Integer	Largest random integer
`REAL_MAX`	Real	Largest real
`REAL_MIN`	Real	Smallest real
`in2m`	Real	Inch to meter ratio (0.0254)
`m2in`	Real	Meter to inch ratio (1.0/0.0254)
`in2mm`	Real	Inch to meter ratio (25.4)
`mm2in`	Real	Meter to inch ratio (1.0/25.4)
`ft2m`	Real	Foot to meter ratio (0.3048)
`m2ft`	Real	Meter to foot ratio (1.0/0.3048)
`lb2kg`	Real	Pound to kilogram ratio (0.45359237)
`kg2lb`	Real	Kilogram to pound ratio (1.0/0.45359237)
`deg2rad`	Real	Degree to radian ratio ($\pi/180$)
`rad2deg`	Real	Radian to degree ratio ($180/\pi$)
`slug2kg`	Real	Slug to kilogram ratio (14.5939)
`kg2slug`	Real	Kilogram to slug ratio (1.0/14.5939)

Operations are strongly typed and perform implicit cast when allowed. For instance `1+2.5` returns a `real` whose value is `3.5`, since one of the two addenda is `real`, while `1/3` returns `0` because both values are implicit integers and thus the integer division is used.

An empty field, delimited by a valid separator (a comma or a semicolon, depending on whether other arguments are expected or not) returns the (program supplied) default value for that field, if supplied by the caller, otherwise the parser automatically defaults to zero.

Multiple expressions can be used, provided they are enclosed in plain brackets and are separated by semicolons; the result of the last expression will be used as the expected numeric value, but all the expressions (which may have persistent effects, like variable declarations and assignments) will be evaluated.

Example.

```
1.                                                  # == 1.
(real r = 2.*pi; integer i = 3; sin(i*r*Time+.87))  # == sin(6*pi*Time+.87)
```

The latter sequence evaluates to $\sin(6\pi t + .87)$, depending on the current value of the pre-defined variable `Time`. The constant `pi` is always defined as π with machine precision (the macro `M_PI` in the standard C header file `math.h`). Other constants are pre-defined, as illustrated in Table 2.2.

When MBDyn is invoked with the '`-H`' option,

```
mbdyn -H
```

it prints the predefined variables table. A typical output is shown in Figure 2.1. Real values are stored with the maximum precision allowed by the underlying `real` type (by default, double precision, 64 bit).

The variable `Time` is declared, defined and initialized[1] from the beginning of the control data section, and during the solution phase it is assigned the value of the current time.

Table 2.3 shows the supported mathematical operators, their precedence and associativity. Notice that precedence may produce unexpected behavior; for example,

```
-2^2 == (-2)^2 = 4
```

Table 2.4 shows the built-in mathematical functions. In addition, type names used as functions provide explicit type casting according to built-in casting rules (Table 2.5). The `<type>_eval` variants evaluate the argument during parsing, rather than run-time. This is useful when parsing strings that will be evaluated later, for example in `string` drive callers or hints, if one wants a particular variable or expression to be evaluated at parsing time.

Example.

- Print a zeros-padded integer: the statement

  ```
  remark: sprintf("%04d", 9);
  ```

  yields

  ```
  line 1, file <initial file>: "0009"
  ```

- Print an integer in hexadecimal format: the statement

[1] Note: a variable is *declared* when its name enters the namespace; it is *defined* when it can be referenced; it is *initialized* when it is first assigned a value.

```
user@host:~> mbdyn -H

MBDyn - MultiBody Dynamics 1.X-Devel
configured on Sep  3 2012 at 12:16:44

Copyright 1996-2023 (C) Paolo Mantegazza and Pierangelo Masarati,
Dipartimento di Ingegneria Aerospaziale <http://www.aero.polimi.it/>
Politecnico di Milano                   <http://www.polimi.it/>

MBDyn is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it
under certain conditions.  Use 'mbdyn --license' to see the conditions.
There is absolutely no warranty for MBDyn.  Use "mbdyn --warranty"
for details.

default symbol table:
  const bool FALSE = 0
  const integer INT_MAX = 2147483647
  const integer INT_MIN = -2147483648
  const integer RAND_MAX = 2147483647
  const real REAL_MAX = 1.79769e+308
  const real REAL_MIN = 2.22507e-308
  const bool TRUE = 1
  const real deg2rad = 0.0174533
  const real e = 2.71828
  const real ft2m = 0.3048
  const real in2m = 0.0254
  const real in2mm = 25.4
  const real kg2lb = 2.20462
  const real kg2slug = 0.0685218
  const real lb2kg = 0.453592
  const real m2ft = 3.28084
  const real m2in = 39.3701
  const real mm2in = 0.0393701
  const real pi = 3.14159
  const real rad2deg = 57.2958
  const real slug2kg = 14.5939

MBDyn terminated normally
```

Figure 2.1: Predefined constants in math parser

Table 2.3: Built-in mathematical operators in math parser (from higher to lower precedence)

Operator	Type	Associativity	Description
+	Unary	—	Plus sign
-	Unary	—	Minus sign
!	Unary	—	NOT
^	Binary	Right	Power
*	Binary	Left	Multiplication
/	Binary	Left	Division
%	Binary	Left	Integer division remainder
+	Binary	Left	Addition
-	Binary	Left	Subtraction
>	Binary	Left	Greater than
>=	Binary	Left	Greater than or equal to
==	Binary	Left	Equal to
<=	Binary	Left	Less than or equal to
<	Binary	Left	Less than
!=	Binary	Left	Not equal
&&	Binary	Left	AND
||	Binary	Left	OR
~|	Binary	Left	XOR (exclusive OR)
=	Binary	Right	Assignment

Note: "left" and "right" refer to the associativity of the operators.
Note: the `string` type only supports the binary "+" operator.

```
remark: sprintf("0x%x", 255);
```

yields

```
line 1, file <initial file>: "0xff"
```

- Print a string that formats multiple numbers: the statement

```
remark: sprintf("N%04d", 9) + " = " + sprintf("%e", 0.1);
```

yields

```
line 1, file <initial file>: "N0009 = 1.000000e-01"
```

2.2.1 Namespaces

The math parser uses the notion of *namespace* to group and uniquely identify functions. A namespace is a (unique) string followed by two colons, as in

```
<namespace>::<subname>
```

The solver recognizes built-in namespaces,

- `default`
- `model`

Table 2.4: Built-in mathematical functions in math parser

Name	Returns	Arg[s]	Description
abs	real	real	absolute value
acos	real	real	arc cosine
acosh	real	real	hyperbolic arc cosine
actan	real	real	arc co-tangent
actan2	real	real, real	(robust) arc co-tangent
actanh	real	real	hyperbolic arc co-tangent
asinh	real	real	hyperbolic arc sine
atanh	real	real	hyperbolic arc tangent
asin	real	real	arc sine
atan	real	real	arc tangent
atan2	real	real, real	(robust) arc tangent
ceil	integer	real	closest integer from above
copysign	real	real, real	first arg with sign of second
cos	real	real	cosine
cosh	real	real	hyperbolic cosine
ctan	real	real	co-tangent
ctanh	real	real	hyperbolic co-tangent
exp	real	real	exponential
floor	integer	real	closest integer from below
in_ee	bool	real, real, real	true when arg1 $\leq$ arg2 $\leq$ arg3, false otherwise
in_el	bool	real, real, real	true when arg1 $\leq$ arg2 $<$ arg3, false otherwise
in_le	bool	real, real, real	true when arg1 $<$ arg2 $\leq$ arg3, false otherwise
in_ll	bool	real, real, real	true when arg1 $<$ arg2 $<$ arg3, false otherwise
log	real	real	natural logarithm
log10	real	real	base 10 logarithm
min	real	real, real	returns the smallest of the two inputs
max	real	real, real	returns the largest of the two inputs
par	real	real	parabolic function
print	void	real	prints a value to standard output
ramp	real	real	ramp function
rand	integer	void	random integer $[0, \text{RAND\_MAX}]$
random	real	void	random real $[-1.0, 1.0]$
round	integer	real	closest integer
seed	void	integer	seeds the random number generator
sign	real	real	sign
sin	real	real	sine
sinh	real	real	hyperbolic sine
sprintf	string	string, any	returns a string with value formatted according to format
sqrt	real	real	square root
sramp	real	real, real	saturated ramp function
step	real	real	step function
stop	integer	bool, integer	stops and returns second arg if first is true
tan	real	real	tangent
tanh	real	real	hyperbolic tangent

Table 2.5: Built-in cast and evaluation functions in math parser

Name	Returns	Arg[s]	Description
bool	bool	(any)	cast to bool
integer	integer	(any)	cast to integer
real	real	(any)	cast to real
string	string	(any)	cast to string
bool_eval	bool	(any)	evaluate arg, casting result to bool
integer_eval	integer	(any)	evaluate arg, casting to integer
real_eval	real	(any)	evaluate arg, casting to real
string_eval	string	(any)	evaluate arg, casting to string

and supports the introduction of user-defined namespaces. The latter can be loaded using the `module load` card described in Section 2.4.6 (see for example the `units` namespace in Section B.7.6 and the related code).

Default Namespace

The functions listed in Table 2.4 are implicitly defined in the `default` namespace, i.e. they should be referenced by writing

```
default::sqrt(2.)
```

but the `default::` portion is optional, as the `default` namespace is implicitly assumed.

Model Namespaces

The namespace that refers to the current model is loaded by default. Its name is `model`, and contains the functions defined in Tables 2.6 and 2.7. With the exception of the `drive` and `sf` functions, all functions in the `model` namespace with the suffix `_prev` (e.g. `position_prev`) use the configuration at the previous time step. They may be useful, for example, when parsing hints (see Section 2.6.41), since `hint` parsing occurs after prediction, but often needs to refer to the configuration at the previous, just completed time step.

Example 1. This string returns the y component of the relative angular velocity between structural nodes "`NODELABEL_1`" and "`NODELABEL_2`" at the current timestep:

```
string, "model::yangvrel(NODELABEL_1,NODELABEL_2)", ...
```

Example 2. This string returns the absolute x component of the position of node "`NODELABEL`" at the previous timestep, plus the value of "`DRIVELABEL_1 * DRIVELABEL_2`" when time is greater than 5 seconds:

```
string, "model::xposition_prev(NODELABEL) + \
    model::drive(DRIVELABEL_1, Time) * model::drive(DRIVELABEL_2, Time) * (Time > 5.)",
```

Example 3. This string returns the product of the value of the scalar function `myfunc` and its first derivative for a given argument:

```
string, "model::sf::myfunc(10.)*model::sf::myfunc(10., 1)"
```

Example 4. To extract the z component of the reaction force, `Fz`, from a joint of label 99:

```
string, "model::element::joint(99, \"Fz\")"
```

Example 5. To extract the current from an electric circuit where a resistor R is defined between electric nodes 1 and 2 (whose state `x` represents the voltage at the node), namely $i = (V_1 - V_2)/R$:

```
string, "(model::node::electric(1, \"x\") - model::node::electric(2, \"x\"))/R"
```

2.2.2 Plugin Variables

Plugin variables, or plugins, as the name states, are pluggable extensions to the math parser that allow to register mechanisms to bind a variable to some means of dynamically generating its value. As a consequence, any time a variable declared as part of a plugin is used, its value is dynamically evaluated executing the related code.

There are built-in plugins that allow to link variables to the value of degrees of freedom and to special parameters computed by nodes and elements.

The syntax of a plugin consists in specially defining a variable so that it gets registered to the appropriate code:

```
set : OSQBR <plugin> , <var> [ , <arglist> ] CSQBR ;
<arglist> ::= <arg> [ , ... ]
```

where `OSQBR` and `CSQBR` stand for 'open' and 'close' square brackets, respectively, which should not be confused with those that indicate optional parameters.

In the following, the built-in plugins[2] are illustrated. For details on nodes, elements, their properties and the information they can expose, see the related sections.

Dof plugin

Note: this plugin is now obsoleted by the `node` *plugin, which allows to access more significant information about nodes, including the value of the degrees of freedom.*

The `dof` plugin allows to link a variable to a specific degree of freedom of a node, or to its derivative, whenever defined. The syntax is

```
<plugin> ::= dof

<arglist> ::= <label> , <type>
    [ , <index> ] ,
    { algebraic | differential }
    [ , prev= <prev> ]

<prev> ::= { true | false }
```

where `label` is the label of the node, `type` is the node type, `index` is the index of the degree of freedom that is requested, if the node has more than one, while `algebraic` and `differential` refer to the value of the dof and of its derivative, respectively. The derivative can be requested only for those degrees of freedom that have one. When `prev` is `true`, the value at the previous time step is used.

[2] Built-in and plug-in may sound contradictory, and in fact they are. However, consider that the mathematical parser is essentially independent of MBDyn (in fact, it has been turned into a command-line calculator in `cl(1)`), so, from this point of view, built-in plug-ins are used to plug MBDyn information into an independent, lower-level piece of code.

Table 2.6: Functions in the `model` namespace

Name	Ret.	Arg[s]	Description
`position`	Real	Integer	norm of structural node position
`position2`	Real	Integer	square norm of structural node position
`xposition`	Real	Integer	X component of structural node position
`yposition`	Real	Integer	Y component of structural node position
`zposition`	Real	Integer	Z component of structural node position
`distance`	Real	Integer, Integer	distance between structural nodes
`distance2`	Real	Integer, Integer	square distance between structural nodes
`xdistance`	Real	Integer, Integer	X component of distance between structural nodes*
`ydistance`	Real	Integer, Integer	Y component of distance between structural nodes*
`zdistance`	Real	Integer, Integer	Z component of distance between structural nodes*
`distancep`	Real	Integer, Integer	relative velocity along distance between structural nodes
`xunitvec`	Real	Integer, Integer	X component of unit vector between structural nodes
`yunitvec`	Real	Integer, Integer	Y component of unit vector between structural nodes
`zunitvec`	Real	Integer, Integer	Z component of unit vector between structural nodes
`angle`	Real	Integer	norm of rotation vector of structural node
`xangle`	Real	Integer	X component of structural node rotation vector
`yangle`	Real	Integer	Y component of structural node rotation vector
`zangle`	Real	Integer	Z component of structural node rotation vector
`anglerel`	Real	Integer, Integer	angle between structural nodes (norm of rotation vector)
`xanglerel`	Real	Integer, Integer	X component of rotation vector between structural nodes*
`yanglerel`	Real	Integer, Integer	Y component of rotation vector between structural nodes*
`zanglerel`	Real	Integer, Integer	Z component of rotation vector between structural nodes*
`velocity`	Real	Integer	norm of structural node velocity
`velocity2`	Real	Integer	square norm of structural node velocity
`xvelocity`	Real	Integer	X component of structural node velocity
`yvelocity`	Real	Integer	Y component of structural node velocity
`zvelocity`	Real	Integer	Z component of structural node velocity
`vrel`	Real	Integer, Integer	norm of relative velocity between structural nodes
`vrel2`	Real	Integer, Integer	square norm of relative velocity between structural nodes
`xvrel`	Real	Integer, Integer	X component of relative velocity between structural nodes*
`yvrel`	Real	Integer, Integer	Y component of relative velocity between structural nodes*
`zvrel`	Real	Integer, Integer	Z component of relative velocity between structural nodes*
`angvel`	Real	Integer	norm of structural node angular velocity
`angvel`	Real	Integer	square norm of structural node angular velocity
`xangvel`	Real	Integer	X component of structural node angular velocity
`yangvel`	Real	Integer	Y component of structural node angular velocity
`zangvel`	Real	Integer	Z component of structural node angular velocity
`angvrel`	Real	Integer, Integer	norm of relative angular velocity between structural nodes
`angvrel2`	Real	Integer, Integer	square norm of rel. angular velocity between structural nodes
`xangvrel`	Real	Integer, Integer	X component of rel. angular velocity between structural nodes*
`yangvrel`	Real	Integer, Integer	Y component of rel. angular velocity between structural nodes*
`zangvrel`	Real	Integer, Integer	Z component of rel. angular velocity between structural nodes*
`current`	Real	String	allows to retrieve data specific to the current context

*"relative" is intended as second node's minus first node's value

Table 2.7: Functions in the model namespace (contd.)

Name	Ret.	Arg[s]	Description
drive	Real	Integer [, Real]	evaluates the drive caller indicated by its label (the first argument) for the input specified as second (optional) argument (Time is used instead, if needed)
sf::<name>	Real	Real [, Integer]	evaluates the scalar function name (<value> [, <order>]) for the input specified as the first argument (the value, optionally followed by the derivative order)
node::<type>	Real	Integer, String	evaluates a private datum of the node type (<label>, <string>) where type is the node type (all types listed in Section 6 are supported), label is the node label, and string indicates the node's private datum
element::<type>	Real	[Integer ,] String	evaluates a private datum of the element type ([<label> ,] <string>) where type is the element type (all types listed in Section 8 are supported), label is the element label (only required if that element is not unique, e.g., it is not gravity or air properties), and string indicates the element's private datum

Example. The variable VARNAME takes the value of the derivative of abstract node NODELABEL

```
set: integer NODELABEL = 1000;
# the node must exist
set: [dof,VARNAME,NODELABEL,abstract,differential];
```

Node plugin

The node plugin allows to link a variable to any data a node can expose, including the values of its degrees of freedom. The syntax is

```
<plugin> ::= node
<arglist> ::= <label> , <type>
    [ , { string= <name> | index= <index> } ] # index is deprecated
```

where

- label is the label of the node,
- type is the node type, and
- the mutually exclusive index and name represent the index of the private datum that is requested, name being a user-friendly representation of the actual index. For a description of valid values for name, consult the *Private Data* of the corresponding node type.

The index form is the default; however, the name form is recommended. Note that name should be enclosed in double quotes, although not strictly required. In fact, some node types allow data names that include square brackets. In those cases, double-quote enclosing is needed to avoid parsing errors, since closing square brackets indicate the end of the plugin variable specification.

Example. This implements a variable `VARNAME` with exactly the same value of the one defined in the example of the `dof` plugin, where the variable takes the value of the derivative of the `abstract` node `NODELABEL`

```
set: integer NODELABEL = 1000;
# the node must exist
set: [node,VARNAME,NODELABEL,abstract,string="xP"];
```

Element plugin

The `element` plugin allows to link a variable to any data an element can expose, including the values of its degrees of freedom. The syntax is

```
<plugin> ::= element
<arglist> ::= <label> , <type>
    [ , { string= <name> | index= <index> } ] # index is deprecated
```

where

- `label` is the label of the element,
- `type` is the element type, and
- the mutually exclusive `index` and `name` represent the index of the private datum that is requested, `name` being a user-friendly representation of the actual index. For a description of valid values for `name`, consult the *Private Data* of the corresponding element type.

The `index` form is the default; however, the `name` form is recommended. Note that `name` should be enclosed in double quotes, although not strictly required. In fact, some element types allow names that include square brackets. In those cases, double-quote enclosing is needed to avoid parsing errors, since closing square brackets indicate the end of the plugin variable specification.

Example. The variable `VARNAME` takes the value of the z component of the reaction force of `joint` `ELEMLABEL`

```
set: integer ELEMLABEL = 1000;
# the joint must exist
set: [element,VARNAME,ELEMLABEL,joint,string="Fz"];
```

Equivalence Between Node/Element Plugin Variables and Node/Element Functions in Model Namespace

The following syntaxes are equivalent:

```
set: [element,VARNAME,ELEMLABEL,joint,string="Fz"];
string, "abs(VARNAME)" # generic use of Fz...
# is equivalent to
string, "abs(model::element::joint(ELEMLABEL,\"Fz\"))" # same generic use of Fz...
```

i.e., setting an element plugin variable `VARNAME` and using it in a string is exactly equivalent to using the corresponding element private data through the `model` namespace. The same is true for the node pluging variables and the corresponding node private data through the `model` namespace.

2.3 Higher-Order Math Structures

Every time a higher-order mathematical structure is expected, it can be preceded by a keyword that influences how the structure is read. All of the available structures support the keyword `null` which causes the structure to be initialized with zeros. When a non-null value is input, it can be followed by the keyword `scale` with a scale factor; the scale factor can be any mathematical expression. This is useful to rescale structure values by reassigning the value of the scale factor. The main data structures are:

2.3.1 3×1 Vector

Implements the type `Vec3`.

1. general case: a sequence of 3 reals, comma-separated.
2. null vector: keyword `null`; the vector is initialized with zeros.

As an example, all the following lines define an empty 3×1 vector:

```
default
null
0.,0.,0.
,,
```

the first case is correct if no default was actually available for that specific vector, thus falling back to three zeros. The following rescales an arbitrary vector

```
cos(pi/3.),0.,sin(pi/3.), scale, 100.
```

2.3.2 6×1 Vector

Implements the type `Vec6`.

1. general case: a sequence of 6 reals, comma-separated.
2. null vector: keyword `null`; the vector is initialized with zeros.

2.3.3 3×3 Matrix

Implements the type `Mat3x3`.

1. general case: a sequence of 9 reals, comma-separated, which represent the row-oriented coefficients a_{11}, a_{12} , ..., a_{32}, a_{33}. *Note: the 9 coefficients can be preceded by the keyword `matr` for consistency with other entities; its use is recommended whenever an ambiguity is possible.*

$$\boldsymbol{m} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{2.1}$$

Example.

```
# general case
1., 0., 0.,
0., 1., 0.,
```

```
0., 0., 1.
# or
matr,
    1., 0., 0.,
    0., 1., 0.,
    0., 0., 1.
```

2. symmetric matrix: keyword `sym`, followed by a sequence of 6 reals, comma-separated, that represents the upper triangle, row-oriented coefficients of a symmetric matrix, e.g. a_{11}, ..., a_{13}, a_{22}, a_{23}, a_{33}.

$$\boldsymbol{m} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} \tag{2.2}$$

Example.

```
# symmetric matrix
sym,
    1., 0., 0.,
        1., 0.,
            1.,
```

3. skew-symmetric matrix: keyword `skew`, followed by a sequence of 3 reals, comma-separated, that are the components of the vector $\boldsymbol{v}$ that generates a skew symmetric matrix $\boldsymbol{m} = \boldsymbol{v} \times :$

$$\boldsymbol{m} = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix} \tag{2.3}$$

Example.

```
# skew-symmetric matrix
skew , 1., 2., 3.
```

4. diagonal matrix: keyword `diag`, followed by a sequence of 3 reals, comma-separated, that represent the diagonal coefficients of a diagonal matrix, namely a_{11}, a_{22}, a_{33}

$$\boldsymbol{m} = \begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{bmatrix} \tag{2.4}$$

Example.

```
# diagonal matrix
diag , 1., 1., 1.
```

5. identity matrix: keyword `eye`; the matrix is initialized as the identity matrix, that is a null matrix except for the diagonal coefficients that are 1. **Example.**

```
# identity matrix
eye
```

6. null matrix: keyword `null`; the matrix is initialized with zeros.

```
# null matrix
null
```

For example, the identity matrix can be defined as:

```
matr, 1.,0.,0.,  0.,1.,0.,  0.,0.,1.
1.,0.,0.,  0.,1.,0.,  0.,0.,1.        # ‘‘matr’’ omitted
sym, 1.,0.,0.,  1.,0.,  1.            # upper triangular part
diag, 1.,1.,1.                        # diagonal
eye                                   # a la matlab
```

Although not required, for better readability it is recommended to format the above data as

```
matr,
    1.,0.,0.,
    0.,1.,0.,
    0.,0.,1.
1.,0.,0.,
0.,1.,0.,
0.,0.,1.                              # ‘‘matr’’ omitted
sym,
    1.,0.,0.,
       1.,0.,
          1.                          # upper triangular part
diag, 1.,1.,1.                        # diagonal
eye                                   # a la matlab
```

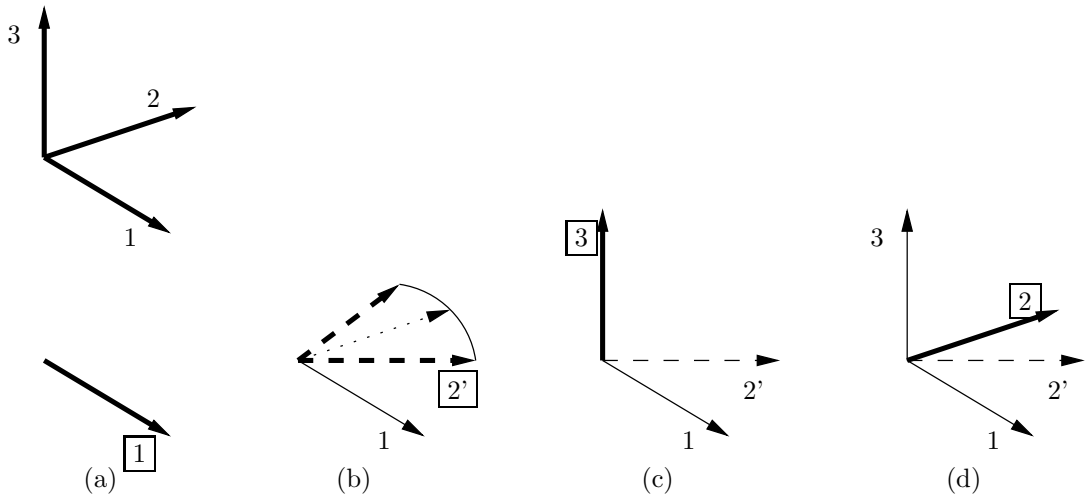


Figure 2.2: Construction of 3×3 orientation matrix from two non-parallel vectors.

2.3.4 (3×3) Orientation Matrix

Implements the type `OrientationMatrix`.

1. general case: two vectors that define an orthonormal reference system, each of them preceded by its index in the final orientation matrix. The procedure that builds the orientation matrix from the two vectors is illustrated in Figure 2.2.

```
<v1> , <v1_1>, <v1_2> , <v1_3> ,
<v2> , <v2_1>, <v2_2> , <v2_3>

<v1> ::= { 1 | 2 | 3 }
<v2> ::= { 1 | 2 | 3 } # <v2> != <v1>
```

The first vector, <v1>, namely vector 1 in subfigure (a), is normalized and assumed to represent the desired direction. The second vector, <v2>, namely vector 2' in subfigure (b), simply concurs in the definition of the plane the vector that is not given is normal to. The third vector, vector 3 in subfigure (c), results from the cross-product of the first two vectors, after normalization. The actual direction of the second vector, vector 2 in subfigure (d), results from the cross product of the third and the first vector.

Examples.

```
1, 1.,0.,0., 2, 0.,1.,0.
```

Equivalent to the identity matrix, i.e. no rotation occurs with respect to the initial reference frame: direction 1 in the final reference frame is parallel to `1.,0.,0.`, which represents direction 1 in the initial reference frame, while direction 2 in the final reference frame is parallel to `0.,1.,0.`, which represents direction 2 in the initial reference frame.

```
1, cos(pi/6.), sin(pi/6.), 0.,
    3, 0.,0.,1.
```

Rotation of $\pi/6$ radian about initial direction 3: direction 1 in the final reference frame results from composing `cos(pi/6.)` in initial direction 1 and `sin(pi/6.)` in initial direction 2, while direction 3 in the final reference frame remains parallel to `0.,0.,1.`, which represents direction 3 in the initial reference frame.

```
2, 0., cos(ALPHA), sin(ALPHA),
    1, 1.,0.,0.
```

Rotation of `ALPHA` (assuming the variable `ALPHA` was previously defined) about initial direction 1: direction 2 in the final reference frame results from composing `cos(ALPHA)` in initial direction 2 and `sin(ALPHA)` in initial direction 2, while direction 3 in the final reference frame remains parallel to `0.,0.,1.`, which represents direction 3 in the initial reference frame.

2. a variant of the above, which may be useful when only one direction really matters, is

```
<v1> , <v1_1>, <v1_2> , <v1_3> ,
<v2> , guess

<v1> ::= { 1 | 2 | 3 }
<v2> ::= { 1 | 2 | 3 } # <v2> != <v1>
```

and is illustrated in the example below:

```
1, 1.,0.,0., 2, guess
```

The keyword guess tells the parser to generate a random vector that is orthogonal to the given one, which is used as the direction indicated by the index (2 in the example). The vector is computed based on a very simple algorithm: it contains

- 1.0 corresponding to the index with smallest modulus, v_1 (min);
- $-v_1$ (min) / v_1 (max) corresponding to the index with the largest modulus, v_1 (max);
- 0.0 for the remaining index.

3. identity matrix: keyword eye; the identity matrix, which means there is no rotation with respect to the global reference frame.

4. a complete orientation matrix: keyword matr followed by the nine, row-oriented, coefficients, namely r_{11}, r_{12}, ..., r_{33}.

```
matr ,
    <r11> , <r12> , <r13> ,
    <r21> , <r22> , <r23> ,
    <r31> , <r32> , <r33>
        [ , tolerance , <tol> ]
```

The tolerance tol is used when checking the orthogonality of the matrix (the default is 10^{-12}).

Example:

```
matr,
    r11, r12, r13,
    r21, r22, r23,
    r31, r32, r33
```

Note: orthogonality is not enforced; if the matrix is not orthogonal within the desired tolerance, only a warning is issued; be sure an orthogonal matrix, within the desired tolerance, is input.

5. Euler angles: keyword euler, followed by the three values, as output by structural nodes. The keywords euler123 (same as euler, also known as Tait-Bryan or Cardano angles), euler313 and euler321 allow to use orientations in the sequence specified by the keyword.

```
{ euler | euler123 | euler313 | euler321 } ,
    [ degrees , ]
    <angle_1> , <angle_2> , <angle_3>
```

Note: the output in the .mov file is in degrees, while in input MBDyn requires angles in radians, unless the values are preceded by the keyword degrees.

Note: the definition of the three angles that are used by the code to express orientations may vary between versions. Currently, Bryant-Cardano angles are used in place of Euler angles; see the note related to the output of the structural nodes in Section 6.5.5, and the Technical Manual. The code will remain consistent, i.e. the same angle definition will be used for input and output, but models over versions may become incompatible, so this syntax should really be used only as a means to quickly reproduce in the input an orientation as resulting from a previous analysis.

6. Orientation vector: keyword vector, followed by the three values, as output by structural nodes. The resulting matrix is computed as

$$\boldsymbol{R} = \exp\left(\boldsymbol{v} \times\right) \tag{2.5}$$

The keyword null is intercepted, since its conventional usage would result in a zero-valued matrix which, by definition, cannot be orthogonal; an error is raised.

2.3.5 6×6 Matrix

Implements the type `Mat6x6`.

1. general case: a sequence of 36 reals, comma-separated, that represent the row-oriented coefficients a_{11}, a_{12} , ..., a_{65}, a_{66}.
2. ANBA format: keyword `anba`, followed by 36 reals, comma-separated, that represent the coefficients of the beam stiffness matrix as generated by the beam section analysis code ANBA, namely the following transformation is performed:
   - axis x, in the section plane in ANBA notation, becomes axis 2 in MBDyn notation;
   - axis y, in the section plane in ANBA notation, becomes axis 3 in MBDyn notation;
   - axis z, the beam axis in ANBA notation, becomes axis 1 in MBDyn notation;

   Note: this format is mainly intended for backwards compatibility with older versions of that beam section analysis software, which used a different numbering convention for the reference frame that is local to the beam section.
3. symmetric matrix: keyword `sym`, followed by a sequence of 21 reals, comma-separated, that represents the upper triangle, row-oriented coefficients of a symmetric matrix, e.g. a_{11}, ..., a_{16}, a_{22}, ..., a_{26}, ..., a_{66}.
4. diagonal matrix: keyword `diag`, followed by a sequence of 6 reals, comma-separated, that represent the diagonal coefficients of a diagonal matrix.
5. identity matrix: keyword `eye`; the matrix is initialized as the identity matrix, that is a null matrix except for the diagonal coefficients that are 1.
6. null matrix: keyword `null`; the matrix is initialized with zeros.

2.3.6 $6 \times N$ Matrix

Implements the type `Mat6xN`.

1. general case: a sequence of $6 \times N$ reals, comma-separated, that represent the row-oriented coefficients a_{11}, a_{12} , ..., $a_{6(N-1)}$, a_{6N}.
2. ANBA format: keyword `anba`, followed by $6 \times N$ reals, comma-separated, that represent the coefficients of the beam stiffness matrix as generated by the code ANBA, namely the following transformation is performed:
   - axis x, in the section plane in ANBA notation, becomes axis 2 in MBDyn notation;
   - axis y, in the section plane in ANBA notation, becomes axis 3 in MBDyn notation;
   - axis z, the beam axis in ANBA notation, becomes axis 1 in MBDyn notation;
3. null matrix: keyword `null`; the matrix is initialized with zeros.

2.3.7 $N \times 6$ Matrix

Implements the type MatNx6.

1. general case: a sequence of $N \times 6$ reals, comma-separated, that represent the row-oriented coefficients
   a_{11}, a_{12} ,..., a_{16},
   a_{21}, a_{22}, ..., a_{26},
   ...,
   a_{N1}, a_{N2}, ..., a_{N6}.

2. ANBA format: keyword anba, followed by $N \times 6$ reals, comma-separated, with the columns representing the coefficients of the beam generalized deformations as generated by the code ANBA, namely the following transformation is performed:
   - axis x, in the section plane in ANBA notation, becomes axis 2 in MBDyn notation;
   - axis y, in the section plane in ANBA notation, becomes axis 3 in MBDyn notation;
   - axis z, the beam axis in ANBA notation, becomes axis 1 in MBDyn notation;

3. null matrix: keyword null; the matrix is initialized with zeros.

2.3.8 $N \times N$ Matrix

Implements the type MatNxN.

1. general case: a sequence of $N \times 6$ reals, comma-separated, that represent the row-oriented coefficients a_{11}, a_{12} , ..., $a_{N(N-1)}$, a_{NN}.

2. null matrix: keyword null; the matrix is initialized with zeros.

2.4 Input Related Cards

(Almost) everywhere in the input file the statement cards defined in the following can be used. They are handled directly by the parsing object, and merely act as an indirect reference to entities that are not explicitly enumerated. They are:

2.4.1 Constitutive Law

```
<card> ::= constitutive law : <label> ,
    [ name , " <name> " , ]
    <dim> , (ConstitutiveLaw<<dim>D>) <constitutive_law> ;
```

Constitutive laws are grouped by their dimensionality dim, which (up to now) can be any of 1, 3, 6 and 7. The constitutive_law is parsed according to the rules described in Section 2.10, and can be referenced later when it needs to be used.

2.4.2 C81 Data

This keyword allows to define and read the `c81 data` airfoil tables that are used by aerodynamic elements.

```
<card> ::= c81 data : <label> [ , name , " <name> " ]
    " <filename> "
    [ , tolerance , <tolerance> ]
    [ , { free format | nrel | fc511 } ]
    [ , flip ]
    [ , echo , " <output_filename> " [ , free format ] ] ;
```

Data is read from file `filename` according to the format specified in the following.

The optional keyword `tolerance` allows to specify the tolerance that is used to determine the boundaries of the linear portion of the lift curve slope.

The traditional format is used unless a format is specified using any of the keywords `free format`, `nrel` or `fc511` (the latter is intentionally not documented). The format is automatically inferred by reading the heading line of the input file.

The optional keyword `flip` instructs MBDyn to "flip" the data, i.e. treat the airfoil data as if the sign of the angle of attack changed along with that of the lift and moment coefficients, but not that of the drag coefficient. This corresponds to considering the airfoil "upside down".

The optional keyword `echo` allows to specify the name of the file that will be generated with the data just read, for cross-checking purposes. If the following optional keyword is `free format`, or if data was read in free format, the echo will also be in free format. Otherwise, it will be in the traditional format.

Traditional Format

The file is in textual form; the traditional format (the default) is:

- first line: `"%30s%2d%2d%2d%2d%2d%2d"` where the first 30 chars are a title string, currently ignored by MBDyn, followed by 6 consecutive two-digit integers that indicate:
  - the number `ML` of *Mach* points for C_l;
  - the number `NL` of angle of attack points for C_l;
  - the number `MD` of *Mach* points for C_d;
  - the number `ND` of angle of attack points for C_d;
  - the number `MM` of *Mach* points for C_m;
  - the number `NM` of angle of attack points for C_m.

  The example in `var/naca0012.c81` contains:

  ```
  1........01........01........0MLNLMDNDMMNM; not part of format
  PROFILO NACA 0012             11391165 947
  ```

- the format of each following line is up to 10 fields of 7 chars each; records longer than 10 fields are broken on multiple lines, with the first field filled with blanks;
- a block containing the C_l data, made of:
  - a record with the first field blank, followed by the `ML` *Mach* values for the C_l;
  - `NL` records containing the angle of attack in the first field, followed by `ML` values of C_l for each *Mach* number; angles of attack wrap around 360 deg, starting from -180 deg.

The example provided in file `var/naca0012.c81` contains 11 *Mach* points and 39 angle of attack records for the C_l of the NACA0012 airfoil at high (unspecified) *Reynolds*:

```
......7......7......7......7......7......7......7......7......7......7; not part of format
       0.     .20    .30    .40    .50    .60    .70    .75    .80
       .90    1.
-180.  0.     0.     0.     0.     0.     0.     0.     0.     0.
       0.     0.
-172.5 .78    .78    .78    .78    .78    .78    .78    .78    .78
       .78    .78
...
```

- a block containing the C_d data, same as for C_l, with `MD` *Mach* points and `ND` angle of attack records;
- a block containing the C_m data, same as for C_l, with `MM` *Mach* points and `NM` angle of attack records.

Free Format

Finally, to allow higher precision whenever available, a native format, based on `c81`, called `free format`, is available. It basically consists in the `c81` format without continuation lines and with arbitrary precision, with fields separated by blanks. The header is made of an arbitrary string, terminated by a semicolon '`;`', followed by the six numbers that define the dimensionality of the expected data.

Example.

```
# FREE FORMAT
this is the header; 2 8 2 2 2 2
        0.0   0.9
-180.0  0.0   0.0
-170.0  1.0   0.9
 -90.0  0.0   0.0
 -10.0 -1.0  -0.9
  10.0  1.0   0.9
  90.0  0.0   0.0
 170.0 -1.0  -0.9
 180.0  0.0   0.0
        0.0   0.9
-180.0  0.1   0.1
 180.0  0.1   0.1
        0.0   0.9
-180.0  0.0   0.0
 180.0  0.0   0.0
```

NREL

The `nrel` format is used to provide airfoil data for wind turbines. The format is very compact, but does not allow to account for Mach dependence, and requires the same angle of attack resolution for all three coefficients. See AeroDyn's User Guide [1] for a description of the format. Please note that MBDyn only supports one airfoil per file, and only considers static coefficients from line 15 on.

To be completed.

Alternative Format

An alternative format, required by some projects, can be used by supplying the optional switch `fc511`; it is intentionally not documented.

Format Conversion

Traditional format files can be automatically converted in free format using the `c81test(1)` utility:

```
$ c81test -d var/naca0012_free_format.c81 var/naca0012.c81
```

generates a free format version of the NACA 0012 airfoil data provided in the distribution.

Miscellaneous

A simple program that allows to read and plot the C81 tables is available at
`http://homepage.mac.com/jhuwaldt/java/Applications/TableReader/TableReader.html`
(thanks to Marco Fossati for pointing it out).

A simple utility that parses the C81 file and computes the aerodynamic coefficients for a given pair of angle of attack and Mach number is `c81test(1)`, available in the MBDyn utils suite. This routine uses exactly the same code internally used by MBDyn, so it should be considered a check of the code rather than a real tool.

2.4.3 Drive Caller

The `drive caller` directive

```
<card> ::= drive caller : <label> ,
    [ name , " <name> " , ]
    (DriveCaller) <drive_caller> ;
```

allows to define a `drive caller` (see Section 2.6) that can be subsequently reused. It is useful essentially in two cases:

a) to define a `drive` that will be used many times throughout a model;

b) to define a `drive` that needs to be used in a later defined part of a model, in order to make it parametric.

2.4.4 Hydraulic fluid

The `hydraulic fluid` directive:

```
<card> ::= hydraulic fluid : <label> ,
    <fluid_type> , <fluid_properties> ;
```

allows to define a hydraulic fluid to be later used in hydraulic elements, see Section 8.11. The fluid is identified by the `label`. The `fluid_type`s, with the related `fluid_properties`, are described in 2.11

2.4.5 Include

The include directive

```
<card> ::= include : " <file_name> " ;
```

allows to include the contents of the file file_name, which must be a valid filename for the operating system in use. The file name must be enclosed in double quotes ("). The full (absolute or relative) path must be given if the included file is not in the directory of the including one.

There is no check for recursively including files, so **the user must take care of avoiding recursions**.

The include directive forces the parser to scan the included file file_name before continuing with the including one. This is very useful if, for instance, a big model can be made of many small models that are meaningful by themselves.

It can be used to replicate parts of the model, by simply using parametric labels for nodes, elements, reference systems, and setting a bias value before multiple-including the same bulk data file. Examples of this usage are given in the tutorials (`https://github.com/mmorandi/MBDyn-web/raw/main/userfiles/documents/tutoria` and examples (`https://www.mbdyn.org/Documentation/Examples.html`).

2.4.6 Module Load

The module load directive:

```
<card> ::= module load : " <file_name> "
    [ , <module_arglist> ] ;

<module_arglist> ::= <arg> [ , ... ]
```

where file_name is the name of a runtime loadable object, causes the object to be opened, and a function `module_init()`, with prototype

```
extern "C" int
module_init(const char *module_name,
        void *data_manager, void *mbdyn_parser);
```

to be executed.

The function is assumed to perform the operations required to initialize the module, eventually taking advantage of the parsing and of the data manager; see the technical manual for details.

The function is also expected to take care of the optional module_arglist arguments; in detail, module developers are encouraged to support help as the first optional argument. The presence of this argument should result in printing to standard output as much information as possible about the use of the module.

The typical use consists in registering some methods for later use. A clear example is given in

```
modules/module-wheel2/module-wheel2.cc
```

where the `module_init()` function registers the user defined element called wheel2 in the set of user-defined element handlers. The module can be retrieved later using the syntax described in Section 8.19. Note, however, that the `module_init()` function may be used for any purpose. A typical use consists in registering any kind of handlers for subsequent use.

Run-time module loading is taken care of by GNU's `libltdl`. Modules are compiled using `libtool` for portability purposes. The resulting modules take the name `libmodule-<name>.la`.

Modules are installed by default in the directory `${prefix}/libexec`. When loaded using only the module name, the default directory is searched. The run-time loading path can be modified by the `loadable path` statement described in Section 5.3.19.

Although `libltdl` is supposed to be portable on a wide variety of platforms (this is what it is designed for, all in all), run-time loading within MBDyn is mainly tested using Linux. Users are encouraged to report problems they might encounter, especially when building modules for different platforms, as this would help making MBDyn more portable.

2.4.7 Print symbol table

The `print symbol table` directive:

```
<card> ::= print symbol table [ :  all | <namespace_list>  ] ;

<namespace_list> ::= <namespace> [ , <namespace_list> ]
```

allows to print to standard output the contents of the parser's symbol table at any stage of the input phase. This may be useful for model debugging purposes.

When no arguments are provided, it prints the symbol table of the math parser. Otherwise, it prints the symbol table associated with a specific namespace, according to the list of arguments (all namespaces when the keyword `all` is used).

2.4.8 Reference

The `reference` directive:

```
<card> ::= reference : <unique_label> ,
    [ name , (string) <name> , ]
    [ position , ]          (Vec3) <absolute_position> ,
    [ orientation , ]       (OrientationMatrix) <absolute_orientation_matrix> ,
    [ velocity , ]          (Vec3) <absolute_velocity> ,
    [ angular velocity , ] (Vec3) <absolute_angular_velocity> ;
```

A `reference` system is declared and defined. It must be given a unique identifier, scanned by the math parser (which means that any regular expression is allowed, and the result is rounded up to the nearest unsigned integer). The entries `absolute_*` are parsed by routines that compute absolute (i.e. referring to the global frame) entities starting from a given entity in a given reference frame. These routines are very general, and make intense use of the `reference` entries themselves, which means that a reference can be recursively defined by means of previously defined `reference` entries.

Alternative form, based on Denavit-Hartenberg parameters, which are particularly used in robotics. It defines the reference system of the second part of a joint with respect to the first part, when the two parts are connected by revolute joints.

```
<card> ::= reference : <unique_label> ,
    [ name , " <name> " , ]
    Denavit Hartenberg ,
    [ reference , { global | <ref_label> } , ]
    <d> , <theta> , <a> , <alpha> ;
```

where:

- `d`: offset d along previous z axis to the common normal (the current x axis);

- `theta`: the angle θ about previous z from previous to current x axis;
- `a`: length a of the common normal; for revolute joints, the radius of the previous z;
- `alpha`: the angle α about the common normal, from previous to new z axis.

The configuration of the first part of the joint is identified by an axis z_1 and a point P_1 on it. The configuration of the second part of the joint is identified by an axis z_2; point P_2 on the second part of the joint results from the definition itself, and is not strictly relevant for the definition. If the two axes are not parallel, they have a common normal, x_2, that crosses both; otherwise, one can arbitrarily choose x_2 as a direction orthogonal to both z_1 (and z_2, since they are parallel, or even coincident). Choose x_1 as a vector normal to z_1, originating from the reference point of the first part of the joint, P_1, and directed towards the reference point of the second part of the joint, P_2 (this is not strictly required, as one can choose axis x_1 at will, as it only provides the reference for the joint rotation). In general, x_1 will not be able to intersect axis z_2, unless axes z_1 and z_2 are co-planar. Point P_2 is the intersection between axes x_2 and z_2. If needed, axes y_1 and y_2 are defined according to the right-hand rule. Then:

- `d` is the distance, along z_1, between the reference point of the first part, P_1, and the plane determined by axes z_2 and x_2;
- `theta` is the angle about axis z_1 between axes x_2 and x_1; it represents the actual rotation of the joint;
- `a` is the distance between axes z_1 and z_2, which is along axis x_2;
- `alpha` is the angle formed by axis z_2 with respect to axis z_1; it corresponds to a rotation about axis x_2.

The new reference data are

$$\boldsymbol{x}_{\mathrm{new}} = \boldsymbol{x}_{\mathrm{old}} + \mathbf{R}_{\mathrm{old}} \left(\mathtt{d}\, \boldsymbol{e}_3 + \exp\left((\mathtt{theta}\, \boldsymbol{e}_3) \times \right) \cdot (\mathtt{a}\, \boldsymbol{e}_1) \right) \tag{2.6a}$$

$$\mathbf{R}_{\mathrm{new}} = \mathbf{R}_{\mathrm{old}} \exp\left((\mathtt{theta}\, \boldsymbol{e}_3) \times \right) \exp\left((\mathtt{alpha}\, \boldsymbol{e}_1) \times \right) \tag{2.6b}$$

$$\boldsymbol{v}_{\mathrm{new}} = \boldsymbol{v}_{\mathrm{old}} + \boldsymbol{\omega}_{\mathrm{old}} \times \mathbf{R}_{\mathrm{old}} \left(\mathtt{d}\, \boldsymbol{e}_3 + \exp\left((\mathtt{theta}\, \boldsymbol{e}_3) \times \right) \cdot (\mathtt{a}\, \boldsymbol{e}_1) \right) \tag{2.6c}$$

$$\boldsymbol{\omega}_{\mathrm{new}} = \boldsymbol{\omega}_{\mathrm{old}} \tag{2.6d}$$

where the subscript "old" refers to the reference frame with label `ref_label1`, if any, or otherwise to the global reference system.

Use of Reference Frames

Every time an absolute or a relative geometric or physical entity is required, it is processed by a set of routines that allow the entity to be expressed in the desired reference frame. The following cases are considered:

- relative position (physical)
- absolute position (physical)
- relative orientation matrix (physical)
- absolute orientation matrix (physical)
- relative velocity (physical)
- absolute velocity (physical)

- relative angular velocity (physical)
- absolute angular velocity (physical)
- relative arbitrary vector (geometric)
- absolute arbitrary vector (geometric)

The caller is responsible for the final interpretation of the input. The caller always supplies the routines a default reference structure the input must be referred to. So, depending on the caller, the entry can be in the following forms:

1. `<entity>`:
   the data supplied in `entity` is intended in the default reference frame
2. `reference , <reference_type> , <entity>`:
   the data are in `reference_type` reference frame, where

   ```
   <reference_type> ::= { global | node | local }
   ```

3. `reference , <reference_label> , <entity>`:
   the data are in `reference_label` reference frame. This reference frame must be already defined.

In some cases, significantly in case of joints (see Section 8.12) that connect two nodes a and b, special reference types are allowed when reading specific entities related to the second node. These reference types compute the value of the entity with respect to the reference frame associated with the first node, a:

1. `other position`:
   a relative position $\tilde{\boldsymbol{p}}$ is intended in the other node's reference frame, with respect to the relative position $\tilde{\boldsymbol{f}}_a$ already specified for the other node,

   $$\tilde{\boldsymbol{f}}_b = \mathbf{R}_b^T \left(\boldsymbol{x}_a + \mathbf{R}_a \left(\tilde{\boldsymbol{f}}_a + \tilde{\boldsymbol{p}} \right) - \boldsymbol{x}_b \right), \tag{2.7}$$

   which is the solution of equation

   $$\boldsymbol{x}_a + \mathbf{R}_a \left(\tilde{\boldsymbol{f}}_a + \tilde{\boldsymbol{p}} \right) = \boldsymbol{x}_b + \mathbf{R}_b \tilde{\boldsymbol{f}}_b; \tag{2.8}$$

2. `other orientation`:
   a relative orientation $\tilde{\mathbf{R}}$ is intended in the other node's reference frame, with respect to the relative orientation $\tilde{\mathbf{R}}_{ha}$ already specified for the other node,

   $$\tilde{\mathbf{R}}_{hb} = \mathbf{R}_b^T \mathbf{R}_a \tilde{\mathbf{R}}_{ha} \tilde{\mathbf{R}}, \tag{2.9}$$

   which is the solution of equation

   $$\mathbf{R}_a \tilde{\mathbf{R}}_{ha} \tilde{\mathbf{R}} = \mathbf{R}_b \tilde{\mathbf{R}}_{hb}; \tag{2.10}$$

3. `other node`:
   a relative position $\tilde{\boldsymbol{p}}$ or a relative orientation $\tilde{\mathbf{R}}$ are intended in the other node's reference frame,

   $$\tilde{\boldsymbol{f}}_b = \mathbf{R}_b^T \left(\boldsymbol{x}_a + \mathbf{R}_a \tilde{\boldsymbol{p}} - \boldsymbol{x}_b \right) \tag{2.11a}$$

   $$\tilde{\mathbf{R}}_{hb} = \mathbf{R}_b^T \mathbf{R}_a \tilde{\mathbf{R}}, \tag{2.11b}$$

   which are the solutions of equations

   $$\boldsymbol{x}_a + \mathbf{R}_a \tilde{\boldsymbol{p}} = \boldsymbol{x}_b + \mathbf{R}_b \tilde{\boldsymbol{f}}_b \tag{2.12a}$$

   $$\mathbf{R}_a \tilde{\mathbf{R}} = \mathbf{R}_b \tilde{\mathbf{R}}_{hb}. \tag{2.12b}$$

Example.

- absolute position:

```
null
reference, global, null
reference, 8, 1., sin(.3*pi), log(3.)
```

- relative orientation matrix (e.g. as required by many constraints and thus referred to a node):

```
eye
reference, node, eye
reference, 8,
    3, 0., 1., 0.,
    1, .5, sqrt(3)/2., 0.
```

Notes:

- the global reference frame has position $\{0, 0, 0\}$, orientation matrix `eye`, velocity $\{0, 0, 0\}$ and angular velocity $\{0, 0, 0\}$.
- if the caller is not related to a node, the reference type `node` is not defined.
- when processing a velocity or an angular velocity, the resulting value always accounts for the velocity and angular velocity of the frame the entry is referred to.

  As an example, if a node is defined in a reference frame $\boldsymbol{R}_R$ that has non-null angular velocity $\boldsymbol{\Omega}_R$, and the position $\boldsymbol{x}_{\text{input}}$ of the node is not coincident with the origin $\boldsymbol{X}_R$ of the reference frame it is attached to, its global velocity and angular velocity result as the composition of the input values and of those of the reference frame:

$$\begin{aligned} \boldsymbol{w} &= \boldsymbol{R}_R \boldsymbol{\omega}_{\text{input}} + \boldsymbol{\Omega}_R \\ \boldsymbol{v} &= \boldsymbol{R}_R \boldsymbol{v}_{\text{input}} + \boldsymbol{V}_R + \boldsymbol{\Omega}_R \times (\boldsymbol{R}_R \boldsymbol{x}_{\text{input}}) \end{aligned}$$

  This, for instance, eases the input of all the parts of a complex system that is moving as a rigid body, by defining a reference frame with the proper initial linear and angular velocity, and then referring all the entities, e.g. the nodes, to that frame, with null local linear and angular velocity.

Recalling the declaration and the definition of reference frames, a simple reference frame definition, with all the entries referring by default to the global system, would be:

```
reference: 1000,
    null,
    eye,
    null,
    null;
```

which represents a redefinition of the global system.

A more verbose, and self-explanatory definition would be:

```
reference: 1000,
    reference, global, null,
    reference, global, eye,
    reference, global, null,
    reference, global, null;
```

The reference frame one is referring to must be repeated for all the entries since they must be allowed to refer to whatever frame is preferred by the user.

A fancier definition would be:

```
reference: Rotating_structure,
    reference, Fixed_structure, null,
    reference, Spindle_1,
        1, 0., 0., 1.,
        3, 0., 1., 0.,
    reference, Fixed_structure, null,
    reference, Spindle_1, 0., 0., Omega_1;
```

Output

The reference frames are used only during the input phase, where they help referring entities either absolute or relative to other entities depending on their internal representation during the analysis. As such, reference frames cannot be "used" or "visualized" neither directly nor indirectly at any time during the analysis or by interpreting the output, because they do not "evolve" nor are attached to any state-dependent entity. To allow their debugging, however, they can be output in the global reference frame according to the representation of structural nodes, as described in Section 6.5.5, by using the `default output` directive with the value `reference frames`, as detailed in Section 5.3.6.

2.4.9 Remark

The `remark` directive:

```
<card> ::= remark [ : <math_expression> [ , ... ] ] ;
```

This directive simply prints to stdout the (list of) expression(s) `math_expression`. It is used to allow rough input debugging. The file name and line number are logged first, followed by the result of the evaluation of the expression(s list).

Example. A file "remarks", containing only the statements

```
remark: "square root of 2", sqrt(2);
set: (
    real EA = 1e6; # N, axial stiffness
    real GA = 1e6; # N, shear stiffness
    real EJ = 1e3; # Nm^2, bending stiffness
    real GJ = 1e3; # Nm^2, torsional stiffness
0);
remark: "Stiffness properties", EA, GA, EJ, GJ;
```

results in

```
user@host:~>$ mbdyn -f remarks

MBDyn - Multi-Body Dynamics 1.6.2
configured on Jun 26 2015 at 10:17:02

Copyright 1996-2023 (C) Paolo Mantegazza and Pierangelo Masarati,
Dipartimento di Ingegneria Aerospaziale <http://www.aero.polimi.it/>
Politecnico di Milano                   <http://www.polimi.it/>

MBDyn is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it
under certain conditions.  Use 'mbdyn --license' to see the conditions.
There is absolutely no warranty for MBDyn.  Use "mbdyn --warranty"
for details.

reading from file "remarks"
line 1, file <remarks>: square root of 2, 1.41421
line 8, file <remarks>: Stiffness properties, 1e+06, 1e+06, 1000, 1000
MBDyn terminated normally
user@host:~>$
```

Notice that in the example above the first expression is actually a string, as it was required up to version 1.6.1; this is no longer required.

2.4.10 Set

The `set` directive:

```
<card> ::= set : <math_expression> ;
```

This directive simply invokes the math parser to evaluate the expression `math_expression` and then discards the result.

It can be used to declare new variables, or to set the values of existing ones. This feature is very useful, since it allows to build parametric models, and to reuse generic model components.

Example.

```
set: integer LABEL = 1000;    # a label
set: real E = 210e+9;         # Pa, steel's elastic module
set: real A = 1e-4;           # m^2, a beam's cross section
set: real K = E*A;            # N, a beam's axial stiffness
```

2.4.11 Setenv

The `setenv` directive:

```
<card> ::= setenv :
    [ overwrite , { yes | no | (bool) <overwrite> } , ]
    " <varname> [ = <value> ] " ;
```

This directive sets the environment variable `varname` to `value`, if given; otherwise, the variable is unset. If `overwrite` is set, the variable is overwritten, if already set. By default, `overwrite` is set according to `setenv(3)`'s overwrite parameter (`no`).

```
# set FILE to "test", if it does not exist
setenv: "FILE=test";
# set FILE to "test", even if it exists
setenv: overwrite, yes, "FILE=test";
# unset FILE
setenv: "FILE";
# set FILE to the empty string, if it does not exist
setenv: "FILE=";
```

See `setenv(3)` and `unsetenv(3)` (or `putenv(3)`, if the former is not available) man pages for details.

2.4.12 Template Drive Caller

The `template drive caller` directive

```
<card> ::= template drive caller : <label> , { " <dim_name> " | (integer) <dim> } ,
    (TplDriveCaller<<Entity>>) <tpl_drive_caller> ;

<dim_name> ::= { 1 | 3 | 6 | 3x3 | 6x6 }

<dim> ::= { 1 | 3 | 6 }
```

allows to define a `template drive caller` (see Section 2.6.42) that can be subsequently reused. The type is defined according to

Entity	dim_name	dim
real	1	1
Vec3	3	3
Vec6	6	6
Mat3x3	3x3	
Mat6x6	6x6	

Template drive callers are useful essentially in two cases:

a) to define a `template drive` that will be used many times throughout a model;

b) to define a `template drive` that needs to be used in part of a model that is defined later, in order to make it parametric.

2.5 Node Degrees of Freedom

A node in MBDyn is an entity that owns public degrees of freedom and instantiates the corresponding public equations. It can lend them to other entities, called elements, to let them write contributions to public equations, possibly depending on the value of the public degrees of freedom.

Usually elements access nodal degrees of freedom through well-defined interfaces, at a high level. But in a few cases, nodal degrees of freedom must be accessed at a very low level, with the bare knowledge of the node label, the node type, the internal number of the degree of freedom, and the order of that degree of freedom (algebraic or differential, if any). The data that allows an entity to track a nodal degree of freedom is called `NodeDof`; it is read according to

```
<node_dof> :: = <node_label> ,
    <node_type>
    [ , <dof_number> ]
    [ , { algebraic | differential } ]
```

The label node_label and the type node_type of the node are used to track the pointer to the desired node.

If node_type refers to a non-scalar node type, the dof_number field is required to indicate the requested degree of freedom.

Finally, the order of the degree of freedom is read, if required. It must be one of algebraic or differential. If the dof_number degree of freedom is differential, both orders can be addressed, while in case of a node referring to an algebraic degree of freedom there is no choice, only the algebraic order can be addressed and thus this field is not required.

The dof_number must be between 1 and the number of degrees of freedom related to the node. Not all numbers might be valid for specific nodes; for example, dof_number values 4, 5, and 6 are not valid for a structural node, Section 6.5 when the order is algebraic.

When the node degree of freedom input syntax is used to address an equation (as in abstract force elements, Section 8.8.2, or in discrete control elements, Section 8.7.2), the distinction between algebraic and differential is meaningless, and thus this field is not required.

2.6 Drives and Drive Callers

Implements the type DriveCaller. Every time some entity can be "driven", i.e. a value can be expressed as dependent on some "external" input, an object of the class DriveCaller is used. The drive essentially represents a scalar function, whose value can change over time or, through some more sophisticated means, can depend on the state of the analysis. Usually, the dependence over time is implicitly assumed, unless otherwise specified. For example, the amplitude of the force applied by a force element (see Section 8.8) is defined by means of a drive; as such, the value of the drive is implicitly calculated as a function of the time. However, a dof drive (see Section 2.6.9) uses a subordinate drive to compute its value based on the value of a degree of freedom of the analysis; as a consequence, the value of the dof drive is represented by the value of the subordinate drive when evaluated as a function of that specific degree of freedom at the desired time (function of function).

The family of the DriveCaller object is very large. The type of the DriveCaller is declared as

```
<drive_caller> ::=
    { <drive_caller_type> [ , <arglist> ]
        | reference , <label> }
```

where arglist, if any, is a comma-separated list of arguments that depends on drive_caller_type. As an exception, a constant DriveCaller (that behaves exactly as a numerical constant with little or no overhead depending on the optimizing capability of the compiler) is assumed when a numeric value is used instead of a keyword.

If the alternative format is used, the keyword reference must be followed by the label of an already defined, valid drive caller (See Section 2.4.3).

Drive callers are listed in Table 2.8.

2.6.1 Array drive

```
<drive_caller> ::= array ,
    <num_drives> ,
        (DriveCaller) <drive_caller>
        [ , ... ]
```

this is simply a front-end for the linear combination of num_drives normal drives; num_drives must be at least 1, in which case a simple drive caller is created, otherwise an array of drive callers is created and at every call their value is added to give the final value of the array drive.

Table 2.8: Drive callers

Name	Differentiable	Notes
array	√	depends on subordinate drive callers
bistop		
closest next	√	depends on subordinate drive caller
const	√	
cosine	√	
cubic	√	
direct	√	
discrete filter		
dof		depends on subordinate drive caller
double ramp	√	
double step	√	
drive	√	depends on subordinate drive callers
element	√	depends on subordinate drive caller
exponential	√	
ginac	√	symbolic differentiation using GiNaC
file		
fourier series	√	
frequency sweep	√	depends on subordinate drive callers
linear	√	
meter	√	
mult	√	depends on subordinate drive callers
node	√	depends on subordinate drive caller
null	√	
parabolic	√	
periodic	√	depends on subordinate drive caller
piecewise linear	√	
ramp	√	
random	√	
sample and hold		
scalar function	√	depends on underlying scalar function
sine	√	
step	√	
step5	√	
string	√	
tanh	√	
time	√	
timestep	√	
unit	√	

2.6.2 Bistop drive

```
<drive_caller> ::= bistop ,
    [ initial status, { active | inactive } , ]
        (DriveCaller) <activation_condition> ,
        (DriveCaller) <deactivation_condition>
```

This drive caller returns 1.0 (TRUE) when its status is active and 0.0 (FALSE) when it is inactive. When in inactive status, it turns to active if the activation_condition is TRUE. When in active status, it turns to inactive if the deactivation_condition is TRUE.

This drive caller is useful to implement a "robust" and irreversible status change, for example the deactivation of something, as

```
bistop,
    initial status, active,
    # never re-activate
    null,
    # deactivate when the vertical component of node 1 velocity becomes negative
    string, "model::node::structural(1, \"XP[3]\") < 0"
```

2.6.3 Closest next drive

```
<drive_caller> ::= closest next ,
    <initial_time> ,
    { forever | <final_time> } ,
    (DriveCaller) <increment>
```

This drive returns a non-zero value when called for the first time with an argument greater that or equal to the current threshold value, which is computed starting from initial_time and incrementing it each time by as many increment values as required to pass the current value of Time. As soon as a threshold value is exceeded, as many values of increment as required to pass the current value of Time are added, and the process repeats.

This drive caller is useful within the output meter statement (see Section 5.3.9) to obtain nearly equally spaced output when integrating with variable time step.

2.6.4 Const(ant) drive

```
<drive_caller> ::= [ const , ] <const_coef>
```

The keyword const can be omitted, thus highlighting the real nature of this driver, that is completely equivalent to a constant, static real value.

2.6.5 Cosine drive

```
<drive_caller> ::= cosine ,
    <initial_time> ,
    <angular_velocity> ,
    <amplitude> ,
    { [ - ] <number_of_cycles> | half | one | forever } ,
    <initial_value>
```

where angular_velocity is $2\pi/T$. This drive actually computes a function of the type

$$f(t) = \texttt{initial\_value} + \texttt{amplitude} \cdot (1 - \cos(\texttt{angular\_velocity} \cdot (t - \texttt{initial\_time}))) . \quad (2.13)$$

The value of number_of_cycles determines the behavior of the drive. If it is positive, number_of_cycles oscillations are performed. If it is negative, the oscillations end after number_of_cycles$-1/2$ cycles at the top of the cosine, with null tangent. Special keywords can be used for number_of_cycles:

- forever: the oscillation never stops;
- one: exactly one cycle is performed (equivalent to number_of_cycles = 1);
- half: exactly half cycle is performed (equivalent to number_of_cycles = -1), so the function stops at

$$f = \texttt{initial\_value} + \begin{cases} 0 & \text{when } t \le \texttt{initial\_time} \\ \texttt{amplitude} \cdot (1 - \cos(\texttt{angular\_velocity} \cdot (t - \texttt{initial\_time}))) & \\ & \text{when } \texttt{initial\_time} < t < \texttt{initial\_time} + \dfrac{\pi}{\texttt{angular\_velocity}} \\ 2 \cdot \texttt{amplitude} & \text{when } \texttt{initial\_time} + \dfrac{\pi}{\texttt{angular\_velocity}} \le t \end{cases} \quad (2.14)$$

  starting and ending with continuous first derivative. In this case, it may be convenient to indicate the angular_velocity as $\pi/$duration, where duration is the time required to complete the half-wave.

Note that this drive caller does not correspond to $\cos(t)$.

Example.

```
# starts at Time = 1.5, duration = T, amplitude = A, a "bump" of height 2*A
..., cosine, 1.5, 2*pi/T, A, one, 0., ...

# starts at Time = .5, duration = T, amplitude = A, half "bump", remains at 2*A
..., cosine, .5, pi/T, A, half, 0., ...

# starts at Time = 1.2, frequency = Omega, amplitude = A, lasts forever
..., cosine, 1.2, Omega, A, forever, 0., ...
```

TODO: add a plot that exemplifies the three cases.

2.6.6 Cubic drive

```
<drive_caller> ::= cubic ,
    <const_coef> ,
    <linear_coef> ,
    <parabolic_coef> ,
    <cubic_coef>
```

The function

$$\begin{aligned} f(t) = & \texttt{const\_coef} \\ & + \texttt{linear\_coef} \cdot t \\ & + \texttt{parabolic\_coef} \cdot t^2 \\ & + \texttt{cubic\_coef} \cdot t^3. \end{aligned} \tag{2.15}$$

2.6.7 Direct drive

```
<drive_caller> ::= direct
```

Transparently returns the input value; the `arglist` is empty. It is useful in conjunction with those drive callers that require their output to be fed into another drive caller, like the `dof`, `node` and `element` drive callers, when the output needs to be used as is.

2.6.8 Discrete filter drive

```
<drive_caller> ::= discrete filter ,
    <n_a> [ , <a>_1 [ , ...] ] ,
    <b_0> ,
    <n_b> [ , <b>_1 [ , ...] ] ,
    (DriveCaller) <input_drive>
```

Filters the output of the ancillary drive caller `input_drive` according to the discrete filter coefficients.

- `n_a`: number of regression coefficients
- $\texttt{a}_i$: i-th regression coefficient, $i \in [1, \texttt{n\_a}]$
- `b_0`: direct transmission coefficient, must always be present; set to zero if not needed
- `n_b`: number of input coefficients
- $\texttt{b}_i$: i-th input coefficient, $i \in [0, \texttt{n\_b}]$; recall that b_0 is treated differently.

Formula:

$$y_k = \sum_{i=1}^{\texttt{n\_a}} \texttt{a}_i \, y_{k-i} + \sum_{i=0}^{\texttt{n\_b}} \texttt{b}_i \, u_{k-i} \tag{2.16}$$

The coefficients of the filter can be computed for example using Matlab, Python or similar tools. To obtain the coefficients of a 2nd order Butterworth filter, with cut frequency at 10 Hz, when the time step is `dt`, use

```
>> [B, A] = butter(2, 10./(1/(2*dt)));
>> a_1 = -A(2);
>> a_2 = -A(3);
>> b_0 = B(1);
>> b_1 = B(2);
>> b_2 = B(3);
```

The relative cutoff frequency (the second parameter in the `butter()` call) is the cutoff frequency of the filter divided by half the sampling frequency, namely $1/(2\,\texttt{dt})$. The corresponding MBDyn input is

```
...,
discrete filter,
    2, a_1, a_2,
    b_0,
    2, b_1, b_2,
    <input_drive> , ...
```

NOTE: requires fixed time step.

2.6.9 Dof drive

```
<drive_caller> ::= dof ,
    (NodeDof) <driving_dof> ,
    (DriveCaller) <func_drive>
```

a NodeDof, namely the reference to a degree of freedom of a node, is read. Then a recursive call to a drive data is read. The driver returns the value of the func_drive drive using the value of the NodeDof as input instead of the time. This can be used as a sort of explicit feedback, to implement fancy springs (where a force is driven through a function by the displacement of the node it is applied to) or an active control system; e.g.:

```
..., dof, 1000, structural, 3, algebraic,
    linear, 0., 1. ...
```

uses the value of the third component (z) of structural node 1000 as is (that is, in a linear expression with null constant coefficient and unit linear coefficient, while

```
..., dof, 1000, abstract, differential,
    string, "2.*exp(-100.*Var)" ...
```

uses the value of the derivative of abstract node 1000 in computing a string expression. Refer to the description of a NodeDof entry for further details.

The same effect can be obtained using the dof plugin as follows:

```
set: [dof,x,1000,structural,1,algebraic];
# ...
    ..., string, "2.*exp(-100.*x)" ...
```

which applies a couple whose amplitude is computed by evaluating a string drive which depends on variable x; this, in turn, is defined as a dof plugin, which causes its evaluation in terms of the selected degree of freedom of the node at each invocation.

2.6.10 Double ramp drive

```
<drive_caller> ::= double ramp ,
    <a_slope> ,
    <a_initial_time> ,
    <a_final_time> ,
    <d_slope> ,
    <d_initial_time> ,
    { forever | <d_final_time> } ,
    <initial_value>
```

The function

$$
\begin{aligned}
&\texttt{<f\_a>} ::= \texttt{a\_slope} \cdot (\texttt{a\_final\_time} - \texttt{a\_initial\_time}) \\
&\texttt{<f\_d>} ::= \texttt{d\_slope} \cdot (\texttt{d\_final\_time} - \texttt{d\_initial\_time}) \\
&f(t) = \texttt{initial\_value} \\
&\quad + \begin{cases}
0 & t < \texttt{a\_initial\_time} \\
\texttt{a\_slope} \cdot (t - \texttt{a\_initial\_time}) & \texttt{a\_initial\_time} \le t \le \texttt{a\_final\_value} \\
\texttt{f\_a} & \texttt{a\_final\_time} < t < \texttt{d\_initial\_time} \\
\texttt{f\_a} + \texttt{d\_slope} \cdot (t - \texttt{d\_initial\_time}) & \texttt{d\_initial\_time} \le t \le \texttt{d\_final\_value} \\
\texttt{f\_a} + \texttt{f\_d} & \texttt{d\_final\_time} < t
\end{cases}
\end{aligned}
\tag{2.17}
$$

2.6.11 Double step drive

```
<drive_caller> ::= double step ,
    <initial_time> ,
    <final_time> ,
    <step_value> ,
    <initial_value>
```

The function

$$
\begin{aligned}
&f(t) = \texttt{initial\_value} \\
&\quad + \begin{cases}
0 & t < \texttt{initial\_time} \\
\texttt{step\_value} & \texttt{initial\_time} \le t \le \texttt{final\_value} \\
0 & \texttt{final\_time} < t
\end{cases}
\end{aligned}
\tag{2.18}
$$

2.6.12 Drive drive

```
<drive_caller> ::= drive ,
    (DriveCaller) <drive_caller1> ,
    (DriveCaller) <drive_caller2>
```

This is simply a "function of function" drive: the output of `drive_caller2` is fed to `drive_caller1` and the result is returned. So the value of `drive_caller2` becomes the input argument to `drive_caller1`. Note that the very same thing occurs, for instance, in the `dof`, `node` and `element` drives, where the value of the dof or of the element's private datum, respectively, are fed into the given drive.

Example.

```
force: 1, abstract,
    1, abstract,
    drive,
```

```
        string, "Var*(Var>0.)",
        sine, 0., 2*pi/.2, 10., forever, 0.;
```

is equivalent to

```
set: real v;
force: 1, abstract,
    1, abstract,
    string, "v=sin(5.*2*pi*Time); 10.*v*(v>0)";
```

2.6.13 Element drive

```
<drive_caller> ::= element ,
    <label> ,
    <type> ,
    [ { string , " <name> " | index , <index> } , ] # index is deprecated
    (DriveCaller) <func_drive>
```

a reference to the private data of an element is read. This is made of: the element's `label`, the element's `type` and a specification of which private data are being referred; the `index` can be directly given, prepended by the keyword `index`, or the symbolic name `name` can be used, prepended by the keyword `string`. If that element allows only one private data, the specification can be omitted. Then a recursive call to a drive data is read. The driver returns the value of the `func_drive` `drive` using the value of the element's private data as input instead of the time. This can be used as a sort of explicit feedback, to implement fancy springs (where a force is driven through a function by the rotation of a joint) or an active control system; e.g.:

```
element, 1000, joint, string, "rz", direct
```

uses the value of the rotation about axis z of a revolute hinge as is (that is, in a linear expression with null constant coefficient and unit linear coefficient, while

```
element, 1000, joint, string, "rz", linear, 0., -60.
```

multiplies the value by -60;

```
element, 1000, joint, index, 1
    string, "2.*exp(-100.*Var)"
```

uses the same value, addressed in an alternative manner using the `index` form, to evaluate a string expression.

The same effect can be obtained using the `element` plugin as follows:

```
set: [element,x,1000,joint,string=rz];
# ...
couple: 1, conservative, 1, 0.,0.,1.,
    string, "2.*exp(-100.*x)";
```

which applies a couple whose amplitude is computed by evaluating a `string` drive which depends on the variable `x`; this, in turn, is defined as an `element` plugin, which causes its evaluation in terms of the element's private data at each invocation.

2.6.14 Exponential drive

```
<drive_caller> ::= exponential ,
    <amplitude_value> ,
    <time_constant_value> ,
    <initial_time> ,
    <initial_value>
```

This drive yields a function that resembles the response of a first-order system to a step input. Its value corresponds to `initial_value` for $t <$ `initial_time`. For $t \geq$ `initial_time`, it grows to `initial_value`+`amplitude_value` exponentially. The growth rate is governed by `time_constant_value`. If `time_constant_value` < 0, the function diverges.

$$f(t) = \texttt{initial\_value} + \texttt{amplitude\_value} \cdot \left(1 - \mathrm{e}^{\left(-\frac{t - \texttt{initial\_time}}{\texttt{time\_constant\_value}} \right)} \right) \tag{2.19}$$

Example.

```
..., exponential, 10.0, 0.2, 5.0, 0.0, ...
```

yields a function identical to the response of the system

$$H(s) = \frac{1}{1 + 0.2s} \tag{2.20}$$

to a step input occurring at time $t = 5.0$ with amplitude $A = 10.0$.

2.6.15 File drive

The `DriveCaller` is attached to a file drive object that must be declared and defined in the `drivers` section of the input file (see Section 7). As a consequence, `file` drive callers can only be instantiated after the `drivers` section of the input file.

```
<drive_caller> ::= file ,
    <drive_label>
    [ , { <column_number> | <user_defined> } ]
    [ , amplitude , <amplitude> ]

<user_defined> ::= <user_defined_type> [ , ...]
```

`drive_label` is the label of the `drive` the `DriveCaller` is attached to, while `column_number` is the number of the column the `DriveCaller` refers to (defaults to 1). An additional scaling factor `amplitude` can be used to rescale the drive value (defaults to 1.0).

2.6.16 Fourier series drive

```
<drive_caller> ::= fourier series ,
    <initial_time> ,
    <angular_velocity> ,
    <number_of_terms> ,
```

```
    <a_0> ,
    <a_1> , <b_1> ,
    [ ... , ]
{ <number_of_cycles> | one | forever } ,
<initial_value>
```

This drive corresponds to a Fourier series of fundamental angular velocity ω, truncated after n terms, over a given number of cycles P and starting at a given initial time t_0 as

$$f(t) = \frac{\texttt{a\_0}}{2} + \sum_{k=1}^{n} \left(\texttt{a}_k \cos\left(k\omega\left(t - t_0\right)\right) + \texttt{b}_k \sin\left(k\omega\left(t - t_0\right)\right)\right)$$

for

$$t_0 \leq t \leq t_0 + P\frac{2\pi}{\omega}$$

The value of $f(t)$, if defined, is added to `initial_value`. The value of `number_of_cycles` determines the behavior of the drive. If it is positive, `number_of_cycles` oscillations are performed. Special keywords can be used for `number_of_cycles`:

- `forever`: the oscillation never stops;
- `one`: exactly one cycle is performed;

the `number_of_terms` must be at least 1.

2.6.17 Frequency sweep drive

```
<drive_caller> ::= frequency sweep ,
    <initial_time> ,
    (DriveCaller) <angular_velocity_drive> ,
    (DriveCaller) <amplitude_drive> ,
    <initial_value> ,
    { forever | <final_time> } ,
    <final_value>
```

this drive recursively calls two other drives that supply the angular velocity and the amplitude of the oscillation. Any drive can be used.

$$\begin{aligned}
\omega(t) &= \texttt{angular_velocity_drive} \\
A(t) &= \texttt{amplitude_drive} \\
t_i &= \texttt{initial_time} \\
t_f &= \texttt{final_time}\ (\texttt{forever:}\ +\infty) \\
f_i &= \texttt{initial_value} \\
f_f &= \texttt{final_value} \\
f(t) &= \begin{cases} f_i & t < t_i \\ f_i + A(t)\sin\left(\omega(t)\left(t - t_i\right)\right) & t_i \leq t \leq t_f \\ f_f & t_f < t \end{cases}
\end{aligned} \tag{2.21}$$

2.6.18 GiNaC

```
<drive_caller> ::= ginac ,
    [ symbol , " <symbol> " , ]
    " <expression> "
```

The function `expression` is evaluated and differentiated, if needed, as a function of the variable passed as the optional `symbol`. If none is passed, `Var` is used. Currently there is no way to share the variables of the math parser used by the `string` drive caller. Not to be confused with the `string` drive caller.

Example.

```
..., ginac, "10*log(1 + Var)" ...
..., ginac, symbol, "x", "10*log(1 + x)" ...
```

2.6.19 Linear drive

```
<drive_caller> ::= linear ,
    <const_coef> ,
    <slope_coef>
```

The function

$$f(t) = \texttt{const\_coef} + \texttt{slope\_coef} \cdot t \tag{2.22}$$

2.6.20 Meter drive

The `meter` drive has value zero except for every `steps` steps, where it assumes unit value.

```
<drive_caller> ::= meter ,
    <initial_time> ,
    { forever | <final_time> }
    [ , steps , <steps_between_spikes> ]
```

the first optional entry, preceded by the keyword `steps`, sets the number of steps between spikes.

Example.

```
..., meter, 0., forever, steps, 10, ...
```

defines a drive whose value is zero except at time step 0. and every 10 time steps from that, where it assumes unit value.

2.6.21 Mult drive

The `mult` drive multiplies the value of two subordinate drives.

```
<drive_caller> ::= mult ,
    (DriveCaller) <drive_1> ,
    (DriveCaller) <drive_2>
```

2.6.22 Node drive

```
<drive_caller> ::= node ,
    <label> ,
    <type> ,
    [ { string , " <name> " | index , <index> } , ] # index is deprecated
    (DriveCaller) <func_drive>
```

a reference to the private data of a node is read. This is made of: the node's label, the node's type and a specification of which private data are being referred; the index can be directly given, prepended by the keyword index, or the symbolic name name can be used, prepended by the keyword string. If that node allows only one private data, the specification can be omitted. Then a recursive call to a drive data is read. The driver returns the value of the func_drive drive using the value of the node's private data as input instead of the time. This can be used as a sort of explicit feedback, to implement fancy springs (where a force is driven through a function by the rotation of a joint) or an active control system; e.g.:

```
..., node, 1000, structural, string, "X[3]",
    linear, 0., 1. ...
```

uses the value of the displacement in direction z (3) of a structural node, Section 6.5.5, as is (that is, in a linear expression with null constant coefficient and unit linear coefficient, see the linear drive, Section 2.6.19), while

```
..., node, 1000, structural, index, 3,
    string, "2.*exp(-100.*Var)" ...
```

uses the same value, addressed in an alternative manner, in computing a string expression.

2.6.23 Null drive

```
<drive_caller> ::= null
```

Zero valued; the arglist is empty.

2.6.24 Parabolic drive

```
<drive_caller> ::= parabolic ,
    <const_coef> ,
    <linear_coef> ,
    <parabolic_coef>
```

The function

$$\begin{aligned} f(t) = &\texttt{const\_coef} \\ &+ \texttt{linear\_coef} \cdot t \\ &+ \texttt{parabolic\_coef} \cdot t^2 \end{aligned} \tag{2.23}$$

2.6.25 Periodic drive

```
<drive_caller> ::= periodic ,
    <initial_time> ,
    <period> ,
    (DriveCaller)<func_drive>
```

The function

$$
\begin{aligned}
f(t) &= 0 && t < \texttt{initial\_time} \\
f(t) &= \texttt{func\_drive}\left(t - \texttt{initial\_time} - \texttt{period} \cdot \text{floor}\left(\frac{t - \texttt{initial\_time}}{\texttt{period}}\right)\right) && t \geq \texttt{initial\_time}
\end{aligned}
\tag{2.24}
$$

2.6.26 Piecewise linear drive

```
<drive_caller> ::= piecewise linear ,
    <num_points> ,
        <point> , <value>
        [ , ... ]
```

Piecewise linear function; the first and the last point/value pairs are extrapolated in case a value beyond the extremes is required. Linear interpolation between pairs is used.

2.6.27 Postponed drive

```
<drive_caller> ::= postponed ,
    <label>
```

This drive is actually a stub for a drive that cannot be defined because it occurs too early, when the data manager is not yet available. A drive caller with `label` must be defined before the drive caller is first used.

Example.

```
begin: initial value;
    # ...
    method: ms, postponed, 99;
    # ...
end: initial value;
# ...
begin: control data;
    # ...
end: control data;
# ...
drive caller: 99, ...
```

A drive caller that depends on the data manager must be used to define the spectral radius of the integration method. A reference to drive caller 99 is used. The drive caller labeled 99 is defined later, when all the required information is available.

2.6.28 Ramp drive

```
<drive_caller> ::= ramp ,
    <slope> ,
    <initial_time> ,
    { forever | <final_time> } ,
    <initial_value>
```

The function

$$f(t) = \texttt{initial\_value} + \begin{cases} 0 & t < \texttt{initial\_time} \\ \texttt{slope} \cdot (t - \texttt{initial\_time}) & \texttt{initial\_time} \leq t \leq \texttt{final\_value} \\ \texttt{slope} \cdot (\texttt{final\_time} - \texttt{initial\_time}) & \texttt{final\_time} < t \end{cases} \tag{2.25}$$

2.6.29 Random drive

```
<drive_caller> ::= random ,
    <amplitude_value> ,
    <mean_value> ,
    <initial_time> ,
    { forever | <final_time> }
    [ , steps , <steps_to_hold_value> ]
    [ , seed , { time | <seed_value> } ]
```

Generates pseudo-random numbers using the `rand(3)` call of the C standard library. Numbers are *uniformly distributed* in the interval [mean_value − amplitude_value, mean_value + amplitude_value).

The first optional entry, steps_to_hold_value, preceded by the keyword steps, sets the number of steps a random value must be held before generating a new random number. The second optional entry, preceded by the keyword seed, sets the new seed for the random number generator. A numeric seed_value value can be used, otherwise the keyword time uses the current time from the internal clock. A given seed can be used to ensure that two simulations use exactly the same random sequence (concurrent settings are not managed, so it is not very reliable).

2.6.30 Sample and hold drive

```
<drive_caller> ::= sample and hold ,
    (DriveCaller) <function> ,
    (DriveCaller) <trigger>
    [ , initial value , <initial_value> ]
```

When trigger is non-zero, the value of function is recorded after convergence at the end of the time step, and returned whenever the drive is called afterwards; when trigger is zero, the last recorded value is returned. When the optional keyword initial value is present, if the trigger is initially zero, the value initial_value is returned until trigger becomes non-zero.

2.6.31 Scalar function drive

```
<drive_caller> ::= scalar function , " <scalar_function_name> "
    [ , <scalar_function_definition> ]
```

The scalar function called scalar_function_name must exist; alternatively, it is defined inline according to scalar_function_definition.

Example.

```
scalar function: "this_is_a_test",
    multilinear,
        -20., 1.,
        -10., 0.1,
          0., 0.,
         10., 0.1,
         20., 1.;
 set: integer MY_DRIVE = 1000;
 drive caller: MY_DRIVE, scalar function, "this_is_a_test";
```

Alternative syntax example:

```
 set: integer MY_DRIVE = 1000;
 drive caller: MY_DRIVE, scalar function, "this_is_a_test",
    multilinear,
        -20., 1.,
        -10., 0.1,
          0., 0.,
         10., 0.1,
         20., 1.;
```

2.6.32 Sine drive

```
<drive_caller> ::= sine ,
    <initial_time> ,
    <angular_velocity> ,
    <amplitude> ,
    { [ - ] <number_of_cycles> | half | one | forever } ,
    <initial_value>
```

where angular_velocity is $2\pi/T$. This drive actually computes

$$f(t) = \texttt{initial\_value} + \texttt{amplitude} \cdot \sin\left(\texttt{angular\_velocity} \cdot (t - \texttt{initial\_time})\right) \tag{2.26}$$

The value of number_of_cycles determines the behavior of the drive. If it is positive, number_of_cycles$-1/2$ oscillations are performed. If it is negative, the oscillations end after number_of_cycles$-3/4$ cycles at the top of the sine, with null tangent. Special keywords can be used for number_of_cycles:

- forever: the oscillation never stops;
- one: exactly half period is performed (equivalent to number_of_cycles = 1);
- half: exactly a quarter of period is performed (equivalent to number_of_cycles = -1), so the

function stops at

$$f(t) = \texttt{initial\_value} + \begin{cases} 0 & \text{when } t < \texttt{initial\_time} \\ \texttt{amplitude} \cdot \sin\left(\texttt{angular\_velocity} \cdot (t - \texttt{initial\_time})\right) & \text{when } \texttt{initial\_time} \le t \le \texttt{initial\_time} + \dfrac{\pi}{2 \cdot \texttt{angular\_velocity}} \\ \texttt{amplitude} & \text{when } \texttt{initial\_time} + \dfrac{\pi}{2 \cdot \texttt{angular\_velocity}} \le t \end{cases} . \tag{2.27}$$

Note that this drive caller does not correspond to $\sin(t)$.

Example.

```
# starts at 1.5, duration = T, amplitude = A, a "bump" of height A
..., sine, 1.5, pi/T, A, one, 0., ...

# starts at .5, duration = T, amplitude = A, half "bump", remains at A
..., sine, .5, pi/(2*T), A, half, 0., ...

# starts at 1.2, frequency = Omega, amplitude = A, lasts forever
..., sine, 1.2, Omega, A, forever, 0., ...
```

2.6.33 Step drive

```
<drive_caller> ::= step ,
    <initial_time> ,
    <step_value> ,
    <initial_value>
```

The function

$$f(t) = \texttt{initial\_value} + \begin{cases} 0 & t < \texttt{initial\_time} \\ \texttt{step\_value} & \texttt{initial\_time} \le t \end{cases} \tag{2.28}$$

2.6.34 Step5 drive

```
<drive_caller> ::= step5 ,
    <initial_time> ,
    <initial_value> ,
    <final_time> ,
    <final_value>
```

The function

$$f(t) = \begin{cases} \texttt{initial\_value} & t \leq \texttt{initial\_time} \\ \texttt{initial\_value} + A\xi^3\left(10 - 15\xi + 6\xi^2\right) & \texttt{initial\_time} < t < \texttt{final\_time} \\ \texttt{final\_value} & \texttt{final\_time} \leq t \end{cases} \tag{2.29}$$

$$A = \texttt{final\_value} - \texttt{initial\_value}$$

$$\xi = \frac{t - \texttt{initial\_time}}{\texttt{final\_time} - \texttt{initial\_time}} \qquad 0 \leq \xi \leq 1$$

i.e. a continuous 5th order polynomial approximation of a step of amplitude A = final_value − initial_value and finite duration final_time − initial_time, starting at time initial_time with value initial_value and ending at time final_time with value final_value, with continuous 1st- and 2nd-order derivatives.

2.6.35 String drive

```
<drive_caller> ::= string ,
    " <expression_string> "
```

expression_string is a string, delimited by double quotes. It is parsed by the math parser when read during input, and evaluated every time the drive is invoked. The variable Time is kept up to date and can be used in the string to compute the return value, e.g.:

```
..., string, "e^(-Time)*cos(2.*pi*Time)" ...
```

generates a cosine modulated by an exponential. Another variable, Var, is set to the value provided by the caller in case the drive is called with an explicit argument as, for instance, in the dof drive (see Section 2.6.9); e.g.:

```
..., string, "e^(-Var)*cos(2.*pi*Time)" ...
```

Expressions may contain other variables and use functions defined in the available namespaces.

Note: originally, the expression was parsed each time it is evaluated; this is not very efficient. Starting from release 1.7.0, a tree-like form of the parsed expression is built, where non-symbolic branches are evaluated once for all, to speed up evaluation. That version can be compiled using the additional compilation flags: `./configure CPPFLAGS="-DUSE_EE=1" CXX="g++ -std=c++11"`

Note: all whitespace in the string is eaten up during input. For example

```
..., string, "1 \
    + cos(2 * pi * Time)", ...
```

is internally stored as

```
..., string, "1+cos(2*pi*Time)", ...
```

It is worth stressing that the expressions are dereferenced (i.e., their value is evaluated) when the drive caller is invoked during the analysis. So, if any variables that appear in the expressions are redefined during input parsing, and one wants the string drive caller to use their value at parsing time rather than run-time, the functions <type>_eval should be used, with

```
<type> ::= { bool | integer | real | string }
```

For example,

```
set: integer CURR_NODE = 0;
# ...
..., string, "model::distance(CURR_NODE, CURR_NODE+1)"
# ...
set: CURR_NODE = CURR_NODE + 100;
# ...
..., string, "model::distance(CURR_NODE, CURR_NODE+1)"
```

used repeatedly, modifying the variable `CURR_NODE` in between, would result in all instances of the string drive caller end up using the last value of `CURR_NODE` instead of the value that was current when the string drive caller was instantiated. Instead,

```
..., string, "integer_eval(model::distance(CURR_NODE, CURR_NODE+1))"
```

would use the value of `CURR_NODE` at the time the string was parsed.

2.6.36 Tanh drive

```
<drive_caller> ::= tanh ,
    <initial_time> ,
    <amplitude> ,
    <nd_slope> ,
    <initial_value>
```

This drive computes a function of the type

$$f(t) = \texttt{initial\_value} + \texttt{amplitude} \cdot \tanh\left(\texttt{nd\_slope} \cdot (t - \texttt{initial\_time})\right).$$

It is differentiable; its derivative is

$$\dot{f}(t) = \frac{\texttt{amplitude} \cdot \texttt{ns\_slope}}{\cosh^2\left(\texttt{nd\_slope} \cdot (t - \texttt{initial\_time})\right)}.$$

2.6.37 Time drive

```
<drive_caller> ::= time
```

Yields the current time; the arglist is empty.

2.6.38 Timestep drive

```
<drive_caller> ::= timestep
```

Yields the current timestep; the arglist is empty.

2.6.39 Unit drive

```
<drive_caller> ::= unit
```

Always 1; the `arglist` is empty.

2.6.40 Deprecated drive callers

- `one`; use `unit` instead.

2.6.41 Hints

Hints have been sponsored by Hutchinson CdR.

In some cases, during the analysis, different entities can be reinitialized as a consequence of some event, typically triggered by the waking up of a so-called `driven` element (See Section 8.20.2). The `DriveCaller` is re-parsed when the entity that owns it is sent a `hint` of the form

```
"drive{ (DriveCaller) <drive_caller> }"
```

is used, where `drive_caller` is an arbitrary `DriveCaller` specification. The whole hint argument needs to be enclosed in double quotes because MBDyn treats it as a string. It is not parsed until needed. At that point, the string (after removing the enclosing double quotes) is fed into the parser.

Typically, the use of a `hint` is needed when the specification of the drive caller parameters depends on the configuration of the system when the entity that uses it is waken up. For example, a `distance joint`, an element that enforces the distance between two nodes, may be waken up by some event. In the example reported below, the initial value of the `cosine` drive caller is computed by extracting the current distance between the nodes at the time the element is waken up:

```
driven: 1, string, "Time > 10.",
    hint, "drive{cosine, 10., 2., .25, half, model::distance_prev(10, 20)}",
joint: 1, distance,
    10,
    20,
    const, 1; # this length is not actually used,
              # since the element is inactive until "Time > 10.";
              # subsequently, the length is replaced by the hint
```

Note that the function `distance_prev` (from the built-in `model` namespace) is used instead of the function `distance`. This is needed because hints are parsed *after* the prediction of the state of the model for a subsequent new time step, i.e. they extract information from the model *after* its current state has been modified by the prediction for the next time step. As a consequence, unless an exactly steady state was reached for a sufficient number of steps to result in a prediction that does not change the state of that part of model, unexpected values could result from a call to a function that returns information based on the *current* state of the model. On the contrary, to access the last converged state of the model within a `hint` one needs to call `model` namespace functions that refer to the *previous* time step. See Section 2.2.1 for further details on the functions of the `model` namespace.

2.6.42 Template Drive

The template drive caller, indicated as `TplDriveCaller<<Entity>>`, is a higher-dimensional `DriveCaller`, whose dimensions are those of `<Entity>`. The family of the `TplDriveCaller<<Entity>>` object is very large. A `TplDriveCaller<<Entity>>` is declared as

```
<tpl_drive_caller> ::=
    { <tpl_drive_caller_type> [ , <arglist> ]
        | reference , <label> }
```

where `arglist`, if any, is a comma-separated list of arguments that depends on `tpl_drive_caller_type`.

If the alternative format is used, the keyword `reference` must be followed by the label of an already defined, valid template drive caller (See Section 2.4.12).

There are four types of template drive callers:

- the `null` template drive caller:

  ```
  <tpl_drive_caller> ::= null
  ```

  a shortcut for nothing;

- the `single` template drive caller:

  ```
  <tpl_drive_caller> ::= single ,
      (DriveCaller) <drive_caller> # in case <Entity> is scalar
      (<Entity>) <entity> , (DriveCaller) <drive_caller> # otherwise
  ```

  corresponding to the expression `entity`·`drive_caller`, where `entity` is a constant of the expected type (3×1 vector, 6×1 vector, 3×3 matrix, 6 matrix are the types currently defined, but since a C++ template has been used, the implementation of new ones is straightforward). If `<Entity>` is a scalar, then `entity` is assumed to be 1.

> The keyword `single` can be omitted; however, this practice is not recommended, since in some contexts such keyword is required to disambiguate the syntax.

In some contexts (e.g. when specifying the direction of a force), in which the `entity` is a `Vec3`, it can be prefixed by an orientation modifier in the form of the specification of a reference frame in which `entity` is expressed; for example,

```
single, 1., 0., 0.                    # in the default context-specific reference
single, reference, 99, 1., 0., 0.      # in reference frame 99
single, reference, global, 1., 0., 0. # in the global reference frame
```

Example. The following example implements a `Vec3` template drive having a constant unitary value in the X direction.

```
single,
    1., 0., 0.,
    const, 1.;
```

The following example implements a `Mat3x3` template drive that is `sine`-valued along the diagonal.

```
single,
    eye,                                # identity matrix
    sine, 0., 2*pi, 1., forever, 0.;    # sine drive caller
```

- the `component` template drive caller:

  ```
  <tpl_drive_caller> ::= component , [ <type> , ]
      <component_drive_caller>
      [ , ... ]
      # as many component_drive_caller as the dimensionality of entity

  <type> ::= { sym | diag }

  <component_drive_caller> ::= { inactive | (DriveCaller) <drive_caller> }
  ```

where each element consists of the corresponding drive caller; the optional `type` is used with matrix-type entities to specify special matrix layouts:

- `sym` indicates that only the upper triangular components are expected;
- `diag` indicates that only the diagonal components are expected;

When `<Entity>` is a scalar, `component` is treated as `single`.

Example. The following example implements a `Vec3` template drive having constant value in the X direction, and a `sine` drive in the Z direction.

```
component,
    const, 1.,                          # X: 1
    inactive,                           # Y: 0
    sine, 0., 2*pi, 1., forever, 0.;    # Z: sine
```

- the `array` template drive caller:

```
<tpl_drive_caller> ::= array ,
    <num_template_drive_callers> ,
        <tpl_drive_caller>
        [ , ... ]
        # num_template_drive_callers instances of tpl_drive_caller
```

which linearly combines a set of arbitrary template drive callers.

In case `<Entity>` is a scalar, a normal array drive caller is actually instantiated, so no overhead is added.

In the case of the `component` template drive caller, as many instances of a `DriveCaller` as the dimensionality of the template drive caller are expected.

At least one `tpl_drive_caller` is expected in the case of the `array` template drive caller. If `num_template_drive_callers` is exactly 1, only a single template drive caller is actually constructed, thus avoiding the overhead related to handling the template drive caller array.

Hints

The template drive can be re-parsed as a consequence of a `hint` (see Section 2.6.41). In this case, the syntax of the `hint` is:

```
"drive<n>{ <tpl_drive_caller> }"
```

where `n` is the dimension of `entity`.

For example, a `TplDriveCaller<Vec3>` drive hint is:

```
"drive3{single, 1., 0., 0., sine, 0., 2*pi, 1., forever, 0.}"
```

which corresponds to a 3×1 vector `1., 0., 0.` that multiplies a `sine` drive caller with the parameters illustrated above.

A `TplDriveCaller<Mat3x3>` drive hint is:

```
"drive3x3{single, eye, sine, 0., 2*pi, 1., forever, 0.}"
```

which corresponds to a 3×3 identity matrix (the keyword `eye`) that multiplies a `sine` drive caller with the parameters illustrated above.

2.7 Scalar functions

A ScalarFunction object computes the value of a function. Almost every scalar function is of type DifferentiableScalarFunction, derived from ScalarFunction, and allows to compute the derivatives of the function as well. Currently implemented scalar functions are

- const: $f(x) = c$
- exp: $f(x) = m \cdot b^{c \cdot x}$
- log: $f(x) = m \cdot \log_b(c \cdot x)$
- pow: $f(x) = x^p$
- sin: $f(x) = A \cdot \sin(\omega \cdot x + \theta)$
- cos: $f(x) = A \cdot \cos(\omega \cdot x + \theta)$
- linear: linear interpolation between the two points (x_1, y_1) and (x_2, y_2)
- cubicspline: cubic natural spline interpolation between the set of points $\{(x_i, y_i), i \in [1, k \geq 3]\}$
- multilinear: multilinear interpolation between the set of points $\{(x_i, y_i), i \in [1, k \geq 2]\}$
- chebychev: Chebychev interpolation between the set of points $\{a, b\}$
- sum: $f(x) = f_1(x) + f_2(x)$
- sub: $f(x) = f_1(x) - f_2(x)$
- mul: $f(x) = f_1(x) \cdot f_2(x)$
- div: $f(x) = f_1(x)/f_2(x)$
- powfun: $f(x) = f_1(x)^{f_2(x)}$

Every ScalarFunction card follows the format

```
<card> :: = scalar function : " <unique_scalar_func_name> " ,
    <scalar_func_type> ,
    <scalar_func_args>
```

The name of the scalar function is a string, thus it needs to be enclosed in double quotes.

The type of scalar function, scalar_func_type, together with relevant arguments, scalar_func_args, are as follows:

Const Scalar Function

```
<scalar_func_type> ::= const

<scalar_func_args> ::= <const_coef>
```

Note: if the scalar_func_type is omitted, a const scalar function is assumed.

Example.

```
scalar function: "const_function", const, 1.e-2;
```

Exp Scalar Function

```
<scalar_func_type> ::= exp

<scalar_func_args> ::=
    [ base , <base> , ]
    [ coefficient , <coef> , ]
    <multiplier_coef>
```

$$f(x) = \texttt{multiplier\_coef} \cdot \texttt{base}^{(\texttt{coef} \cdot x)} \tag{2.30}$$

Note: the optional `base` must be positive (defaults to 'e', Neper's number); `coef` defaults to 1.

Example.

```
scalar function: "exp_function", exp, 1.e-2; # 1.e-2*e^x
```

Log Scalar Function

```
<scalar_func_type> ::= log

<scalar_func_args> ::=
    [ base , <base> , ]
    [ coefficient , <coef> , ]
    <multiplier_coef>
```

$$f(x) = \texttt{multiplier\_coef} \cdot \log_{\texttt{base}}(\texttt{coef} \cdot x) \tag{2.31}$$

Note: the optional `base` (defaults to 'e', Neper's number, resulting in natural logarithms) must be positive. The optional value `coef`, prepended by the keyword `coefficient`, (defaults to 1). The argument must be positive, otherwise an exception is thrown.

Example.

```
scalar function: "log_function", log, 1.e-2; # 1.e-2*log(x)
```

Pow Scalar Function

```
<scalar_func_type> ::= pow

<scalar_func_args> ::= <exponent_coef>
```

$$f(x) = x^{\texttt{exponent\_coef}} \tag{2.32}$$

Example.

```
scalar function: "pow_function", pow, 3.; # x^3.
```

Sine Scalar Function

```
<scalar_func_type> ::= sin

<scalar_func_args> ::=
    <amplitude_coeff>,
    <angular_frequency>,
    <phase_offset>
```

$$f(x) = \texttt{amplitude\_coeff} \cdot \sin(\texttt{angular\_frequency} \cdot x + \texttt{phase\_offset}) \tag{2.33}$$

Note: The value phase_offset is in radians and angular_frequency is in rad·s^{-1}.

Cosine Scalar Function

```
<scalar_func_type> ::= cos

<scalar_func_args> ::=
    <amplitude_coeff>,
    <angular_frequency>,
    <phase_offset>
```

$$f(x) = \texttt{amplitude\_coeff} \cdot \cos(\texttt{angular\_frequency} \cdot x + \texttt{phase\_offset}) \tag{2.34}$$

Note: The value phase_offset is in radians and angular_frequency is in rad·s^{-1}.

Linear Scalar Function

```
<scalar_func_type> ::= linear

<scalar_func_args> ::= <point> , <point>

<point>            ::= <x>, <y>
```

A line passing through the two points,

$$f(x) = \mathtt{y}_i + \frac{\mathtt{y}_f - \mathtt{y}_i}{\mathtt{x}_f - \mathtt{x}_i}(x - \mathtt{x}_i) \tag{2.35}$$

where $\mathtt{x}_i$, $\mathtt{y}_i$ are the coordinates of the first point, while $\mathtt{x}_f$, $\mathtt{y}_f$ are the coordinates of the second point.

Example.

```
scalar function: "linear_function", linear, 0., 0., 1., 1.;
```

Cubic Natural Spline Scalar Function

```
<scalar_func_type> ::= cubic spline

<scalar_func_args> ::= [ do not extrapolate , [ bailout , ] ]
    <point> ,
    <point>
    [ , ... ]
    [ , end ]

<point>            ::= <x> , <y>
```

The `end` delimiter is required if the card needs to continue (i.e. the `ScalarFunction` definition is embedded in a more complex statement); it can be omitted if the card ends with a semicolon right after the last point.
Unless `do not extrapolate` is set, when the input is outside the provided values of `x` the value is extrapolated. If `do not extrapolate` is set and `bailout` is *not* set then the scalar function returns the value of the nearest point. If, instead, `bailout` is set then MBDyn will terminate the simulation.

Multilinear Scalar Function

```
<scalar_func_type> ::= multilinear

<scalar_func_args> ::= [ do not extrapolate , [ bailout , ] ]
    <point> ,
    <point>
    [ , ... ]
    [ , end ]

<point>            ::= <x> , <y>
```

Unless `do not extrapolate` is set, when the input is outside the provided values of `x` the value is extrapolated using the slope of the nearest point pair. If `do not extrapolate` is set and `bailout` is *not* set then the scalar function returns the value of the nearest point. If, instead, `bailout` is set then MBDyn will terminate the simulation.

Chebychev Scalar Function

```
<scalar_func_type> ::= chebychev

<scalar_func_args> ::=
    <lower_bound> , <upper_bound> ,
    [ do not extrapolate , ]
    <coef_0> [ , <coef_1> [ , ... ] ]
    [ , end ]
```

Chebychev polynomials of the first kind are defined as

$$T_n(\xi) = \cos\left(n \cos^{-1}(\xi)\right), \tag{2.36}$$

with

$$\xi = 2\frac{x}{b-a} - \frac{b+a}{b-a}, \tag{2.37}$$

where $a =$ lower_bound and $b =$ upper_bound, which corresponds to the series

$$T_0(\xi) = 1 \tag{2.38}$$
$$T_1(\xi) = \xi \tag{2.39}$$
$$\dots$$
$$T_n(\xi) = 2\xi T_{n-1}(\xi) - T_{n-2}(\xi). \tag{2.40}$$

For example, the first five coefficients are

$$T_0(\xi) = 1 \tag{2.41}$$
$$T_1(\xi) = \xi \tag{2.42}$$
$$T_2(\xi) = 2\xi^2 - 1 \tag{2.43}$$
$$T_3(\xi) = 4\xi^3 - 3\xi \tag{2.44}$$
$$T_4(\xi) = 8\xi^4 - 8\xi^2 + 1 \tag{2.45}$$

This scalar function implements the truncated series in the form

$$f(x) = \sum_{k=0,n} c_k T_k(\xi), \tag{2.46}$$

where $c_k =$ coef_<k>. The first derivative of the series is obtained by considering

$$\frac{\mathrm{d}}{\mathrm{d}\xi} T_0(\xi) = 0 \tag{2.47}$$
$$\frac{\mathrm{d}}{\mathrm{d}\xi} T_1(\xi) = 1 \tag{2.48}$$
$$\dots$$
$$\frac{\mathrm{d}}{\mathrm{d}\xi} T_n(\xi) = 2T_{n-1}(\xi) + 2\xi \frac{\mathrm{d}}{\mathrm{d}\xi} T_{n-1}(\xi) - \frac{\mathrm{d}}{\mathrm{d}\xi} T_{n-2}(\xi), \tag{2.49}$$

so the first derivative of the scalar function is

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \frac{\mathrm{d}\xi}{\mathrm{d}x} \sum_{k=1,n} c_k \frac{\mathrm{d}}{\mathrm{d}\xi} T_k(\xi). \tag{2.50}$$

Subsequent derivatives follow the rule

$$\frac{\mathrm{d}^i}{\mathrm{d}\xi^i} T_n(\xi) = 2i \frac{\mathrm{d}^{i-1}}{\mathrm{d}\xi^{i-1}} T_{n-1}(\xi) + 2\xi \frac{\mathrm{d}^i}{\mathrm{d}\xi^i} T_{n-1}(\xi) - \frac{\mathrm{d}^i}{\mathrm{d}\xi^i} T_{n-2}(\xi). \tag{2.51}$$

Differentiation of order higher than 1 is not currently implemented.

Sum Scalar Function

<scalar_func_type> ::= sum

<scalar_func_args> ::= (ScalarFunction) <f1> , (ScalarFunction) <f2>

$$f(x) = \mathrm{f1}(x) + \mathrm{f2}(x) \tag{2.52}$$

Example.

```
scalar function: "first_function", const, 1.;
scalar function: "second_function", const, 2.;
scalar function: "sum_function", sum, "first_function", "second_function";
```

Sub Scalar Function

```
<scalar_func_type> ::= sub

<scalar_func_args> ::= (ScalarFunction) <f1> , (ScalarFunction) <f2>
```

$$f(x) = \texttt{f1}(x) - \texttt{f2}(x) \tag{2.53}$$

Example.

```
scalar function: "first_function", const, 1.;
scalar function: "second_function", const, 2.;
scalar function: "sub_function", sub, "first_function", "second_function";
```

Mul Scalar Function

```
<scalar_func_type> ::= mul

<scalar_func_args> ::= (ScalarFunction) <f1> , (ScalarFunction) <f2>
```

$$f(x) = \texttt{f1}(x) \cdot \texttt{f2}(x) \tag{2.54}$$

Example.

```
scalar function: "first_function", const, 1.;
scalar function: "second_function", const, 2.;
scalar function: "mul_function", mul, "first_function", "second_function";
```

Div Scalar Function

```
<scalar_func_type> ::= div

<scalar_func_args> ::= (ScalarFunction) <f1> , (ScalarFunction) <f2>
```

$$f(x) = \texttt{f1}(x) / \texttt{f2}(x) \tag{2.55}$$

Note: division by zero is checked, and an exception is thrown.

Example.

```
scalar function: "first_function", const, 1.;
scalar function: "second_function", const, 2.;
scalar function: "div_function", div, "first_function", "second_function";
```

Powfun Scalar Function

```
<scalar_func_type> ::= powfun

<scalar_func_args> ::= (ScalarFunction) <f1> , (ScalarFunction) <f2>
```

$$f(x) = \texttt{f1}(x)^{\texttt{f2}(x)} \tag{2.56}$$

Example.

```
scalar function: "first_function", const, 1.;
scalar function: "second_function", const, 2.;
scalar function: "powfun_function", powfun, "first_function", "second_function";
```

2.8 Friction

A friction-based element needs the definition of at least a `friction model` and of a `shape function`.

2.8.1 1D Friction models

The one dimensional `friction model` input format is:

```
<friction_model> ::= <friction_type> ,
    <friction_arglist>
```

where `friction_function` gives the static friction as a function of sliding velocity.

Currently implemented friction models are:

1. `modlugre`
   This friction model is based on the one presented in [2] by Pierre Dupont, Vincent Hayward, Brian Armstrong and Friedhelm Altpeter, known as 'LuGre'. The input format is:

   ```
   <friction_type> ::= modlugre

   <friction_arglist> ::=
       <sigma0> ,
       <sigma1> ,
       <sigma2> ,
       <kappa> ,
       (ScalarFunction) <friction_function>
   ```

2. `discrete coulomb`
   This is a Coulomb model with viscous friction and internal states to resolve stick/slip conditions. This model, not making use of a cone complementary problem solver, is *experimental* and may lead to spurious convergence issue. Its use is *discouraged*.

   ```
   <friction_model> ::= discrete coulomb

   <friction_arglist> ::=
       (ScalarFunction) <friction_function>
       [ , sigma2 , <sigma2> ]
       [ , velocity ratio , <vel_ratio> ]
   ```

where sigma2 gives the viscous friction; vel_ratio defaults to 0.8, and is used to discriminate stick/slip conditions. Note that discrete coulomb cannot handle stick conditions when the reaction force in the constraint goes to zero (resulting in a singular Jacobian matrix). The preload parameter needs to be used to make sure the joint is preloaded as appropriate (see the documentation of the specific joint for details).

2.8.2 2D Friction models

Two dimensional friction models, friction model 2D are required whenever the sliding direction can change, as when sliding on a plane, or with a spherical hinge. The general syntax is:

```
<friction_model_2D> ::= <friction_type_2D> ,
    <friction_arglist_2D>
```

where friction_function gives the static friction as a function of sliding velocity.

Currently implemented friction models are:

1. modlugre 2D

   The modlugre 2D model is based on [3], but does not model the moment related to the rotation around the contact point normal. In other words,it assumes pure sliding, without relative rotations, at the point of contact. It is recommended to set sigma2 to a zero.

   The input format is:

   ```
   <friction_type_2D> ::= modlugre 2D

   <friction_arglist_2D> ::=
       <sigma0> ,
       <sigma1> ,
       <sigma2> ,
       <kappa> ,
       (ScalarFunction) <friction_function>
   ```

   where arguments have the same meaning those defined for the modlugre one dimensional friction model.

2. discrete coulomb 2D
   This is a Coulomb model with viscous friction and internal states to resolve stick/slip conditions. Just like the 1D Coulomb model, this model does not use a cone complementary problem solver (while it should), thus is *experimental* and may lead to spurious convergence issue. Its use is *strongly discouraged*.

   ```
   <friction_model> ::= discrete coulomb 2D

   <friction_arglist> ::=
       (ScalarFunction) <friction_function>
       [ , sigma2 , <sigma2> ]
       [ , velocity ratio , <vel_ratio> ]
   ```

   where sigma2 gives the viscous friction; vel_ratio defaults to 0.8, and is used to discriminate stick/slip conditions. Note that discrete coulomb 2D cannot handle stick conditions when the reaction force in the constraint goes to zero (resulting in a singular Jacobian matrix). The preload parameter needs to be used to make sure the joint is preloaded as appropriate (see the documentation of the specific joint for details).

2.8.3 Shape functions

Implements the type `shape function`. The input format of shape functions is:

```
<shape_function> ::= <shape_function_type>
    [ , <shape_function_arglist> ]
```

1D shape functions

Currently implemented shape functions for `friction model 1D` are:

1. `simple`
   This shape function is equal to one. It does not need arguments.

   ```
   <shape_function_type> ::= simple

   <shape_function_arglist> ::=
   ```

2. `simple plane hinge`

   ```
   <shape_function_type> ::= simple plane hinge

   <shape_function_arglist> ::=
   ```

   This is the shape function of a `revolute hinge`; the radius will be prescribed from the joint; the `revolute hinge` is assumed to be subject to small loads.

3. `screw joint`

   ```
   <shape_function_type> ::= screw joint

   <shape_function_arglist> ::=
       radius, <average_radius_value>,
       half thread angle, <half_thread_angle_value>
   ```

   This is the shape function of a `screw` joint. With reference to Fig. 2.3 the `radius` r is the average contact radius required to compute the friction-induced moment, while the `half thread angle` angle γ should be equal to half of the thread angle.

2D shape functions

Two dimensional shape functions,

```
<shape_function_2D> ::= <shape_function_type_2D>
    [ , <shape_function_arglist_2D> ]
```

mat be required together with a `friction model 2D`. The `simple 2D` shape function syntax is:

```
<shape_function_type_2D> ::= simple 2D

<shape_function_arglist_2D> ::=
```

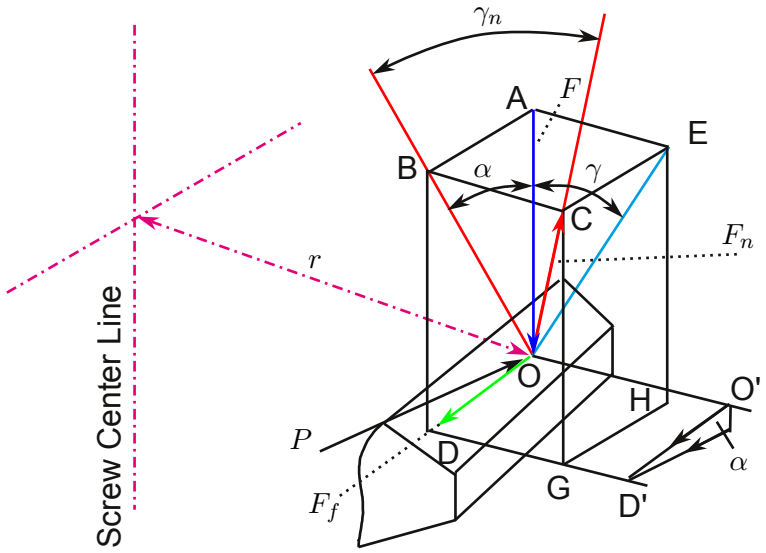


Figure 2.3: 3D screw thread sketch.

2.9 Shapes

The Shape entities are objects that return a value depending on one (or two, for 2D shapes) dimensionless abscissa, ranging $[-1, 1]$. At present, only 1D shapes are used, by aerodynamic elements. A Shape input format is:

```
<shape_1D> ::= <shape_type> ,
    <shape_arglist>
```

The shapes currently available are:

1. const

```
<shape_type> ::= const

<shape_arglist> ::= <const_value>
```

$$f = \texttt{const\_value} \tag{2.57}$$

2. piecewise const

```
<shape_type> ::= piecewise const

<shape_arglist> ::=
    <number_of_points> ,
        <abscissa> , <value>
        [ , ... ]
```

$$f(\xi) = \texttt{value}_1 \qquad \xi \leq \texttt{abscissa}_1 \tag{2.58a}$$
$$f(\xi) = \texttt{value}_i \qquad \texttt{abscissa}_i \leq \xi < \texttt{abscissa}_{i+1} \tag{2.58b}$$
$$f(\xi) = \texttt{value}_N \qquad \texttt{abscissa}_N \leq \xi \tag{2.58c}$$

3. `linear`

```
<shape_type> ::= linear

<shape_arglist> ::=
    <value_at_-1> ,
    <value_at_1>
```

$$f(\xi) = \frac{\texttt{value\_at\_-1} + \texttt{value\_at\_1}}{2} + \frac{\texttt{value\_at\_-1} - \texttt{value\_at\_1}}{2}\xi \tag{2.59}$$

4. `piecewise linear`

```
<shape_type> ::= piecewise linear

<shape_arglist> ::=
    <number_of_points> ,
        <abscissa> , <value>
        [ , ... ]
```

$$f(\xi) = \texttt{value}_1 \qquad \xi \leq \texttt{abscissa}_1 \tag{2.60a}$$
$$f(\xi) = \texttt{value}_i \frac{\texttt{abscissa}_{i+1} - \xi}{\texttt{abscissa}_{i+1} - \texttt{abscissa}_i} + \texttt{value}_{i+1} \frac{\xi - \texttt{abscissa}_i}{\texttt{abscissa}_{i+1} - \texttt{abscissa}_i} \qquad \texttt{abscissa}_i \leq \xi < \texttt{abscissa}_{i+1} \tag{2.60b}$$
$$f(\xi) = \texttt{value}_N \qquad \texttt{abscissa}_N \leq \xi \tag{2.60c}$$

5. `parabolic`

```
<shape_type> ::= parabolic

<shape_arglist> ::=
    <value_at_-1> ,
    <value_at_0> ,
    <value_at_1>
```

$$f(\xi) = \texttt{value\_at\_-1} \cdot \xi\,(\xi - 1) + \texttt{value\_at\_0} \cdot \left(1 - \xi^2\right) + \texttt{value\_at\_1} \cdot \xi\,(\xi + 1) \tag{2.61}$$

This form of input has been chosen since, being the shapes mainly used to interpolate values, it looks more "natural" to insert the mapping values at characteristic points. For `piecewise linear` shapes, there must be `number_of_points` pairs of abscissæ and values; abscissæ must be in the range $[-1, 1]$, in strict ascending order.

2.10 Constitutive Laws

Implements the type ConstitutiveLaw<<Entity>, <DerivativeOfEntity>>. Every time a "deformable" entity requires a constitutive law, a template constitutive law is read. This has been implemented by means of C++ templates in order to allow the definition of a general constitutive law when possible. The "deformable" elements at present are:

- rod and genel spring and related elements (1D);
- deformable hinge and deformable displacement joint elements (3D);
- deformable joint, beam and solid elements (6D).
- solid elements for incompressible constitutive laws (7D).
- solid elements for constitutive laws subject to finite strain (9D).

Constitutive laws are also used in non-structural components, to allow some degree of generality in defining input/output relationships. Some constitutive laws are meaningful only when related to some precise dimensionality. In some special cases, general purpose (genel) elements use 1D constitutive laws to express an arbitrary dependence of some value on a scalar state of the system. Table 2.9 shows the availability of each constitutive law.

The meaning of the input and output parameters of a constitutive law is dictated by the entity that uses it. In general, the user should refer to the element the constitutive law is being instantiated for, in order to understand what the input and the output parameters are supposed to be.

Usually, constitutive laws can be directly defined when required, according to the definition of an element. However, the special card described in Section 2.4.1 allows to define constitutive laws stand-alone, and attach them to the elements by means of the following mechanism:

```
<constitutive_law> ::=
    { <constitutive_law_definition> | reference , <label> }
```

where <constitutive_law_definition> is described in the following, while <label> is the label of a previously defined constitutive law of the appropriate dimensionality, as described in Section 2.4.1.

The constitutive laws are entered as follows:

```
<constitutive_law_definition> ::= <specific_const_law>
    [ , prestress , (Entity) <prestress> ]
    [ , prestrain , (TplDriveCaller<<Entity>>) <prestrain> ]

<specific_const_law> ::= <const_law_name> ,
    <const_law_data>
```

where const_law_name is the name of the constitutive law and const_law_data depends on the specific constitutive law. The latter fields, whose type depends on the dimensionality of the constitutive law, are optional, under the assumption that the constitutive law is the last portion of a card, or that any ambiguity can be avoided. The data specific to the currently available constitutive laws must be entered as follows:

2.10.1 Linear elastic, linear elastic isotropic

```
<specific_const_law> ::= linear elastic [ isotropic ] ,
    (real) <stiffness>
```

the isotropic stiffness coefficient; the word isotropic can be omitted, essentially because it has no meaning for scalar constitutive laws.

Table 2.9: Constitutive laws dimensionality

Constitutive law	1D	3D	6D	7D	9D
linear elastic, linear elastic isotropic	√	√	√		
linear elastic generic	√	√	√		
linear elastic generic axial torsion coupling			√		
cubic elastic generic	√	√			
inverse square elastic	√				
log elastic	√				
linear elastic bistop	√	√	√		
double linear elastic	√	√			
isotropic hardening elastic	√	√	√		
scalar function elastic, scalar function elastic isotropic	√	√	√		
scalar function elastic orthotropic	√	√	√		
linear viscous, linear viscous isotropic	√	√	√		
linear viscous generic	√	√	√		
linear viscoelastic, linear viscoelastic isotropic	√	√	√		
linear viscoelastic generic	√	√	√		
linear time variant viscoelastic generic	√	√	√		
linear viscoelastic generic axial torsion coupling			√		
cubic viscoelastic generic	√	√			
double linear viscoelastic	√	√			
turbulent viscoelastic	√				
linear viscoelastic bistop	√	√	√		
shock absorber	√				
symbolic elastic	√	√	√		
symbolic viscous	√	√	√		
symbolic viscoelastic	√	√	√		
ann elastic	√	√	√		
ann viscoelastic	√	√	√		
nlsf elastic	√	√	√		
nlsf viscous	√	√	√		
nlsf viscoelastic	√	√	√		
nlp elastic	√	√	√		
nlp viscous	√	√	√		
nlp viscoelastic	√	√	√		
hookean linear elastic isotropic			√	√	
hookean linear viscoelastic isotropic			√		
neo hookean elastic			√		
neo hookean viscoelastic			√		
mooney rivlin elastic			√	√	√
bilinear isotropic hardening			√	√	
linear viscoelastic maxwell 1		√	√		
linear viscoelastic maxwell n		√	√		
mfront small strain			√		
mfront finite strain					√

Constitutive law wrapper	1D	3D	6D	7D	9D
array (wrapper)	√	√	√		
axial (wrapper)		√			
bistop (wrapper)	√	√	√		
drive caller (wrapper)	√	√	√		
invariant angular (wrapper)		√			

Example.

```
constitutive law: 1, name, "scalar isotropic law",
    1, linear elastic, 1.e9;
constitutive law: 2, name, "3D isotropic law",
    3, linear elastic isotropic, 1.e9;
constitutive law: 3, name, "6D isotropic law",
    6, linear elastic isotropic, 1.e9;
```

2.10.2 Linear elastic generic

```
<specific_const_law> ::= linear elastic generic ,
    (DerivativeOfEntity) <stiffness>
```

the stiffness matrix. In case of 1D, the type is scalar, and there is no distinction between `generic` and `isotropic`, while, in case of $N \times 1$ vectors, the type is the corresponding $N \times N$ matrix.

Example.

```
constitutive law: 1, name, "scalar isotropic law",
    1, linear elastic generic, 1.e9;
constitutive law: 2, name, "3D isotropic law",
    3, linear elastic generic,
        sym, 1.e9,  0.,    0.,
                    1.e6, -1.e5,
                          1.e6;
constitutive law: 3, name, "6D isotropic law",
    6, linear elastic generic,
        diag, 1.e9, 1.e9, 1.e9, 1.e6, 1.e6, 1.e6;
```

2.10.3 Linear elastic generic axial torsion coupling

```
<specific_const_law> ::=
    linear elastic generic axial torsion coupling ,
        (DerivativeOfEntity) <stiffness> ,
        (real) <coupling_coefficient>
```

this is defined only for 6×1 vectors, where the torsion stiffness, coefficient a_{44} in the stiffness matrix, depends linearly on the axial strain, ε_1, by means of `coupling_coefficient`, i.e. the run-time torsion stiffness is

$$a_{44} = GJ + \texttt{coupling\_coefficient} \cdot \varepsilon_1. \tag{2.62}$$

This `<coupling_coefficient>`, in the classical nonlinear beam theory, is estimated according to some geometric property [4]; a general approach to the computation of prestressed beam properties is presented in [5], which is implemented in some versions of the ANBA software.

2.10.4 Cubic elastic generic

```
<specific_const_law> ::=
    cubic elastic generic
```

```
(Entity) <stiffness_1> ,
(Entity) <stiffness_2> ,
(Entity) <stiffness_3>
```

this is defined only for scalar and 3×1 vectors; the constitutive law is written according to the formula

$$f = \texttt{stiffness\_1} \cdot \varepsilon + \texttt{stiffness\_2} \cdot |\varepsilon| \, \varepsilon + \texttt{stiffness\_3} \cdot \varepsilon^3 \tag{2.63}$$

2.10.5 Inverse square elastic

```
<specific_const_law> ::= inverse square elastic ,
    (real) <stiffness> , (real) <ref_length>
```

this is defined only for scalars. The force is defined as:

$$f = \frac{\texttt{stiffness}}{\left(\texttt{ref\_lenght} \cdot (1 + \varepsilon)\right)^2} \tag{2.64}$$

2.10.6 Log elastic

```
<specific_const_law> ::= log elastic ,
    (DerivativeOfEntity) <stiffness>
```

this is defined only for scalars. The force is defined as:

$$f = \texttt{stiffness} \cdot \log\left(1 + \varepsilon\right) \tag{2.65}$$

2.10.7 Linear elastic bistop

```
<specific_const_law> ::= linear elastic bistop ,
    (DerivativeOfEntity) <stiffness> ,
    [ initial status , { inactive | active | (bool) <status> } , ]
    (DriveCaller) <activating_condition> ,
    (DriveCaller) <deactivating_condition>
```

2.10.8 Double linear elastic

```
<specific_const_law> ::= double linear elastic ,
    (real) <stiffness_1> ,
    (real) <upper_strain> ,
    (real) <lower_strain> ,
    (real) <stiffness_2>
    [, third stiffness <stiffness_3>]
```

this is defined for scalar and 3×1 vectors. In the scalar case the meaning of the entries is straightforward, while in case of 3×1 vectors the constitutive law is isotropic but in the local direction 3, where, in case of strain out of the upper or lower bound, the `stiffness_2` is used. If `third stiffness` is given, its value is used in the strain region below `lower_strain`.

2.10.9 Isotropic hardening elastic

```
<specific_const_law> ::= isotropic hardening elastic ,
    (real) <stiffness> ,
    (real) <reference_strain>
    [ , linear stiffness , <linear_stiffness> ]
```

this constitutive law is defined as follows:

$$f = \texttt{stiffness}\,\frac{\beta + \alpha\,|\boldsymbol{\varepsilon}|^2}{1 + \alpha\,|\boldsymbol{\varepsilon}|^2}\varepsilon$$

where $\alpha = 3/\,|\texttt{reference\_strain}|^2$, and $\beta = \texttt{linear\_stiffness}/\texttt{stiffness}$. The resulting constitutive law, in the scalar case, is somewhat soft/hard when β is lower/greater than 1 and $|\varepsilon|$ is smaller than `reference_strain`, approaching `linear_stiffness` when $\varepsilon \to 0$, while it grows to quasi-linear for larger $|\varepsilon|$, with slope `stiffness`.

2.10.10 Scalar function elastic, scalar function elastic isotropic

This constitutive law is based on a `ScalarFunction` to represent an analytical force-displacement curve of a single variable that is automatically differentiated to compute the slope of the curve, namely the local stiffness.

```
<specific_const_law> ::= scalar function elastic [ isotropic ] ,
    (DifferentiableScalarFunction) <function>
```

this constitutive law is defined as follows:

$$f_i = \texttt{function}\,(\boldsymbol{\varepsilon}_i)$$

the force is computed for each direction as a function of the respective strain component using the same function. When used for 1D elements, the word `isotropic` can be omitted.

Note: the `Scalar function elastic` and the `scalar function elastic isotropic` constitutive laws do not support `prestress` nor `prestrain`.

2.10.11 Scalar function elastic orthotropic

```
<specific_const_law> ::= scalar function elastic [ orthotropic ] ,
    { (DifferentiableScalarFunction) <function> | null }
    [ , ... ]
```

this constitutive law is defined as follows:

$$f_i = \texttt{function}_i\,(\boldsymbol{\varepsilon}_i)$$

the force is computed for each direction as a function of the respective strain component using a specific function for each component. If no force ought to be used for a direction, for example because that direction is constrained by a kinematic joint, the keyword `null` can be used to indicate that no function is expected for that component. When used for 1D elements, the word `orthotropic` can be omitted; note that in this case a `scalar function elastic isotropic` constitutive law is actually instantiated.

Example.

```
scalar function: "myfunc", multilinear,
    -1., -100.,
    -.5, -70.,
    0., 0.,
    .5, 70.,
    1., 100.;
constitutive law: 1000, 3,
    scalar function elastic orthotropic,
        null,
        null,
        "myfunc";
```

indicates that the constitutive law is only defined in direction 3 as a multilinear function.

Note: the `Scalar function elastic orthotropic` constitutive law does not support `prestress` nor `prestrain`.

2.10.12 Linear viscous, linear viscous isotropic

```
<specific_const_law> ::= linear viscous [ isotropic ] ,
    (real) <viscosity>
```

the linear viscous coefficient.
Note: this constitutive law does not require any prestrain template drive caller.

2.10.13 Linear viscous generic

```
<specific_const_law> ::= linear viscous generic ,
    (DerivativeOfEntity) <viscosity>
```

the linear viscous $N \times N$ matrix.
Note: this constitutive law does not require any prestrain template drive caller.

2.10.14 Linear viscoelastic, linear viscoelastic isotropic

```
<specific_const_law> ::= linear viscoelastic [ isotropic ] ,
    (real) <stiffness> ,
    { (real) <viscosity>
        | proportional , (real) <factor> }
```

the isotropic stiffness and viscosity coefficients. If `proportional` is given, then `viscosity` $=$ `factor` $\cdot$ `stiffness`.

2.10.15 Linear viscoelastic generic

```
<specific_const_law> ::= linear viscoelastic generic ,
    (DerivativeOfEntity) stiffness ,
    { (DerivativeOfEntity) <viscosity>
        | proportional , (real) <factor> }
```

the linear stiffness and viscosity $N \times N$ matrices. If `proportional` is given, then `viscosity` $=$ `factor` $\cdot$ `stiffness`.

2.10.16 Linear time variant viscoelastic generic

```
<specific_const_law> ::= linear time variant viscoelastic generic ,
    (DerivativeOfEntity) <stiffness> ,
    (DriveCaller) <stiffness_scale> ,
    { (DerivativeOfEntity) <viscosity>
        | proportional , (real) <factor> } ,
    { same | (DriveCaller) <viscosity_scale> }
```

the linear stiffness and viscosity matrices are multiplied by the respective scale factors,

$$\boldsymbol{f} = \texttt{stiffness} \cdot \texttt{stiffness\_scale} \cdot \boldsymbol{\varepsilon} + \texttt{viscosity} \cdot \texttt{viscosity\_scale} \cdot \dot{\boldsymbol{\varepsilon}}. \tag{2.66}$$

If `proportional` is given, then $\texttt{viscosity} = \texttt{factor} \cdot \texttt{stiffness}$. The keyword `same` uses for the drive caller `viscosity_scale` the same drive caller `stiffness_scale` defined for the elastic portion of the constitutive law.

Example.

```
linear time variant viscoelastic,
    1000., cosine, 2., pi/.2, .1/2, half, 1.,
    100., cosine, 2., pi/.2, 1/2, half, 1.
```

At 2s, the stiffness grows of 10% from the nominal value in .2s, while the damping doubles. See Section 2.6 for details on the syntax of `DriveCaller` entities.

Beware that arbitrarily changing the stiffness and the damping of an elastic component during the execution of the simulation may have no physical meaning. The intended use of this feature is for tailoring the analysis; for example, a higher damping level may be desirable to smooth out a non-physical transient, and later return to the appropriate damping value.

2.10.17 Linear viscoelastic generic axial torsion coupling

```
<specific_const_law> ::=
    linear viscoelastic generic axial torsion coupling ,
        (DerivativeOfEntity) <stiffness> ,
        { (DerivativeOfEntity) <viscosity>
            | proportional , (real) <factor> }
        (real) <coupling_coefficient>
```

this is defined only for 6×1 vectors; it is the viscoelastic extension of the `linear elastic generic axial torsion coupling` constitutive law (see Section 2.10.3).

2.10.18 Cubic viscoelastic generic

```
<specific_const_law> ::=
    cubic elastic generic
        (Entity) <stiffness_1> ,
        (Entity) <stiffness_2> ,
        (Entity) <stiffness_3> ,
        (DerivativeOfEntity) <viscosity>
```

this is defined only for scalar and 3×1 vectors; the constitutive law is written according to the formula

$$f = \texttt{stiffness\_1} \cdot \varepsilon + \texttt{stiffness\_2} \cdot |\varepsilon| \, \varepsilon + \texttt{stiffness\_3} \cdot \varepsilon^3 + \texttt{viscosity} \cdot \dot{\varepsilon}$$

and it is mainly intended for use with human body models where the stiffness of the joints is typically given in this form.

2.10.19 Double linear viscoelastic

```
<specific_const_law> ::= double linear viscoelastic ,
    (real) <stiffness_1> ,
    (real) <upper_strain> ,
    (real) <lower_strain> ,
    (real) <stiffness_2> ,
    (real) <viscosity>
    [ , second damping , (real) <viscosity_2> ]
```

this is analogous to the `double linear elastic` constitutive law, except for the isotropic viscosity term. The second viscosity value is used when the strain is outside the `lower_strain` – `upper_strain` range.

When this constitutive law is used with 3×1 vectors, the double linear elastic and viscous term only applies to component 3.

2.10.20 Turbulent viscoelastic

```
<specific_const_law> ::= turbulent viscoelastic ,
    (real) <stiffness> ,
    (real) <parabolic_viscosity>
    [ , (real) <threshold>
        [ , (real) <linear_viscosity> ] ]
```

the constitutive law has the form:

$$\boldsymbol{f} \;=\; \texttt{stiffness}\, \boldsymbol{\varepsilon} + k\, \dot{\boldsymbol{\varepsilon}}$$

where:

$$k = \begin{cases} \texttt{linear\_viscosity} & |\dot{\boldsymbol{\varepsilon}}| \leq \texttt{threshold} \\ \texttt{parabolic\_viscosity} & |\dot{\boldsymbol{\varepsilon}}| > \texttt{threshold} \end{cases}$$

if `threshold` is null, or not defined, the constitutive law is always parabolic. If the `linear_viscosity` is not defined, it is computed based on `parabolic_viscosity` and on `threshold` to give a continuous force curve (with discontinuous slope). Otherwise, it can be set by the user to give a discontinuous force curve, as observed in some fluids at intermediate Reynolds number.

2.10.21 Linear viscoelastic bistop

```
<specific_const_law> ::= linear viscoelastic bistop ,
    (DerivativeOfEntity) <stiffness> ,
    (DerivativeOfEntity) <viscosity> ,
    [ initial status , { inactive | active | (bool) <status> } , ]
    (DriveCaller) <activating_condition> ,
    (DriveCaller) <deactivating_condition>
```

2.10.22 GRAALL damper

This very experimental constitutive law, based on a nonlinear model for a hydraulic damper to be used in landing gear modeling, has been moved to `module-damper-graall`.

2.10.23 shock absorber

This constitutive law implements a landing gear hydraulic shock absorber:

```
<specific_const_law> ::= shock absorber ,
    [ prestrain , <value> , ]
    <reference_pressure> ,
    <reference_area_for_force_computation> ,
    <interaction_coefficient> ,
    <polytropic_exponent> ,
    [ epsilon max , <upper_strain_bound> , ]
    [ epsilon min , <lower_strain_bound> , ]
    [ penalty , <penalty_factor_for_strain> ,
       <penalty_factor_for_strain_rate> , ]
    [ metering , (DriveCaller)<metering_area> ,
       [ negative , (DriveCaller)<metering_area_for_negative_strain_rate> , ]
    [ orifice , (DriveCaller)<orifice_area> , ]
    <fluid_area> ,
    <fluid_density> ,
    <drag_coefficient_/_reference_length> # scales strain rate to velocity
    [ , friction , <reference_epsilon_prime> ,
        <friction_amplitude_coefficient> ]
```

where

- the `interaction_coefficient` is represented by

$$\texttt{kinematic scale} \cdot \frac{LA}{V_0}$$

  where `kinematic scale` is the ratio between the stroke of the shock absorber and that of the gas;

- `upper_strain_bound` is the upper strain bound; it must be at least larger than the prestrain, and defaults to 0, i.e. the shock absorber, at rest, is assumed to be fully extended;

- `lower_strain_bound` is the lower strain bound; it must be at least smaller than the prestrain, and defaults to -0.5, i.e. the shock absorber is assumed to allow a contraction equal to half its full length;

- the `penalty_factor_for_strain` defaults to 1e+9; it is active only when strain bounds are violated;

- the `penalty_factor_for_strain_rate` defaults to 0; it is active only when strain bounds are violated;

- the `metering_area` is given by a `DriveCaller` and is strain dependent; if the keyword `negative` is used, then the `metering_area_for_negative_strain_rate` is used when the strain rate is negative, i.e. the shock absorber is being compressed, while `metering_area` is used only when the shock absorber is extending;

- the orifice drive determines the area of an additional orifice, which essentially depends on the sign of the strain rate; it is used to implement relief valves;
- ...

This constitutive law contributes to the output of the element it is associated with. The name associated with each output contribution can be used to reference the corresponding value as element private data. The contributions are

1. "p" gas pressure;
2. "A" metering area;
3. "Fe" elastic force;
4. "Fv" viscous force.

2.10.24 symbolic elastic

The implementation of the family of symbolic constitutive laws is based on GiNaC (http://www.ginac.de/), a free software package for symbolic algebra manipulation. It is essentially used to automatically differentiate the user-supplied expression that describes the relationship between the output and the input.

The syntax is

```
<specific_const_law> ::= symbolic elastic ,
    epsilon , " <epsilon> " [ , ... ] ,
    expression , " <expression> " [ , ... ]
```

where epsilon is the symbol describing the input parameter as it will be used in expression.

For constitutive laws with more than one dimension, a string for each epsilon and one for each expression are expected. For example:

```
# 1D symbolic constitutive law
constitutive law: 1001, 1, symbolic elastic,
    epsilon, "eps",
    expression, "1000.*eps + 5.*eps^3";
# 3D symbolic constitutive law
constitutive law: 1003, 3, symbolic elastic,
    epsilon, "eps1", "eps2", "eps3",
    expression,
        "1000.*eps1 + 5.*eps1^3 - 10.*eps2*eps3",
        "1000.*eps2 + 5.*eps2^3 - 10.*eps3*eps1",
        "1000.*eps3 + 5.*eps3^3 - 10.*eps1*eps2";
```

Note: right now, the symbols defined within the mathematical parser are not available within symbolic constitutive laws.

2.10.25 symbolic viscous

The syntax is

```
<specific_const_law> ::= symbolic viscous ,
    epsilon prime , " <epsilon_prime> " [ , ... ] ,
    expression , " <expression> " [ , ... ]
```

where epsilon_prime is the symbol describing the derivative of the input parameter as it will be used in expression.

For constitutive laws with more than one dimension, a string for each epsilon_prime and one for each expression are expected.

2.10.26 symbolic viscoelastic

The syntax is

```
<specific_const_law> ::= symbolic viscoelastic ,
    epsilon , " <epsilon> " [ ,  ... ] ,
    epsilon prime , " <epsilon_prime> " [ , ... ] ,
    expression , " <expression> " [ , ... ]
```

where epsilon and epsilon_prime are the symbols describing the input parameter and its derivative as they will be used in expression.

For constitutive laws with more than one dimension, a string for each epsilon, one for each epsilon_prime, and one for each expression are expected.

2.10.27 ann elastic

The implementation of the family of ann constitutive laws is based on an Artificial Neural Network library that is embedded in MBDyn.

The syntax is

```
<specific_const_law> ::= ann elastic ,
    " <file_name> "
```

where the file file_name contains the parameters of the network. What is mainly significant to users is the need to scale inputs and outputs to match the amplitudes and possibly the offsets used in the training. For this purpose, the last two rows of the input files contain coefficients

```
b1 b0
a1 a0
```

which are used to scale the input u and the output y according to the transformation

$$\overline{u} = b_1 u + b_0 \tag{2.67}$$

$$\overline{y} = a_1 y + a_0 \tag{2.68}$$

so that the actual output is

$$y = \frac{f\left(b_1 u + b_0\right) - a_0}{a_1} \tag{2.69}$$

2.10.28 ann viscoelastic

The syntax is

```
<specific_const_law> ::= ann viscoelastic ,
    " <file_name> "
```

where the file file_name contains the parameters of the network.

2.10.29 nlsf viscoelastic

This constitutive law was sponsored by Hutchinson CdR.

For an n-dimensional constitutive law, it implements the formula

$$f_i = \sum_{j=1,n} k'_{0ij}\varepsilon_j + f'_i(\varepsilon_i) + \sum_{j=1,n} k''_{0ij}\dot{\varepsilon}_j + f''_i(\dot{\varepsilon}_i) \tag{2.70}$$

where f'_i and f''_i are arbitrary instances of the `scalar functions` primitive, which includes piecewise linear and spline regularization of given data, while k'_{0ij} and k''_{0ij} are the constant coefficients of a linear viscoelastic model, that account for the cross-couplings between the stresses and the strains and strain rates.

Elastic and viscous variants are defined. They differ from the viscoelastic one by only allowing the specific fraction of the input data.

Syntax. The syntax is

```
<specific_const_law> ::= nlsf viscoelastic ,
    (DerivativeOfEntity) <kappa_0'> ,
    { null | (ScalarFunction) <diag_force'> }
    [ , ... ] ,
    { (DerivativeOfEntity) <kappa_0''> | proportional , <coef> } ,
    { null | (ScalarFunction) <diag_force''> }
    [ , ... ]

<specific_const_law> ::= nlsf elastic ,
    (DerivativeOfEntity) <kappa_0'> ,
    { null | (ScalarFunction) <diag_force'> }
    [ , ... ]

<specific_const_law> ::= nlsf viscous ,
    (DerivativeOfEntity) <kappa_0''> ,
    { null | (ScalarFunction) <diag_force''> }
    [ , ... ]
```

The terms `kappa_0'` and `kappa_0''` are indicated as `DerivativeOfEntity` because in the generic case they are the result of a differential operator that computes the derivative of the force vector as function of the strain or strain rate vectors. So, for a 3D constitutive law, the force, the strain and the strain rate are 3×1 vectors, while `kappa_0'` and `kappa_0''` are 3×3 matrices. Matrix input, in MBDyn, requires to write the whole set of coefficients, row-wise; however, it can be described in many synthetic manners if matrices have some special properties, as typical linear constitutive laws do.

For example, symmetric, diagonal and empty matrices have specific short forms; in the 3×3 case they are

```
# generic 3x3 matrix:
10., -2., -2.,
-2., 20., -8.,
-2., -8., 20.

# symmetric 3x3 matrix (same as above matrix):
sym,
```

```
10., -2., -2.,
     20., -8.,
          20.

# diagonal 3x3 matrix (diagonal of above matrix):
diag,
10., 20., 20.

# empty 3x3 matrix:
null
```

The terms `diag_force'` and `diag_force''` refer to the names of the scalar functions that are defined for each component of the force vector. There must be as many terms as the dimensions of the constitutive law. These terms can either be `null`, if no non-linear contribution is defined for that component, or contain a string, enclosed in double quotes, which must refer to an already defined scalar function. The same scalar function can be used multiple times.

Example. A 1D and a 3D constitutive law using the same exponential scalar function:

```
# define a scalar function
scalar function: "exponential", exp, coefficient, -2., 5.;

# define a 1D constitutive law
constitutive law: 1000, 1,
    nlsf viscoelastic,
        10.,
            null,               # stiffness is just linear
        0.,
            "exponential";      # damping is exponential

# define a 3D constitutive law
constitutive law: 3000, 3,
    nlsf viscoelastic,
        sym,
            10., -2., -2.,
                 10., -2.,
                      10.,
            null,               # stiffness is just linear
            null,               # stiffness is just linear
            null,               # stiffness is just linear
        null,
            "exponential",      # damping is exponential...
            null,               # ...but in direction 1 only!
            null;
```

The first constitutive law corresponds to

$$f = 10.0 \cdot \varepsilon + 5.0 \cdot \mathrm{e}^{-2\dot{\varepsilon}} \tag{2.71}$$

while the second one corresponds to

$$\boldsymbol{f} = \begin{bmatrix} 10.0 & -2.0 & -2.0 \\ -2.0 & 10.0 & -2.0 \\ -2.0 & -2.0 & 10.0 \end{bmatrix} \begin{Bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \end{Bmatrix} + \begin{Bmatrix} 5.0 \cdot \mathrm{e}^{-2\dot{\varepsilon}_1} \\ 0.0 \\ 0.0 \end{Bmatrix} \tag{2.72}$$

2.10.30 nlp viscoelastic

This constitutive law was sponsored by Hutchinson CdR.

This constitutive law consists in the linear combination of nonlinear elastic and viscous effects whose coefficients are arbitrary nonlinear functions of the strain ε:

$$f_i = \sum_{j=1,3} \left(k'_{0ij} + \delta_{ij} k'_i\left(\varepsilon_i\right)\right) \varepsilon_j + \sum_{j=1,3} \left(k''_{0ij} + \delta_{ij} k''_i\left(\varepsilon_i\right)\right) \dot{\varepsilon}_j, \tag{2.73}$$

where δ_{ij} is Kronecker's operator; k'_i and k''_i are implemented as arbitrary scalar functions, which includes piecewise linear and spline regularization of given data.

Elastic and viscous variants are defined. They differ from the viscoelastic one by only allowing the specific fraction of the input data.

Syntax. The syntax is

```
<specific_const_law> ::= nlp viscoelastic ,
    (DerivativeOfEntity) <kappa_0'> ,
    { null | (ScalarFunction) <diag_stiffness> }
    [ , ... ] ,
    { (DerivativeOfEntity) <kappa_0''> | proportional , <coef> } ,
    { null | (ScalarFunction) <diag_damping> }
    [ , ... ]

<specific_const_law> ::= nlp elastic ,
    (DerivativeOfEntity) <kappa_0'> ,
    { null | (ScalarFunction) <diag_stiffness> }
    [ , ... ]

<specific_const_law> ::= nlp viscous ,
    (DerivativeOfEntity) <kappa_0''> ,
    { null | (ScalarFunction) <diag_damping> }
    [ , ... ]
```

Example. A 1D and a 3D constitutive law using the same exponential scalar function:

```
# define a scalar function
scalar function: "exponential", exp, coefficient, -2., 5.;

# define a 1D constitutive law
constitutive law: 1000, 1,
    nlp viscoelastic,
        10.,
            null,               # stiffness is just linear
```

```
        0.,
            "exponential";      # damping slope is exp.

    # define a 3D constitutive law
    constitutive law: 3000, 3,
        nlp viscoelastic,
            sym,
                10., -2., -2.,
                     10., -2.,
                          10.,
                null,               # stiffness is just linear
                null,               # stiffness is just linear
                null,               # stiffness is just linear
            null,
                "exponential",      # damping slope is exp...
                null,               # ...but in direction 1 only!
                null;
```

The first constitutive law corresponds to

$$f = 10.0 \cdot \varepsilon + 5.0 \cdot \mathrm{e}^{-2\varepsilon} \cdot \dot{\varepsilon} \tag{2.74}$$

while the second one corresponds to

$$\boldsymbol{f} = \begin{bmatrix} 10.0 & -2.0 & -2.0 \\ -2.0 & 10.0 & -2.0 \\ -2.0 & -2.0 & 10.0 \end{bmatrix} \begin{Bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \end{Bmatrix} + \begin{bmatrix} 5.0 \cdot \mathrm{e}^{-2\varepsilon_1} & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} \begin{Bmatrix} \dot{\varepsilon}_1 \\ \dot{\varepsilon}_2 \\ \dot{\varepsilon}_3 \end{Bmatrix} \tag{2.75}$$

Note: although the syntax of this constitutive law and that of the `nlsf viscoelastic` one (Section 2.10.29) is essentially identical, their behavior may be quite different, as indicated by Equations (2.70) and (2.73).

2.10.31 Hookean linear elastic isotropic

Author: Reinhard Resch The hookean linear elastic isotropic constitutive law implements the classical three dimensional form of Hooke's law for solid materials. See [6] or [7] for example. For this constitutive law, there are an elastic variant, a viscoelastic variant and an elastic variant for displacement/pressure (u/p-c) formulations. The latter formulation is supporting also fully incompressible materials (e.g. `nu` $= \frac{1}{2}$). Those constitutive laws are valid only for solid elements. See also section 8.16.

Syntax. The syntax is

```
<specific_const_law> ::= hookean linear elastic isotropic ,
  E , (real) <E> ,
  nu , (real) <nu>

<specific_const_law> ::= hookean linear viscoelastic isotropic ,
  E , (real) <E> ,
  nu , (real) <nu>,
  beta, (real) <beta>
```

`E` Young's modulus

`nu` Poisson's ratio

`beta` Damping coefficient

In case of a Total Lagrangian Formulation, the relation between Second Piola-Kirchhoff stress tensor $\boldsymbol{S}$ and Green-Lagrange strain tensor $\boldsymbol{G}$ is:

$$\boldsymbol{S}_{ij} = 2\,\mu\left(\boldsymbol{G}_{ij} + \texttt{beta}\,\dot{\boldsymbol{G}}^{\star}_{ij}\right) + \lambda\,\delta_{ij}\left(\operatorname{trace}\boldsymbol{G} + \texttt{beta}\,\operatorname{trace}\dot{\boldsymbol{G}}^{\star}\right) \tag{2.76}$$

$$\mu = \frac{\texttt{E}}{2\,(1+\texttt{nu})} \tag{2.77}$$

$$\lambda = \frac{\texttt{E}\,\texttt{nu}}{(1+\texttt{nu})\,(1-2\,\texttt{nu})} \tag{2.78}$$

See also equation 8.89.

Example.

```
constitutive law: 1000, name, "steel", 6, ## usable only for non u/p-c elements
            hookean linear elastic isotropic,
                E, 200000e6,
                nu, 0.3;

constitutive law: 1000, name, "steel", 7, ## usable only for u/p-c elements
            hookean linear elastic isotropic,
                E, 200000e6,
                nu, 0.3;

constitutive law: 1001, name, "steel", 6, ## usable only for non u/p-c elements
            hookean linear viscoelastic isotropic,
                E, 200000e6,
                nu, 0.3,
                beta, 1e-6;
```

2.10.32 Neo-Hookean

Author: Reinhard Resch Neo-Hookean isotropic hyperelastic constitutive laws can be used to model rubber like materials with solid elements. See also section 8.16.

Syntax. The syntax is

```
<specific_const_law> ::= neo hookean elastic ,
  E , (real) <E> ,
  nu , (real) <nu>

<specific_const_law> ::= neo hookean viscoelastic ,
  E , (real) <E> ,
  nu , (real) <nu> ,
  beta , (real) <beta>
```

The implementation is based on [8]. Input parameters for Neo-Hookean materials are Young's modulus `E` and Poisson's ratio `nu`. For the viscoelastic variant, there is an additional input parameter `beta` which is a damping coefficient. In contradiction to [8], damping effects are always considered proportional to initial stiffness. The Neo-Hookean constitutive law is describing the relation between Second Piola-Kirchhoff stress tensor $\boldsymbol{S}$ and Right Cauchy-Green strain tensor $\boldsymbol{C}$. See also equation 8.89.

$$S_{ij} = \mu\,\delta_{ij} + \left[\lambda\left(III_C - \sqrt{III_C}\right) - \mu\right]\left[\boldsymbol{C}^{-1}\right]_{ij} + \texttt{beta}\left[\mu\,\dot{\boldsymbol{C}}^{\star}_{ij} + \frac{1}{2}\,\delta_{ij}\,\lambda\,\mathrm{trace}\left(\dot{\boldsymbol{C}}^{\star}\right)\right] \tag{2.79}$$

$$\boldsymbol{C}^{-1} = \frac{1}{III_C}\left(\boldsymbol{C}\,\boldsymbol{C} - I_C\,\boldsymbol{C} + II_C\,\boldsymbol{I}\right) \tag{2.80}$$

$$I_C = \mathrm{trace}\left(\boldsymbol{C}\right) \tag{2.81}$$

$$II_C = \frac{1}{2}\left(\mathrm{trace}\left(\boldsymbol{C}\right)^2 - \mathrm{trace}\left(\boldsymbol{C}\,\boldsymbol{C}\right)\right) \tag{2.82}$$

$$III_C = \det\left(\boldsymbol{C}\right) \tag{2.83}$$

$$\mu = \frac{\texttt{E}}{2\,(1+\texttt{nu})} \tag{2.84}$$

$$\lambda = \frac{\texttt{E}\,\texttt{nu}}{(1+\texttt{nu})\,(1-2\,\texttt{nu})} \tag{2.85}$$

Example.

```
constitutive law: 1000, name, "elastic rubber", 6,
            neo hookean elastic,
                E, 5e6,
                nu, 0.49;

constitutive law: 1001, name, "viscoelastic rubber", 6,
            neo hookean viscoelastic,
                E, 5e6,
                nu, 0.49,
                beta, 1e-3;
```

2.10.33 Mooney-Rivlin

Author: Reinhard Resch Mooney-Rivlin constitutive laws, which are based on [6], are commonly used for rubber materials. They are similar to Neo-Hookean constitutive laws.

Syntax. The syntax is

```
<specific_const_law> ::= mooney rivlin elastic ,
  { C1 , (real) <C1> ,
    C2 , (real) <C2> ,
    kappa, (real) <kappa> |
    E , (real) <E> ,
    nu , (real) <nu>
    [ , delta , (real) <delta> ]
  }
```

There are two alternative sets of input parameters. Either Young's modulus `E`, Poisson's ratio `nu` and the optional ratio `delta`, or Mooney-Rivlin parameters `C1`, `C2` and the bulk modulus `kappa` may be provided.

If delta $= 0$ or C2 $= 0$, a classical Neo-Hookean constitutive law is obtained [6]. In the compressible case, this is not exactly the same as the constitutive law described in section 2.10.32. Equations 2.86 to 2.89 show the relations between the material parameters. Mooney Rivlin constitutive laws may be used for pure displacement based formulations as well as for displacement/pressure (u/p-c) formulations. In the latter case, even fully incompressible materials (e.g. nu $= \frac{1}{2}$) may be used.

$$G = \frac{\mathtt{E}}{2\,(1+\mathtt{nu})} \tag{2.86}$$

$$\mathtt{C1} = \frac{G}{2\,(1+\mathtt{delta})} \tag{2.87}$$

$$\mathtt{C2} = \mathtt{delta}\,\mathtt{C1} \tag{2.88}$$

$$\mathtt{kappa} = \frac{\mathtt{E}}{3\,(1-2\,\mathtt{nu})} \tag{2.89}$$

Equation 2.90 is based on [6] and shows the relationship between the Right Cauchy-Green tensor $\boldsymbol{C}$ and the Second Piola-Kirchhoff stress tensor $\boldsymbol{S}$.

$$\begin{aligned}\boldsymbol{S} = {} & 2\left\{\mathtt{C1}\,III_C^{-\frac{1}{3}}\,\boldsymbol{I} + \mathtt{C2}\,III_C^{-\frac{2}{3}}\,(I_C\,\boldsymbol{I} - \boldsymbol{C})\right. \\ & \left. + \left[\frac{1}{2}\,\mathtt{kappa}\left(III_C - \sqrt{III_C}\right) - \frac{1}{3}\,\mathtt{C1}\,I_C\,III_C^{-\frac{1}{3}} - \frac{2}{3}\,\mathtt{C2}\,II_C\,III_C^{-\frac{2}{3}}\right]\boldsymbol{C}^{-1}\right\}\end{aligned} \tag{2.90}$$

$$I_C = \mathrm{trace}\,(\boldsymbol{C}) \tag{2.91}$$

$$II_C = \frac{1}{2}\left(\mathrm{trace}\,(\boldsymbol{C})^2 - \mathrm{trace}\,(\boldsymbol{C}\,\boldsymbol{C})\right) \tag{2.92}$$

$$III_C = \det\,(\boldsymbol{C}) \tag{2.93}$$

Example.

```
constitutive law: 1001, name, "rubber 1", 6, ## usable only for non u/p-c elements
            mooney rivlin elastic,
                C1, 5e6,
                C2, 1.25e6,
                kappa, 1200e6; ## nearly incompressible

constitutive law: 1002, name, "rubber 2", 6, ## usable only for non u/p-c elements
            mooney rivlin elastic,
                E, 37.5e6,
                nu, 0.4948, ## nearly incompressible
                delta, 0.25;

constitutive law: 1001, name, "rubber 3", 7, ## usable only for u/p-c elements
            mooney rivlin elastic,
                E, 37.5e6,
                nu, 0.4948,   ## nearly incompressible
                delta, 0.25;

constitutive law: 1002, name, "rubber 4", 7, ## usable only for u/p-c elements
            mooney rivlin elastic,
                E, 37.5e6,
                nu, 0.5,   ## fully incompressible
                delta, 0.25;
```

2.10.34 Bilinear isotropic hardening

Author: Reinhard Resch Also bilinear isotropic hardening constitutive laws can be used only for solid elements. This constitutive law is based on the classical incremental theory of plasticity of VON MISES [6]. It is applicable for large deformations, but only for small strains. In addition to Young's modulus E and Poisson's ratio nu, the plastic tangent modulus ET and the equivalent initial yield stress sigmayv are required input parameters. In case of yielding, the relationship between equivalent plastic strain increment $\Delta\bar{e}^P$ and equivalent stress ${}^{t+\Delta t}\bar{\sigma}$ is defined as [6]:

$$\Delta\bar{e}^P = \frac{{}^{t+\Delta t}\bar{\sigma} - {}^{t}\sigma_y}{E_P} \tag{2.94}$$

$$E_P = \frac{\mathrm{E}\,\mathrm{ET}}{\mathrm{E}-\mathrm{ET}} \tag{2.95}$$

ET $= 0$ gives an ideal plastic material without any hardening effect.

Displacement/pressure (u/p-c) formulations Also bilinear isotropic hardening constitutive laws may be used for pure displacement based formulations as well as displacement/pressure (u/p-c) formulations. Usually u/p-c formulations are preferred because the plastic volumetric strains are zero, which can also lead to nearly incompressible behavior [6].

Syntax. The syntax is

```
<specific_const_law> ::= bilinear isotropic hardening ,
  E , (real) <E> ,
  nu , (real) <nu> ,
  ET , (real) <ET> ,
  sigmayv , (real) <sigmayv>
```

Example.

```
constitutive law: 1000, name, "steel", 6, ## usable only for non u/p-c elements
            bilinear isotropic hardening,
                     E, 210000e6,
                     nu, 0.3,
                     ET, 2000e6,
                     sigmayv, 235e6;

constitutive law: 1000, name, "steel", 7, ## usable only for u/p-c elements
            bilinear isotropic hardening,
                     E, 210000e6,
                     nu, 0.3,
                     ET, 2000e6,
                     sigmayv, 235e6;
```

2.10.35 Linear viscoelastic Maxwell

Author: Reinhard Resch This viscoelastic constitutive law is also called "Generalized Maxwell model" or "Wiechert" model. It is based on [9].

Syntax. The syntax is

```
<specific_const_law> ::= linear viscoelastic maxwell n ,
    E0 , (real) <E0> ,
        (integer) <num_elem> ,
        <maxwell_element> [ , ... ] ,
    C, (DerivativeOfEntity) <C>

<maxwell_element> ::= E1 , <E1> , eta1 , <eta1>
```

There is also a simplified variant which uses only a single Maxwell element. It is also called "Zener" model.

```
<specific_const_law> ::= linear viscoelastic maxwell 1 ,
    E0 , (real) <E0> ,
    <maxwell_element> ,
    C, (DerivativeOfEntity) <C>
```

The linear viscoelastic Maxwell constitutive law can be used to consider creep and relaxation effects of polymeric materials with beam- and solid-elements. For that purpose it is required to fit the constants $\texttt{E0}$, $\texttt{E1}_1$, ... $\texttt{E1}_{\texttt{num\_elem}}$ to experimental data. The stress and strain rates of the Maxwell constitutive law are described by equation 2.96 and equation 2.97 respectively.

$$\boldsymbol{\sigma} = \texttt{E0}\,\texttt{C}\,\boldsymbol{\varepsilon} + \sum_{i=1}^{\texttt{num\_elem}} \texttt{E1}_i\,\texttt{C}\,(\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_i^v) \tag{2.96}$$

$$\dot{\boldsymbol{\varepsilon}}_i^v = \frac{\texttt{E1}_i}{\texttt{eta1}_i}\,\texttt{C}\,(\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_i^v) \tag{2.97}$$

If the constitutive law is isotropic, and it is used for solid elements, then the following expression, which is equivalent to Hooke's law, can be used for $\texttt{C}$.

$$\texttt{C} = \frac{1}{(1+\nu)\,(1-2\,\nu)} \begin{pmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}\,(1-2\,\nu) & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2}\,(1-2\,\nu) & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2}\,(1-2\,\nu) \end{pmatrix} \tag{2.98}$$

In this case ν is Poisson's ratio and $\texttt{E0}$ is equivalent to Young's modulus for low strain rates.

Eigenanalysis It should be emphasized, that the linear viscoelastic Maxwell constitutive law is using internal states to represent the viscous strains $\boldsymbol{\varepsilon}_i^v$. For that reason it behaves like a pure elastic material without any damping during an Eigenanalysis according section 4.1.1.

Example.

```
constitutive law: 1000, name, "Wiechert", 6,
            linear viscoelastic maxwell n,
            E0, 2.1000000000000000e+05,
            2,
               E1, 1.8900000000000000e+05,
               eta1, 3.7800000000000000e+05,
```

```
                E1, 2.1000000000000000e+04,
                eta1, 4.2000000000000000e+04,
            C, eye;

constitutive law: 1001, name, "Zener", 6,
            linear viscoelastic maxwell 1,
            E0, 2.1000000000000000e+05,
            E1, 1.8900000000000000e+05,
            eta1, 3.7800000000000000e+05,
            C, eye;
```

2.10.36 MFront code generator

Author: Reinhard Resch

MFront is a code generator, developed by Thomas Helfer et al., which can be used to generate constitutive laws, also called "behaviours", for several commercial and open-source solvers[10]. See also the following references:

- TFEL/MFront
- MFrontGenericInterfaceSupport
- MFrontGallery

Most of the constitutive laws developed with MFront may be used also with MBDyn, provided that MBDyn was compiled using the "–with-mfront" flag.

2.10.37 mfront small strain/mfront finite strain

MFront distinguishes between different types of constitutive laws:

- `mfront small strain` is a (`ConstitutiveLaw<6D>`) applicable only to small strains
- `mfront finite strain` is a (`ConstitutiveLaw<9D>`) applicable also to finite strains

Those constitutive laws can be used only with solid elements. See also section 8.16 and section 8.16.7.

Remark Since MFront is a powerful tool with many options, the following section should be considered just as a simple tutorial instead of a comprehensive manual on how to use MFront.

Syntax.

```
<specific_const_law> ::= { mfront small strain | mfront finite strain },
        library path , " <mfront_library_path> " ,
        name, " <mfront_behaviour_name> "
        [  parameters (integer) <num_parameters> , <mfront_parameter> [ , ... ]   ]
        [  properties (integer) <num_properties> , <mfront_property> [ , ... ]   ]

        <mfront_parameter> :: = " <mfront_parameter_name> ", (real) <mfront_parameter_value>
        <mfront_property> :: = " <mfront_property_name> ", (real) <mfront_property_value>
```

Example.

```
    constitutive law: 1, name, "mfront_linear_elastic_demo1", 6,
                      mfront small strain,
                      library path, "src/libBehaviour.so",
                      name, "MFrontLinearElasticDemo1",
                      parameters, 2,
                      "YoungModulus", 210e9,
                      "PoissonRatio", 0.26;
```

Create an MFront source file The following code shows an example, on how to define a linear elastic constitutive law using MFront's domain specific language. It will be equivalent to the Hookean linear elastic isotropic constitutive law defined in section 2.10.31.

```
@DSL Implicit;
@Behaviour MFrontLinearElasticDemo1;
@Description {
  "This file implements a linear elastic constitutive law "
  "and it's purpose is just for demonstration"
}

@Brick "StandardElasticity"{
  young_modulus: 150e9,
  poisson_ratio: 0.3
};
```

Compile an MFront source file into a shared library In order to use such a constitutive law in MBDyn, it will be necessary to build a shared library from one or more MFront source files using the MFront code generator.

```
mfront --interface=generic --build MFrontLinearElasticDemo1.mfront
```

Locate the shared library The parameter <mfront_library_path> must point to the generated library which will be located under "src/libBehaviour.so" by default.

How to define material properties All the default values for material properties and parameters will be taken from the MFront source file. However, those default values may be overridden by means of an <mfront_property> or <mfront_parameter> entry. In order to get a description of all the available properties and parameters, the following command may be used:

```
mfront-query --parameters --material-properties MFrontLinearElasticDemo1.mfront
```

In case of the previous example, the following output will be printed:

```
- YoungModulus (young): the Young's modulus of an isotropic material
- PoissonRatio (nu): the Poisson ratio of an isotropic material
...
```

2.10.38 array

This constitutive law wrapper linearly combines the output of multiple constitutive laws. The syntax is

```
<specific_const_law> ::= array ,
    <number> ,
        <wrapped_const_law> [ , ... ]
```

The type of the resulting constitutive law is computed by combining that of the underlying constitutive laws. For example, if an elastic and a viscous constitutive law are combined, the type of the resulting array is viscoelastic. If `number` is 1, the underlying constitutive law is instantiated, and the array is ignored. Each underlying constitutive law can have a specific value of prestress and prestrain, as allowed by its own definition. No overall prestress nor prestrain is allowed by the `array` wrapper.

Example.

```
# the constitutive law
array, 2,
    linear elastic, 1000.,
    linear viscous, 10.

# is equivalent to
linear viscoelastic, 1000., 10.
```

2.10.39 axial wrapper

This constitutive law wrapper transforms a 1D constitutive law into a corresponding 3D constitutive law that acts about an axis that is fixed in the relative reference of the first node. It is intended to be used with `deformable hinge` joints that act between nodes constrained by a `revolute hinge` joint, and thus are only allowed relative rotation about one axis. The syntax is

```
<specific_const_law> ::= axial wrapper ,
    (Vec3)<dir> ,
    (ConstitutiveLaw<doublereal, doublereal>) <wrapped_const_law>
```

The 1D constitutive law `wrapped_const_law` acts about (or along) direction `dir`.

Example.

```
# the constitutive law
axial wrapper,
    1., 0., 0.,           # the direction
    linear elastic, 1000.  # the wrapped constitutive law
```

2.10.40 bistop

This wrapper applies the logic of the bistop, that is activate or deactivate the constitutive property based on distinct activation and deactivation conditions, with memory, to a generic underlying constitutive law.

The syntax is

```
<specific_const_law> ::= bistop ,
    [ initial status , { inactive | active | (bool) <status> } , ]
    [ capture reference strain , { yes | no | (bool) <capture> } , ]
```

```
(DriveCaller) <activating_condition> ,
(DriveCaller) <deactivating_condition> ,
(ConstitutiveLaw<<Entity>, <DerivativeOfEntity>>) <wrapped_const_law>
```

When the state is `inactive` (`false`, that is zero; the default) it is ignored (zero force and slopes are returned); when the state is `active` (`true`, that is non-zero), the constitutive law is used. With memory means that once the activation condition is triggered, the status remains `active` until the deactivation condition is triggered and the status is turned into `inactive`. Conversely, when the status is `inactive` it remains such until the activation condition is triggered, turning it into `active`.

By default, when transitioning from `inactive` to `active` (namely, when the `activating_condition` changes from `false` to `true`), the reference strain is reset to the current strain value, $\varepsilon_{\text{curr}}$, so the underlying `wrapped_const_law` is subsequently evaluated as `wrapped_const_law`$(\varepsilon - \varepsilon_{\text{curr}})$. If `capture reference strain` is `no` (or `false`, that is zero), the reference strain is not reset.

2.10.41 drive caller wrapper

This wrapper multiplies a generic constitutive law by a scalar `drive caller`. The syntax is

```
<specific_const_law> ::= drive caller wrapper ,
    (DriveCaller) <multiplier> ,
    (ConstitutiveLaw<<Entity>, <DerivativeOfEntity>>) <wrapped_const_law>
```

The wrapped constitutive law `wrapped_const_law` is multiplied by `multiplier`, i.e.

$$\boldsymbol{f}\left(\boldsymbol{\varepsilon}, \dot{\boldsymbol{\varepsilon}}, t\right) = w(t) \cdot \hat{\boldsymbol{f}}\left(\boldsymbol{\varepsilon}, \dot{\boldsymbol{\varepsilon}}, t\right) \tag{2.99}$$

where $\hat{\boldsymbol{f}}$ is the wrapped constitutive law, `wrapped_const_law`, and $w(t)$ is the `drive caller` `multiplier`; as a consequence, the gradients of the force $\boldsymbol{f}$ with respect to the strain $\boldsymbol{\varepsilon}$ and of its rate, $\dot{\boldsymbol{\varepsilon}}$, are

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\varepsilon}} = w(t) \cdot \frac{\partial \hat{\boldsymbol{f}}}{\partial \boldsymbol{\varepsilon}} \tag{2.100a}$$

$$\frac{\partial \boldsymbol{f}}{\partial \dot{\boldsymbol{\varepsilon}}} = w(t) \cdot \frac{\partial \hat{\boldsymbol{f}}}{\partial \dot{\boldsymbol{\varepsilon}}} \tag{2.100b}$$

Example.

```
# the constitutive law
drive caller wrapper,
    string, "Time >= T0",  # the drive caller
    linear elastic, 1000.  # the wrapped constitutive law
```

2.10.42 invariant angular

This is not a constitutive law, but rather a wrapper for constitutive laws used within the "attached" variant of the `deformable hinge` joint. As such, it can only be used with the 3D dimensionality. It basically allows to refer an ancillary 3D constitutive law to an orientation that is intermediate with respect to the orientations of the two connected nodes.

```
<specific_const_law> ::= invariant angular ,
    <xi> , <wrapped_const_law>
```

xi is the fraction of relative orientation the ancillary constitutive law will be referred to, with respect to the first node of the joint. Its value should be comprised between 0 (attached to the first node) and 1 (attached to the second node); a value of 1/2 yields the invariant deformable hinge joint, but any value is allowed, not limited to within the [0, 1] range.

Note: the contribution to the Jacobian matrix will be incomplete when the underlying constitutive law has a viscous contribution. This may slow down convergence, or even prevent it. If this is an issue, the invariant deformable hinge should be used, since it does not suffer from this limitation, although it only allows xi = 1/2.

2.11 Hydraulic fluid

Hydraulic fluid data defines the constitutive properties of hydraulic fluids, which are generally required by hydraulic elements. Hydraulic fluid data can be defined in two ways, according to the BNF:

```
<hydraulic_fluid> ::=
    { <fluid_type> , <fluid_properties>
        | reference , <label> }
```

The latter references a previously defined hydraulic fluid dataset, described in Section 2.4.4. The available types fluid_type, with the related fluid_properties, are:

2.11.1 Incompressible

```
<fluid_type> ::= incompressible

<fluid_properties> ::=
    <property> [ , ... ]

<property> ::=
    { density , <density>
        | viscosity , <viscosity> # dynamic viscosity
        | pressure , <pressure>
        | temperature , <temperature> }
```

2.11.2 Linearly compressible

```
<fluid_type> ::= linear compressible

<fluid_properties> ::=
    <property> [ , ... ]

<property> ::=
    { density , <ref_density> ,
            { sound celerity , <sound_celerity> | <beta> } , <ref_pressure>
        | viscosity , <viscosity> # dynamic viscosity
        | temperature , <temperature> }
```

If the keyword sound celerity is used, the bulk modulus beta is computed as

$$\texttt{beta} = \texttt{ref\_density} \cdot \texttt{sound\_celerity}^2 \tag{2.101}$$

2.11.3 Linearly compressible, with thermal dependency

```
<fluid_type> ::= linear thermal compressible

<fluid_properties> ::=
    <property> [ , ... ]

<property> ::=
    { density , <ref_density> ,
            { sound celerity , <sound_celerity> | <beta> } , <ref_pressure>
            <alpha> , <ref_temperature>
        | viscosity , <viscosity> } # dynamic viscosity
```

2.11.4 Super (linearly compressible, with thermal dependency)

```
<fluid_type> ::= super

<fluid_properties> ::=
    <property> [ , ... ]

<property> ::=
    { density , <ref_density> ,
            { sound celerity , <sound_celerity> | <beta> } , <ref_pressure>
        | viscosity , <viscosity> # dynamic viscosity
        | temperature , <temperature> }
```

according to equation

$$\begin{aligned} \rho_- &= \rho_0 + \texttt{ref\_density} \cdot \frac{1}{2}(1 + \tanh(a\,(p - p_0))) \\ \rho &= \rho_- && p < p_0 \\ \rho &= \rho_- + \frac{p - p_0}{\texttt{beta}} && p > p_0 \end{aligned} \tag{2.102}$$

Note: highly experimental; currently, ρ_0, a and p_0 are hardcoded in SI units; the syntax will change

2.11.5 Exponential compressible fluid, with saturation

```
<fluid_type> ::= exponential

<fluid_properties> ::=
    [ <property> [ , ... ] , ] <p_sat>

<property> ::=
    { density , <ref_density> ,
            { sound celerity , <sound_celerity> | <beta> } , <ref_pressure>
        | viscosity , <viscosity> # dynamic viscosity
        | temperature , <temperature> }
```

where p_sat is the saturation pressure, according to equation

$$\rho = \texttt{ref\_density} \cdot \mathrm{e}^{\dfrac{p - \texttt{ref\_pressure}}{\texttt{beta}}} \qquad p > \texttt{p\_sat}$$
$$\rho = \texttt{ref\_density} \cdot \mathrm{e}^{1000\dfrac{p - \texttt{ref\_pressure}}{\texttt{beta}}} \qquad p < \texttt{p\_sat} \tag{2.103}$$

Note: this fluid constitutive law is loosely inspired by AMESim's simply corrected compressible fluid.

2.12 Authentication Methods

Some authentication methods are defined and made available to specific program modules; they are used to authenticate before accessing some resources of the program while it is running. The syntax is:

```
<authentication_method> ::= <method> [ , <specific_data> ]
```

Authentication methods in general expect some authentication tokens to be input. Usually a user name and a password are required.

2.12.1 Note on security and confidentiality

No encryption is used in communications, unless provided by the underlying mechanism (e.g. some SASL mechs), so the authentication methods are very rough and should be considered as insecure. TLS may be considered in the future. If confidentiality is required, SASL with at least DIGEST-MD5 is strongly recommended; otherwise no authentication should be used. As an alternative, a SSH tunnel may be established between the client and the server machine, and simple authentication can be used. Otherwise, if the user has direct access to the server where the computation is being run, sockets with `local` namespace can be used, and security can be enforced by means of the access privileges of the socket file. Since some of the UN*X systems do not honor socket permissions, a portable way to exploit filesystem access permissions is to put the socket in a dedicated directory, and use the permissions of the directory to control access to the socket.

Available methods are:

2.12.2 No authentication

```
<authentication_method> ::= no auth
```

2.12.3 Password

```
<authentication_method> ::= password

<specific_data> ::=
    user , " <user_name> " ,
    credentials , { prompt | " <user_cred> " }
    [ , salt format , <salt_format> ]
```

In case the keyword prompt is given as credentials, the user is prompted for a password; user_cred is used otherwise. The optional parameter salt_format allows to specify different formats for the `salt`, for those `crypt(3)` extensions that support more sophisticated encryption mechanisms (e.g. MD5). See `crypt(3)` for details. If the credentials are preceded by the string `{CRYPT}`, they are assumed to be already encrypted, and the remaining portion is used.

2.12.4 PAM (Pluggable Authentication Modules)

```
<authentication_method> ::= pam

<specific_data> ::=
    [ user , " <user_name> " ]
```

The *Linux-PAM* Pluggable Authentication Modules can be used to authenticate a user. If no user name is provided, the effective user id, as provided by the `geteuid(2)` system function, is used to retrieve the username of the owner of mbdyn process. the `user` must be valid. The authentication is performed through a system-dependent `pam` configuration file. No checks on the validity of the account or on the permission of opening a session are made; account, session and password changes should be explicitly denied to `mbdyn` to avoid possible security breaks (see the following example). The interested reader should consult the documentation that comes with the package, try for instance `http://www.kernel.org/pub/linux/libs/pam/` for details.

An example is provided with the package, in `/etc/pam.d/mbdyn`:

```
### use either of the following:
auth       required     /lib/security/pam_unix_auth.so
# auth       required     /lib/security/pam_pwdb.so
#
### no account, session or password allowed
account    required     /lib/security/pam_deny.so
session    required     /lib/security/pam_deny.so
password   required     /lib/security/pam_deny.so
```

which allows authentication by using standard Un*x or `libpwdb` based authentication.

2.12.5 SASL (Simple Authentication and Security Layer)

```
<authentication_method> ::= sasl

<specific_data> ::= [ <parameter> [ , ... ] ]

<parameter> ::=
    { user , " <user_name> "
        | mechanism , " <preferred_mechanism> " }
```

This is the preferred authentication method because it is mechanism-independent, it can be reasonably secure and automatically selects the most appropriate mechanism available on both the client and the server machine. It requires Cyrus SASL 2 (See `http://asg.web.cmu.edu/sasl/` for details, and follow the documentation to obtain a working setup).

2.13 Miscellaneous

Finally there are some miscellaneous points:

- (UN*X systems) Environment variables whose name starts with MBDYN may be defined and passed to an execution of the mbdyn command. The following are recognized at present:

  1. `MBDYNVARS=<expr_list>` where `expr_list` is a list of mathematical expressions separated by semicolons. They are parsed and evaluated; declared variables are added to the symbol table and can be subsequently dereferenced during the whole execution of the program.

2. MBDYN_<type>_<name>=<value>, where type is a legal mbdyn type (integer, real or string; see Table 2.1 for details), name is a legal symbol name and value is a legal expression.

Example: sample code to execute MBDyn from within a python script, passing it the time step in the real variable DT:

```
import os;
dt = 1.e-3;
os.environ['MBDYNVARS'] = 'real DT = ' + str(dt) + ';';
os.system('mbdyn -f input.mbd -o output > output.txt 2>&1 &');
# use 'dt' in the python peer's analysis as appropriate...

# NOTE: a trailing '&' is placed to send mbdyn in background;
# the call to system() will return immediately, and the python
# script will have the opportunity to interact with the mbdyn
# execution, e.g., through socket streams
```

same thing, but now using Matlab:

```
dt = 1.e-3;
setenv('MBDYNVARS', sprintf('real DT = %s', dt));
system('mbdyn -f input.mbd -o output > output.txt 2>&1');
# use 'dt' in the matlab peer's analysis as appropriate...

# NOTE: no trailing '&' is placed to avoid sending mbdyn in background;
# the call to system() will return when the simulation is over
```

Note on running mbdyn from Matlab: often, Matlab links its own dynamic libraries, which may be incompatible with those used to build mbdyn. For this reason, it may be convenient to run mbdyn from within a shell script that sets a "sane" environment. An example script, `mbdyn.sh`, can be found in `contrib/SimulinkInterface/`. For example:

```
system('./mbdyn.sh -f input.mbd -o output > output.txt 2>&1');
```

Note on using Windows >= 10 and WSL: when executed by calling `system()` from Matlab running in Windows 10, Matlab must be instructed to use Windows Subsystem for Linux (WSL). Furthermore, the correct paths need to be used. To this end, use

```
system('wsl /absolute/path/to/mbdyn/mbdyn.sh \
    -f /absolute/path/to/input/input.mbd \
    -o /absolute/path/to/output/output \
    > /absolute/path/to/output/output.txt 2>&1');
```

Note: in this case, WSL will not see the environment populated by calling `setenv()` within Matlab. To populate the environment, one can place the environment variables definition directly in the command line. For example:

```
dt = 1.e-3;
mbdynvars = sprintf('real DT = %s', dt);
system(['wsl MBDYNVARS="' mbdynvars '" /absolute/path/to/mbdyn/mbdyn.sh \
```

```
    -f /absolute/path/to/input/input.mbd \
    -o /absolute/path/to/output/output \
    > /absolute/path/to/output/output.txt 2>&1']);
# use 'dt' in the matlab peer's analysis as appropriate...
```

Note: writing the output directly to a Windows partition can significantly slow down the execution of MBDyn.

- Newlines and indentations are not meaningful. But good indentation habits can lead to better and more readable input files.
- Everything that follows the character '`#`' is considered a remark, and is discarded until the end of the line. This can occur everywhere in the file, even inside a math expression (if any problems occur, please let me know, because chances are it is a bug!)
- A new style for comments has been introduced, resembling the C programming language style: everything comprised between the marks `/*` and `*/` is regarded as a remark:

```
/*
 * useful comments make input files readable!
 */
```

This can happen (almost) everywhere in the text except in the middle of a keyword.

- (UN*X systems) Whenever a file name contains a portion of the form `$VARNAME` or `${VARNAME}`, appropriate expansion from environment is performed; `VARNAME` is

```
VARNAME ::= [_a-zA-Z][_a-zA-Z0-9]*
```

namely, it must begin with a letter or an underscore, and can be made of underscores, letters and digits.

- (UN*X systems) Whenever a file name is required, if the string containing the file name starts with '`$`', shell environment variable expansion occurs.

Example: expand the local socket's name in `external` forces

```
# ...
path, "$MBSOCK",
# ...
```

Example: sample code to execute MBDyn from within a python script, passing it the name of the local socket to be used for an `external` force:

```
import os;
import tempfile;
tmpdir = tempfile.mkdtemp('', '.mbdyn_');
path = tmpdir + '/mbdyn.sock';
os.environ['MBSOCK'] = path;
os.system('mbdyn -f input.mbd -o output > output.txt 2>&1 &');
# use 'path' as the name of the socket as shown in the previous example
# ...
# when done...
os.rmdir(tmpdir);
```

This code:

  - creates a uniquely named temporary directory in the default temporary directory (e.g. `/tmp`) only accessible to the owner of the process, according to `/tmp/.mbdyn_XXXXXX`, where `XXXXXX` are six random chars (see `mkdtemp(3)` for details);
  - appends the name `/mbdyn.sock` to form the full path of the socket;
  - sets the full path of the socket in the environment under the name `MBSOCK`;
  - executes MBDyn; its input file is supposed to use "`$MBSOCK`" as path name of a local socket.

  This procedure guarantees that a co-simulation performed between MBDyn and the peer Python script uses a unique, protected socket and thus does not interfere with other co-simulations running on the same system.

- (UN*X systems) Whenever a file name is required, the shell-like syntax for home directories (i.e. `~/filename` or `~user/filename` is automatically resolved if legal [user and] filename values are inserted. Home expansion occurs after environment variable expansion (see above).
- The `license` and the `warranty` statements respectively show on the standard output the license and the warranty statement under which the code is released. They do not affect the simulation.

Chapter 3

Data

The data section is read directly by the driver program. It is included between the cards:

```
begin : data ;
    # ...
end : data ;
```

3.1 Problem

The only active card is the problem card.

The syntax is

```
<card> ::= problem : <problem_name> ;

<problem_name> ::= { initial value | inverse dynamics } ;
```

For long time the only supported problem type was the initial value problem. As a consequence, if no problem card is present, an initial value problem is assumed. An experimental inverse dynamics problem is supported as well.

Chapter 4

Problems

This section is used to insert all the data related to the problem that one needs MBDyn to solve in the simulation. The section data are included between the cards:

```
begin : <problem_name> ;
    # ...
end : <problem_name> ;
```

Implemented problems are:

- `initial value`, the time integration of mechanical and multidisciplinary problems formulated as Differential-Algebraic Equations (DAE). It can be downgraded to the solution of static and kinematic problems, by selecting purely static or kinematic contributions to the governing equations, thus giving time the role of an ordinal parameter.
- `inverse dynamics`, the computation of the generalized forces required to make a generic system perform a given trajectory. This problem is currently under development, so it is not discussed in detail.

4.1 Initial-Value Problem

At present, the main problem is `initial value`, which solves initial value problems by means of generic integration schemes that can be cast in a broad family of multistep and, experimentally, Implicit Runge-Kutta-like schemes [11].

The syntax of the module is:

```
begin : initial value ;
    # ...
end : initial value ;
```

At present, there are a number of cards that can be grouped as follows, based on the integration phase they refer to.

4.1.1 General Data

those data that refer to the main integration phase or the simulation as a whole. They are:

Initial Time

```
<card> ::= initial time : <time> ;
```

Final Time

```
<card> ::= final time : { forever | <time> } ;
```

Strategy

```
<card> ::= strategy : <strategy_type> [ , <strategy_data> ] ;
```

The available strategies are:

- `no change`

  ```
  <strategy_type> ::= no change
  ```

  the time step is never changed. However, this prevents simulation termination when the maximum number of iterations is reached. As a consequence, its use only makes sense when repeating a time step results in some modification of the analysis related to some user-defined function.

- `factor`

  ```
  <strategy_type> ::= factor

  <strategy_data> ::=
      <reduction_factor> ,
      <steps_before_reduction> ,
      <raise_factor> ,
      <steps_before_raise> ,
      <min_iterations>
      [ , <max_iterations> ]
  ```

  the time step is reduced or raised of the proper factor not before a minimum number of time steps; it is reduced if more than `max_iterations` are performed at a time step, or it the current step do not converge; it is raised if less than `min_iterations` are performed at a time step; `max_iterations` defaults to the global `max_iterations` simulation value; however, it is better to set it to a lower value, leaving some spare iterations before bailing out the simulation; the simulation bails out if two consecutive solution steps are performed without converging.

- `change`

  ```
  <strategy_type> ::= change

  <strategy_data> ::= (DriveCaller) <time_step_pattern>
  ```

  the time step is changed according to the `time_step_pattern` pattern. If the time step returned by the drive caller does not change when a step needs to be repeated, the analysis is interrupted.

  Note: the `change` strategy is not normally intended to provide adaptive time step, it rather allows to prescribe a given variable time step pattern based on the rule defined by the `time_step_pattern DriveCaller`.

In any case, step change only occurs after the first step, which is performed using the `time_step` value provided with the `time step` statement.

Min Time Step

```
<card> ::= min time step : <min_time_step> ;
```

Defines the minimum value allowed for the time step. This statement is only meaningful when variable time step is used. If the time step change strategy tries to use a time step smaller than `min_time_step`, the simulation aborts.

Max Time Step

```
<card> ::= max time step : { <max_time_step> | unlimited } ;
```

Defines the maximum value allowed for the time step. This statement is only meaningful when variable time step is used. If the time step change strategy tries to use a time step larger than `max_time_step`, the value `max_time_step` is used instead.

Time Step

```
<card> ::= time step : <time_step> ;
```

The initial time step. This value is used throughout the simulation unless some variable time step strategy is defined using the `strategy` statement.

Tolerance

```
<card> ::= tolerance : { null | <residual_tolerance> }
        [ , test , { none | norm | minmax | relnorm | sepnorm } [ , scale ] ]
    [ , { null | <solution_tolerance> }
        [ , test , { none | norm | minmax } ] ] ;
```

The only mandatory value is `residual_tolerance`, the tolerance used for the residual test; the keyword `null` disables the residual testing, disabling the computation of the test on the residual. The `test` mechanism is used to select what kind of test must be performed; currently, only `norm` (the default) and `minmax` are supported for the `<solution_tolerance>`, with the additional `relnorm` and `sepnorm` making sense only for the residual.

The special value `none` means that the test is not actually computed; it is the default when the test is disabled by setting the tolerance to zero, or by using the keyword `null`; however, it can be restored to any mechanism for output purposes.

The `relnorm` test can be useful when the magnitude of the generalized forces involved in the model is unknown or is known to vary considerably during the simulation, thus one does not know what value of tolerance may make sense for the problem at hand. In this case `relnorm` allows to compute a reference scale, so that the converge test will be mostly unaffected by problem-dependent variables, like the system of units chosen for the simulation. The automatic scaling comes with a small run time penalty, because two global vectors needs to be assembled instead of one whenever the model residual is computed.
Actually, assuming $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$ to be the nonlinear problem at hand, the residual test is modified from the default

$$\sqrt{\boldsymbol{f}^T \boldsymbol{f}} < \texttt{residual\_tolerance} \tag{4.1}$$

to

$$\frac{\sqrt{\boldsymbol{f}^T \boldsymbol{f}}}{\sqrt{\tilde{\boldsymbol{f}}^T \tilde{\boldsymbol{f}}}} < \texttt{residual\_tolerance} \tag{4.2}$$

where $\tilde{\boldsymbol{f}}$ is the vector in which the absolute values of all the contributions from the elements are assembled. The sepnorm test is a generalization of relnorm: the residual check is performed independently for each set of equations with the same physical dimension, and the corresponding residual norm is automatically scaled. The worst residual among all the different sets of equations is returned. See the tecman for an exhaustive explanation of both approaches. A meaningful tolerance for both relnorm and sepnorm is generally in the range [1.E-6, 1.E-4].

The optional parameter scale is used to enable the scaling of the residual before performing the test; default scale factors can be set for each type of degree of freedom owner, as seen for the control data item of Section 5.3.15; some entities allow individual scaling, as seen for nodes in Chapter 6; by default, no scaling takes place. A tolerance solution_tolerance to test the solution (the difference between the states at two iterations) is allowed, and also in this case a test mechanism can be chosen; by default, no test on the solution convergence is done. Currently, no scaling is allowed in the solution test.

Example.

```
# residual test by means of the norm
tolerance: 1.e-6;
# residual test with minmax method
tolerance: 1.e-6, test, minmax;
# residual test with norm method and scaling
tolerance: 1.e-6, test, norm, scale;
# solution test
tolerance: null, 1.e-9;
# residual and solution test
# (the first that succeeds breaks the loop)
tolerance: 1.e-6, 1.e-9;
# residual test with computation of solution norm
tolerance: 1.e-6, null, test, norm;
# residual test relative to the magnitude of the norm vector
tolerance: 1.e-6, test, relnorm;
# residual test relative to the magnitude of the norm vector, computed on
# consistent physical domains
tolerance: 1.e-6, test, sepnorm;
```

Max Iterations

```
<card> ::= max iterations : <max_iterations> [ , at most ] ;
```

Error out after max_iterations without passing the convergence test. The default value is zero.

If the optional keywords at most are given, the step is considered successfully performed after max_iterations regardless of passing the convergence test, as soon as the error reduced since the first iteration. For example, when using

```
max iterations: 5, at most;
```

the convergence pattern

```
...
    Iteration(0) 1.16374e+06 J
            SolErr 0
    Iteration(1) 8330.76 J
            SolErr 0
```

```
        Iteration(2) 43768.8 J
                SolErr 0
        Iteration(3) 2981.01 J
                SolErr 0
        Iteration(4) 118839 J
                SolErr 0
        Iteration(5) 8654.02 J
    ...
```

succeeds, since 8654.02 < 1.16374e+06, while

```
    ...
        Iteration(0) 1.66901e+06 J
                SolErr 0
        Iteration(1) 2.06327e+09 J
                SolErr 0
        Iteration(2) 2.64337e+10 J
                SolErr 0
        Iteration(3) 1.2131e+12 J
                SolErr 0
        Iteration(4) 7.08449e+12 J
                SolErr 0
        Iteration(5) 2.18828e+15 J
Max iterations number 5 has been reached during Step=6, Time=0.06; \
TimeStep=0.01 cannot be reduced further; aborting...
An error occurred during the execution of MBDyn; aborting...
```

fails since 2.18828e+15 > 1.66901e+06.

This option should be used with extreme care.

Modify Residual Test

```
    <card> ::= modify residual test ;
```

modify the residual test taking in account the rate of change of the status.

Enforce Constraint Equations

```
    <card> ::= enforce constraint equations : {
            constraint violations [, scale factor, <scale_factor>] |
        constraint violation rates
} ;
```

modify the residual test for the algebraic constraint equations only: if the option `constraint violations` is chosen then the algebraic equations residual is multiplied by `<dCoef>` when computing the residual test. This compensates the fact that the algebraic equations are divided by `<dCoef>` in order to improve the linear system condition number (cfr. MBDyn's technical manual), so that the residual tends to become bigger for very small time steps. If the optional keyword `scale factor` is chosen then the constraint equations are scaled by `<scale_factor>` when computing the residual test. The default is to scale by `<scale_factor>`=1./`<initial_time_step>`. If, instead, the option `constraint violation rates` is chosen, then nothing changes.

Method

```
<card> ::= method : <method_data> ;
```

where

```
<method_data> ::= { { { ms | ms2 } | ms3 | ms4 | hope
ss2 | ss3 | ss4 | Bathe | msstc3 |
mssth3 | msstc4 | mssth4 | msstc5 | mssth5 |
    {hybrid, <defaul_hybrid_method> }
    (DriveCaller) <differential_radius>
    [ , (DriveCaller) <algebraic_radius> ] } |
{ DIRK33 | DIRK43 | DIRK54 } |
crank nicolson |
implicit euler |
```

The first implemented method is Crank-Nicolson:

```
<method_data> ::= crank nicolson
```

The following are original methods:

- `ms` (`ms2` is an alias) is discussed in [11], see also

  `https://www.mbdyn.org/Documentation/Publications.html`

- `ms3` is a three-step method, discussed in [12]
- `ms4` is a four-step method, discussed in [12]
- `hope` is a multi-stage method, in which a step is performed using Crank-Nicolson's method, and the following with the `ms` method.
- `ss2`, `ss3` and `ss4` are single step methods equivalent to the corresponding multi steps
- `Bathe` is a 3-stage singly diagonally-implicit Runge–Kutta method with second-order accuracy, A-stability and tunable algorithmic dissipation
- `msstc3`, `msstc4` and `msstc5` are $(n+1)$-stage (with $n = 3, 4, 5$ respecively) singly diagonally-implicit Runge–Kutta methods with second-order accuracy, tunable algorithmic dissipation, preserving low-frequency dynamics as much as possible. They employ the trapezoidal rule from the second to the nth stage, and a general formula in the last one. See [13, 14] and MBDyn's technical manual for details.
- `mssth3`, `mssth4` and `mssth5` are $(n+1)$-stage (with $n = 3, 4, 5$ respecively) singly diagonally-implicit Runge–Kutta methods designed to have nth-order accuracy, A-stability, and tunable algorithmic dissipation. They are designed so that
  - each stage, from the second-one, has at least second-order accuracy;
  - the overall accuracy is nth-order, and on this basis, the local truncation error is minimized;
  - A-stability and tunable algorithmic dissipation for linear analysis are achieved.

  See [13, 14] and MBDyn's technical manual for details.

The methods are unconditionally stable, at least second-order accurate, and can be tuned to give the desired algorithmic dissipation by setting the value of the asymptotic spectral radius. The radius can be set independently for the differential and the algebraic variables, and a driver is used, to allow parameter dependent radius variation.

- DIRK33, DIRK34 and DIRK54 are third-order four-stages, third-order five stages and fourth-order six stages, L-stable stiffly-accurate methods from [15].

The ms method proved to be more accurate at high values of asymptotic radius (low dissipation), while hope shows some greater more accuracy at low values of the radius (high dissipation). They look nearly equivalent at radii close to 0.4, with the former giving the best trade-off between algorithmic dissipation and accuracy at about 0.6. The algebraic_radius can be omitted, defaulting to the same value used for the differential one. It is unclear whether a different spectral radius can help in increasing accuracy or dissipation of the algebraic unknowns.
The multistep method, when the asymptotic radius is zero, degenerates in the Backward Differentiation Formulas of order two. A shortcut to this case is provided as

```
<method_data> ::= bdf [ , order , <order> ]
```

The keyword order can be used to indicate a specific order of the Backward Differentiation Formulas (BDF); only first order (implicit Euler) and second order formulas are currently implemented, and the default is the second order formula, which is the most useful. The first order formula may be of help for very specific problems. It can also be selected using the shortcut

```
<method_data> ::= implicit euler
```

The hybrid method allows elements to select different integration methods for specific degrees of freedom. By default, the integration method defined after the hybrid keyword is used for all other degrees of freedom. For example a model for elastohydrodynamic lubricated bearings could use the ms method for all structural degrees of freedom, and the crank nicolson method for the mass conserving cavitation model. At the moment only module-hydrodynamic_plain_bearing2 supports this option.

```
<default_hybrid_method> ::= implicit euler | crank nicolson | ms2 | hope
```

Nonlinear Solver

The nonlinear solver solves a nonlinear problem $F(x) = 0$. The syntax is

```
<card> ::= nonlinear solver : <nonlinear_solver_name>
    [ , <nonlinear_solver_data> ] ;
```

Currently available nonlinear solvers are:

Newton-Raphson.

```
<nonlinear_solver_name> ::= Newton Raphson

<nonlinear_solver_data> ::=
    [ { true
    | modified , <iterations>
        [ , keep jacobian matrix ]
        [ , honor element requests ] } ] ;
```

if modified, the number of iterations the same Jacobian matrix will be reused, and thus factored only once, is expected. If the option keep jacobian matrix is selected, the Jacobian matrix is preserved for the desired iterations even across time steps. By default, the Jacobian matrix is recomputed at the beginning of each time step. If the option honor element requests is selected, the factorization is updated also when an element changes the structure of its equations. The default behavior is to ignore such requests[1]. This nonlinear solver is well tested.

Line search. *Author: Reinhard Resch*

```
<nonlinear_solver_name> ::= {line search |
                             bfgs |
                             mcp newton fb |
                             mcp newton min fb |
                             siconos newton fb |
                             siconos newton min fb}

<nonlinear_solver_data> ::=
    [ { true
    | modified , <iterations>
        [ , keep jacobian matrix ]
        [ , honor element requests ] } ]
    [ , tolerance x , <tolerance_x> ]
    [ , tolerance min , <tolerance_min> ]
    [ , max iterations , <max_iterations> ]
    [ , alpha , <alpha> ]
    [ , lambda min , <lambda_min>
        [ , relative , { yes | no | (bool) <relative> } ] ]
    [ , lambda factor min , <lambda_factor_min> ]
    [ , max step , <max_step> ]
    [ , zero gradient , continue , { yes | no | (bool) <zero_gradient_continue> } ]
    [ , divergence check , { yes | no | (bool) <divergence_check> }
        [ , factor , <factor> ] ]
    [ , algorithm , { cubic | factor } ]
    [ , scale Newton step , { yes | no | (bool) <scale_Newton_step> }
        [ , min scale , <min_scale> ] ]
    [ , print convergence info , { yes | no | (bool) <print_convergence_info> } ]
    [ , verbose , { yes | no | (bool) <verbose> } ]
    [ , abort at lambda min , { yes | no | (bool) <abort_at_lambda_min> } ]
    [ , mcp tolerance, (real) <mcp_tolerance> ]
    [ , mcp sigma , (real) <mcp_sigma> ]
    [ , mcp rho , (real) <mcp_rho> ]
    [ , mcp p , (real) <mcp_p> ] ;
```

- tolerance_x ≥ 0; default: tolerance_x $= 10^{-7}$
- tolerance_min ≥ 0; default: tolerance_min $= 10^{-8}$
- max_iterations ≥ 0; default: max_iterations $= 200$

[1] At present, only few elements that can change the structure of the equations, or at least radically change the Jacobian matrix, actually issue this request.

- `alpha` ≥ 0; default: `alpha` $= 10^{-4}$
- `lambda_min` ≥ 0; default: `lambda_min` $= 10^{-2}$
- $0 <$ `lambda_factor_min` < 1; default: `lambda_factor_min` $= 10^{-1}$
- `max_step` ≥ 0; default: `max_step` $= 100$
- `factor` ≥ 0; default: `factor` $= 1$
- $0 \leq$ `min_scale` ≤ 1; default: `min_scale` $= 10^{-3}$
- `mcp_tolerance` ≥ 0; default: `mcp_tolerance` $= 2.2\,10^{-16}$
- $1 \geq$ `mcp_sigma` ≥ 0; default: `mcp_sigma` $= 0.9$
- $1 \geq$ `mcp_rho` ≥ 0; default: `mcp_rho` $= 10^{-8}$
- `mcp_p` ≥ 0; default: `mcp_p` $= 2.1$

Brief description of the solver The line search solver should be used mainly if the Newton Raphson solver diverges and further reduction of the time step is not possible or not available. In many situations this solver is able to handle larger time steps or load increments than the ordinary Newton Raphson solver. However the total number of iterations can increase for large time steps. There are different variants of the line search solver.

- `line search`; This is the original implementation based on [16]
- `bfgs`; Another line search variant using Broyden updates which is also based on [16]
- `mcp newton fb`; A line search solver which supports Mixed (nonlinear) Complementarity Problems based on a Fischer Burmeister reformulation. It is a re-implementation of [17].
- `mcp newton min fb`; Another MCP solver based on a Fischer Burmeister reformulation which uses a min reformulation of the descent direction. It is also a re-implementation of [17].
- `siconos mcp newton fb`; This MCP solver is equivalent to `mcp newton fb` and is a wrapper for INRIA's Siconos library [17].
- `siconos mcp newton min fb`; A MCP solver equivalent to `mcp newton min fb` which is also a wrapper for INRIA's Siconos library [17].

MBDyn must be configured with `--with-siconos` in order to use the `siconos mcp newton fb` or `siconos mcp newton min fb` solver.

The ordinary Newton Raphson strategy The Newton Raphson Solver solves a problem of the form $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}$ by means of the iteration $\delta\boldsymbol{x} = \boldsymbol{J}^{-1}\,\boldsymbol{F}$.

$\delta\boldsymbol{x}$ The increment of the solution $\boldsymbol{x}$ during one iteration.

$\boldsymbol{J}$ The modified Jacobian matrix. For example during the initial assembly phase $\boldsymbol{J} = -\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}$. On the other hand during the regular solution phase of an initial value problem $\boldsymbol{J} = -\frac{\partial \boldsymbol{F}}{\partial \dot{\boldsymbol{x}}} - c\,\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}$ and $\delta\dot{\boldsymbol{x}} = \boldsymbol{J}^{-1}\,\boldsymbol{F}$ whereas $\delta\boldsymbol{x} = c\,\delta\dot{\boldsymbol{x}}$. But that's totally transparent to the nonlinear solver.

If the prediction for $\boldsymbol{x}$ is close enough to the solution of the problem, the convergence rate will be quadratic as long as $\boldsymbol{J}$ is exact and $\boldsymbol{F}$ is continuously differentiable in the neighborhood of $\boldsymbol{x}$. Otherwise the rate of convergence will be not as good or even worse, the solution could diverge. The standard way to overcome such problems is to reduce the time step or to use adaptive time step control. If there are still convergence problems, the line search solver can be used.

The basic idea of line search The line search solver from [16] uses the following modified strategy:

$$\delta \boldsymbol{x} = \lambda \, \boldsymbol{J}^{-1} \, \boldsymbol{F} \tag{4.3}$$

$0 < \lambda \leq 1$ a scalar parameter

In other words the direction of the increment $\delta \boldsymbol{x}$ is the same like with the Newton Raphson solver but the step size can be optionally reduced by the factor λ in order to avoid divergence.

How to determine λ The parameter λ is chosen by the algorithm at each iteration in order to minimize the scalar function $f = \frac{1}{2} \boldsymbol{F}^T \boldsymbol{F}$ during the line search along the Newton direction $\delta \boldsymbol{x}$. Since the Newton direction $\delta \boldsymbol{x}$ is a descent direction for f, it is guaranteed that the residual will decrease at each iteration as long as $\nabla f \neq \boldsymbol{0}$, $\boldsymbol{J}$ is exact and $\boldsymbol{F}(\boldsymbol{x})$ is continuously differentiable in the neighborhood of $\boldsymbol{x}$.

Local minimum However it could happen that the solver converges to a local minimum of f where $\nabla f = \boldsymbol{0}$ but $\boldsymbol{F} \neq \boldsymbol{0}$. The default behavior of the solver is to bail out in such a situation unless `zero gradient`, `continue`, `yes` has been specified. This flag has been added in order to test the solver but should not be used in general. If the solver bails out because the gradient ∇f is close to zero, it is recommended to decrease either `<tolerance_min>` or to reduce the time step.

Cubic interpolation of $f(\lambda)$ Two different strategies in order to determine λ have been implemented. The original one published in [16] which uses a cubic interpolation of $f(\lambda)$ will be used if `algorithm`, `cubic` was specified in the input file. In order to avoid too small increments $\delta \boldsymbol{x}$, a value for `lambda_factor_min` can be specified.

$$\lambda_{n+1} \geq \texttt{lambda\_factor\_min} \, \lambda_n \tag{4.4}$$

Multiplication by a constant factor Also a simple algorithm, which multiplies λ by the constant factor `lambda_factor_min` at each line search iteration, can be used if `algorithm`, `factor` is specified in the input file.

The condition for backtracking According to [16] the line search is completed if

$$f \quad < \quad f_{prev} + \texttt{alpha} \, s \tag{4.5}$$

$s = (\nabla f)^T \, \boldsymbol{J}^{-1} \, \boldsymbol{F} = -\boldsymbol{F}^T \, \boldsymbol{F} < 0$: the initial slope of decrease of f is the projection of the gradient ∇f to the Newton direction $\delta \boldsymbol{x}$.

$\nabla f = -\boldsymbol{J}^T \, \boldsymbol{F}$: the gradient of f

f_{prev} : the value of f at the previous iteration

If line search is not successful If λ becomes excessively small during line search, the solution might be too close to the prediction for $\boldsymbol{x}$. In that case the solver will bail out unless `abort at lambda min`, `no` has been specified.

How to avoid excessively slow convergence In order to avoid too small increments $\delta \boldsymbol{x}$, the minimum value for λ can be specified by `lambda_min`, or `tolerance_x` can be increased. In this case a reduction of the residual at each iteration is no longer guaranteed. For that reason `abort at lambda min`, `no` should be specified and `factor` in `divergence check` should be increased.

By default also a relative test for λ according to equation 4.6 will be used unless `lambda min`, `<lambda_min>`, `relative`, `no` has been specified.

$$\begin{aligned} \lambda &\geq \max\left[\texttt{lambda\_min}, \min\left(\frac{\texttt{tolerance\_x}}{t_\lambda}, 1\right)\right] \\ t_\lambda &= \max\left[\frac{\left|\left(\boldsymbol{J}^{-1}\,\boldsymbol{F}\right)_i\right|}{\max\left(\left|\boldsymbol{x}_i\right|, \left|\dot{\boldsymbol{x}}_i\right|, 1\right)}\right]_{i=1}^{n} \end{aligned} \tag{4.6}$$

Equation 4.6 was adapted from [16]. If one wishes only the relative test, `lambda_min` $= 0$ should be specified.

How to avoid too large increments In order to avoid too large increments $\delta \boldsymbol{x}$, a scale factor γ has been introduced in [16].

$$\begin{aligned} \gamma &= \max\left[\texttt{min\_step\_scale}, \min\left(\frac{p}{||\boldsymbol{J}^{-1}\,\boldsymbol{F}||}, 1\right)\right] \\ p &= \texttt{max\_step\_size}\,\max\left(\sqrt{\boldsymbol{x}^T\,\boldsymbol{x} + \dot{\boldsymbol{x}}^T\,\dot{\boldsymbol{x}}}, n\right) \\ \delta\boldsymbol{x} &= \gamma\,\lambda\,\boldsymbol{J}^{-1}\,\boldsymbol{F} \end{aligned} \tag{4.7}$$

n The dimension of $\boldsymbol{x}$

p The maximum step size

By specifying smaller values for `max_step_size` one can force the solver to search in the proximity of the prediction for $\boldsymbol{x}$. However the rate of convergence could be reduced in that way.

Solver diagnostic Like for other nonlinear solvers, the convergence can be printed to the standard output by means of a `output: iterations` statement in the initial value section. Moreover this solver prints also the line search history if `print convergence info`, `yes` was specified. This could be helpful in order to decide whether to increase or decrease `lambda_min` or the maximum number of iterations. If one wants detailed information about the decisions of the solver, `verbose`, `yes` should be specified.

General notes If one specifies `abort at lambda min`, `no`, `scale Newton step`, `no`, `divergence check`, `no` and `lambda_min` $= 1$, one gets the same behavior like Newton Raphson. Moreover if `lambda_min` < 1 and `algorithm`, `factor` was specified, the behavior will be similar to a damped Newton method.

Matrix free line search variant A matrix free variant of the line search algorithm is based on the Broyden Fletcher Goldfarb Shanno algorithm and QR updates of the Jacobian. The `bfgs` algorithm has also been adapted from [16]. It can be used only together with the `qr` or the `spqr` linear solvers. If the solution is highly nonlinear but sufficiently continuous, then the number of Jacobian evaluations of the `bfgs` solver is usually significanlty lower than the `line search` solver.

MCP solvers A Mixed (nonlinear) Complementarity problem involves the solution of a combined system of equations and inequalities.

$$\begin{aligned} \mathbf{F}_e(\mathbf{x}) &= \mathbf{0} && (4.8)\\ \mathbf{F}_i(\mathbf{x}) &\geq \mathbf{0} && (4.9)\\ \mathbf{x}_i &\geq \mathbf{0} && (4.10)\\ \mathbf{F}_i(\mathbf{x})^T \mathbf{x}_i &= 0 && (4.11)\\ \mathbf{F}(\mathbf{x}) &= \begin{pmatrix} \mathbf{F}_e(\mathbf{x}) \\ \mathbf{F}_i(\mathbf{x}) \end{pmatrix} && (4.12)\\ \mathbf{x} &= \begin{pmatrix} \mathbf{x}_e \\ \mathbf{x}_i \end{pmatrix} && (4.13) \end{aligned}$$

- $\mathbf{F}_e$ ... equations
- $\mathbf{F}_i$ ... inequalities
- $\mathbf{x}_e$ ... regular variables
- $\mathbf{x}_i$ ... complementary variables

Typical applications involve unilateral contact between rigid bodies (`module-uni_in_plane`) or the solution of the Reynolds equation based on a mass conserving cavitation model (`module-hydrodynamic_plain_bearing2`). In order to solve a MCP problem, it is required to add the statement `enable mcp, yes` to MCP capable elements and to use a MCP capable nonlinear solver. Nevertheless, MCP solvers could be used also to solve non MCP problems. All MCP capable solvers allow the following additional arguments:

- `mcp_tolerance` If the magnitude of a complementary variable is smaller than `mcp_tolerance`, it is considered as zero.
- `mcp_rho` Threshold for choosing the descent direction
- `mcp_sigma` Threshold for backtracking
- `mcp_p` Coefficient for choosing the descent direction

Trilinos NOX solver *Author: Reinhard Resch*

The `NOX` solver is based on Sandias Trilinos library [18]. Support for Trilinos must be enabled during compilation by the `--with-trilinos` flag. Although `NOX` can be compiled with `MPI` parallelism, MBDyn currently does not support a `MPI` based solution using `NOX`. Indeed `NOX` is a huge library with many parameters and options. Within this section, only those parameters which are used by MBDyn, are listed. For a complete reference of the `NOX` solver see also NOX-website.

```
<nonlinear_solver_name> ::= nox

<nonlinear_solver_data> ::=
    [ { true
    | modified , (integer) <outer_iter_before_ass>
        [ , keep jacobian matrix ]
        [ , honor element requests ] } ]
    [ , linear solver, { gmres | cg | cgs | tfqmr | bicgstab } ]
    [ , linear solver tolerance, (real) <linsol_tol> ]
    [ , linear solver max iterations, (integer) <linsol_max_iter> ]
```

```
[ , krylov subspace size, (integer) <krylov_subspace> ]
[ , inner iterations before assembly, (integer) <inner_iter_before_ass> ]
[ , weighted rms relative tolerance, (real) <wrms_rel_tol> ]
[ , weighted rms absolute tolerance, (real) <wrms_abs_tol> ]
[ , forcing term, { constant | type 1 | type 2 } ]
[ , forcing term min tolerance, (real) <forcing_term_min_tol> ]
[ , forcing term max tolerance, (real) <forcing_term_max_tol> ]
[ , forcing term alpha, (real) <alpha> ]
[ , forcing term gamma, (real) <gamma> ]
[ , use preconditioner as solver, { yes | no } ]
[ , jacobian operator, { newton | newton krylov } ]
[ , direction, { newton |
                 steepest descent |
                 nonlinear cg |
                 broyden } ]
[ , solver, { line search based |
              trust region based |
              inexact trust region based |
              tensor based } ]
[ , line search method, { backtrack |
                          polynomial |
                          more thuente } ]
[ , sufficient decrease condition { armijo goldstein | ared pred }]
[ , line search max iterations (integer) <line_search_max_iter> ]
[ , minimum step, (real) <minimum_step> ]
[ , recovery step, (real) <recovery_step> ]
[ , recovery step type, { constant | last computed step } ]
[ , verbose, { yes | no } ]
[ , print convergence info, { yes | no } ] ;
```

Linear solver and inner iterations If the option `direction` is set to `newton`, an inexact Newton method is applied. This method uses iterative linear solvers to compute the Newton direction. The following linear solvers provided by AztecOO are supported:

`gmres` Restarted generalized minimal residual

`cg` Conjugate gradient (not applicable if the Jacobian is unsymmetrical which is the usual case)

`cgs` Conjugate gradient squared

`tfqmr` Transpose-free quasi-minimal residual

`bicgstab` Bi-conjugate gradient with stabilization

The `gmres` solver is accepting also the parameter `<krylov_subspace>` which defines the number of inner iterations before a restart.

Preconditioner All linear solvers from the previous paragraph require proper preconditioning. For that purpose `NOX` uses MBDyn's direct linear solvers defined in section 4.1.1. Since an update of the preconditioner might be expensive, it can be updated only after a fixed number of outer iterations defined by the parameter `<outer_iter_before_ass>`, after the number of inner iterations exceeded

<inner_iter_before_ass> or on request by elements if honor element requests is set. At the moment only the newton krylov Jacobian operator and the broyden direction are considering the parameter <outer_iter_before_ass>. If the maximum number of inner iterations (<linsol_max_iter>) is exceeded, an update of the preconditioner is also enforced. If the flag use preconditioner as solver is set, NOX does not use any iterative solver. Instead MBDyn's own linear solvers from section 4.1.1 are applied directly.

Convergence criteria for inner iterations The convergence criteria of the linear solver is called forcing term. If forcing term is set to constant, the parameter <linsol_tol> defines the tolerance of the linear solver. Otherwise the actual tolerance can be adjusted automatically by the solver between <forcing_term_min_tol> and <forcing_term_max_tol>. In case of forcing term type 2, the parameters <alpha> and <gamma> can be used to tune the linear solver. See also NOX::Direction::Newton for further information.

Convergence criteria for outer iterations In addition to the convergence criteria defined in section 4.1.1, NOX also supports the parameters <wrms_rel_tol> and <wrms_abs_tol> which are based on the weighted root mean square norm for the solution update between iterations. See also NOX::StatusTest::NormWRMS for further information.

Jacobian operator NOX is able to evaluate a Jacobian vector product, required for all iterative linear solvers, without the need to assemble the expensive Jacobian sparse matrix. This option will be used if jacobian operator is set to newton krylov. In contrast to the matrix free Jacobian operator implemented in the Trilinos library, which is based on a finite difference approximation, MBDyn implements an accurate and inexpensive automatic differentiation approach. If jacobian operator is set to newton, the sparse Jacobian matrix is used to evaluate the Jacobian vector product.

Newton like directions

newton Uses traditional Newton step. See NOX::Direction::Newton.

steepest descent See NOX::Direction::SteepestDescent

nonlinear cg Uses a nonlinear conjugate gradient method. See NOX::Direction::NonlinearCG

broyden Uses a limited memory Broyden update which can be very efficient in some cases. See NOX::Direction::Broyden

Solvers The following solvers are supported by NOX:

line search based See NOX::Solver::LineSearchBased

trust region based See NOX::Solver::TrustRegionBased

inexact trust region based See NOX::Solver::InexactTrustRegionBased

tensor based See NOX::Solver::TensorBased

Line search solver The line search based solver also supports the following options for backtracking:

backtrack See NOX::LineSearch::Backtrack

polynomial See NOX::LineSearch::Polynomial

more thuente See NOX::LineSearch::MoreThuente

line search max iterations Maximum number of residual function evaluations.

minimum step Minimum acceptable step length (e.g. <minimum_step> = 1 means full Newton step)

recovery step Step to take when the line search fails

recovery step type Determines the step size to take when the line search fails

Ţhe more thuente and polynomial option also allow different options for the sufficient decrease condition.

armijo goldstein See Armijo-Goldstein Condition

ared pred See Ared/Pred Condition

Diagnostic output

verbose Output only warning messages from the NOX solver

print convergence info Print full status information for inner and outer iterations

Matrix free. *Author: Giuseppe Quaranta*

```
<nonlinear_solver_name> ::= matrix free

<nonlinear_solver_data> ::= { bicgstab | gmres }
    [ , tolerance , <tolerance> ]
    [ , steps , <steps> ]
    [ , tau , <tau> ]
    [ , eta , <eta> ]
    [ , preconditioner ,
       { full jacobian matrix [ , steps , <steps> ]
           [ , honor element requests ] } ];
```

where tolerance is the iterative linear solver tolerance; steps is the maximum number of linear solver iterations; tau is a measure of the configuration perturbation used to obtain a matrix-free approximation of the product of the Jacobian matrix times the solution vector; it is seldom necessary to change the default value of this parameter, mainly for very "stiff" problems. By setting eta the convergence requirements are changed; further reference for this parameter may be found in [19]; do not change it unless you know what you are doing. The value steps of the steps keyword of the preconditioner sub-block sets the maximum number of iterations that can be performed without requiring an update of the preconditioner; this parameter can heavily influence the performance of the matrix-free solver. If the option honor element requests is selected, the preconditioner is updated also when an element changes the structure of its equations. The default behavior is to ignore such requests[2]. This nonlinear solver is experimental.

[2]See note above

Eigenanalysis

```
<card> ::= eigenanalysis :
    { <when> | list , <num_times> , <when> [ , ... ] }
    [ , { suffix width , { compute | <width> }
        | suffix format , " <format> "
        | output [ full ] matrices
        | output sparse matrices
        | output eigenvectors
        | output geometry
        | matrix output precision, <matrix_precision>,
        | results output precision, <results_precision>,
        | parameter , <param>
        | mode, <mode_options>
        | lower frequency limit , <lower>
        | upper frequency limit , <upper>
        | <method> }
        [ , ... ] ]
<method> ::=
    { use lapack [ , balance , { no | scale | permute | all } ]
        | use arpack , <nev> , <ncv> , <tol> [ , max iterations, <max_iter> ]
        | use jdqz , <nev> , <ncv> , <tol>
        | use external }
<mode_options> ::=
    { largest magnitude      | smallest magnitude |
      largest real part      | smallest real part |
      largest imaginary part | smallest imaginary part }
```

Performs the direct eigenanalysis of the problem. This functionality is experimental; see [20] for theoretical aspects. Direct eigenanalysis based on the matrices of the system only makes sense when the system is in a steady configuration, so the user needs to ensure this configuration has been reached. Moreover, not all elements currently contribute to the Jacobian matrix of the system, so YMMV. In case of rotating systems, a steady configuration could be reached when the model is expressed in a relative reference frame, using the `rigid body kinematics` card (see Section 5.3.18).

- `when` indicates at what time the eigenanalysis is performed. The analysis is performed at the first time step for which `Time` $\geq$ `when`. If the keyword `list` is given, `num_times` values of `when` are expected. At most `num_times` eigenanalyses are performed; the output file name is composed using the position of the corresponding `when` in the array of times; if `when` $<$ `initial_time` the first eigenanalysis is performed before the "derivatives" phase; if `when` $=$ `initial_time` the first eigenanalysis is performed after the "derivatives" phase and before the first regular step;

- `width` is the width of the field that contains the eigenanalysis number in the related output file name; the resulting number is prefixed with zeros; when `compute` is used, the width is set in order to accommodate the largest number;

- `format` is the format that is going to be used; it must contain exactly one format specification for an integer; for example, to have zero-prefixed, three digit file names, use `"%03d"`;

- `param` is the value of the coefficient that is used to project the problem in the discrete time domain; it represents a singularity point in the transformation from the discrete to the continuous eigenvalues,

so it should not coincide with an eigenvalue of the problem; by default the value is estimated automatically if `lower frequency limit` or `upper frequency limit` are defined;

$$\texttt{<param>} = \begin{cases} \frac{1}{2\,\pi\,5\,\texttt{<upper>}} & \text{if } \texttt{<mode>} = \texttt{smallest magnitude} \\ \frac{5}{2\,\pi\,\texttt{<lower>}} & \text{if } \texttt{<mode>} = \texttt{largest imaginary part} \\ 1 & \text{otherwise} \end{cases}$$

- `mode` Krylov subspace methods like ARPACK can compute only a subset of available eigenmodes. With this option the user may specify the eigenvalues Λ_k which should be computed; depending on the value of `<param>`, the smallest magnitude of Λ_k is not necessarily the smallest magnitude of the corresponding physical eigenvalue λ_k; see also 4.15; by default `largest imaginary part` is used for ARPACK and `smallest magnitude` is used for LAPACK and JDQZ;

- `output full matrices` causes the output of the (full) problem matrices in a `.m` file, intended to be loaded by tools like Octave/Matlab. If NetCDF output is enabled with the related card (see Section 5.3.11), the matrices along with the eigenanalysis parameters are written in an `.nc` file as described in Section C.1.5;

- `output sparse matrices` causes the output of the problem matrices in sparse form in a `.m` file, intended to be loaded by tools like Octave/Matlab, and in the NetCDF `.nc` file as described in Section C.1.5, if NetCDF output is enabled;

- `output eigenvectors` causes the output of the eigenvalues and of the eigenvectors in a `.m` file, intended to be loaded by tools like Octave/Matlab, and/or in the NetCDF `.nc` file as described in Section C.1.5, if NetCDF output is enabled;

- `output geometry` causes the output of the reference configuration of structural nodes, if any, and of the indexes that need to be used in order to extract the perturbation of their position and orientation from the eigenvectors;

- `matrix output precision` sets the precision for the text output of the problem matrices in the `.m` file to `matrix_precision`;

- `results output precision` sets the precision for the text output of the eigenanalysis results in the `.m` file to `results_precision`;

- `use lapack` performs the eigenanalysis using LAPACK's `dggev()` routine (generalized unsymmetric eigenanalysis using the QZ decomposition);

- `balance` performs matrix permutation (`permute`), scaling (`scale`), both (`all`) using LAPACK's `dggevx()` routine, or none (`no`) using `dggev()`; by default, matrix permutation is performed, using `dggevx()`;

- `use arpack` performs the eigenanalysis using ARPACK's `dnaupd()` and `dneupd()` routines (canonical unsymmetric eigenanalysis, using the Implicitly Restarted Arnoldi Method, IRAM). It requires the additional parameters:

  - `nev`, the number of the desired eigenvalues;
  - `ncv`, the number of the Arnoldi vectors;
  - `tol`, the tolerance (positive; zero means machine error).
  - `max_iter`, the maximum number of iterations to perform (default 300)

- `use jdqz` performs the eigenanalysis using JDQZ (Jacobi-Davidson QZ decomposition). It requires the same parameters of ARPACK; actually, it allows a lot of tuning, but its input is not formalized yet (experimental).
- `use external` is a placeholder to tell MBDyn not to perform any eigenanalysis but rather honor any of the "output" statements that can be honored without performing the analysis; for example, writing the matrices (`output matrices`, `output sparse matrices`), the geometry (`output geometry`), and so on;
- `lower frequency limit` only outputs those eigenvalues and eigenvectors that are complex and whose imaginary part is larger than `lower` Hz;
- `upper frequency limit` only outputs those eigenvalues and eigenvectors that are complex and whose imaginary part is smaller than `upper` Hz;

The `eigenanalysis` functionality is experimental, so the syntax could need adjustments and change across versions.

Note: in practical cases, the use of lapack with `balance` appeared to be quite sensitive to the problem data. In some cases, the use of `scale` may introduce errors in eigenvalues. The use of `permute` is usually beneficial: it allows to directly single out some of those spurious eigenvalues that correspond to constraint equations.

Note: in practical cases, the usefulness of arpack appeared to be questionable, especially in those cases where a significant amount of constraints is present. It may be beneficial in those cases where the number of deformable components is much larger than the number of constraint equations.

Output A brief summary of the eigenvalues is output in the `.out` file. Most of the information is dumped in a `.m` file in a format that is suitable for execution with scientific software like Octave and Matlab. The execution of the file populates the environment of those software with a set of variables that contains the results of the eigenanalysis.
Binary output in NetCDF format, if supported, is written in the `.nc` file. Refer to Section 5.3.11 for details on how to activate NetCDF output and to Section C.1.5 for a complete description of the binary output.
According to the options used in the `eigenanalysis` card, the `.m` file may contain:

- `Aplus`: matrix $\mathbf{J}_{(+c)} = \boldsymbol{f}_{/\dot{\boldsymbol{x}}} + \texttt{dCoef} \cdot \boldsymbol{f}_{/\boldsymbol{x}}$, either in full or sparse format;
- `Aminus`: matrix $\mathbf{J}_{(-c)} = \boldsymbol{f}_{/\dot{\boldsymbol{x}}} - \texttt{dCoef} \cdot \boldsymbol{f}_{/\boldsymbol{x}}$, either in full or sparse format;
- `dCoef`: the coefficient used to build the matrices $\mathbf{J}_{(+c)}$ and $\mathbf{J}_{(-c)}$;
- `alpha`: an array of three columns containing $\alpha_r =$ `alpha(:, 1)`, $\alpha_i =$ `alpha(:, 2)` and $\beta =$ `alpha(:, 3)`, such that the k-th discrete-time eigenvalue Λ_k is

$$\Lambda_k = \frac{\alpha_r(k) + \mathrm{i}\alpha_i(k)}{\beta(k)}. \tag{4.14}$$

  Note that α_r, α_i and β can be all zero; only a fraction of the eigenvalues may be output; the corresponding continuous time domain eigenvalue λ_k is

$$\lambda_k = \frac{1}{\texttt{dCoef}} \frac{\Lambda_k - 1}{\Lambda_k + 1} = \frac{1}{\texttt{dCoef}} \frac{\alpha_r(k)^2 + \alpha_i(k)^2 - \beta(k)^2 + \mathrm{i}2\alpha_i(k)\beta(k)}{\left(\alpha_r(k) + \beta(k)\right)^2 + \alpha_i(k)^2}, \tag{4.15}$$

  where `dCoef` is defined in the `.m` file;

- `VR`: the matrix of the right eigenvectors, $\mathbf{V}_R$, solution of the problem

$$\left(\mathbf{J}_{(+c)} - \Lambda(\mathtt{k})\mathbf{J}_{(-c)}\right)\mathbf{V}_R(:,\mathtt{k}) = \mathbf{0}$$

  only the eigenvectors corresponding to the selected eigenvalues are output;

- `VL`: the matrix of the left eigenvectors, $\mathbf{V}_L$, solution of the problem

$$\left(\mathbf{J}^T_{(+c)} - \mathrm{conj}(\Lambda(\mathtt{k}))\mathbf{J}^T_{(-c)}\right)\mathbf{V}_L(:,\mathtt{k}) = \mathbf{0}$$

  only the eigenvectors corresponding to the selected eigenvalues are output;

- `X0`: a vector containing the reference position and orientation of the structural nodes, if any (excluding any dummy node); the vector contains, for each node, the three components of the reference position and the three components of the Euler vector corresponding to the reference orientation; all data are referred to the absolute reference frame;

- `idx`: a vector containing the reference row index of each structural node in the eigenvectors;

- `labels`: a vector containing the labels of each structural node, if any (excluding any dummy node);

Caveat: the contents of this file may change across versions.

Interpretation of linearized problem. The typical interpretation of the linearized problem

$$\boldsymbol{f}_{/\dot{\boldsymbol{x}}}\Delta\dot{\boldsymbol{x}} + \boldsymbol{f}_{/\boldsymbol{x}}\Delta\boldsymbol{x} = \mathbf{0} \tag{4.16}$$

in descriptor form is

$$\mathbf{E}\Delta\dot{\boldsymbol{x}} = \mathbf{A}\Delta\boldsymbol{x} \tag{4.17}$$

with

$$\mathbf{E} = \boldsymbol{f}_{/\dot{\boldsymbol{x}}} = \frac{\mathbf{J}_{(+c)} + \mathbf{J}_{(-c)}}{2} \tag{4.18a}$$

$$\mathbf{A} = -\boldsymbol{f}_{/\boldsymbol{x}} = -\frac{\mathbf{J}_{(+c)} - \mathbf{J}_{(-c)}}{2c} \tag{4.18b}$$

Example.

```
# to access the "x" component of each node of each eigenvector
octave:1> VR(idx + 1, :)
# to plot the "y" component of each node of each eigenvector,
# with respect to the reference "x" component of the position of the nodes
octave:2> plot(X0(1:6:end), VR(idx + 2, :))
# to plot all components of eigenvector 5 in 3D,
# with an 'x' in the undeformed position
# and a 'o' in the deformed position, scaled by 0.1
octave:3> mode = 5; scale = 0.1;
octave:4> plot3(X0(1:6:end), X0(2:6:end), X0(3:6:end), 'x',
                X0(1:6:end) + scale*VR(idx + 1, mode), ...
                X0(2:6:end) + scale*VR(idx + 2, mode), ...
                X0(3:6:end) + scale*VR(idx + 3, mode), 'o')
```

Abort After

```
<card> ::= abort after :
    { input
        | assembly
        | derivatives
        | dummy steps
        | regular step , (integer) <step_number> } ;
```

mainly used to cleanly check models and simulations at various phases. When set to `input`, the simulation ends after input is completed; when set to `assembly`, the simulation ends after initial assembly, if required; when set to `derivatives`, the simulation ends after the computation of the derivatives, i.e. after the system is solved at the initial time to compute initial momentum and momenta moment derivatives and reaction forces; when set to `dummy steps`, the simulation ends after the dummy steps execution, if required. When set to `regular step` and a single step method is used, then the simulation ends after `<step_number>` steps. However, if a multistep method is used, then the simulation ends after `<step_number>` plus the number of startup steps (e.g. `<step_number>` + 1 for `ms2`, `<step_number>` + 2 for `ms3`, `<step_number>` + 3 for `ms4`). In any case, the output is generated with the system configured as resulting from the last computation phase, so this mode can be useful to check how the system is actually configured after phases that are not usually output.

Linear Solver

```
<card> ::= linear solver :
        { naive | umfpack | klu | y12 | lapack | superlu | taucs
        | pardiso | pardiso_64 | watson | pastix | qr | spqr
        | aztecoo | amesos | siconos dense | siconos sparse }
    [ , { map | cc | dir | grad } ]
    [ , { colamd | mmdata | amd | given | metis } ]
    [ , { mt | multithread } , <threads> ]
    [ , workspace size , <workspace_size> ]
    [ , pivot factor , <pivot_factor> ]
    [ , drop tolerance , <drop_tolerance> ]
    [ , block size , <block_size> ]
    [ , scale , { no | always | once | row max | row sum | column max | column sum
            | lapack | iterative | row max column max }
        [ , scale tolerance, (real) <scale_tolerance> ]
        [ , scale iterations, (integer) <scale_max_iter> ]
    ]
    [ , tolerance, (real) <refine_tolerance> ]
    [ , max iterations, (integer) <refine_max_iter> ]
    [ , preconditioner, { umfpack | klu | lapack | ilut | superlu | mumps |
        scalapack | dscpack | pardiso | paraklete | taucs | csparse } ]
;
```

Umfpack. The default, when available, is `umfpack`. Since it dynamically allocates memory as required, the `workspace size` parameter is ignored. The `umfpack` linear solver honors the `block size` keyword, which refers to an internal parameter. The default value (32) is usually good for most problems; however, it has been observed that a lower value can slightly improve performances on small problems (e.g. a rigid rotorcraft model with some 180 degrees of freedom showed a 10% speedup with a block size of 4, while a

rigid robot showed even higher speedups, of the order of 30%, with a block size between 4 and 16). With respect to scaling, the `scale` option supports `max` and `sum` which respectively scale the matrix according to each row's largest absolute value and to the sum of the absolute values of the coefficients in the row (the latter is the default), and `always` and `once` which scale the matrix according to Lapack's `dgeequ(3)` algorithm respectively all times or with factors computed the first time the matrix is factored.

Naive. The `naive` solver is built-in, so it is always present. When `umfpack` is not available, it is used by default. The naive solver uses a full storage but keeps track of the non-zeros. This allows a very efficient solution of sparse matrices, with a $O(1)$ access cost to the coefficients. It ignores the `workspace size` parameter. See [21] for details.

> **Important note:** the naive solver should always be used with some reordering option. The `colamd` option is always available (see discussion below), since it is built-in. Other options are experimental, and should be used with care. If this option is not used, the factorization may not be robust. If the `linear solver` statement is not used, and the `naive` solver is used by default (because no other solver with higher priority is present), **no reordering option is set by default.** In this case, it is safe to explicitly force the linear solver with a valid reordering option. For example
>
> ```
> linear solver: naive, colamd;
> ```

The Naive solver supports matrix scaling using Lapack's `dgeequ`'s function. Scaling may occur `always`, i.e. for each factorization, or `once`, thus preserving the scaling factors resulting from the analysis of the first matrix that is factored.

KLU. The `klu` solver is provided by University of Florida's SparseSuite. It is very efficient (usually faster than `umfpack`), but it should be used with care, because in the current implementation, to exploit its efficiency, the symbolic factorization is reused across factorizations. Whenever the solver realizes that the symbolic factorization needs to be regenerated, because the structure of the matrix changed (e.g. when using `driven` elements, but not only in that case), the matrix is already invalidated, and should be regenerated. This is not handled correctly yet. Matrix scaling can be performed `always` or `once` according to Lapack's `dgeequ(3)` algorithm respectively all times or with factors computed the first time the matrix is factored

Y12. The `y12` linear solver is provided for historical reasons and should not be used for practical purposes. It requires static allocation of the workspace; the optimal size `workspace_size` is between 3 and 5 times the number of nonzero coefficients of the sparse matrix. By default, twice the size of the full matrix is allocated (which sounds a bit nonsense, but is conservative enough); then the linear solver automatically uses an optimal amount of memory based on the results of the previous factorization; this could result in a failure if the filling of the matrix changes dramatically between two factorizations.

Lapack. The `lapack` linear solver uses LAPACK's `dgetrf()` and `dgetrs()` functions with a dense storage, and thus ignores the `workspace size` parameter. It is fairly inefficient for most practical applications, when compared to sparse solvers, except when problems are very small (less than 60 equations) and very dense.

SuperLU. Finally, `superlu` is an experimental linear solver that is able to perform the factorization in a multi-threaded environment. If more than one CPU is available, it automatically uses all the CPUs; the keyword `mt` allows to enforce the desired number of `threads`. See also the `threads` keyword for more details on multi-threaded solution.

PARDISO PARDISO is a multi-threaded sparse direct solver developed at the University of Basel [22]. MBDyn's `pardiso` and `pardiso_64` solvers provide interfaces to the PARDISO variant included in Intel's MKL library. See also MKL-PARDISO-website. `pardiso_64` uses 64-bit integers for array indices whereas `pardiso` uses 32-bit integers. By default, automatic scaling and non-symmetric weighted matching are activated for this solver. Nevertheless, `pardiso` and `pardiso_64` should not be used without iterative refinement enabled, because they are using static pivoting.

Watson Watson Sparse Matrix Package (WSMP) is a parallel linear solver for large sparse system of equations developed by IBM [23]. See also IBM-WSMP-website.

PaStiX The `pastix` linear solver from the Institut National de Recherche en informatique et Automatique is a high performance parallel solver for very large sparse linear systems based on direct methods [24]. See also PaStiX-gitlab. It is recommended to enable iterative refinement for this solver because it uses static pivoting.

QR The `qr` linear solver is a dense linear solver based on $Q\,R$ decomposition instead of $L\,U$. See the book Numerical Recipes in C [16] for further information. It should be used mainly in combination with the `bfgs` nonlinear solver for small Jacobian matrices but can be used like any other linear solver.

SPQR The SuiteSparseQR solver `spqr` from [25] is an implementation of the multifrontal sparse $Q\,R$ factorization method. It can be used in combination with the `bfgs` nonlinear solver for larger Jacobian matrices which cannot be handled efficiently by `qr`. The `spqr` solver is using a sparse representation of the Jacobian and it's $Q\,R$ factors as long as no BFGS update is applied. Once an BFGS update is applied to the $Q\,R$ factors, a dense upper triangular solver from Lapack is used until the Jacobian is rebuild. However, usually the most expensive part is the $Q\,R$ decomposition of the sparse Jacobian.

Aztecoo In contrast to most other solvers, `aztecoo` is an iterative linear solver based on Sandia's Trilinos library [18]. Different preconditioners are supported. However the incomplete LU factorization option `ilut` cannot be used if there are any joints in the model. This solver accepts a tolerance defined by `<refine_tolerance>` and a maximum number of iterations `<refine_max_iter>`.

Amesos Amesos is also based on Sandia's Trilinos library [18]. It provides interfaces to several direct linear solvers.

Siconos dense, Siconos sparse Siconos is a library for the solution of nonsmooth systems [17]. In order to use the `siconos mcp newton fb` or `siconos mcp newton min fb` nonlinear solver, it is required to choose the `siconos dense` linear solver.

Other. Historically, MBDyn supported the `meschach` and `harwell` solvers. `Meschach` has been removed, while `harwell` is now deprecated and undocumented. Other linear solvers may be supported for experimental reasons (for example, `taucs`, `watson`); however, they are not documented as they are not intended for general use.

Miscellaneous keywords. The keywords `map` (Sparse Map), `cc` (Column Compressed) and `dir` (Direct Access) can be used to modify the way the sparse matrix is handled:

- `map` uses an array of binary trees to store the matrix coefficients; right before the factorization, the matrix is transformed in the form required by each linear solver, usually Column Compressed (also

known as Harwell-Boeing format) or Index (as used by y12). The matrix is regenerated each time an assembly is required.

- cc uses the compact format resulting from the transformation required by the linear solver to store the coefficients in subsequent assemblies, thus saving the time required to transform the matrix. If the profile of the matrix changes, the compact form is invalidated, and the matrix is reset to map form. As a consequence, this matrix form allows faster access and reduced factorization cost, but in certain cases may require an additional cost.

- dir uses a full index table to grant direct access time to the actual location of each coefficient. In principle it should be a bit faster than cc, at the cost of some memory waste. However, in practical applications, the cost of these operations is a very small fraction of the factorization time. If the problem is quite large, the memory consumption might become unacceptable; if the problem is quite small, the access time of cc is quite limited ($\log_2$ (<nz>), where nz is the number of non-zeroes in a column) so it is definitely comparable to that of cc.

- grad uses a compressed sparse vector representation for each row of the matrix. It is available only if use automatic differentiation was enabled according section 5.1.

Only umfpack, klu, y12, superlu and pastix linear solvers allow these settings.

The keyword colamd is honored by the naive, spqr and SuperLU linear solvers; it enables the column rearrangement that minimizes the sparsity of the factored matrix, as implemented in libcolamd (part of University of Florida's SparseSuite package). The keyword mmdata is honored by the scalar SuperLU linear solver only; it enables column rearrangement based on the MMD ordering of the symmetric matrix $A^T A$. Additional options are amd, metis and given ordering which are supported by the spqr linear solver.

The pivot_factor is a real number, which in a heuristic sense can be regarded as the threshold for the ratio between two coefficients below which they are switched, so 0.0 means no pivoting, while 1.0 means switch as soon as the norm of the ratio is less than unity.

The keyword drop tolerance enables incomplete factorization. Elements of the factorization whose absolute value is below the drop_tolerance threshold are discarded. By default it is 0. Only umfpack allows this setting.

The keyword block size is specific to umfpack. See the related documentation for the meaning.

The keyword scale indicates whether the matrix, the right-hand side and the solution must be scaled before/after linear solution. By default no scaling takes place (no). Only naive, klu, umfpack and pastix allows this setting. It can be used in order to improve the condition number of the Jacobian matrix. See also section 4.1.4. If one of the keywords row max, column max, row sum or column sum is used, either rows or columns of the Jacobian are scaled by their maximum absolute value or sum of absolute values. In order to scale rows and columns of the Jacobian at the same time, one can choose between Lapacks dgeequ algorithm (lapack), the Sinkhorn-Knopp algorithm [26] (row max column max) and the algorithm from Amestoy et.al. [27] (iterative).

In case of a high condition number of the Jacobian matrix, one should enable iterative refinement by means of the <refine_max_iter> option. It is supported by the umfpack and pastix solvers.

Table 4.1 summarizes the properties that can be set for each type of linear solver; Table 4.2 summarizes the memory usage of the linear solvers; Table 4.3 summarizes how each linear solver deals with pivoting policies.

Solver

Deprecated; see linear solver (Section 4.1.1).

Table 4.1: Linear solvers properties

Solver	***Map***	***Column Compr.***	***Dir***	***Grad***	***Multi-Thread***	***Worksp. Size***	***Pivot Factor***	***Block Size***
Naive					(√)		√	
Umfpack	√	√	√	√			√	√
KLU	√	√	√	√			√	
Y12m	√	√	√			√	√	
Lapack							√	
SuperLU	√	√	√		√		√	
TAUCS	?	?	?		?	?	?	√
Pardiso	√			√	√			
Watson	√	√	√	√	√		√	
Pastix	√	√	√	√	√			
QR								
SpQR	√			√				
Siconos sparse						√		

Table 4.2: Linear solvers memory usage

Solver	***Workspace***	***Memory Size***	***Block Size Allocation***
Naive		full	
Umfpack		dynamic	default=32
KLU		dynamic	
Y12m	default=$2 \times n^2$ $3nz \rightarrow 5nz$	static	
Lapack		full	
SuperLU		dynamic	
TAUCS		?	
Watson		dynamic	
Pastix		dynamic	
QR		full	
SpQR		dynamic	

Table 4.3: Linear solvers pivot handling

Solver	***Pivot***	***Default***	***Description***
Naive	1.0→0.0	1.e−5	
Umfpack	0.0→1.0	0.1	
KLU	0.0→1.0	0.1	
Y12m	boolean: none=0.0, full=1.0	full	
Lapack	1.0→0.0		
SuperLU	0.0→1.0	1.0	
TAUCS			
Watson			

Threads

```
<card> ::= threads :
    { auto | disable | [ { assembly | solver } , ] <threads> }
```

By default, if enabled at compile time, the assembly is performed in a multi-threaded environment if more than one CPU is available and the selected solver allows it (e.g. if it supports any form of compressed matrix handling, `cc` or `grad`). However, this behavior can be influenced by the `threads` directive. The value `auto` is the default; the value `disable` reverts to the usual scalar behavior; otherwise `threads`, the desired number of threads, is read and used. If it is prefixed by the keyword `assembly`, the desired number of threads is used only in assembly; if it is prefixed by the keyword `solver`, the desired number of threads is used only during factorization/solution, if the linear solver supports it (currently, only `naive` and `superlu` solvers do). As an example, on a 4 CPU architecture, to use 2 threads for assembly and 4 threads for solution, use

```
threads: assembly, 2;
solver: superlu, cc, mt, 4;
```

4.1.2 Derivatives Data

Data related to the 'derivatives' phase. Right after the initial assembly and before the simulation starts, the so-called derivatives solution is performed. The system is solved with the kinematic unknowns constrained, in order to properly determine the dynamic unknowns, namely momenta and constraint reactions. For this purpose, the coefficient that relates the state perturbation to the derivative perturbation must be set to a value that is small enough to allow the determination of accurate derivatives with very small change in the states. This coefficient should be zero, but this leads to matrix singularity, so it must be chosen by the user, since it is highly problem dependent. A rule-of-thumb is: if the system has small stiffness and high inertia, the coefficient can be big, if the system has high stiffness and small inertia, the coefficient must be small.

The derivatives solution is always performed and cannot be disabled. If for any reason it does not converge, to practically disable it one can set a very large tolerance. Subsequent time steps may start with a less than ideal initialization, resulting in a rough transient.

Derivatives Tolerance

```
<card> ::= derivatives tolerance : <tolerance> ;
```

Derivatives Max Iterations

```
<card> ::= derivatives max iterations : <max_iterations> ;
```

Derivatives Coefficient

```
<card> ::= derivatives coefficient : <coefficient>
    | [ <coefficient> , ] auto
        [ , max iterations , <max_iterations> ]
        [ , factor , <factor> ]
    ;
```

4.1.3 Dummy Steps Data

Data related to the 'dummy steps' phase. The dummy steps are intended as a sort of numerical computation of the second order derivatives of the constraint equations. The displacement constraint equations in an index three Differential-Algebraic Equations system (DAE) represent the second derivative effect of the constraints on the kinematic unknowns. Thus, to ensure the proper initial conditions, the constraint equations should be differentiated twice. To simplify the code, those equations have been differentiated once only, in the initial assembly, the second derivation being performed numerically by the dummy steps. During these steps the system converges to a solution whose error from the sought solution is bounded. For this reason, the dummy steps should be performed with a very short time step, to seek accuracy, and with a high numerical damping, to cancel as quickly as possible the high frequency numerical noise.

Note: no step change, modified Newton-Raphson and so on is allowed during the dummy steps.

Note: in most practical applications, dummy steps should not be used.

Dummy Steps Tolerance

```
<card> ::= dummy steps tolerance : <tolerance> ;
```

Dummy Steps Max Iterations

```
<card> ::= dummy steps max iterations : <max iterations> ;
```

Dummy Steps Number

```
<card> ::= dummy steps number : <number> ;
```

number of dummy steps.

Dummy Steps Ratio

```
<card> ::= dummy steps ratio : <ratio> ;
```

ratio of the time step to be used in the dummy steps to the regular time step.

Dummy Steps Method

```
<card> ::= dummy steps method : <method_data> ;
```

Same as for the normal simulation method.

4.1.4 Output Data

Model output, e.g. nodes and elements output, is handled by the data manager in the `control data` block; the solver only deals with simulation-related output, and takes care of some debug output.

Output

This command requests special output related to the solution phase.

```
<card> ::= output : <item> [ , ... ] ;

<item> ::=
    { iterations
```

```
        | residual
        | solution
        | jacobian matrix
        | messages
        | counter
        | bailout
        | matrix condition number
        | solver condition number
        | cpu time
        | none }
```

The keyword `counter` logs on the standard output a string that summarizes the statistics of the time step just completed. By default it is off; the same information, by default, is logged in the `.out` file. The keyword `iterations` logs on `stdout` a detailed output of the error at each iteration inside a time step. The keywords `residual`, `solution` and `jacobian matrix` log to `stdout` the residual, the solution and the Jacobian matrix; they are mainly intended for last resort debugging purposes, as the output can be extremely verbose. The item `messages` refers to all messaging on the `.out` file; by default, it is on. The special item `bailout` instructs the solver to output the residual (as with `residual`) only in case of no convergence, before bailing out. The special item `none` clears the output flags; the items are additive, so, for instance, to clear out the default and add the iterations output, use:

```
output: none, iterations;
```

Output meter

This command conditions the output selected by the `output` command to a meter, expressed in terms of a drive caller. Only when the drive caller returns `true` (!0), the output occurs.

```
<card> ::= output meter : (DriveCaller) <when> ;
```

4.1.5 Real-Time Execution

Initially implemented by Michele Attolico
Extensively reworked by Pierangelo Masarati
This statement forces MBDyn to be scheduled in real-time, based on the capabilities of the underlying OS.

```
<card> ::= real time : [ { RTAI | POSIX } , ]
    <mode>
    [ , reserve stack , <stack_size> ]
    [ , allow nonroot ]
    [ , cpu map , <cpu-map> ]
    [ , output , { yes | no } ]
    [ , hard real time ]
    [ , real time log [ , file name , <command_name> ] ]

<mode> ::=
    { [ mode , period , ] time step , <time_step>
    | mode , semaphore [ , <semaphore_args> ]
    | mode , IO }
```

where:

- the optional RTAI or POSIX keywords indicate whether RTAI (Linux Real-Time Application Interface, http://www.rtai.org/) or POSIX should be used to schedule the execution of MBDyn in real-time (defaults to RTAI);
- when the keyword mode is set to period, the execution is scheduled periodically; when mode is set to semaphore, the execution waits for an external trigger to start a new step; when mode is set to IO, the execution assumes synchronization to be provided by waiting on blocking I/O streams (note: semaphore is not implemented yet, so semaphore_args is currently undefined; the default mode is period);
- the time_step, required when scheduling is periodic, is the sample rate in nanoseconds (ns); usually, it matches the time step of the simulation, but it can be different for other purposes;
- allow nonroot means that the program should be allowed to run as a regular (non-root) user while performing hard real-time operations with the underlying OS (by default, only the superuser is allowed to execute in real-time, since this requires the process to acquire the highest priority; only honored by RTAI; POSIX real-time will simply ignore the scheduling and continue as a regular simulation);
- the keyword reserve stack instructs the program to statically reserve the desired stack size by means of the mlockall(2) system call; it should be used to ensure that no page swapping occurs during the real-time simulation; a minimal default value (1024 bytes) is set in any case;
- the keyword cpu map allows the program to force its execution on a specific subset of CPUs on multiprocessor hardware; the syntax of the cpu_map field is a byte-sized integer (between 0 and 255) whose active bits are the desired CPUs (up to the number of available CPUs or 4, whichever is lower);
- the keyword output determines whether output is entirely disabled or not; this option is only available to POSIX real-time, and should always be set to no in order to disable any output, so that I/O only occurs for the purpose of interprocess communication;
- the keyword hard real time instructs the program to run in hard real-time; the default is soft real-time (RTAI only; by default scheduling is in soft real-time);
- the keyword real time log enables logging by means of RTAI mailboxes; an optional file name argument allows to set the command_name of the logging process that is used. It should be a low priority soft real-time process that communicates with the simulator by means of the RTAI mailbox called logmb. The log process mailbox may receive a sequence of message consisting in two integers: the time step number and the amount of nanoseconds that time step overrun the scheduling period.

  Note: if no command_name is given, to guarantee the detection of the default log command the process PATH environment variable is augmented by prepending ".:BINPATH"; by default, BINPATH is the ${bindir} directory of the build environment, so it can be set by using the --bindir configure switch.

The keywords must be given in the sequence reported above. Real-time simulation is mostly useless without interaction with external programs. The input is dealt with by the stream file drivers described in Sections 7.1.4 and 7.1.5. The output is dealt with by the stream output elements described in Sections 8.14.1, 8.14.2, and 8.14.3.

4.2 Other Problems

Other types of problems are either possible or being developed. Typically, static problems can be obtained in terms of a downgrading of the initial value problem, by eliminating time-dependent interactions and treating time as an ordering parameter.

A special problem that is currently under development is `inverse dynamics`.

Chapter 5

Control Data

This section is read by the manager of all the bulk simulation data, namely the nodes, the drivers and the elements. It is used to set some global parameters closely related to the behavior of these entities, to tailor the initial assembly of the joints in case of structural simulations, and to tell the data manager how many entities of every type it should expect from the following sections. Historically this is due to the fact that the data structure for nodes and elements is allocated at the beginning with fixed size. This is going to change, giving raise to a "free" and resizeable structure. But this practice is to be considered reliable since it allows a sort of double-check on the entities that are inserted.

5.1 Enable support for automatic differentiation

Author: Reinhard Resch

```
<card> :: = use automatic differentiation ;
```

This keyword must be used in order to enable support for automatic differentiation in MBDyn. All the elements listed in table 5.1 can use automatic differentiation. Also elements without support for automatic differentiation may be used in a model with `use automatic differentiation` enabled.

5.2 Assembly-Related Cards

The initial assembly related cards are:

5.2.1 Skip Initial Joint Assembly

```
<card> ::= skip initial joint assembly ;
```

This directive inhibits the execution of the initial joint assembly. Note that for a model to behave correctly, the initial joint assembly should always succeed. A correct model succeeds with 0 iterations, i.e. it intrinsically satisfies the constraints from the beginning. However, the initial joint assembly is more than a simple compliance test; it represents a static preprocessor for models. See the directives `use` in Section 5.2.3 and `initial stiffness` in Section 5.2.4 for more details on performing appropriate initial joint assembly.

Table 5.1: Elements with support for automatic differentiation (AD)

element	supported with AD disabled	supported with AD enabled
beam3	√	√
modal	√	√
total joint	√	√
total pin joint	√	√
body	√	√
automatic structural	√	√
hexahedron8		√
hexahedron20		√
hexahedron20r		√
pentahedron15		√
tetrahedron10		√
pressureq4		√
pressureq8		√
pressureq8r		√
pressuret6		√
tractionq4		√
tractionq8		√
tractionq8r		√
tractiont6		√
hydrodynamic plain bearing		√
hydrodynamic plain bearing2		√
journal bearing		√
triangular contact		√
ball bearing contact		√
unilateral in plane		√

5.2.2 Initial Assembly of Deformable and Force Elements

```
<card> ::= initial assembly of deformable and force elements ;
```

This directive enables the initial assembly of deformable and force elements. Previous MBDyn versions used to assemble them by default, while they should not really cotribute to the intitial assembly. As a backward-incompatible change they do no more contribute to the initial assembly, but the previous behavior can be recovered back with this flag.

5.2.3 Use

```
<card> ::= use : <item_list> , in assembly ;

<item_list> ::= <item> [ , ... ]
<item> ::=
    { rigid bodies
        | gravity
        | forces
        | beams
        | solids
        | surface loads
        | aerodynamic elements
        | loadable elements }
```

`joints` are used by default, and cannot be added to the list. `beams` are used by default, too, but can be added to the list essentially for backward compatibility.

5.2.4 Initial Stiffness

```
<card> ::= initial stiffness : <position_stiffness> [ , <velocity_stiffness> ] ;
```

This directive affects the stiffness of the dummy springs that constrain the position and the orientation (`position_stiffness`) and the linear and angular velocity (`velocity_stiffness`) of the structural nodes during the initial assembly; the default is 1.0 for both. Note that each node can use a specific value; see Section 6.5 for details.
Note that the same value is used for the position and the orientation, so this stiffness is dimensionally inconsistent; It should really be intended as a penalty coefficient. The same considerations apply to the penalty value for linear and angular velocities.

5.2.5 Omega Rotates

```
<card> ::= omega rotates : { yes | no } ;
```

Sets whether the imposed angular velocity should be considered attached to the node or fixed in the global system during the initial assembly.

5.2.6 Tolerance

```
<card> ::= tolerance : <tolerance> ;
```

The tolerance that applies to the initial joint assembly; this tolerance is used to test the norm 2 of the residual, because it is very important, for a correct start of the simulation, that the algebraic constraints be satisfied as much as possible. The alternate statement `initial tolerance` is tolerated for backwards compatibility.

5.2.7 Max Iterations

```
<card> ::= max iterations : <max_iterations> ;
```

The number of iterations that are allowed during the initial assembly. The alternate statement `max initial iterations` is tolerated for backwards compatibility.

Note: by default, `max_iterations` is zero, i.e., the analysis fails if at least one iteration is required. Users must explicitly set this parameter to a value greater than zero if they want the solver to try and fix an inconsistent model.

5.3 General-Purpose Cards

5.3.1 Title

```
<card> ::= title : " <simulation_title> " ;
```

5.3.2 Print

```
<card> ::= print : <item> [ , ... ] ;

<item> ::=
    { dof stats
        | [ { dof | equation } ] description
        | [ { element | node } ] connection
        | all
        | none }
        [ , to file ]
```

- The keyword `dof stats` enables printing of all potential dof owner entities at initial assembly and at regular assembly, so that the index of all variables can be easily identified;
- the keyword `dof description` adds extra variable description;
- the keyword `equation description` adds extra equation description;
- the keyword `description` is a shortcut for both `dof description` and `equation description`;
- the keyword `element connection` prints information about element connectivity;
- the keyword `node connection` prints information about node connectivity;
- the keyword `connection` is a shortcut for both `node connection` and `element connection`;
- the keyword `all` enables all dof statistics printing;
- the keyword `none` disables all dof statistics printing (the default).
- If the keyword `to file` is used, then the output is written to the `.dof` file. Otherwise it is written to the standard output.

Note that, apart from `none`, the other values are additive, i.e. the statement

```
print: dof stats;
print: dof description;
```

is equivalent to

```
print: dof stats, dof description;
```

while the statement

```
print: none;
```

disables all.

5.3.3 Make Restart File

Restart files can be used to save the current state of a simulation to a file. This file can then be used to restore the previous state. There are currently two types of restart files. The `classic` restart file format is a new input file for MBDyn, whereas the `binary` restart file is a collection of all internal states of all the elements and nodes in the model. With the `binary` restart file, it is possible to run multi-stage simulations using the keyword `load restart file`.

```
<card> ::= make restart file
    [ : { iterations , <iterations_between_restarts>
        | time, (real) <time_between_restarts>
        | times, (integer) <number_of_restarts>, (real) <restart_1>, ..., <restart_N>
        | format, { classic | binary } }] ;
```

The default (no arguments) is to make the `classic` restart file only at the end of the simulation.

Note: the `make restart file` statement is experimental and essentially abandoned.

5.3.4 Load restart file

Load the state of a previous simulation, generated by "`make restart file`: `format`, `binary`" and reuse it as a starting point the current simulation. The initial time and all the internal states of the current simulation will be overridden by the contents of the restart file. A typical use case of `load restart file` is a static analysis as the first stage, followed by a preloaded eigenanalysis or transient analysis as the second stage.

```
<card> ::= load restart file (string) " <restart_file_name> " ;
```

5.3.5 Select Timeout

```
<card> ::= select timeout , { forever | <timeout> } ;
```

exit after `timeout` minutes when waiting for connections on `stream drives` or `stream output elements`. By default, no timeout is used, so `select(2)` waits forever.

5.3.6 Default Output

```
<card> ::= default output : <output_list> ;

<output_list> ::= { all | none | <output_item> } [ , ... ]

<output_item> ::=
    { reference frames
    | drive callers
```

```
    ----
    | abstract nodes
    | electric nodes
    | hydraulic nodes
    | structural nodes [ | accelerations ]
    | thermal nodes
    ----
    | aerodynamic elements
    | aeromodals
    | air properties
    | beams
    | electric bulk elements
    | electric elements
    | external elements
    | forces
    | genels
    | gravity
    | hydraulic elements
    | induced velocity elements
    | joints
    | loadable
    | plates
    | solids
    | rigid bodies
    | thermal elements }
```

Here the default output flag for a type of node or element can be set. It can be overridden for each entity either when it is created or later, for entity aggregates, in each entity module, by means of the `output` directive for `nodes` (see Section 6.7.1) and `elements` (see Section 8.20.4). Special values are:

- `all` enables output of all entities;
- `none` disables output of all entities;
- `reference frames` by default, reference frames are not output; when enabled, a special file `.rfm` is generated, which contains all the reference frames defined, using the syntax of the `.mov` file.
- `accelerations` enables output of linear and angular accelerations for dynamic structural nodes, which are disabled by default. Accelerations output can be enabled on a node by node basis; see `structural node` (see Section 6.5) for details.

Since values are additive, except for `none`, to select only specific entities use `none` first, followed by a list of the entities whose output should be activated.

Example.

```
begin: control data;
    # ...
    # disable all except structural nodes
    default output: none, structural nodes;
    # ...
end: control data;
```

Table 5.2: Predefined units systems

	Length	Mass	Time	Current	Temperature
MKS	m	kg	s	A	K
CGS	cm	kg	s	A	K
MMTMS	mm	ton	ms	A	K
MMKGMS	mm	kg	ms	A	K

5.3.7 Output units

This statement is intended to document in the .log file and in the NetCDF results what system of units the input file is written in, and thus the results.

The user should remember that MBDyn works with numbers: it is the responsibility of the user to build an input file using a consistent system of units. The specification of a different unit system has *no effect whatsoever* on the input file parsing and/or on the simulation, it only affects the information displayed in the output, and it intended merely as a reminder for the user.

```
<card> ::= output units : <units system> ;

<units system> ::=
    { MKS
        | CGS
        | MMTMS
        | MMKGMS
        | <custom units system> }

<custom units system> ::= Custom,
    Length,  (string) <unit_for_length>,
    Mass,  (string) <unit_for_mass>,
    Time,  (string) <unit_for_time>,
    Current,  (string) <unit_for_electric_current>,
    Temperature, (string) <unit_for_temperature> ;
```

With the `<custom units system>` the user needs to define all the principal units of measure. The pre-defines units systems are reported in Table 5.2.

5.3.8 Output Frequency

This statement is intended for producing partial output.

```
<card> ::= output frequency : <steps> ;
```

Despite the perhaps misleading name, this statement causes output to be produced every `steps` time steps, starting from the initial time. A more general functionality is now provided by the `output meter` statement (Section 5.3.9).

5.3.9 Output Meter

A drive that causes output to be generated when different from zero, while no output is created when equal to zero. It is useful to reduce the size of the output file during analysis phases that are not of interest.

```
<card> ::= output meter : (DriveCaller) <meter> ;
```

The functionality of the deprecated `output frequency` statement can be reproduced by using the `meter` drive caller as follows:

```
output meter: meter, 0., forever, steps, 10;
```

When integrating with variable time step, one may want to use the `closest next DriveCaller` (see Section 2.6.3); for example,

```
output meter: closest next, 2., forever, const, 0.01;
```

causes output to occur at times greater than or equal to multiples of 0.01 s, starting at 2 s.

5.3.10 Output Precision

Sets the desired output precision for those file types that honor it (currently, all the native output except the `.out` file). The default is 6 digits; since the output is in formatted plain text, the higher the precision, the larger the files and the slower the simulation.

```
<card> ::= output precision : <number_of_digits> ;
```

This is not an issue when using the NetCFD output, which returns double precision and is faster. See Section C for more info.

5.3.11 Output Results

This deprecated statement was intended for producing output in formats compatible with other software. See Appendix D for a description of the types of output that MBDyn can provide. Most of them are produced in form of post-processing, based on the default raw output.

Right now, the `output results` statement is only used to enable the experimental support for NetCDF output, which eventually will replace the current textual output:

```
<card> ::= output results : netcdf [ , <file_format> ]
    [ , [ no ] sync ] [ , [ no ] text ] ;
```

where

```
<file_format> ::= { classic | classic64 | nc4 | nc4classic }
```

allows one to choose the preferred file format, with `nc4` the default[1]. The optional `sync` keyword forces syncing of the output file after each time step; this allows to monitor the results of an ongoing simulation, at the expense of a slightly increased I/O time (`no sync` is also provided, in case the default changes). If the optional keyword `no text` is present, standard output in ASCII form is disabled (`text` is also provided, in case the default changes).

5.3.12 Default Orientation

This statement is used to select the default format for orientation output. For historical reasons, MBDyn always used the '123' form of Euler angles (also known as Tait-Bryan or Cardano angles). This statement allows to enable different formats:

[1] Currently, the default has been brought back to `classic`, since using `nc4` results in much slower analyses.

```
<card> ::= default orientation : <orientation_type> ;

<orientation_type> ::=
    { euler123
        | euler313
        | euler321
        | orientation vector
        | orientation matrix }
```

where

- `euler123` is the historical representation by means of three angles, in degrees, that represent three consecutive rotations about axes 1, 2 and 3 respectively, always applied to the axis as it results from the previous rotation (also known as Tait-Bryan or Cardano angles);
- `euler313` is similar to `euler123`, but the rotations, in degrees, are about axes 3, 1 and 3 in the given sequence. This is the usual form for Euler angles.
- `euler321` is similar to `euler123`, but the rotations, in degrees, are about axes 3, 2 and 1 in the given sequence.
- `orientation vector` is the vector whose direction indicates the axis of the rotation that produces the orientation, and whose modulus indicates the magnitude of that rotation, in radian;
- `orientation matrix` yields the orientation matrix itself. *Note: this selection implies that the number of columns required for the output in textual format changes from 3 to 9.*

The default remains `euler123`, in degrees.

Note: this change implies that by default the selected way will be used to represent orientations in input and output. This flag is not fully honored throughout the code, yet. Right now, only `structural nodes` and selected elements can output orientations as indicated by `default orientation`. However, there is no direct means to detect what format is used in the `.mov` file (while it is easy, for example, in the `.nc` file generated by NetCDF). As a consequence, it is the user's responsibility to keep track of what representation is being used and treat output accordingly.

5.3.13 Default Aerodynamic Output

This statement is used to select the default output of built-in aerodynamic elements.

```
<card> ::= default aerodynamic output :
    <custom_output_flag> [ , ... ] ;

<custom_output_flag> ::=
    { position
        | orientation [ , orientation description , <orientation_type> ]
        | velocity
        | angular velocity
        | configuration # ::= position, orientation, velocity, angular velocity
        | force
        | moment
        | forces        # ::= force, moment
        | all }         # equivalent to all the above
```

The orientation_type is defined in Section 5.3.12.

Flags add up to form the default aerodynamic element output request. Flags may not be repeated.

Output occurs for each integration point. The kinematics refers to location, orientation, and linear and angular velocity of the reference point. The forces are actual forces and moments contributing to the equilibrium of a specific node. By default, only force and moment are output. The custom output is only available in NetCDF format; see Section C.1.4.

5.3.14 Default Beam Output

This statement is used to select the default output of beam elements.

```
<card> ::= default beam output : <custom_output_flag> [ , ... ] ;

<custom_output_flag> ::=
    { position
        | orientation [ , orientation description , <orientation_type> ]
        | configuration        # ::= position, orientation
        | force
        | moment
        | forces               # ::= force, moment
        | linear strain
        | angular strain
        | strains              # ::= linear strain, angular strain
        | linear strain rate
        | angular strain rate
        | strain rates         # ::= linear strain rate, angular strain rate
        | all }                # equivalent to all the above
```

The orientation_type is defined in Section 5.3.12.

Flags add up to form the default beam output request. Flags may not be repeated. Output refers to the beam's "evaluation points". Strain rates are only available from viscoelastic beams; even if set, elastic beams will not output them.

By default, only forces are output, to preserve compatibility with the original output format. The custom output is only available in NetCDF format; see Section C.1.4.

5.3.15 Default Scale

```
<card> ::= default scale : <scale_list> ;

<scale_list> ::= <scale_pair> [ , <output_list> ]

<scale_pair> ::= { all | none | <scale_item> } , <scale_factor>

<scale_item> ::=
    { abstract nodes
        | electric nodes
        | hydraulic nodes
        | structural nodes
        | thermal nodes }
```

Define the residual scaling factor for all dof owners, or for specific types of dof owners. In principle, all dof owners should allow to define a scale factor, since they define a set of equations. In practice, only the above listed types support this option.

Residual scaling is only active when specifically requested by the related option for the tolerance keyword in the problem-specific section. See Section 4.1.1 for further details.

5.3.16 Finite Difference Jacobian Meter

```
<card> ::= { finite difference jacobian meter | jacobian check } :
                  (DriveCaller) <compute_finite_difference_time>
    [ , iterations, (DriveCaller) <compute_finite_difference_iteration> ]
    [ , { forward mode automatic differentiation | [ coefficient, (real) <delta>, ]
                                                   [ order, (integer) <k> , ] } ]
    [ , output
        [ , { none | all } ]
        [ , matrices, { yes | no }  ]
        [ , statistics, { yes | no } ]
        [ , statistics iteration, { yes | no } ]
    ]
```

When compute_finite_difference_time and compute_finite_difference_iteration are true, compute the problem Jacobian matrix by means of finite differences, and write on standard output both the analytic and finite difference Jacobian matrices. It is possible to tune the accuracy of the finite difference approximation by means of the finite difference perturbation parameter delta and the finite difference order k. Those options can be used to check if the implementation of the residual vector and the Jacobian matrix are consistent. However, in order to check if the implementation of the Jacobian matrix and the implementation of the matrix free Jacobian vector product are consistent, the keyword forward mode automatic differentiation should be used instead. By default, all output options are enabled and the sparse matrices are printed in triplet format. If it is desired to print only the maximum difference between the analytical Jacobian and the finite difference Jacobian matrix, then the keywords statistics and/or statistics iteration may be used. This option makes sense only for debugging, and may lead to really cumbersome screen output.

Examples

```
## Output the finite difference Jacobian and the analytical Jacobian matrix
## every 1e-3 seconds and always at the third iteration of each time step,
## using a 6th order accurate finite difference formula and a perturbation
## parameter of delta=1e-5.
finite difference jacobian meter: closest next, 0., forever, 1e-3,
                                  iterations, string, "Var == 3",
                                  coefficient, 1e-5,
                                  order, 6,
                                  output, none, matrices, yes;

## Output the maximum difference between the Jacobian free matrix vector product and the
## analytical Jacobian matrix over every iteration and every step at the end of the simulation.
jacobian check: one, one,
                forward mode automatic differentiation,
                output, none, statistics, yes;
```

5.3.17 Model

```
<card> ::= model : <model_type> ;

<model_type> ::= static
```

This statement allows to set the model type to `static`, which means that all dynamic structural nodes will be treated as static, and inertia forces ignored. Gravity and centripetal acceleration will only be considered as forcing terms. See the `structural` node (Section 6.5) for details.

5.3.18 Rigid Body Kinematics

```
<card> ::= rigid body kinematics : <rbk_data> ;

<rbk_data> ::= { <const_rbk> | <drive_rbk> }

<const_rbk> ::= const
        [ , position , (Vec3) <abs_position> ]
        [ , orientation , (OrientationMatrix) <abs_orientation> ]
        [ , velocity , (Vec3) <abs_velocity> ]
        [ , angular velocity , (Vec3) <abs_angular_velocity> ]
        [ , acceleration , (Vec3) <abs_acceleration> ]
        [ , angular acceleration , (Vec3) <abs_angular_acceleration> ]

<drive_rbk> ::= drive
        [ , position , (TplDriveCaller<Vec3>) <abs_position> ]
        [ , orientation , (TplDriveCaller<Vec3>) <abs_orientation_vector> ]
        [ , velocity , (TplDriveCaller<Vec3>) <abs_velocity> ]
        [ , angular velocity , (TplDriveCaller<Vec3>) <abs_angular_velocity> ]
        [ , acceleration , (TplDriveCaller<Vec3>) <abs_acceleration> ]
        [ , angular acceleration , (TplDriveCaller<Vec3>) <abs_angular_acceleration> ]
```

In principle, the kinematic parameters should be consistent. However, in most cases this is not strictly required, nor desirable. In fact, if the model is made only of rigid bodies, algebraic constraints and deformable restraints, the case of a system rotating at constant angular velocity does not require `abs_angular_velocity` to be the derivative of `abs_orientation`, since the latter never appears in the forces acting on the system. Similarly, the `abs_position` and `abs_velocity` do not appear as well, as soon as the latter is constant.

However, if other forces that depend on the absolute motion (position, orientation, and velocity) participate, this is no longer true. This is the case, for example, of aerodynamic forces, which depend on the velocity of the body with respect to the airstream, whose velocity is typically expressed in the global reference frame.

5.3.19 Loadable path

```
<card> ::= loadable path : [ { set | add } , ] " <path> " ;
```

This card allows to either set (optional keyword `set`) or augment (optional keyword `add`) the search path for run-time loadable modules using the path specified as the mandatory argument `path`.

Note: this command should be considered obsolete, but its replacement is not implemented yet. It impacts the loading of all run-time loadable modules, not only that of `user defined` or `loadable` elements (Section 8.19. See for example the `module load` statement in Section 2.4.6.

5.4 Model Counter Cards

The following counters can be defined:

5.4.1 Nodes

- abstract nodes
- electric nodes
- hydraulic nodes
- parameter nodes
- structural nodes
- thermal nodes

5.4.2 Drivers

- file drivers

5.4.3 Elements

- aerodynamic elements
- aeromodals
- air properties
- automatic structural elements
- beams
- bulk elements
- electric bulk elements
- electric elements
- external elements
- forces
- genels
- gravity
- hydraulic elements
- induced velocity elements
- joints
- joint regularizations
- loadable elements

- output elements
- solids
- surface loads
- rigid bodies

Chapter 6

Nodes

The nodes section is enclosed in the cards:

```
begin : nodes ;
    # ...
end : nodes ;
```

Every node card has the format:

```
<card> ::= <node_type> : <node_label>
    <additional_args>
    [ , scale , { default | <scale> } ]
    [ , output , { yes | no | (bool)<output_flag> } ]
    [ , <extra_arglist> ] ;
```

where node_type is one of the following:

- abstract
- electric
- hydraulic
- parameter
- structural
- thermal

The data manager reads the node type and the label and checks for duplication. If the node is not defined yet, the appropriate read function is called, which parses the rest of the card and constructs the node.

The optional scale keyword is only supported by

- abstract
- electric
- hydraulic
- structural
- thermal

nodes. See Section 5.3.15 for further details.

6.1 Abstract Node

```
<additional_args> ::= , { algebraic | differential }
    [ , value , <initial_value>
    [ , derivative , <derivative_initial_value> ] ]
```

Note: abstract nodes are ancestors of all scalar node types. Many `genel` *and* `electric` *elements can be connected to* `abstract` *nodes as well, since they directly use the ancestor class.*

Output. The value of abstract nodes is output with file extension `.abs`; for each time step the output of the required nodes is written. The format of each row is

- the label of the node
- the value of the node
- the value of the node derivative, when `differential` (the default)

Private Data. The following data are available:

1. `"x"` state
2. `"xP"` state time derivative, when `differential` (the default)

6.2 Electric Node

```
<additional_args> ::= [ , value , <initial_value>
    [ , derivative , <derivative_initial_value> ] ]
```

Note: the keywords `value` *and* `derivative` *have been introduced recently;* `value` *is not mandatory, resulting in a warning, while* `derivative` *is required. The same applies to the* `abstract node` *and to the* `hydraulic node`*; the latter is an algebraic node, so only* `value` *is allowed.*

Private Data. The following data are available:

1. `"x"` voltage
2. `"xP"` voltage time derivative

6.3 Hydraulic Node

The hydraulic node represents the pressure at a given location of a hydraulic circuit. It is derived from the generic scalar algebraic node; as a consequence, it can be connected to all elements that operate on generic scalar nodes.

The syntax is

```
<additional_args> ::= [ , value , <initial_value> ]
```

Private Data. The following data are available:

1. `"x"` pressure

6.4 Parameter Node

NOTE: `parameter` nodes are essentially obsolete; in most cases their purpose can be better fulfilled by `element` and `node` drive callers.

```
<additional_args> ::= { [ , value , <initial_value> ]
    | , element
    | , sample and hold , (NodeDof) <signal> , <sample_time>
    | , beam strain gage , <y> , <z> }
```

The parameter node is derived from the class scalar algebraic node, but it is used in a rather peculiar way: it doesn't own any degree of freedom, so it does not participate in the solution; it is rather used as a sort of placeholder for those elements that require to be connected to a scalar node that is not otherwise significant to the analysis. Thanks to the availability of the `parameter` node, these elements do not need be reformulated with a grounded node, while the parameter node value can be changed during the solution by several means, listed in the following.

Element. When the argument list starts with the keyword `element`, the parameter node expects to be bound to an element, and to access bulk element data (see the `bind` statement, Section 8.20.1).

Sample and Hold. When the argument list starts with the keyword `sample and hold`, followed by a `NodeDof` specification and a sample time, the parameter node contains the value of the input signal, namely the value of the node, for the duration of the `sample_time`. This may be useful to preserve the value of some signal across time steps.

Beam Strain Gage. When the argument list starts with the keyword `beam strain gage`, followed by the coordinates of a point on the section of a beam, the `parameter` node expects to be bound to a `beam` element, and to access the measure of the axial strain at point `x`, `y` in the section plane as a combination of section strains and curvatures:

$$\varepsilon = \nu_x + \mathtt{z} \cdot \kappa_y - \mathtt{y} \cdot \kappa_z, \tag{6.1}$$

where

- ν_x is the axial strain of the beam;
- κ_y is the bending curvature of the beam about the y axis;
- κ_z is the bending curvature of the beam about the z axis.

The span-wise location of the point where the strain is evaluated is set in the `bind` statement (see Section 8.20.1).

Note: measuring strains by means of derivatives of interpolated positions and orientations may lead to inaccurate results; force summation should be used instead.

6.5 Structural Node

Structural nodes can have 6 degrees of freedom (position and orientation), and thus describe the kinematics of rigid-body motion in space, or 3 degrees of freedom (position) and thus describe the kinematics of point mass motion in space.

The former has been originally implemented in MBDyn; the latter has been added in MBDyn 1.5.0, mainly to support the implementation of membrane elements.

The 6 dof structural node can be `static`, `dynamic`, `modal` or `dummy`.

The 3 dof structural node can be `static` or `dynamic`.

Elements that only require displacement can be connected to either type of nodes; when connected to a 6 degree of freedom node, they only make use of position/velocity and only contribute to force equilibrium equations.

Elements that require both displacement and orientation can only be connected to 6 degree of freedom nodes.

6.5.1 Static Node

The `static` keyword means no inertia is related to that node, so it must be appropriately constrained or attached to elastic elements. Static nodes are useful when there is no need to apply inertia to them, thus saving 6 degrees of freedom.

6.5.2 Dynamic Node

The `dynamic` keyword means inertia can be attached to the node, so it provides linear and angular momenta degrees of freedom, and automatically generates the so-called `automatic structural` elements.

6.5.3 Modal Node

The `modal` node is basically a regular `dynamic` node that must be used to describe the rigid reference motion of a `modal` joint. See Section 8.12.31 for further details.

6.5.4 Syntax

The syntax of the 6 degrees of freedom rigid-body motion structural node is

```
<additional_args> ::= , { static | dynamic | modal } ,
    [ position , ]          (Vec3)              <absolute_position> ,
    [ orientation , ]       (OrientationMatrix) <absolute_orientation_matrix> ,
        [ orientation description , <orientation_type> , ]
    [ velocity , ]          (Vec3)              <absolute_velocity> ,
    [ angular velocity , ] (Vec3)               <absolute_angular_velocity>
    [ , assembly
      , (real) <position_initial_stiffness>
      , (real) <velocity_initial_stiffness>
      , { yes | no | (bool) <omega_rotates?> } ]
```

The `orientation_type` is defined in Section 5.3.12.

The syntax of the 3 degrees of freedom point mass motion structural node is

```
<additional_args> ::= , { static displacement | dynamic displacement } ,
    [ position , ] (Vec3) <absolute_position> ,
    [ velocity , ] (Vec3) <absolute_velocity> ,
    [ , assembly
      , (real) <position_initial_stiffness>
      , (real) <velocity_initial_stiffness> ]
```

When a node's initial configuration is coincident with that of a single reference, instead of the `position`, `orientation`, `velocity`, and `angular velocity` one can respectively use

```
<additional_args> ::= , { static | dynamic | modal } ,
    at reference , <reference_label>
        [ , orientation description , <orientation_type> ]
    [ , assembly
      , (real) <position_initial_stiffness>
      , (real) <velocity_initial_stiffness>
      , { yes | no | (bool) <omega_rotates?> } ]
```

or

```
<additional_args> ::= , { static displacement | dynamic displacement } ,
    at reference , <reference_label>
    [ , assembly
      , (real) <position_initial_stiffness>
      , (real) <velocity_initial_stiffness> ]
```

The orientation_type is defined in Section 5.3.12.

The stiffness parameters and the omega_rotates? flag override the default values optionally set by the initial stiffness and omega rotates keywords in the control data block. They are optional, but they must be supplied all together if at least one is to be input.

The omega_rotates? parameter determines whether the initial angular velocity should follow or not the node as it is rotated by the initial assembly procedure. It can take values yes or no; a numerical value of 0 (no) or 1 (yes) is supported for backward compatibility, but its use is deprecated.

The dynamic and the modal node types allow the optional output keyword accelerations after the standard node output parameters:

```
<extra_arglist> ::= accelerations , { yes | no | (bool) <value> }
```

to enable/disable the output of the linear and angular accelerations of the node. Since they are computed as a postprocessing, they are not required by the regular analysis, so they should be enabled only when strictly required.

Also note that accelerations may be inaccurate, since they are reconstructed from the momentum and the momenta moment derivatives through the inertia properties associated with dynamic nodes.

The keyword accelerations can be used for dummy nodes; however, the dummy node will compute and output the linear and angular accelerations only if the node it is connected to can provide them.

Accelerations output can be controlled by means of the default output statement (see Section 5.3.6).

If the static model type is used in the control data block, all dynamic structural nodes are actually treated as static. This is a shortcut to ease running static analyses without the need to modify each node of a dynamic model.

6.5.5 Dummy Node

The dummy structural node has been added to ease the visualization of the kinematics of arbitrary points of the system during the simulation. It does not provide any degrees of freedom, and it must be attached to another node.

Elements cannot be directly connected to dummy nodes. They must be connected to the underlying node the dummy node is attached to.

The syntax for dummy nodes is:

```
<additional_args> ::= , dummy , <base_node_label> , <type> , <dummy_node_data>
```

Dummy nodes take the label base_node_label of the node they are attached to, followed by type, the type of dummy node, possibly followed by specific data, dummy_node_data.

The following dummy node types are available:

- offset:

  ```
  <type> ::= offset

  <dummy_node_data> ::=
      (Vec3)              <relative_offset> ,
      (OrientationMatrix) <relative_orientation_matrix>
          [ , orientation description , <orientation_type> ]
  ```

  It outputs the configuration of a point offset from the base node.

- relative frame:

  ```
  <type> ::= relative frame

  <dummy_node_data> ::= <reference_node_label>
      [ , position , (Vec3) <reference_offset> ]
      [ , orientation , (OrientationMatrix) <reference_orientation_matrix> ]
          [ orientation description , <orientation_type> , ]
      [ , pivot node , <pivot_node_label>
          [ , position , (Vec3) <pivot_offset> ]
          [ , orientation , (OrientationMatrix) <pivot_orientation_matrix> ] ]
  ```

  It outputs the configuration of the base node in the frame defined by the node of label `reference_node_label`, optionally offset by `reference_offset` and with optional relative orientation `reference_orientation_matrix`.

  If a `pivot_node_label` is given, the relative frame motion is transformed as if it were expressed in the reference frame of the pivot node, optionally offset by `pivot_offset` and with optional relative orientation `pivot_orientation_matrix`.

The `orientation_type` is defined in Section 5.3.12.

Example.

```
set: real Omega = 1.;
structural: 1, static, null, eye, null, 0.,0.,Omega;
structural: 1000, dummy, 1, offset, 1.,0.,0., eye;
structural: 1001, dummy, 1, relative frame, 1000;
structural: 2000, dynamic,
    0.,0.,1.,
    1, 1.,0.,0., 2, 0.,0.,1.,
    null,
    null,
    accelerations, yes;
```

Output. Structural nodes generate two kinds of output files:

- the `.mov` file;
- the `.ine` file.

The output of displacement-only 3 degree of freedom nodes is identical to that of 6 degree of freedom nodes, with the meaningless fields filled with zeros.

The `.mov` Output File.

The first refers to the kinematics of the node; its extension is `.mov`, and for each time step it contains one row for each node whose output is required. The rows contain:

1	the label of the node
2–4	the three components of the position of the node
5–7	the three Euler angles that define the orientation of the node
8–10	the three components of the velocity of the node
11–13	the three components of the angular velocity of the node
14–16	the three components of the linear acceleration of the `dynamic` and `modal` nodes (optional)
17–19	the three components of the angular acceleration of the `dynamic` and `modal` nodes (optional)

All the quantities are expressed in the global frame, except for the `relative frame` type of `dummy` node, whose quantities are, by definition, in the relative frame.

Other two variants of this output are available. The output of the orientation can be modified by requesting the three Euler angles, in 3 formats, the three components of the Euler vector, or the nine components of the orientation matrix.

When the Euler angles are requested, columns 5 to 7 contain them, in degrees, according to the requested output format: by default, the '123' sequence is used (which results in the so-called Tait-Bryan or Cardano angles); the '313' and '321' sequences are supported as well.

When the Euler vector is requested, columns 5 to 7 contain its components. Recall that the Euler vector's direction represents the rotation axis, whereas its norm represents the magnitude, in radian.

Beware that it is not possible to discriminate between the flavours of Euler angles and the Euler vector without knowing what output type was requested.

When the orientation matrix is requested, columns 5 to 13 contains the elements of the matrix, written row-wise, that is: r_{11}, r_{12}, r_{13}, r_{21}, ..., r_{33}. Note that in this case the total column count changes and, if accelerations are not requested, it corresponds to that of a structural node with optional accelerations, so it might be hard to discriminate in this case as well.

This ambiguity is resolved when results are output in NetCDF format. See Section C.1.3 for details.

Note: actually, the angles denoted as "Euler angles" are the three angles that describe a rotation made of a sequence of three steps: first, a rotation about global axis 1, followed by a rotation about axis 2 of the frame resulting from the previous rotation, concluded by a rotation about axis 3 of the frame resulting from the two previous rotations. To consistently transform this set of parameters into some other representation, see the tools `eu2rot(1)`, `rot2eu(1)`, `rot2eup(1)`, `rot2phi(1)`. The functions that compute the relationship between an orientation matrix and the set of three angles and vice versa are `MatR2EulerAngles()` and `EulerAngles2MatR()`, in `matvec3.h`.

The `.ine` Output File.

The second output file refers only to dynamic nodes, and contains their inertia; its extension is `.ine`. For each time step, it contains information about the inertia of all the nodes whose output is required. Notice that more than one inertia body can be attached to one node; the information in this file refers to the sum of all the inertia related to the node.

The rows contain:

1	the label of the node
2–4	item the three components of the momentum in the absolute reference frame
5–7	item the three components of the momenta moment in the absolute reference frame, with respect to the coordinates of the node, thus to a moving frame
8–10	the three components of the derivative of the momentum
11–13	the three components of the derivative of the momentum moment

Private Data. The following data are available:

all structural nodes:

1. `"X[1]"` position in global direction 1
2. `"X[2]"` position in global direction 2
3. `"X[3]"` position in global direction 3
4. `"x[1]"` position in direction 1, in the reference frame of the node
5. `"x[2]"` position in direction 2, in the reference frame of the node
6. `"x[3]"` position in direction 3, in the reference frame of the node
7. `"Phi[1]"` orientation vector in global direction 1
8. `"Phi[2]"` orientation vector in global direction 2
9. `"Phi[3]"` orientation vector in global direction 3
10. `"XP[1]"` velocity in global direction 1
11. `"XP[2]"` velocity in global direction 2
12. `"XP[3]"` velocity in global direction 3
13. `"xP[1]"` velocity in direction 1, in the reference frame of the node
14. `"xP[2]"` velocity in direction 2, in the reference frame of the node
15. `"xP[3]"` velocity in direction 3, in the reference frame of the node
16. `"Omega[1]"` angular velocity in global direction 1
17. `"Omega[2]"` angular velocity in global direction 2
18. `"Omega[3]"` angular velocity in global direction 3
19. `"omega[1]"` angular velocity in direction 1, in the reference frame of the node
20. `"omega[2]"` angular velocity in direction 2, in the reference frame of the node
21. `"omega[3]"` angular velocity in direction 3, in the reference frame of the node
22. `"E[1]"` Cardan angle 1 (about global direction 1)
23. `"E[2]"` Cardan angle 2 (about local direction 2)
24. `"E[3]"` Cardan angle 3 (about local direction 3)
25. `"E313[1]"` Cardan angle 1 (about global direction 3)
26. `"E313[2]"` Cardan angle 2 (about local direction 1)
27. `"E313[3]"` Cardan angle 3 (about local direction 3)
28. `"E321[1]"` Cardan angle 1 (about global direction 3)

29. "E321[2]" Cardan angle 2 (about local direction 2)
30. "E321[3]" Cardan angle 3 (about local direction 1)
31. "PE[0]" Euler parameter 0
32. "PE[1]" Euler parameter 1
33. "PE[2]" Euler parameter 2
34. "PE[3]" Euler parameter 3

dynamic nodes only:

35. "XPP[1]" acceleration in global direction 1
36. "XPP[2]" acceleration in global direction 2
37. "XPP[3]" acceleration in global direction 3
38. "xPP[1]" acceleration in direction 1, in the reference frame of the node
39. "xPP[2]" acceleration in direction 2, in the reference frame of the node
40. "xPP[3]" acceleration in direction 3, in the reference frame of the node
41. "OmegaP[1]" angular acceleration in global direction 1
42. "OmegaP[2]" angular acceleration in global direction 2
43. "OmegaP[3]" angular acceleration in global direction 3
44. "omegaP[1]" angular acceleration in direction 1, in the reference frame of the node
45. "omegaP[2]" angular acceleration in direction 2, in the reference frame of the node
46. "omegaP[3]" angular acceleration in direction 3, in the reference frame of the node
47. "phi[1]" orientation vector in direction 1, in the reference frame of the node
48. "phi[2]" orientation vector in direction 2, in the reference frame of the node
49. "phi[3]" orientation vector in direction 3, in the reference frame of the node

Note: Euler parameters actually do not take into account the whole orientation of a node, since they are post-processed from the orientation matrix. As a consequence, they only parametrize the minimum norm orientation that yields the current orientation matrix of the node. The same applies to the orientation vector φ.

Note: if accelerations are requested using the `string` form, their computation is enabled even if it was not explicitly enabled when the node was instantiated. However, if the `index` form is used, their computation must have already been explicitly enabled.

Note: dummy nodes based on dynamic nodes inherit the capability to provide access to linear and angular accelerations.

6.6 Thermal Node

```
<additional_args> ::= [ , value , <initial_value>
    [ , derivative , <derivative_initial_value> ] ]
```

Note: the keywords `value` *and* `derivative` *have been introduced recently;* `value` *is not mandatory, resulting in a warning, while* `derivative` *is required. The same applies to the* `abstract node` *and to the* `hydraulic node`*; the latter is an algebraic node, so only* `value` *is allowed.*

Private Data. The following data are available:

1. `"x"` temperature
2. `"xP"` temperature time derivative

6.7 Miscellaneous

6.7.1 Output

There is an extra card, that is used to modify the output behavior of nodes:

```
<card> ::= output : <node_type> , <node_list> ;

<node_list> ::= { <node_label>  [ , ... ]
    | range , <node_start_label> , <node_end_label> }
```

`node_type` is a valid node type that can be read in the `nodes` block. In case the keyword `range` is used, all nodes of that type with label between `node_start_label` and `node_end_label` are set.
Note: if a node should never (`no`*) or always (*`yes`*) be output, its output flag should be set directly on the node card. The global behavior of all the nodes of a type can be set from the* `control data` *block by adding the node type to the item list in the* `default output` *card. Then, the specific output flag of sets of nodes can be altered by means of the* `output` *card in the* `nodes` *block. This allows high flexibility in the selection of the desired output. The same remarks apply to the output of the elements.*

If `node_type` is `structural`, the optional keyword `accelerations` can be used right after the `node_type` to enable the output of the accelerations.

Chapter 7

Drivers

The drivers section is enclosed by the cards:

```
begin : drivers ;
    # ...
end : drivers ;
```

Every driver card has the format:

```
<card> ::= <driver_type> : <arglist> ;
```

At present the only type driver_type of drivers supported is file.

7.1 File Drivers

The file drivers are defined by the statement

```
file : <file_arglist> ;
```

A comprehensive family of file drivers is available; their syntax is:

```
<file_arglist> ::= <label> , <normal_arglist>
```

The following file drivers are supported:

7.1.1 Fixed Step

```
<normal_arglist> ::= fixed step ,
    { count | <steps_number> } ,
    <columns_number> ,
    initial time , { from file | <initial_time> } ,
    time step , { from file | <time_step> } ,
    [ interpolation , { linear | const } , ]
    [ { pad zeroes , { yes | no }
        | bailout , { none | upper | lower | any } } , ]
    " <file_name> "
```

If the keyword `count` is used instead of the `steps_number`, the number of time records provided by the file is obtained by counting the valid records.

If the keyword `from file` is used instead of either `initial_time` or `time_step`, the value is read from the file. In this case, comment lines of the form

```
# initial time: <initial_time>
# time step: <time_step>
```

are expected; the corresponding values are parsed and checked. A valid value provided in the input file always overrides any corresponding value provided in the corresponding comment line.

The value at an arbitrary time is interpolated from the available data. If the requested value is out of time bounds, zero is returned, unless `pad zeroes` is set to `no`, which means that the first or the last set of values is respectively used. As an alternative, if `bailout` is set to `upper`, `lower` or `any`, the simulation is stopped as soon as the time goes out of the respective "time" bounds.

If `interpolation` is `linear` (the default), a linear interpolation is used. Otherwise, if `interpolation` is `const`, the value at the beginning of the time step that includes the current time is used.

The file format is

```
# an arbitrary number of comment lines starting with '#'
#
# comment lines may provide special parameters
# like "initial time", "time step" as indicated above; for example
#
# initial time: 0
# time step: 0.01
#
# channel #1 channel #2 ... channel #n
  1.         2.         ... 100.
...
```

Example.

```
    begin: drivers;
        file: 100,
            fixed step,
            100,       # n. steps
            1,         # n. channels
            0,         # initial_time
            0.01,      # time_step
            "input.dat";

        file: 200,
            fixed step,
            count,     # n. steps
            1,         # n. channels
            from file, # initial_time
            from file, # time_step
            "input.dat";
    end: drivers;
```

7.1.2 Variable Step

```
<normal_arglist> ::= variable step ,
    <channels_number> ,
    [ interpolation , { linear | const } , ]
    [ { pad zeroes , { yes | no }
        | bailout , { none | upper | lower | any } , } , ]
    " <file_name> "
```

The same considerations of the `fixed step` type apply.

The file format is

```
# an arbitrary number of lines starting with '#'
#
# time channel#1 channel#2 ... channel#n
  0.   1.        2.        ... 100.
  0.01 1.2       1.8       ...  90.
...
```

The number of available records is computed as the number of non-comment lines (lines not starting with a hash mark '`#`'). The first column contains the time, while the remaining `channels_number` columns contain the values of the input channels at that time. Time values must grow monotonically.

7.1.3 Socket

```
<normal_arglist> ::= socket ,
    <columns_number> ,
    [ initial values , <value #1> , ... , ]
    { local , " <file_name> "
        | [ port , <port_number> , ] (AuthenticationMethod) <authentication> }
```

The driver binds to a socket listening on port `port_number` (defaults to 9011) or on the named pipe `file_name`; at the beginning of each time step, in case of connection, the driver expects some input data in text format, consisting in an authentication token (if required). The authentication token is usually in the form

```
user: <user_name> NEWLINE
password: <user_password> NEWLINE
```

where `NEWLINE` is a literal 'newline' (i.e. '`\n`'). White-spaces may be significant in `user_name`, and surely are in `user_password`.

The identification is followed by a label token, in the form

```
label: <label> NEWLINE
```

indicating the column being edited, followed by the desired changes; the connection is terminated by a single mark followed by a newline:

```
. NEWLINE
```

The permitted operations, at present, are:

```
value: <value> NEWLINE
```

sets the value the drive will assume from the current step on

```
inc: { yes | no } NEWLINE
```

tells whether to switch on or off the increment mode, resulting in subsequent value commands being "set" rather than "add"

```
imp: { yes | no } NEWLINE
```

tells whether to switch on or off the impulse mode; when on, subsequent value commands to be applied for one step only. At present, impulse mode supersedes any incremental mode, namely the value of the drive is reset to zero after one step. This behavior may change in the future.

7.1.4 RTAI Mailbox

This special drive is a variant of the Socket Stream (Section 7.1.5; under development yet) that reads the input data from a RTAI mailbox in non-blocking mode. It is intended as a means of communication between different processes running in real-time. The syntax is:

```
<normal_arglist> ::= RTAI input ,
    stream drive name , " <stream_name> " ,
    [ create , { yes | no } , ]
    [ host , " <host_name> " , ]
    [ { [ non ] blocking } , [ ... ] ]
    <columns_number> ;
```

where

- stream_name is the name of the mailbox (a unique string no more than 6 characters long);
- the create keyword determines whether the mailbox must be created or looked-up as already existing on the system;
- host_name is the name or the IP of the remote host where the mailbox resides; note that if this field is given, create must be set to no and the mailbox must be already defined on the remote host;
- the number of channels columns_number determines how many columns can be accessed via the file drive caller mechanism of Section 2.6.15, as for all the other file drivers.

This part of the program is rapidly evolving, so do please not expect too much documentation and backward compatibility.

Note: at present, these elements require that the simulation be run in real-time mode (see Section 4.1.5); future development will allow to emulate the use of these elements also when the simulation is not run in real-time, e.g. for modeling or model debugging purposes.

7.1.5 Stream

```
<normal_arglist> ::= stream ,
    name , " <stream_name> " ,
    create , { yes | no } ,
    [ { local , " <path> " ,
        | [ port , <port_number> , ] [ host , " <host_name> " , ] } ]
    [ socket type , { tcp | udp } , ]
    [ { [ no ] signal
        | [ non ] blocking } , [ ... ] , ]
```

```
    [ input every , <steps> , ]
    [ receive first , { yes | no } , ]
    [ timeout , <timeout> , ]
    [ echo , " <echo_file_name> "
        [ , precision , <precision> ]
        [ , shift , <shift> ] , ]
    { <columns_number>
        [ , initial values , <value #1> , ... ]
        [ , modifier , <content_modifier> ]
    | <user_defined> }

<user_defined> ::= <user_defined_type> [ , ... ]

<content_modifier> ::=
    copy
    | copy cast ,
        [ swap ,
            { <cast_type> [ , ... ]
            | detect
            | { yes | no | (bool) <swap> } } , ]
        { all , <cast_type> | <one_cast> [ , ... ] }
        [ , size , (int)<buffer_size> ]

<one_cast> ::=
    <cast_type>
    | skip , (int)<num_bytes>

<cast_type> ::=
    int8_t
    | uint8_t
    | int16_t
    | uint16_t
    | int32_t
    | uint32_t
    | float     # assuming float is 4 bytes
    | double    # assuming double is 8 bytes
```

The `stream` drive allows MBDyn to receive streamed inputs from remote processes both during regular simulations and during real-time simulations under RTAI. If the simulation is run in real-time, it uses RTAI mailboxes, otherwise regular UNIX sockets are used, either in the `local` or in the `internet` namespace.

This drive type is intended to allow the development of real-time models by running regular simulations for the purpose of debugging the model and the control process without the overhead and the potential problems of running in real-time. Then the same model can be run in real-time without changing the details of the communication with the controller process.

Non real-time simulation During non real-time simulations, streams operate in blocking mode. The meaning of the parameters is:

- `stream_name` indicates the name the stream will be known as; it is mostly useless, and must be no more than 6 characters long, since it is only allowed for compatibility with RTAI's mailboxes;
- the instruction `create` determines whether the socket will be created or looked for by MBDyn; if `create` is set to `no`, MBDyn will retry for 60 seconds and then give up;
- the keyword `local` indicates that a socket in the local namespace will be used; if `create` is set to `yes`, the socket is created, otherwise it must exist.
- either of the keywords `port` or `host` indicate that a socket in the internet namespace will be used; if `create` is set to `yes`, `host_name` indicates the host that is allowed to connect to the socket; it defaults to any host (0.0.0.0); if `create` is set to `no`, `host_name` indicates what host to connect to; the default is localhost (127.0.0.1); the default port is 9011;
- `socket type` defaults to `tcp`;
- the keyword `no signal` disables raising a `SIGPIPE` in case the stream is read after it was closed by the peer;
- the keyword `input every` allows to read new driver values every `steps` time steps;
- the keyword `receive first` allows to enable/disable receiving data at the initial time (defaults to `yes`, thus data are expected at the initial time, during derivatives);
- the keyword `timeout` allows to set a timeout for each read operation; the timeout is expressed in seconds as a floating point number, with a theoretical micro-second resolution (the actual resolution is left to the OS);
- the keyword `echo` allows to echo the streamed data into file `echo_file_name`, using the optional `precision` and a `shift` for the time, to make sure step changes are captured correctly (a small negative value is recommended); the resulting file is suitable to re-execute the simulation using a `variable step` file driver, using a `const` value for the `interpolation` option;
- the keyword `initial values`, followed by `columns_number` real values, allows to set the initial values of each channel; this is useful for example if the initial value of a channel is needed during initial assembly, before the first set of values is read from the peer.
- the keyword `modifier` installs a data modification layer.
- The `copy` case leaves data unaltered. The `copy cast` case performs data casting for each channel according to the subsequent rules.
  - the optional keyword `swap` instructs MBDyn to swap bytes in order to make transmitted values platform-independent with respect to endianness, under the assumption that endianness is the same for all data types. The keyword `swap` may be followed by a list of the types that must be swapped, included in those defined in `cast_type`. Alternatively, the keyword `detect` causes the endianness of the platform to be detected and data converted in big endian form if needed. Otherwise, conversion is either forced for all types in case of `yes` or non-zero value of `swap`, or ignored in case of `no` or zero value of `swap`.
  - if the keyword `all` is present, a single `one_cast` cast rule is applied to all channels. Otherwise, a `one_cast` rule is read for each channel. Only the types listed above are recognized. The special value `skip` actually causes `num_bytes` to be skipped before the subsequent `one_cast` rule is applied.

– the optional keyword `size` is used to provide the entire size of the buffer, in case only a portion of it is going to be used.

If no socket type is specified, i.e. none of the `local`, `port` and `host` keywords are given, a socket is opened by default in the internet namespace with the default IP and port; the `create` keyword is mandatory.

Real-time simulation During real-time simulations, streams wrap non-blocking RTAI mailboxes. The meaning of the parameters is:

- the parameter `stream_name` indicates the name the stream will be known as in RTAI's resource namespace; it must no more than 6 characters long, and actually represents the mailbox name;
- the instruction `create` determines whether the mailbox will be created or looked for by MBDyn;
- the keyword `local` is ignored;
- the keyword `host` indicates that a mailbox on a remote host will be used; it is useless when `create` is set to `yes`, because RTAI does not provide the possibility to create remote resources; if none is given, a local mailbox is assumed;
- the keyword `port` is ignored.

The parameter `columns_number` determines how many channels will be used. A channel is a double typed number; a `stream drive` can read an arbitrary number of simultaneous channels.

Chapter 8

Elements

The elements section is enclosed in the cards:

```
begin : elements ;
    # ...
end : elements ;
```

Every element card has the following format:

```
<card> ::= <elem_type> :
    <arglist>
    [ , output , { yes | no | (bool)<output_flag> } ]
    [ , <extra_arglist> ] ;
```

where elem_type is one of the following:

- structural elements:
  - automatic structural
  - beam
  - body
  - couple
  - gravity
  - joint
  - joint regularization
  - plate
  - solid
  - surface loads
- aerodynamic elements:
  - aerodynamic beam2
  - aerodynamic beam3
  - aerodynamic body
  - aeromodal

  - aircraft instruments
  - air properties
  - induced velocity
- electric elements:
  - electric
- hydraulic elements:
  - hydraulic
- thermal elements:
  - thermal
- output elements:
  - RTAI output
  - stream output
  - stream motion output
- generic elements:
  - bind
  - bulk
  - force
  - genel
  - loadable
  - user defined
- miscellaneous element cards

In case of elements that can be instantiated only once, like the gravity or the air properties elements, the arglist does not contain any label; otherwise, a label is expected first, to allow for checks on duplicated elements, namely:

```
<arglist> ::= <label> , <normal_arglist>
```

The data manager reads the element type and the label and checks for duplication. If the element is not defined yet, the proper read function is called, which parses the rest of the card and constructs the element. The elements are read as follows.

8.1 Aerodynamic Elements

8.1.1 Aerodynamic Body/Aerodynamic Beam2/3

These elements share the description of the aerodynamics. The former assumes the aerodynamic surface to be rigid, and takes its configuration from a single node, while the latter respectively relies on a two or three-node beam and use the same interpolation functions of the beam to compute the configuration at an arbitrary point.

The input format is:

```
<elem_type> ::= { aerodynamic body | aerodynamic beam2 | aerodynamic beam3 }

<normal_arglist> ::= <connectivity> ,
    (Shape<1D>)          <surface_chord> ,
    (Shape<1D>)          <surface_aerodynamic_center> ,
    (Shape<1D>)          <surface_b_c_point> ,
    (Shape<1D>)          <surface_twist> ,
    [ tip loss , (Shape<1D>) <tip_loss> , ]
    <integration_points>
    [ , control , (DriveCaller) <control_drive> ]
    [ , <airfoil_data> ]
    [ , unsteady , { bielawa } ]
    [ , jacobian , { yes | no | <bool> } ]
    [ , <custom_output> ]

<extra_arglist> ::= { std | Gauss | node }
```

where

```
<connectivity> ::= { <body_conn> | <beam2_conn> | <beam3_conn> }

# <elem_type> ::= aerodynamic body
<body_conn> ::= <node_label>
    [ , [ user defined ] induced velocity , <induced_velocity_label>
        [ , passive ] , ]
    (Vec3)              <relative_surface_offset> ,
    (OrientationMatrix) <relative_surface_orientation> ,
    (real)              <surface_span>

# <elem_type> ::= aerodynamic beam2
<beam2_conn> ::= <beam2_label>
    [ , [ user defined ] induced velocity , <induced_velocity_label>
        [ , passive ] , ]
    (Vec3)              <relative_surface_offset_1> ,
    (OrientationMatrix) <relative_surface_orientation_1> ,
    (Vec3)              <relative_surface_offset_2> ,
    (OrientationMatrix) <relative_surface_orientation_2> ,

# <elem_type> ::= aerodynamic beam3
<beam3_conn> ::= <beam3_label>
    [ , [ user defined ] induced velocity , <induced_velocity_label>
        [ , passive ] , ]
    (Vec3)              <relative_surface_offset_1> ,
    (OrientationMatrix) <relative_surface_orientation_1> ,
    (Vec3)              <relative_surface_offset_2> ,
    (OrientationMatrix) <relative_surface_orientation_2> ,
    (Vec3)              <relative_surface_offset_3> ,
    (OrientationMatrix) <relative_surface_orientation_3> ,

<airfoil_data> ::=
    { naca 0012
```

```
            | rae 9671
            | [ theodorsen , ] c81 , <c81_data> }
```

and

```
    <c81_data> ::=
        { <c81_label>
            | multiple , <airfoil_number> ,
                <c81_label> , <end_point>
                [ , ... ]
            | interpolated , <airfoil_number> ,
                <c81_label> , <position>
                [ , ... ] }
```

The `custom_output` optional data consists in

```
    <custom_output> ::= custom output ,
        <custom_output_flag> [ , ... ]
```

The values of `custom_output_flag` are defined in Section 5.3.13.

Flags add up to form the custom output request. Flags may not be repeated. By default, only forces are output. The custom output is only available in NetCDF format; see Section C.1.4.

The field `induced_velocity` indicates the label of the `induced velocity` element that this element is linked to. This means that the element can get information about the induced velocity and should supply information about the forces it generates. If the keyword `user defined induced velocity` is used, then `induced_velocity` refers to a `user defined` element. Note: if the keyword `induced velocity` is used and no induced velocity element is found with the label `induced velocity` a `user defined` element with that label is looked up.

An arbitrary relative orientation and offset is allowed for all elements with respect to the nodes they are linked to. This means that the aerodynamic beam offsets refer to the position of the beam's nodes, and have nothing to do with offsets related to the structural beam element.

The `Shape<1D>` entities are used to compute the physical chord (`surface_chord`, dimensional; must be non-negative), aerodynamic center (`surface_aerodynamic_center`, dimensional, positive when forward of the reference line), velocity measurement point (`surface_b_c_point`, the point where the kinematic boundary conditions are evaluated, positive when backwards?) and twist (`surface_twist`, positive when nose up), in radians, as functions of the dimensionless abscissa along the span.

The optional `tip loss` keyword allows the user to define an additional shape, `tip_loss`, whose value is used to scale the value of the normal component of the aerodynamic force. By default, the scale factor is 1. The `tip_loss` shape should have a value comprised between 0 and 1.

In any case, the user must be aware of the fact that Gauss integration will be used over the span of the element. This consists in evaluating forces at specific spanwise stations. This assumes that the function to be integrated over the spanwise domain is regular. Sharp variations, like tip loss concentrated in the outermost 2% of a rotor blade span, might be entirely missed when using too little spanwise elements with too little integration points.

The value resulting from the `control_drive` is added to the local angle of attack to mimick a deflection of a control surface that extends for the whole span of the element. As such, it does not directly correspond to the deflection of the surface, but rather to

$$\texttt{control\_drive} = \frac{C_{L/\delta}}{C_{L/\alpha}}\delta$$

where δ is the actual deflection of the control surface and $C_{L/\alpha}$ and $C_{L/\delta}$ respectively are the lift stability and control derivatives of the airfoil, namely the partial derivatives of the airfoil's lift coefficient with

respect to the angle of attack, α, and the control surface deflection. All angles are intended in radians. *Caveat: no account is made of any change in aerodynamic moment resulting from the deflection of the control surface.*

The span of the `aerodynamic body` element is set by the user; the offset vector points to the center-span of the element. The span of the `aerodynamic beam2` and `aerodynamic beam3` elements is computed based on the metric of the beam, from the first to the last node.

The aerodynamic center and the velocity measurement points are measured relative to the centerline of the elements, that is the line in direction 3 of the local frame from the end of the offset vector. This line is assumed to be at the 25% of the airfoil chord when steady aerodynamic coefficients are used (`unsteady_flag` = 0). The direction 1 is assumed to be the "reference" line of the airfoil, from the trailing edge to the leading edge (points "forward"), while direction 2 is normal to the other two and goes from the lower to the upper side of the airfoil (points "up"). Figure 8.1 shows the arrangement of the airfoil geometry and properties.

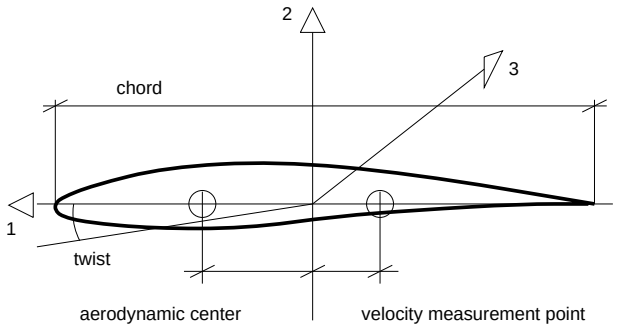


Figure 8.1: Airfoil geometry

The `airfoil_data` defaults to a built-in NACA 0012 semi-analytical model (FIXME: the unsteady correction is buggy; use the `c81` mode instead).

The `multiple` mode of the c81 data allows to specify more than one airfoil for an aerodynamic element; the transition between airfoils is sharp. The integer `airfoil_number` indicates how many airfoils are expected; the real `end_point` indicates where the influence zone for that airfoil ends, expressed in terms of a non-dimensional abscissa spanning $(-1, 1)$ along the reference line, roughly along axis 3 of the aerodynamic reference frame; `end_point` must not lie outside the element. So, for example, if airfoil NACA 0015 is used in the leftmost part of an element up to 1/4 span, NACA 0012 is used from 1/4 to 3/4 span, and NACA 0009 is used in the remaining rightmost 1/4, the syntax is:

```
set: integer naca0015 = 15;
set: integer naca0012 = 12;
set: integer naca0009 = 9;
c81 data: naca0015, "naca0015.c81";
c81 data: naca0012, "naca0012.c81";
c81 data: naca0009, "naca0009.c81";
# beginning of aerodynamic element definition...
    multiple, 3,
        naca0015, -0.5,    # from -1.0 to -0.5
        naca0012,  0.5,    # from -0.5 to  0.5
        naca0009,  1.0,    # from  0.5 to  1.0
# ...rest of aerodynamic element definition
```

The `interpolated` mode of the `c81` data allows to specify a smooth transition between different airfoils inside an element. The interpolation occurs at the integration points where the aerodynamic

data are required, and it is performed once for all at the beginning of the analysis. Since this operation is time consuming, and essentially unnecessary, the interpolated data can be generated once for all with the utility `util/c81merge` once the position of the integration point is known, and the `multiple` mode can be used to directly provide the interpolated data to the aerodynamic element.

The `theodorsen` aerodynamic data uses C81 data, and superimposes Wagner's approximation of the Theodorsen incompressible unsteady correction of 2D lift and moment coefficients. It is experimental.

The `extra_arglist` allows to define the style of the output. The `std` style (the default), the `Gauss` and the `node` styles are illustrated in the output section.

Output

Aerodynamic elements, both bodies and beams, write their output with file extension `.aer`; for each time step the required elements are output. In any case the label of the element is output first. Three different formats are available: `std` (the default), `Gauss` and `node`.

`std` (or Coefficients at Gauss points): the output consists in a set of 8 numbers for each block, that describe data at each Gauss integration point; multiple blocks for a single element are written on the same line. The format is:

- the angle of attack at that station, in degrees (namely, the angle between the component of the airfoil velocity, evaluated at the velocity measurement point, that in the airfoil plane and a reference line on the airfoil)
- the local yaw angle, in degrees (namely, the angle whose tangent is the ratio between the axial and the inplane components of the airfoil velocity)
- the local Mach number
- the local lift coefficient
- the local drag coefficient
- the local aerodynamic moment coefficient
- a private number
- another private number

When `aerodynamic beam2` and `aerodynamic beam3` elements are considered, the output is repeated for each portion of the beam; so, for example, a two-node beam is split in two portions, so the output contains 2 × `integration_points` data blocks, while a three-node beam is split in three portions, so the output contains 3 × `integration_points` data blocks.

`node`: the format is:

- the label of the node
- the three components of the force applied to the node
- the three components of the couple applied to the node

When `aerodynamic beam2` and `aerodynamic beam3` elements are considered, the output is repeated on the same line for each node the element is connected to.

`Gauss` (or Forces at Gauss points): the output consists in the forces and moments per unit length at each Gauss integration point; the format is:

- the direction of the wind velocity relative to the element frame

- the lift,
- the drag,
- and the aerodynamic moment per unit length

When `aerodynamic beam2` `aerodynamic beam3` elements are considered, the output is repeated on the same line for each portion of beam.

8.1.2 Aeromodal Element

Note: implemented by Giuseppe Quaranta; documented by Alessandro Scotti.
This element is used to model an aerodynamic modal element, i.e. an unsteady aerodynamic model that inherits the structural motion from a `modal joint` element. Its definition is very similar to that of the modal element, but it also includes some data representing unsteady aerodynamics in the time domain trough the residualization matrices. This element is defined as follows:

```
<elem_type> ::= aeromodal

<normal_arglist> ::=
    <reference_modal_joint> ,
    (OrientationMatrix) <orientation> ,
    <reference_chord> ,
    <number_of_aerodynamic_states> ,
    [ rigid , ]
    [ gust , <Vff> , ]
    " <modal_matrices_file> "
```

With this formulation, anytime an `aeromodal` element is defined, the user needs to declare the number of modal aerodynamic elements in use in the `control data` section. An `air properties` card definition is also required.

The label `reference_modal_joint` indicates the `modal joint` associated with the `aeromodal` element. The modal joint must be connected to a `modal` node; clamped modal joints are not supported.

The keyword `rigid` indicates that the generalized aerodynamic forces provided by the model include global forces and moments associated to the rigid body motion of the underlying modal element (FIXME: untested).

The keyword `gust` activates an optional gust model, which is totally undocumented; for further information, please contact the Author(s).

The `modal_matrices_file` file is an ASCII file that contains the matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{D}_0$, $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$, of a state space model according to the representation

$$\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{q}$$
$$\boldsymbol{f} = q\left(\boldsymbol{C}\boldsymbol{x} + \boldsymbol{D}_0\boldsymbol{q} + \frac{c}{2V_\infty}\boldsymbol{D}_1\dot{\boldsymbol{q}} + \left(\frac{c}{2V_\infty}\right)^2\boldsymbol{D}_2\ddot{\boldsymbol{q}}\right)$$

where $\boldsymbol{x}$ are the `number_of_aerodynamic_states` (`na`) aerodynamic state variables, $\boldsymbol{q}$ are the `ns` modal variables that describe the structural motion as defined in the related `modal joint`, c is the reference length, V_∞ is the free airstream velocity, $q = \rho V_\infty^2/2$ is the dynamic pressure, and $\boldsymbol{f}$ are the unsteady aerodynamic forces applied to the structural dynamics equations.

When the keyword `rigid` is present, the number of modal variables `ns` includes the modes provided by the `modal joint` plus 6 rigid-body modes corresponding to rigid-body displacement and rotation of the `modal node`.

The file is formatted as follows:

```
*** MATRIX A
(<na> x <na> coefficients)
*** MATRIX B
(<na> x <ns> coefficients)
*** MATRIX C
(<ns> x <na> coefficients)
*** MATRIX D0
(<ns> x <ns> coefficients)
*** MATRIX D1
(<ns> x <ns> coefficients)
*** MATRIX D2
(<ns> x <ns> coefficients)
```

Example.

```
aeromodal: WING, WING_JOINT,
    eye,
    131.25, 10, "ha145b.fea";
```

The aeromodal element is declared with the label WING. This element is attached to a modal joint named WING_JOINT. The orientation of the aerodynamic reference with respect to the nodal reference is here expressed by the identity matrix (eye). The aerodynamic element chord is 131.25 inches. This quantity must be consistent with the system chosen to define the whole model (SI, for example; in this case, British Units). The next field, 10, indicates the number of states needed to use the aerodynamic model. ha145b.fea is the name of the file that contains the state space model matrices, obtained with an approximation chosen by the user. In this particular case, a 10 states Padé approximation has been chosen. This example is taken from the Bisplinghoff Ashley Halfman (BAH) Jet Transport Wing cantilevered wing with modal aerodynamic frequency response, computed by a double-lattice method at Mach 0.0. Data were extracted from the MSC-NASTRAN aeroelastic example file, named ha145b, while the aerodynamic state-space fitting has been computed using a Padé polynomial approximation (by Pasinetti & Mantegazza, [28]). All quantities are expressed in inches and pounds.

8.1.3 Aircraft Instruments

```
<elem_type> ::= aircraft instruments

<normal_arglist> ::= <aircraft_node>
    [ , orientation , { flight mechanics | aeroelasticity
        | (OrientationMatrix) <relative_orientation> } ]
        [ , initial latitude , (real)<initial_latitude>  } ]
        [ , initial longitude , (real)<initial_longitude>  } ]
        [ , earth radius , (real)<earth_radius>  } ]
```

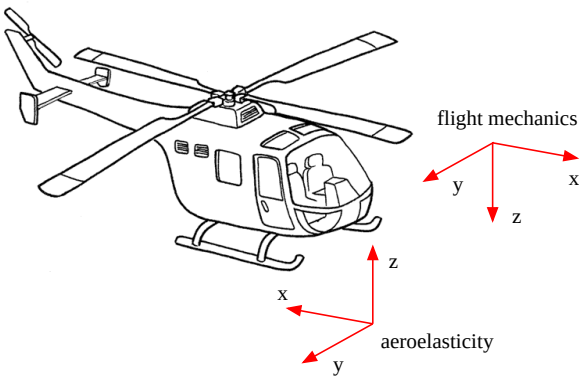
The aircraft_node represents the aircraft; it is assumed that the positive x direction of the node is tail to nose of the aircraft, the positive z direction of the node is top to bottom of the aircraft, and the positive y direction of the node is to the right of the pilot. The node representing the aircraft is intended in a "world" reference frame whose positive z direction points upward and whose positive x direction points north (note that currently the world is flat).

An optional orientation can be added to change the orientation of the aircraft with respect to the node. This is useful, for example, with aeroelasticity, in which conventionally the positive direction of the x axis is nose to tail, and the positive direction of the z axis is bottom to top. The keyword

`flight mechanics` indicates that the node representing the aircraft is oriented according to the default orientation of the element, whereas the keyword `aeroelasticity` indicates that the node representing the aircraft is oriented accordingly. Arbitrary orientations can be dealt with by providing an appropriate `relative_orientation` matrix.

Values for the initial latitude and longitude can be provided, in radians. If not set, they are supposed null at the initial time. Also the earth radius can optionally be provided, with the default value being the average, **6371005 m**.

Please note that longitude and latitude are calculated assuming a "flat" world, discarding the Earth's surface curvature.



The available measures are accessed during the simulation by defining appropriate `parameter` nodes, and by binding the `aircraft instruments` element private data to the nodes by means of the `bind` mechanism, or directly by means of the `element` drive (see Section 2.6.13).

Private Data The following data are available:

- `"airspeed"`: the airspeed as seen by the reference point on the aircraft, i.e. the absolute value of the combination of the airstream speed and of the node speed, with units m/s
- `"groundspeed"`: the absolute value of the projection of the node speed in the xy plane of the "world", with units m/s
- `"altitude"`: the z component of the node position with respect to the "world", in meters
- `"attitude"`: the angle between the x axis of the local frame and the "world" xy plane, i.e. $\sin^{-1}(r_{31})$, in radians
- `"bank"`: the angle between the y axis of the local frame and the "world" xy plane, i.e. $\sin^{-1}(r_{32})$, in radians
- `"turn"`: the turn rate, the z component of the angular velocity expressed in the "world" reference frame, measured in rad/min
- `"slip"`: the sideslip angle, the angle between the x axis of the local frame and the projection of the velocity vector in the xy plane of the local frame, units are radians (also `"sideslip"`)
- `"verticalspeed"`: the z component of the node velocity in the "world" reference system, in m/s
- `"angleofattack"`: the angle between the local z and x components of the velocity (airstream plus node), expressed in the reference frame of the aircraft, in radians (also `"aoa"`)

- `"heading"`: the angle between the x axis of the aircraft and the "north" (the global x axis) about the global z axis, in radians
- `"longitude"`: the aicraft longitude, in radians
- `"latitude"`: the aicraft latitude, in radians
- `"rollrate"`: the component of the angular velocity of the node representing the aircraft along the x axis of the node itself (positive when the right wing moves downward), in rad/s
- `"pitchrate"`: the component of the angular velocity of the node representing the aircraft along the y axis of the node itself (positive when the nose pitches upward), in rad/s
- `"yawrate"`: the component of the angular velocity of the node representing the aircraft along the z axis of the node itself (positive when the right wing moves backward), in rad/s

Additional data, not available through usual aircraft instruments, are available:

- `"init_x1"`: x component of initial position, in absolute frame
- `"init_x2"`: y component of initial position, in absolute frame
- `"init_longitude"`: initial longitude
- `"init_latitude"`: initial latitude
- `"body_axb"`: x component of linear acceleration, in body frame
- `"body_ayb"`: y component of linear acceleration, in body frame
- `"body_azb"`: z component of linear acceleration, in body frame
- `"body_pd"`: x component of pitch angular acceleration, in body frame
- `"body_qd"`: y component of pitch angular acceleration, in body frame
- `"body_rd"`: z component of pitch angular acceleration, in body frame
- `"body_vx"`: x component of air speed, in body frame
- `"body_vy"`: y component of air speed, in body frame
- `"body_vz"`: z component of air speed, in body frame
- `"density"`: density at the body's altitude
- `"soundcelerity"`: sound celerity at the body's altitude
- `"staticpressure"`: static pressure at the body's altitude
- `"temperature"`: temperature at the body's altitude

Example.

```
# expose heading using a parameter node
# ...
structural: AIRCRAFT, dynamic, ...
parameter: HEADING, /* bound to */ element;
# ...
aircraft instruments: AIRCRAFT, AIRCRAFT, orientation, aeroelasticity;
bind: AIRCRAFT, aircraft instruments, HEADING, string, "heading";

# implement a minimal SCAS on roll & yaw
# ...
structural: AIRCRAFT, dynamic, ...
# ...
aircraft instruments: AIRCRAFT, AIRCRAFT, orientation, flight mechanics;
couple: AIRCRAFT, follower,
    AIRCRAFT,
        position, null,
        component,
            element, AIRCRAFT, aerodynamic element, string, "rollrate",
                linear, 0., -ROLL_GAIN,
            0.,
            element, AIRCRAFT, aerodynamic element, string, "yawrate",
                linear, 0., -YAW_GAIN;
```

8.1.4 Air Properties

The properties of the airstream are made of the physical properties of the air plus the description of the airstream velocity direction and amplitude. The former can be expressed in different forms, while the latter are based on three-dimensional drive callers, TplDriveCaller<Vec3>.

```
<elem_type> ::= air properties

<arglist> ::=
    { (DriveCaller) <air_density> , (real) <sound_celerity>
        | std , { { SI | British }
            [ , temperature deviation , <delta_T> ]
            | <p0> , (DriveCaller) <rho0> ,
                <T0> , <dT/dz> , <R> , <g0> , <z1> , <z2> }
        [ , reference altitude , <z0> ] } ,
    (TplDriveCaller<Vec3>) <air_speed>
    [ , gust , <gust_model> [ , ... ] ]
```

The first form consists in the bare input of the air density, in form of a drive caller, and of the sound celerity, e.g.:

```
air properties: 1.225, 340.,
    1.,0.,0., 150.;
```

The second form uses standard air properties, both in the international system (SI) or in British units. A temperature deviation and an altitude offset can be defined, e.g.:

```
air properties: std, SI, temperature deviation, -55,
    reference altitude, 1000.,
    1.,0.,0., 150.;
```

where standard properties in SI are used, with a temperature deviation of -55 K and a reference altitude of 1000 m. The air properties are computed based on the z position of the point where the `air properties` are requested (plus the optional altitude offset). The last possibility lets the user input all the parameters required to compute the `air properties` based on the z position of the point where they are requested, namely the reference pressure `p0`, the reference density `rho0`, the reference temperature `T0`, the initial temperature gradient `dT/dz`, the gas constant `R`, the initial gravity acceleration `g0`, the bottom and top altitudes of the null temperature gradient region `z1` and `z2`; e.g., for SI units:

```
air properties: std,
    101325.,        /* Pa, 1 bar */
    1.2250,         /* kg/m^3 */
    288.16,         /* K, ISA: 15 C */
    -6.5e-3,        /* K/m */
    287.,           /* J/kgK */
    9.81,           /* m/s^2 */
    11000.,         /* m */
    25000.,         /* m */
    temperature deviation, -55,
    reference altitude, 1000.,
    1.,0.,0., 150.;
```

The air properties are defined according to the formulas

$$\left.\begin{aligned} T &= \mathtt{T0} + \mathtt{dT/dz} \cdot z \\ p &= \mathtt{p0} \left(\frac{T}{\mathtt{T0}}\right)^{-\frac{\mathtt{g0}}{\mathtt{dT/dz}\cdot\mathtt{R}}} \\ \rho &= \mathtt{rho0} \left(\frac{T}{\mathtt{T0}}\right)^{-\left(\frac{\mathtt{g0}}{\mathtt{dT/dz}\cdot\mathtt{R}}+1\right)} \end{aligned}\right\} \qquad z < \mathtt{z1} \tag{8.1a}$$

$$\left.\begin{aligned} T &= \mathtt{T0} + \mathtt{dT/dz} \cdot \mathtt{z1} \\ p &= \left(\mathtt{p0} \left(\frac{T}{\mathtt{T0}}\right)^{-\frac{\mathtt{g0}}{\mathtt{dT/dz}\cdot\mathtt{R}}}\right) \mathrm{e}^{-\mathtt{g0}\frac{z-\mathtt{z1}}{\mathtt{dT/dz}\cdot\mathtt{R}}} \\ \rho &= \mathtt{rho0} \left(\frac{T}{\mathtt{T0}}\right)^{-\left(\frac{\mathtt{g0}}{\mathtt{dT/dz}\cdot\mathtt{R}}+1\right)} \mathrm{e}^{-\mathtt{g0}\frac{z-\mathtt{z1}}{\mathtt{dT/dz}\cdot\mathtt{R}}} \end{aligned}\right\} \qquad \mathtt{z1} \le z < \mathtt{z2} \tag{8.1b}$$

The case of $z > \mathtt{z2}$ is not currently handled; the formulas for $z \ge \mathtt{z1}$ are actually used. The value of z is computed by adding the z component of the position where the air properties are computed to the reference altitude `z0`.

The asymptotic air properties are characterized by `air_speed`, the `TplDriveCaller<Vec3>` of the air speed, expressed in the global reference frame. The possibility to vary the density and the airspeed using a driver has little physical meaning, especially the former; is intended as a practical means to gradually introduce the desired air properties, and the related airloads, in the analysis.

Gust

If the optional gust keyword is used, a gust model can be added. Note that a very elementary gust model, represented by a uniform change in airstream speed and direction can be implemented by using a time-dependent airstream drive.

Gusts can also be appended later to the air properties element by using the statement

```
gust : <gust_model> ;
```

Front 1D Gust The syntax of the front 1D gust model is:

```
<gust_model> ::= front 1D ,
    (Vec3)        <front_direction> ,
    (Vec3)        <perturbation_direction> ,
    (real)      <front_velocity> ,
    (DriveCaller) <front_profile>
```

This model consists in a uniform front, defined as

$$\boldsymbol{v}\left(\boldsymbol{x},t\right)=\boldsymbol{n}g\left(V_{\text{ref}}\cdot t-\boldsymbol{f}\cdot\boldsymbol{x}\right)$$

where

- $\boldsymbol{v}$ is the velocity perturbation in the global frame;
- $\boldsymbol{x}$ is the position, in the global frame, of the point whose airstream velocity is being evaluated;
- t is the current time;
- $\boldsymbol{n}$ is the unit vector perturbation_direction that defines the direction of the velocity perturbation in the global frame;
- $g\left(\cdot\right)$ is the function front_profile that defines the gust profile as a function of $x=V_{\text{ref}}\cdot t-\boldsymbol{f}\cdot\boldsymbol{x}$, i.e., the local position (in units of length) of the measure point in a frame moving with the gust front along the $\boldsymbol{f}$ direction;
- $\boldsymbol{f}$ is the unit vector front_direction that defines the direction of propagation of the front in the global frame;
- V_{ref} is the constant velocity front_velocity of propagation of the front in direction $\boldsymbol{f}$.

As an example, a transverse cosine-shaped gust, with a wavelength of 100 m and a peak velocity of 5 m/s moving downstream at the airstream speed, 100 m/s, in standard air, starting along axis x of the global frame at -200 m with respect to the origin of the global frame, is presented:

```
set: real waveLength = 100.; # m
set: real V_inf = 100.;      # m/s
set: real V_g = 5.;          # m/s
set: real x_0 = -200;         # m, position of the gust's local frame at Time = 0.
air properties: std, SI,
    1.,0.,0., const, V_inf,  # reference airstream along X
    gust, front 1D,
        1.,0.,0.,            # front moving along X
        0.,0.,1.,            # gust along Z
        V_inf,               # front moving at V_inf
        cosine, -x_0, pi/waveLength, V_g/2., one, 0.;
        # step, -x_0, V_g, 0.;
```

Scalar Function Wind Profile The syntax of the `scalar function` wind profile, implemented as a gust model within the `air properties`, is:

```
<gust_model> ::= scalar function ,
    reference position , (Vec3) <X0> ,
    reference orientation , (OrientationMatrix) <R0> ,
    (ScalarFunction) <sf>
```

It yields a uniform velocity profile along the x axis of the reference orientation as a function of the z axis component of the relative position; namely, given the relative position

$$z = \boldsymbol{e}_3 \cdot (\boldsymbol{x} - \texttt{X0}) \,, \tag{8.2}$$

the velocity is

$$\boldsymbol{v} = \boldsymbol{e}_1 \cdot \texttt{sf}(z), \tag{8.3}$$

where $\boldsymbol{e}_i$ is the i-th axis of the reference orientation `R0`.

Power Law Wind Profile The syntax of the `power law` wind profile, implemented as a gust model within the `air properties`, is:

```
<gust_model> ::= power law ,
    reference position , (Vec3) <X0> ,
    reference orientation , (OrientationMatrix) <R0> ,
    reference elevation , <z_ref> ,
    reference velocity , (DriveCaller) <v_ref> ,
    exponent , <exponent>
```

It yields a uniform velocity profile along the x axis of the reference orientation as a function of the z axis component of the relative position; namely, given the relative position

$$z = \boldsymbol{e}_3 \cdot (\boldsymbol{x} - \texttt{X0}) \,, \tag{8.4}$$

the velocity is

$$\boldsymbol{v} = \boldsymbol{e}_1 \cdot \texttt{v\_ref} \left(\frac{z}{\texttt{z\_ref}} \right)^{\texttt{exponent}} , \tag{8.5}$$

where $\boldsymbol{e}_i$ is the i-th axis of the reference orientation `R0`. Typical values of `exponent` are about 0.1.

Logarithmic Wind Profile The syntax of the `logarithmic` wind profile, implemented as a gust model within the `air properties`, is:

```
<gust_model> ::= logarithmic ,
    reference position , (Vec3) <X0> ,
    reference orientation , (OrientationMatrix) <R0> ,
    reference elevation , <z_ref> ,
    reference velocity , (DriveCaller) <v_ref> ,
    surface roughness length , <z_0>
```

It yields a uniform velocity profile along the x axis of the reference orientation as a function of the z axis component of the relative position; namely, given the relative position

$$z = \boldsymbol{e}_3 \cdot (\boldsymbol{x} - \texttt{X0}) \,, \tag{8.6}$$

the velocity is

$$\boldsymbol{v} = \boldsymbol{e}_1 \cdot \texttt{v\_ref} \cdot \frac{\log(z/\texttt{z\_0}) - \psi_m}{\log(\texttt{z\_ref}/\texttt{z\_0}) - \psi_m}, \tag{8.7}$$

where $\boldsymbol{e}_i$ is the i-th axis of the reference orientation R0.

The surface roughness length describes the typical roughness of the surrounding surface. It is a very small number in case of smooth surface (e.g. 0.01m for grass), or 1/20 to 1/30 of the typical obstacle's size (e.g. 1m for woods).

Output The output occurs in the `.air` file, which contains:

- a fake label, always set to 1
- the air density
- the sound celerity
- the three components of the reference air speed with respect to the inertial reference frame

Private Data The following data are available:

- `"vxinf"` the x component of the airstream speed (without any gust contribution)
- `"vyinf"` the y component of the airstream speed (without any gust contribution)
- `"vzinf"` the z component of the airstream speed (without any gust contribution)
- `"vinf"` the module of the airstream speed (without any gust contribution)

8.1.5 Generic Aerodynamic Force

This element computes generalized aerodynamic forces parametrized on dynamic pressure and angle of attack and sideslip angle.

```
<elem_type> ::= generic aerodynamic force

<normal_arglist> ::= <node_label> ,
    [ position , <relative_position> , ]
    [ orientation , <relative_orientation> , ]
    [ reference surface , <reference_surface> , ]
    [ reference length , <reference_length> , ]
    [ alpha first , { no | yes } , ]
    { <data_file_specification> | reference , <gaf_data_label> }

<data_file_specification> ::= file ,
    [ { angle units , { radians | degrees }
        | scale angles , <angle_scale_factor> } , ]
    [ scale lengths , <length_scale_factor> , ]
    " <data_file_name> "
```

This element computes aerodynamic forces and moments $\boldsymbol{f}_a$, $\boldsymbol{m}_a$ proportional to the dynamic pressure $q = \rho \boldsymbol{v}^T \boldsymbol{v}/2$ and to empirical coefficients tabulated as functions of the angle of attack α and the sideslip angle β. The angles are computed from the relative airstream velocity $\boldsymbol{v}$ expressed in the reference frame of the body. The velocity is computed at the reference point, optionally offset from the node `node_label` by the offset `relative_position` (defaults to `null`), in a reference frame optionally oriented from that of the node `node_label` by the orientation matrix `relative_orientation` (defaults to `eye`), namely

$$\mathbf{o} = (\texttt{Vec3})\ \texttt{relative\_position} \tag{8.8a}$$

$$\mathbf{R}_h = (\texttt{OrientationMatrix})\ \texttt{relative\_orientation} \tag{8.8b}$$

$$\mathbf{v} = \mathbf{R}_h^T \mathbf{R}_n^T \left(\dot{\boldsymbol{x}}_n + \boldsymbol{\omega}_n \times \mathbf{R}_n \mathbf{o} \right), \tag{8.8c}$$

where $\boldsymbol{x}_n$ and $\mathbf{R}_n$ are the position and orientation of the node, and $\dot{\boldsymbol{x}}_n$ and $\boldsymbol{\omega}_n$ are its linear and angular velocity. By default the angles are computed from the components of $\boldsymbol{v}$ according to the formulas

$$\alpha = \operatorname{atan}\left(\frac{v_3}{\|v_1\|} \right) \tag{8.9a}$$

$$\beta = -\operatorname{atan}\left(\frac{v_2}{\operatorname{sign}(v_1)\sqrt{v_1^2 + v_2^2}} \right); \tag{8.9b}$$

as a consequence, $-\pi/2 \leq \alpha \leq \pi/2$ and $-\pi \leq \beta \leq \pi$. If the optional keyword `alpha first` is set to `yes`, the alternative formulas

$$\alpha = \operatorname{atan}\left(\frac{v_3}{\operatorname{sign}(v_1)\sqrt{v_1^2 + v_2^2}} \right) \tag{8.10a}$$

$$\beta = -\operatorname{atan}\left(\frac{v_2}{\|v_1\|} \right). \tag{8.10b}$$

are used; as a consequence, $-\pi \leq \alpha \leq \pi$ and $-\pi/2 \leq \beta \leq \pi/2$. In both cases, they assume that the body reference system is loosely aligned as:

- axis 1 goes from tail to nose of the aircraft;
- axis 2 points "to the right";
- axis 3 points towards the ground when the aircraft is in level flight.

Force and moment are consistently applied to the node as

$$\boldsymbol{f}_n = \mathbf{R}_n \mathbf{R}_h \boldsymbol{f}_a \tag{8.11a}$$

$$\boldsymbol{m}_n = \boldsymbol{R}_n \mathbf{R}_h \mathbf{m}_a + \mathbf{R}_n \left(\boldsymbol{o} \times (\mathbf{R}_h \boldsymbol{f}_a) \right) \tag{8.11b}$$

Note: this element does not contribute to the Jacobian matrix of the problem; as such, it may deteriorate the convergence properties of the solution when aerodynamic forces dominate the problem. Moreover, it is not suitable for the direct computation of the eigenvalues of a problem about a steady solution.

Note: this element, by itself, may not be adequate to model the rigid body aerodynamics of an aircraft, since it does not account for the dependence of force and moment on angle rates.

Output The following output is available:

1. column 1: element label
2. column 2: the angle of attack α
3. column 3: the sideslip angle β
4. columns 4–6: the components of force $\boldsymbol{f}_a$ in local x, y and z directions
5. columns 7–9: the components of moment $\boldsymbol{m}_a$ in local x, y and z directions, about the reference point (node plus offset)
6. columns 10–12: the components of force $\boldsymbol{f}_n$ in global x, y and z directions
7. columns 13–15: the components of moment $\boldsymbol{m}_n$ in global x, y and z directions, about the node

Private Data The `generic aerodynamic force` element outputs the following private data:

- `"Fx"`, `"Fy"`, `"Fz"`: force components in the global reference frame;
- `"Mx"`, `"My"`, `"Mz"`: moment components in the global reference frame, about the node;
- `"fx"`, `"fy"`, `"fz"`: force components in the local reference frame;
- `"mx"`, `"my"`, `"mz"`: moment components in the local reference frame, about the reference point;
- `"alpha"`: angle of attack (in radian)
- `"beta"`: sideslip angle (in radian)

The values computed during the last residual assembly are returned.

Generic Aerodynamic Element Data

Data is stored in ASCII format in a file.

An arbitrary number of comment lines is allowed at the beginning. Comment lines start with either a percent '`%`' or a hash mark '`#`' in the first column. Their content is discarded until the end of the line.

The first non-comment line must contain two integers separated by whitespace. The integers represent the expected number of angle of attack and sideslip angle values, N_α and N_β.

Another arbitrary number of comment lines is allowed.

A set of $N_\alpha \cdot N_\beta$ lines is expected. No comments or empty lines are allowed in between. Each line contains:

- column 1: the angle of attack, α
- column 2: the sideslip angle, β
- columns 3–5: the force coefficients $f_{x/q}$, $f_{y/q}$, $f_{z/q}$
- columns 6–8: the moment coefficients $m_{x/q}$, $m_{y/q}$, $m_{z/q}$

Notes:

1. lines are sorted as follows: all values of α are defined for each value of β; the same values of α are expected for each value of β;

2. the angle ranges are $-\pi/2 \leq \alpha \leq \pi/2$ and $-\pi \leq \beta \leq \pi$; or, if `alpha first` is `yes`, they are $-\pi \leq \alpha \leq \pi$ and $-\pi/2 \leq \beta \leq \pi/2$;

3. the angles are expected in radians; use the mutually exclusive optional keywords `angle units`, to specify either `radians` or `degrees`, or `scale angles`, to specify the `angle_scale_factor`. The `angle_scale_factor` is the factor that when multiplied by the angle transforms it into radians; for example, if angles are provided in degrees then `angle_scale_factor` $= \pi/180$;

4. $q = 1/2\rho V^2$ is the local reference dynamic pressure, where V is the norm of the velocity at the reference point;

5. the coefficients express forces and moments in the reference frame attached to the body;

6. the coefficients are either expected in dimensional or non-dimensional form. In the former case, the force coefficients represent areas, while the moment coefficients represent volumes, since they need to be multiplied by the dynamic pressure to become forces and moments. In the latter case, they are pure numbers; a `reference_surface` and `reference_length` must be defined in the configuration of the corresponding `generic aerodynamic force` element. When dimensional coefficients are specified, they can be rescaled by using the optional keyword `scale lengths` to specify the `length_scale_factor`.

Example. The content of the file `example.dat` is

```
# This is an example of data for the "generic aerodynamic force" element
# 5 values of angle of attack (alpha) and 4 values of sideslip angle (beta)
# are provided
5 4
# alpha beta fx/q fy/q fz/q mx/q my/q mz/q
-90 -180  0  0  0  0  0  0
-20 -180  0  0  0  0  0  0
  0 -180  0  0  0  0  0  0
 20 -180  0  0  0  0  0  0
 90 -180  0  0  0  0  0  0
-90  -20  0  0  0  0  0  0
-20  -20  0  0  0  0  0  0
  0  -20  0  0  0  0  0  0
 20  -20  0  0  0  0  0  0
 90  -20  0  0  0  0  0  0
-90   20  0  0  0  0  0  0
-20   20  0  0  0  0  0  0
  0   20  0  0  0  0  0  0
 20   20  0  0  0  0  0  0
 90   20  0  0  0  0  0  0
-90  180  0  0  0  0  0  0
-20  180  0  0  0  0  0  0
  0  180  0  0  0  0  0  0
 20  180  0  0  0  0  0  0
 90  180  0  0  0  0  0  0
```

The corresponding statement in the input file is

```
set: integer GAF_NODE = 10;
set: integer GAF_ELEM = 20;
generic aerodynamic force: GAF_ELEM, GAF_NODE,
    file, angle units, degrees, "example.dat";
```

8.1.6 Induced velocity

The induced velocity element is used to associate the aerodynamic elements that model the lifting surfaces of an aircraft, or the blades of a helicopter rotor, when some inflow related computations are required.

By means of different inflow models, and by means of the aerodynamic load contributions supplied by the aerodynamic elements, the induced velocity element is able to compute the induced velocity at an arbitrary point on the lifting surface or rotor disk. This velocity term in turn is used by the aerodynamic elements to determine a better estimate of the boundary conditions.

The syntax of the induced velocity elements is:

```
<elem_type> ::= induced velocity

<normal_arglist> ::= <induced_velocity_type> , <induced_velocity_data>
```

Rotor

Currently, induced velocity models are only implemented for helicopter and cycloidal rotors. For the latter, see Section B.1.3. Originally, this type of element was known as rotor, and the original syntax is preserved for backwards compatibility.

The syntax of the helicopter rotor induced velocity element is:

```
<induced_velocity_type> ::= rotor

<induced_velocity_data> ::= <craft_node> ,
        [ orientation , (OrientationMatrix) <rotor_orientation> , ]
    <rotor_node> ,
    induced velocity , <induced_velocity_model>
```

The optional rotor_orientation is required when axis 3 of the craft_node is not aligned with the rotor axis; axis 3 of the rotor_node must be aligned with the rotor axis.

There are five models of induced velocity. The first is no induced velocity; the syntax is:

```
<induced_velocity_model> ::= no
```

There is no argument list. This element does not compute any induced velocity, but still computes the rotor traction for output purposes, if output is required. The others have a fairly common syntax. The first three are uniform, glauert and mangler induced velocity models:

```
<induced_velocity_model> ::=
    { uniform | glauert [ , type , <glauert_type> ] | mangler } ,
    <reference_omega> , <reference_radius>
    [ , <option> [ , ... ] ]

<glauert_type> ::= { glauert | coleman | drees
    | payne | white and blake | pitt and peters | howlett }
```

```
<option> ::=
    { ground , <ground_node>
        | delay , (DriveCaller) <memory_factor>
        | max iterations , <max_iterations>
        | tolerance , <tolerance>
        | eta , <eta>
        | correction , <hover_correction_factor>, <ff_correction_factor> }
```

- The `reference_omega` field is used to decide whether the induced velocity computation must be inhibited because the rotor speed is very low.
- The `reference_radius` field is used to make the rotor related parameters non-dimensional.
- The `ground` parameter is used to inform the rotor about the proximity to the ground; the z component of the distance between the rotor and the ground nodes, in the ground node reference frame (direction 3, positive), is used for an approximate correction of the axial inflow velocity [29].
- The `memory_factor`, the `hover_correction_factor` and the `ff_correction_factor` (forward flight) are used to correct the nominal induced velocity, according to the formula

$$U_{\text{effective}} = (1 - \texttt{memory\_factor})\, U_{\text{nominal}} + \texttt{memory\_factor}\, U_{\text{previous}}$$

  with

$$U_{\text{nominal}} = \frac{T}{2\rho A V_{\text{tip}} \sqrt{\dfrac{\lambda^2}{\texttt{hover\_correction\_factor}^4} + \dfrac{\mu^2}{\texttt{ff\_correction\_factor}^2}}}$$

  The `delay` parameter is used to linearly combine the current reference induced velocity with the induced velocity at the previous step; no delay means there is no memory of the previous value. The memory factor behaves like a discrete first-order low-pass filter. As a consequence, its behavior depends on the integration time step. The `memory_factor` parameter defaults to 0. The `hover_correction_factor` and `ff_correction_factor` parameters default to 1.
- The `max_iterations`, `tolerance` and `eta` parameters refer to the iteration cycle that is performed to compute the nominal induced velocity. After `max_iterations`, or when the absolute value of the difference between two iterations of the nominal induced velocity is less than `tolerance`, the cycle ends. Only a fraction `eta` of the difference between two iterations of the nominal induced velocity is actually used; `eta` defaults to 1. The default is to make only one iteration, which is backward-compatible with the original behavior.

The last induced velocity model uses a dynamic inflow model, based on [30], with 3 inflow states. The syntax is:

```
<induced_velocity_model> ::= dynamic inflow ,
    <reference_omega> ,
    <reference_radius>
    [ , <option> [ , ... ] ]

<option> ::=
    { ground , <ground_node>
```

```
| initial value , <const_vel> , <cosine_vel> , <sine_vel>
| max iterations , <max_iterations>
| tolerance , <tolerance>
| eta , <eta>
| correction , <hover_correction_factor> , <ff_correction_factor> }
```

Most of the parameters are the same as for the previous models. The optional `delay` parameter is no longer allowed. The three states, corresponding to uniform, fore-aft and lateral inflow, can be explicitly initialized by means of the optional `initial value` parameter.

Output The following output is available for all rotor elements:

1. column 1: element label
2. columns 2–4: rotor force in x, y and z directions (longitudinal, lateral and thrust components)
3. columns 5–7: rotor moment about x, y and z directions (roll, pitch and torque components)
4. column 8: mean inflow velocity, based on momentum theory
5. column 9: reference velocity at rotor center, sum of airstream and `craft_node` node velocity
6. column 10: rotor disk angle
7. column 11: advance parameter μ
8. column 12: inflow parameter λ
9. column 13: advance/inflow angle $\chi = \tan^{-1}(\mu/\lambda)$
10. column 14: reference azimuthal direction ψ_0, related to rotor yaw angle
11. column 15: boolean flag indicating convergence in reference induced velocity computation internal iterations
12. column 16: number of iterations required for convergence

The `dynamic inflow` model adds the columns

13. column 17: constant inflow state
14. column 18: sine inflow state (lateral)
15. column 19: cosine inflow state (longitudinal)

Rotor force and moment (columns 2–4 and 5–7) are the aerodynamic force and moment exerted by the rotor aerodynamics on the `rotor_node`, projected in the reference frame of the `craft_node`, optionally modified by the `rotor_orientation` matrix. The conventional naming of longitudinal (or drag), lateral and thrust force, and roll, pitch and torque moment, refer to a rotorcraft whose x axis is the longitudinal (nose to tail) axis, whose y axis is the lateral (portside) axis, and whose z axis is the vertical (bottom to top) axis.

Private Data The following data are available:

1. `"Tx"` rotor force in x direction (longitudinal force)
2. `"Ty"` rotor force in y direction (lateral force)
3. `"Tz"` rotor force in z direction (thrust)
4. `"Mx"` rotor moment about x direction (roll moment)
5. `"My"` rotor moment about y direction (pitch moment)
6. `"Mz"` rotor moment about z direction (torque)

The rotor force and moment components are expressed in the same reference frame described in the Output Section above.

8.1.7 Rotor

Deprecated; see `induced velocity` (Section 8.1.6).

8.2 Automatic structural

The so called `automatic structural` element is automatically generated when a dynamic structural node is instantiated. As such, when defined in the `elements` block, the element already exists. The only reason to repeat its definition is to modify the values of the momentum and of the momenta moment, and to initialize their derivatives. The label must match that of the node it refers to.

```
<elem_type> ::= automatic structural

# for 6 dof structural nodes
<normal_arglist> ::=
    (Vec3) <momentum> ,
    (Vec3) <momenta_moment> ,
    (Vec3) <momentum_derivative> ,
    (Vec3) <momenta_moment_derivative>

# for 3 dof structural nodes
<normal_arglist> ::=
    (Vec3) <momentum> ,
    (Vec3) <momentum_derivative>
```

All the provided values are recomputed during the initial derivatives phase, so they should be intended as initial values for the Newton iteration. In general, there is no need to provide this data; they can speed up initial convergence in case of systems that are not at rest in the initial configuration, with kinematic constraints that strongly affect the motion.

Private Data The following data are available:

1. `"beta[1]"` momentum in global direction 1
2. `"beta[2]"` momentum in global direction 2
3. `"beta[3]"` momentum in global direction 3

4. `"gamma[1]"` momenta moment in global direction 1
5. `"gamma[2]"` momenta moment in global direction 2
6. `"gamma[3]"` momenta moment in global direction 3
7. `"betaP[1]"` momentum derivative in global direction 1
8. `"betaP[2]"` momentum derivative in global direction 2
9. `"betaP[3]"` momentum derivative in global direction 3
10. `"gammaP[1]"` momenta moment derivative in global direction 1
11. `"gammaP[2]"` momenta moment derivative in global direction 2
12. `"gammaP[3]"` momenta moment derivative in global direction 3
13. `"KE"` kinetic energy

8.3 Beam Elements

The family of finite volume beam elements implemented in MBDyn allows to model slender deformable structural components with a high level of flexibility.

The beam is defined by a reference line and by a manifold of orientations attached to the line. It is assumed that the direction 1 of the orientations lies along the reference line, but it is not strictly required to be tangent to it even in the reference configuration.

The beam element is defined by its nodes; currently, 2 and 3 node beam elements are implemented. Each node of the beam is related to a `structural node` by an offset and an optional relative orientation, to provide topological flexibility.

The beam element is modeled by means of an original Finite Volume approach [31], which computes the internal forces as functions of the straining of the reference line and orientation at selected points along the line itself, called *evaluation points*, which lie somewhere between two pairs of beam nodes.

At each evaluation point, a 6D constitutive law must be defined, which defines the relationship between the strains, the curvatures of the beam and their time derivatives and the internal forces and moments at the evaluation points.

The strains and curvatures and their time derivatives are obtained from the nodal positions and orientations by differentiating the interpolation functions.

The 6D constitutive laws are defined as

$$\begin{Bmatrix} F_x \\ F_y \\ F_z \\ M_x \\ M_y \\ M_z \end{Bmatrix} = \boldsymbol{f}\left(\begin{Bmatrix} \varepsilon_x \\ \gamma_y \\ \gamma_z \\ \kappa_x \\ \kappa_y \\ \kappa_z \end{Bmatrix}, \begin{Bmatrix} \dot{\varepsilon}_x \\ \dot{\gamma}_y \\ \dot{\gamma}_z \\ \dot{\kappa}_x \\ \dot{\kappa}_y \\ \dot{\kappa}_z \end{Bmatrix}\right)$$

where, if the convention of using x as beam axis is followed:

- F_x is the axial force component;
- F_y and F_z are the shear force components;
- M_x is the torsional moment component;

- M_y and M_z are the bending moment components;
- ε_x is the axial strain component;
- γ_y and γ_z are the shear strain components;
- κ_x is the torsional curvature component;
- κ_y and κ_z are the bending curvature component;
- $\boldsymbol{f}$ is an arbitrary function that defines the constitutive law.

8.3.1 Beam Section Constitutive Law

Typically, linear elastic or viscoelastic constitutive laws are used, although one may want to implement specific nonlinear elastic or elastic-plastic constitutive laws.

Beam Section Characterization

MBDyn allows the broadest generality in defining what a linear elastic constitutive law contains, since the entire 6×6 constitutive matrix can be input. This means that internal forces and moments can be arbitrarily related to generalized strains and curvatures. However, to make sense, a constitutive matrix at the section level, must satisfy some constraints, e.g. it is expected to be symmetric, although this is not strictly enforced by the code.

However, most of the info about the extra-diagonal terms of the stiffness matrix are not usually available. One easy way to work this around is to resort to any so-called composite beam section characterization analysis available in the literature.

For details, the reader is referred to [32] for a review of the topic, to [33] for an early work on the subject, and to [34] for a more recent review of the original formulation. The software that implements this type of analysis is called ANBA++. It is not free software, so far. Prospective users can contact the authors, through MBDyn developers.

Disclaimer

The following paragraphs are intended as a means to help users preparing data for MBDyn models in a consistent manner. By no means they indicate that the beam section stiffness properties must be provided in a specific reference frame. On the contrary, MBDyn allows as much generality as possible, and actually the variety of choices is redundant, since equivalent properties can be input in different ways.

This is intended to allow the code to suit the users' needs regardless of the original format of the input data. As such, all the transformations reported in the following are only intended as suggestions and should not be taken literally. For instance, rotations and offsets of reference points could be reversed, changing the values of the offsets, without affecting the final result.

The most important aspect of MBDyn notion of beam section properties is that the reference point and orientation, although arbitrary, must be unique, and the common notions of center of axial strain, shear center (and center of mass) have no special meaning.

Equivalent 6×6 Section of Isotropic Beam

When an isotropic beam section is considered, the $6\times$ constitutive matrix, referred to an arbitrary point in the section, with an arbitrary orientation, can always be written in terms of elementary stiffness and geometrical properties. These are the properties that are usually available in tabular form either from

simplified beam section analysis or by experiments. A sketch of a generic section is shown in Figure 8.3, where the arbitrary reference frame indicated by axes x, y and z originates from an arbitrary reference point on the section.

Isotropic uniform beam sections allow to group the internal forces and moments in two sets, together with their conjugated generalized strains: those related to shear stress and strain, and those related to axial stress and strain, as illustrated in Figure 8.2. There is no direct coupling between the two groups,

	ε_x	γ_y	γ_z	κ_x	κ_y	κ_z
F_x	A				A	A
F_y		S	S	S		
F_z		S	S	S		
M_x		S	S	S		
M_y	A				A	A
M_z	A				A	A

Figure 8.2: Constitutive coefficients grouping (S: shear, A: axial)

at the section level, so the corresponding coupling coefficients are always zero. This is no longer true when material anisotropy must be taken into account.

The 3×3 sub-blocks can be separately transformed in diagonal form by referring the corresponding properties to appropriate separate points in the beam section, and by applying an appropriate rotation about the axis of the beam.

Axial Stress and Strain Properties

This section considers the submatrix represented by the coefficients marked as A in Figure 8.2, under the assumption that it is symmetric.

First, the problem of obtaining axial stiffness properties referred to a generic point in a generic orientation is considered, when the properties referred to the axial strain center in the principal reference frame are known.

Then, the problem of extracting the location of the axial strain center and of the principal reference frame, and the principal bending stiffnesses from generic data is presented as well.

The two problems are complementary. Usually, the first one needs to be considered when engineering properties are available and the generic constitutive properties required by MBDyn need to be computed.

Diagonal to Generic Properties. First the transformation from diagonal to generic properties is considered. This transformation consists in rotating the section properties and then in referring them to a common reference point in the blade section.

The diagonal properties are described by the constitutive matrix

$$\left\{ \begin{array}{c} F_x \\ M_y \\ M_z \end{array} \right\}^{\dagger} = \left[\begin{array}{ccc} EA & 0 & 0 \\ & EJ_y & 0 \\ \text{sym.} & & EJ_z \end{array} \right] \left\{ \begin{array}{c} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{array} \right\}. \tag{8.12}$$

This constitutive matrix is expressed in a reference frame that is centered in the center of axial strain, indicated with the subscript as, and oriented according to the bending principal axes.

A rotation α about axis x is used to transform the properties into the common reference frame of the

beam section. The internal forces and moments are thus transformed according to

$$\begin{aligned}\left\{\begin{array}{c} F_x \\ M_y \\ M_z \end{array}\right\}^* &= [R]_{\text{axial}} \left\{\begin{array}{c} F_x \\ M_y \\ M_z \end{array}\right\}^\dagger \\ &= \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{array}\right] \left\{\begin{array}{c} F_x \\ M_y \\ M_z \end{array}\right\}^\dagger, \end{aligned} \tag{8.13}$$

while the strains and curvatures are transformed according to

$$\begin{aligned}\left\{\begin{array}{c} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{array}\right\}^* &= [R]^{-T}_{\text{axial}} \left\{\begin{array}{c} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{array}\right\}^\dagger \\ &= [R]_{\text{axial}} \left\{\begin{array}{c} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{array}\right\}^\dagger. \end{aligned} \tag{8.14}$$

As a consequence, the constitutive relationship becomes

$$\begin{aligned}\left\{\begin{array}{c} F_x \\ M_y \\ M_z \end{array}\right\}^* &= [R]_{\text{axial}} [A]^\dagger [R]^T_{\text{axial}} \left\{\begin{array}{c} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{array}\right\}^* \\ &= \left[\begin{array}{ccc} EA & 0 & 0 \\ & EJ_y\cos^2\alpha + EJ_z\sin^2\alpha & (EJ_y - EJ_z)\cos\alpha\sin\alpha \\ \text{sym.} & & EJ_z\cos^2\alpha + EJ_y\sin^2\alpha \end{array}\right] \left\{\begin{array}{c} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{array}\right\}^*. \end{aligned} \tag{8.15}$$

An offset of the reference point results from the internal force and moment transformation

$$\begin{aligned}\left\{\begin{array}{c} F_x \\ M_y \\ M_z \end{array}\right\} &= [T]_{\text{axial}} \left\{\begin{array}{c} F_x \\ M_y \\ M_z \end{array}\right\}^* \\ &= \left[\begin{array}{ccc} 1 & 0 & 0 \\ z_{as} & 1 & 0 \\ -y_{as} & 0 & 1 \end{array}\right] \left\{\begin{array}{c} F_x \\ M_y \\ M_z \end{array}\right\}^*. \end{aligned} \tag{8.16}$$

Similarly, the strains and curvatures are transformed according to the relationship

$$\left\{\begin{array}{c} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{array}\right\} = [T]^{-T}_{\text{axial}} \left\{\begin{array}{c} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{array}\right\}^*. \tag{8.17}$$

The constitutive relationship becomes

$$\begin{aligned}\left\{\begin{array}{c} F_x \\ M_y \\ M_z \end{array}\right\} &= [T]_{\text{axial}} [A]^* [T]^T_{\text{axial}} \left\{\begin{array}{c} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{array}\right\} \\ &= [T]_{\text{axial}} [R]_{\text{axial}} [A]^\dagger [R]^T_{\text{axial}} [T]^T_{\text{axial}} \left\{\begin{array}{c} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{array}\right\} \\ &= \left[\begin{array}{ccc} A_{11} & A_{12} & A_{13} \\ & A_{22} & A_{23} \\ \text{sym.} & & A_{33} \end{array}\right] \left\{\begin{array}{c} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{array}\right\}. \end{aligned} \tag{8.18}$$

The values of the coefficients are

$$A_{11} = EA \tag{8.19a}$$
$$A_{12} = z_{as} EA \tag{8.19b}$$
$$A_{13} = -y_{as} EA \tag{8.19c}$$
$$A_{22} = EJ_y \cos^2\alpha + EJ_z \sin^2\alpha + z_{as}^2 EA \tag{8.19d}$$
$$A_{23} = (EJ_y - EJ_z)\cos\alpha \sin\alpha - y_{as} z_{as} EA \tag{8.19e}$$
$$A_{33} = EJ_z \cos^2\alpha + EJ_y \sin^2\alpha + y_{as}^2 EA \tag{8.19f}$$

Generic to Diagonal Properties. Consider now a generic axial portion of the constitutive properties, symmetric and usually positive-definite. The constitutive matrix can be transformed to diagonal form by moving the reference point by an offset in the plane of the section, and then by rotating the properties about axis x.

The offset is applied by the internal forces and moments transformation

$$\begin{Bmatrix} F_x \\ M_y \\ M_z \end{Bmatrix}^* = [T]^{-1}_{\text{axial}} \begin{Bmatrix} F_x \\ M_y \\ M_z \end{Bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -z_{as} & 1 & 0 \\ y_{as} & 0 & 1 \end{bmatrix} \begin{Bmatrix} F_x \\ M_y \\ M_z \end{Bmatrix}. \tag{8.20}$$

The corresponding strains and curvatures transformation is

$$\begin{Bmatrix} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{Bmatrix}^* = [T]^{T}_{\text{axial}} \begin{Bmatrix} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{Bmatrix}. \tag{8.21}$$

The transformed constitutive relationship is

$$\begin{Bmatrix} F_x \\ M_y \\ M_z \end{Bmatrix}^* = [T]^{-1}_{\text{axial}} [A] [T]^{-T}_{\text{axial}} \begin{Bmatrix} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{Bmatrix} = \begin{bmatrix} A_{11} & A_{12} - z_{as}A_{11} & A_{13} + y_{as}A_{11} \\ & A_{22} - 2z_{as}A_{12} + z_{as}^2 A_{11} & A_{23} - z_{as}A_{13} + y_{as}A_{12} - y_{as}z_{as}A_{11} \\ \text{sym.} & & A_{33} + 2y_{as}A_{13} + y_{as}^2 A_{11} \end{bmatrix} \begin{Bmatrix} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{Bmatrix}^*. \tag{8.22}$$

The location that decouples the axial force from the bending moments is

$$y_{as} = -\frac{A_{13}}{A_{11}} \tag{8.23a}$$
$$z_{as} = \frac{A_{12}}{A_{11}} \tag{8.23b}$$

When the location of Eq. (8.23) is considered, Eq. (8.22) becomes

$$\begin{Bmatrix} F_x \\ M_y \\ M_z \end{Bmatrix}^* = \begin{bmatrix} A_{11} & 0 & 0 \\ & A_{22} - A_{12}^2/A_{11} & A_{23} - A_{12}A_{13}/A_{11} \\ \text{sym.} & & A_{33} - A_{13}^2/A_{11} \end{bmatrix} \begin{Bmatrix} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{Bmatrix}^*$$
$$= \begin{bmatrix} A_{11}^* & 0 & 0 \\ & A_{22}^* & A_{23}^* \\ \text{sym.} & & A_{33}^* \end{bmatrix} \begin{Bmatrix} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{Bmatrix}^* . \tag{8.24}$$

When a rotation about axis x is considered, the internal forces and moments are transformed according to the relationship

$$\begin{Bmatrix} F_x \\ M_y \\ M_z \end{Bmatrix}^\dagger = [R]_{\text{axial}}^T \begin{Bmatrix} F_x \\ M_y \\ M_z \end{Bmatrix}^* , \tag{8.25}$$

and the strains and curvatures are transformed according to

$$\begin{Bmatrix} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{Bmatrix}^\dagger = [R]_{\text{axial}}^T \begin{Bmatrix} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{Bmatrix}^* . \tag{8.26}$$

The constitutive relationship becomes

$$\begin{Bmatrix} F_x \\ M_y \\ M_z \end{Bmatrix}^\dagger = [R]_{\text{axial}}^T [A]^* [R]_{\text{axial}} \begin{Bmatrix} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{Bmatrix}^\dagger$$
$$= \begin{bmatrix} A_{11}^\dagger & 0 & 0 \\ & A_{22}^\dagger & A_{23}^\dagger \\ \text{sym.} & & A_{33}^\dagger \end{bmatrix} \begin{Bmatrix} \varepsilon_x \\ \kappa_y \\ \kappa_z \end{Bmatrix}^\dagger , \tag{8.27}$$

with

$$A_{11}^\dagger = A_{11}^* \tag{8.28a}$$
$$A_{22}^\dagger = A_{22}^* \cos^2\alpha + A_{33}^* \sin^2\alpha + 2A_{23}^* \cos\alpha \sin\alpha \tag{8.28b}$$
$$A_{23}^\dagger = (A_{33}^* - A_{22}^*) \cos\alpha \sin\alpha + A_{23}^* \left(\cos^2\alpha - \sin^2\alpha\right) \tag{8.28c}$$
$$A_{33}^\dagger = A_{33}^* \cos^2\alpha + A_{22}^* \sin^2\alpha - 2A_{23}^* \cos\alpha \sin\alpha. \tag{8.28d}$$

The constitutive relationship is diagonal when $A_{23}^\dagger = 0$, namely

$$\alpha = \frac{1}{2} \tan^{-1} \left(\frac{A_{22}^* - A_{33}^*}{2A_{23}^*} \right)$$
$$= \frac{1}{2} \tan^{-1} \left(\frac{A_{11} (A_{22} - A_{33}) - A_{12}^2 + A_{13}^2}{A_{11}A_{23} - A_{12}A_{13}} \right). \tag{8.29}$$

Shear Stress and Strain Properties

Consider now the submatrix represented by the coefficients marked as S in Figure 8.2, under the assumption that it is symmetric, as indicated in Equation (8.30):

$$\begin{Bmatrix} F_y \\ F_z \\ M_x \end{Bmatrix} = \begin{bmatrix} S_{11} & S_{12} & S_{13} \\ & S_{22} & S_{23} \\ \text{sym.} & & S_{33} \end{bmatrix} \begin{Bmatrix} \gamma_y \\ \gamma_z \\ \kappa_x \end{Bmatrix} \tag{8.30}$$

The orientation of the shear force components about the section axis can be selected in order to decouple them; by applying the transformation

$$\begin{aligned}\left\{\begin{array}{c} F_y \\ F_z \\ M_x \end{array}\right\}^* &= [R_{\text{shear}}] \left\{\begin{array}{c} F_y \\ F_z \\ M_x \end{array}\right\} \\ &= \left[\begin{array}{ccc} \cos\beta & -\sin\beta & 0 \\ \sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{array}\right] \left\{\begin{array}{c} F_y \\ F_z \\ M_x \end{array}\right\} \end{aligned} \tag{8.31}$$

The angle that decouples the shear forces is

$$\beta = \frac{1}{2}\tan^{-1}\left(\frac{2S_{12}}{S_{22}-S_{11}}\right)$$

representing a rotation about the axis x of the beam with respect to the origin of the initial reference frame as shown in Figure 8.3, and the resulting coefficients are

$$GA_y = S_{11}\cos^2\beta + S_{22}\sin^2\beta - 2S_{12}\sin\beta\cos\beta \tag{8.32}$$

$$GA_z = S_{11}\sin^2\beta + S_{22}\cos^2\beta + 2S_{12}\sin\beta\cos\beta \tag{8.33}$$

the shear block becomes

$$\begin{aligned}\left\{\begin{array}{c} F_y \\ F_x \\ M_x \end{array}\right\}^* &= \left[\begin{array}{ccc} GA_y & 0 & S_{13}\cos\beta - S_{23}\sin\beta \\ & GA_z & S_{13}\sin\beta + S_{23}\cos\beta \\ \text{sym.} & & S_{33} \end{array}\right] \left\{\begin{array}{c} \gamma_y \\ \gamma_z \\ \kappa_x \end{array}\right\}^* \\ &= \left[\begin{array}{ccc} GA_y & 0 & S^*_{13} \\ & GA_z & S^*_{23} \\ \text{sym.} & & S_{33} \end{array}\right] \left\{\begin{array}{c} \gamma_y \\ \gamma_z \\ \kappa_x \end{array}\right\}^* \end{aligned}$$

The transformation of Equation (8.34) moves the point of application of the shear force of an arbitrary amount $\{y, z\}$ in the beam section, with respect to the reference frame rotated by β about the axis x of the beam, as indicated in Figure 8.3:

$$\begin{aligned}\left\{\begin{array}{c} F_y \\ F_z \\ M_x \end{array}\right\}^\dagger &= [T_{\text{shear}}] \left\{\begin{array}{c} F_y \\ F_z \\ M_x \end{array}\right\}^* \\ &= \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ z & -y & 1 \end{array}\right] \left\{\begin{array}{c} F_y \\ F_z \\ M_x \end{array}\right\}^* \end{aligned} \tag{8.34}$$

So the transformed shear block of the constitutive matrix becomes

$$\begin{aligned}\left\{\begin{array}{c} F_y \\ F_z \\ M_x \end{array}\right\}^\dagger &= [T_{\text{shear}}] \left\{\begin{array}{c} F_y \\ F_z \\ M_x \end{array}\right\}^* \\ &= [T_{\text{shear}}]\,[A]\,[T_{\text{shear}}]^T \left\{\begin{array}{c} \gamma_y \\ \gamma_z \\ \kappa_x \end{array}\right\}^\dagger \\ &= \left[\begin{array}{ccc} GA_y & 0 & S^*_{13} + zGA_y \\ & GA_z & S^*_{23} - yGA_z \\ \text{sym.} & & S_{33} - yS^*_{23} + zS^*_{13} + z\left(S^*_{13} + zGA_y\right) - y\left(S^*_{23} - yGA_z\right) \end{array}\right] \left\{\begin{array}{c} \gamma_y \\ \gamma_z \\ \kappa_x \end{array}\right\}^\dagger \end{aligned} \tag{8.35}$$

If the position of the point is selected in such a manner that the shear force and the torsional moment are decoupled, i.e., according to the definition of center of shear force (the point in a beam section where the application of a transverse force results in no twist)

$$\begin{aligned} y &= \frac{S^*_{23}}{GA_z} \\ z &= -\frac{S^*_{13}}{GA_y} \end{aligned}$$

the shear block becomes

$$\begin{aligned} \left\{ \begin{array}{c} F_y \\ F_x \\ M_x \end{array} \right\}^\dagger &= \left[\begin{array}{ccc} GA_y & 0 & 0 \\ & GA_z & 0 \\ \text{sym.} & & S_{33} - {S^*_{13}}^2/GA_y - {S^*_{23}}^2/GA_z \end{array} \right] \left\{ \begin{array}{c} \gamma_y \\ \gamma_z \\ \kappa_x \end{array} \right\}^\dagger \\ &= \left[\begin{array}{ccc} GA_y & 0 & 0 \\ & GA_z & 0 \\ \text{sym.} & & GJ \end{array} \right] \left\{ \begin{array}{c} \gamma_y \\ \gamma_z \\ \kappa_x \end{array} \right\}^\dagger \end{aligned}$$

When the shear and torsional stiffnesses, and the position of the shear strain center and the orientation of the shear axes are available, the shear portion of the stiffness matrix can be computed by reversing the order of the transformations described in Equations (8.31–8.34), i.e.:

$$\left[\begin{array}{ccc} S_{11} & S_{12} & S_{13} \\ & S_{22} & S_{23} \\ \text{sym.} & & S_{33} \end{array} \right] = [R_{\text{shear}}]^T \, [T_{\text{shear}}]^{-1} \left[\begin{array}{ccc} GA_y & 0 & 0 \\ 0 & GA_z & 0 \\ 0 & 0 & GJ \end{array} \right] [T_{\text{shear}}]^{-T} \, [R_{\text{shear}}] \tag{8.36}$$

This expression implies that the stiffness properties are referred to an arbitrary point at $\{-y, -z\}$ from the shear center, in the shear reference frame, followed by a rotation into the section reference frame by an amount $-\beta$. The resulting coefficients are

$$\begin{aligned} S_{11} &= GA_y \cos^2\beta + GA_z \sin^2\beta \\ S_{12} &= (GA_z - GA_y) \sin\beta \cos\beta \\ S_{13} &= yGA_z \sin\beta - zGA_y \cos\beta \\ S_{22} &= GA_z \cos^2\beta + GA_y \sin^2\beta \\ S_{23} &= yGA_z \cos\beta + zGA_y \sin\beta \\ S_{33} &= GJ + z^2 GA_y + y^2 GA_z \end{aligned}$$

Note that the order of the rotation and reference point transportation is reversed with respect to the axial properties; this is mostly done for convenience in computing the coefficients, because the opposite would result in more complicated formulas; however, their development the other way 'round is straightforward.

Generic Anisotropic Beam Section

When a generic anisotropic beam section is considered, the partitioning of axial and shear constitutive properties of Figure 8.2 is no longer possible. The axial and shear straining can be completely and arbitrarily coupled, resulting in a full constitutive matrix of the beam section,

$$\left\{ \begin{array}{c} \boldsymbol{f} \\ \boldsymbol{m} \end{array} \right\}^\dagger = \left[\begin{array}{cc} \mathbf{K}_{\boldsymbol{f\nu}} & \mathbf{K}_{\boldsymbol{f\kappa}} \\ \mathbf{K}_{\boldsymbol{m\nu}} & \mathbf{K}_{\boldsymbol{m\kappa}} \end{array} \right]^\dagger \left\{ \begin{array}{c} \boldsymbol{\nu} \\ \boldsymbol{\kappa} \end{array} \right\}^\dagger, \tag{8.37}$$

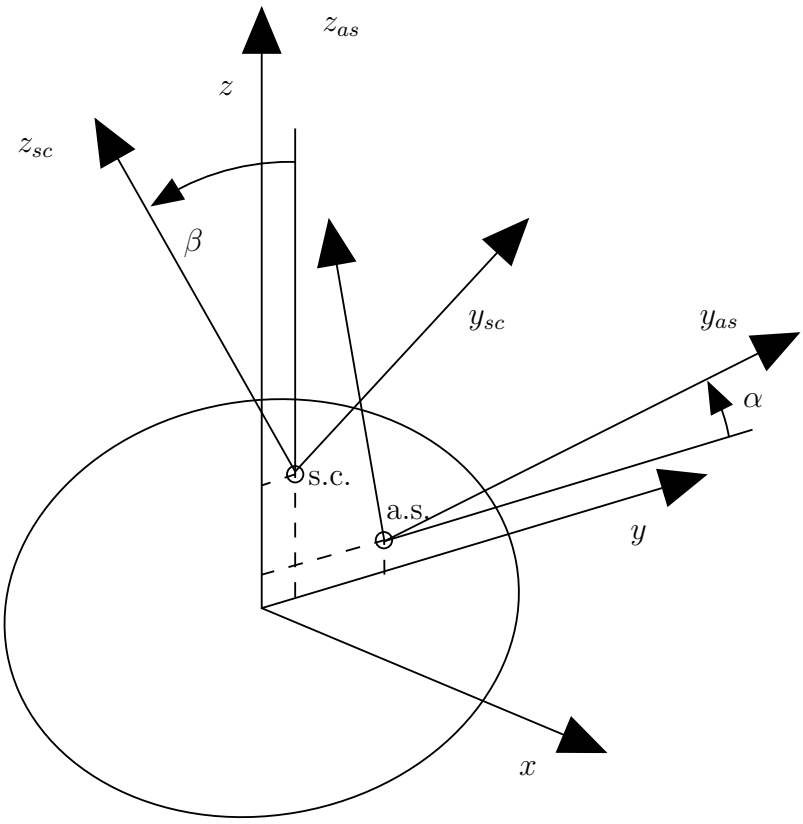


Figure 8.3: Beam section

where $\boldsymbol{f}$ and $\boldsymbol{m}$ are the internal force and moment vectors, while $\boldsymbol{\nu}$ and $\boldsymbol{\kappa}$ are the linear and angular strains. Usually, $\mathbf{K}_{\boldsymbol{m\nu}} \equiv \mathbf{K}_{\boldsymbol{f\kappa}}^T$, while $\mathbf{K}_{\boldsymbol{f\nu}}$ and $\mathbf{K}_{\boldsymbol{m\kappa}}$ are symmetric.

The reference point or the reference orientation of the constitutive matrix can be changed by a sequence of transformations consisting in a rotation and an offset.

The internal force and moment, after a rotation defined by the rotation matrix $\mathbf{R}$, become

$$\left\{ \begin{array}{c} \boldsymbol{f} \\ \boldsymbol{m} \end{array} \right\}^* = \left[\begin{array}{cc} \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} \end{array} \right] \left\{ \begin{array}{c} \boldsymbol{f} \\ \boldsymbol{m} \end{array} \right\}^\dagger . \tag{8.38}$$

Similarly, the strains become

$$\begin{aligned} \left\{ \begin{array}{c} \boldsymbol{\nu} \\ \boldsymbol{\kappa} \end{array} \right\}^* &= \left[\begin{array}{cc} \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} \end{array} \right]^{-T} \left\{ \begin{array}{c} \boldsymbol{\nu} \\ \boldsymbol{\kappa} \end{array} \right\}^\dagger \\ &= \left[\begin{array}{cc} \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} \end{array} \right] \left\{ \begin{array}{c} \boldsymbol{\nu} \\ \boldsymbol{\kappa} \end{array} \right\}^\dagger . \end{aligned} \tag{8.39}$$

As a consequence, the re-oriented constitutive relationship is

$$\begin{aligned} \left\{ \begin{array}{c} \boldsymbol{f} \\ \boldsymbol{m} \end{array} \right\}^* &= \left[\begin{array}{cc} \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} \end{array} \right] \left[\begin{array}{cc} \mathbf{K}_{\boldsymbol{f\nu}} & \mathbf{K}_{\boldsymbol{f\kappa}} \\ \mathbf{K}_{\boldsymbol{m\nu}} & \mathbf{K}_{\boldsymbol{m\kappa}} \end{array} \right]^\dagger \left[\begin{array}{cc} \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} \end{array} \right]^T \left\{ \begin{array}{c} \boldsymbol{\nu} \\ \boldsymbol{\kappa} \end{array} \right\}^* \\ &= \left[\begin{array}{cc} \mathbf{R}\mathbf{K}_{\boldsymbol{f\nu}}^\dagger \mathbf{R}^T & \mathbf{R}\mathbf{K}_{\boldsymbol{f\kappa}}^\dagger \mathbf{R}^T \\ \mathbf{R}\mathbf{K}_{\boldsymbol{m\nu}}^\dagger \mathbf{R}^T & \mathbf{R}\mathbf{K}_{\boldsymbol{m\kappa}}^\dagger \mathbf{R}^T \end{array} \right] \left\{ \begin{array}{c} \boldsymbol{\nu} \\ \boldsymbol{\kappa} \end{array} \right\}^* \\ &= \left[\begin{array}{cc} \mathbf{K}_{\boldsymbol{f\nu}} & \mathbf{K}_{\boldsymbol{f\kappa}} \\ \mathbf{K}_{\boldsymbol{m\nu}} & \mathbf{K}_{\boldsymbol{m\kappa}} \end{array} \right]^* \left\{ \begin{array}{c} \boldsymbol{\nu} \\ \boldsymbol{\kappa} \end{array} \right\}^* . \end{aligned} \tag{8.40}$$

The internal moment, after considering an offset $\boldsymbol{o} = [0, y, z]^T$ of the reference point of the constitutive properties, becomes $\boldsymbol{m} = \boldsymbol{m}^* + \boldsymbol{o} \times \boldsymbol{f}^*$. The internal force and moment then becomes

$$\left\{ \begin{array}{c} \boldsymbol{f} \\ \boldsymbol{m} \end{array} \right\} = \left[\begin{array}{cc} \mathbf{I} & \mathbf{0} \\ \boldsymbol{o}\times & \mathbf{I} \end{array} \right] \left\{ \begin{array}{c} \boldsymbol{f} \\ \boldsymbol{m} \end{array} \right\}^* . \tag{8.41}$$

Similarly, the linear and angular strain vectors become

$$\left\{ \begin{array}{c} \boldsymbol{\nu} \\ \boldsymbol{\kappa} \end{array} \right\} = \left[\begin{array}{cc} \mathbf{I} & \mathbf{0} \\ \boldsymbol{o}\times & \mathbf{I} \end{array} \right]^{-T} \left\{ \begin{array}{c} \boldsymbol{\nu} \\ \boldsymbol{\kappa} \end{array} \right\}^* . \tag{8.42}$$

As a consequence, the offset constitutive relationship becomes

$$\begin{aligned} \left\{ \begin{array}{c} \boldsymbol{f} \\ \boldsymbol{m} \end{array} \right\} &= \left[\begin{array}{cc} \mathbf{I} & \mathbf{0} \\ \boldsymbol{o}\times & \mathbf{I} \end{array} \right] \left[\begin{array}{cc} \mathbf{K}_{\boldsymbol{f\nu}} & \mathbf{K}_{\boldsymbol{f\kappa}} \\ \mathbf{K}_{\boldsymbol{m\nu}} & \mathbf{K}_{\boldsymbol{m\kappa}} \end{array} \right]^* \left[\begin{array}{cc} \mathbf{I} & \boldsymbol{o}\times^T \\ \mathbf{0} & \mathbf{I} \end{array} \right] \left\{ \begin{array}{c} \boldsymbol{\nu} \\ \boldsymbol{\kappa} \end{array} \right\} \\ &= \left[\begin{array}{cc} \mathbf{K}_{\boldsymbol{f\nu}}^* & \mathbf{K}_{\boldsymbol{f\nu}}^* \boldsymbol{o}\times^T + \mathbf{K}_{\boldsymbol{f\kappa}}^* \\ \boldsymbol{o}\times \mathbf{K}_{\boldsymbol{f\nu}}^* + \mathbf{K}_{\boldsymbol{m\nu}}^* & \boldsymbol{o}\times \mathbf{K}_{\boldsymbol{f\nu}}^* \boldsymbol{o}\times^T + \mathbf{K}_{\boldsymbol{m\nu}}^* \boldsymbol{o}\times^T + \boldsymbol{o}\times \mathbf{K}_{\boldsymbol{f\kappa}}^* + \mathbf{K}_{\boldsymbol{m\kappa}}^* \end{array} \right] \left\{ \begin{array}{c} \boldsymbol{\nu} \\ \boldsymbol{\kappa} \end{array} \right\} \\ &= \left[\begin{array}{cc} \mathbf{K}_{\boldsymbol{f\nu}} & \mathbf{K}_{\boldsymbol{f\kappa}} \\ \mathbf{K}_{\boldsymbol{m\nu}} & \mathbf{K}_{\boldsymbol{m\kappa}} \end{array} \right] \left\{ \begin{array}{c} \boldsymbol{\nu} \\ \boldsymbol{\kappa} \end{array} \right\}^* . \end{aligned} \tag{8.43}$$

It might be tempting to find what offset and rotation allows to decouple the force and moment constitutive properties. This can be sought by defining a generic transformation

$$\left\{ \begin{array}{c} \boldsymbol{f} \\ \boldsymbol{m} \end{array} \right\} = \left[\begin{array}{cc} \mathbf{I} & \mathbf{0} \\ \mathbf{T} & \mathbf{I} \end{array} \right] \left\{ \begin{array}{c} \boldsymbol{f} \\ \boldsymbol{m} \end{array} \right\}^\dagger , \tag{8.44}$$

such that

$$\begin{aligned}\left\{\begin{array}{c}\boldsymbol{f}\\ \boldsymbol{m}\end{array}\right\} &= \left[\begin{array}{cc}\mathbf{I} & \mathbf{0}\\ \mathbf{T} & \mathbf{I}\end{array}\right]\left[\begin{array}{cc}\mathbf{K}_{\boldsymbol{f\nu}} & \mathbf{K}_{\boldsymbol{f\kappa}}\\ \mathbf{K}_{\boldsymbol{m\nu}} & \mathbf{K}_{\boldsymbol{m\kappa}}\end{array}\right]^{\dagger}\left[\begin{array}{cc}\mathbf{I} & \mathbf{T}^T\\ \mathbf{0} & \mathbf{I}\end{array}\right]\left\{\begin{array}{c}\boldsymbol{\nu}\\ \boldsymbol{\kappa}\end{array}\right\}\\ &= \left[\begin{array}{cc}\mathbf{K}^{\dagger}_{\boldsymbol{f\nu}} & \mathbf{K}^{\dagger}_{\boldsymbol{f\nu}}\mathbf{T}^T + \mathbf{K}^{\dagger}_{\boldsymbol{f\kappa}}\\ \mathbf{T}\mathbf{K}^{\dagger}_{\boldsymbol{f\nu}} + \mathbf{K}^{\dagger}_{\boldsymbol{m\nu}} & \mathbf{T}\mathbf{K}^{\dagger}_{\boldsymbol{f\nu}}\mathbf{T}^T + \mathbf{K}^{\dagger}_{\boldsymbol{m\nu}}\mathbf{T}^T + \mathbf{T}\mathbf{K}^{\dagger}_{\boldsymbol{f\kappa}} + \mathbf{K}^{\dagger}_{\boldsymbol{m\kappa}}\end{array}\right]\left\{\begin{array}{c}\boldsymbol{\nu}\\ \boldsymbol{\kappa}\end{array}\right\}.\end{aligned} \tag{8.45}$$

The expected decoupling results from

$$\mathbf{T} = -\mathbf{K}^{\dagger}_{\boldsymbol{m\nu}}\left(\mathbf{K}^{\dagger}_{\boldsymbol{f\nu}}\right)^{-1}. \tag{8.46}$$

However, the resulting transformation $\mathbf{T}$ is not guaranteed to have the skew-symmetric structure of $\boldsymbol{o}\times$, thus the decoupling may not be reducible to an offset.

The reverse transformation is relatively straightforward, after noticing that, according to Eq. (8.41),

$$\begin{aligned}\left\{\begin{array}{c}\boldsymbol{f}\\ \boldsymbol{m}\end{array}\right\}^{*} &= \left[\begin{array}{cc}\mathbf{I} & \mathbf{0}\\ \boldsymbol{o}\times & \mathbf{I}\end{array}\right]^{-1}\left\{\begin{array}{c}\boldsymbol{f}\\ \boldsymbol{m}\end{array}\right\}\\ &= \left[\begin{array}{cc}\mathbf{I} & \mathbf{0}\\ -\boldsymbol{o}\times & \mathbf{I}\end{array}\right]\left\{\begin{array}{c}\boldsymbol{f}\\ \boldsymbol{m}\end{array}\right\}.\end{aligned} \tag{8.47}$$

So, as expected, it is sufficient to revert the sign of the offset to revert the transformation. The signs of the constitutive relationship change accordingly.

Locking Correction for Two-Node Beam

The three-node finite volume element has been implemented first, and uses conventional polynomial parabolic interpolation of the nodal displacements and orientations; the two-node finite volume element has been introduced later. This latter element presents some shear-locking, which, for linear elastic constitutive laws, may be overcome by correcting the section stiffness matrix in a relatively straightforward form:

$$\hat{\boldsymbol{K}} = \left(\boldsymbol{F} + \frac{L^2}{12}\boldsymbol{T}\boldsymbol{F}\boldsymbol{T}^T\right)^{-1}, \tag{8.48}$$

where $\boldsymbol{F} = \boldsymbol{K}^{-1}$ is the compliance matrix of the section, L is the length of the beam, i.e. the distance between the two reference points obtained by adding the optional offset to the nodes, and

$$\boldsymbol{T} = \left[\begin{array}{cc}\mathbf{0} & \boldsymbol{e}_x\times\\ \mathbf{0} & \mathbf{0}\end{array}\right]$$

is the "arm" matrix that appears in the differential equilibrium equation

$$\boldsymbol{\vartheta}_{/x} - \boldsymbol{T}^T\boldsymbol{\vartheta} + \boldsymbol{\phi} = 0,$$

where $\boldsymbol{\vartheta} = \left[\boldsymbol{f}^T\boldsymbol{m}^T\right]^T$ are the internal forces and moments, while $\boldsymbol{\phi}$ are the external forces and moments per unit span.

There are no provisions to automatically apply the correction when defining the constitutive law of the section. The two-node beam has been re-implemented using a helicoidal interpolation of the nodal positions and orientations, to improve its capability to undergo large displacements and relative rotations. It is activated by using the keyword `hbeam2` instead of `beam2`. However, to reduce the shear-locking effect, the stiffness properties still need to be manually corrected according to Equation (8.48). The `hbeam2` element is *experimental*, and should be used only for development purposes.

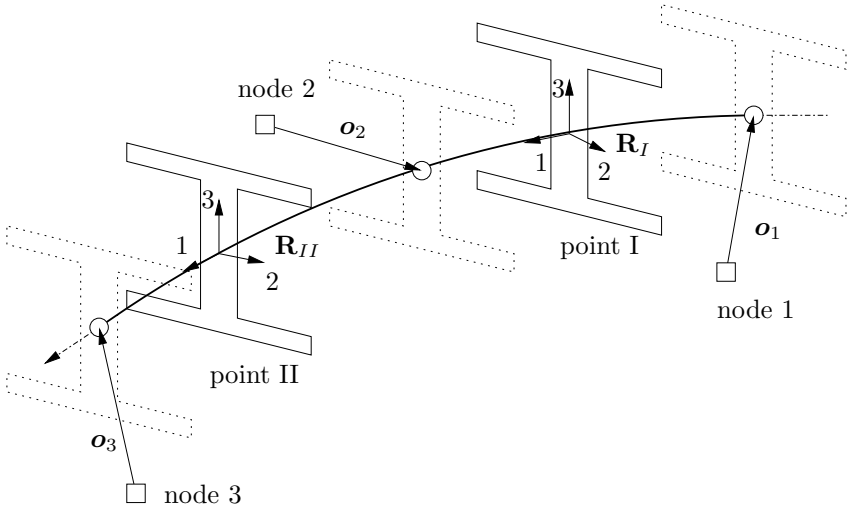


Figure 8.4: Geometry of the three-node beam element.

8.3.2 Three-node beam element

The three-node beam element is described in detail in [31]. Each of the three points the beam element connects is referred to a structural node but can have an arbitrary offset to allow high generality in locating the structural reference line of the beam. Figure 8.4 illustrates the geometry of the three-node beam element.

The finite volume formulation presented in [31] is used. As a consequence, the internal forces and moments are evaluated at two points that are at about midpoint between nodes 1 and 2, and nodes 2 and 3 (at $\xi = -1/\sqrt{3} \cong -0.57735$ and $\xi = 1/\sqrt{3} \cong 0.57735$ of a non-dimensional abscissa ξ running from $\xi = -1$ at node 1 to $\xi = 1$ at node 3).

So the constitutive properties must be supplied in these points, as well as the orientation matrices $\mathbf{R}_I$ and $\mathbf{R}_{II}$, that express the orientation of the reference system the constitutive properties are expressed in with respect to the global frame (the axial force is conventionally defined in direction 1). Any of the supported 6D constitutive laws can be supplied to define the constitutive properties of each beam section.

The traditional input format is

```
<elem_type> ::= beam3

<normal_arglist> ::=
    <node_1_label> , (Vec3) <relative_offset_1> ,
    <node_2_label> , (Vec3) <relative_offset_2> ,
    <node_3_label> , (Vec3) <relative_offset_3> ,
    (OrientationMatrix) <orientation_matrix_section_I> ,
    (ConstitutiveLaw<6D>) <constitutive_law_section_I> ,
    { same | (OrientationMatrix) <orientation_matrix_section_II> } ,
    { same | (ConstitutiveLaw<6D>) <constitutive_law_section_II> }
    [ , <custom_output> ]
```

The `relative_offset_<i>`, with `i`=1,2,3, are the vectors $\boldsymbol{o}_i$, with i=1,2,3, of Figure 8.4.

The orientation matrices `orientation_matrix_section_<j>`, with `j`=I,II, are the section orientation matrices $\mathbf{R}_j$, with j=I,II, of Figure 8.4. They represent the (absolute) orientation in which the respective

constitutive properties are expressed.

The first keyword same, alternative to the orientation_matrix_section_II, means that the same orientation defined for the first point will be used for the second point.

The second keyword same, alternative to the constitutive_law_section_II, means that the same constitutive law defined for the first point will be used for the second point.

If any of the constitutive laws is either viscous or viscoelastic, the viscoelastic variant of the beam element is used. Otherwise, the elastic variant is used.

A more complete input format is

```
<elem_type> ::= beam3

<normal_arglist> ::=
    <node_1_label> ,
        [ position , ] (Vec3) <relative_offset_1> ,
        [ orientation , (OrientationMatrix) <relative_orientation_1> , ]
    <node_2_label> ,
        [ position , ] (Vec3) <relative_offset_2> ,
        [ orientation , (OrientationMatrix) <relative_orientation_2> , ]
    <node_3_label> ,
        [ position , ] (Vec3) <relative_offset_3> ,
        [ orientation , (OrientationMatrix) <relative_orientation_3> , ]
    { (OrientationMatrix) <orientation_matrix_section_I>
        | from nodes } ,
    (ConstitutiveLaw<6D>) <constitutive_law_section_I> ,
    { same
        | (OrientationMatrix) <orientation_matrix_section_II>
        | from nodes } ,
    { same
        | (ConstitutiveLaw<6D>) <constitutive_law_section_II> }
    [ , <custom_output> ]
```

This format is a superset of the traditional one, which is extended by adding the possibility to set, for each node, the relative orientation relative_orientation_<i>, with i=1,2,3, with respect to the node itself. Such orientation can be subsequently used to interpolate the orientation matrices at the evaluation points, by providing the keyword from nodes instead of the orientation matrix. If the keyword same is used for the second evaluation point, the same method is used to compute the orientation matrix.

The custom_output optional data consists in

```
<custom_output> ::= custom output , <custom_output_flag> [ , ... ]
```

The values of custom_output_flag are defined in Section 5.3.14.

Flags add up to form the custom output request. Flags may not be repeated. Strain rates are only available from viscoelastic beams. By default, only forces are output, to preserve compatibility with the original output format. The custom output is only available in NetCDF format; see Section C.1.4.

As an example, a simple beam element, with diagonal section stiffness matrix is presented:

```
set: integer beam_label = 1000;
set: integer beam_node1 = 2001;
set: integer beam_node2 = 2002;
set: integer beam_node3 = 2003;
set: real EA = 1e6;   # N
set: real GAy = .6e6; # N
```

```
set: real GAz = .6e6; # N
set: real GJ = 1.e3;  # Nm^2
set: real EJy = 2.e3; # Nm^2
set: real EJz = 1.e4; # Nm^2
beam3: beam_label,
    beam_node1, reference, node, null,
    beam_node2, reference, node, null,
    beam_node3, reference, node, null,
    eye,
    linear elastic generic, diag,
        EA, GAy, GAz, GJ, EJy, EJz,
    same,
    same;
```

A not-so-simple beam section, where the center of axial strain and the shear center are not coincident, is illustrated below. The node offset is used to align the reference line with the shear center, and the axial strain center offset is used in the constitutive matrix:

```
set: integer beam_label = 1000;
set: integer beam_node1 = 2001;
set: integer beam_node2 = 2002;
set: integer beam_node3 = 2003;
set: real EA = 1e6;    # N
set: real GAy = .6e6;  # N
set: real GAz = .6e6;  # N
set: real GJ = 1.e3;   # Nm^2
set: real EJy = 2.e3;  # Nm^2
set: real EJz = 1.e4;  # Nm^2
set: real yas = 2.e-2; # m
set: real zas = 1.e-2; # m
set: real ysc = 4.e-2; # m
set: real zsc = 2.e-2; # m
set: real y = yas-ysc; # compute the axial strain center
set: real z = zas-zsc; # wrt/ the shear center
beam3: beam_label,
    beam_node1, reference, node, 0.,ysc,zsc,
    beam_node2, reference, node, 0.,ysc,zsc,
    beam_node3, reference, node, 0.,ysc,zsc,
    eye,
    linear elastic generic, sym,
        EA, 0.,  0.,  0., z*EA,       -y*EA,
            GAy, 0.,  0., 0.,          0.,
                 GAz, 0., 0.,          0.,
                      GJ, 0.,          0.,
                          EJy+z^2*EA, -z*y*EA,
                                       EJz+y^2*EA,
    same,
    same;
```

A piezoelectric actuator beam element is available; an arbitrary linear piezoelectric actuation matrix is required, together with the labels of the abstract nodes that represent the input signal tensions, as

follows:

```
<normal_arglist> ::=
    <node_1_label> , (Vec3) <relative_offset_1> ,
    <node_2_label> , (Vec3) <relative_offset_2> ,
    <node_3_label> , (Vec3) <relative_offset_3> ,
    (OrientationMatrix) <orientation_matrix_section_I> ,
    (ConstitutiveLaw<6D>) <constitutive_law_section_I> ,
    { same | (OrientationMatrix) <orientation_matrix_section_II> } ,
    { same | (ConstitutiveLaw<6D>) <constitutive_law_section_II> } ,
    piezoelectric actuator ,
        <electrodes_number> ,
        <abstract_node_label_list> ,
        (Mat6xN) <piezoelectric_matrix_I> ,
        { same | (Mat6xN) <piezoelectric_matrix_II> }
    [ , <custom_output> ]
```

where the abstract_node_label_list is the list of the labels of the abstract nodes that represent the electrodes.

A fully coupled piezoelectric beam element is available as well; an arbitrary linear piezoelectric actuation matrix is required, together with the labels of the abstract nodes that represent the input signal tensions, as follows:

```
<normal_arglist> ::=
    <node_1_label> , (Vec3) <relative_offset_1> ,
    <node_2_label> , (Vec3) <relative_offset_2> ,
    <node_3_label> , (Vec3) <relative_offset_3> ,
    (OrientationMatrix) <orientation_matrix_section_I> ,
    (ConstitutiveLaw<6D>) <constitutive_law_section_I> ,
    { same | (OrientationMatrix) <orientation_matrix_section_II> } ,
    { same | (ConstitutiveLaw<6D>) <constitutive_law_section_II> } ,
    piezoelectric beam ,
        <electrodes_number> ,
        <abstract_node_label_list> ,
        (Mat6xN) <piezoelectric_matrix_I> ,
        (MatNx6) <piezoelectric_charges_def_I> ,
        (MatNxN) <piezoelectric_charges_potential_I> ,
        {
            same
           |
            (Mat6xN) <piezoelectric_matrix_II>
            (MatNx6) <piezoelectric_charges_def_II> ,
            (MatNxN) <piezoelectric_charges_potential_II>
        }
    [ , <custom_output> ]
```

where the abstract_node_label_list is the list of the labels of the abstract nodes that represent the electrodes, piezoelectric_charges_def give to compute the charge per unit of length as a function of the beam generalized deformation measures, and piezoelectric_charges_potential the charge per unit of length as a function of the electrodes potential.

Private Data The following data are available:

1. `"ex"` axial strain
2. `"ey"` shear strain (local axis 2)
3. `"ez"` shear strain (local axis 3)
4. `"kx"` curvature about local axis 1 (torsional)
5. `"ky"` curvature about local axis 2 (bending)
6. `"kz"` curvature about local axis 3 (bending)
7. `"Fx"` axial force
8. `"Fy"` shear force (direction 2)
9. `"Fz"` shear force (direction 3)
10. `"Mx"` moment about local axis 1 (torsional)
11. `"My"` moment about local axis 2 (bending)
12. `"Mz"` moment about local axis 3 (bending)
13. `"Xx"` absolute position component 1
14. `"Xy"` absolute position component 2
15. `"Xz"` absolute position component 3
16. `"Phix"` absolute orientation vector component 1
17. `"Phiy"` absolute orientation vector component 2
18. `"Phiz"` absolute orientation vector component 3
19. `"Omegax"` absolute angular velocity component 1
20. `"Omegay"` absolute angular velocity component 2
21. `"Omegaz"` absolute angular velocity component 3
22. `"ePx"` axial strain rate
23. `"ePy"` shear strain rate (local axis 2)
24. `"ePz"` shear strain rate (local axis 3)
25. `"kPx"` curvature rate about local axis 1 (torsional)
26. `"kPy"` curvature rate about local axis 2 (bending)
27. `"kPz"` curvature rate about local axis 3 (bending)

Each string is prefixed by `"pI."` or `"pII."` to specify data related to either the first or the second point:

```
"pI.Fx"     # axial force at point I
"pII.kz"    # bending curvature about local axis 3 at point II
```

Private data related to point I are numbered from 1 to 27; private data related to point II are numbered from 28 to 54.

Inertia Properties

So-called "consistent inertia" is not implemented for beam elements; one needs to use body (rigid-body) elements (Section 8.4) instead. Their suggested usage is detailed here.

In principle, from the inertial point of view, the beam should be split in "chunks" as discussed earlier. Distributed inertia loads are associated with the motion of each chunk. For moderate straining, one can safely assume that the chunks remain rigid with respect to inertia loads computation.

For the sake of simplicity, it is assumed that the beam is straight, with uniform inertia properties, where the x axis is the beam axis, and y and z are principal inertia axes of the section. Moreover, it is assumed that the nodes are equally spaced (i.e. the mid-node is at midspan).

Consider the mass per unit span m, the inertia moment per unit span about the beam's axis, i_{xx}, and those about the two remaining axes, i_{yy}, i_{zz}, with $i_{xx} = i_{yy} + i_{zz}$. The length of the beam element is ℓ.

According to the theory, the beam is cut at two stations, set at $\pm\ell/(2\sqrt{3})$ from the mid-node. As a consequence, the length of the middle chunk is $\ell_{\text{mid}} = \ell/\sqrt{3}$, whereas that of the end chunks is $\ell_{\text{end}} = \ell(1 - 1/\sqrt{3})/2$.

The corresponding bodies are

```
# body associated with first end node
body: 1,
    1, # label of first end node
    m*ell_end, # mass associated with first end node
    reference, node, ell_end/2, 0., 0., # the center of mass of this chunk
                                        # is offset along the beam axis
    diag,
        i_xx*ell_end,
        i_yy*ell_end + m*ell_end^3/12, # i_yy may often be neglected
        i_zz*ell_end + m*ell_end^3/12; # i_zz may often be neglected

# body associated with mid-node
body: 2,
    2, # label of mid-node
    m*ell_mid, # mass associated with mid-node
    reference, node, null, # the center of mass of this chunk is at the node
    diag,
        i_xx*ell_mid,
        i_yy*ell_mid + m*ell_mid^3/12,
        i_zz*ell_mid + m*ell_mid^3/12;

# body associated with last end node
body: 3,
    3, # label of last end node
    m*ell_end,
    reference, node, -ell_end/2, 0., 0.,
    diag,
        i_xx*ell_end,
        i_yy*ell_end + m*ell_end^3/12,
        i_zz*ell_end + m*ell_end^3/12;
```

Often, i_{yy} and i_{zz} can be neglected, significantly when the beam element is slender enough (i.e. when $\ell\sqrt{mA/i_{yy}} \approx \ell\sqrt{mA/i_{zz}} \ll 1$).

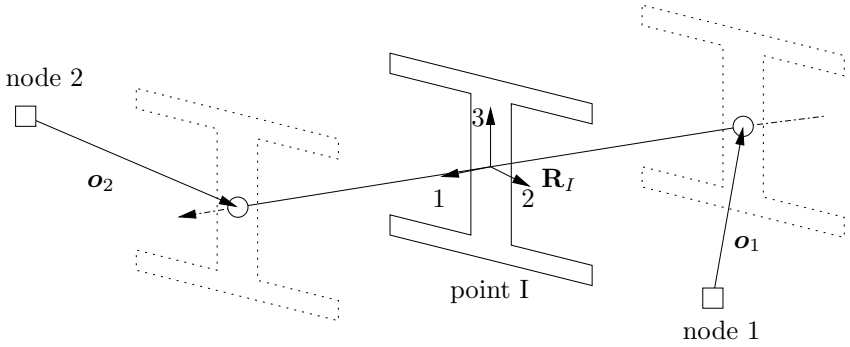


Figure 8.5: Geometry of the two-node beam element.

Moreover, the chunks do not necessarily need to be cut exactly at $\pm\ell/(2\sqrt{3})$; setting $\ell_{\text{mid}} = \ell/2$ and $\ell_{\text{end}} = \ell/4$ gives good results.

When assembling strings of beam elements, one would need to connect two `body` elements to each end node, each of them accounting from the inertial contribution of the two adjacent beam elements. Alternatively, a single `body` can be used, which accounts for the contribution of the two elements. If the two adjacent beam elements are identical (i.e. same inertia properties, same length ℓ), an end body would look like

```
body: 3,
    3, # label of last end node
    m*(2*ell_end),
    reference, node, null,
    diag,
        i_xx*(2*ell_end),
        i_yy*(2*ell_end) + m*(2*ell_end)^3/12,
        i_zz*(2*ell_end) + m*(2*ell_end)^3/12;
```

If the two adjacent beam elements differ, one can use the `condense` option to collect the inertia contributions from each beam element and let the solver do the merge. See Section 8.4 for more details.

If the beam element is significantly curved, one needs to evaluate the correct position of the center of mass, and correct the inertia tensor accordingly.

If axes y, z are not principal inertia axes, one needs to take into account the cross-coupling terms in the inertia tensor.

8.3.3 Two-node beam element

Similar considerations apply to the two-node beam. Its geometry is illustrated in Figure 8.5.

The syntax is

```
<elem_type> ::= beam2

<normal_arglist> ::=
    <node_1_label> , (Vec3) <relative_offset_1> ,
    <node_2_label> , (Vec3) <relative_offset_2> ,
    (OrientationMatrix) <orientation_matrix_section_I> ,
    (ConstitutiveLaw<6D>) <constitutive_law_section_I>
    [ , piezoelectric actuator ,
```

```
        <electrodes_number> ,
        <abstract_node_label_list> ,
        (Mat6xN) <piezoelectric_matrix_I> ]
    [ , <custom_output> ]
```

A more complete form is

```
<elem_type> ::= beam2

<normal_arglist> ::=
    <node_1_label> ,
        [ position , ] (Vec3) <relative_offset_1> ,
        [ orientation , (OrientationMatrix) <relative_orientation_1> , ]
    <node_2_label> ,
        [ position , ] (Vec3) <relative_offset_2> ,
        [ orientation , (OrientationMatrix) <relative_orientation_2> , ]
    { (OrientationMatrix) <orientation_matrix_section_I>
        | from nodes } ,
    (ConstitutiveLaw<6D>) <constitutive_law_section_I>
    [ , piezoelectric actuator ,
        <electrodes_number> ,
        <abstract_node_label_list> ,
        (Mat6xN) <piezoelectric_matrix_I> ]
    [ , <custom_output> ]
```

Private Data. The same private data indicated for the beam3 element is available (see Section 8.3.2). No prefix must be specified ("pI." is implicit).

Example. As an example, a simple beam element, with diagonal section stiffness matrix is presented:

```
set: integer beam_label = 1000;
set: integer beam_node1 = 2001;
set: integer beam_node2 = 2002;
set: real L = .4;     # m
set: real EA = 1e6;   # N
set: real GAy = .6e6; # N
set: real GAz = .6e6; # N
set: real GJ = 1.e3;  # Nm^2
set: real EJy = 2.e3; # Nm^2
set: real EJz = 1.e4; # Nm^2
beam2: beam_label,
    beam_node1, reference, node, null,
    beam_node2, reference, node, null,
    eye,
    linear elastic generic, diag,
        EA, 1./(1./GAy+L^2/12./EJz), 1./(1./GAz+L^2/12./EJy),
        GJ, EJy, EJz;
```

Note that the shear terms have been naïvely inverted to eliminate shear locking, according to Equation (8.48).

Inertia Properties

According to the theory, the beam is cut in half. As a consequence, the length of each chunk is $\ell_{\text{end}} = \ell/2$.

With reference to the symbols defined for the three-node beam element, the corresponding bodies are

```
# body associated with first end node
body: 1,
    1, # label of first end node
    m*ell_end, # mass associated with first end node
    reference, node, ell_end/2, 0., 0., # the center of mass of this chunk
                                        # is offset along the beam axis
    diag,
        i_xx*ell_end,
        i_yy*ell_end + m*ell_end^3/12, # i_yy may often be neglected
        i_zz*ell_end + m*ell_end^3/12; # i_zz may often be neglected

# body associated with last end node
body: 2,
    2, # label of last end node
    m*ell_end,
    reference, node, -ell_end/2, 0., 0.,
    diag,
        i_xx*ell_end,
        i_yy*ell_end + m*ell_end^3/12,
        i_zz*ell_end + m*ell_end^3/12;
```

Similar considerations to those made for the three-node beam element apply also in this case.

8.3.4 Output

The output related to beam elements is contained in a file with extension `.act`; for each time step, the output is written for those element it was requested. The internal forces and moments are computed from the interpolated strains along the beam by means of the constitutive law, at the evaluation points. The format is:

- column 1: the label of the beam;
- columns 2–4: the three components of the force at the first evaluation point, oriented according to the reference frame of that beam section;
- columns 5–7: the three components of the moment at the first evaluation point, oriented according to the reference frame of that beam section.

The three-node beam element generates six additional columns:

- columns 8–10: the three components of the force at the second evaluation point, oriented according to the reference frame of that beam section;
- columns 11–13: the three components of the moment at the second evaluation point, oriented according to the reference frame of that beam section.

More detailed output is allowed when using NetCDF; see Section C.1.4 for details.

8.3.5 Notes

Two-node beam elements should be used with care.

8.4 Body

The body element describes a lumped rigid body when connected to a regular, 6 degree of freedom structural node, or a point mass when connected to a rotationless, 3 degree of freedom structural node.

```
<elem_type> ::= body

<normal_arglist> ::= <node_label> ,
    { [ allow negative mass , ] <one_body>
        | [ allow negative mass , ] <one_pointmass>
        | variable mass , <one_vm_body>
        | condense , <num_masses> [ , allow negative mass ] ,
            { <one_body> | <one_pointmass> } [ , ... ] }

<one_body> ::=
    (real) <mass> ,
    (Vec3)   <relative_center_of_mass> ,
    (Mat3x3) <inertia_matrix>
    [ , orientation , (OrientationMatrix) <orientation_matrix>]

<one_pointmass> ::=
    (real) <mass>      # when <node_label> is a displacement node

<one_vm_body> ::=
    (DriveCaller)            <mass> ,
    (TplDriveCaller<Vec3>)   <relative_center_of_mass> ,
    (TplDriveCaller<Mat3x3>) <variable_mass_inertia_matrix> ,
    (TplDriveCaller<Mat3x3>) <variable_geometry_inertia_matrix>
```

If only one mass is defined, the first method should be used. Otherwise, many masses can be referred to the same element by means of the keyword condense, followed by the number of expected masses num_masses. The format of each sub-mass is the same as for the single mass input (actually, when condense is not supplied, num_masses is assumed to be 1). If allow negative mass is specified, it will be possible to use negative masses for the body. *Warning*: the negative mass feature is not officially supported and could cause unexpected behavior (for example, gravity will still act as if the mass was positive).

The inertia_matrix is always referred to the center of mass of the mass that is being added. It can be rotated locally by means of the extra orientation_matrix supplied after the (optional) keyword orientation.

Note: in many commercial finite element software, the off-diagonal elements of the inertia matrix are defined with a minus sign; for instance, NASTRAN's CONM2 lumped inertia card expects the values as indicated in Figure 8.6. However, the matrix is reconstructed as

$$\text{NASTRAN} ::= \begin{bmatrix} \texttt{M} & & & & & \\ & \texttt{M} & & \text{symmetric} & & \\ & & \texttt{M} & & & \\ & & & \texttt{I11} & & \\ & & & -\texttt{I21} & \texttt{I22} & \\ & & & -\texttt{I31} & -\texttt{I32} & \texttt{I33} \end{bmatrix}$$

see for instance *NASTRAN V70.5 Quick Reference Guide* for details.
On the contrary, MBDyn directly reads the matrix that will be used in the computation, i.e. **without the minus signs in the off-diagonal terms**, as reported below:

$$\text{MBDyn} ::= \begin{bmatrix} \texttt{i11} & \texttt{i12} & \texttt{i13} \\ & \texttt{i22} & \texttt{i23} \\ \text{sym.} & & \texttt{i33} \end{bmatrix}$$

So:

$$\begin{aligned} \texttt{i11} &= \texttt{I11} \\ \texttt{i22} &= \texttt{I22} \\ \texttt{i33} &= \texttt{I33} \\ \texttt{i12} &= -\texttt{I21} \\ \texttt{i13} &= -\texttt{I31} \\ \texttt{i23} &= -\texttt{I32} \end{aligned}$$

The `variable mass` variant requires the mass, `mass`, the location of the center of mass with respect to the node in the reference frame of the node, `relative_center_of_mass`, and the inertia matrix about the center of mass must be provided in form of *differentiable* drive callers of the specific dimension. The inertia matrix is logically split in two contributions: the first one, `variable_mass_inertia_matrix`, is related to the variation of mass; the second one, `variable_geometry_inertia_matrix`, is related to the change of geometry at fixed instantaneous value of the mass. The instantaneous inertia matrix with respect to the node, in the reference frame of the node, is computed as

$$\tilde{\mathbf{J}}_n = \texttt{variable\_mass\_inertia\_matrix} \tag{8.49}$$

$$+ \texttt{variable\_geometry\_inertia\_matrix} \tag{8.50}$$

$$+ \texttt{mass} \cdot \texttt{relative\_center\_of\_mass} \times \texttt{relative\_center\_of\_mass} \times {}^{T}. \tag{8.51}$$

See the technical manual for more details.

The inertia properties of the model can be logged and verified by means of the `inertia` keyword, as detailed in Section 8.20.3.

Private Data The following data are available:

1. `"E"` kinetic energy
2. `"V"` potential (i.e. gravitational) energy
3. `"m"` mass

```
$.......2.......3.......4.......5.......6.......7.......8.......
CONM2   EID     G       CID     M       X1      X2      X3
        I11     I21     I22     I31     I32     I33
```

Figure 8.6: NASTRAN `CONM2` card

Example.

```
set: integer NODE_LABEL = 100;
set: integer BODY_LABEL = 100;

# single mass example
body: BODY_LABEL, NODE_LABEL,
    8.,                             # mass
    reference, node, 0., 0., 0.,    # c.m. offset
    diag, 4.8, 4.8, .4;             # inertia tensor

# three masses example (equivalent to the previous one)
body: BODY_LABEL, NODE_LABEL,
    condense, 3,
    4.,                             # mass 1 (mid)
    reference, node, 0., 0., 0.,    # c.m. offset 1
    diag, .4, .4, .2,               # inertia tensor 1
    2.,                             # mass 2 (top)
    reference, node, 0., 0., 1.,    # c.m. offset 2
    diag, .2, .2, .1,               # inertia tensor 2
    2.,                             # mass 3 (bottom)
    reference, node, 0., 0., -1.,   # c.m. offset 3
    diag, .2, .2, .1;               # inertia tensor 3
```

8.5 Bulk Elements

The bulk element is intended as a sort of NASTRAN's CELAS card, that can be used to apply a stiffness term on an arbitrary degree of freedom. Extensions are planned to different kind of elements. The syntax of the bulk element is:

```
<elem_type> ::= bulk

<normal_arglist> ::= <bulk_type> , <bulk_arglist>
```

At present only the stiffness spring type is available.

8.5.1 Stiffness spring

```
<bulk_type> ::= stiffness spring

<bulk_arglist> ::=
    (NodeDof) <dof> ,
    (DriveCaller) <stiffness_drive>
```

The equation related to the desired dof of the linked node is added a contribution based on the value of the desired degree of freedom (even the derivative can be used) multiplied times the stiffness.

Note: this family of elements has been partially superseded by the genel elements, which allow more generality.

8.6 Couple

A variant of force; see Section 8.8 for details.

8.7 Electric Elements

electric elements are those elements that model electric and electronic devices, dealing with abstract degrees of freedom more than with electric ones (from the program's point of view they are exactly the same, the difference is only semantic). The true electric elements, such resistors, switches and so on, are classified as electric bulk elements. The syntax for electric elements is:

```
<normal_arglist> ::= <electric_type> , <electric_arglist>
```

The electric elements implemented at present are:

- accelerometer
- displacement
- motor
- discrete control

The syntax is described in the following.

8.7.1 Accelerometer

```
<electric_type> ::= accelerometer

<electric_arglist> ::=
    { translational | rotational } ,
    <struct_node_label> ,
    <abstract_node_label> ,
    ((Unit)Vec3) <measure_direction>
    [ , position , (Vec3) <position> ]
```

The position is optional; it is meaningless for rotational accelerometers. The measure is taken along or about direction measure_direction; the vector is internally normalized to unity.
Legacy element: accelerometer with built-in transfer function

```
<electric_type> ::= accelerometer

<electric_arglist> ::= <struct_node_label> ,
    <abstract_node_label> ,
    ((Unit)Vec3) <measure_direction> ,
    (real) <omega> ,
    (real) <tau> ,
    (real) <csi> ,
    (real) <kappa>
```

The label struct_node_label defines the node whose acceleration is being measured; the label abstract_node_label defines the abstract node that will receive the output signal. An electric node can be used as well

(?). The measure is taken along direction measure_direction; the vector is internally normalized to unity. The transfer function of the accelerometer is:

$$\frac{e_0}{a} = \texttt{kappa} \cdot \frac{\texttt{tau} \cdot s}{(1 + \texttt{tau} \cdot s)\,(1 + 2 \cdot \texttt{csi} \cdot (s/\texttt{omega}) + (s/\texttt{omega})^2)}$$

where e_0 is the output signal, a is the input (the acceleration) and s is Laplace's variable.

8.7.2 Discrete control

This element implements a discrete equation that can be used to represent the behavior of a discrete linear controller. The control matrices can be either provided, or identified during the simulation by recursive least squares.

Assume the original system consists in a generic nonlinear process that depends on a set of measures $\boldsymbol{y}$ and a set of inputs $\boldsymbol{u}$ at a finite number of time steps past the current time, t_k, namely

$$\boldsymbol{f}\left(\boldsymbol{y}_{k-i}, \boldsymbol{u}_{k-j}\right) = \boldsymbol{0}, \tag{8.52}$$

with $i = 0, p$ and $j = 0, q$. Only $\boldsymbol{y}_k$ is unknown, and thus represents the output of the process. It is assumed that $\boldsymbol{u}_k$ is known, and represents an input to the process.

This element implements a controller of the form

$$\boldsymbol{u}_{ck} = \sum_{i=1,p} \mathbf{B}_{ci} \boldsymbol{u}_{k-i} + \sum_{j=1,q} \mathbf{A}_{cj} \boldsymbol{y}_{k-j}, \tag{8.53}$$

where $\boldsymbol{u}_{ck}$ is the control input that must be applied at time t_k in addition to any exogenous input $\boldsymbol{u}_{ek}$, so that $\boldsymbol{u}_k = \boldsymbol{u}_{ek} + \boldsymbol{u}_{ck}$. The control input $\boldsymbol{u}_{ck}$ can only depend on the measures and the inputs at previous time steps.

Note that the symbols commonly used for discrete systems are here reversed, since the element is intended to compute the control signals at time t_k, $\boldsymbol{u}_{ck}$, based on the previous value of the controls, $\boldsymbol{u}_{k-i} = \boldsymbol{u}_{e(k-i)} + \boldsymbol{u}_{c(k-i)}$, and on the previous value of the motion of the original system, $\boldsymbol{y}_{k-j}$; $\boldsymbol{u}_e$ indicates a generic measured exogenous input to the system to be controlled.

The order p and q of the auto-regressive and exogenous portions of the system can differ. In detail, the order p can be zero; in that case, the system implements a *finite impulse response* function.

This element is not self-starting; it assumes that both inputs and outputs at times before the start time are zero.

The so-called "output" signals, indicated with $\boldsymbol{y}$, can be instances of either NodeDof or DriveCaller objects. The so-called "input" signals, indicated with $\boldsymbol{u}$, must be instances of NodeDof object. This implies that instances of NodeDof objects need the corresponding equations to be (at least) statically defined. In fact, the discrete control element uses the "output" NodeDof values, the $\boldsymbol{y}$, to compute the corresponding "inputs", the $\boldsymbol{u}$. The latter are substantially added as a right-hand side to the equations of the corresponding NodeDof objects. As a consequence, other elements must contribute to the left-hand side of all the NodeDof equations in order to make them defined.

Note that other elements may also contribute to the right-hand side of the "input" NodeDof object. Specifically, other abstract forces may contribute to their value. In that case, the additional forces represent the exogenous inputs. They are considered as part of the input used by the discrete control element, since the value $\boldsymbol{u}_k$ to be used in the control equation is extracted from the value of the corresponding NodeDof objects.

Figure 8.7 illustrates the behavior of the element. The typical suggested approach is illustrated later in an example.

The syntax is

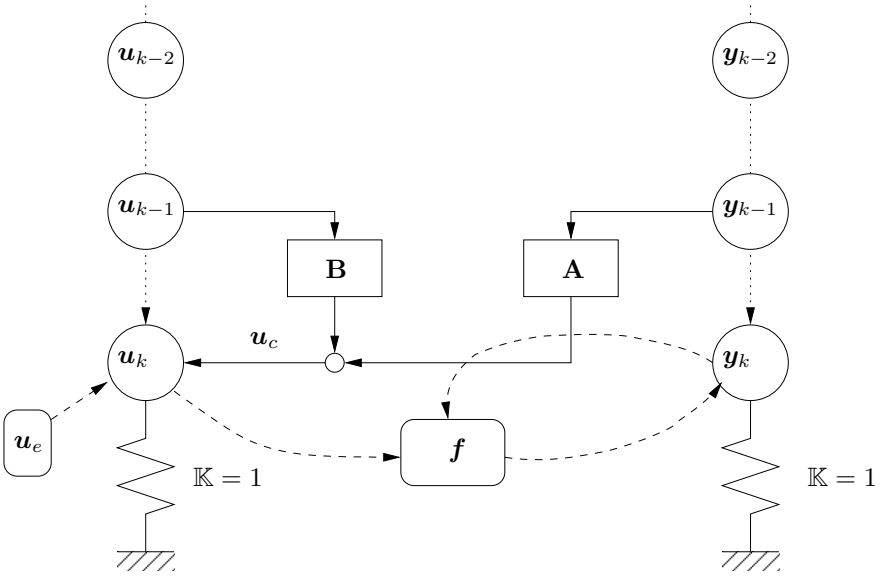


Figure 8.7: Discrete control layout.

```
<electric_type> ::= discrete control

<electric_arglist> ::= <num_outputs> , <num_inputs> ,
    <order_A> [ , fir , <order_B> ] ,
    <num_iter> ,
    <control_data> ,
    outputs ,
        (ScalarValue) <output_value>
            [ , scale , (DriveCaller) <scale> ]
        [ , ... ] ,
    inputs ,
        (NodeDof) <input_dof>
        [ , ... ]

<output_value> ::=
    { [ node dof , ] (NodeDof) <output_dof>
        | drive , (DriveCaller) <output_drive> }
```

The lists of the output and input dofs follows. The inputs do not require the `order_A` and `order_B` fields, since they are simply used to compute the control forces, and thus identify an equation. `order_B` defaults to `order_A` unless a `fir` control is chosen.

The `control_data` has the syntax:

```
<control_data> ::= <control_type> , <control_arglist>
```

At present only a simple form of control is implemented. Other types to come are system identification, both recursive and one-shot, and adaptive control, with different models and schemes, all based on Generalized Predictive Control (GPC) and Deadbeat Control. The `control_data` syntax is:

```
<control_data> ::=
    { control , " <control_matrices_file> "
```

```
        | identification , <identification_data>
        | adaptive control , <adaptive_control_data> }
```

Control

The file control_matrices_file must contain the matrices a_c, b_c of the control in plain text (as generated by Matlab, for instance):
a_{c1},
$\ldots$,
a_{cp},
b_{c1},
$\ldots$,
b_{cp}
where p is the order_A of the controller and the matrices a_c have num_inputs rows and num_outputs columns, while the matrices b_c have num_inputs rows and num_inputs columns.

Identification

```
<identification_data> ::=
    { arx | armax }
    [ , <forgetting_factor> ]
    [ , <persistent_excitation> ]
    [ , file , " <output_file_name> " ]
```

The forgetting factor is defined as

```
<forgetting_factor> ::=
    forgettingfactor ,
      { const , <d>
          | dynamic , <n1> , <n2> , <rho> , <fact> , <kref> , <klim> }
```

The default is a const forgetting factor with d = 1.

The persistent_excitation is defined as

```
<persistent_excitation> ::=
    excitation , (DriveCaller) <excitation_drive> [ , ... ]
```

where num_inputs instances of excitation_drive must be defined. By default, no persistent excitation is defined.

Adaptive control

The adaptive_control_data card is

```
<adaptive_control_data> ::=
    [ { arx | armax } , ]
    [ periodic , <periodic_factor> , ]
    { gpc ,
        <prediction_advancing_horizon> ,
        <control_advancing_horizon> ,
        <prediction_receding_horizon> ,
        [ prediction weights , <Wi> [ , ... ] , ]
        [ control weights , <Ri> [ , ... ] , ]
```

```
    (DriveCaller) <weight_drive>
| deadbeat ,
    <prediction_advancing_horizon> ,
    <control_advancing_horizon> }
[ , <forgetting_factor> ]
[ , <persistent_excitation> ]
[ , trigger , (DriveCaller) <trigger_drive> ]
[ , desired output , (DriveCaller) <output_drive> [ , ... ] ]
[ , file , " <file_name> " ]
```

The default is `arx`.

The meaning of the keywords is:

- the `periodic_factor` defaults to 0;
- if the keyword `prediction weights` is present, a number of weights `Wi` equal to

  (`prediction_advancing_horizon` − `prediction_receding_horizon`)

  must be supplied. If the keyword `control weights` is present, a number of weights `Ri` equal to `control_advancing_horizon` must be supplied. If the keywords are not defined, the corresponding weights default to 1;
- the `desired output` option requires `num_outputs` drives to be defined.

Private Data

The following data are available:

1. `"u[<idx>]"` value of the `idx`-th control output.

Examples

Consider the case of a discrete controller that computes a set of control signals $\boldsymbol{u}_k$ by adding to their value at the previous step, $\boldsymbol{u}_{k-1}$, a contribution coming from some measure $\boldsymbol{y}_{k-1}$. The control matrices are

$$\mathbf{A}_{c1} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \tag{8.54a}$$

$$\mathbf{B}_{c1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{8.54b}$$

They are defined in the control data file `discretecontrol.dat` as

```
a_11 a_12
a_21 a_22
1.0 0.0
0.0 1.0
```

The model consists in two abstract nodes for the control input `NodeDof` objects. They need to be grounded. This can be formulated by using a `spring support genel` for each `abstract` node, characterized by a unit spring coefficient, such that

$$1 \cdot x_{u1} = u_1 \tag{8.55a}$$

$$1 \cdot x_{u2} = u_2 \tag{8.55b}$$

Moreover, assume that the first measure, y_1, comes from a NodeDof object, while the second one, y_2, comes from a DriveCaller object. The equation corresponding to y_1 must be appropriately grounded as well, for example by means of yet another spring support genel. This is outside the scope of the present example, so it will assume the output nodes are defined appropriately.

The model is

```
set: integer U_1 = 1;
set: integer U_2 = 2;
set: integer Y_1 = 11;
# ...
abstract: U_1;
abstract: U_2;
# ...
genel: U_1, spring support,
    U_1, abstract, algebraic,
    linear elastic, 1.;
genel: U_2, spring support,
    U_2, abstract, algebraic,
    linear elastic, 1.;
electric: 99, discrete control,
    2,    # number of 'outputs' y
    2,    # number of 'inputs' u
    1,    # order of output matrices A (same for input matrices B)
    1,    # update every iteration
    control, "discretecontrol.dat",
    outputs,
        node dof, Y_1, abstract, algebraic, scale, 1.e-3,
        drive, sine, .5, 2*pi, 1., forever, 0, scale, 1.e-3,
    inputs,
        U_1, abstract, algebraic,
        U_2, abstract, algebraic;
```

8.7.3 Displacement

```
<electric_type> ::= displacement

<electric_arglist> ::=
    <struct_node_1_label> , (Vec3) <relative_offset_1> ,
    <struct_node_2_label> , (Vec3) <relative_offset_2> ,
    <abstract_node_label>
```

$$\boldsymbol{d} = \boldsymbol{x}_b + \boldsymbol{o}_b - \boldsymbol{x}_a - \boldsymbol{o}_a \tag{8.56a}$$

$$x = \sqrt{\boldsymbol{d}^T \boldsymbol{d}} \tag{8.56b}$$

The value x is added to the right-hand side of the equation of the abstract node.

8.7.4 Motor

```
<electric_type> ::= motor
```

```
<electric_arglist> ::=
    <struct_node_1_label> ,
    <struct_node_2_label> ,
    ((Unit)Vec3) <direction_relative_to_node_1> ,
    <abstract_node_1_label> ,
    <abstract_node_2_label> ,
    (real) <dG>,
    (real) <dL>,
    (real) <dR>
    [ , initial current , <i_0> ]
```

This element implements a simple DC motor, whose equations are

$$\omega = \boldsymbol{e}^T (\boldsymbol{\omega}_2 - \boldsymbol{\omega}_1) \tag{8.57a}$$
$$V = V_2 - V_1 \tag{8.57b}$$
$$V = \mathtt{dL}\frac{\mathrm{d}i}{\mathrm{d}t} + \mathtt{dR}\, i + \mathtt{dG}\, \omega \tag{8.57c}$$
$$\boldsymbol{c}_1 = -\boldsymbol{e}\, \mathtt{dG}\, i \tag{8.57d}$$
$$\boldsymbol{c}_2 = \boldsymbol{e}\, \mathtt{dG}\, i \tag{8.57e}$$
$$i_1 = -i \tag{8.57f}$$
$$i_2 = i \tag{8.57g}$$

where dL is the inductance, dR is the resistance, and dG is the motor torque/back-EMF constant.

The motor applies an internal torque, $C = \mathtt{dG}\, i$, about direction $\boldsymbol{e} =$ direction_relative_to_node_1, with opposite sign to both the structural nodes it is connected to, namely struct_node_1_label and struct_node_2_label. The input vector that represents direction $\boldsymbol{e}$ is internally normalized to unity.

It also contributes with a current i to both the electrical nodes it is connected to, the abstract nodes abstract_node_1_label and abstract_node_2_label.

The element assumes that the only allowed relative motion between nodes 1 and 2 is a rotation about direction $\boldsymbol{e}$, so appropriate joints (e.g. a total joint, a revolute hinge, or revolute pin) must be in place.

8.8 Force

The force element is a general means to introduce a right-hand side to the equations, i.e. an explicit contribution to the equations. There is a basic distinction between abstract and structural forces: abstract forces apply to arbitrary equations, while structural forces (and couples) are specific to structural nodes and have a spatial characterization. Essentially, structural forces have three components that may depend on arbitrary parameters (usually the simulated time), and a location in space. Structural couples have three parameter-dependent components. The syntax of the force element is:

```
<elem_type> ::= { force | couple }

<normal_arglist> ::= <force_type> , <force_arglist>

<force_type> ::=
    { abstract [ internal ]                                  # force
        | { absolute | follower | total } [ internal ]       # force, couple
        | modal                                              # force
        | external { structural | modal } [ mapping ] }      # force
```

The abstract force_type applies to generic equations;

The structural force types (absolute, follower, total) only apply to equilibrium equations related to structural nodes. The couple element can only be structural. It is discussed in this section because of its input syntax commonality with the structural force elements.

The modal force_type applies to equations related to a modal element.

The external structural and the external modal types are essentially used to provide an easy to use interface for coupling with external software in a loose or tight manner.

8.8.1 Output

The output is discussed according to the types of forces. The label of the element is output first in all the cases.

8.8.2 Abstract force

```
<elem_type> ::= force

<force_type> ::= abstract

<force_arglist> ::=
    (NodeDof)     <dof> ,
    (DriveCaller) <force_magnitude>
```

the dof field is a normal NodeDof; no order is required since the force simply applies to the equation related to the node, regardless of the order.

Output

The format is:

- the label of the element;
- the label of the node[1] the force is applied to;
- the value of the force.

8.8.3 Abstract reaction force

```
<elem_type> ::= force

<force_type> ::= abstract internal

<force_arglist> ::=
    (NodeDof)     <dof_1> ,
    (NodeDof)     <dof_2> ,
    (DriveCaller) <force_magnitude>
```

the dof_1 and dof_2 fields are normal NodeDof but no order is required since the force simply applies to the equations related to the nodes, regardless of the order, with opposite magnitudes.

[1] Since the abstract force type can connect to NodeDof, information provided here can be partial if the NodeDof actually represents a component of a node with more than one degree of freedom.

Output

The format is:

- the label of the element;
- the label of the first node[2] the force is applied to;
- the value of the force applied to the first node;
- the label of the second node[3] the force is applied to;
- the value of the force applied to the second node (opposite to that applied to the first node).

8.8.4 Structural force

```
<elem_type> ::= force

<force_type> ::= { absolute | follower }

<force_arglist> ::=
    <node_label> ,
    position , (Vec3) <relative_arm> ,
    (TplDriveCaller<Vec3>) <force_value>

<force_type> ::= total

<force_arglist> ::=
    <node_label>
    [ , position , (Vec3) <relative_arm> ]
    [ , force orientation , (Mat3x3) <force_orientation> ]
    [ , moment orientation , (Mat3x3) <moment_orientation> ]
    [ , force , (TplDriveCaller<Vec3>) <force_value> ]
    [ , moment , (TplDriveCaller<Vec3>) <moment_value> ]
```

The vector `relative_arm` defines the offset with respect to the node of the point where the force is applied. The drive `force_value` defines the value of the force as a function of time. The `total` force is intrinsically follower. See also the `total joint` (Section 8.12.48).

The force and the moment applied to node `node_label` are thus

- `absolute` force element:

$$\boldsymbol{f} = \texttt{force\_value} \tag{8.58a}$$
$$\boldsymbol{m} = (\mathbf{R} \cdot \texttt{relative\_arm}) \times \texttt{force\_value}; \tag{8.58b}$$

- `follower` force element:

$$\boldsymbol{f} = \mathbf{R}\,\texttt{force\_value} \tag{8.59a}$$
$$\boldsymbol{m} = \mathbf{R}\,(\texttt{relative\_arm} \times \texttt{force\_value})\,; \tag{8.59b}$$

[2] See footnote 1.
[3] See footnote 1.

- `total` force element:

$$\boldsymbol{f} = \mathbf{R}\,\texttt{force\_orientation} \cdot \texttt{force\_value} \tag{8.60a}$$

$$\boldsymbol{m} = \mathbf{R}\,(\texttt{relative\_arm} \times (\texttt{force\_orientation} \cdot \texttt{force\_value}) + \texttt{moment\_orientationmoment\_value})\,; \tag{8.60b}$$

$\mathbf{R}$ is the orientation matrix of the node `node_label`.

Example.

```
force: 10, absolute,
    1000,
        position, .5, 0., 0.,
    1., 0., 0.,
        const, 25.;
force: 20, follower,
    2000,
        position, null,
    component,
        const, 0.,
        ramp, 10., 0., 1., 0.,
        file, 5, 2;
force: 30, total,
    2000,
        position, .5, 0., 0.,
        force orientation, eye,
        moment orientation, eye,
    1., 0., 0.,
        const, 25.,
    component,
        const, 0.,
        ramp, 10., 0., 1., 0.,
        file, 5, 2;
```

Known issues. The direction of a force (a `TplDriveCaller<Vec3>`) by default is interpreted in the reference frame of the node the force acts upon. When using the `single` style of `TplDriveCaller<Vec3>`, one can refer to another reference system (the direction specified by the user is automatically transformed from that reference frame to the one of the node). However, this causes an ambiguity when the default `single` specifier is omitted, rather than explicitly written. A reference system different from the default one can be specified using the syntax

```
single , reference , <reference_frame> ,
        (Vec3) <force_value>
```

Note that

```
1., 0., 0.                          # equivalent to single, 1., 0., 0.
```

and

```
single, 1., 0., 0.                  # equivalent to 1., 0., 0.
```

are equivalent. However,

```
reference, global, 1., 0., 0.            # incorrect!
```

is incorrect; the correct form is

```
single, reference, global, 1., 0., 0.    # correct
```

Currently, a reference frame different from that of the node cannot be specified for other styles of template drive caller (e.g. the component style).

Output

The format is:

- the label of the element;
- the label of the node the force is applied to;
- the three components of the force;
- the arm of the force, in the global frame (i.e. referred to point $\{0, 0, 0\}$ and oriented as the global frame)

8.8.5 Structural internal force

```
<elem_type> ::= force

<force_type> ::= { absolute | follower } internal

<force_arglist> ::=
    <node_1_label> ,
    position , (Vec3) <relative_arm_1> ,
    <node_2_label> ,
    position , (Vec3) <relative_arm_2> ,
    (TplDriveCaller<Vec3>) <force_value>

<force_type> ::= total internal

<force_arglist> ::=
    <node_1_label> ,
    [ position , (Vec3) <relative_arm_1> , ]
    [ force orientation , (Mat3x3) <force_orientation_1> , ]
    [ moment orientation , (Mat3x3) <moment_orientation_1> , ]
    <node_2_label>
    [ , position , (Vec3) <relative_arm_2> ]
    [ , force orientation , (Mat3x3) <force_orientation_2> ]
    [ , moment orientation , (Mat3x3) <moment_orientation_2> ]
    [ , force , (TplDriveCaller<Vec3>) <force_value> ]
    [ , moment , (TplDriveCaller<Vec3>) <moment_value> ]
```

The format is basically identical to the previous case, except for the definition of the second node.

Output

The format is:

- the label of the element;
- the label of the first node the force is applied to;
- the three components of the force applied to the first node;
- the arm of the force with respect to the first node, in the global frame (i.e. referred to point $\{0, 0, 0\}$ and oriented as the global frame).
- the label of the second node the force is applied to;
- the three components of the force applied to the second node (opposite to that applied to the first node);
- the arm of the force with respect to the second node, in the global frame (i.e. referred to point $\{0, 0, 0\}$ and oriented as the global frame).

Example.

```
# constant structural force
force: 1, absolute,
    10, position, null,
    0., 0., 1.,
        const, 100.;

# constant structural internal force
force: 1, absolute internal,
    10, position, null,
    20, position, null,
    0.,0.,1.,
        const, 100.;
```

8.8.6 Structural couple

The structural couple is defined by using `couple` as `elem_type`.

```
<elem_type> ::= couple

<force_type> ::= { absolute | follower }

<force_arglist> ::=
    <node_label> ,
    [ position , (Vec3) <relative_arm> , ]
    (TplDriveCaller<Vec3>) <couple_value>
```

The vector `relative_arm` defines the offset with respect to the node of the point where the couple is applied. It is not used in the analysis (since it makes no sense), but it can be optionally provided for future reference (e.g. for the visualization of the couple). The drive `couple_value` defines the value of the couple as a function of time.

The moment applied to node `node_label` is thus

- `absolute` couple element:

$$\boldsymbol{m} = \texttt{couple\_value} \tag{8.61}$$

- `follower` couple element:

$$\boldsymbol{m} = \mathbf{R} \cdot \texttt{couple\_value} \tag{8.62}$$

$\mathbf{R}$ is the orientation matrix of the node `node_label`.

Output

The format is:

- the label of the element;
- the label of the node the couple is applied to;
- the three components of the couple.

8.8.7 Structural internal couple

The structural couple is defined by using `couple` as `elem_type`.

```
<elem_type> ::= couple

<force_type> ::= { absolute | follower } internal

<force_arglist> ::=
    <node_1_label> ,
    [ position , (Vec3) <relative_arm_1> , ]
    <node_2_label> ,
    [ position , (Vec3) <relative_arm_2> , ]
    (TplDriveCaller<Vec3>) <couple_value>
```

The format is basically identical to the previous case, except for the definition of the second node.

Output

The format is:

- the label of the element;
- the label of the first node the couple is applied to;
- the three components of the couple applied to the first node;
- the label of the second node the couple is applied to;
- the three components of the couple applied to the second node (opposite to that applied to the first node);

Note: by using a `dof`, a `node` or an `element` drive, a simple feedback control can be easily implemented. Note however that the dependence of a force on some internal state does not result in adding the corresponding contribution to the Jacobian matrix: the force remains explicit. Their improper use may result in missing convergence, or singularity of the Jacobian matrix of the problem.

8.8.8 Modal

This element allows to define a set of generalized forces acting on the equations related to the generalized variables of a `modal` joint element.

```
<elem_type> ::= force

<force_type> ::= modal

<force_arglist> ::=
    <modal_label> ,
        [ list , <number_of_modes> , <mode1> [ , ... ] , ]
    (DriveCaller) <force1>
        [ , resultant , (Vec3) <f> , (Vec3) <m> ]
    [ , ... ]
```

It requires the label of the modal joint it is connected to, `modal_label`. If the optional keyword `list` is given, the number of excited modes `number_of_modes` and the list of the excited modes is expected, otherwise it assumes all modes of the `modal_label` modal joint will be excited.

A list of `drive callers` (see Section 2.6). that provide the excitation value for each mode is required last. If the keyword `list` was given, the drive callers must be `number_of_modes`, otherwise they must be as many as the modes of the excited modal joint element.

If the `modal` joint element is connected to a modal node instead of being clamped to the ground, the optional keyword `resultant` is allowed for each mode drive caller. The keyword `resultant` is followed by two 3×1 vectors containing the resultant force and moment associated with the related mode.

If the keyword `list` was given, the mode numbers must be the actual mode numbers defined in the FEM model.

Example. In this example, all the modes available in the FEM data file are used by the modal joint, and all the modes are excited by the modal force

```
joint: MODAL_JOINT, modal, MODAL_NODE,
    5,              # number of modes
    from file,
    "model.fem",
    0;              # no interface nodes
force: MODAL_FORCE, modal, MODAL_JOINT,
    const, 5.,
    const, 10.,
    const, 1.,
    const, -1.,
    const, -3.;
```

In this example, only three modes of the same FEM data file of the previous example are used by the modal joint; only the last mode is excited by the modal force

```
joint: MODAL_JOINT, modal, MODAL_NODE,
    3,              # number of modes
        list, 1, 3, 5,
    from file,
    "model.fem",
    0;              # no interface nodes
```

```
force: MODAL_FORCE, modal, MODAL_JOINT,
    list, 1,       # only one mode is excited
        5,         # it is mode #5 in "model.fem"
    const, -3.;
```

Note that the mode is indicated by its number in the FEM data file.

8.8.9 External forces

External forces are elements that allow to communicate with an external software that computes forces based on information on the kinematics of the model.

Different communication schemes are supported. By default, communication occurs via files. Currently supported communication schemes are

```
<external_force_communicator> ::= { <file> | <edge> | <socket> }

<file> ::=                              # the default
    " <input_file_name> " [ , unlink ] ,
    " <output_file_name> " [ , no clobber ]
    [ , <common_parameters> ]

<edge> ::= edge ,
    " <flag_file_name> " ,
    " <data_file_name> "
    [ , <common_parameters> ]

<socket> ::= socket ,
    [ create , { yes | no } , ]
    { path , <path> | port , <port> [ , host , <host> ] }
    [ , <common_parameters> ]

<common_parameters> ::= <common_parameter> [ , ... ]

<common_parameter> ::=
    { sleep time , <sleep_time>
        | precision , { default | <digits> }   # not valid for socket
        | coupling , { staggered | loose | tight | <coupling_steps> }
        | send after predict , { yes | no } }
```

File communicator

`input_file_name` is the name of the file MBDyn expects to find with the values of the force. The content depends on the purpose it is used for. The optional keyword `unlink` indicates that MBDyn is supposed to unlink the file as soon as it is ready to read another one, as a primitive form of inter-process communication.

`output_file_name` is the name of the file MBDyn creates each time it intends to communicate the kinematics of the model to the external software. The contents depend on the purpose it is used for. The option `no clobber` indicates that MBDyn is supposed to wait until the external software removed the file before generating a new one, as a primitive form of inter-process communication.

EDGE communicator

This communicator has been developed in cooperation with Luca Cavagna, to support communication with FOI's EDGE.

The file `flag_file_name` is used to synchronize the communication between the processes. The same file is truncated and reused by both processes. The syntax of the file changed since EDGE 5.1; this change is reflected from MBDyn 1.3.16 on. The file contains a two lines header, followed by a line containing a single digit. The header is

```
UPDATE,N,0,0,1
FLAG,I,1,1,0
```

The digit indicates the command. Its meaning is:

0 : external software is initializing; MBDyn must wait;

1 : external software is busy; MBDyn must wait;

2 : external software is ready to read kinematics; MBDyn can start reading forces and writing kinematics;

3 : MBDyn finished writing kinematics; external software can start reading kinematics and writing forces (written by MBDyn when data file is ready);

4 : external software converged; MBDyn should go to convergence, advance to the next time step, and start writing new step kinematics;

5 : external software decided to quit; MBDyn should stop communicating.

So far, there are no provisions for re-executing a time step, nor for communicating either the time or the time step, so adaptive time stepping cannot be implemented.

`data_file_name` is the name of the file used to actually send data between the two processes. Its contents depend on the purpose it is used for, as described in the subsequent sections.

Socket communicator

The optional parameter `create`, when set to `yes`, indicates that MBDyn will create the socket, and the peer will have to connect to it. Otherwise, when set to `no`, it indicates that MBDyn will try to connect to an already existing socket created by the peer. Connecting to a peer is attempted while reading the input file. Sockets are created when the input file has been read. MBDyn waits until all sockets have been connected to by peers before the simulation is started.

MBDyn supports local (unix) sockets, defined using the `path` parameter, and inet sockets, defined using the `port` parameter. When `create` is set to `yes`, the optional keyword `host` allows to define what interface MBDyn will listen on. When `create` is set to `no`, the optional keyword `host` indicates what host to connect to. It defaults to 'localhost'.

Common parameters

The optional parameter `sleep_time` determines how long MBDyn is supposed to sleep while waiting for a new input file to appear or for an old output file to disappear.

The optional parameter `precision` determines the precision used in writing the output file; the default is 16 digits. This is only meaningful for communicators based on textual files.

The optional parameter `coupling` determines whether the coupling will be `staggered` (communicate each time step, with one step delay), `loose` (communicate each time step, no delay, the default) or `tight`

(communicate each iteration); otherwise, if `coupling_steps` is provided, the communication occurs every `coupling_steps`.

The optional parameter `send after predict` allows to indicate whether MBDyn must send the predicted motion or not when playing a tight coupling loop.

Staggered Coupling. When the coupling is `staggered`, the communication pattern is:

1. MBDyn receives a set of forces sent by the external peer the first time the residual is assembled within time step k; the external peer generated that set of forces based on the kinematics at step $k-1$;
2. MBDyn sends a set of kinematic parameters related to step k after convergence.

As a consequence, MBDyn solves the kinematics at the current time step, k, using forces evaluated for the kinematics at the previous time step, $k-1$. Something like

$$\boldsymbol{f}\left(\boldsymbol{y}_k, \dot{\boldsymbol{y}}_k, \boldsymbol{u}_{k-1}\right) = \boldsymbol{0}. \tag{8.63}$$

Note: not implemented yet.

Loose Coupling. When the coupling is `loose`, the communication pattern is:

1. MBDyn sends a set of kinematic parameters related to step k after predict.
2. MBDyn receives a set of forces sent by the external peer the first time the residual is assembled within time step k; the external peer generated that set of forces based on the predicted kinematics at step k;

As a consequence, MBDyn solves the kinematics at the current time step, k, using forces evaluated for the predicted kinematics at the current time step, k. Something like

$$\boldsymbol{f}\left(\boldsymbol{y}_k, \dot{\boldsymbol{y}}_k, \boldsymbol{u}_k^{(0)}\right) = \boldsymbol{0}. \tag{8.64}$$

Tight Coupling. When the coupling is `tight`, the communication pattern is:

1. MBDyn sends the predicted kinematics for step k during the "after predict" phase (unless `send after predict` is set to `no`);
2. MBDyn receives a set of forces sent by the external peer each time the residual is assembled; those forces are computed based on the kinematics at iteration j
3. as soon as, while reading the forces, MBDyn is informed that the external peer converged, it continues iterating until convergence using the last set of forces it read.

As a consequence, MBDyn solves the kinematics at the current time step, k, at iteration j, using forces evaluated for the kinematics at the same time step, k, but at iteration $j-1$. Something like

$$\boldsymbol{f}\left(\boldsymbol{y}_k^{(j)}, \dot{\boldsymbol{y}}_k^{(j)}, \boldsymbol{u}_k^{(j-1)}\right) = \boldsymbol{0}. \tag{8.65}$$

8.8.10 External Structural

This element allows to communicate with an external software that computes forces applied to a pool of nodes and may depend on the kinematics of those nodes. Communication occurs by means of the communicators illustrated earlier.

```
<elem_type> ::= force

<force_type> ::= external structural

<force_arglist> ::= <external_force_communicator> ,
    [ reference node , <ref_node_label> , ]
    [ { labels , { yes | no }
            | sorted , { yes | no }
            | orientation ,
                { none | orientation matrix | orientation vector | euler 123 }
            | accelerations , { yes | no }
            | use reference node forces , { yes | no }
                [ , rotate reference node forces , { yes | no } ] }
        [ , ... ] , ]
    <num_nodes> ,
        <node_label> [ , offset , (Vec3) <offset> ]
        [ , ... ]
    [ , echo , <echo_file_name> ]
```

- `use reference node forces` only meaningful when `reference node` is used. It assumes the external solver is sending the forces and moments related to the reference node. They correspond to the rigid-body forces and moments applied to the whole system. As such, the forces and moments applied to each node are removed accordingly. If it is set to `no`, reference node forces and moments will be ignored.
- The orientation style `none` implies that only positions, velocities and accelerations will be output (the latter only if `accelerations` is set to `yes`).
- `echo` causes the output of the reference configuration to be produced in a file called `echo_file_name`. If the file exists, it is overwritten, if allowed by file system permissions. The format is that of the communicator in stream form.

File communicator

The forces are read from the input file in textual form, each line formatted as follows:

- a label (if `labels` is set to `yes`);
- three components of force in the global frame;
- three components of moment in the global frame, referred to the node as the pole.

The label indicates what node the force and the moment apply to; each force is applied in a point that may be optionally offset from the corresponding node location according to `offset`. If optional keyword `sorted` is set to `no`, the forces might be read in arbitrary order, so they need to be recognized by the label. The option `sorted` is only meaningful when `labels` is set to `yes`.

The kinematics are written to the output file in textual form, each line formatted as follows:

- a label (if `labels` is set to `yes`);
- the position in the global frame;
- the orientation of the node with respect to the global frame:
  - if `orientation` is set to `orientation matrix`, the orientation matrix in the global frame, row-oriented: R_{11}, R_{12}, R_{13}, R_{21}, ..., R_{32}, R_{33};
  - if `orientation` is set to `orientation vector`, the orientation vector's components;
  - if `orientation` is set to `euler 123`, the three Euler angles, in degrees, according to the 1,2,3 convention;
- the velocity with respect to the global frame;
- the angular velocity with respect to the global frame.

If the optional keyword `accelerations` is set to `yes`, at the end of each line the following optional fields will appear:

- the acceleration with respect to the global frame;
- the angular acceleration with respect to the global frame.

The label indicates what node the kinematics is related to; the position and the velocity refer to a point that may be optionally offset from the node location according to `offset`.

If a `reference node` was given, the first record is related to the reference node itself. The following records contain forces and moments in the reference frame of the reference node.

EDGE communicator

TBD

Socket communicator

TBD

Example. The following defines an `external structural` force with the file communicator

```
set: integer FORCE_LABEL = 1000;
set: integer NODE_1 = 10;
set: integer NODE_2 = 20;
force: FORCE_LABEL, external structural,
    "IN.dat", unlink,
    "OUT.dat", no clobber,
    sleep time, 10,
    precision, 12,
    coupling, loose,
    2, # number of nodes
        NODE_1,
        NODE_2, offset, 0.,0.,.5;
```

Output

An external structural element writes one line for each connected node at each time step in the `.frc` file. Each line contains:

- in column 1 the label of the element and that of the corresponding node; the format of this field is `element_label@node_label`;
- in columns 2–4 the three components of the force;
- in columns 5–7 the three components of the moment.

If a reference node is defined, a special line is output for the reference node, containing:

- in column 1 the label of the element and that of the corresponding node; the format of this field is `element_label#node_label`;
- in columns 2–4 the three components of the force applied to the reference node, as received from the peer;
- in columns 5–7 the three components of the moment applied to the reference node, as received from the peer.
- in columns 8–10 the three components of the force that are actually applied to the reference node, in the global reference frame;
- in columns 11–13 the three components of the moment with respect to the reference node that are actually applied to the reference node, in the global reference frame;
- in columns 14–16 the three components of the force resulting from the combination of all nodal forces, in the global reference frame;
- in columns 17–19 the three components of the moment with respect to the reference node resulting from the combination of all nodal forces and moments, in the global reference frame;

8.8.11 External structural mapping

This element allows to communicate with an external software that computes forces applied to a pool of nodes and may depend on the kinematics of those nodes through a linear mapping. Communication occurs by means of the communicators illustrated earlier.

```
<elem_type> ::= force

<force_type> ::= external structural mapping

<force_arglist> ::= <external_force_communicator> ,
    [ reference node , <ref_node_label> , ]
    [ { labels , { yes | no }
            | orientation ,
                { none | orientation matrix | orientation vector | euler 123 }
            | accelerations , { yes | no }
            | use reference node forces , { yes | no }
                [ , rotate reference node forces , { yes | no } ] }
        [ , ... ] , ]
    <num_points> ,
```

```
    <node_label> , offset [ , <point_label> ] , (Vec3) <offset>
        [ , ... ]
[ , echo , " <echo_file_name> "
    [ , { precision , <digits>
            | surface , " <surface_file_name> "
            | output , " <output_file_name> "
            | order , <order>
            | basenode , <basenode>
            | weight , { inf | (real) <weight> } }
        [ , ... ] ]
    [ , stop ] ]
[ , mapped points number , { from file | <mapped_points> } ,
    { full | sparse } mapping file , " <mapping_file_name> "
        [ , threshold , <threshold> ]
    [ , { mapped labels file , " <mapped_labels_file_name> "
        | <mapped_label> [ , ... ] } ] ]
```

A set of `num_points` points is generated first, by computing the kinematics of the points originating from the rigid-body motion of the structural nodes `node_label` according to the specified offset. Multiple `offset` blocks can appear for each `node_label`.

The `echo` keyword causes the positions of those points to be written in a file named `echo_file_name`. If the keyword `stop` is present, the simulation ends here. This is useful to generate the bulk data that is needed to compute the linear mapping matrix. The optional keyword `precision` allows to set `precision` digits in the logged data; the other optional keywords refer to values that are written verbatim to the file. In detail, `surface_file_name` is the name of the file that contains the coordinates of the peer's points, `output_file_name` is the name of the file that will contain the mapping matrix, `order` is an integer ($1 \leq$ `order` ≤ 3), `basenode` is the number of nodes for each region (`basenode` > 0), `weight` is a weighting coefficient (`weight` ≥ -2 or `inf`)

When the `reference node` keyword is present, the kinematics of the points is formulated in the reference frame of node `ref_node_label`. The forces are expected in the same reference frame. In this case, if `use reference node forces` is set to `yes`, the force and moment related to node `ref_node_label` returned by the external solver are used; if `rotate reference node forces` is set to `yes`, they are first rotated in the global reference frame.

When the `mapped points number` keyword is present, a mapping matrix is read from the file named `mapping_file_name` (same as `output_file_name` when the `echo` keyword is used). If the keyword `from file` is used, the number of `mapped_points` is computed from the number of rows of the matrix in file `mapping_file_name`. The matrix must be (3 × `mapped_points`) × (3 × `num_points`). If the `mapped points number` keyword is not present, the element behaves as if the mapping were the identity matrix, namely the position and the velocity of the points rigidly offset from MBDyn's nodes are directly passed to the peer solver. See the `modal` variant for a description of the mapping file format.

The labels of the original points, indicated as `point_label`, and the labels of the mapped points, either contained in the `mapped_labels_file_name`, or directly listed as `mapped_label`, must be provided when `labels` is set to `yes`.

Mapping. The mapping consists in a constant matrix that allows to compute the position and the velocity of the mapped points (subscript 'peer') as functions of a set of points rigidly offset from MBDyn's nodes (subscript 'mbdyn'),

$$\boldsymbol{x}_{\text{peer}} = \mathbf{H}\boldsymbol{x}_{\text{mbdyn}} \tag{8.66a}$$

$$\dot{\boldsymbol{x}}_{\text{peer}} = \mathbf{H}\dot{\boldsymbol{x}}_{\text{mbdyn}} \tag{8.66b}$$

The same matrix is used to map back the forces onto MBDyn's nodes based on the preservation of the work done in the two domains,

$$\delta \boldsymbol{x}_{\text{mbdyn}}^T \boldsymbol{f}_{\text{mbdyn}} = \delta \boldsymbol{x}_{\text{peer}}^T \boldsymbol{f}_{\text{peer}} = \delta \boldsymbol{x}_{\text{mbdyn}}^T \mathbf{H}^T \boldsymbol{f}_{\text{peer}} \tag{8.67}$$

which implies

$$\boldsymbol{f}_{\text{mbdyn}} = \mathbf{H}^T \boldsymbol{f}_{\text{peer}} \tag{8.68}$$

To compute matrix **H** one needs to:

1. gather the coordinates of the set of points used by the peer process (subscript 'peer' in the formulas above) in a file, formatted as

   ```
   <num_points>
   <x_1> <y_1> <z_1>
   ...
   <x_n> <y_n> <z_n>
   ```

   If `reference node` is defined, the points need to be expressed in a reference frame coincident with that of node `ref_node_label`.

2. execute MBDyn's input file after uncommenting the `echo` line in the definition of the external structural mapping force element, terminated by the `stop` keyword, to generate the `echo_file_name` file with MBDyn's points coordinates (subscript 'mbdyn' in the formulas above).

3. start `octave`

4. make sure the folder `contrib/MLS/` is in octave's path

   ```
   addpath('/path/to/contrib/MLS')
   ```

5. execute the `create_mls_interface` function

   ```
   create_mls_interface('echo_file_name');
   ```

   This generates the mapping matrix **H**, using the files `echo_file_name` and `surface_file_name` (the latter is written in `echo_file_name` by the `surface` keyword). The matrix is stored in the file `output_file_name`. The `sparse` storage is recommended, since for usual problems the matrix is significantly sparse.
   NOTE: this operation can take minutes when the mapped domains consist of thousands of points.

6. comment the `echo` line in the definition of the external structural mapping force element.

From this point on, when MBDyn is executed it loads the mapping matrix from file `output_file_name` using sparse storage, automatically ignoring coefficients whose absolute value is below `threshold` (defaults to 0).

The parameters of the mapping, as documented in `create_mls_interface`, are

```
% function create_mls_interface([inputfilename])
%       inputfilename [optional argument] string that contains the name
%       of the file created by the MBDyn external mapping force element
%       using the "echo" option
%       Additional rows must be either added in the comment section of the file
%       or can be enforced using the appropriate options of the "echo" keyword
```

```
%       if input values different from default are required. In detail
%
%       # surface: ...
%               used to indicate the name of the file that contains
%               the external element grid data. Default: 'surface.dat'
%
%       # order: used to indicate the polynomial order of the base used by MLS
%               The value must be in the range [1,3]. Default is 2
%
%       # basenode: number of nodes included in the local MLS support
%               should be higher than 0
%
%       # weight: continuity order of the RBF weight function
%               The value must be in the range [-2,inf]
%               -1 means constant weight == 1 in the support
%               -2 means a special weight function == 0 @ the query point
%               default 2
%
%       # output: name of the file where the interface matrix is saved in sparse format
```

Example.

```
    force: 100, external structural mapping,
        socket,
            create, yes,
            path, "/tmp/mbdyn.sock",
            no signal,
        coupling, tight,
        reference node, 0,
        points number, 9,
            10,
                offset, null,       # centered at node
                offset, 0., 1., 0., # y offset
                offset, 0., 0., 1., # z offset
            20,
                offset, null,       # ...
                offset, 0., 1., 0.,
                offset, 0., 0., 1.,
            30,
                offset, null,
                offset, 0., 1., 0.,
                offset, 0., 0., 1.,
        # NOTE: uncomment the line below to generate data required for the mapping
        # echo, "points.dat", surface, "surf.dat", output, "mapping.dat", stop,
        mapped points number, 27,
        sparse mapping file, "mapping.dat";
```

File communicator

TBD

EDGE communicator

TBD

Socket communicator

TBD

The format of the socket communicator is relatively simple; however, implementors may exploit the peer library `libmbc`, whose C API is defined in `mbc.h`. A C++ wrapper is defined in `mbcxx.h`. A python wrapper is also available.

Output

TBD

8.8.12 External modal

This element allows to communicate with an external software that computes forces applied to a `modal` joint element, and may depend on the values of the corresponding generalized variables. Communication occurs by means of the communicators illustrated earlier.

```
<elem_type> ::= force

<force_type> ::= external modal

<force_arglist> ::= <external_force_communicator> ,
    <modal_label>
    [ , accelerations ]
    [ , type , { rigid | modal | all } ]
```

`modal_label` is the label of the `modal` joint element the force element is connected to.

If the optional keyword `accelerations` is present, the output will also contain the modal accelerations.

If the optional keyword `type` is present, the mutually exclusive keywords `rigid`, `modal` or `all` inform MBDyn about whether only the rigid or the modal portions of the force are expected. The default is `all`, but some communicators might require a specific value. If `rigid` or `all` are used, the `modal` joint element must be connected to a `modal` node. The number of modes of the modal force must match that of the active modes in the `modal` joint element.

File communicator

If `type` is `rigid` or `all`, the first line of the input file must contain:

- the label of the `modal` node the `modal` joint element is connected to, or 0 if the element is grounded;
- three components of force in the `modal` node reference frame;
- three components of moment in the `modal` node reference frame.

If the `modal` joint element is grounded, the force and moment are ignored. If `type` is `modal` or `all`, each of the following lines contains the value of the generalized force for the respective mode.

The output file file is written in textual form. If `type` is `rigid` or `all`, the first line contains:

- the label of the `modal` node the `modal` joint element is connected to, or 0 if the element is grounded;

- the position in the global frame;
- the orientation matrix in the global frame;
- the velocity with respect to the global frame;
- the angular velocity with respect to the global frame.

If the optional keyword `accelerations` is present, at the end of each line the following optional fields will appear:

- the acceleration with respect to the global frame;
- the angular acceleration with respect to the global frame.

If the `modal` joint element is grounded, all the data corresponds to that of the origin of the global reference frame. If `type` is `modal` or `all`, each of the following lines contains, for each mode:

- the value of the generalized variable;
- the value of the first derivative of the generalized variable.

If the optional keyword `accelerations` is present, at the end of each line the following optional field will appear:

- the value of the second derivative of the generalized variable.

EDGE communicator

When the `edge` communicator is used, the `accelerations` keyword is ignored. `type` must be either `rigid` or `modal`. If both need to be used, two `external modal` forces must be instantiated: one with `type` set to `rigid` and the other with `type` set to `modal`.

Rigid. When `type` is `rigid`, the data file written by the external software is expected to contain:

```
* this is a comment
body_forces,R,1,6,0
<fx> <fy> <fz> <mx> <my> <mz>
```

where `fx`, `fy`, `fz`, `mx`, `my`, `mz` respectively are the force and moment components oriented according to the global reference frame. The moments are referred to the point that represents the origin of the FE model reference frame, or to either of the `origin position` or the `origin node` of the `modal` element, if defined.

The data file written by MBDyn contains:

```
* this is a comment
body_dynamics,N,0,0,3
* Body linear velocity in body axes
VRELV,R,1,3,0
<vx> <vy> <vz>
* Body angular velocity in body axes
VRELM,R,1,3,0
<omegax> <omegay> <omegaz>
* Body reference frame cosines (listed by columns)
OMGMAN,R,3,3,0
```

```
<R11> <R21> <R31>
<R12> <R22> <R32>
<R13> <R23> <R33>
```

where:

- vx, vy, vz are the components of the velocity of the rigid body, in body axes;
- omegax, omegay, omegaz are the components of the angular velocity of the rigid body, in body axes;
- R11, R12, R13, R21, R22, R23, R31, R32, R33 are the components of the orientation matrix.

Modal. When type is modal, the external software is expected to write a data file containing:

```
* this is a comment
modal_force_flow,R,<n>,1,0
<f1> ... <fn>
```

where n is the number of modes (must match the number of active modes in the modal joint element), while f1 to fn are the amplitudes of the modal forces.

The data file written by MBDyn contains:

```
* this is a comment
modal_state,N,0,0,2
modal_coordinate,R,1,<n>,0
<q1> ... <qn>
modal_velocity,R,1,<n>,0
<qp1> ... <qpn>
```

where:

- n is again the number of modes;
- q1 to qn are the amplitudes of the modal coordinates;
- qp1 to qpn are the amplitudes of the derivatives of the modal coordinates.

Each field can be separated by an arbitrary amount of whitespace, and by at most one "comma" (","). Although specifically developed for ease of interfacing with EDGE, this format is open, and thus can be implemented by other software in order to easily couple with MBDyn.

Socket communicator

When the socket communicator is used, the communication layer is established according to the definition provided at page 235. For a single external modal force with socket communicator, type can be any of rigid, modal, or all.

Example.

```
force: 10, external modal,
    socket,
        create, yes,
        path, "/tmp/mbdyn.sock",
        no signal,
    coupling, tight,
    2;    # modal joint label
```

Output

An external modal element writes multiple lines at each time step in the `.frc` file.

If `type` is `ridig` or `all`, the line contains:

- in column 1 the label of the element and that of the corresponding node; the format of this field is `element_label#node_label`;
- in columns 2–4 the three components of the force;
- in columns 5–7 the three components of the moment.

If `type` is `modal` or `all`, each line contains:

- in column 1 the label of the element and the corresponding mode; the format of this field is `element_label.mode_number`;
- in column 2 the amplitude of the corresponding modal force.

8.8.13 External modal mapping

This element allows to communicate with an external software that computes forces applied to a set of structural nodes, and may depend on the values of the corresponding variables. Modal forces are actually computed, based on the value of modal variables. The element thus projects the motion of the set of structural nodes in the space of a set of modes before communicating it to the peer. Similarly, it receives generalized forces that are projected back in the space of the structural nodes before assembly. Communication occurs by means of the communicators illustrated earlier for the `external modal` case.

```
<elem_type> ::= force

<force_type> ::= external modal mapping

<force_arglist> ::= <external_force_communicator> ,
    [ reference node , <ref_node_label> , ]
    [ use rigid body forces , { yes | no } ,
        [ rotate rigid body forces , { yes | no } , ] ]
    [ accelerations , { yes | no } , ]
    [ type , { rigid | modal | all } , ]
    nodes number , <num_nodes> ,
        <node_1_label> [ , ... ] ,
    modes number , { from file | <num_modes> } ,
    { full | sparse } mapping file , " <mapping_file_name> "
        [ , threshold , <threshold> ]
        [ , transpose ]
```

`ref_node_label` is the label of the reference node if the modes express the motion relative to a specific node.

The mapping file contains the matrix that maps the nodal and modal displacements, velocities and forces. When `full mapping file` is used, a file containing `num_modes` $\times$ (6 $\times$ `num_nodes`) coefficients is expected.

When the optional `transpose` keyword is used, the file is expected to contain the same number of coefficients, but organized as (6 $\times$ `num_nodes`) $\times$ `num_modes` coefficients.

When `sparse mapping file` is used, a file containing

<row> <col> <value>

triplets is expected, with $1 \leq$ row $\leq$ num_modes and $1 \leq$ col $\leq 6 \times$ num_nodes. Both files may start with an arbitrary number of lines beginning with a hash mark ('#'). They are treated as comments and ignored.

If a threshold is given, only elements whose absolute value is larger than threshold are retained. The value of threshold defaults to 0.

The mapping matrix is internally stored and handled as sparse, regardless of the file format.

When the keyword from file is used, the number of modes num_modes is computed from the matrix contained in the file mapping_file_name.

If the optional keyword accelerations is present, the output will also contain the modal accelerations.

If the optional keyword type is present, the mutually exclusive keywords rigid, modal or all inform MBDyn about whether only the rigid or the modal portions of the force are expected. The default is all, but some communicators might require a specific value. If rigid or all are used, the reference node must be defined.

File communicator

Same as external modal.

EDGE communicator

Same as external modal.

Socket communicator

Same as external modal.

Example.

```
force: 10, external modal mapping,
    socket,
        create, yes,
        path, "/tmp/mbdyn.sock",
        no signal,
    coupling, tight,
    send after predict, no,
    # comment for absolute test
    reference node, 0,
    nodes number, 3,
        1, 2, 3,
    modes number, 3,
    # full mapping file, "Hp_full.dat",
    sparse mapping file, "Hp_sparse.dat",
    type, modal;
```

where file Hp_sparse.dat contains

```
# Created by Octave 3.0.1, Mon Dec 14 22:20:31 2009 CET <mbdyn@polimi.it>
# name: Hps
# type: sparse matrix
```

```
# nnz: 7
# rows: 3
# columns: 18
3 4 0.3333333333333332
2 8 0.2352941176470588
1 9 0.4
3 10 0.3333333333333333
2 14 0.9411764705882351
1 15 0.7999999999999999
3 16 0.3333333333333333
```

corresponding to

$$\texttt{Hps} = \left[\begin{array}{cccccc|cccccc|cccccc} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.4 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.8 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.235 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.941 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.333 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.333 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.333 & 0.0 & 0.0 \end{array}\right]$$

Output

An external modal mapping element writes multiple lines at each time step in the `.frc` file.

If `type` is `ridig` or `all`, the line contains:

- in column 1 the label of the element and that of the corresponding node; the format of this field is `element_label@node_label`;
- in columns 2–4 the three components of the force, as received from the peer;
- in columns 5–7 the three components of the moment, as received from the peer;
- in columns 8–10 the three components of the force, in the global reference frame;
- in columns 11–13 the three components of the moment with respect to the reference node, in the global reference frame;
- in columns 14–16 the three components of the force, in the global reference frame, resulting from the combination of the mapped forces;
- in columns 17–19 the three components of the moment with respect to the reference node, in the global reference frame, resulting from the combination of the mapped forces and moments.

If `type` is `modal` or `all`, two types of output are generated:

- a line for each node participating in the mapping, containing:
  - in column 1 the label of the element and the corresponding node; the format of this field is `element_label#node_label`;
  - in columns 2–4 the three components of the corresponding nodal force, in the global reference frame;
  - in columns 5–7 the three components of the corresponding nodal moment, in the global reference frame;
  - in columns 8–10 the three components of the nodal displacement, either in the global reference frame, or in that of the reference node, if any;
  - in columns 11–13 the three components of the nodal orientation, either in the global reference frame, or in that of the reference node, if any;

  - in columns 14–16 the three components of the nodal velocity, either in the global reference frame, or in that of the reference node, if any;
  - in columns 17–19 the three components of the nodal angular velocity, either in the global reference frame, or in that of the reference node, if any.

- a line for each mode the motion is mapped to, containing:
  - in column 1 the label of the element and the corresponding mode; the format of this field is `element_label.mode_number`;
  - in column 2 the amplitude of the corresponding modal force;
  - in column 3 the amplitude of the corresponding modal variable;
  - in column 4 the amplitude of the corresponding modal variable derivative.

8.9 Genel Element

GENEL is the compact form for GENeral ELement. Those elements that cannot in general be classified in a precise way, or are just under development and thus are not collected in a class of their own until their configuration is stabilized, usually are classified as GENEL. The syntax of the GENEL elements is:

```
<elem_type> ::= genel

<normal_arglist> ::= <genel_type> , <genel_arglist>
```

The output goes in a file with extension `.gen`; only few elements actually generate output.
At present, the GENEL class contains very basic general elements and some elements specifically developed for rotorcraft analysis. The latter could be moved to a more specific class in future releases.

8.9.1 General Purpose Elements

Clamp

```
<genel_type> ::= clamp

<genel_arglist> ::=
    (NodeDof) <clamped_node> ,
    { from node | (DriveCaller) <imposed_value> }
```

This element simply forces one arbitrary degree of freedom to assume a value depending on the drive. If the keyword `from node` is used, then the initial value of `<clamped_node>` is used as an imposed value for that node. This is useful especially to clamp structural displacement nodes, if their position is defined in a non trivial reference frame.

Output The format is:

- the label of the element
- the value of the reaction unknown

Distance

```
<genel_type> ::= distance

<genel_arglist> ::=
    (NodeDof) <node_1> ,
    (NodeDof) <node_2> ,
    (DriveCaller) <imposed_distance>
```

This element forces the difference between two arbitrary degrees of freedom to assume the value dictated by the driver.

Output The format is:

- the label of the element
- the value of the reaction unknown

Spring

```
<genel_type> ::= spring

<genel_arglist> ::=
    (NodeDof) <node_1> ,
    (NodeDof) <node_2> ,
    (ConstitutiveLaw<1D>) <const_law>
```

The constitutive law must be `elastic`, but the `distance` genel can apply to arbitrary order degrees of freedom, even between degrees of freedom of different order.

Spring support

```
<genel_type> ::= spring support

<genel_arglist> ::=
    (NodeDof) <node> ,
    (ConstitutiveLaw<1D>) <const_law>
```

The `spring support` must use the `algebraic` value of a `differential` node, but it can use an arbitrary constitutive law, i.e. an elastic constitutive law for a spring, or a viscous constitutive law for a damper, and so on.

Cross spring support

```
<genel_type> ::= cross spring support

<genel_arglist> ::=
    (NodeDof) <row_node> ,
    (NodeDof) <col_node> ,
    (ConstitutiveLaw<1D>) <const_law>
```

It writes a term depending on the col_node degree of freedom in an arbitrary manner (given by the const_law) to the row_node equation.

The cross spring support must use the algebraic value of a differential node, but can use an arbitrary constitutive law, i.e. an elastic constitutive law for a spring, or a viscous constitutive law for a damper, and so on.

Mass

```
<genel_type> ::= mass

<genel_arglist> ::=
    (NodeDof) <node> ,
    (DriveCaller) <mass>
```

The mass must use the algebraic value of a differential node. The derivative of the differential value of the dof is differentiated in a state-space sense, and an inertial driven term is applied to the equation related to the dof:

$$\begin{aligned} m\dot{u} + \ldots &= f \\ u - \dot{x} &= 0 \end{aligned}$$

Scalar filter

```
<genel_type> ::= scalar filter

<genel_arglist> ::=
    (NodeDof) <output_node> ,
    { [ node dof , ] (NodeDof) <input_node>
        | drive , (DriveCaller) <input_value> } ,
    [ canonical form , { controllable | observable } , ]
    <output_order> [ , <output_coef_list> ] ,
    <input_order> [ , <input_coef_list> ]
    [ , gain , <gain> ]
    [ , balance , { yes | no } ]
```

This element models a scalar filter of the form

$$A(s)\, y = B(s)\, u$$

where A, B are polynomials of arbitrary order, namely

$$y(s) = \frac{b_0 s^{n_n} + b_1 s^{n_n - 1} + \ldots + b_{n_n}}{a_0 s^{n_d} + a_1 s^{n_d - 1} + \ldots + a_{n_d}} u(s)$$

The filter must be proper, namely the output_order n_d must be greater than or at most equal to the input_order, n_n. The polynomial A is assumed to be monic, so the coefficient $a_0 = 1$ and only the coefficients from a_1 to a_{n_d} must be input, while all the coefficients of polynomial B are required, i.e. from b_0 to b_{n_n}.

If a gain is supplied, all the coefficients of B are multiplied by the gain.

Originally, the input needed to be taken from a ScalarDof; now, it can also be taken from a drive, which can completely replace the original form (e.g. by means of the dof, the node and the element drives) and requires much less effort because no auxiliary node needs to be set up.

The filter is internally rewritten as a `state space SISO` element. By default, the `controllable` canonical form is used, namely

$$\begin{Bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \vdots \\ \dot{x}_{n_d} \end{Bmatrix} = \begin{bmatrix} -a_1 & -a_2 & -a_3 & \dots & -a_{n_d} \\ 1 & 0 & 0 & & 0 \\ 0 & 1 & 0 & & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n_d} \end{Bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} u(t)$$

$$y = \begin{bmatrix} b_1 & b_2 & b_3 & \dots & b_{n_d} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n_d} \end{Bmatrix}$$

Otherwise, a `observable` canonical form can be used, namely

$$\begin{Bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \vdots \\ \dot{x}_{n_d} \end{Bmatrix} = \begin{bmatrix} -a_1 & 1 & 0 & \dots & 0 \\ -a_2 & 0 & 1 & & 0 \\ -a_3 & 0 & 0 & & 0 \\ \vdots & & & \ddots & \vdots \\ -a_{n_d} & 0 & 0 & \dots & 0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n_d} \end{Bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n_d} \end{bmatrix} u(t)$$

$$y = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n_d} \end{Bmatrix}$$

In case the filter is not strictly proper, because $n_n = n_d = n$, it is turned into a strictly proper one by the transformation

$$\begin{aligned} H(s) &= \frac{B(s)}{A(s)} \\ &= \frac{b_0 s^n + b_1 s^{n-1} + \ldots + b_n}{s^n + a_1 s^{n-1} + \ldots + a_n} \\ &= b_0 + \frac{B(s) - b_0 A(s)}{A(s)} \\ &= b_0 + \frac{+(b_1 - b_0 a_1) s^{n-1} + \ldots + (b_n - b_0 a_n)}{s^n + a_1 s^{n-1} + \ldots + a_n} \end{aligned}$$

and the state space realization is modified by using the resulting numerator coefficients and by introducing a direct feedthrough term $b_0 u(t)$ in the output.

Example. To model a 2nd order Butterworth filter of equation

$$H(s) = \frac{1}{\left(\frac{s}{\omega_c}\right)^2 + \sqrt{2}\left(\frac{s}{\omega_c}\right) + 1} \tag{8.69}$$

use

```
set: real OMEGA_C = 1.*2*pi;                    # filter at 1Hz
genel: 1, scalar filter,
    100, abstract, algebraic,                   # output node
    drive, sine, 0., 2.*pi, 1., forever, 0.,    # input drive
    2, sqrt(2.)*OMEGA_C, OMEGA_C^2,             # a_1, a_2
    0, 1.,                                      # b_0
    gain, OMEGA_C^2;
```

Note that the formula of Eq. (8.69) had to be rearranged in order to make the denominator monic.

State space SISO

```
<genel_type> ::= state space SISO

<genel_arglist> ::=
    (NodeDof) <output_node> ,
    { [ node dof , ] (NodeDof) <input_node>
        | drive , (DriveCaller) <input_value> } ,
    <state_order>
    [ , matrix E , <matrix_E_coef_list> ] ,
    matrix A , <matrix_A_coef_list> ,
    matrix B , <matrix_B_coef_list> ,
    matrix C , <matrix_C_coef_list>
    [ , matrix D , <D_coef> ]
    [ , gain , <gain> ]
    [ , balance , { yes | no } ]
    [ , value , <x0_list>
        [ , derivative , <x0p_list> ] ]
```

This element models a scalar (SISO) state space filter of the form

$$\begin{aligned} \boldsymbol{E}\dot{\boldsymbol{x}} &= \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}u \\ y &= \boldsymbol{C}\boldsymbol{x} + Du \end{aligned}$$

where

- $\boldsymbol{E}$ is a(n optional) matrix `state_order` $\times$ `state_order`, defaulting to the identity matrix $\boldsymbol{I}$,
- $\boldsymbol{A}$ is a matrix `state_order` $\times$ `state_order`,
- $\boldsymbol{B}$ is a vector `state_order` $\times$ 1,
- $\boldsymbol{C}$ is a vector 1 $\times$ `state_order`, and
- D is a(n optional) scalar.

The matrices are read row-oriented; e.g., for matrix $\boldsymbol{A}$:

```
matrix A,
   a11, a12, ..., a1N,
   a21, a22, ..., a2N,
   ...,
   aN1, aN2, ..., aNN
```

If LAPACK is available, the matrices are balanced by default, unless explicitly disabled by using the balance optional keyword. If matrix **E** is defined, the dggbal() routine is used; the dgebal() routine is used otherwise.

The state and its derivative can be initialized using the keywords value and derivative. The initialization of the derivative only makes sense when the filter is in descriptor form.

State space MIMO

```
<genel_type> ::= state space MIMO

<genel_arglist> ::=
    <num_outputs> , (NodeDof) <output_node_list> ,
    <num_inputs> ,
        { [ node dof , ] (NodeDof) <input_node>
            | drive , (DriveCaller) <input_value> }
        [ , ... ] ,
    <state_order>
    [ , matrix E , <matrix_E_coef_list> ] ,
    matrix A , <matrix_A_coef_list> ,
    matrix B , <matrix_B_coef_list> ,
    matrix C , <matrix_C_coef_list>
    [ , matrix D , <matrix_D_coef_list> ]
    [ , gain , <gain> ]
    [ , balance , { yes | no } ]
    [ , value , <x0_list>
        [ , derivative , <x0p_list> ] ]
```

This element models a vector (MIMO) state space filter of the form

$$\begin{aligned} \boldsymbol{E}\dot{\boldsymbol{x}} &= \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{u} \\ \boldsymbol{y} &= \boldsymbol{C}\boldsymbol{x} + \boldsymbol{D}\boldsymbol{u} \end{aligned}$$

where

- $\boldsymbol{E}$ is a(n optional) matrix state_order $\times$ state_order, defaulting to the identity matrix $\boldsymbol{I}$,
- $\boldsymbol{A}$ is a matrix state_order $\times$ state_order,
- $\boldsymbol{B}$ is a matrix state_order $\times$ num_inputs,
- $\boldsymbol{C}$ is a matrix num_outputs $\times$ state_order, and
- $\boldsymbol{D}$ is a(n optional) matrix num_outputs $\times$ num_inputs.

The matrices are read row-oriented.

If LAPACK is available, the matrices are balanced by default, unless explicitly disabled by using the balance optional keyword. If matrix **E** is defined, the dggbal() routine is used; the dgebal() routine is used otherwise.

The state and its derivative can be initialized using the keywords value and derivative. The initialization of the derivative only makes sense when the filter is in descriptor form.

Example. To model the same 2nd order Butterworth filter of Eq. (8.69), namely

$$H(s) = \frac{1}{\left(\frac{s}{\omega_c}\right)^2 + \sqrt{2}\left(\frac{s}{\omega_c}\right) + 1} \tag{8.70}$$

i.e.

$$\left\{ \begin{array}{c} \dot{x}_1 \\ \dot{x}_2 \end{array} \right\} = \left[\begin{array}{cc} 0 & 1 \\ -\omega_c^2 & -\sqrt{2}\omega_c \end{array} \right] \left\{ \begin{array}{c} x_1 \\ x_2 \end{array} \right\} + \left[\begin{array}{c} 0 \\ 1 \end{array} \right] u \tag{8.71a}$$

$$y = \left[\begin{array}{cc} \omega_c^2 & 0 \end{array} \right] \left\{ \begin{array}{c} x_1 \\ x_2 \end{array} \right\} \tag{8.71b}$$

use

```
set: real OMEGA_C = 1.*2*pi;                    # filter at 1Hz
genel: 1, state space SISO,
    100, abstract, algebraic,                   # output node
    drive, sine, 0., 2.*pi, 1., forever, 0.,    # input drive
    2,
    matrix A,
        0., 1.,
        -(OMEGA_C^2), -sqrt(2.)*OMEGA_C,
    matrix B,
        0.,
        1.,
    matrix C,
        0, 1.,
    gain, OMEGA_C^2,
    balance, yes;
```

Note that in `-(OMEGA_C^2)` the brackets are necessary because the unary minus takes precedence over the power operator.

8.9.2 Special Rotorcraft Genel Elements

The GENEL elements specifically developed for use in rotorcraft analysis currently include:

- the `swashplate`;
- the `rotor trim`.

Swashplate

The `swashplate` GENEL is used to transform the controls of a rotor, in terms of collective and fore/aft and lateral cyclic pitch, into the elongations of the actuators that actually move the swash plate. The syntax of the `swashplate` is:

```
<genel_type> ::= swash plate

<genel_arglist> ::=
    <collective_abstract_node>
```

```
    [ , limits , <min_collective> , <max_collective> ] ,
<fore/aft_abstract_node>
    [ , limits , <min_fore/aft> , <max_fore/aft> ] ,
<lateral_abstract_node>
    [ , limits , <min_lateral> , <max_lateral> ] ,
<actuator_1_abstract_node_label> ,
<actuator_2_abstract_node_label> ,
<actuator_3_abstract_node_label>
[ , <dynamic_coef> , <cyclic_factor> , <collective_factor> ]
```

The first three abstract nodes contain the input values of collective, fore/aft and cyclic pitch (they can be actuated by means of abstract forces), and the limits on the "angles" can be easily set. The last three nodes will contain the values of the stroke of the actuators.

The first actuator should be put at the azimuthal position that makes the current blade assume the fore/aft pitch, so that its pitch link is atan $(1/\omega_\beta)$ before the front direction. The other actuators should follow, 120 deg apart in clockwise direction.

The limits on the actuators will simply force the value of the control inputs to remain in the boundaries regardless of the input values.

The last three optional parameters are a dynamic coefficient that is used to add some dynamics to the actuators' stroke, namely the input variables are applied a sort of `spring stiffness bulk` element, while the actuators' strokes are applied a transfer function of the first order, namely $\alpha\dot{x} + x = f$, where $\alpha =$ `dynamic_coef` and f is the desired stroke, so the smaller is α, the more the behavior is static.

The `cyclic_factor` and the `collective_factor` parameters are used to scale the inputs from angles in the desired units to strokes, that usually are dimensional parameters. The actual strokes are made of the collective contribution multiplied by `collective_factor`, and the cyclic contribution multiplied by `collective_factor` $\times$ `cyclic_factor`.

Rotor Trim

The syntax of the `rotor trim` is

```
<genel_type> ::= rotor trim

<genel_arglist> ::= [ <rotor_trim_type> ] ,
    <type_specific_rotor_trim_data> ,
    <thrust_node_label> ,
    <longitudinal_moment_node_label> ,
    <lateral_moment_node_label> ,
    (DriveCaller) <desired_thrust_coefficient> ,
    (DriveCaller) <desired_longitudinal_moment_coefficient> ,
    (DriveCaller) <desired_lateral_moment_coefficient> ,
    <rotor_lock_number> ,
    <rotor_nondimensional_flap_frequency> ,
    <thrust_time_constant> , <moments_time_constant> ,
    <thrust_gain> , <moments_gain>
    [ , trigger , (DriveCaller) <trigger> ]

<rotor_trim_type> ::= { rotor | generic }

# for rotor
<type_specific_rotor_trim_data> ::=
```

```
    <rotor_label>

# for generic
<type_specific_rotor_trim_data> ::=
    [ reference node , <ref_node_label> , ]
    (DriveCaller) <thrust> ,
    (DriveCaller) <longitudinal_moment> ,
    (DriveCaller) <lateral_moment> ,
    <rotor_radius> ,
    (DriveCaller) <angular_velocity> ,
    (DriveCaller) <advance_ratio>
```

The `rotor trim` is experimental; it allows to set the controls of a generic helicopter, in conjunction with the `swashplate` element, to asymptotically obtain the desired level of thrust and moment coefficients. The corresponding behavior in terms of trim values (flapping angles and shaft angle) has not been implemented yet. For details, see [35].

8.10 Gravity Element

```
<elem_type> ::= gravity

<arglist> ::= <gravity_model> , <gravity_data>

<gravity_model> ::=
    [ uniform , ]          # uniform gravity field (default)
        (TplDriveCaller<Vec3>) <gravity_acceleration>
    | central ,            # central gravity field
        [ origin , (Vec3) <absolute_origin> , ]
        mass , (real) <central_gravity_field_mass> ,
        G , { si | (real) <universal_gravity_constant> }
```

The 3D drive caller `gravity_acceleration` represents the gravity acceleration in the global reference frame. The keyword `uniform` is optional; a uniform gravity field is assumed when `gravity_model` is missing.

The optional vector `absolute_origin` is the absolute origin of the central gravity field. The scalar `mass` is the central mass of the central gravity field. the scalar `universal_gravity_constant` is the universal gravity constant; if the keyword `si` is used, the value $\mathbf{6.67384} \times \mathbf{10}^{-11}\ \mathbf{m}^3\ \mathbf{kg}^{-1}\ \mathbf{s}^{-2}$ is used (source: http://physics.nist.gov/cuu/Constants/index.html.

8.11 Hydraulic Element

Note: under development; syntax subjected to changes
Initially implemented by: Lamberto Puggelli
Reviewed by: Pierangelo Masarati

```
<elem_type> ::= hydraulic

<normal_arglist> ::= <hydr_elem_type> , <hydr_elem_data>
```

The field hydraulic_element_data usually contains information about the required hydraulic fluid properties, which are handled by means of a HydraulicFluid. This can be directly inserted, following the syntax described in Section 2.11 preceded by the keyword hydraulic fluid, or a previously defined hydraulic fluid can be recalled by using the keyword reference followed by the label of the desired hydraulic fluid:

```
<hydraulic_fluid_properties> ::=
    { <hydraulic_fluid_specification>
        | reference , <hydraulic_fluid_label> }
```

8.11.1 Accumulator

This element implements a simple accumulator. Three options are available: gas, weight, and spring. The element must be connected to a hydraulic network via a hydraulic node. The rate of accumulated fluid depends on the pressure difference between the accumulator chamber and of the node. Although a weight mode is provided, it does not account for the actual gravity field; it assumes that the accumulator is mounted with the piston in the vertical position and that gravity is uniform. Although the mass of the piston is required, it does not account for the actual motion the system is subjected to; it assumes that the cylinder is in an inertial reference frame. Use with care!

```
<hydr_elem_type> ::= accumulator

<hydr_elem_data> ::=
    <node> ,
    <stroke> ,
    [ start , <initial_position> , ]
    <area> ,
    <pipe_area> ,
    <mass> ,
    [ loss in , <loss_in> , ]
    [ loss out , <loss_out> , ]
    [ gas , <press_0> , <press_max>, <exponent> , ]
    [ weight , <weight> , ]
    [ spring , <spring> , <force_0> , ]
    <c_disp> ,
    <c_vel> ,
    <c_acc> ,
    hydraulic fluid , (HydraulicFluid) <fluid_1>
```

where

- stroke, the total stroke of the accumulator, must be positive
- initial_position, the initial position of the accumulator, must be non-negative and less than or equal to stroke
- area, the area of the accumulator, must be positive
- pipe_area, the area of the pipe leading to the accumulator, must be positive
- mass, the mass of the accumulator, must be positive
- loss_in, (?, must be non-negative); defaults to 1.0

- loss_out (?, must be non-negative); defaults to 0.5
- if the keyword gas is used, it must be followed by
  - press_0, the pressure of the gas when the accumulator is empty, must be positive
  - press_max, the maximum pressure of the gas, must be greater than or equal to press_0
  - exponent, the exponent of the polytropic curve that describes the state of the gas, must be positive
- if the keyword weight is used, it must be followed by weight; must be positive (TBD)
- if the keyword spring is used, it must be followed by
  - spring, the stiffness of the spring, must be positive
  - force_0, the preload of the spring when the accumulator is empty, must be non-negative
- c_disp, c_vel, and c_acc are stiffness, damping, and acceleration-related coefficients used to deal with end position reactions; must be non-negative

The three types of accumulation principles (gas, weight, and spring) can be simultaneously present, although only some combinations make sense.

8.11.2 Actuator

The actuator element models the hydraulic and interactional aspects of a two-way linear hydraulic actuator. The two hydraulic nodes represent the pressure in the two chambers of the cylinder. The two structural nodes represent the cylinder and the piston. Their relative motion is assumed to consist essentially of a displacement along a given direction in the reference frame of the first structural node. The user must take care of this by constraining the two nodes with an appropriate set of joint elements (e.g., a total joint that only allows relative motion along the axis of the cylinder; the relative rotation about the same axis may indifferently be allowed or constrained). The mass of the piston must be provided using appropriate elements associated with the piston's node. Mechanical friction between the piston and the cylinder (e.g., associated with sealings) must be taken care of using appropriate mechanical elements (e.g., a rod joint with a viscous constitutive law). Other hydraulic aspects of this component, like leakages between the two chambers and between the chambers and the outside, and the related pressure losses, must be explicitly taken care of, e.g. by means of minor loss elements.

```
<hydr_elem_type> ::= actuator

<hydr_elem_data> ::=
    <node_1> , <node_2> ,
    <struct_node_1> , (Vec3) <offset_1> ,
    <struct_node_2> , (Vec3) <offset_2> ,
    [ direction , ((Unit)Vec3) <direction> , ]
    <area_1> ,
    <area_2> ,
    <cylinder_length> ,
    hydraulic fluid , (HydraulicFluid) <fluid_1> ,
    { same | hydraulic fluid , (HydraulicFluid) <fluid_2> }
```

The vector direction is internally normalized to unit norm. It represents the direction of the stroke in the reference system of struct_node_1. By default, it is direction 3 in the reference frame of the structural node struct_node_1.

Example. See the "actuator" example at

https://raw.githubusercontent.com/mmorandi/MBDyn-web/main/userfiles/documents/examples/actuator.mbd

and the related chapter of the tutorials

https://github.com/mmorandi/MBDyn-web/raw/main/userfiles/documents/tutorials.pdf

8.11.3 Control Valve

```
<hydr_elem_type> ::= control valve

<hydr_elem_data> ::=
    <node_1> , <node_2> ,
    <node_3> , <node_4> ,
    <area> ,
    [ loss , <loss_factor> , ]
    (DriveCaller) <state> ,
    hydraulic fluid , (HydraulicFluid) <fluid>
```

This element represents a valve that connects `node_1` to `node_2` and `node_3` to `node_4` when `state` is positive, and `node_1` to `node_3` and `node_2` to `node_4` when `state` is negative. The flow area is proportional to `area` times the norm of `state`, which must be comprised between -1 and 1. If `loss_factor` is defined, it represents the fraction of area that leaks when `state` is exactly zero.

8.11.4 Dynamic Control Valve

```
<hydr_elem_type> ::= dynamic control valve

<hydr_elem_data> ::=
    <node_1> , <node_2> ,
    <node_3> , <node_4> ,
    (DriveCaller) <force> ,
    <initial_displacement> ,
    <max_displacement> ,
    <duct_width> ,
    [ loss , <loss_factor> , ]
    <valve_diameter> ,
    <valve_density> ,
    <displacement_penalty> ,
    <velocity_penalty> ,
    <acceleration_penalty> ,
    hydraulic fluid , (HydraulicFluid) <fluid>
```

This element represents a valve that connects `node_1` to `node_2` and `node_3` to `node_4` when the displacement is positive and `node_1` to `node_3` and `node_2` to `node_4` when the displacement is negative, accounting for the dynamics of the valve body. The control force `force` is applied to the valve, whose geometric and structural properties are described by `initial_displacement`, `max_displacement`, `duct_width`, `valve_diameter` and `valve_density`. Again the `loss_factor`, if defined, represents the fraction of the area that leaks when the displacement is zero. Finally, `displacement_penalty`, `velocity_penalty` and `acceleration_penalty` are the penalty coefficients for displacement, velocity and acceleration when the maximum stroke is reached.

8.11.5 Dynamic Pipe

Same syntax as the pipe hydraulic element (Section 8.11.11), it also considers fluid compressibility in terms of pressure time derivative, and thus the corresponding dynamic effect.

```
<hydr_elem_type> ::= dynamic pipe
```

Output

1: label

2: pressure at node 1

3: pressure at node 2

4: pressure derivative at node 1

5: pressure derivative at node 2

6: flow at node 1

7: flow at node 2

8: flow rate at node 1

9: flow rate at node 2

10: density at node 1

11: reference density

12: density at node 2

13: derivative of density with respect to pressure at node 1

14: derivative of density with respect to pressure at node 2

15: Reynolds number

16: flow type flag (boolean; true when flow is turbulent, false otherwise)

8.11.6 Flow Valve

```
<hydr_elem_type> ::= flow valve

<hydr_elem_data> ::=
    <node_1> , <node_2> , <node_3> ,
    <diaphragm_area> ,
    <valve_mass> ,
    <pipe_area> ,
    <valve_max_area> ,
    <spring_stiffness> ,
    <spring_preload> ,
    <width> ,
    <max_stroke> ,
    <displacement_penalty> ,
    <velocity_penalty> ,
    <acceleration_penalty> ,
    hydraulic fluid , (HydraulicFluid) <fluid>
```

Output

1: label

2: valve displacement

3: valve velocity

4: valve acceleration

5: flow through way 1

6: flow through way 2

7: flow through way 3

8.11.7 Imposed Flow

No specific element has been implemented to impose the flow at one node. The `abstract` variant of the `force` element (Section 8.8.2) can be used instead. The magnitude of the abstract force is the mass flow extracted from the circuit at that node. In fact, a negative value of the abstract force means that the flow enters the node.

Example.

```
set: integer HYDRAULIC_NODE_LABEL = 100;
set: integer IMPOSED_FLOW_LABEL = 200;
force: IMPOSED_PRESSURE_LABEL, abstract,
    HYDRAULIC_NODE_LABEL, hydraulic,
    const, -1e-3;
```

8.11.8 Imposed Pressure

No specific element has been implemented to impose the pressure at one node. The `clamp` variant of the `genel` element (Section 8.9.1) can be used instead.

Example.

```
set: integer HYDRAULIC_NODE_LABEL = 100;
set: integer IMPOSED_PRESSURE_LABEL = 200;
genel: IMPOSED_PRESSURE_LABEL, clamp,
    HYDRAULIC_NODE_LABEL, hydraulic,
    const, 101325.0;
```

8.11.9 Minor Loss

A pressure loss between two pressure nodes.

```
<hydr_elem_type> ::= minor loss

<hydr_elem_data> ::=
    <node_1> , <node_2> ,
    <k12> , <k21> , <area> ,
    hydraulic fluid , (HydraulicFluid) <fluid>
```

Coefficients k12 and k21 characterize the pressure loss when the flow goes from node_1 to node_2 and vice versa. Turbulent flow is assumed.

8.11.10 Orifice

```
<hydr_elem_type> ::= orifice

<hydr_elem_data> ::=
    <node_1> , <node_2> ,
    <pipe_diameter> ,
    <orifice_area> , # diaphragm area
    [ area , <pipe_area> , ] # defaults to (<pipe_diameter>/2)^2*pi
    [ reynolds , <critical_reynolds_number> , ] # defaults to 10
    hydraulic fluid , (HydraulicFluid) <fluid>
```

8.11.11 Pipe

This element models a simple pipeline connecting two hydraulic nodes. In detail, it models the pressure loss due to fluid viscosity according to a constitutive law that depends on the Reynolds number computed on average fluid velocity and hydraulic diameter. Transition between laminar and turbulent flow is also modeled.

```
<hydr_elem_type> ::= pipe

<hydr_elem_data> ::=
    <node_1> , <node_2> ,
    <hydraulic_diameter> ,
    [ area , <area> , ]
    <length> ,
    [ turbulent , ]
    [ initial value , <flow> , ]
    hydraulic fluid , (HydraulicFluid) <fluid>
```

When area is not given, it defaults to the equivalent area computed according to the hydraulic diameter. The flag turbulent forces the flow to be considered turbulent since the first iteration, when the initial conditions would fall in the transition region. The initial value parameter is the initial value of the flow internal state.

Example.

```
set: integer NODE_1 = 10;
set: integer NODE_2 = 20;
set: integer PIPE = 100;
set: integer FLUID = 1000;
# ...
hydraulic: PIPE, pipe, NODE_1, NODE_2,
    5e-3, 100.e-3,
    fluid, reference, FLUID;
```

8.11.12 Pressure Flow Control Valve

```
<hydr_elem_type> ::= pressure flow control valve

<hydr_elem_data> ::=
    <node_1> , <node_2> ,
    <node_3> , <node_4> ,
    <node_5> , <node_6> ,
    (DriveCaller) <force> ,
    <initial_displacement> ,
    <max_displacement> ,
    <duct_width> ,
    [ loss , <loss_factor> , ]
    <valve_diameter> ,
    <valve_density> ,
    <displacement_penalty> ,
    <velocity_penalty> ,
    <acceleration_penalty> ,
    hydraulic fluid , (HydraulicFluid) <fluid>
```

Same as Dynamic Control Valve (8.11.4), only the pressures at node_5 and node_6 are applied at the sides of the valve body and participate in the force balance.

8.11.13 Pressure Valve

```
<hydr_elem_type> ::= pressure valve

<hydr_elem_data> ::=
    <node_1> , <node_2> ,
    <area> ,
    <mass> ,
    <max_area> ,
    <spring_stiffness> ,
    <spring_preload> ,
    <width> ,
    <displacement_penalty> ,
    <velocity_penalty> ,
    <acceleration_penalty> ,
    hydraulic fluid , (HydraulicFluid) <fluid>
```

8.11.14 Tank

Not documented yet.

8.11.15 Three Way Minor Loss

A pressure loss between three pressure nodes, depending on the sign of the pressure drop.

```
<hydr_elem_type> ::= three way minor loss

<hydr_elem_data> ::=
```

```
<node_1> , <node_2> , <node_3> ,
<k12> , <k31> , <area_12> , <area_31> ,
hydraulic fluid , (HydraulicFluid) <fluid>
```

Coefficients `k12` and `k31` and the respective values of area characterize the pressure loss when the flow goes from `node_1` to `node_2` and from `node_3` to `node_1`, respectively. Turbulent flow is assumed.

8.12 Joint Element

Joint elements connect structural nodes. Many different joints are available. Joints may have internal degrees of freedom (the reaction forces) when they introduce kinematic constraints in form of algebraic relationships between the coordinates of the nodes they connect. Joints that introduce configuration-dependent forces are flexible joints. From the point of view of the input syntax there is no difference between the two classes. A typical joint entry is

```
<elem_type> ::= joint

<normal_arglist> ::= <joint_type> , <joint_arglist>
```

The output is written to a file with extension `.jnt`. The output is generally made of a standard part, plus some extra information depending on the type of joint, which, when available, is described along with the joint description. Here the standard part is described:

- column 1: the label of the joint
- columns 2–4: the three components of the reaction force in a local reference
- columns 5–7: the three components of the reaction couple in a local frame
- columns 8–10: the three components of the reaction force in the global frame
- columns 11–13: the three components of the reaction couple, rotated into the global frame

Legal joint types, with relative data, are:

8.12.1 Angular acceleration

This joint imposes the absolute angular acceleration of a node about a given axis.

```
<joint_type> ::= angular acceleration

<joint_arglist> ::=
    <node_label> ,
    ((Unit)Vec3) <relative_direction> ,
    (DriveCaller) <acceleration>
```

The axis is `relative_direction`; it is internally normalized to unity.

Private Data

The following data are available:

1. `"M"` constraint reaction moment along joint direction
2. `"wp"` imposed angular acceleration along joint direction

8.12.2 Angular velocity

This joint imposes the absolute angular velocity of a node about a given axis.

```
<joint_type> ::= angular velocity

<joint_arglist> ::=
    <node_label> ,
    ((Unit)Vec3) <relative_direction> ,
    (DriveCaller) <velocity>
```

The rotation axis of the imposed angular velocity is `relative_direction`; it is internally normalized to unity. It rotates with the node indicated by `node_label`.

Private Data

The following data are available:

1. `"w"` imposed angular velocity about joint direction

8.12.3 Axial rotation

This joint is equivalent to a `revolute hinge` (see Section 8.12.38), but the angular velocity about axis 3 is imposed by means of the driver.

```
<joint_type> ::= axial rotation

<joint_arglist> ::=
    <node_1> ,
        [ position , ] (Vec3) <relative_offset_1> ,
        [ orientation , (OrientationMatrix) <relative_orientation_matrix_1> , ]
    <node_2> ,
        [ position , ] (Vec3) <relative_offset_2> ,
        [ orientation , (OrientationMatrix) <relative_orientation_matrix_2> , ]
    (DriveCaller) <angular_velocity>
```

This joint forces `node_1` and `node_2` to rotate about relative axis 3 with imposed angular velocity `angular_velocity`.

Private Data

The following data are available:

1. `"rz"` relative rotation angle about revolute axis
2. `"wz"` relative angular velocity about revolute axis
3. `"Fx"` constraint reaction force in node 1 local direction 1
4. `"Fy"` constraint reaction force in node 1 local direction 2
5. `"Fz"` constraint reaction force in node 1 local direction 3
6. `"Mx"` constraint reaction moment about node 1 local direction 1
7. `"My"` constraint reaction moment about node 1 local direction 2
8. `"Mz"` constraint reaction moment about node 1 local direction 3

Hints

When wrapped by a driven element, the following hints are honored:

- hinge{1} the relative orientation of the joint with respect to node 1 is reset;
- hinge{2} the relative orientation of the joint with respect to node 2 is reset;
- offset{1} the offset of the joint with respect to node 1 is reset;
- offset{2} the offset of the joint with respect to node 2 is reset;
- unrecognized hints are passed to the friction model, if any.

8.12.4 Beam slider

This joint implements a slider, e.g. it constrains a structural node on a string of three-node beams, as discussed in [36].

```
<joint_type> ::= beam slider

<joint_arglist> ::=
    <slider_node_label> ,
    (Vec3) <relative_offset> ,
    [ orientation , (OrientationMatrix) <relative_orientation> , ]
    [ type , { spherical | classic | spline } , ]
     <beam_number> ,
        { <3_node_beam> ,
             { same | (Vec3) <first_node_offset> } ,
[ orientation , { same | (OrientationMatrix) <first_node_orientation> } , ]
            (Vec3) <mid_node_offset> ,
[ orientation , (OrientationMatrix) <mid_node_orientation> , ]
            (Vec3) <end_node_offset> ,
[ orientation , (OrientationMatrix) <end_node_orientation> , ] }
            [ {...} ]
    [ , initial beam , <initial_beam> ]
    [ , initial node , <initial_node> ]
    [ , smearing , <smearing_factor> ]
[ , friction ,
    [ preload , <const_value> , ]
    <friction_model> ,
    <shape_function> ]
```

There are three types of slider:

- the spherical slider does not constrain the orientation of the node;
- the classical slider does allow rotation only about the sliding line;
- the spline slider constrain the orientation of the node.

Give the number of beams covered by the slider as the beam_number variable. Give an ordered list of beams where the slider can glide as a series of beam_number recurrences of the corresponding labels, 3_node_beam, followed by the proper orientation descriptions. For each node of each beam element, the

offset and the orientation of the slider can be defined; except for the first element, the offset and the orientation of the first node can be specified using the keyword same, which causes the node to take the same value of the last node of the previous beam. The initial_beam and initial_node indices serve as hints to set the initial contact point of the sliding node. *Warning*: the initial_beam and initial_node indices do not correspond to the labels of the beam and node but rather to their positions, counting from 1, in the list given to the slider and in the beam's declaration, respectively. The smearing_factor determines the (non-dimensional) extension of the interference segment when the node passes from one segment to another. Further information is available in [36]. A simple example of two beam sliders goes as follows:

```
set: real smr = .1;
joint: 100, beam slider,
        51, null,
                orientation, eye,
        type, classic,
        5,
                1, null, null, null,
                2, null, null, null,
                3, null, null, null,
                4, null, null, null,
                5, null, null, null,
        initial beam, 1,
        initial node, 1,
        smearing, smr;
joint: 101, beam slider,
        11, null,
                orientation, eye,
        type, classic,
        5,
                51, null, null, null,
                52, null, null, null,
                53, null, null, null,
                54, null, null, null,
                55, null, null, null,
        initial beam, 5,
        initial node, 3,
        smearing, smr;
```

In addition to the standard output the beam slider joint outputs:

- column 14: the label of the current beam;
- column 15: the value of the current curvilinear abscissa;
- columns 16-18: the local direction vector;

If friction is present the additional output is:

- column 19: the friction force along the along the local direction vector;
- column 20: the friction coefficient;
- column 21: the relative sliding velocity along the local direction vector;

8.12.5 Brake

This element models a wheel brake, i.e. a constraint that applies a frictional internal torque between two nodes about an axis. The frictional torque depends on the normal force that is applied as an external input by means of the same friction models implemented for regular joints.

```
<joint_type> ::= brake

<joint_arglist> ::=
    <node_1_label> , (Vec3) <relative_offset_1>
    [ , orientation , (OrientationMatrix) <relative_orientation_matrix_1> ] ,
    <node_2_label> , (Vec3) <relative_offset_2>
    [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ] ,
    friction , <average_radius> ,
        [ preload , <const_value> , ]
        <friction_model> ,
        <shape_function> ,
    (DriveCaller) <normal_force>
```

Note: a `revolute hinge` *(see Section 8.12.38) between the same two nodes must be defined as well, such that the only allowed relative kinematics between the two nodes are a rotation about relative axis 3.*

Output

In addition to the standard output, the `brake` joint outputs:

- the three Euler-Cardano angles that express the relative rotation between the two connected nodes, in degrees;
- the three components of the relative angular velocity, in the reference frame of node 2;
- any output specific for the friction model in use;
- the value of the force the brake is activated with.

Private Data

The following data are available:

1. `"rz"` relative rotation angle about brake axis
2. `"wz"` relative angular velocity about brake axis

8.12.6 Cardano hinge

This joint implements a Cardano's joint, also known as Hooke's joint or Universal joint, which is made of a sequence of two revolute hinges orthogonal to each other, one about relative axis 2 and one about relative axis 3 of the reference systems defined by the two `orientation` statements. In other words, this joint constrains the relative axis 3 of node 1 to be always orthogonal to the relative axis 2 of node 2. As a result, torque is transmitted about axis 1 of both nodes. The relative position is constrained as well.

```
<joint_type> ::= Cardano hinge

<joint_arglist> ::=
```

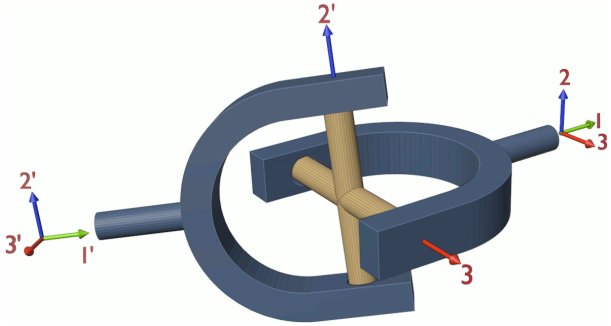


Figure 8.8: Cardano hinge.

```
<node_1_label> ,
    position , (Vec3) <relative_offset_1>
    [ , orientation , (OrientationMatrix) <relative_orientation_matrix_1> ] ,
<node_2_label> ,
    position , (Vec3) <relative_offset_2>
    [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ]
```

The joint is shown in Figure 8.8 where the origins and orientations of `node_1` and `node_2` are given by axes 1, 2, 3 and $1'$, $2'$, $3'$, respectively. The central, constrained, axes 3 and $2'$ are obtained after having applied `relative_offset` and `relative_orientation_matrix` transformations to `node_1` and `node_2`, respectively. Note: this joint does not represent a constant velocity joint, so, when a steady deflection between the two nodes is present, a constant velocity about axis 1 of one node results in an oscillating velocity about axis 1 for the other node. The constant velocity joint is implemented by the `gimbal joint` (Section 8.12.22), which essentially consists of a sequence of two `Cardano hinge` (see Section 8.12.6) joints with the hinges in reversed order, assembled in a manner that the deflection is split half and half between the two joints.

8.12.7 Cardano pin

This joint implements a "Cardano" joint between a node and the ground. The absolute position is also constrained. See above for details about the formulation.

```
<joint_type> ::= Cardano pin

<joint_arglist> ::=
    <node_label> ,
        position , (Vec3) <relative_offset>
        [ , orientation  , (OrientationMatrix) <relative_orientation_matrix> ] ,
    position  , (Vec3) <absolute_pin_position>
    [ , orientation , (OrientationMatrix) <absolute_pin_orientation_matrix> ]
```

Note: this is equivalent to a `Cardano hinge` (see Section 8.12.6) when one node is grounded.

8.12.8 Cardano rotation

This joint implements a "Cardano" joint, which is made of a sequence of two revolute hinges orthogonal to each other. The relative position is not constrained. See the `Cardano hinge` for more details.

```
<joint_type> ::= Cardano rotation

<joint_arglist> ::=
    <node_1_label>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_1> ] ,
    <node_2_label>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ]
```

Note: this is equivalent to a Cardano hinge (see Section 8.12.6) without position constraint. Or, in other words, a Cardano hinge (see Section 8.12.6) is made of a Cardano rotation on top of a spherical hinge (see Section 8.12.45).

8.12.9 Clamp

This joint grounds all 6 degrees of freedom of a node in an arbitrary position and orientation that remains fixed.

```
<joint_type> ::= clamp

<joint_arglist> ::= <node_label> ,
    [ position , ]
        { node | (Vec3) <absolute_position> } ,
    [ orientation , ]
        { node | (OrientationMatrix) <absolute_orientation_matrix> }
```

The keyword `node` forces the joint to use the node's position and reference frame. Otherwise, they must be entered in the usual way for these entities. The default value for `position` and `orientation` are the position and the orientation of the clamped node.

Private Data

The following data are available:

1. `"Fx"` constraint reaction force in global direction 1
2. `"Fy"` constraint reaction force in global direction 2
3. `"Fz"` constraint reaction force in global direction 3
4. `"Mx"` constraint reaction moment in local direction 1
5. `"My"` constraint reaction moment in local direction 2
6. `"Mz"` constraint reaction moment in local direction 3

8.12.10 Coincidence

not implemented yet, use a `spherical hinge` (see Section 8.12.45) and a `prismatic` instead.

8.12.11 Deformable Axial Joint

This joint implements a configuration (i.e. relative motion) dependent moment that is exchanged between two nodes about an axis rigidly attached to the first node. It is intended to be used in conjunction with another joint (for example a `revolute hinge` or a `total joint`) that constrains the relative rotation between the two nodes about the other axes, although no check is in place.

```
<joint_type> ::= deformable axial joint

<joint_arglist> ::=
    <node_1_label>
        [ , position , (Vec3) <relative_offset_1> ]
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_1> ] ,
    <node_2_label>
        [ , position , (Vec3) <relative_offset_2> ]
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ] ,
    (ConstitutiveLaw<1D>) <const_law>
```

The relative rotation is assumed to take place about axis 3 in the local reference frame of the joint, as defined by `relative_orientation_matrix_1`.

Private Data

The following data are available:

1. `"rz"` relative rotation about axis 3, used as strain in the constitutive law
2. `"wz"` relative angular velocity about axis 3, used as strain rate in the constitutive law
3. `"Mz"` moment about axis 3, output by the constitutive law

In addition, the joint provides access to any private data provided by the constitutive law. They are accessed by prefixing the name of the data with the string `"constitutiveLaw."`; see the specific constitutive law description of the available data in Section 2.10.

Hints

When wrapped by a `driven` element, the `deformable axial joint` honors the following hints:

- `hinge{1}` the orientation with respect to node 1 of the reference used to compute the linear strain is reset to the value resulting from the node 2 hinge orientation and the node 1 orientation

$$\tilde{\boldsymbol{R}}_{1h} = \boldsymbol{R}_1^T \boldsymbol{R}_2 \tilde{\boldsymbol{R}}_{2h}$$

- `hinge{2}` the orientation with respect to node 2 of the reference used to compute the linear strain is reset to the value resulting from the node 1 hinge orientation and the node 2 orientation

$$\tilde{\boldsymbol{R}}_{2h} = \boldsymbol{R}_2^T \boldsymbol{R}_1 \tilde{\boldsymbol{R}}_{1h}$$

- unrecognized hints are passed through to the constitutive law.

Output

In addition to the standard output, the `deformable axial joint` outputs:

- column 14: the relative rotation about axis 3

If the constitutive law is either viscous or viscoelastic:

- column 15: the relative angular velocity about axis 3

8.12.12 Deformable displacement hinge

Deprecated; use the `deformable displacement joint` (see Section 8.12.13) instead.

8.12.13 Deformable displacement joint

This joint implements a configuration dependent force that is exchanged between two points associated with two nodes with an offset. The force may depend, by way of a generic 3D constitutive law, on the relative position and velocity of the two points, expressed in the reference frame of node 1.
The constitutive law is attached to the reference frame of node 1, so the sequence of the connections may matter in case of anisotropic constitutive laws, if the relative orientation of the two nodes changes during the analysis.

```
<joint_type> ::= deformable displacement joint

<joint_arglist> ::=
    <node_1_label> ,
        position , (Vec3) <relative_offset_1>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_1> ] ,
    <node_2_label> ,
        position , (Vec3) <relative_offset_2>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ] ,
    (ConstitutiveLaw<3D>) <const_law>
```

Note: a variant of this element is under development, which refers the material reference frame to an orientation that is intermediate between those of the two nodes. See the `invariant deformable displacement joint` (Section 8.12.27).

Private Data

The following data are available:

1. `"dx"` relative displacement in node 1 local direction 1
2. `"dy"` relative displacement in node 1 local direction 2
3. `"dz"` relative displacement in node 1 local direction 3
4. `"vx"` relative velocity in node 1 local direction 1
5. `"vy"` relative velocity in node 1 local direction 2
6. `"vz"` relative velocity in node 1 local direction 3
7. `"Fx"` constraint reaction force in node 1 local direction 1

8. `"Fy"` constraint reaction force in node 1 local direction 2

9. `"Fz"` constraint reaction force in node 1 local direction 3

In addition, the joint provides access to any private data provided by the constitutive law. They are accessed by prefixing the name of the data with the string `"constitutiveLaw."`; see the specific constitutive law description of the available data in Section 2.10.

Hints

When wrapped by a `driven` element, the `deformable displacement joint` honors the following hints:

- `hinge{1}` the orientation with respect to node 1 of the reference used to compute the linear strain is reset to the value resulting from the node 2 hinge orientation and the node 1 orientation

$$\tilde{\boldsymbol{R}}_{1h} = \boldsymbol{R}_1^T \boldsymbol{R}_2 \tilde{\boldsymbol{R}}_{2h}$$

- `hinge{2}` the orientation with respect to node 2 of the reference used to compute the linear strain is reset to the value resulting from the node 1 hinge orientation and the node 2 orientation

$$\tilde{\boldsymbol{R}}_{2h} = \boldsymbol{R}_2^T \boldsymbol{R}_1 \tilde{\boldsymbol{R}}_{1h}$$

- `offset{1}` the offset of the joint with respect to node 1 is reset to the value resulting from the node 2 offset and the node 1 position

$$\tilde{\boldsymbol{f}}_1 = \boldsymbol{R}_1^T \left(\boldsymbol{x}_2 + \boldsymbol{R}_2 \tilde{\boldsymbol{f}}_2 - \boldsymbol{x}_1 \right)$$

- `offset{2}` the offset of the joint with respect to node 2 is reset to the value resulting from the node 1 offset and the node 2 position

$$\tilde{\boldsymbol{f}}_2 = \boldsymbol{R}_2^T \left(\boldsymbol{x}_1 + \boldsymbol{R}_1 \tilde{\boldsymbol{f}}_1 - \boldsymbol{x}_2 \right)$$

- unrecognized hints are passed through to the constitutive law.

Output

In addition to the standard output, the `deformable displacement joint` outputs:

- columns 14–16: the three components of the linear strain.

If the constitutive law is either viscous or viscoelastic:

- columns 17–19: the three components of the linear strain rate.

8.12.14 Deformable Hinge

This joint implements a configuration dependent moment that is exchanged between two nodes. The moment may depend, by way of a generic 3D constitutive law, on the relative orientation and angular velocity of the two nodes, expressed in the reference frame of node 1.
The constitutive law is attached to the reference frame of node 1, so the sequence of the connections may matter in case of anisotropic constitutive laws, if the relative orientation of the two nodes changes during the analysis.

```
<joint_type> ::= deformable hinge

<joint_arglist> ::=
    <node_1_label>
        [ , position , (Vec3) <relative_position_1> ]
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_1> ] ,
    <node_2_label>
        [ , position , (Vec3) <relative_position_2> ]
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ] ,
    (ConstitutiveLaw<3D>) <const_law>
    [ , orientation description , <orientation_type> ]
```

The `orientation_type` is defined in Section 5.3.12.

Note: a variant of this element has been developed, which refers the material reference frame to an orientation that is intermediate between those of the two nodes. See the `invariant deformable hinge` (Section 8.12.28).

Note: this joint only applies internal moments; no forces and no constraints to displacements are applied. Usually, it is best used in conjunction with a `spherical hinge` (see Section 8.12.45), or any kind of joint that constrains the relative displacement of the nodes as appropriate.

Private Data

The following data are available:

1. `"rx"` relative rotation vector component in node 1 local direction 1
2. `"ry"` relative rotation vector component in node 1 local direction 2
3. `"rz"` relative rotation vector component in node 1 local direction 3
4. `"wx"` relative angular velocity in node 1 local direction 1
5. `"wy"` relative angular velocity in node 1 local direction 2
6. `"wz"` relative angular velocity in node 1 local direction 3
7. `"Mx"` constraint reaction moment in node 1 local direction 1
8. `"My"` constraint reaction moment in node 1 local direction 2
9. `"Mz"` constraint reaction moment in node 1 local direction 3

In addition, the joint provides access to the private data provided by the constitutive law. They are accessed by prefixing the name of the data with the string `"constitutiveLaw."`; see the specific constitutive law description of the available data in Section 2.10.

Hints

When wrapped by a `driven` element, the `deformable hinge` joint honors the following hints:

- `hinge{1}` the orientation with respect to node 1 of the reference used to compute the angular strain is reset to the value resulting from the node 2 hinge orientation and the node 1 orientation

$$\tilde{\boldsymbol{R}}_{1h} = \boldsymbol{R}_1^T \boldsymbol{R}_2 \tilde{\boldsymbol{R}}_{2h}$$

- `hinge{2}` the orientation with respect to node 2 of the reference used to compute the angular strain is reset to the value resulting from the node 1 hinge orientation and the node 2 orientation

$$\tilde{\boldsymbol{R}}_{2h} = \boldsymbol{R}_2^T \boldsymbol{R}_1 \tilde{\boldsymbol{R}}_{1h}$$

- unrecognized hints are passed through to the constitutive law.

Output

In addition to the standard output, the `deformable joint` outputs:

- columns 14–16 (or 14–22): the angular strain in the selected orientation format (either three or nine columns, according to the selected `orientation description`, or to its default value, see Section 5.3.12).

If the constitutive law is either viscous or viscoelastic:

- columns 17–19 (or 23–25): the three components of the angular strain rate.

8.12.15 Deformable joint

This joint implements a configuration dependent force and moment that are exchanged by two points associated with two nodes with an offset. The force the and moment may depend, by way of a generic 6D constitutive law, on the relative position, orientation, velocity and angular velocity of the two points, expressed in the reference frame of node 1. It may be thought of as a combination of a `deformable displacement joint` (see Section 8.12.13) and of a `deformable hinge` (see Section 8.12.14); a significant difference is that the `deformable joint` uses a 6D `constitutive law` (see Section 2.10), so the force and the moment may both depend on the displacement and on the orientation.
The constitutive law is attached to the reference frame of node 1, so the sequence of the connections may matter in case of anisotropic constitutive laws, if the relative orientation of the two nodes changes during the analysis.
This element may add a considerable overhead because of the computation of the cross-coupling effects between the forces and moments and the relative positions and orientations; if the constitutive law does not couple relative displacements and orientations, a better choice consists in combining a `deformable hinge` and a `deformable displacement joint`.

```
<joint_type> ::= deformable joint

<joint_arglist> ::=
    <node_1_label> ,
        position , (Vec3) <relative_offset_1>
        [ , orientation ,(OrientationMatrix) <relative_orientation_matrix_1> ] ,
    <node_2_label> ,
        position , (Vec3) <relative_offset_2>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ] ,
    (ConstitutiveLaw<6D>) <const_law>
    [ , orientation description , <orientation_type> ]
```

The `orientation_type` is defined in Section 5.3.12.

Private Data

The following data are available:

1. `"dx"` relative displacement in node 1 local direction 1
2. `"dy"` relative displacement in node 1 local direction 2
3. `"dz"` relative displacement in node 1 local direction 3
4. `"rx"` relative rotation vector component in node 1 local direction 1
5. `"ry"` relative rotation vector component in node 1 local direction 2
6. `"rz"` relative rotation vector component in node 1 local direction 3
7. `"vx"` relative velocity in node 1 local direction 1
8. `"vy"` relative velocity in node 1 local direction 2
9. `"vz"` relative velocity in node 1 local direction 3
10. `"wx"` relative angular velocity in node 1 local direction 1
11. `"wy"` relative angular velocity in node 1 local direction 2
12. `"wz"` relative angular velocity in node 1 local direction 3
13. `"Fx"` constraint reaction force in node 1 local direction 1
14. `"Fy"` constraint reaction force in node 1 local direction 2
15. `"Fz"` constraint reaction force in node 1 local direction 3
16. `"Mx"` constraint reaction moment in node 1 local direction 1
17. `"My"` constraint reaction moment in node 1 local direction 2
18. `"Mz"` constraint reaction moment in node 1 local direction 3

In addition, the joint provides access to the private data provided by the constitutive law. They are accessed by prefixing the name of the data with the string `"constitutiveLaw."`; see the specific constitutive law description of the available data in Section 2.10.

Hints

When wrapped by a `driven` element, the `deformable joint` honors the same hints of the `deformable displacement joint`.

Output

In addition to the standard output, the `deformable joint` outputs:

- columns 14–16: the three components of the linear strain;
- columns 17–19 (or 17–25): the angular strain in the selected orientation format (either three or nine columns, according to the selected `orientation description`, or to its default value, see Section 5.3.12).

If the constitutive law is either viscous or viscoelastic:

- columns 20–22 (or 26–28): the three components of the linear strain rate;
- columns 23–25 (or 29–31): the three components of the angular strain rate.

8.12.16 Distance

This joint forces the distance between two points, each relative to a node, to assume the value indicated by the drive. If no offset is given, the points are coincident with the node themselves.

```
<joint_type> ::= distance

<joint_arglist> ::=
    <node_1_label> ,
    [ position , <relative_offset_1> , ]
    <node_2_label> ,
    [ position , <relative_offset_2> , ]
    { (DriveCaller) <distance> | from nodes }
```

The `relative_offset_*` are the distances of each end of the joint from the relative nodes in the node reference frame. The `distance` and the `distance with offset` joints do not allow null distance.

Note: in case the keyword `from nodes` is used, a constant drive caller is automatically instantiated for the `distance`. Its value is computed from the initial positions of the nodes; if any of the offsets is specified, the distance between the offset points is actually considered.

Current

When parsing the drive caller, the following values are made available through the `model::current` interface:

- `"L"`: the distance between the points connected by the joint, as it would be computed when `from nodes` is used.

For example, to smoothly change the distance from the initial value to a desired value:

```
set: const integer DISTANCE = 99;
set: const integer NODE1 = 1;
set: const integer NODE2 = 2;
set: const real DESIRED_DISTANCE = .33;
# ...
joint: DISTANCE, distance,
    NODE1, position, null,
    NODE2, position, null,
```

```
    cosine, 0., pi/.5,
        (DESIRED_DISTANCE - model::current("L"))/2,  # half (desired - initial)
        half,
        model::current("L");                         # start from initial
```

Output

The extra output is:

- the three components of unit vector representing the imposed distance direction in the global frame
- the norm of the imposed distance

Private Data

The following data are available:

1. `"d"` enforced distance

Hints

When wrapped by a `driven` element, the following hints are honored:

- unrecognized hints are passed through to the distance drive.

8.12.17 Distance with offset

This element has been deprecated in favor of the `distance` joint element, which now supports offsets. It may be dropped in future releases.

Private Data

See the `distance` joint element.

8.12.18 Drive displacement

This joint imposes the relative position between two points optionally offset from two structural nodes, in the form of a vector that expresses the direction of the displacement in the reference frame of node 1, whose amplitude is defined by a drive.

```
<joint_type> ::= drive displacement

<joint_arglist> ::=
    <node_1_label> , (Vec3) <offset_1> ,
    <node_2_label> , (Vec3) <offset_2> ,
    (TplDriveCaller<Vec3>) <relative_position>
```

This element is superseded by the `total joint`, see Section 8.12.48.

Private Data

The following data are available:

1. "dx" imposed displacement component along node 1 axis 1
2. "dy" imposed displacement component along node 1 axis 2
3. "dz" imposed displacement component along node 1 axis 3
4. "fx" reaction force component along node 1 axis 1
5. "fy" reaction force component along node 1 axis 2
6. "fz" reaction force component along node 1 axis 3

Hints

When wrapped by a driven element, the following hints are honored:

- offset{1} the offset of the joint with respect to node 1 is reset by pointing to the drive displacement point;
- offset{2} the offset of the joint with respect to node 2 is reset by pointing to the drive displacement point;
- unrecognized hints are passed through to the TplDriveCaller<Vec3>.

8.12.19 Drive displacement pin

This joint imposes the absolute position of a point optionally offset from a structural node, in the form of a vector that expresses the direction of the displacement in the absolute reference frame, whose amplitude is defined by a drive.

```
<joint_type> ::= drive displacement pin

<joint_arglist> ::=
    <node_label> , (Vec3) <node_offset> ,
    (Vec3) <offset> ,
    (TplDriveCaller<Vec3>) <position>
```

This element is superseded by the total pin joint, see Section 8.12.49.

Private Data

The following data are available:

1. "dx" imposed displacement component along absolute axis 1
2. "dy" imposed displacement component along absolute axis 2
3. "dz" imposed displacement component along absolute axis 3
4. "fx" reaction force component along absolute axis 1
5. "fy" reaction force component along absolute axis 2
6. "fz" reaction force component along absolute axis 3

Hints

When wrapped by a `driven` element, the following hints are honored:

- `offset{1}` the offset of the joint with respect to the node is reset by pointing to the driven displacement point;
- `offset{0}` the offset of the joint with respect to the absolute reference frame is reset by pointing to the driven displacement point;
- unrecognized hints are passed through to the `TplDriveCaller<Vec3>`.

8.12.20 Drive hinge

This joint imposes the relative orientation between two nodes, in the form of a rotation about an axis whose amplitude is defined by a drive.

```
<joint_type> ::= drive hinge

<joint_arglist> ::=
    <node_1_label>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_1> ] ,
    <node_2_label>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ] ,
    (TplDriveCaller<Vec3>) <hinge_orientation>
```

This element is superseded by the `total joint`, see Section 8.12.48.

Note: this element is experimental; now it is more reliable, but it is limited to $\|$`hinge_orientation`$\| < \pi$.

Private Data

The following data are available:

1. `"rx"` imposed relative rotation about node 1 hinge axis 1
2. `"ry"` imposed relative rotation about node 1 hinge axis 2
3. `"rz"` imposed relative rotation about node 1 hinge axis 3
4. `"Mx"` reaction moment about node 1 hinge axis 1
5. `"My"` reaction moment about node 1 hinge axis 2
6. `"Mz"` reaction moment about node 1 hinge axis 3

When wrapped by a `driven` element, the following hints are honored:

- `hinge{1}` the orientation with respect to node 1 of the reference in which the enforced orientation is expressed is reset;
- `hinge{2}` the orientation with respect to node 2 of the reference in which the enforced orientation is expressed is reset;
- unrecognized hints are passed through to the `TplDriveCaller<Vec3>`.

8.12.21 Gimbal hinge

This joint has not been implemented yet; use a `gimbal rotation` and a `spherical hinge` (see Section 8.12.45) to emulate.

8.12.22 Gimbal rotation

A homokinetic joint without position constraints; this joint, in conjunction with a `spherical hinge` (see Section 8.12.45) joint, should be used to implement an ideal tiltrotor gimbal instead of a `Cardano rotation` (see Section 8.12.8). See the technical manual and [37] for details. It is equivalent to a series of two Cardano's joints (the `Cardano hinge`, see Section 8.12.6) rotated 90 degrees apart, each accounting for half the relative rotation between axis 3 of each side of the joint.

```
<joint_type> ::= gimbal rotation

<joint_arglist> ::=
    <node_1_label>
    [ , orientation , (OrientationMatrix) <relative_orientation_matrix_1> ] ,
    <node_2_label>
    [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ]
    [ , orientation description , <orientation_type> ]
```

The `orientation_type` is defined in Section 5.3.12.

This joint allows nodes 1 and 2 to rotate about relative axes 1 and 2.

Output

The `gimbal rotation` joint outputs the three components of the reaction couple in the local frame (that of node 1) and in the global frame.

The extra columns 14 and 15 contain the angles ϑ and φ. The extra columns from 16 on contain the relative rotation between nodes 1 and 2, in the format determined either by an explicit selection of the orientation description, or by the default orientation description defined by the `default orientation` keyword (see Section 5.3.12).

The three components of the reaction couple in the local frame, and the angles ϑ and φ are also available as private data of the element under the names `lambda[<i>]`, with `i` = 1, 2, 3, `theta` and `phi`. For a description of the formulation and of the angles describe above, see the Technical Manual.

Private Data

The following data are available:

1. `"lambda[1]"` constraint reaction moment about node 1 local axis 1
2. `"lambda[2]"` constraint reaction moment about node 1 local axis 2
3. `"lambda[3]"` constraint reaction moment about node 1 local axis 3
4. `"theta"` relative angle ϑ
5. `"phi"` relative angle φ

8.12.23 Imposed displacement

This joint imposes the relative position between two points, optionally offset from two structural nodes, along a given direction that is rigidly attached to the first node. The amplitude of the displacement is defined by a drive.

```
<joint_type> ::= imposed displacement

<joint_arglist> ::=
    <node_1_label> , (Vec3) <offset_1> ,
    <node_2_label> , (Vec3) <offset_2> ,
    (Vec3) <direction> ,
    (DriveCaller) <relative_position>
```

This element is superseded by the `total joint`, see Section 8.12.48.

Private Data

The following data are available:

1. `"d"` imposed displacement along `direction`, in node 1 reference frame;
2. `"f"` reaction force along `direction`, in node 1 reference frame.

Hints

When wrapped by a `driven` element, the following hints are honored:

- `offset{1}` the offset of the joint with respect to node 1 is reset by pointing exactly to the imposed displacement point;
- `offset{2}` the offset of the joint with respect to node 2 is reset by pointing exactly to the imposed displacement point;
- unrecognized hints are passed through to the drive.

8.12.24 Imposed displacement pin

This joint imposes the absolute displacement of a point optionally offset from a structural node, along a direction defined in the absolute reference frame. The amplitude of the displacement is defined by a drive.

```
<joint_type> ::= imposed displacement pin

<joint_arglist> ::=
    <node_label> , (Vec3) <node_offset> ,
    (Vec3) <offset> ,
    (Vec3) <direction> ,
    (DriveCaller) <position>
```

This element is superseded by the `total pin joint`, see Section 8.12.49.

Private Data

The following data are available:

1. `"x"` imposed displacement along `direction`, in the absolute reference frame;
2. `"f"` reaction force along `direction`, in the absolute reference frame.

Hints

When wrapped by a `driven` element, the following hints are honored:

- `offset{1}` the offset of the joint with respect to the node is reset by pointing exactly to the imposed displacement point;
- `offset{0}` the offset of the joint with respect to the absolute reference frame is reset by pointing exactly to the imposed displacement point;
- unrecognized hints are passed through to the drive.

8.12.25 In line

This joint forces a point relative to the second node to move along a line attached to the first node.

```
<joint_type> ::= in line

<joint_arglist> ::=
    <node_1_label> ,
        [ position , ] (Vec3) <relative_line_position> ,
        [ orientation , ] (OrientationMatrix) <relative_orientation> ,
    <node_2_label>
        [ , offset , (Vec3) <relative_offset> ]
    [ , friction ,
        [ preload , <const_value> , ]
        <friction_model> ,
        <shape_function> ]
```

A point, optionally offset by `relative_offset` from the position of node `node_2_label`, slides along a line that passes through a point that is rigidly offset by `relative_line_position` from the position of `node_1_label`, and is directed as direction 3 of `relative_orientation`. Note that the `friction` keyword can currently only be used with the `in line` joint without `offset`. The joint is shown in Figure 8.9 where the origin and orientation of `node_1` are given by axis $1'$, $2'$, $3'$ and `relative_line_position` and `relative_orientation` are the transformations that yield axis $1'_0$, $2'_0$, $3'_0$ when applied to its coordinate axis. The origin and orientation of `node_2` are given by axis 1, 2, 3 and axis 1_0, 2_0, 3_0 is the result of applying the `relative_offset` translation to its coordinate axis.

Output

The output occurs in the `.jnt` file, according to default joint output for the first 13 columns; if friction is present the additional output is:

- column 14: the friction force along the line of the joint (applied in negative and positive directions for nodes 1 and 2, respectively);

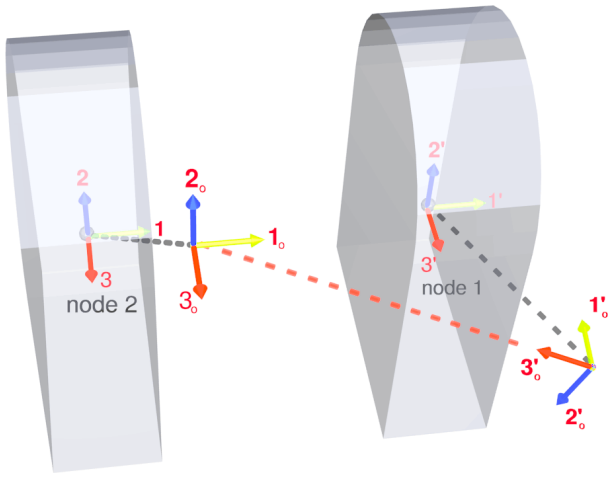


Figure 8.9: Inline joint.

- column 15: the friction coefficient;
- column 16: the relative sliding velocity;
- column 17: the relative sliding displacement.

8.12.26 In plane

This joint forces a point relative to the second node to move in a plane attached to the first node.

```
<joint_type> ::= in plane

<joint_arglist> ::=
    <node_1_label> ,
    [ position , ] (Vec3) <relative_plane_position> ,
    ((Unit)Vec3) <relative_normal_direction> ,
    <node_2_label>
    [ , offset , (Vec3) <relative_offset> ]
```

A point, optionally offset by `relative_offset` from the position of node `node_2_label`, slides on a plane that passes through a point that is rigidly offset by `relative_plane_position` from the position of `node_1_label`, and is perpendicular to `relative_normal_direction`. The vector `relative_normal_direction` is internally normalized to unity.

8.12.27 Invariant deformable displacement joint

Under development; right now, use the `deformable displacement joint` (see Section 8.12.13) instead.

8.12.28 Invariant deformable hinge

This (experimental) element is a variant of the `deformable hinge` (see Section 8.12.14); refer to that joint for the input syntax.
The *invariant* form of the `deformable hinge` joint refers the constitutive law to an orientation that is intermediate between those of the nodes it connects. As a result, the moment exchanged between the

two nodes is invariant to the sequence of definition of the nodes when anisotropic constitutive laws are used. It is worth stressing that determining the constitutive law for this element may be tricky, since usual measurement approaches directly measure forces and moments in a reference frame that is attached to one of the two ends of the component, so in practical cases it might be more appropriate to use the *variant* form of the joint, and consistently referring the constitutive law to the same end used to measure the mechanical properties of the component.

8.12.29 Linear acceleration

This joint imposes the absolute linear acceleration of a node along a given axis.

```
<joint_type> ::= linear acceleration

<joint_arglist> ::=
    <node_label> ,
    ((Unit)Vec3) <relative_direction> ,
    (DriveCaller) <acceleration>
```

The axis is `relative_direction`; it is internally normalized to unity.

Private Data

The following data are available:

1. `"F"` constraint reaction force along joint direction
2. `"a"` imposed acceleration along joint direction

8.12.30 Linear velocity

This joint imposes the absolute linear velocity of a node along a given axis.

```
<joint_type> ::= linear velocity

<joint_arglist> ::=
    <node_label> ,
    ((Unit)Vec3) <relative_direction> ,
    (DriveCaller) <velocity>
```

The axis is `relative_direction`; it is internally normalized to unity.

Private Data

The following data are available:

1. `"v"` imposed velocity along joint direction

8.12.31 Modal

Original implementation: Felice Felippone;
Initial review: Giuseppe Quaranta;
Current review: Pierangelo Masarati.

This joint implements a Component Mode Synthesis (CMS) deformable body. Its interface with the multibody domain is represented by clamps that constrain the multibody interface nodes to the position and orientation of the corresponding FEM nodes.

```
<joint_type> ::= modal

<joint_arglist> ::=
    { <reference_modal_node> | clamped
        [ , position , (Vec3) <absolute_position> ]
        [ , orientation , (OrientationMatrix) <absolute_orientation> ] } ,
    <mode_number> ,
    [ list , <mode> [ , ... ] , ]
    [ initial value ,
        { mode , <j> , <q_j> , <qP_j> [ , ... ]  # 1 <= j <= mode_number
            | <q_1> , <qP_1> [ , ... ] } , ]
    { <FEM_node_number> | from file } ,
    [ { no damping
        | damping from file
        | rayleigh damping , <mass_damping_coef> , <stiffness_damping_coef>
        | single factor damping , <damping_factor>
        | diag damping ,
            { all , <damping_factor> [ , ...]
            | <num_damped_modes> , <mode_damping> [ , ... ] } } , ]
    " <FEM_data_file> " ,
    [ [ { mass | damping | stiffness } ] threshold , <threshold> , ]
    [ { create binary | use binary | update binary }
        [ , ... ] , ]
    [ echo , " <echo_FEM_data_file> " [ , precision , <precision_digits> ] , ]
    [ use invariant 9 , ]
    [ { origin node , <origin_node> | origin position , (Vec3) <pos> } , ]
    <interface_nodes_number> ,
        [ output , { yes | no | (bool)<output_flag_for_all_interfaces> } , ]
        [ interface tolerance , <interface_tolerance> , ]
        <interface_node> [ , ... ]

<mode_damping> ::= <mode_index> , <mode_damping_factor>

<interface_node> ::=
    { " <FEM_node_label> "
            | (integer) <FEM_node_label>
            | find closest [ , interface tolerance , <interface_tolerance> ] } ,
        <multibody_label> , (Vec3) <offset_of_FEM_node>
            [ , output , { yes | no | (bool)<output_flag_for_this_interface> } ]
```

The `reference_modal_node` is a special dynamic structural node that is required to handle the rigid body motion of the modal joint. Its input is completely analogous to that of the `dynamic` structural nodes, see Section 6.5, only the keyword `dynamic` must be replaced by `modal`.

If no rigid body dynamics is required, e.g. if the modal element is clamped, the `clamped` option can be used, which allows to set the optional `absolute_position` and `absolute_orientation`. The former is the location, in the multibody global reference system, of the origin of the FEM reference system. The latter is the orientation, in the multibody global reference frame, of the FEM reference system. They default to zero and identity, which means that the global multibody reference frame and the FEM reference system are coincident.

The mode count in `mode_number` is not required to match the number of modes listed in the FEM

data file; if a lower number is given, only the first mode_number modes are used; moreover, a list of active modes can be given, to use non-consecutive modes; e.g., to use modes 1, 2, 3 and 6 out of a set of 10 modes defined in the FEM data file, write

```
...,
4,                          # number of active modes
    list, 1, 2, 3, 6,       # list of active modes
...
```

By default, the origin of the FEM grid is placed either in the position of the modal node, with its orientation, or in the absolute position and orientation in case the modal element is clamped, unless one of the mutually exclusive origin node and origin position optional keywords is used. The origin node keyword defines what FEM node corresponds to the modal node in the multibody domain, or what FEM node corresponds to the optional absolute position and orientation in case of a clamped modal element. The origin position keyword defines where, in the FEM model reference frame, the modal node is placed, or the location that corresponds to the optional absolute position and orientation in case of a clamped modal element.

Note that having the modal node coincident with a FEM node does not guarantee that the two remain coincident during the analysis. This is only true if the modal shape displacements and rotations of the FEM node are identically zero (i.e. in case of *attached modes*). Otherwise, the FEM node initially coincident with the modal node departs from it as long as the modal coordinates (the mode shape multipliers) are non-zero.

The FEM labels can be strings made of any character, provided they do not contain blanks. The strings must be enclosed in double quotes. For legacy reasons, a label not enclosed in double quotes is accepted, provided it is an integer.

The list of matchings between FEM and multibody nodes needs special care. The offset_of_FEM_node field contains the distance of the FEM node from the respective multibody node; by default, this is expressed in the reference frame of the multibody node.

Each FEM node which needs to be exposed in the multibody domain must be in the list of "interface nodes", where it is paired with the corresponding multibody node. It is suggested that the latter be defined as a static node, unless additional mass needs to be attached to it in the multibody domain, since the multibody node will be "clamped" to the modal joint. Each interface_node can be defined using its FEM_node_label (the FEM node label, either a string or an integer), or using the special keyword find closest. In the latter case, the FEM node which is closest to the multibody node (including the offset) is looked up.

If any of mass threshold, damping threshold, stiffness threshold are given, matrix elements whose absolute value is lower than the threshold are discarded from the mass, damping and stiffness matrices, respectively. The keyword threshold is a catchall for all, meaning that the same threshold is used for all matrices.

The FEM_data_file can be generated by NASTRAN, following the procedure illustrated in Appendix A.3.

It is strongly recommended that constrained modal analysis be used for otherwise free bodies, with the statically determined constraint consisting of clamping the FEM node that will coincide with the node indicated as reference_modal_node, and using the origin node to make that point the origin of the FEM frame.

Note about reference frames: the coincidence constraint between multibody and FEM nodes is written between the local frame of the FEM node and the global frame of the multibody node. As such, the multibody nodes must be oriented as the modal node the modal joint refers to, if any, or as the reference orientation of the modal joint, if it is clamped.

Note about initial assembly: it is very important that multibody and FEM nodes at interfaces are given with a high degree of accuracy, because the initial assembly procedure of the modal element does

not behave very well (it's on the TODO list with a very low priority); as a consequence, pay very much attention to the input of these nodes, until more robust procedures are developed. One trick is to build models incrementally. *Note: offsets between FEM and multibody nodes should be avoided unless strictly required.*

Note about the FEM file: the format of the `FEM_data_file` is relatively straightforward; it is reported in Appendix A.1. Initial modal displacements and velocities can be added, if required, by manually editing the file; however this practice is discouraged, unless strictly required.

Note about large FEM files: using a very large `FEM_data_file` may require long time for reading and parsing ASCII floats. The keyword `use binary` instructs MBDyn to use a binary version of the `FEM_data_file`. This binary version is automatically generated by MBDyn if requested by means of the keyword `create binary`. The binary version of the `FEM_data_file` is used only if its timestamp is more recent than that of the ASCII version. The keyword `update binary` instructs MBDyn to regenerate the binary version when the ASCII version is more recent.

Note about structural damping: information about structural damping can be provided by specifying any of the `no damping`, `damping from file`, `rayleigh damping`, `single factor damping`, or `diag damping` keywords.

- `no damping`: instructs the solver that no damping must be used.
- `damping from file`: the solver expects a(n optional) generalized damping matrix in the database (as `RECORD GROUP 13`); such matrix is used when `damping from file` is specified, while it is ignored and overridden by any of the other specified forms of damping provided in the specification of the modal joint element. The generalized damping matrix appeared in MBDyn 1.5.3;
- `rayleigh damping`: it is followed by `mass_damping_coef` and `stiffness_damping_coef`, which are used to compute the damping matrix as

$$\mathbf{C} = \texttt{mass\_damping\_coef} \cdot \mathbf{M} + \texttt{stiffness\_damping\_coef} \cdot \mathbf{K} \tag{8.72}$$

  It appeared in MBDyn 1.7.1.
- `single factor damping`: it is followed by the damping factor $\xi = \texttt{damping\_factor}$ that is applied to all modes. It appeared in MBDyn 1.7.1.
- `diag damping`: it is followed by the number of damped modes `num_damped_modes`, or by the keyword `all`. If `all` is used, as many damping factors as the available modes are expected. Otherwise, a list of `mode_damping` values follows. Each occurrence of `mode_damping` is made of the mode's index `mode_index` and the related damping value `damping_factor`.

In the last two cases, a damping factor is required. The corresponding damping for the i-th mode is computed as

$$c_i = 2\,\texttt{damping\_factor}_i\,\sqrt{k_i m_i}$$

where $\texttt{damping\_factor}_i$ is the value provided for the list of values provided for `diag damping` (for each mode). For example, a factor of 0.01 means 1% damping.

In previous versions, a (misleading) `proportional damping` form was present, which is now deprecated and replaced by `single factor damping`.

Output

Up to MBDyn 1.2.6, output occurred in a specific file that needed to be mandatorily given as the last argument to each modal element definition.

Now output occurs in a `.mod` file, which contains, for each time step, as many rows as all the modes of all the modal elements whose output is enabled. Each row is structured as follows:

- a field containing the label of the modal joint and the number of the mode, separated by a dot (e.g. `label.mode`)
- the value of the modal unknown
- the value of the first derivative of the modal unknown
- the value of the second derivative of the modal unknown

Note: the number of the mode in the first field of the output, after the dot, is the ordinal of the mode in the FEM data file. If only a portion of the modes was selected using the `list` keyword, the mode numbers will reflect the selected modes. For example, if the `modal` joint was defined as

```
set: MODAL_ELEM = 99;
set: MODAL_NODE = 101;
joint: MODAL_ELEM, modal, MODAL_NODE,
    3, list, 1, 7, 9, # only use modes 1, 7 and 9 of a larger basis
    from file,
    "modal.fem",
    0;                # no FEM nodes
```

the output will look like

```
99.1 0.000000e+00 0.000000e+00 0.000000e+00
99.7 0.000000e+00 0.000000e+00 0.000000e+00
99.9 0.000000e+00 0.000000e+00 0.000000e+00
```

The global motion of the n-th point on the FEM mesh is given by the formula

$$\boldsymbol{x}_n = \boldsymbol{x}_{\text{modal}} + \mathbf{R}_{\text{modal}} \left(\boldsymbol{f}_{0_n} + \mathbf{f}_{1_n} \boldsymbol{q} \right) \tag{8.73}$$

where $\boldsymbol{x}_{\text{modal}}$ and $\mathbf{R}_{\text{modal}}$ are the position vector and the orientation matrix of the modal node, if any, $\boldsymbol{f}_{0_n}$ is the position of the n-th point in the floating frame of reference, $\boldsymbol{f}_{1_n}$ is the matrix of the shapes that express the displacement on the n-th point in the floating frame of reference, and $\boldsymbol{q}$ are the modal coordinates, i.e. the data in column 2 of the modal output file. In the Private Data, the i-th component of the position of the n-th node is given by `"x[<n>,<i>]"` (see the related section for details).

Private Data

The following data are available:

1. `"q[<mi>]"` the value of the mode amplitude indicated by `mi`
2. `"qP[<mi>]"` the value of the mode amplitude derivative indicated by `mi`
3. `"qPP[<mi>]"` the value of the mode amplitude second derivative indicated by `mi`
4. `"x[<n>,<i>]"` FEM node `n` position `i` component value
5. `"xP[<n>,<i>]"` FEM node `n` velocity `i` component value
6. `"xPP[<n>,<i>]"` FEM node `n` acceleration `i` component value
7. `"w[<n>,<i>]"` FEM node `n` angular velocity `i` component value
8. `"wP[<n>,<i>]"` FEM node `n` angular acceleration `i` component value

When the mode indicator mi is a number <m>, then m represents the name of mode; when mi is the character '#' followed by a number <m>, then m is the index of the mode. The two cases may differ when only a subset of the modes defined in the FEM data file are actually used. For example, if mode_number == 3 and in the joint definition the syntax "list, 1, 7, 9" was used, to indicate that of the three modes to be used in the analysis the first mode is the first one in the database, whereas the second mode is the seventh one in the database, and the third mode is the ninth one in the database, then "q[1]" is equivalent to "q[#1]", whereas "q[7]" is equivalent to "q[#2]", and "q[9]" is equivalent to "q[#3]", and possibly more readable.

"FEM" node means that the label of the FEM node must be used; this allows access also to the motion of nodes that are not attached to a multibody node. Such motion is intended as expressed in the global reference frame. No information on FEM node orientation is currently available. For example, "x[1001,1]" provides access to the x component of the position of node 1001; "xPP[N21,3]" provides access to the z component of the acceleration of node N21.

8.12.32 Offset displacement joint

Author: Reinhard Resch

This element is a rigid body coupling constraint similar to NASTRAN's RBE2 element, but it may be used also in the context of large deformations and large rotations. It is needed mainly to provide an interface between solid or membrane elements, which are using structural displacement nodes with only three degrees of freedom, and the large family of elements (e.g. joints, forces, ...) which can be used only for structural nodes with six degrees of freedom.

```
<joint_type> ::= offset displacement joint

<joint_arglist> ::=
   (StructNode) <node_label_1> ,
   (StructDispNode) <node_label_2>
   [ , position , <position_node_2> ]
```

The position of the node_label_2 is constrained in a way, that it's following exactly the rigid body motion of node_label_1. So, a rigid connection is created.

Output

The output occurs in the .jnt file, which contains the reaction force and reaction couple related to the node_label_1.

Private Data

The following data is available:

1. "Fx" reaction force at node_label_1 along node_label_1 axis 1
2. "Fy" reaction force at node_label_1 along node_label_1 axis 2
3. "Fz" reaction force at node_label_1 along node_label_1 axis 3
4. "Mx" reaction moment at node_label_1 along node_label_1 axis 1
5. "My" reaction moment at node_label_1 along node_label_1 axis 2
6. "Mz" reaction moment at node_label_1 along node_label_1 axis 3

7. "fx" reaction force at node_label_1 along global axis 1
8. "fy" reaction force at node_label_1 along global axis 2
9. "fz" reaction force at node_label_1 along global axis 3
10. "mx" reaction moment at node_label_1 along global axis 1
11. "my" reaction moment at node_label_1 along global axis 2
12. "mz" reaction moment at node_label_1 along global axis 3

Example. This example mimics an instance of NASTRAN's RBE2 element.

```
begin: nodes;
  structural: 609, static, reference, ref_id_solid, null,
                           reference, ref_id_solid, eye,
                           reference, ref_id_solid, null,
                           reference, ref_id_solid, null;
  structural: 1, dynamic displacement,
                           reference, ref_id_solid, 5.1e+02,  1.0e+01, -1.0e+01,
                           reference, ref_id_solid, null;
  structural: 2, dynamic displacement,
                           reference, ref_id_solid, 5.1e+02,  -1.0e+01, -1.0e+01,
                           reference, ref_id_solid, null;
  ...
  structural: 8, dynamic displacement,
                           reference, ref_id_solid, 9.9e+02,  -1.0e+01, 1.0e+01,
                           reference, ref_id_solid, null;
end: nodes;
begin: elements;
  joint: 1, offset displacement joint, 609, 5;
  joint: 2, offset displacement joint, 609, 6;
  joint: 3, offset displacement joint, 609, 7;
  joint: 4, offset displacement joint, 609, 9;
  joint: 5, offset displacement joint, 609, 13;
  joint: 6, offset displacement joint, 609, 14;
  joint: 7, offset displacement joint, 609, 15;
  joint: 8, offset displacement joint, 609, 16;
end: elements;
```

8.12.33 Plane displacement

This joint allows two nodes to move in the common relative 1–2 plane and to rotate about the common relative axis 3.

```
<joint_type> ::= plane displacement

<joint_arglist> ::=
    <node_1_label> ,
        position , (Vec3) <relative_offset_1>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_1> ] ,
```

```
    <node_2_label> ,
        position , (Vec3) <relative_offset_2>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ]
```

Note: this element is temporarily disabled; combine an `in plane` and a `revolute rotation` (see Section 8.12.40) joint, or use the `total joint`, see Section 8.12.48.

8.12.34 Plane displacement pin

This joint allows a node to move in the relative 1–2 plane and to rotate about the relative axis 3 with respect to an absolute point and plane. See also Section 8.12.33.

```
<joint_type> ::= plane displacement pin

<joint_arglist> ::=
    <node_label> ,
        position , (Vec3) <relative_offset>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix> ] ,
    position , (Vec3) <absolute_pin_position>
    [ , orientation , (OrientationMatrix) <absolute_pin_orientation_matrix> ]
```

Note: this element is temporarily disabled; combine an `in plane` and a `revolute rotation` (see Section 8.12.40) joint instead, using a grounded node, or use a `total pin joint`, see Section 8.12.49.

8.12.35 Plane hinge

This joint has been renamed `revolute hinge` (see Section 8.12.38); the old name has been deprecated and its support may be discontinued in future versions.

8.12.36 Plane pin

This joint has been renamed `revolute pin` (see Section 8.12.39); the old name has been deprecated and its support may be discontinued in future versions.

8.12.37 Prismatic

This joints constrains the relative orientation of two nodes, so that their orientations remain parallel. The relative position is not constrained. The initial orientation of the joint must be compatible: use the `orientation` keyword to assign the joint initial orientation.

```
<joint_type> ::= prismatic

<joint_arglist> ::=
    <node_1_label>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_1> ] ,
    <node_2_label>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ] ,
```

Hints

When wrapped by a `driven` element, the following hints are honored:

- `hinge{1}` the relative orientation of the joint with respect to node 1 is reset;

- `hinge{2}` the relative orientation of the joint with respect to node 2 is reset;

8.12.38 Revolute hinge

This joint only allows the relative rotation of two nodes about a given axis, which is axis 3 in the reference systems defined by the two `orientation` statements.

```
<joint_type> ::= revolute hinge

<joint_arglist> ::=
    <node_1_label> ,
        position , (Vec3) <relative_offset_1>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_1> ] ,
    <node_2_label> ,
        position , (Vec3) <relative_offset_2>
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ]
    [ , initial theta , <initial_theta> ]
    [ , friction , <average_radius> ,
        [ preload , <const_value> , ]
        <friction_model> ,
        <shape_function>
        [ , reaction force components, { full | axial | normal | preload only } ]
    ]
```

The joint is shown in Figure 8.10 where the axes shown are the those obtained after having applied the `relative_offset` and `relative_orientation_matrix` transformations to the coordinate axes of `node_1` and `node_2`.

Note: this element can be thought of as the combination of a `spherical hinge` (see Section 8.12.45), that constrains the three components of the relative position of the nodes, and of a `revolute rotation` (see Section 8.12.40), that constrains the relative orientation of the nodes so that only rotation about a common axis is allowed.

Rationale

The rationale for having two statements to indicate the position and orientation of the same entity is that the joint is supposed to constrain the position and orientation of two points, each attached to a node. This is what is typically known from the geometry of the components of a mechanism.

Friction

If `friction` is specified then

- the `average_radius` r is used to compute the friction-induced moment about the joint's axis as $M = frF_f$, where f is the friction coefficient, computed from the 1-D `friction_model` and `shape_function`, and F_f is a force, defined in the following;
- `preload` is a preload force used to account for initial interference of the joint; even if there is no preload in the real joint, one should set it to a small number $\neq 0$ [FIXME: should preload be > 0 ? In that case, should we check for negative (or null) values? Should it be a drive caller?] in order to deal with ill-conditioning of the friction computation occurring whenever the real joint reaction force is almost null;

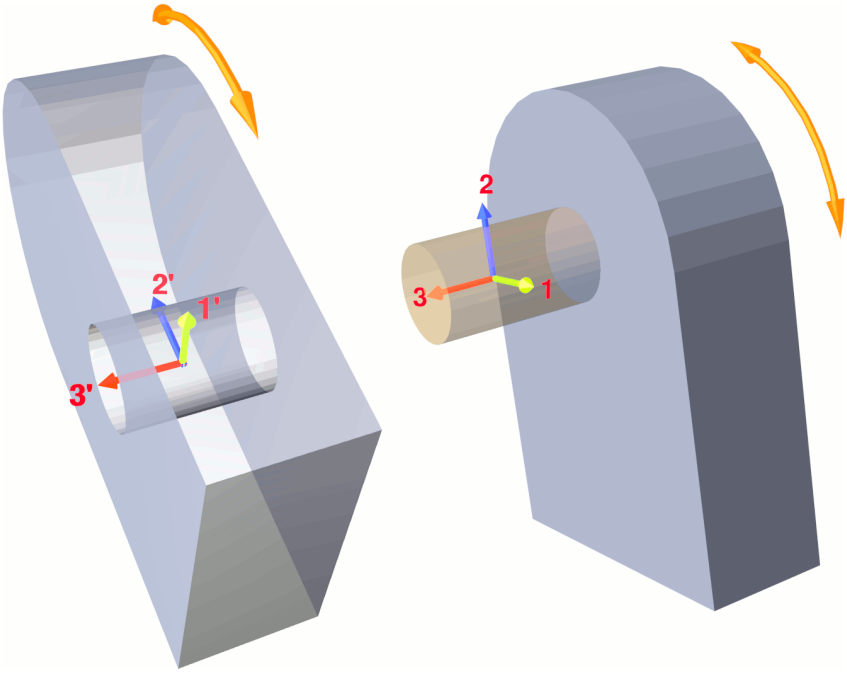


Figure 8.10: Revolute hinge.

- $F_f = \max(||\boldsymbol{F}_r||,$ `preload`$)$, where $\boldsymbol{F}_r$ is some portion of the joint reaction force, specified by `reaction components`
- `reaction force components` specifies which components of the reaction force should be used to compute $\boldsymbol{F}_r$; more specifically
  - `full` (default behavior): use the whole reaction force (without friction) vector; friction will contribute not only to the torsional moment, but also to the overall force transmitted by the joint to the constrained nodes;
  - `axial`: use the component along the joint axis of the reaction force (without friction); friction will contribute only to the torsional moment and not to the overall force transmitted by the joint to the constrained nodes, but will depend on the axial component of the reaction force;
  - `normal`: use the component normal to the joint axis of the reaction force (without friction); friction will contribute not only to the torsional moment, but also to the overall force transmitted by the joint to the constrained nodes;
  - `preload only`: assume that the friction comes only from the preload, thus $\boldsymbol{F}_r = \boldsymbol{0}$, and friction will only contribute to the torsional moment, not to the overall force transmitted by the joint to the constrained nodes, and will not depend on the joint reaction force.

Output

The output occurs in the `.jnt` file, according to default joint output for the first 13 columns; specific columns contain:

- columns 14–16: the so-called *Euler* angles (in degrees) that describe the relative rotation; only the third component is relevant, the first two essentially indicate the accuracy of the rotation constraint;

- column 17–19: the relative angular velocity; only the third component is relevant, the first two are zero.

If friction is present:

- column 20: the friction moment about the revolute axis;
- subsequent columns: any friction model specific data (quite possibly the friction coefficient).

Private Data

The following data are available:

1. `"rz"` relative rotation angle about revolute axis
2. `"wz"` relative angular velocity about revolute axis
3. `"Fx"` constraint reaction force in node 1 local direction 1
4. `"Fy"` constraint reaction force in node 1 local direction 2
5. `"Fz"` constraint reaction force in node 1 local direction 3
6. `"Mx"` constraint reaction moment about node 1 local direction 1
7. `"My"` constraint reaction moment about node 1 local direction 2
8. `"Mz"` constraint reaction moment about node 1 local direction 3

Hints

When wrapped by a `driven` element, the following hints are honored:

- `hinge{1}` the relative orientation of the joint with respect to node 1 is reset;
- `hinge{2}` the relative orientation of the joint with respect to node 2 is reset;
- `offset{1}` the offset of the joint with respect to node 1 is reset;
- `offset{2}` the offset of the joint with respect to node 2 is reset;
- unrecognized hints are passed through to the friction model, if any.

8.12.39 Revolute pin

This joint only allows the absolute rotation of a node about a given axis, which is axis 3 in the reference systems defined by the two `orientation` statements.

```
<joint_type> ::= revolute pin

<joint_arglist> ::=
    <node_label> ,
        position, (Vec3) <relative_offset>
        [ , orientation, (OrientationMatrix) <relative_orientation_matrix> ] ,
    position , (Vec3) <absolute_pin_position>
    [ , orientation , (OrientationMatrix) <absolute_pin_orientation_matrix> ]
    [ , initial theta , <initial_theta> ]
```

Note: this is equivalent to a `revolute hinge` (see Section 8.12.38) when one node is grounded.

Private Data

The following data are available:

1. `"rz"` relative rotation angle about revolute axis
2. `"wz"` relative angular velocity about revolute axis
3. `"Fx"` constraint reaction force in node 1 local direction 1
4. `"Fy"` constraint reaction force in node 1 local direction 2
5. `"Fz"` constraint reaction force in node 1 local direction 3
6. `"Mx"` constraint reaction moment about node 1 local direction 1
7. `"My"` constraint reaction moment about node 1 local direction 2
8. `"Mz"` constraint reaction moment about node 1 local direction 3

Hints

When wrapped by a `driven` element, the following hints are honored:

- `hinge{1}` the relative orientation of the joint with respect to the node is reset;
- `hinge{0}` the relative orientation of the joint with respect to the absolute frame is reset;
- `offset{1}` the offset of the joint with respect to the node is reset;
- `offset{0}` the offset of the joint with respect to the absolute frame is reset;

8.12.40 Revolute rotation

This joint allows the relative rotation of two nodes about a given axis, which is axis 3 in the reference systems defined by the two `orientation` statements. The relative position is not constrained.

```
<joint_type> ::= revolute rotation

<joint_arglist> ::=
    <node_1_label>
        [ , position , (Vec3) <relative_offset_1> ]
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_1> ] ,
    <node_2_label>
        [ , position , (Vec3) <relative_offset_2> ]
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ]
```

Rationale

A revolute joint without position constraints; this joint, in conjunction with an `inline` joint, should be used to constrain, for example, the two nodes of a hydraulic actuator.

Private Data

The following data are available:

1. "rz" relative rotation angle about revolute axis
2. "wz" relative angular velocity about revolute axis
3. "Mx" constraint reaction moment about node 1 local direction 1
4. "My" constraint reaction moment about node 1 local direction 2
5. "Mz" constraint reaction moment about node 1 local direction 3

Hints

When wrapped by a driven element, the following hints are honored:

- hinge{1} the relative orientation of the joint with respect to node 1 is reset;
- hinge{2} the relative orientation of the joint with respect to node 2 is reset;

8.12.41 Rigid body displacement joint

Author: Reinhard Resch

This element is a load distributing coupling constraint similar to NASTRAN's RBE3 element. However, it may be used also in the context of large deformations and large rotations. It is intended mainly to provide an interface between solid elements, which are using structural displacement nodes with only three degrees of freedom, and the large family of elements (e.g. joints, forces, ...) which can be used only for structural nodes with six degrees of freedom.

```
<joint_type> ::= rigid body displacement joint

<joint_arglist> ::=
   (StructNode) <node_label_master> ,
   (integer) <slave_node_number> ,
   <slave_node_data>
   [ , alpha, (real) <alpha> ] ;

<slave_node_data> :: =
   <one_slave_node_1>
   [ , ... ]
   [ , <one_slave_node_N> ];

<one_slave_node_i> :: =
   (StructDispNode) <node_label_slave_i>
   [ , position , (Vec3) <position_slave_node_i> ]
   [ , weight , (real) <weight_slave_node_i> ]
```

Kinematics The position and orientation of the node_label_master is constrained in a way, that it's following the weighted average of the rigid body motion of a set of slave_node_number slave nodes. But, no artificial stiffness will be added to the slave nodes. Since the slave nodes do not have any rotational degrees of freedom, a minimum number of three non coincident slave nodes is required. If the optional parameter position_slave_node is present, it is used to compute the relative offset between node_label_slave and node_label_master. Otherwise, the initial position of node_label_slave is used to compute the offset.

Corner nodes of quadratic elements There is also an optional weighting factor weight_slave_node per node, which may be used for second order solid elements in order to achieve a smooth stress distribution. Negative values for weight_slave_node are allowed (e.g. for corner nodes of quadratic hexahedrons), but the sum of all weighting factors must be always greater than zero. See also section 8.16.10 for an example on how to create a rigid body displacement joint automatically from a Finite Element mesh.

Scaling The optional parameter alpha may be used to scale the constraint equations in order to improve the condition number of the Jacobian matrix. However, it has no effect on the result and the default value should be fine for most applications.

Output

The output occurs in the .jnt file, which contains the reaction force and reaction couple related to the node_label_master.

Private Data

The following data is available:

1. "Fx" reaction force at node_label_master along node_label_master axis 1
2. "Fy" reaction force at node_label_master along node_label_master axis 2
3. "Fz" reaction force at node_label_master along node_label_master axis 3
4. "Mx" reaction moment at node_label_master along node_label_master axis 1
5. "My" reaction moment at node_label_master along node_label_master axis 2
6. "Mz" reaction moment at node_label_master along node_label_master axis 3
7. "fx" reaction force at node_label_master along global axis 1
8. "fy" reaction force at node_label_master along global axis 2
9. "fz" reaction force at node_label_master along global axis 3
10. "mx" reaction moment at node_label_master along global axis 1
11. "my" reaction moment at node_label_master along global axis 2
12. "mz" reaction moment at node_label_master along global axis 3

Example. This example mimics an instance of NASTRAN's RBE3 element.

```
begin: nodes;
  structural: 609, static, reference, ref_id_solid, null,
                           reference, ref_id_solid, eye,
                           reference, ref_id_solid, null,
                           reference, ref_id_solid, null;
  structural: 1, dynamic displacement,
                           reference, ref_id_solid, 5.1e+02,  1.0e+01, -1.0e+01,
                           reference, ref_id_solid, null;
  structural: 2, dynamic displacement,
                           reference, ref_id_solid, 5.1e+02,  -1.0e+01, -1.0e+01,
                           reference, ref_id_solid, null;
  ...
  structural: 8, dynamic displacement,
                           reference, ref_id_solid, 9.9e+02,  -1.0e+01, 1.0e+01,
                           reference, ref_id_solid, null;
end: nodes;
begin: elements;
   joint: 100, rigid body displacement joint,
          609, ## master node label
            8, ## eight slave nodes
            5, weight, 1.3333333333333348e+02, ## midside nodes
            6, weight, 1.3333333333333351e+02,
            7, weight, 1.3333333333333351e+02,
            9, weight, 1.3333333333333351e+02,
           13, weight, -3.3333333333333414e+01, ## corner nodes
           14, weight, -3.3333333333333428e+01,
           15, weight, -3.3333333333333421e+01,
           16, weight, -3.3333333333333421e+01;
 end: elements;
```

8.12.42 Rod

The rod element represents a force between two nodes that depends on the relative position and velocity of two points, each rigidly attached to a structural node with an optional offset. The direction of the force is also based on the relative position of the points: it is the line that passes through them. If no offset is defined, the points are the nodes themselves.

The syntax is:

```
<joint_type> ::= rod

<joint_arglist> ::=
    <node_1_label> ,
        [ position , (Vec3) <relative_offset_1> , ]
    <node_2_label> ,
        [ position , (Vec3) <relative_offset_2> , ]
    { (real) <rod_length> | from nodes }
    (ConstitutiveLaw<1D>) <const_law>
```

rod_length is the distance between the points connected by the rod that represents the reference length

of the rod (see ℓ_0 below). When the form `from nodes` is used, such distance is computed considering the points connected by the rods, including the offsets if defined.

Note on the use of the constitutive law

The constitutive law `const_law` receives as input the rod axial strain

$$\varepsilon = \frac{\ell - \ell_0}{\ell_0} = \frac{\ell}{\ell_0} - 1, \tag{8.74}$$

where $l_0 =$ `rod_length` and l is the distance between the two points; in case of viscous or viscoelastic constitutive law, the axial strain derivative

$$\dot{\varepsilon} = \frac{\dot{\ell}}{\ell_0} \tag{8.75}$$

is also used. It returns the corresponding rod axial force,

$$F = F\left(\frac{\ell - \ell_0}{\ell_0}, \frac{\dot{\ell}}{\ell_0}\right).$$

If a `prestrain` ε_p is defined, it consists in an imposed value of the axial strain that is subtracted from the geometrical strain before being passed to the constitutive law; if a `prestress` F_p is defined, it consists in an imposed axial force value that is added to the one obtained from the constitutive law, namely

$$F = F_p + F\left(\frac{\ell - \ell_0}{l_0} - \varepsilon_p, \frac{\dot{\ell}}{\ell_0}\right).$$

For example, a linear elastic constitutive law for the rod element is defined as

$$F = EA\frac{\ell - \ell_0}{\ell_0},$$

where EA is the axial stiffness of the rod. The corresponding syntax would be:

```
set: const real L_0 = 0.35;  # reference length
set: const real EA = 7.0e+6; # axial stiffness
joint: 99, rod,
    1, position, null,
    2, position, null,
    L_0,
    linear elastic, EA;
```

To model a lumped spring of stiffness K, such that $F = K(\ell - \ell_0)$ and thus $EA = K\ell_0$, one needs to use

```
set: const real L_0 = 0.35;  # reference length
set: const real K = 2.0e+7;  # lumped spring stiffness
joint: 99, rod,
    1, position, null,
    2, position, null,
    L_0,
    linear elastic, K*L_0;
```

Similar considerations apply to the damping characteristic of a viscoelastic constitutive law:

```
set: const real L_0 = 0.35;  # reference length
set: const real K = 2.0e+7;  # lumped spring stiffness
set: const real C = 2.0e+2;  # lumped damper characteristic
joint: 99, rod,
    1, position, null,
    2, position, null,
    L_0,
    linear viscoelastic, K*L_0, C*L_0;
```

For further details on the supported constitutive laws, see Section 2.10.

Use of "current"

When parsing the constitutive law, the following run-time computed values are made available through the `model::current(<string>)` model namespace interface:

- `string ::= "L0"`: the length of the rod `rod_length` as input by the user, or computed from the distance between the points connected by the rod when `from nodes` is used;
- `string ::= "L"`: the length of the rod computed from the distance between the points connected by the rod; identical to `"L0"` when `from nodes` is used.

For example, if one knows the value of K and wants the axial stiffness in a linear elastic constitutive law to reflect its dependence on the reference length of the rod as computed from the initial configuration, the following syntax does the trick:

```
set: const real K = 2.e+7;
# ...
joint: 99, rod,
    1, position, null,
    2, position, null,
    from nodes,                              # compute length from nodes
    linear elastic, K*model::current("L0"); # compute EA as K*rod_length
```

Output

The output occurs in the `.jnt` file, which contains:

- the label
- the three components of the internal force in the reference frame of the element (column 2, component x, contains the force)
- the three components of the internal moment in the reference frame of the element (always zero)
- the three components of the internal force in the global reference frame
- the three components of the internal moment in the global reference frame (always zero)
- the length of the rod
- the three components of the rod axis in the global reference frame (a unit vector)
- the length rate of the rod (derivative of length wrt/ time)
- optional data appended by the constitutive law

Private Data

The following data are available:

1. "F" force between the two nodes/connection points
2. "l" distance between the two nodes/connection points
3. "lP" distance rate between the two nodes/connection points

The rod joint can also give access the private data provided by the constitutive law. It is accessed by prefixing the name of the data with the string "constitutiveLaw."; see the specific constitutive law description of the available data in Section 2.10.

8.12.43 Rod with offset

The syntax is:

```
<joint_type> ::= rod with offset

<joint_arglist> ::=
    <node_1_label> ,
        (Vec3) <relative_offset_1> ,
    <node_2_label> ,
        (Vec3) <relative_offset_2> ,
    { (real) <rod_length> | from nodes } ,
    (ConstitutiveLaw<1D>) <const_law>
```

Analogous to the rod joint with the optional offsets; see Section 8.12.42 for details.

8.12.44 Screw joint

This joint models a screw, aligned with axis 3 defined by the first node orientation statement. Friction, if declared, can prevent the relative motion.

```
<joint_type> ::= screw

<joint_arglist> ::=
    <node_1_label> ,
        [ position , (Vec3) <relative_offset_1> , ]
        [ orientation , (OrientationMatrix) <relative_orientation_matrix_1> , ]
    <node_2_label> ,
        [ offset , (Vec3) <relative_offset_2> , ]
    pitch , <screw_pitch_angle>
    [ , friction ,
        <friction_model> ,
        <shape_function> ]
```

The shape function should be a screw joint shape function.

Output

The output occurs in the `.jnt` file, according to default joint output for the first 13 columns; specific columns contain:

- column 14: the overall relative rotation angle between the two nodes, in degrees, along the prescribed screw direction,
- columns 15–17: the relative rotation vector ax($\log(\boldsymbol{R}_1^T \boldsymbol{R}_2)$);

If friction is present:

- column 18: the relative velocity v along the friction force direction;
- column 19: the friction-induced moment value about the screw axis;
- subsequent columns: any friction model specific data (quite possibly the friction coefficient).

8.12.45 Spherical hinge

This joint constrains the relative position of two nodes; the relative orientation is not constrained.

```
<joint_type> ::= spherical hinge

<joint_arglist> ::=
    <node_1_label> ,
        [ position , (Vec3) <relative_offset_1> , ]
        [ orientation , (OrientationMatrix) <relative_orientation_matrix_1> , ]
    <node_2_label>
        [ , position , (Vec3) <relative_offset_2> ]
        [ , orientation , (OrientationMatrix) <relative_orientation_matrix_2> ]
    [ , friction ,
        <friction_model_2D> ,
        <shape_function_2D> ]
```

The joint is shown in Figure 8.11 where the axes shown are the those obtained after having applied the `relative_offset` and `relative_orientation_matrix` transformations to the coordinates of `node_1` and `node_2`. Note: the orientation matrix, set by means of the `orientation` keyword, is used for output purposes only. A default identity matrix is assumed.

Hints

When wrapped by a `driven` element, the following hints are honored:

- `offset{1}` the offset of the joint with respect to node 1 is reset;
- `offset{2}` the offset of the joint with respect to node 2 is reset;

Output

The output occurs in the .jnt file, according to default joint output for the first 13 columns; specific columns contain:

- columns 14—16: the relative rotation vector (if the input file prescribe `orientation matrix` output for the orientation then these would be columns 15–23, and the following columns would be shifted accordingly);

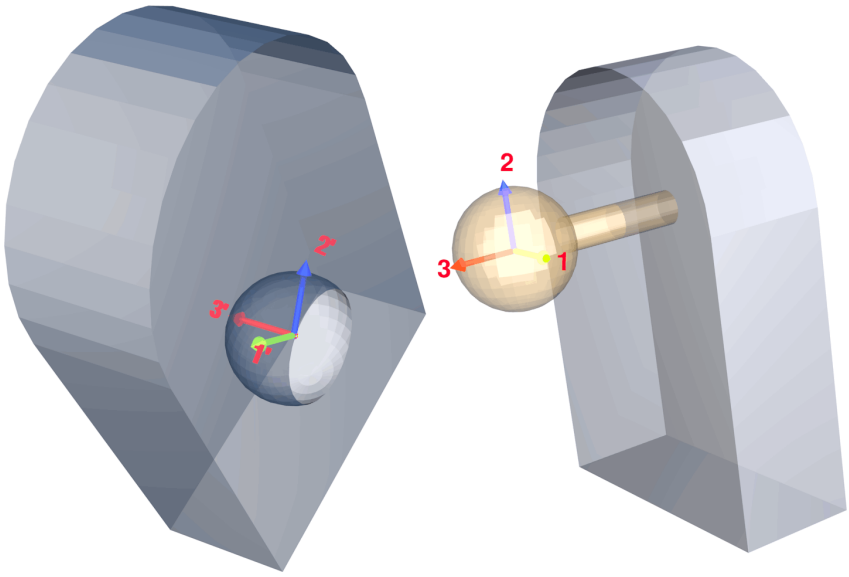


Figure 8.11: Spherical hinge.

If friction is present, the output has some additional fields:

- columns 17–19: frictional moment;
- columns 20–22: direction $\boldsymbol{n}$ of the normal (no friction) reaction force;
- columns 23–25: direction $\boldsymbol{t}_1$ of the frictional force;
- columns 26–28: direction $\boldsymbol{t}_2$ of the frictional force;
- column 29: friction coefficient f_1 in direction $\boldsymbol{t}_1$;
- column 30: friction coefficient f_2 in direction $\boldsymbol{t}_2$;
- columns 31–33: normal (no friction) reaction force $\boldsymbol{F}$;
- columns 34–36: friction reaction force $\boldsymbol{F}_1$ in direction $\boldsymbol{t}_1$;
- columns 37–39: friction reaction force $\boldsymbol{F}_2$ in direction $\boldsymbol{t}_2$;

8.12.46 Spherical pin

This joint constrains the absolute position of a node; the relative orientation is not constrained.

```
<joint_type> ::= spherical pin

<joint_arglist> ::=
    <node_label> ,
        [ position , (Vec3) <relative_offset> , ]
        [ orientation , (OrientationMatrix) <relative_orientation> , ]
    position , (Vec3) <absolute_pin_position>
    [ , orientation , <absolute_orientation> ]
```

Note: this is equivalent to a spherical hinge (see Section 8.12.45) when one node is grounded. An alternative way to model a grounded spherical hinge requires the use of another node, clamped by a clamp joint.

8.12.47 Total equation

Original implementation: Alessandro Fumagalli, Marco Morandini.

See also total reaction (see Section 8.12.50). See [38]. TODO

8.12.48 Total joint

Original implementation: Alessandro Fumagalli;
Review: Pierangelo Masarati.

See [39]. This element allows to arbitrarily constrain specific components of the relative position and orientation of two nodes. The value of the constrained components of the relative position and orientation can be imposed by means of drives. As such, this element allows to mimic the behavior of most ideal constraints that connect two nodes.

```
<joint_type> ::= total joint

<joint_arglist> ::=
    <node_1_label>
        [ , position , (Vec3) <relative_offset_1> ]
        [ , position orientation , (OrientationMatrix) <rel_pos_orientation_1> ]
        [ , rotation orientation , (OrientationMatrix) <rel_rot_orientation_1> ]
    , <node_2_label>
        [ , position , (Vec3) <relative_offset_2> ]
        [ , position orientation , (OrientationMatrix) <rel_pos_orientation_2> ]
        [ , rotation orientation , (OrientationMatrix) <rel_rot_orientation_2> ]
    [ , position constraint ,
        <position_status> , <position_status> , <position_status> ,
        (TplDriveCaller<Vec3>) <imposed_relative_position> ]
    [ , orientation constraint ,
        <orientation_status> , <orientation_status> , <orientation_status> ,
        (TplDriveCaller<Vec3>) <imposed_relative_rotation> ]

<position_status> ::=
        { inactive | active | position | velocity | <status> }

<orientation_status> ::=
        { inactive | active | rotation | angular velocity | <status> }
```

The relative position imposed by the position constraint is imposed in a reference frame rigidly attached to the first node, in the optional offset relative_offset_1, and optionally further oriented by the rel_pos_orientation_1 matrix.

The relative orientation imposed by the orientation constraint is imposed in a reference frame rigidly attached to the first node, optionally further oriented by the rel_rot_orientation_1 matrix. It consists in the Euler vector that expresses the imposed relative orientation, in radian.

The keyword active means that the constraint is active with respect to that component of relative motion, so the related motion component is constrained, while inactive means that the constraint is not active with respect of that component of relative motion, so the related motion component is not

constrained. Otherwise, a boolean can be provided in `status` to indicate that the degree of constraint is either inactive (0) or active (1); this may be useful, for instance, to make constraint activation conditional in parametric input files. The same applies to the status of the components of the impose orientation.

If a component of relative position or orientation is active, the corresponding component of the imposed position or orientation is enforced, otherwise it is ignored; however, the complete three-dimensional vectors of the imposed relative position or orientation must be provided.

When the `position constraint` is enforced, the keyword `active` is equivalent to `position`; if the keyword `velocity` is used, the constraint imposes the velocity of that component of the relative position, resulting in a non-holonomic constraint.

Similarly, when the `orientation constraint` is enforced, the keyword `active` is equivalent to `rotation`; if the keyword `angular velocity` is used, the constraint imposed the angular velocity of that component of the relative orientation, resulting in a non-holonomic constraint.

Output

The output occurs in the `.jnt` file, which contains:

- column 1: the label
- columns 2–4: the three components of the internal force in the reference frame of the element
- columns 5–7: the three components of the internal moment in the reference frame of the element
- columns 8–10: the three components of the internal force in the global reference frame
- columns 11–13: the three components of the internal moment in the global reference frame
- columns 14–16: the three components of the relative displacement in the reference frame of the element
- columns 17–19: the three components of the relative rotation vector in the reference frame of the element
- columns 20–22: the three components of the relative velocity in the reference frame of the element
- columns 23–25: the three components of the relative angular velocity in the reference frame of the element

Private Data

The following data are available:

1. `"px"` relative position along node 1 position orientation axis 1
2. `"py"` relative position along node 1 position orientation axis 2
3. `"pz"` relative position along node 1 position orientation axis 3

- alternatively, `"X[i]"`, `i` = 1, 2, 3, relative position along node 1 position orientation axis `i`

4. `"rx"` relative orientation about node 1 rotation orientation axis 1
5. `"ry"` relative orientation about node 1 rotation orientation axis 2
6. `"rz"` relative orientation about node 1 rotation orientation axis 3

- alternatively, "Phi[i]", i = 1, 2, 3, relative orientation about node 1 rotation orientation axis i

7. "Fx" reaction force along node 1 position orientation axis 1
8. "Fy" reaction force along node 1 position orientation axis 2
9. "Fz" reaction force along node 1 position orientation axis 3

- alternatively, "f[i]", i = 1, 2, 3, reaction force along node 1 position orientation axis i

10. "Mx" reaction moment about node 1 rotation orientation axis 1
11. "My" reaction moment about node 1 rotation orientation axis 2
12. "Mz" reaction moment about node 1 rotation orientation axis 3

- alternatively, "m[i]", i = 1, 2, 3, reaction moment about node 1 rotation orientation axis i

13. "dx" imposed relative position along node 1 position orientation axis 1
14. "dy" imposed relative position along node 1 position orientation axis 2
15. "dz" imposed relative position along node 1 position orientation axis 3
16. "tx" imposed relative orientation about node 1 rotation orientation axis 1
17. "ty" imposed relative orientation about node 1 rotation orientation axis 2
18. "tz" imposed relative orientation about node 1 rotation orientation axis 3
19. "vx" relative velocity along node 1 position orientation axis 1
20. "vy" relative velocity along node 1 position orientation axis 2
21. "vz" relative velocity along node 1 position orientation axis 3

- alternatively, "V[i]", i = 1, 2, 3, relative velocity along node 1 position orientation axis i

22. "wx" relative angular velocity about node 1 rotation orientation axis 1
23. "wy" relative angular velocity about node 1 rotation orientation axis 2
24. "wz" relative angular velocity about node 1 rotation orientation axis 3

- alternatively, "Omega[i]", i = 1, 2, 3, relative angular velocity about node 1 rotation orientation axis i

Hints

When wrapped by a driven element, the following hints are honored:

- offset{1} the offset of the joint with respect to node 1 is reset;
- offset{2} the offset of the joint with respect to node 2 is reset;
- position-hinge{1} the relative orientation of the relative position constraint with respect to node 1 is reset;
- position-hinge{2} the relative orientation of the relative position constraint with respect to node 2 is reset;

- `orientation-hinge{1}` the relative orientation of the relative orientation constraint with respect to node 1 is reset;
- `orientation-hinge{2}` the relative orientation of the relative orientation constraint with respect to node 2 is reset;
- `position-drive3` resets the relative position drive; the hint is passed to the Vec3 drive hint parser.
- `orientation-drive3` resets the relative orientation drive; the hint is passed to the Vec3 drive hint parser.

Example. This example mimicks an instance of NASTRAN's `RBE2` element that constrains displacement in x and y, and rotation about z:

```
set: const integer EID = 100;
set: const integer GN = 200;
set: const integer GM1 = 300;

# ...

# $.......2.......3.......4.......5.......
# RBE2    EID     GN      CM      GM1
joint: EID, total joint,
    GN,
        position, reference, node, null,
        position orientation, reference, node, eye,
        rotation orientation, reference, node, eye,
    GM1,
        position, reference, other node, null,
        position orientation, reference, other node, eye,
        rotation orientation, reference, other node, eye,
     position constraint,
        active,   # "active" if CM includes "1"; "inactive" otherwise
        active,   # "active" if CM includes "2"; "inactive" otherwise
        inactive, # "active" if CM includes "3"; "inactive" otherwise
        null,
     orientation constraint,
        inactive, # "active" if CM includes "4"; "inactive" otherwise
        inactive, # "active" if CM includes "5"; "inactive" otherwise
        active,   # "active" if CM includes "6"; "inactive" otherwise
        null;
```

8.12.49 Total pin joint

This element allows to arbitrarily constrain specific components of the absolute position and orientation of a node. The value of the constrained components of the absolute position and orientation can be imposed by means of drives. As such, this element allows to mimic the behavior of most ideal constraints that ground one node.

```
<joint_type> ::= total pin joint
```

```
<joint_arglist> ::=
    <node_label>
        [ , position , (Vec3) <relative_offset> ]
        [ , position orientation , (OrientationMatrix) <rel_pos_orientation> ]
        [ , rotation orientation , (OrientationMatrix) <rel_rot_orientation> ]
    [ , position , (Vec3) <absolute position> ]
    [ , position orientation , (OrientationMatrix) <abs_pos_orientation> ]
    [ , rotation orientation , (OrientationMatrix) <abs_rot_orientation> ]
    [ , position constraint ,
        <position_status> , <position_status> , <position_status> ,
        (TplDriveCaller<Vec3>) <imposed_absolute_position> ]
    [ , orientation constraint ,
        <orientation_status> , <orientation_status> , <orientation_status> ,
        (TplDriveCaller<Vec3>) <imposed_absolute_rotation> ]

<position_status> ::=
        { inactive | active | position | velocity | <status> }

<orientation_status> ::=
        { inactive | active | rotation | angular velocity | <status> }
```

The non-holonomic variant is not implemented yet.

Output

The output occurs in the `.jnt` file, which contains:

- column 1: the label
- columns 2–4: the three components of the internal force in the reference frame of the element
- columns 5–7: the three components of the internal moment in the reference frame of the element
- columns 8–10: the three components of the internal force in the global reference frame
- columns 11–13: the three components of the internal moment in the global reference frame
- columns 14–16: the three components of the absolute displacement in the reference frame of the element
- columns 17–19: the three components of the absolute rotation vector in the reference frame of the element
- columns 20–22: the three components of the absolute velocity in the reference frame of the element
- columns 23–25: the three components of the absolute angular velocity in the reference frame of the element

Private Data

The following data are available:

1. `"px"` absolute position along absolute position orientation axis 1
2. `"py"` absolute position along absolute position orientation axis 2

3. "pz" absolute position along absolute position orientation axis 3

- alternatively, "X[i]", i = 1, 2, 3, absolute position along absolute position orientation axis i

4. "rx" absolute orientation about absolute rotation orientation axis 1
5. "ry" absolute orientation about absolute rotation orientation axis 2
6. "rz" absolute orientation about absolute rotation orientation axis 3

- alternatively, "Phi[i]", i = 1, 2, 3, absolute orientation about absolute rotation orientation axis i

7. "Fx" reaction force along absolute position orientation axis 1
8. "Fy" reaction force along absolute position orientation axis 2
9. "Fz" reaction force along absolute position orientation axis 3

- alternatively, "f[i]", i = 1, 2, 3, reaction force along absolute position orientation axis i

10. "Mx" reaction moment about absolute rotation orientation axis 1
11. "My" reaction moment about absolute rotation orientation axis 2
12. "Mz" reaction moment about absolute rotation orientation axis 3

- alternatively, "m[i]", i = 1, 2, 3, reaction moment about absolute rotation orientation axis i

13. "dx" imposed absolute position along absolute position orientation axis 1
14. "dy" imposed absolute position along absolute position orientation axis 2
15. "dz" imposed absolute position along absolute position orientation axis 3
16. "tx" imposed absolute orientation about absolute rotation orientation axis 1
17. "ty" imposed absolute orientation about absolute rotation orientation axis 2
18. "tz" imposed absolute orientation about absolute rotation orientation axis 3
19. "vx" absolute velocity along absolute position orientation axis 1
20. "vy" absolute velocity along absolute position orientation axis 2
21. "vz" absolute velocity along absolute position orientation axis 3

- alternatively, "V[i]", i = 1, 2, 3, absolute velocity along absolute position orientation axis i

22. "wx" absolute angular velocity about absolute rotation orientation axis 1
23. "wy" absolute angular velocity about absolute rotation orientation axis 2
24. "wz" absolute angular velocity about absolute rotation orientation axis 3

- alternatively, "Omega[i]", i = 1, 2, 3, absolute angular velocity about absolute rotation orientation axis i

Hints

When wrapped by a `driven` element, the following hints are honored:

- `offset{1}` the offset of the joint with respect to the node is reset;
- `offset{0}` the absolute position of the joint is reset;
- `position-hinge{1}` the relative orientation of the absolute position constraint with respect to the node is reset;
- `position-hinge{0}` the absolute orientation of the absolute position constraint is reset;
- `orientation-hinge{1}` the relative orientation of the absolute orientation constraint with respect to the node is reset;
- `orientation-hinge{0}` the absolute orientation of the absolute orientation constraint is reset;
- `position-drive3` resets the absolute position drive; the hint is passed to the Vec3 drive hint parser.
- `orientation-drive3` resets the absolute orientation drive; the hint is passed to the Vec3 drive hint parser.

Using `hints` with a `total pin joint` may be tricky. For example, when the `position constraint` must be modified, it is important to understand what this field indicates. In the case of the `total pin joint`, it indicates the relative position of the point attached to the node and expressed by the first `position` keyword with respect to the absolute position of the point expressed by the second `position` keyword.

Let us assume that the `structural node` labeled `NODE1` is defined in a given initial position, `{X0, Y0, Z0}`. During the analysis, the position of the node changes. At time `T1`, the node is "grabbed" by a `total pin joint`, which prescribes a given *displacement* from the location at which the node is grabbed. The syntax would be

```
driven: 30, string, "Time >= T1",
    hint, "position-drive3{array, 2,
        # first vector, position of node at time T1
        model::xposition(NODE1) - X0,
                model::yposition(NODE1) - Y0,
                model::zposition(NODE1) - Z0,
            const, 1.,
        # second vector, prescribed displacement
        1., 0., 0.,
            cosine, T1, 2*pi, 9., half, 0.
        }",
joint: 30, total pin joint,
    NODE1,
        position, reference, global, X0, Y0, Z0,
        position orientation, reference, global, eye,
        rotation orientation, reference, global, eye,
    # ground
        # reference point on ground coincident with node initial position
        position, reference, global, X0, Y0, Z0,
        position orientation, reference, global, eye,
        rotation orientation, reference, global, eye,
```

```
        position constraint, active, active, active,
            null, # irrelevant, will be overridden by hint
        orientation constraint, inactive, inactive, inactive,
            null;
```

That is, an array of 3D `template drive callers`, i.e. the sum of 2 vectors, needs to be created. The first drive caller is the result of computing the difference between the position of the node at the current time step and the initial position; it is a constant vector. The second drive caller is the prescribed displacement from the position of the node at time `T1`.

An alternative way to produce the same result is to define the prescribed relative displacement in terms of an absolute displacement, i.e.

```
    driven: 30, string, "Time >= T1",
        hint, "position-drive3{array, 2,
            # first vector, position of node at time T1
            model::xposition(NODE1), model::yposition(NODE1), model::zposition(NODE1),
                const, 1.,
            # second vector, prescribed displacement
            1., 0., 0.,
                cosine, T1, 2*pi, 9., half, 0.
            }",
    joint: 30, total pin joint,
        NODE1,
            position, reference, global, X0, Y0, Z0,
            position orientation, reference, global, eye,
            rotation orientation, reference, global, eye,
        # ground
            # reference point on ground in origin rather than node initial position
            position, reference, global, null,
            position orientation, reference, global, eye,
            rotation orientation, reference, global, eye,
        position constraint, active, active, active,
            null, # irrelevant, will be overridden by hint
        orientation constraint, inactive, inactive, inactive,
            null;
```

Notice that in both cases the prescribed relative position, in the definition of the `total pin joint`, is irrelevant, since it will be overridden by that defined in the `hint`. As such, an arbitrary value, e.g. `null`, is used.

8.12.50 Total reaction

Original implementation: Alessandro Fumagalli, Marco Morandini.

See also `total equation` (see Section 8.12.47). TODO

8.12.51 Universal hinge

Deprecated in favour of the `Cardano hinge` (see Section 8.12.6).

8.12.52 Universal pin

Deprecated in favour of the `Cardano pin` (see Section 8.12.7).

8.12.53 Universal rotation

Deprecated in favour of the `Cardano rotation` (see Section 8.12.8).

8.12.54 Viscous body

```
<joint_type> ::= viscous body

<joint_arglist> ::=
    <node_label> ,
        [ position ] , (Vec3) <relative_offset> ,
        [ orientation , (OrientationMatrix) <relative_orientation> , ]
        (ConstitutiveLaw<6D>) <const_law>
```

This element defines a force and a moment that depend on the absolute linear and angular velocity of a body, projected in the reference frame of the node itself. The force and moment are defined as a 6D viscous constitutive law.

8.12.55 Lower Pairs

Revolute

Use a `revolute hinge`, or a `total joint`.

Prismatic

Use a `prismatic` and an `inline`, or a `total joint`.

Screw

Use a `screw` joint.

Cylindrical

Use a `revolute rotation` and an `inline`, or a `total joint`.

Spherical

Use a `spherical hinge`, or a `total joint`.

Planar

Use a `revolute rotation` and an `in plane`, or a `total joint`.

8.13 Joint Regularization

The `joint regularization` element is used to modify algebraic constraint equations in order to improve ill-conditioned problems.

```
<elem_type> ::= joint regularization

<normal_arglist> ::= <type> [ , <data> ]

<type> ::= Tikhonov

<data> ::= { <coef> | list , <coef_1> [ , ... ] }
```

An element is instantiated, which requires an underlying algebraic constraint (a `joint` element that instantiates algebraic equations) with the same label to exist. In that case, the algebraic constraint equations are modified to regularize the problem. The `joint` that is regularized must exist, and it must write constraint equations. If the keyword `list` is used, the coefficients for each internal state are expected; the number of coefficients is determined by the joint.

8.13.1 Tikhonov

The `Tikhonov` joint regularization type consists in modifying a constraint, expressed by an algebraic equation $\boldsymbol{\Phi}(\boldsymbol{x}, t)$ and the corresponding Lagrangian multipliers $\boldsymbol{\lambda}$, namely

$$\left(\boldsymbol{\Phi}^T_{/\boldsymbol{x}}\boldsymbol{\lambda}\right)_{/\boldsymbol{x}} \Delta\boldsymbol{x} + \boldsymbol{\Phi}^T_{/\boldsymbol{x}}\Delta\boldsymbol{\lambda} = \boldsymbol{F} - \boldsymbol{\Phi}^T_{/\boldsymbol{x}}\boldsymbol{\lambda} \tag{8.76}$$

$$\boldsymbol{\Phi}_{/\boldsymbol{x}}\Delta\boldsymbol{x} = -\boldsymbol{\Phi}(\boldsymbol{x}, t) \tag{8.77}$$

by adding the multipliers to the constraint equation, weighed by the coefficient `coef` (or by the list of coefficients `coef_<i>`, in case different coefficients are given):

$$\left(\boldsymbol{\Phi}^T_{/\boldsymbol{x}}\boldsymbol{\lambda}\right)_{/\boldsymbol{x}} \Delta\boldsymbol{x} + \boldsymbol{\Phi}^T_{/\boldsymbol{x}}\Delta\boldsymbol{\lambda} = \boldsymbol{F} - \boldsymbol{\Phi}^T_{/\boldsymbol{x}}\boldsymbol{\lambda} \tag{8.78}$$

$$\boldsymbol{\Phi}_{/\boldsymbol{x}}\Delta\boldsymbol{x} - \texttt{coef}\cdot\Delta\boldsymbol{\lambda} = -\boldsymbol{\Phi}(\boldsymbol{x}, t) + \texttt{coef}\cdot\boldsymbol{\lambda} \tag{8.79}$$

The Tikhonov regularization allows the constraint to be violated by an amount that depends on the multipliers. In this sense, the coefficient `coef` should be considered sort of a compliance: the larger the coefficient, the larger the constraint violation for a given value of the reaction $\boldsymbol{\lambda}$.

8.14 Output Elements

Output elements take care of inter-process communication. These elements can use specific communication means, depending on the type of simulation they are used for, and can communicate specific types of data.

8.14.1 Stream output

This is a special element which takes care of sending output to external processes by means of either `local` or `inet` sockets during batch or real-time simulations. This topic is under development, so expect frequent changes, and please do not count too much on backward compatibility.

The syntax is:

```
<elem_type> ::= stream output

<arglist> ::=
    stream name , " <stream_name> " ,
```

```
    [ send after , { convergence | predict } , ]
create , { yes | no } ,
[ { local , " <socket_name> " , |
    [ port , <port_number> , ]
    [ host , " <host_name> " , ] } ]
[ socket type , { tcp | udp } , ]
[ { [ no ] signal
    | [ non ] blocking
    | [ no ] send first
    | [ do not ] abort if broken } [ , ... ] , ]
[ output every , <steps> , ]
[ echo , <file_name>
    [ , precision , <precision> ]
    [ , shift , <shift> ] , ]
<content>
```

The stream output allows MBDyn to send streamed outputs to remote processes during both batch and real-time simulations, using sockets either in the `local` or in the `inet` namespace. If the simulation is run in real-time using RTAI, RTAI mailboxes are used instead.

- `stream_name` is the name of the RTAI mailbox where the output is written (a unique string no more than six characters long); it is basically ignored by the `stream output` element except when using RTAI;
- `send after` specifies when the output is sent. By default, it is sent after `convergence`; as an alternative, after `predict` can be used to send the predicted values instead.
- the `create` keyword determines whether the socket must be created or looked-up as already existing on the system; if `create` is set to `no`, MBDyn will retry for 60 seconds and then give up;
- `socket_name` is the path of the `local` socket that is used to communicate between processes;
- `port_number` is the port number to be used with a `inet` socket. The default port number is 9011 (intentionally unassigned by IANA). If no `host_name` is given, `localhost` will be used;
- `host_name` is the name or the IP of the remote host where the mailbox resides; note that if this field is given, `create` must be set to `no`. The simulation will not start until the socket is created on the remote host;
- `socket type` defaults to `tcp`;
- the flag `no signal` requests that no `SIGPIPE` be raised when sending through a socket when the other end is broken (by default, `SIGPIPE` is raised);
- the flag `non blocking` requests that operations on the socket do not block (or block, in case of `blocking`, the default);
- the flag `no send first` requests that no send occurs before the first time step (by default, data are always sent);
- the flag `do not abort if broken` requests that the simulation continues in case the connection breaks. No further data send will occur for the duration of the simulation (the default);
- the field `output every` requests output to occur only every `steps`;

- the field `echo` causes the content that is sent to the peer to be echoed on file `file_name`; the optional parameter `precision` determines the number of digits used in the output; the optional parameter `shift` is currently unused;
- the field `content` is detailed in the next section.

This element, when used with the `motion` content type, obsoletes the `stream motion output` element (see Section 8.14.3). When the simulation is executed in real-time using RTAI, this element obsoletes the `RTAI output` element (see Section 8.14.2).

Streamed output

Different types of data can be sent. The most general form is called `values`, consisting in an arbitrary set of independent scalar channels. A form specifically intended to communicate the motion of a mechanical system is called `motion`, consisting in a subset of the kinematics of a set of structural nodes:

```
<content> ::= { <values> | <motion> | <user_defined> }

<values> ::= [ values , ]
    <channel_number> ,
        <value>
        [ , ... ]
    [ , modifier , <content_modifier> ]

<value> ::=
    { [ nodedof , ] (NodeDof) <output_dof>
        | drive , (DriveCaller) <drive_data> }

<motion> ::= motion ,
    [ output flags ,
        { position
            | orientation matrix
            | orientation matrix transpose
            | velocity
            | angular velocity }
          [ , ... ] , ]
    { all | <struct_node_label> [ , ... ] }

<user_defined> ::= <user_defined_type> [ , ... ]
```

where

- the (optional) keyword `values` indicates that a set of independent scalar channels is output by the element;
- the number of channels `channel_number` that are written determines how many `value` entries must be read. In case of `nodedof` (the default, deprecated), they must be valid scalar dof entries, which can be connected in many ways to nodal degrees of freedom; in case of `drive`, they can be arbitrary functions, including node or element private data;
- the keyword `modifier` defines how the content of the stream is modified before being sent; see Section 7.1.5 for details.

- the keyword `motion` indicates that a subset of the kinematic parameters of a set of structural nodes is output by the element. As opposed to the `values` case, which is intended for generic interprocess communication output, this content type is intended to ease and optimize the output of the motion of structural nodes, to be used for on-line visualization purposes. By default, only the position of the selected nodes is sent. This is intended for interoperability with a development version of EasyAnim which can read the motion info (the so-called "van" file) from a stream. The optional keyword `output flags` allows to request the output of specific node kinematics: the node position, orientation matrix (either row- or column-major), the velocity and the angular velocity. The default is equivalent to `output flags, position`.

Non real-time simulation

During non real-time simulations, streams operate in blocking mode. The meaning of the parameters is:

- `stream_name` indicates the name the stream would be known as by RTAI; it must be no more than 6 characters long, and mostly useless;
- the instruction `create` determines whether MBDyn will create the socket, or try to connect to an existing one;
- the keyword `local` indicates that a socket in the local namespace will be used; if `create` is set to `yes`, the socket is created, otherwise it must exist.
- either of the keywords `port` or `host` indicate that a socket in the internet namespace will be used;

  if `create` is set to `yes`, `host_name` indicates the host that is allowed to connect to the socket; it defaults to any host (`0.0.0.0`); if `create` is set to `no`, `host_name` indicates what host to connect to; the default is localhost (`127.0.0.1`); the default port is `9011` (intentionally unassigned by IANA);
- the flag `no signal` is honored;
- the flag `non blocking` is honored;
- the flag `no send first` is honored;
- the flag `do not abort if broken` is honored.

If no socket type is specified, i.e. none of the `local`, `port` and `host` keywords are given, a socket is opened by default in the internet namespace with the default IP and port; the `create` keyword is mandatory.

Real-time simulation

During real-time simulations, streams wrap non-blocking RTAI mailboxes. The meaning of the parameters is:

- the parameter `stream_name` indicates the name the stream will be known as in RTAI's resource namespace; it must be exactly 6 characters long;
- the instruction `create` determines whether the mailbox will be created or looked for by MBDyn;
- the keyword `local` is ignored;
- the keyword `host` indicates that a mailbox on a remote host will be used; it is useless when `create` is set to `yes`, because RTAI does not provide the possibility to create remote resources; if none is given, a local mailbox is assumed;

- the keyword `port` is ignored.
- the flag `no signal` is ignored;
- the flag `non blocking` is honored; however, blocking mailboxes make little sense, and real-time synchronization using RTAI should not rely on blocking mailboxes;
- the flag `no send first` is ignored (although it should be honored when the mailbox is blocking);
- the flag `do not abort if broken` is ignored; the program is always terminated if a mailbox is no longer available.

8.14.2 RTAI output

This element is actually used only when the simulation is scheduled using RTAI; otherwise, the corresponding `stream output` element is used (see Section 8.14.1). As a consequence, its explicit use is discouraged and deprecated. The `stream output` element should be used instead.

8.14.3 Stream motion output

This element type is obsoleted by the `stream output` element with the `motion` content type (see Section 8.14.3). The syntax is:

```
<elem_type> ::= stream motion output

<arglist> ::=
    stream name , " <stream_name> " ,
    create , { yes | no } ,
    [ { local , " <socket_name> " ,
            | [ port , <port_number> , ] [ host , " <host_name> " , ] } ]
    [ { [ no ] signal
            | [ non ] blocking
            | [ no ] send first
            | [ do not ] abort if broken }
        [ , ... ] , ]
    <motion>
```

Its support may be discontinued in the future.

8.15 Plate Elements

8.15.1 Shell4

Authors: Marco Morandini and Riccardo Vescovini

The shell4 elements model a four-node shell. The syntax is

```
<elem_type> ::= { shell4eas | shell4easans }

<normal_arglist> ::=
    <node_1_label> , <node_2_label> , <node_3_label> , <node_4_label> ,
    <shell_constitutive_law_data>
```

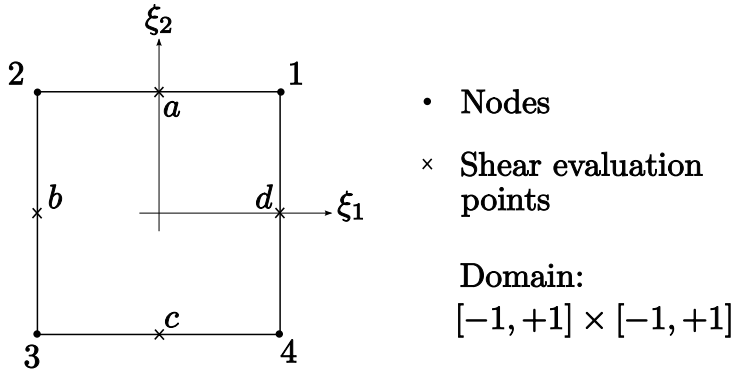


Figure 8.12: Shell: definitions

```
<shell_constitutive_law_data> ::=
    { [ matr , ] <12x12_matrix>
        | sym , <upper_triangular_12x12_matrix>
        | diag , <diagonal_12x12_matrix>
        | isotropic , <isotropic_data> }
    [ , prestress , (Vec12) <prestress> ]

<isotropic_data> ::=
    { { E | Young modulus } , <E>
        | { nu | Poisson modulus } , <nu>
        | { G | shear modulus } , <G>
        | as , <as>    # TODO: clarify
        | at , <at>    # TODO: clarify
        | thickness , <thickness> }
    [ , ... ]
```

The names `shell4eas` and `shell4easans` respectively stand for "Enhanced Assumed Strain" (EAS) and "Enhanced Assumed Strain-Assumed Natural Strain" (EAS-ANS).

Nodes are numbered according to Figure 8.12.

Only linear elastic constitutive properties can be currently modeled. They consist in a 12×12 matrix that expresses the force and moment fluxes as functions of linear and angular strains according to

$$\begin{Bmatrix} n_{11} \\ n_{12} \\ n_{13} \\ n_{21} \\ n_{22} \\ n_{23} \\ m_{11} \\ m_{12} \\ m_{13} \\ m_{21} \\ m_{22} \\ m_{23} \end{Bmatrix} = [D] \begin{Bmatrix} \varepsilon_{11} \\ \varepsilon_{12} \\ \varepsilon_{13} \\ \varepsilon_{21} \\ \varepsilon_{22} \\ \varepsilon_{23} \\ \kappa_{11} \\ \kappa_{12} \\ \kappa_{13} \\ \kappa_{21} \\ \kappa_{22} \\ \kappa_{23} \end{Bmatrix} + \texttt{prestress} \tag{8.80}$$

The `prestress` is a force and moment per unit span contribution to the force and moment fluxes.

Typically, only a membrane prestress makes sense, namely

```
<prestress> ::= <T11> , <T12> , 0. , # force/span on face with normal 1
                <T21> , <T22> , 0. , # force/span on face with normal 2
                0.    , 0.    , 0. , # moment/span on face with normal 1
                0.    , 0.    , 0.   # moment/span on face with normal 2
```

with T21 ≡ T12.

When the isotropic keyword is used, only two of the optional sub-keywords E, nu and G are required, as the remaining parameter can be computed from the other two according to the relationship

$$\mathtt{G} = \frac{\mathtt{E}}{2\,(1+\mathtt{nu})}. \tag{8.81}$$

If all are provided, they must be consistent. The optional parameters as and at are not documented yet; the default should be used.

When the (optional) matr (or sym) form is used, the required data for an orthotropic plate, from Classical Laminate Theory (CLT), with the principal axes of orthotropy parallel to the edges of the structure and laminate, and with a symmetrical arrangement of the layers, are

$$\begin{Bmatrix} N_x \\ N_y \\ N_{xy} \\ M_x \\ M_y \\ M_{xy} \end{Bmatrix} = \begin{bmatrix} CE_x & C\nu_{xy}E_y & 0 & 0 & 0 & 0 \\ C\nu_{xy}E_y & CE_y & 0 & 0 & 0 & 0 \\ 0 & 0 & 2G_{xy}h & 0 & 0 & 0 \\ 0 & 0 & 0 & DE_x & D\nu_{xy}E_y & 0 \\ 0 & 0 & 0 & D\nu_{xy}E_y & DE_y & 0 \\ 0 & 0 & 0 & 0 & 0 & 2F \end{bmatrix} \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_{xy} \\ \kappa_x \\ \kappa_y \\ \kappa_{xy} \end{Bmatrix} \tag{8.82}$$

with

$$C = \frac{h}{1-\nu_{xy}\nu_{yx}} \tag{8.83a}$$

$$D = \frac{h^3}{12(1-\nu_{xy}\nu_{yx})} \tag{8.83b}$$

$$F = \frac{G_{xy}h^3}{12} \tag{8.83c}$$

The data actually required by MBDyn are

$$\begin{Bmatrix} N_{11} \\ N_{12} \\ N_{13} \\ N_{21} \\ N_{22} \\ N_{23} \\ M_{11} \\ M_{12} \\ M_{13} \\ M_{21} \\ M_{22} \\ M_{23} \end{Bmatrix} = \begin{bmatrix} CE_x & 0 & 0 & 0 & C\nu_{xy}E_y & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2G_{xy}h & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \alpha_s G_{xy}h & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2G_{xy}h & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ C\nu_{xy}E_y & 0 & 0 & 0 & CE_y & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \alpha_s G_{xy}h & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2F & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & DE_x & 0 & -D\nu_{xy}E_y & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2F\alpha_t & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -D\nu_{xy}E_y & 0 & 0 & DE_y & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2F & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2F\alpha_t \end{bmatrix} \begin{Bmatrix} \varepsilon_{11} \\ \varepsilon_{12} \\ \varepsilon_{13} \\ \varepsilon_{21} \\ \varepsilon_{22} \\ \varepsilon_{23} \\ \kappa_{11} \\ \kappa_{12} \\ \kappa_{13} \\ \kappa_{21} \\ \kappa_{22} \\ \kappa_{23} \end{Bmatrix} \tag{8.84}$$

where α_s is the shear factor, and the torsional factor α_t is the material coefficient typical for a 6-field shell theory, which does not appear in conventional shell theories. It should be viewed as an analogue of the shear factor. Values of α_t from 0 to 1 have negligible influence on displacements and on the internal energy of the system. See [40] for details.

8.15.2 Membrane4

Authors: Marco Morandini and Tommaso Solcia

The membrane4 element models a four-node membrane. The syntax is

```
<elem_type> ::= membrane4eas

<normal_arglist> ::=
    <node_1_label> , <node_2_label> , <node_3_label> , <node_4_label> ,
    <membrane_constitutive_law_data>

<membrane_constitutive_law_data> ::=
    { [ matr , ] <3x3_matrix>
        | sym , <upper_triangular_3x3_matrix>
        | diag , <diagonal_3x3_matrix>
        | isotropic , <isotropic_data> }
    [ , prestress , (Vec3) <prestress> ]

<isotropic_data> ::=
    { { E | Young modulus } , <E>
        | { nu | Poisson modulus } , <nu>
        | { G | shear modulus } , <G>
        | thickness , <thickness> }
    [ , ... ]
```

The name `membrane4eas` stands for "Enhanced Assumed Strain" (EAS). Nodes are numbered according to Figure 8.12. Only linear elastic constitutive properties can be currently modeled. They consist in a 3×3 matrix that expresses the force fluxes as functions of linear strains according to

$$\left\{ \begin{array}{c} n_{11} \\ n_{22} \\ n_{12} \end{array} \right\} = [D] \left\{ \begin{array}{c} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{12} \end{array} \right\} + \texttt{prestress} \tag{8.85}$$

The `prestress` is a force per unit span contribution to the force fluxes,

```
<prestress> ::= <T11> , <T22> , <T12>
```

When the `isotropic` keyword is used, only two of the optional sub-keywords `E`, `nu` and `G` are required, as the remaining parameter can be computed from the other two according to the relationship

$$\texttt{G} = \frac{\texttt{E}}{2\,(1+\texttt{nu})}. \tag{8.86}$$

If all are provided, they must be consistent.

When the (optional) `matr` (or `sym`) form is used, the required data for an orthotropic membrane are

$$\left\{ \begin{array}{c} N_{11} \\ N_{22} \\ N_{12} \end{array} \right\} = \left[\begin{array}{ccc} CE_1 & C\nu_{21}E_1 & 0 \\ C\nu_{12}E_2 & CE_2 & 0 \\ 0 & 0 & 2G_{12}h \end{array} \right] \left\{ \begin{array}{c} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{12} \end{array} \right\} \tag{8.87}$$

with

$$C = \frac{h}{1-\nu_{12}\nu_{21}} \tag{8.88a}$$

$$\nu_{12}E_2 = \nu_{21}E_1 \tag{8.88b}$$

Example:

```
membrane4eas: 100, 1, 2, 3, 4,
    matr, E1*H/(1-NU12*NU21),      E1*H*NU21/(1-NU12*NU21), 0,
          E2*NU12*H/(1-NU12*NU21), E2*H/(1-NU12*NU21),      0,
          0,                       0,                       2*G12*H,
    prestress, PS, PS, 0.;
```

8.16 Solid Elements

Author: Reinhard Resch

8.16.1 Scope

In order to model general three dimensional solid structures with complex shape and subject to large deformations, large strain and nonlinear material, a generic family of Isoparametric Finite Elements is implemented in MBDyn. Those elements are based on a Total Lagrangian Formulation according to [6]. Elements based on a pure displacement based formulation as well as a displacement/pressure formulation (u/p-c) are available. Shape functions and integration points are based on [6], [41] and [42]. See also table 8.2 and table 8.3 for a list of supported element types.

8.16.2 Element input format

```
<elem_type> ::= { hexahedron8 | hexahedron20 | hexahedron20r | hexahedron27 |
                  pentahedron15 | tetrahedron10 | tetrahedron20
                  hexahedron8upc | hexahedron20upc | hexahedron20upcr |
                  pentahedron15upc | tetrahedron10upc |
                  hexahedron8f | hexahedron20f | hexahedron20fr | hexahedron27f |
                  pentahedron15f | tetrahedron10f | tetrahedron20f }

  <normal_arglist> ::=
      [ static , ] [ lumped mass , ] <struct_node_data>, [ <hydr_node_data>, ]
      <nodal_density>, <constitutive_law_data> ;

   <struct_node_data> :: =
      (StructDispNode) <struct_node_1_label> ,
      (StructDispNode) <struct_node_2_label> ,
      ... ,
      (StructDispNode) <struct_node_N_label>


   <hydr_node_data> :: =
      (HydraulicNode) <hydr_node_1_label> ,
      (HydraulicNode) <hydr_node_2_label> ,
      ... ,
      (HydraulicNode) <hydr_node_N_label>

   <nodal_density> :: =
      (real) <rho_1>, (real) <rho_2> , ... , (real) <rho_N>
```

```
<constitutive_law_data> :: =
   { <constitutive_law_data_6D> | <constitutive_law_data_7D> | <constitutive_law_data_9D> }

 <constitutive_law_data_6D> :: =
   (ConstitutiveLaw<6D>) <constitutive_law_1> ,
   { (ConstitutiveLaw<6D>) <constitutive_law_2> | same } ,
   ... ,
   { (ConstitutiveLaw<6D>) <constitutive_law_M> | same }

 <constitutive_law_data_7D> :: =
   (ConstitutiveLaw<7D>) <constitutive_law_1> ,
   { (ConstitutiveLaw<7D>) <constitutive_law_2> | same } ,
   ... ,
   { (ConstitutiveLaw<7D>) <constitutive_law_M> | same }

 <constitutive_law_data_9D> :: =
   (ConstitutiveLaw<9D>) <constitutive_law_1> ,
   { (ConstitutiveLaw<9D>) <constitutive_law_2> | same } ,
   ... ,
   { (ConstitutiveLaw<9D>) <constitutive_law_M> | same }
```

8.16.3 Constitutive law types to be used for solid elements

Each solid element type described in section 8.16.2 has a dedicated type of constitutive law which must be used. See table 8.1.

	constitutive law		
element type	6D	7D	9D
hexahedron8	√		
hexahedron20	√		
hexahedron20r	√		
hexahedron27	√		
pentahedron15	√		
tetrahedron10	√		
tetrahedron20	√		
hexahedron8upc		√	
hexahedron20upc		√	
hexahedron20upcr		√	
pentahedron15upc		√	
tetrahedron10upc		√	
hexahedron8f			√
hexahedron20f			√
hexahedron20fr			√
hexahedron27f			√
pentahedron15f			√
tetrahedron10f			√
tetrahedron20f			√

Table 8.1: Constitutive laws to be used for each solid element type

8.16.4 Displacement based elements versus displacement/pressure elements (u/p-c)

All pure displacement based elements (e.g. `hexahedron8`, ...) require only structural displacement nodes, whereas displacement/pressure based elements (e.g. `hexahedron8upc`, ...) require additional hydraulic nodes, in order to represent the hydrostatic pressure. Usually the number of hydraulic nodes is lower than the number of structural nodes in order to prevent locking phenomenon [6]. Due to the use of dedicated hydraulic nodes, the hydrostatic portion of the stress tensor is always continuous across compatible elements with common nodes [6]. For that reason, those elements have the suffix *-`upc` in their element names, which stands for displacement/pressure-continuous (u/p-c). In addition to that, even fully incompressible constitutive laws like Mooney Rivlin are supported by u/p-c elements [6].

8.16.5 Dynamic versus static elements

By default, dynamic solid elements with inertia effects enabled are created, unless the keyword `static` is used in the element description or the statement "`model`: `static`" was present inside the control data block according section 5.3.17.

8.16.6 Gravity and rigid body kinematics

All solid elements including static ones, are supporting gravity loads according section 8.10 and rigid body kinematics according section 5.3.18.

8.16.7 Material properties and constitutive laws

For that reason, the density of the material must be provided also for static solid elements. Solid elements may have variable density within a single element. Also in case of constant density, density values must be provided for each node, and it will be interpolated between nodes. Any linear or nonlinear elastic or viscoelastic constitutive law can be used, if it fulfills the following requirements: A constitutive law must expect the components $\boldsymbol{\varepsilon}$ of the Green-Lagrange strain tensor $\boldsymbol{G}$, and optionally it's scaled time derivatives $\dot{\boldsymbol{\varepsilon}}^{\star}$ as input, and return the components $\boldsymbol{\sigma}$ of the Second Piola-Kirchhoff stress tensor $\boldsymbol{S}$ as output. The following types of constitutive laws are implemented:

- linear elastic isotropic
- linear viscoelastic isotropic,
- hyperelastic isotropic
- elasto-plastic with isotropic hardening
- generalized Maxwell viscoelastic
- constitutive laws provided by the MFront code generation tool

See also section 2.10.31, section 2.10.32, section 2.10.33, section 2.10.34, section 2.10.35 and section 2.10.36. In case of viscoelastic constitutive laws, the strain rates $\dot{\boldsymbol{\varepsilon}}$ are scaled in order to make the effect of structural damping independent on the strain [8].

$$\boldsymbol{\sigma} = \boldsymbol{f}\left(\boldsymbol{\varepsilon},\, \dot{\boldsymbol{\varepsilon}}^{\star}\right) \tag{8.89}$$

$$\boldsymbol{\sigma} = \begin{pmatrix} S_{11} & S_{22} & S_{33} & S_{12} & S_{23} & S_{31} \end{pmatrix}^{T} \tag{8.90}$$

$$\boldsymbol{\varepsilon} = \begin{pmatrix} G_{11} & G_{22} & G_{33} & 2\,G_{12} & 2\,G_{23} & 2\,G_{31} \end{pmatrix}^{T} \tag{8.91}$$

$$\dot{\boldsymbol{\varepsilon}}^{\star} = \begin{pmatrix} \dot{G}^{\star}_{11} & \dot{G}^{\star}_{22} & \dot{G}^{\star}_{33} & 2\,\dot{G}^{\star}_{12} & 2\,\dot{G}^{\star}_{23} & 2\,\dot{G}^{\star}_{31} \end{pmatrix}^{T} \tag{8.92}$$

$$\dot{\boldsymbol{G}}^{\star} = \boldsymbol{C}^{-1}\,\dot{\boldsymbol{G}}\,\boldsymbol{C}^{-1}\,\det\boldsymbol{F} \tag{8.93}$$

$$\boldsymbol{G} = \frac{1}{2}\left(\boldsymbol{C} - \boldsymbol{I}\right) \tag{8.94}$$

$$\boldsymbol{C} = \boldsymbol{F}^{T}\,\boldsymbol{F} \tag{8.95}$$

$$\boldsymbol{F} = \nabla\boldsymbol{u} + \boldsymbol{I} \tag{8.96}$$

$\boldsymbol{f}$ Constitutive law

$\boldsymbol{G}$ Green-Lagrange strain tensor

$\boldsymbol{S}$ Second Piola-Kirchhoff stress tensor

$\boldsymbol{C}$ Right Cauchy-Green strain tensor

$\boldsymbol{F}$ Deformation gradient

$\boldsymbol{u}$ Deformation field

Constitutive laws for incompressible materials All Finite Elements based on a pure displacement based formulation (e.g. `hexahedron8`, ...) require 6D constitutive laws, whereas elements based on a displacement/pressure (u/p-c) formulation (e.g. `hexahedron8upc`, ...) require 7D constitutive laws. In the latter case, the additional component of the strain tensor $\boldsymbol{\varepsilon}_7$ is used to pass the interpolated hydrostatic pressure $\tilde{p}$, and the additional component of the stress tensor $\boldsymbol{\sigma}_7$ is used to return an expression related to the condition of volumetric compatibility. See equations 8.97 to 8.99.

$$\boldsymbol{\sigma} = \begin{pmatrix} S_{11} & S_{22} & S_{33} & S_{12} & S_{23} & S_{31} & \frac{1}{\kappa}\left(\bar{p} - \tilde{p}\right) \end{pmatrix}^{T} \tag{8.97}$$

$$\boldsymbol{\varepsilon} = \begin{pmatrix} G_{11} & G_{22} & G_{33} & 2\,G_{12} & 2\,G_{23} & 2\,G_{31} & \tilde{p} \end{pmatrix}^{T} \tag{8.98}$$

For isotropic materials, the condition of volumetric compatibility is [6].

$$\int_{V} \frac{1}{\kappa}\left(\bar{p} - \tilde{p}\right)\,\frac{\partial\tilde{p}}{\partial\hat{p}_i}\,dV = 0 \tag{8.99}$$

$\bar{p}$ Hydrostatic pressure calculated from the strain tensor $\boldsymbol{G}$

$\tilde{p}$ Hydrostatic pressure interpolated from the hydraulic nodes of the element

$\hat{p}$ Hydrostatic pressure at the hydraulic nodes of the element

κ Bulk modulus (e.g. $\kappa = \frac{E}{3\,(1-2\nu)}$ for linear elastic isotropic materials)

Constitutive laws based on the deformation gradient and the First Piola-Kirchhoff stress tensor For a few constitutive laws (e.g. `mfront finite strain`), it is necessary to provide the unsymmetric deformation gradient $\boldsymbol{F}$ as input value instead of the symmetric Green Lagrange strain tensor $\boldsymbol{G}$. As a consequence, it is necessary to use a `ConstitutiveLaw<9D>` in order to pass all nine components of the unsymmetric 3×3 tensor $\boldsymbol{F}$.

Conventions Since this type of constitutive law was intended mainly to provide support for a constitutive law called `mfront finite strain`, the conventions of the MFrontGenericInterfaceSupport library are applied[10]. See also section 2.10.36.

$$\boldsymbol{\sigma} = \begin{pmatrix} P_{11} & P_{22} & P_{33} & P_{12} & P_{21} & P_{13} & P_{31} & P_{23} & P_{32} \end{pmatrix}^T \tag{8.100}$$

$$\boldsymbol{\varepsilon} = \begin{pmatrix} F_{11} & F_{22} & F_{33} & F_{12} & F_{21} & F_{13} & F_{31} & F_{23} & F_{32} \end{pmatrix}^T \tag{8.101}$$

$$\boldsymbol{P} = \boldsymbol{F}\,\boldsymbol{S} \tag{8.102}$$

$\boldsymbol{P}$ First Piola-Kirchhoff stress tensor

Constitutive laws and integration points One constitutive law must be provided per integration point. As a consequence it is possible to build elements with varying elastic properties across a single element. The actual number of integration points is shown in table 8.2 and table 8.3.

8.16.8 Output

Output of stress and strain By default six components of the Cauchy stress tensor $\bar{\boldsymbol{\tau}}$ and six strains $\bar{\boldsymbol{\varepsilon}}$ at element nodes are written to the output file. See also equation 8.103 to equation 8.105 which are based on [7]. Since stress and strain are evaluated at integration points instead of nodes, it is required to extrapolate $\bar{\boldsymbol{\tau}}$ and $\bar{\boldsymbol{\varepsilon}}$ from integration points to nodes using Lapack's DGELSD function.

$$\boldsymbol{\tau} = \frac{1}{\det \boldsymbol{F}}\,\boldsymbol{F}\,\boldsymbol{S}\,\boldsymbol{F}^T \tag{8.103}$$

$$\bar{\varepsilon}_\alpha = \sqrt{C_{\alpha\alpha}} - 1 \tag{8.104}$$

$$\sin\vartheta_{\alpha\beta} = \frac{C_{\alpha\beta}}{(1+\bar{\varepsilon}_\alpha)\,(1+\bar{\varepsilon}_\beta)} \tag{8.105}$$

$$\bar{\boldsymbol{\tau}} = \begin{pmatrix} \tau_{11} & \tau_{22} & \tau_{33} & \tau_{12} & \tau_{23} & \tau_{31} \end{pmatrix}^T \tag{8.106}$$

$$\bar{\boldsymbol{\varepsilon}} = \begin{pmatrix} \bar{\varepsilon}_1 & \bar{\varepsilon}_2 & \bar{\varepsilon}_3 & \sin\vartheta_{12} & \sin\vartheta_{23} & \sin\vartheta_{31} \end{pmatrix}^T \tag{8.107}$$

Output of accelerations By default dynamic solid elements are using a consistent mass matrix. In order to enable output of accelerations for dynamic structural nodes, it is required to use a lumped mass matrix. For that purpose the keyword `lumped mass` must be used. Due to the special handling of accelerations in MBDyn, only solid elements with lumped mass matrix enabled can be used to compute accelerations for structural nodes as described in section 6.5.4.

element type	nodes	node order	integration points	order	integration	references
`hexahedron8`	8	8.13a	8	1	full	[6]
`hexahedron20`	20	8.13b	27	2	full	[6]
`hexahedron20r`	20	8.13c	8	2	reduced	[41]
`hexahedron27`	27	8.14	27	2	full	[42]
`pentahedron15`	15	8.13d	21	2	full	[42]
`tetrahedron10`	10	8.13e	5	2	full	[42]
`tetrahedron20`	20	8.13f	11	3	full	?

Table 8.2: Finite Element Types using a pure displacement based formulation

element type	nodes-u	node order-u	nodes-p	integration points	order	integration	references
hexahedron8upc	8	8.13a	1	8	1	full	[6]
hexahedron20upc	20	8.13b	8	27	2	full	[6]
hexahedron20upcr	20	8.13c	8	8	2	reduced	[41]
pentahedron15upc	15	8.13d	6	21	2	full	[42]
tetrahedron10upc	10	8.13e	4	5	2	full	[42]

Table 8.3: Finite Element Types using a displacement/pressure formulation

element type	nodes-u	node order-u	integration points	order	integration	references
hexahedron8f	8	8.13a	8	1	full	[6]
hexahedron20f	20	8.13b	27	2	full	[6]
hexahedron20fr	20	8.13c	8	2	reduced	[41]
pentahedron15f	15	8.13d	21	2	full	[42]
tetrahedron10f	10	8.13e	5	2	full	[42]
tetrahedron20f	20	8.13f	11	3	full	?

Table 8.4: Finite Element Types using the deformation gradient and the First Piola-Kirchhoff stress tensor

Private Data The following private data is available:

1. "E" Kinetic energy

Kinetic energy is always zero if the keyword static was used or if the statement "model: static" was present inside the control data block according section 5.3.17.

8.16.9 Example:

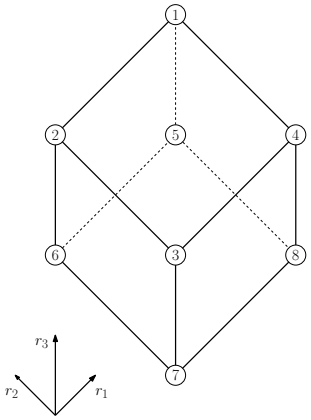
```
constitutive law: 1, name, "solid1", 6,
        linear viscoelastic generic,
        matr , 2.82e+11, 1.21e+11, 1.21e+11, 0.00e+00, 0.00e+00, 0.00e+00,
               1.21e+11, 2.82e+11, 1.21e+11, 0.00e+00, 0.00e+00, 0.00e+00,
               1.21e+11, 1.21e+11, 2.82e+11, 0.00e+00, 0.00e+00, 0.00e+00,
               0.00e+00, 0.00e+00, 0.00e+00, 8.07e+10, 0.00e+00, 0.00e+00,
               0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00, 8.07e+10, 0.00e+00,
               0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00, 8.07e+10,
               proportional, 1.0e-04;

hexahedron8: 100, 1, 2, 3, 4, 5, 6, 7, 8,
             7850., 7850., 7850., 7850., 7850., 7850., 7850., 7850.,
             reference, 1, same, same, same, same, same, same, same;

hexahedron8: 101, 5, 6, 7, 8, 9, 10, 11, 12,
             7850., 7850., 7850., 7850., 7850., 7850., 7850., 7850.,
             linear elastic isotropic, 210000e6, 0.3, same, same, same, same, same, same, same;
```

8.16.10 Pre- and post-processing

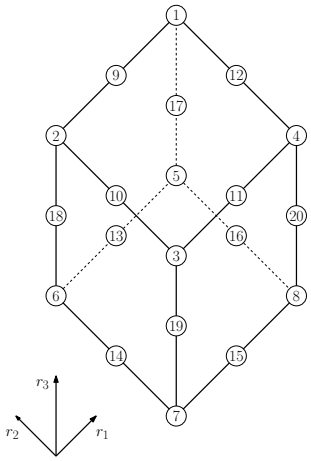
Since it would become tedious to create the mesh manually, pre- and post-processing of solid models can be performed by means of `GNU Octave`, `mboct-fem-pkg` and `Gmsh`. See also the following example and figure 8.15 on how to generate the input files and how to load the output generated by MBDyn:

```
## load the package
pkg load mboct-fem-pkg;
## load the mesh file in Gmsh format
mesh = fem_pre_mesh_import("cooks_membrane.msh", "gmsh");
## Apply a fill in reducing ordering based on METIS
mesh = fem_pre_mesh_reorder(mesh);
## assign material properties to material number 1
mesh.material_data(1).E = 240000; ## Young's modulus [Pa]
mesh.material_data(1).nu = 0.49; ## Poisson's ratio [1]
mesh.material_data(1).rho = 1000; ## density [kg/m^3]
mesh.material_data(1).type = "neo hookean"; ## hyperelastic rubber like material
## allocate material assignment
mesh.materials.iso20 = zeros(rows(mesh.elements.iso20), 1, "int32");
## locate a group of solid elements with id 10
grp_idx_solid = find([[mesh.groups.iso20].id] == 10);
## locate a group of surface elements with id 12
grp_idx_clamp = find([[mesh.groups.quad8].id] == 12);
## assign the material number one to all elements in group 10
mesh.materials.iso20(mesh.groups.iso20(grp_idx_solid).elements) = 1;
## allocate a new node number
node_id_interface = rows(mesh.nodes) + 1;
## define the position of a new node
mesh.nodes(node_id_interface, 1:3) = [100e-3, 50e-3, 20e-3];
## create an RBE3 element for the interface node (this will become a rigid body displacement joint)
## all nodes inside group 13 will be coupled to the interface node
mesh.elements.rbe3 = fem_pre_mesh_rbe3_from_surf(mesh, 13, node_id_interface, "quad8");
## allocate the DOF status for all nodes
load_case_dof.locked_dof = false(size(mesh.nodes));
## lock all DOF's at surface number 12
load_case_dof.locked_dof(mesh.groups.quad8(grp_idx_clamp).nodes, 1:3) = true;
## apply a force Fx = 5N * sin(2 * pi * t) and Fy = 5N * cos(2 * pi * t) at the interface node
load_case(1).loads = [5, 0, 0, 0, 0, 0];
load_case(2).loads = [0, 5, 0, 0, 0, 0];
load_case(1).loaded_nodes = [node_id_interface];
load_case(2).loaed_nodes = [node_id_interface];
opts.forces.time_function = {'string, "sin(2 * pi * Time)"', 'string, "cos(2 * pi * Time)"'};
## define the node type for all displacement only nodes with three degrees of freedom
opts.struct_nodes.type = repmat(MBDYN_NODE_TYPE_DYNAMIC_STRUCT_DISP, rows(mesh.nodes), 1);
## make sure that a static structural node
## with six degrees of freedom is created for the interface node
opts.struct_nodes.type(node_id_interface) = MBDYN_NODE_TYPE_STATIC_STRUCT;
## write all the nodes to file "cooks_membrane.nod"
opts = mbdyn_pre_solid_write_nodes(mesh, "cooks_membrane.nod", opts);
## write all the constitutive laws to file "cooks_membrane.csl"
opts = mbdyn_pre_solid_write_const_laws(mesh, "cooks_membrane.csl", opts);
```

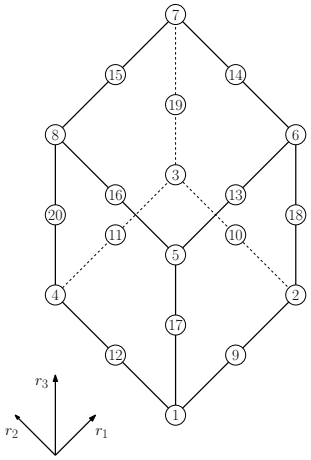
```
## write all the elements to file "cooks_membrane.elm"
opts = mbdyn_pre_solid_write_elements(mesh, load_case_dof, load_case, "cooks_membrane.elm", opts);
## define the location of the output file
opt_sol.output_file = "cooks_membrane_output";
## run MBDyn
info = mbdyn_solver_run("cooks_membrane.mbd", opt_sol);
## load the output
[mesh_sol, sol] = mbdyn_post_load_output_sol(opt_sol.output_file);
## display results using Gmsh
fem_post_sol_external(mesh_sol, sol);
```



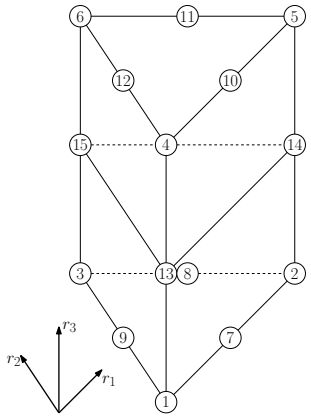
(a) hexahedron8



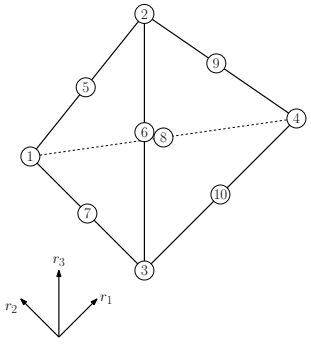
(b) hexahedron20



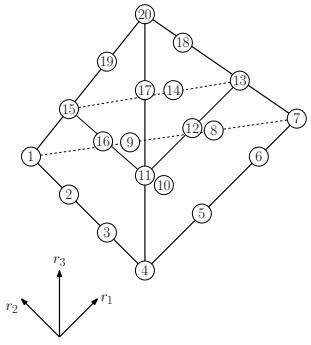
(c) hexahedron20r



(d) pentahedron15



(e) tetrahedron10



(f) tetrahedron20

Figure 8.13: Node order

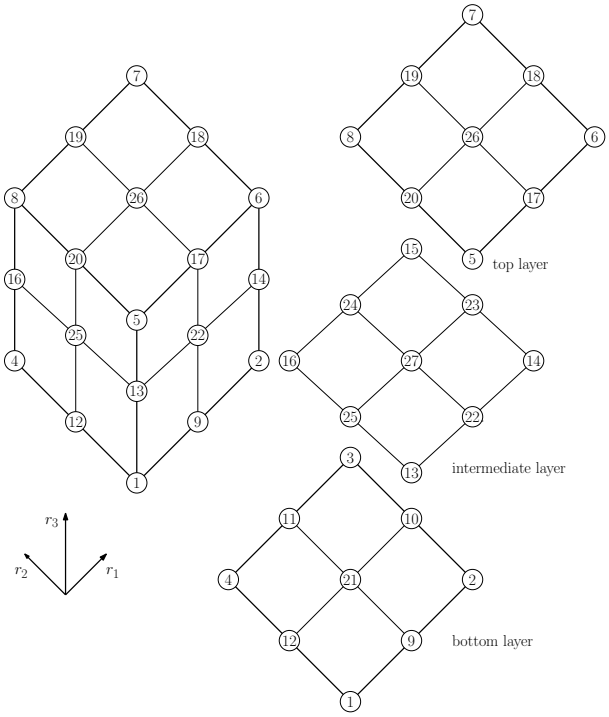


Figure 8.14: Node order: hexahedron27

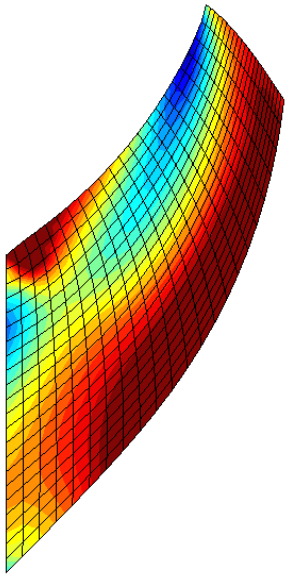


Figure 8.15: Cook's membrane: deformed shape and VON MISES stress rendered using Gmsh

8.17 Surface Loads

Author: Reinhard Resch

8.17.1 Pressure Loads

Pressure loads are intended mainly to apply prescribed pressure boundary conditions at the surface of solid elements. However, they could be used also for shell and membrane elements. The value of the pressure can be imposed by means of a set of drive callers, or by means of a set of abstract nodes. By convention, pressure loads are always applied in the opposite direction of the surface normal vector. So, it is important, that the surfaces are oriented properly when creating the mesh (e.g. use *ReorientMesh* in `Gmsh`). See also table 8.5 for a list of supported elements.

```
<elem_type> ::= { pressureq4 | pressureq8 | pressureq8r | pressureq9 | pressuret6 }

<normal_arglist> ::=
<struct_node_data> , <pressure_load_data> ;

<struct_node_data> :: =
(StructDispNode) <struct_node_1_label> ,
(StructDispNode) <struct_node_2_label> ,
... ,
(StructDispNode) <struct_node_N_label>

<pressure_load_data> :: =
{ from drives , <press_drive_data> |
from nodes  , <press_node_data> }

<pressure_drive_data> :: =
(DriveCaller) <press_drive_1> ,
(DriveCaller) <press_drive_2> ,
... ,
(DriveCaller) <press_drive_N>

<pressure_node_data> :: =
(AbstractNode) <press_node_1> ,
(AbstractNode) <press_node_2> ,
... ,
(AbstractNode) <press_node_N>
```

Example:

```
drive caller: 1, string, "1.5 * Time";
drive caller: 2, string, "2.5 * Time";
drive caller: 3, string, "3.5 * Time";
drive caller: 4, string, "4.5 * Time";

pressureq4: 100, 1, 2, 3, 4, from nodes, 10, 11, 12, 13;

pressureq4: 101, 5, 6, 7, 8, from drives,
```

```
                reference, 1,
                reference, 2,
                reference, 3,
                reference, 4;
```

8.17.2 Surface Traction's

Surface traction's are similar to pressure loads but they do not necessarily act perpendicular to the surface. So, they can be used to impose a prescribed shear stress at the surface of a solid body. Actually there are absolute and relative surface traction's. Absolute surface traction's are applied with respect to the global reference frame and do not change their direction and magnitude if the surface is moved or deformed during the simulation. In contradiction to that, relative surface traction's are also applied with respect to the global reference frame, but their direction and magnitude is changed similar to pressure loads, if the surface is deformed or moved. See also table 8.5 for a list of supported elements.

```
<elem_type> ::= { tractionq4 | tractionq8 | tractionq8r | tractionq9 | traction6 }

<normal_arglist> ::=
    [ absolute , ] <struct_node_data> , from drives , <traction_load_data>
    [ , orientation, <traction_orient_data> ] ;

    <struct_node_data> :: =
    (StructDispNode) <struct_node_1_label> ,
    (StructDispNode) <struct_node_2_label> ,
    ... ,
    (StructDispNode) <struct_node_N_label>

    <traction_load_data> :: =
    (TplDriveCaller<Vec3>) <traction_stress_1> ,
    (TplDriveCaller<Vec3>) <traction_stress_2> ,
    ... ,
    (TplDriveCaller<Vec3>) <traction_stress_N>

    <traction_orient_data> :: =
    (OrientationMatrix) <orientation_matrix_1> ,
    (OrientationMatrix) <orientation_matrix_2> ,
    ... ,
    (OrientationMatrix) <orientation_matrix_M>
```

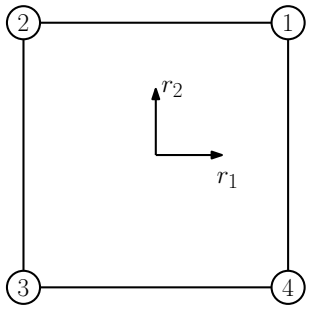
Example:

```
reference: 10,
         position, reference, global, null,
         orientation, reference, global, 1, 0., 1., 0.,
                                         2, 1., 0., 0.,
         velocity, reference, global, null,
         angular velocity, reference, global, null;

...
```

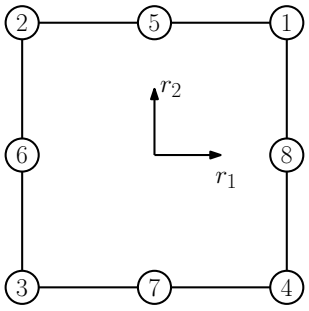
```
template drive caller: 1, 3, component,
         string, "1.5 * Time",
         string, "2.5 * Time",
         string, "3.5 * Time";

...

template drive caller: 4, 3, component,
         string, "4.5 * Time",
         string, "5.5 * Time",
         string, "6.5 * Time";

tractionq4: 100, 1, 2, 3, 4, from drives,
             reference, 1,
             reference, 2,
             reference, 3,
             reference, 4,
             orientation,
                reference, 10, eye,
                reference, 20, eye,
                reference, 30, eye,
                reference, 40, eye;
```

element type pressure	element type traction	nodes	node order	integration points	order	references
pressureq4	tractionq4	4	8.16a	4	1	[6]
pressureq8	tractionq8	8	8.16b	9	2	[6]
pressureq8r	tractionq8r	8	8.16c	9	2	[41]
pressureq9	tractionq9	9	8.16d	9	2	[42]
pressuret6	tractiont6	6	8.16e	7	2	[42]

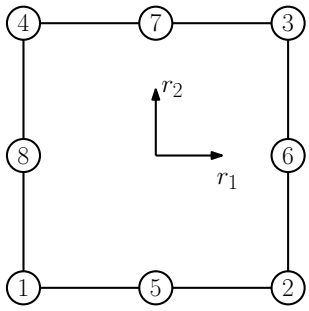
Table 8.5: Finite Element Types for surface loads



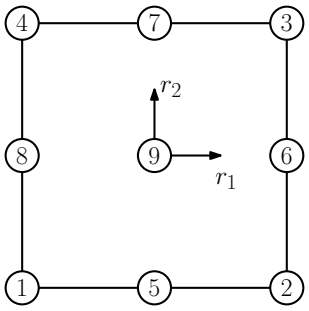
(a) `pressureq4`, `tractionq4`



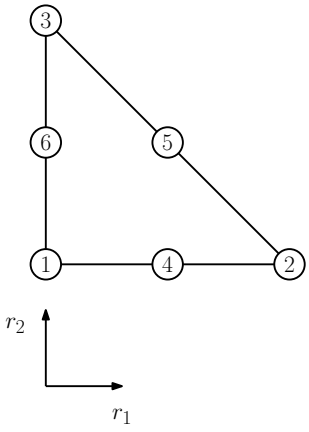
(b) `pressureq8`, `tractionq8`



(c) `pressureq8r`, `tractionq8r`



(d) `pressureq9`, `tractionq9`



(e) `pressuret6`, `tractiont6`

Figure 8.16: Node order: surface loads

8.18 Thermal Elements

8.18.1 Capacitance

This element implements a simple linear thermal capacitance associated with a thermal node. It computes the heat flow into the capacitance node as

$$q = -\texttt{dCapacitance} \cdot \dot{T}$$

The input format is:

```
<elem_type> ::= capacitance

<normal_arglist> ::=
    <node_label> ,
    (real) <dCapacitance>
```

where `node_label` is the thermal node the thermal capacitance is connected to, and `dCapacitance` is the (positive) value of the thermal capacitance.

Example.

```
thermal: 99, capacitance,
    11,    # node label
    0.11;  # capacitance
```

8.18.2 Resistance

This element implements a simple linear thermal resistance between two thermal nodes. It computes the heat flow between the two nodes as

$$q_{21} = \frac{T_2 - T_1}{\texttt{dResistance}}$$

The input format is:

```
<elem_type> ::= resistance

<normal_arglist> ::=
    <node_1_label> ,
    <node_2_label> ,
    (real) <dResistance>
```

where `node_1_label` and `node_2_label` are the thermal nodes the thermal resistance is connected to, and `dResistance` is the (positive) value of the thermal resistance.

Example.

```
thermal: 98, resistance,
    11,    # node 1 label
    12,    # node 2 label
    0.99;  # resistance
```

8.18.3 Source

This element implements a simple thermal source associated with a thermal node. It computes the heat flow into the node as

$$q = \texttt{Source}$$

The input format is:

```
<elem_type> ::= capacitance

<normal_arglist> ::=
    <node_label> ,
    (DriveCaller) <Source>
```

where node_label is the thermal node the thermal source is connected to, and Source is a DriveCaller.

Example.

```
thermal: 97, source,
    12,    # node label
    # source; grows as 1-cos from 0 s to 1 s, raising from 0. to 0.9
    cosine, 0., pi, 0.45, half, 0.;
```

8.19 User-Defined Elements

8.19.1 Loadable Element

Note: the loadable element is deprecated in favor of the user defined element.

The loadable element is a wrapper for a user-defined element that is compiled in a separated module and linked run-time. The module should provide a comprehensive set of functions according to a specified API; default functions are available if no special features are required. Implementation of modules can be very easy, but a deep knowledge of the internals of the code is required when special tasks need to be performed. There are virtually no limits on what a loadable element can do.

The syntax is simply:

```
<elem_type> ::= loadable

<normal_arglist> ::= " <module_name> "
    [ , name , " <calls> " ]
    [ , <module_data> ]
```

where module_name is the name of the module file; as soon as the file is checked and the binding of the structure with function bindings succeeded, a function called `read()` is invoked, passing it the input stream. This function is in charge of reading module_data following the general syntax of the input file.

An alternative form is

```
<normal_arglist> ::= reference , " <name> "
    [ , <module_data> ]
```

where name is the name by which the loadable element recorded itself when registered via the module load directive, as described in Section 2.4.6. As a consequence, the following forms are equivalent:

```
# direct runtime loading
loadable: 1, "/some/module.so";
# "/some/module.so" registers itself as "some_module"
module load: "/some/module.so";
loadable: 2, reference, "some_module";
# works also as joint (might obsolete loadable elements)
joint: 3, some_module;
```

It is advisable that the function `read()` prints some help message when the first field of `module_data` is the keyword `help`. All the helpers and the high-level structures are available, such as drivers, constitutive laws, reference frames. Refer to each module for a description (if available) of the features and of the input/output format. `module_name` should be a full path to the module function. If the name starts with a slash "/", the full path name is used. Otherwise the module is searched in the colon-separated list of directories contained in the environment variable `LD_LIBRARY`, then among the libraries listed in `/etc/ld.so.cache`, and finally in `/usr/lib` and in `/lib` (see `dlopen(3)`). At last, it is searched in the current directory, and the extension `.so` is added if missing. The string `calls` represents the name of the structure that contains the bindings to the functions. The default is `calls`.
Refer to `$(BASE)/mbdyn/base/loadable.h` for a description of the functions that are allowed. An example module is given in directory

```
$(BASE)/modules/module-template/
```

which can be used as a starting point to build a custom module. The `loadable` element interface allows to link modules in different languages, e.g. C or FORTRAN77; simply use `module-template` as a guideline to providing callbacks to the `loadable` element interface and to collect required info from the main program (e.g. node positions, equation indices and everything else that is required for appropriate connection), then call the functions that actually do the work in other languages from inside the callbacks.

8.19.2 User-Defined Element

The `user defined` element is a much more streamlined form of custom element definition than the `loadable` element. From the point of view of the syntax the differences are minimal; however, from an implementation point of view, the `user defined` element is preferable.

The definition of a `user defined` element requires two steps. In the first step, a run-time loadable module is loaded using the `module load` directive (see Section 2.4.6). This registers a handler to the `user defined` element type along with a `name` used to reference it. In the second step, an instance of that `user defined` element type is created, referencing it by `name`.

The syntax of the `user defined` element is

```
<elem_type> ::= user defined

<normal_arglist> ::= <name> [ , <module_data> ]
```

As for the `loadable` element, it is recommended that some useful output is given if the first `module_data` is the keyword `help`.

An example module is given in directory

```
$(BASE)/modules/module-template2/
```

which can be used as a starting point to build a custom module.

Example.

```
module load: "libmodule-template2";

user defined: 1000, template2, help;
```

8.19.3 General Discussion on Run-Time Loadable Modules

In general, to call external functions from C++ one needs to declare them as

```
#include <sys/types.h>
extern "C" {
    int a_C_function(int arg, double darg);
    int a_F77_subroutine(int32_t *iarg, double *darg);
}
```

The same applies to FORTRAN 77 functions; only, the naming convention usually is compiler dependent; some compilers turn all the names to uppercase or lowercase (remember that FORTRAN 77 is case insensitive); other compilers add underscores at the beginning or at the end of the names. Check what is the behavior of your compiler, by compiling a simple program with your favorite FORTRAN 77 compiler, then looking at it with the utility `nm(1)`, which will show how the symbols are represented internally. For instance, the code

```
C This is a compiler test
      SUBROUTINE F77SUB(I, D)
      INTEGER*4 I
      REAL*8 D(*)

      D(I) = 0.0

      END
```

when compiled with `g77(1)` on a GNU/Linux system, yields:

```
[masarati@mbdyn manual]$ g77 -c f77sub.f
[masarati@mbdyn manual]$ nm f77sub.o
00000000 T f77sub_
```

That is, `g77(1)` lowercases all symbols, and adds a trailing underscore. Macros to automatically detect and take care of this behavior are planned.
To compile loadable modules, one needs to configure the package as follows:

```
./configure --with-module=<module_name>
```

where `module_name` is the name of the directory the module is placed in with the `module-` part stripped; e.g. to compile the tire module that resides in `$(BASE)/modules/module-wheel2` one must type

```
./configure --with-module=wheel2
```

Multiple modules can be compiled by typing the list of the names separated by blanks.

The modules need to resolve some of the symbols that are in the main executable; until a full working libtool support is implemented, this must be done by hand. The `g++(1)` compiler requires the switch '`-rdynamic`' to be added to the loader's flags.

For example,

```
./configure --with-module=<module_name> LDFLAGS="-rdynamic"
```

8.20 Miscellaneous

This section lists some extra cards that do not correspond to any specific simulation entity, but rather alter the behavior of existing entries or cause special operations to be undertaken during model input.

8.20.1 Bind

The statement bind does not really define an element. It is rather used to instruct a parameter node about which parameter of an element it is bound to. The parameter node must exist, and the element the node is being bound to, of type elem_type and label element_label, must have been already defined. The complete syntax is:

```
<elem_type> ::= bind

<arglist> ::=
    <element_label> ,
    <element_type> ,
    <parameter_node_label> ,
    <bind_args>
```

where bind_args depend on the type of parameter node.

Element

When binding an element to an element parameter node, each element makes a number of specific parameters available. A detailed list is given for each element in the private data section. In that case:

```
<bind_args> ::=
    { [ index , ] <parameter_index>
        | string , " <parameter_name> " }
```

The value of parameter_index must be legal, i.e. between 1 and the maximum number of parameters made available by the element. The alternative form, using the keyword string followed by the parameter_name, allows more friendly definition of the binding. The name of the parameter depends on the element whose property is being bound. A complete listing of the parameters that a parameter node can be bound to can be found in the 'Private data' subsection of each element's specification.

Example. The parameter node ANGLE is bound to the rotation of a revolute hinge (see Section 8.12.38).

```
# ... problem block

begin: control data;
    structural nodes: 2;
    parameter nodes: 1;
    forces: 2;
    # ... other control data
end: control data;

set: integer NODE1 = 1000;
set: integer NODE2 = 2000;
set: integer ANGLE = 5000;
set: integer REVOLUTE = 6000;
```

```
begin: nodes;
    structural: NODE1, dynamic, null, eye, null, null;
    structural: NODE2, dynamic, null, eye, null, null;
    parameter: ANGLE, element;

    # ... other nodes
end: nodes;

begin: elements;
    joint: REVOLUTE, revolute hinge,
        NODE1,
            position, reference, node, null,
            orientation, reference, node, eye,
        NODE2,
            position, reference, node, null,
            orientation, reference, node, eye;
    bind: REVOLUTE, joint, ANGLE, string, "rx";
    couple: 1, NODE1, 0.,0.,1.,
        dof, ANGLE, parameter, 1, linear, 0.,1.;
    couple: 2, NODE2, 0.,0.,1.,
        element, REVOLUTE, joint, string, "rx", linear, 0.,1.;

    # ... other elements
end: elements;
```

Note that the same element data, i.e. the revolute hinge relative rotation angle, is used to drive a couple in two different ways; the latter, by means of the `element` drive (see Section 2.6.13) is more direct, but the former, by means of the `dof` drive (see Section 2.6.9) through the `bind` mechanism has the additional effect of updating the `parameter` node, which can be used to connect `genel` elements for special purposes.

Beam strain gage

When binding an element to a `beam strain gage` parameter node, the `element_type` field must be `beam`. In that case:

```
<bind_args> ::= <beam_evaluation_point>
```

where `beam_evaluation_point` is the evaluation point of the beam element where the internal strain and curvatures must be evaluated. It must be 1 for 2-node beams, while it can be either 1 or 2 for 3-node beams.

Example.

```
# ... problem block

begin: control data;
    parameter nodes: 1;
    beams: 1;
    # ... other control data
end: control data;
```

```
set: integer BEAM = 100;
set: integer STRAIN = 200;

begin: nodes;
    parameter: STRAIN, beam strain gage, 0.0, 0.1;
    # ... other nodes
end: nodes;

begin: elements;
    beam3: BEAM, ...; # beam data
    bind: BEAM, beam, STRAIN, 1;
    # ... other elements
end: elements;
```

8.20.2 Driven Element

The driven type is not an element by itself. It is a wrapper that masks another element and switches it on and off depending on the (boolean) value of a drive. It can be used to emulate a variable topology model, where some elements simply don't contribute to the residual or to the Jacobian matrices when their drive has a certain value. Since the drivers can be arbitrary functions of the time, or other parameters including the value of any degree of freedom, the driven elements can be "driven" in a very flexible way. Every element can be driven, except those that can be instantiated once only. The syntax for a driven element is:

```
<elem_type> ::= driven

<normal_arglist> ::= (DriveCaller) <element_driver> ,
    [ initial state , { from drive | active | inactive | (bool) <state> } ]
    [ hint , " <hint> " [ , ... ] ] ,
    <driven_element>

<driven_element> ::=
    { existing : <driven_elem_type> , <driven_elem_label>
        | <element_card> }
```

When the keyword existing is used, an existing element of type driven_elem_type and label driven_elem_label is looked for, and it is wrapped by the driven element. In this case, no new element is instantiated. The label of the element must match that of the driving element given at the beginning. For consistency with the syntax, and for more flexibility, even when wrapping an existing element the output flags can be set at the end of the card. This flag overrides the one set when the driven element was instantiated.

The initial state keyword overrides the evaluation of the element_driver during the initialization. The value from drive is the default behavior; the other keywords and the boolean value state are self-explanatory.

Otherwise, a regular element is read after the driving element's declaration; it is then instantiated and wrapped by the driven element wrapper. Note that after the keyword existing or after the driven element type, a colon is used as a separator. This is probably the only exception to the rule that the colon can only follow a description at the beginning of a card. The label driven_elem_label of the driven element must match that of the driving element used at the beginning of the driven card.

Example. A pin constraint between two rigid bodies is released:

```
set: integer BODY_1 = 1;
set: integer BODY_2 = 2;
# ...
structural: BODY_1, dynamic,
    null, eye, null, null;
structural: BODY_2, dynamic,
    null, eye, null, null;
# ....
body: BODY_1, BODY_1,
    1000., null, diag, 100.,100.,1.;
body: BODY_2, BODY_2,
    10., null, diag, 1.e-1,1.e-1,1.e-3;
# ...
# this constraint will be released when Time = 10 s
driven: 1, string, "Time < 10.",
joint: 1, spherical hinge,
    BODY_1,
        position, null,
    BODY_2,
        position, null;
```

Example. an axial rotation (see Section 8.12.3) joint is replaced by a revolute hinge (see Section 8.12.38) when the desired spin velocity, measured as the angular velocity of the second node (assuming, for instance, that the first one is fixed), is reached. The value of an abstract node is used to input the angular velocity to the axial rotation joint.

```
set: integer BODY_1 = 1;
set: integer BODY_2 = 2;
set: integer CONTROL_OUTPUT = 3;
# ...
structural: BODY_1, static,
    null, eye, null, null;
structural: BODY_2, dynamic,
    null, eye, null, null;
abstract: CONTROL_OUTPUT;
# ....
driven: 1, node, BODY_2, structural, string, "Omega[3]",
    string, "Var < 100.",
joint: 1, axial rotation,
    BODY_1,
        position, null,
        orientation, 1, 1.,0.,0., 3, 0.,0.,1.,
    BODY_2,
        position, null,
        orientation, 1, 1.,0.,0., 3, 0.,0.,1.,
    node, CONTROL_OUTPUT, abstract, string, "x",
        linear, 0.,1.;
driven: 2, node, BODY_2, structural, string, "Omega[3]",
    string, "Var >= 100.",
joint: 2, revolute hinge,
```

```
    BODY_1,
        position, null,
        orientation, 1, 1.,0.,0., 3, 0.,0.,1.,
    BODY_2,
        position, null,
        orientation, 1, 1.,0.,0., 3, 0.,0.,1.;
```

Hint

The hint feature consists in allowing the setup of elements to be computed when they are activated rather than at startup. For example, a joint that is activated after some simulation time may need to compute its relative position and orientation from the parameters of the simulation; a drive that controls the evolution of the relative configuration of a joint may need to infer its parameters from the current configuration of the overall system; and so on.

Currently, only few elements, significantly joints, support and honor hints. The typical syntax is given by a keyword followed by some optional parameters within curly brackets. For instance, the hint that instructs a joint to compute the offset with respect to the second node when it is activated is

```
driven: 10, string, "Time > 1.", hint, "offset{2}",
joint: 10, ...
```

A similar form is used to instruct a joint to compute the relative orientation with respect to node 1:

```
driven: 20, string, "Time > 1.", hint, "hinge{1}",
joint: 20, ...
```

A distance joint may be fed a new drive by using:

```
set: const real T0 = 1.;
driven: 30, string, "Time > T0",
    hint, "drive{cosine, T0, pi/.1, -.9/2., half, model::distance(1,2)}",
joint: 30, distance, ...
```

A drive hinge joint may be fed a new drive by using:

```
set: const real T0 = 1.;
driven: 40, string, "Time > T0",
    # note: the lines wrap for typographical reasons
    # in the actual file, the string has to be a single line
    hint, "drive3{model::xdistance(1,2), \
        model::ydistance(1,2), model::zdistance(1,2), \
        cosine, T0, pi/.1, -.9/2., half, model::distance(1,2)}",
joint: 30, drive hinge, ...
```

This feature will likely be extended to other elements and generalized as much as possible.

8.20.3 Inertia

This card causes the equivalent inertia properties of a subset of elements to be computed.

```
<card> ::= inertia : <label>
    [ , name , " <inertia_name> " ]
    [ , position , (Vec3) <reference_position> ]
```

```
[ , orientation , (Mat3x3) <reference_orientation> ]
, <type_subset> [ , ... ]
[ , output , { no | yes | log | both | always } [ , ... ] ] ;

<type_subset> ::= <type> , { all | <label> [ , ... ] }

<type> ::= { body | joint | solid | loadable }
```

where `type` currently can be `body`, `joint`, `solid` and `loadable`, although more elements associated with inertia might participate in the future. All elements whose labels are listed must exist, and duplicates are detected and considered errors. The keyword `all` causes all the elements of type `type` to be included in the list.
The only effect of the `inertia` statement is to log each `inertia` entry in the `.log` file in a self explanatory form, referred both to the `global` reference frame and to a reference frame originating from `reference_position` and oriented as `reference_orientation`. The optional parameter `output` may be used to alter the default behavior:

- `no` disables the output, making the `inertia` statement essentially ineffective;
- `yes` enables output to standard output;
- `log` enables output to the `.log` file (the default);
- `both` enables output to both standard output and `.log` file.
- `always` causes inertia properties to be re-computed at each time step. To this end, an `inertia` card must appear in the `control data` block, declaring how many corresponding `inertia` elements will be created. The number of inertia elements declared in the `control data` section must match the number of `inertia` cards in the `elements` block whose output is set to `always`.

Private Data The following private data is available for all inertia elements with `output`, `always` enabled:

1. `"X[1]"` position of the center of mass with respect to the global reference frame
2. `"X[2]"`
3. `"X[3]"`
4. `"Phi[1]"` orientation vector of principal axes with respect to the global reference frame
5. `"Phi[2]"`
6. `"Phi[3]"`
7. `"V[1]"` velocity of center of mass with respect to the global reference frame
8. `"V[2]"`
9. `"V[3]"`
10. `"Omega[1]"` angular velocity with respect to the global reference frame
11. `"Omega[2]"`
12. `"Omega[3]"`

13. "J[1][1]" total matrix of inertia with respect to reference_position and reference_orientation
14. "J[1][2]"
15. "J[1][3]"
16. "J[2][1]"
17. "J[2][2]"
18. "J[2][3]"
19. "J[3][1]"
20. "J[3][2]"
21. "J[3][3]"
22. "Jcg[1][1]" total matrix of inertia with respect to center of mass and the global reference frame
23. "Jcg[1][2]"
24. "Jcg[1][3]"
25. "Jcg[2][1]"
26. "Jcg[2][2]"
27. "Jcg[2][3]"
28. "Jcg[3][1]"
29. "Jcg[3][2]"
30. "Jcg[3][3]"
31. "JP[1]" total principal moments of inertia
32. "JP[2]"
33. "JP[3]"
34. "m" total mass
35. "beta[1]" total momentum with respect to the global reference frame
36. "beta[2]"
37. "beta[3]"
38. "gamma[1]" total momenta moment with respect to the center of mass and the global reference frame
39. "gamma[2]"
40. "gamma[3]"

Output The following output is available in the `.inl` file for all inertia elements with `output`, `always` enabled:

1. column 1: element label
2. columns 2–4: total momentum
3. columns 5–7: total momenta moment with respect to center of mass
4. column 8: total mass
5. column 9–11: center of mass
6. column 12-14: velocity of center of mass
7. column 15-17: angular velocity
8. column 18-20: position of center of mass with respect to `reference_position`
9. column 21-23: position of center of mass with respect to `reference_position` and `reference_orientation`
10. column 24-26: principal moments of inertia
11. column 27-29: orientation vector of principal axes with respect to the global reference frame
12. column 30-38: moment of inertia at center of mass
13. column 39-47: moment of inertia with respect to `reference_position` and `reference_orientation`

8.20.4 Output

This card does not instantiate any `element`; it rather enables output of selected elements, and it is analogous to that of the `nodes` (see Section 6.7.1):

```
<card> ::= output : <elem_type> , <elem_list> ;

<elem_list> ::=
    { <elem_label>  [ , ... ]
        | range , <elem_start_label> , <elem_end_label> }
```

`elem_type` is a valid element type that can be read as card name in the `elements` block. In case the keyword `range` is used, all existing elements comprised between `elem_start_label` and `elem_end_label` are set; missing ones are silently ignored.

Appendix A

Modal Element Data

A.1 FEM File Format

This section describes the format of the FEM input to the `modal` joint of MBDyn (Section 8.12.31). These data can be obtained, for example:

- from Code Aster (`http://www.code-aster.org/`), as discussed in Section A.2;
- from NASTRAN (`http://www.mscsoftware.com/`) output, using the `femgen` utility, as detailed in Appendix A.3 (in short, it processes binary output from NASTRAN, as defined by means of appropriate ALTER files provided with MBDyn sources, into a file that is suitable for direct input in MBDyn);
- from mboct-fem-pkg (`https://github.com/octave-user/mboct-fem-pkg`) as discussed in Section A.4;
- by manually crafting the output of your favorite FEM analysis: since it is essentially a plain ASCII file, it can be generated in a straightforward manner from analogous results obtained with almost any FEM software, from experiments or manually generated from analytical or numerical models of any kind.

The format is:

```
<comments>
** RECORD GROUP 1,<any comment to EOL; "HEADER">
<comment>
<REV> <NNODES> <NNORMAL> <NATTACHED> <NCONSTRAINT> <NREJECTED>
<comments; NMODES = NNORMAL + NATTACHED + NCONSTRAINT - NREJECTED>
** RECORD GROUP 2,<any comment to EOL; "FINITE ELEMENT NODE LIST">
<FEMLABEL> [...NNODES]
<comments; FEM label list: NNODES integers>
** RECORD GROUP 3,<any comment to EOL; "INITIAL MODAL DISPLACEMENTS">
<MODEDISP> [...NMODES]
<comments; initial mode displacements: NMODES reals>
** RECORD GROUP 4,<any comment to EOL; "INITIAL MODAL VELOCITIES">
<MODEVEL> [...NMODES]
<comments; initial mode velocities: NMODES reals>
** RECORD GROUP 5,<any comment to EOL; "NODAL X COORDINATES">
<FEM_X> [...NNODES]
<comments; FEM node X coordinates>
```

```
** RECORD GROUP 6,<any comment to EOL; "NODAL Y COORDINATES">
<FEM_Y> [...NNODES]
<comments; FEM node Y coordinates>
** RECORD GROUP 7,<any comment to EOL; "NODAL Z COORDINATES">
<FEM_Z> [...NNODES]
<comments; FEM node Z coordinates>
** RECORD GROUP 8,<any comment to EOL; "NON-ORTHOGONALIZED MODE SHAPES">
<comment; NORMAL MODE SHAPE # 1>
    <FEM1_X> <FEM1_Y> <FEM1_Z> <FEM1_RX> <FEM1_RY> <FEM1_RZ>
    [...NNODES]
<comment; NORMAL MODE SHAPE # 2>
    <FEM2_X> <FEM2_Y> <FEM2_Z> <FEM2_RX> <FEM2_RY> <FEM2_RZ>
    [...NNODES]
[...NMODES]
<comments; for each MODE, for each NODE: modal displacements/rotations>
** RECORD GROUP 9,<any comment to EOL; "MODAL MASS MATRIX. COLUMN-MAJOR FORM">
<M_1_1>      [...] <M_1_NMODES>
[...]
<M_NMODES_1> [...] <M_NMODES_NMODES>
<comments; the modal mass matrix in column-major (symmetric?)>
** RECORD GROUP 10,<any comment to EOL; "MODAL STIFFNESS MATRIX. COLUMN-MAJOR FORM">
<K_1_1>      [...] <K_1_NMODES>
[...]
<K_NMODES_1> [...] <K_NMODES_NMODES>
<comments; the modal stiffness matrix in column-major (symmetric?)>
** RECORD GROUP 11,<any comment to EOL; "DIAGONAL OF LUMPED MASS MATRIX">
<M_1_X> <M_1_Y> <M_1_Z> <M_1_RX> <M_1_RY> <M_1_RZ>
[...]
<M_NNODES_X> [...] <M_NNODES_RZ>
<comments; the lumped diagonal mass matrix of the FEM model>
** RECORD GROUP 12,<any comment to EOL; "GLOBAL INERTIA PROPERTIES">
<MASS>
<X_CG> <Y_CG> <Z_CG>
<J_XX> <J_XY> <J_XZ>
<J_YX> <J_YY> <J_YZ>
<J_ZX> <J_ZY> <J_ZZ>
<comments; the global inertia properties of the modal element>
** RECORD GROUP 13,<any comment to EOL; "MODAL DAMPING MATRIX. COLUMN-MAJOR FORM">
<C_1_1>      [...] <C_1_NMODES>
[...]
<C_NMODES_1> [...] <C_NMODES_NMODES>
<comments; the modal damping matrix in column-major (symmetric?)>
** RECORD GROUP 14,<any comment to EOL; "INVARIANT 3. COLUMN-MAJOR FORM">
<I3_1_1> <I3_1_2> <I3_1_3> [...]
<I3_2_1> <I3_2_2> <I3_2_3> [...]
<I3_3_1> <I3_3_2> <I3_3_3> [...]
<comments; the invariant 3 in column-major form>
** RECORD GROUP 15,<any comment to EOL; "INVARIANT 4. COLUMN-MAJOR FORM">
<I4_1_1> <I4_1_2> <I4_1_3> [...]
<I4_2_1> <I4_2_2> <I4_2_3> [...]
<I4_3_1> <I4_3_2> <I4_3_3> [...]
<comments; the invariant 4 in column-major form>
** RECORD GROUP 16,<any comment to EOL; "INVARIANT 8. COLUMN-MAJOR FORM">
<I8_1_1_1> <I8_1_2_1> <I8_1_3_1> <I8_1_1_2> [...] <I8_1_3_NMODES>
```

```
<I8_2_1_1> <I8_2_2_1> <I8_2_3_1> <I8_2_1_2> [...] <I8_2_3_NMODES>
<I8_3_1_1> <I8_3_2_1> <I8_3_3_1> <I8_3_1_2> [...] <I8_3_3_NMODES>
<comments; the invariant 8 in column-major form>
** RECORD GROUP 17,<any comment to EOL; "INVARIANT 5. COLUMN-MAJOR FORM">
<I5_1_1_1> <I5_1_2_1> <I5_1_3_1> [...] <I5_1_NMODES_1> <I5_1_1_2> [...] <I5_1_NMODES_NMODES>
<I5_2_1_1> <I5_2_2_1> <I5_2_3_1> [...] <I5_2_NMODES_1> <I5_2_1_2> [...] <I5_2_NMODES_NMODES>
<I5_3_1_1> <I5_3_2_1> <I5_3_3_1> [...] <I5_3_NMODES_1> <I5_3_1_2> [...] <I5_3_NMODES_NMODES>
<comments; the invariant 5 in column-major form>
** RECORD GROUP 18,<any comment to EOL; "INVARIANT 9- COLUMN-MAJOR FORM">
<I9_1_1_1_1> <I9_1_2_1_1> <I9_1_3_1_1> [...] <I9_1_NMODES_1_1> <I9_1_1_2_1> [...] <I9_1_3_NMODES_NMODES>
<I9_2_1_1_1> <I9_2_2_1_1> <I9_2_3_1_1> [...] <I9_2_NMODES_1_1> <I9_2_1_2_1> [...] <I9_2_3_NMODES_NMODES>
<I9_3_1_1_1> <I9_3_2_1_1> <I9_3_3_1_1> [...] <I9_3_NMODES_1_1> <I9_3_1_2_1> [...] <I9_3_3_NMODES_NMODES>
<comments; the invariant 9 in column-major form>
** RECORD GROUP 19,<any comment to EOL; "MODAL STRESS STIFFENING MATRIX.">
<NSTRESSSTIFFMAT>
<ISTRESSSTIFFINDEX_1>
<K0_1_1_ISTRESSSTIFFINDEX_1>      <K0_1_2_ISTRESSSTIFFINDEX_1>      [...] <K0_1_NMODES_ISTRESSSTIFFINDEX_1>
<K0_2_1_ISTRESSSTIFFINDEX_1>      <K0_2_2_ISTRESSSTIFFINDEX_1>      [...] <K0_2_NMODES_ISTRESSSTIFFINDEX_1>
[...]                             [...]                             [...] [...]
<K0_NMODES_1_ISTRESSSTIFFINDEX_1> <K0_NMODES_2_ISTRESSSTIFFINDEX_1> [...] <K0_3_NMODES_ISTRESSSTIFFINDEX_1>
<ISTRESSSTIFFINDEX_2>
<K0_1_1_ISTRESSSTIFFINDEX_2>      <K0_1_2_ISTRESSSTIFFINDEX_2>      [...] <K0_1_NMODES_ISTRESSSTIFFINDEX_2>
<K0_2_1_ISTRESSSTIFFINDEX_2>      <K0_2_2_ISTRESSSTIFFINDEX_2>      [...] <K0_2_NMODES_ISTRESSSTIFFINDEX_2>
[...]                             [...]                             [...] [...]
<K0_NMODES_1_ISTRESSSTIFFINDEX_2> <K0_NMODES_2_ISTRESSSTIFFINDEX_2> [...] <K0_3_NMODES_ISTRESSSTIFFINDEX_2>
<ISTRESSSTIFFINDEX_NSTRESSSTIFFMAT>
<K0_1_1_ISTRESSSTIFFINDEX_NSTRESSSTIFFMAT>      [...] <K0_1_NMODES_ISTRESSSTIFFINDEX_NSTRESSSTIFFMAT>
<K0_2_1_ISTRESSSTIFFINDEX_NSTRESSSTIFFMAT>      [...] <K0_2_NMODES_ISTRESSSTIFFINDEX_NSTRESSSTIFFMAT>
[...]                                           [...]
<K0_NMODES_1_ISTRESSSTIFFINDEX_NSTRESSSTIFFMAT> [...] <K0_NMODES_NMODES_ISTRESSSTIFFINDEX_NSTRESSSTIFFMAT>
<comments; the set of modal stress stiffening matrices in column-major form>
** END OF FILE
```

An arbitrary number of comment lines may appear where `<comments[...]>` is used; only one comment line must appear where where `<comment[...]>` is used.

The beginning of a record is marked

```
** RECORD GROUP <RID>
```

where the number RID indicates what record is being read. The size of each record, i.e. the number of values that are expected, is defined based on the header record, so MBDyn should be able to detect incomplete or mis-formatted files.

The records contain:

- RECORD GROUP 1, a.k.a. the "header", contains a summary of the contents of the file:
  - REV is a string that indicates the revision number; it is currently ignored;
  - NNODES is the number of (exposed) FEM nodes in the FEM model;
  - NNORMAL is the number of normal modes;
  - NATTACHED is the number of "attached", i.e. static, modes;
  - NCONSTRAINT is the number of constraint modes;
  - NREJECTED is the number of rejected modes.

Currently, the number of available modes is computed as

$$\texttt{NMODES} = \texttt{NNORMAL} + \texttt{NATTACHED} + \texttt{NCONSTRAINT} - \texttt{NREJECTED}$$

because modes are treated in a generalized manner, so there is no need to consider the different types of shapes in a specific manner. Typically, one should set all numbers to 0, except for NNORMAL, which should be set equal to the total number of modes actually present in the data set, regardless of their type. Remember that MBDyn can still select a subset of the available modes to be used in the analysis, so that there is no need to further edit this file.

- RECORD GROUP 2 contains a listing of the NNODES labels of the (exposed) FEM nodes. The labels can be any string of text, and are separated by blanks (as intended by the `isspace(3)` C standard library function). A label cannot thus contain any amount of whitespace.
- RECORD GROUP 3 contains the initial values of the NMODES modal unknowns (optional; set to zero if omitted);
- RECORD GROUP 4 contains the initial values of the NMODES modal unknowns derivatives (optional; set to zero if omitted);
- RECORD GROUP 5 contains the X component of the position of the NNODES FEM nodes in the reference frame attached to the modal node (or to the origin node, if given).
- RECORD GROUP 6 contains the Y component of the data above;
- RECORD GROUP 7 contains the Z component of the data above;
- RECORD GROUP 8 contains the non-orthogonalized components of the NMODES modes; for each mode, the three components X, Y, Z of the modal displacement, and the three components RX, RY, RZ of the linearized modal rotations are listed; each mode shape is separated by a comment line, which typically is

  ```
  **    NORMAL MODE SHAPE #  <N>
  ```

  for readability;
- RECORD GROUP 9 contains the modal mass matrix, i.e. a square, NMODES $\times$ NMODES matrix that contains the reduced mass $\mathbf{m}$ resulting from the multiplication

$$\mathbf{m} = \mathbf{X}^T\mathbf{M}\mathbf{X} \tag{A.1}$$

  When only normal modes are used, it is diagonal. It can be semi-definite positive, or even zero, if a partially or entirely static model is considered.
- RECORD GROUP 10 contains the modal stiffness matrix, i.e. a square, NMODES $\times$ NMODES matrix that contains the reduced stiffness $\mathbf{k}$ resulting from the multiplication

$$\mathbf{k} = \mathbf{X}^T\mathbf{K}\mathbf{X} \tag{A.2}$$

  When only normal modes are used, it is diagonal; in that case, the diagonal contains the modal mass times the square of the eigenvalues, i.e. $k_{ii} = \omega_i^2 m_{ii}$. It should be definite positive; in fact, rigid degrees of freedom that would make it semi-definite should rather be modeled by combining separate modal submodels by way of multibody connections, so that the multibody capability to handle finite relative displacements and rotations is exploited. Note however that the positive-definiteness of the generalized stiffness matrix is not a requirement.

- RECORD GROUP 11 contains the lumped inertia matrix associated with the NNODES (exposed) FEM nodes; for each node, the X, Y, Z, RX, RY and RY inertia coefficients are listed (they can be zero). The nodal mass coefficients Y and Z should be equal to coefficient X. The resulting diagonal matrix should satisfy the constraint illustrated in Equation A.1.

- RECORD GROUP 12 contains the global inertia properties of the modal element: the total mass, the three components of the position of the center of mass in the local frame of the FEM model, and the 3×3 inertia matrix with respect to the center of mass, which is supposed to be symmetric and positive definite (or semi-definite, if the model is static).

- RECORD GROUP 13 contains the modal damping matrix, i.e. a square, NMODES $\times$ NMODES matrix that contains the reduced damping $\mathbf{c}$ resulting from the multiplication

$$\mathbf{c} = \mathbf{X}^T \mathbf{C} \mathbf{X} \tag{A.3}$$

  It can be semi-definite positive, or even zero, if no damping is considered. This record is optional. If given, it is used in the analysis unless overridden by any damping specification indicated in the modal joint element definition. The generalized damping matrix appeared in MBDyn 1.5.3.

- RECORD GROUP 14 contains $\mathcal{I}_3$, a $3 \times$ NMODES matrix that contains static coupling between rigid body and FEM node displacements. See the tecman.

- RECORD GROUP 15 contains $\mathcal{I}_4$, a $3 \times$ NMODES matrix that contains static coupling between rigid body rotations and FEM node displacements. See the tecman.

- RECORD GROUP 16 contains $\mathcal{I}_8$, a $3 \times 3 \times$ NMODES matrix which is required to consider centrifugal loads. See the tecman.

- RECORD GROUP 17 contains $\mathcal{I}_5$, a $3 \times$ NMODES $\times$ NMODES matrix that contains the static coupling between FEM node displacements. See the tecman.

- RECORD GROUP 18 contains $\mathcal{I}_9$, a $3 \times 3 \times$ NMODES $\times$ NMODES matrix which is required to consider the so called "spin softening" effect. See the tecman.

- RECORD GROUP 19 contains $\boldsymbol{K}_{0\omega}$, $\boldsymbol{K}_{0r}$, $\boldsymbol{K}_{0t}$, $\boldsymbol{K}_{0F}$ and $\boldsymbol{K}_{0M}$, a set of NMODES $\times$ NMODES matrices which are required to consider the so called "stress-stiffening" effect. See the tecman. In order to reduce the computational cost, it is not required to provide all $12 + 6 \times$ NNODES matrices. Instead only a subset of available matrices may be provided. Each matrix is identified by it's <ISTRESSSTIFFINDEX> tag according to the following table:

$$\begin{aligned}
\boldsymbol{K}_{0\omega} &\rightarrow [1 \dots 6] \\
\boldsymbol{K}_{0r} &\rightarrow [7 \dots 9] \\
\boldsymbol{K}_{0t} &\rightarrow [10 \dots 12] \\
\boldsymbol{K}_{0F_1} &\rightarrow [13 \dots 15] \\
\boldsymbol{K}_{0M_1} &\rightarrow [16 \dots 18] \\
\boldsymbol{K}_{0F_2} &\rightarrow [19 \dots 21] \\
\boldsymbol{K}_{0M_2} &\rightarrow [22 \dots 24]
\end{aligned} \tag{A.4}$$

Note that RECORD GROUP 11 and RECORD GROUP 12 used to be mutually exclusive in earlier versions. The reason for accepting RECORD GROUP 12 format, regardless of RECORD GROUP 11 presence, is related to the fact that in many applications the FEM nodal inertia may not be available and, at the same

time, a zero-order approximation of the inertia is acceptable. The reason for allowing both is that when `RECORD GROUP 11` is present, its data were originally used to compute all inertia invariants, including the rigid-body ones that can be provided by `RECORD GROUP 12`. This in some cases may represent an unacceptable approximation; `RECORD GROUP 12` data can be used to replace those invariants by a better estimate, when available.

Both `RECORD GROUP 11` and `RECORD GROUP 12` may be absent. This case only makes sense when a zero-order approximation of the inertia is acceptable and the rigid-body motion of the modal element is not allowed, i.e. the element is clamped to the ground (see the keyword `clamped` in Section 8.12.31 for details).

Although the format loosely requires that no more than 6 numerical values appear on a single line, MBDyn is very forgiving about this and can parse the input regardless of the formatting within each record. However this liberality may lead to inconsistent or unexpected parsing behavior, so use at own risk.

A.1.1 Usage

The modal joint element is intended to model generically flexible elements whose flexibility is modeled in terms of linear superposition of deformable shapes expressed in a floating frame of reference (FFR).

The reference (rigid-body) motion is handled by a special multibody node, the `modal` node. In practice, this node "carries around" the (local) reference frame in which the finite element model (FEM) is defined. When the modal joint is to be directly connected to the ground, the modal node can be omitted using the `clamped` keyword instead of the node's label.

Interaction of the modal joint with the multibody environment should occur through the use of *interface* nodes. Interface nodes are regular structural nodes that are clamped to their corresponding FEM nodes; `static` nodes are recommended, to save 6 equations, unless one intends to explicitly connect additional inertia to them.

(Note: in case the FEM mode shapes are defined as so-called *attached modes*, i.e. in case the FE model was clamped at the origin of its reference frame, the `modal` node can be directly interfaced to the multibody environment as a regular interface node; otherwise the user needs to understand that such node is interfacing the *origin* of the FEM reference frame.)

For example, if you want to permit arbitrary relative rotation between two independent instances of the modal joint element, which are connected by a `revolute hinge` joint, you should define the two instances of the modal element each with its own modal node, and each with an interface node at the respective locations of the hinge. At this point, the interface nodes can be connected by a regular `revolute hinge` joint element.

A.1.2 Example: Dynamic Model

As an example, a very simple, hand-made FEM model file is presented below. It models a FEM model made of three aligned nodes, where inertia is evenly distributed. Note that each line is prefixed with a two-digit line number that is not part of the input file. Also, for readability, all comments are prefixed by "**", in analogy with the mandatory "** RECORD GROUP" lines, although not strictly required by the format of the file.

```
01  ** MBDyn MODAL DATA FILE
02  ** NODE SET "ALL"
03
04
05  ** RECORD GROUP 1, HEADER
06  **   REVISION,  NODE,  NORMAL, ATTACHMENT, CONSTRAINT, REJECTED MODES.
07        REV0         3         2         0         0         0
```

```
08  **
09  ** RECORD GROUP 2, FINITE ELEMENT NODE LIST
10       1001 1002 1003
11
12  **
13  ** RECORD GROUP 3, INITIAL MODAL DISPLACEMENTS
14   0 0
15  **
16  ** RECORD GROUP 4, INITIAL MODALVELOCITIES
17   0 0
18  **
19  ** RECORD GROUP 5, NODAL X COORDINATES
20   0
21   0
22   0
23  **
24  ** RECORD GROUP 6, NODAL Y COORDINATES
25  -2.
26   0
27   2.
28  **
29  ** RECORD GROUP 7, NODAL Z COORDINATES
30   0
31   0
32   0
33  **
34  ** RECORD GROUP 8, MODE SHAPES
35  **    NORMAL MODE SHAPE #  1
36  0 0 1 0 0 0
37  0 0 0 0 0 0
38  0 0 1 0 0 0
39  **    NORMAL MODE SHAPE #  2
40  1 0 0 0 0 0
41  0 0 0 0 0 0
42  1 0 0 0 0 0
43  **
44  ** RECORD GROUP 9, MODAL MASS MATRIX
45  2 0
46  0 2
47  **
48  ** RECORD GROUP 10, MODAL STIFFNESS MATRIX
49  1 0
50  0 1e2
51  **
52  ** RECORD GROUP 11, DIAGONAL OF LUMPED MASS MATRIX
53  1 1 1 1 1 1
54  1 1 1 1 1 1
55  1 1 1 1 1 1
```

The corresponding global inertia properties are:

$$m = 3 \tag{A.5}$$

$$\boldsymbol{x}_{CM} = \left\{ \begin{array}{c} 0 \\ 0 \\ 0 \end{array} \right\} \tag{A.6}$$

$$\mathbf{J} = \left[\begin{array}{ccc} 11 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 11 \end{array} \right] \tag{A.7}$$

A.1.3 Example: Static Model

As an example, a very simple, hand-made FEM model file is presented below. It models a FEM model made of three aligned nodes, where inertia is only associated with the mid-node. As a consequence, the three mode shapes must be interpreted as static shapes, since the modal mass matrix is null. Note that each line is prefixed with a two-digit line number that is not part of the input file. Also, for readability, all comments are prefixed by "**", in analogy with the mandatory "** RECORD GROUP" lines, although not strictly required by the format of the file.

```
01  ** MBDyn MODAL DATA FILE
02  ** NODE SET "ALL"
03
04
05  ** RECORD GROUP 1, HEADER
06  **   REVISION,  NODE,  NORMAL, ATTACHMENT, CONSTRAINT, REJECTED MODES.
07        REV0         3         2         0         0         0
08  **
09  ** RECORD GROUP 2, FINITE ELEMENT NODE LIST
10       1001 1002 1003
11
12  **
13  ** RECORD GROUP 3, INITIAL MODAL DISPLACEMENTS
14   0 0
15  **
16  ** RECORD GROUP 4, INITIAL MODALVELOCITIES
17   0 0
18  **
19  ** RECORD GROUP 5, NODAL X COORDINATES
20   0
21   0
22   0
23  **
24  ** RECORD GROUP 6, NODAL Y COORDINATES
25  -2.
26   0
27   2.
28  **
29  ** RECORD GROUP 7, NODAL Z COORDINATES
30   0
31   0
32   0
33  **
34  ** RECORD GROUP 8, MODE SHAPES
```

```
35  **    NORMAL MODE SHAPE #  1
36  0 0 1 0 0 0
37  0 0 0 0 0 0
38  0 0 1 0 0 0
39  **    NORMAL MODE SHAPE #  2
40  1 0 0 0 0 0
41  0 0 0 0 0 0
42  1 0 0 0 0 0
43  **
44  ** RECORD GROUP 9, MODAL MASS MATRIX
45  0 0
46  0 0
47  **
48  ** RECORD GROUP 10, MODAL STIFFNESS MATRIX
49  1 0
50  0 1e2
51  **
52  ** RECORD GROUP 12, GLOBAL INERTIA
53   3
54   0  0  0
55  11  0  0
56   0  3  0
57   0  0 11
```

The same global inertia properties of the model in Section A.1.2 have been used; as a result, the two models show the same rigid body dynamics behavior, but the dynamic model also shows deformable body dynamic behavior, while the static one only behaves statically when straining is involved.

A.2 Code Aster Procedure

The `fem_data_file` can be generated by Code Aster, using the macro provided in the `cms.py` file, which is located in directory `contrib/CodeAster/cms/` of the distribution, and is installed in directory `$PREFIX/share/py/`.

Preparing the input for Code Aster essentially requires to prepare the bulk of the mesh either manually or using some meshing tools (e.g. `gmsh`), as illustrated in the related documentation and tutorials, and then writing a simple script in `python` (`http://www.python.org/`) that defines the execution procedure.

To produce the data for MBDyn's modal element, a specific macro needs to be invoked during the execution of Aster. The macro instructs Aster about how the solution procedure needs to be tailored to produce the desired results, and then generates the `.fem` file with all the requested data.

The steps of the procedure are as follows:

1. prepare a Code Aster input model (the `.mail` file), containing the nodes, the connections and at least a few groups of nodes, as indicated in the following;
2. prepare a Code Aster command file (the `.comm` file), following, for example, one of the given templates;
3. as soon as the model, the materials, the element properties and any optional boundary constrains are defined, call the `CMS` macro with the appropriate parameters. The syntax of the macro is:

```
CMS(
    MAILLAGE = <mesh_concept> ,
    [ INTERFACE = <interface_node_set_name> , ]
```

```
        EXPOSED = _F( { GROUP_NO = <exposed_node_set_name> | TOUT = 'OUI' } ),
        MODELE = <model_concept> ,
        CARA_ELEM = <element_properties_concept> ,
        CHAM_MATER = <material_properties_concept> ,
        [ CHAR_MECA = <boundary_conditions_concept> ,
        [ OPTIONS = _F(
            [ NMAX_FREQ = <maximum_number_of_frequencies> ]
        ), ]
        OUT = _F(
            TYPE = { 'MBDYN' } ,
                [ PRECISION = <number_of_digits> , ]
                [ DIAG_MASS = { 'OUI' | 'NON' } , ]
                [ RIGB_MASS = { 'OUI' | 'NON' } , ]
                FICHIER = <output_file_name>
        )
    );
```

where:

- MAILLAGE is the name of a python object returned by LIRE_MAILLAGE;
- INTERFACE is a string containing the name of the group of nodes, defined in the mesh file, that will be used as interface to add Craig-Bampton shapes to the base (if omitted, only normal modes are used);
- EXPOSED allows two formats:
  - GROUP_NO is a string containing the name of the group of nodes, defined in the mesh file, that will be exposed, i.e. that will appear in the `.fem` file generated by this macro;
  - TOUT is a string that must be set to 'OUI', indicating that all nodes will be exposed (technically speaking, all nodes involved in at least one connection);
- MODELE is the name of a python object returned by AFFE_MODELE;
- CARA_ELEM is the name of a python object returned by AFFE_CARA_ELEM;
- CHAM_MATER is the name of a python object returned by AFFE_MATERIAU;
- CHAR_MECA is the name of a python object returned by AFFE_CHAR_MECA (optional);
- OPTIONS allows a set of generic options to be passed:
  - NMAX_FREQ is the number of frequencies expected from the eigenanalysis, and thus the number of normal modes that will be added to the modal base; the smallest frequencies are used;
- OUT describes the parameters related to the type of output the macro generates; right now, only output in the format specified in this section is available:
  - TYPE indicates the type of output the macro is requested to generate; it must be 'MBDYN';
  - FICHIER ("file" in French) is the name of the file where the generated data must be written; it must be a fully qualified path, otherwise it is not clear where the file is actually generated;
  - PRECISION contains the number of digits to be used when writing the data (for example, PRECISION = 8 results in using the format specifier `%16.8e`);
  - DIAG_MASS requests the output of the diagonal of the mass matrix (`RECORD GROUP 11`); this option really makes sense only if all nodes are output (namely, if the nodes set passed to EXPOSED consists in all nodes);

– DIAG_MASS requests the output of the rigid-body inertia properties (RECORD GROUP 12);

the macro returns a python object consisting in the generalized model it used to generate the file, as returned by MACR_ELEM_DYNA.

4. prepare a .export file, either following the examples or using astk;
5. run Aster.

Annotated examples are provided in directory contrib/CodeAster/cms/.

A.3 NASTRAN Procedure

The fem_data_file can be generated using NASTRAN, by means of the ALTER cards provided in directory etc/modal.d/ of the distribution.

The steps of the procedure are as follows:

1. prepare a NASTRAN input card deck, made of bulk data, eigenanalysis properties data and/or static loads (details about this phase are currently missing and will be provided in future releases of the manual).
2. complete the NASTRAN input file by putting some specific ALTER cards. In detail:
   (a) the file MBDyn_NASTRAN_alter_1.nas contains ALTER definitions for static solutions; appropriate loading subcases for each solution must be provided in the case control and in the bulk data sections of the input file;
   FIXME: I don't know how to use static shapes only.
   (b) the file MBDyn_NASTRAN_alter_2.nas contains ALTER definitions for eigenanalysis solutions; an appropriate eigenanalysis method, with the related data card must be provided in the case control and in the bulk data sections of the input file;
   FIXME: I don't know how to use this together with static shapes; I only get the normal mode shapes, even if the matrices are complete.
   (c) the file MBDyn_NASTRAN_alter_3.nas contains ALTER definitions for eigenanalysis solutions; an appropriate eigenanalysis method, with the related data card must be provided in the case control and in the bulk data sections of the input file;
   Note: this works; see tests/modal/beam.README.

   Exactly one of these files must be included at the very top of the NASTRAN input file; they already include the appropriate SOL statement, so the input file must begin with

```
$ Replace '#' below with number that matches your needs
INCLUDE 'MBDyn_NASTRAN_alter_#.nas'
CEND
$... any other executive control and bulk data card
```

   The static solution of case (a: SOL 101) and the eigensolution of case (b: SOL 103) need to be performed in sequence; if only the eigensolution is to be used, the ALTER file of case (c: SOL 103) must be used. The static solution of case (a) generates a binary file mbdyn.stm; the eigensolutions of cases (b–c) generate two binary files, mbdyn.mat and mbdyn.tab, which, in case (b), include the static solutions as well. The ALTER currently included in the MBDyn distribution work correctly only with the following PARAM data card:

```
PARAM,POST,-1
```

3. Run NASTRAN.

4. Run the tool `femgen`, which transforms the above binary files into the `fem_data_file`. The file name is currently requested as terminal input; this is the name of the file that will be used in the input model for `mbdyn`. Conventionally, the `.fem` extension is used.

A.4 Procedure for mboct-fem-pkg

mboct-fem-pkg is a Finite Element Toolkit for GNU-Octave. It can be used to generate modal element data for MBDyn. In addition to that, postprocessing of the combined output from several modal elements is possible using Gmsh. The following GNU-Octave code gives an example on how to create modal element data for MBDyn with mboct-fem-pkg.

```
## load the package
pkg load mboct-fem-pkg;
## load the mesh file in Gmsh format
mesh = fem_pre_mesh_import("conrod_gmsh.msh", "gmsh");
mesh.material_data.E = 70000e6; ## Young's modulus [Pa]
mesh.material_data.nu = 0.3; ## Poisson's ratio [1]
mesh.material_data.rho = 2700; ## density [kg/m^3]
mesh.material_data.alpha = 1e-8; ## mass damping [s^-1]
mesh.material_data.beta = 1e-7; ## stiffness damping [s]
## allocate material assignment
mesh.materials.iso20 = zeros(rows(mesh.elements.iso20), 1, "int32");
## locate a group of solid elements with id 10
grp_idx_beam = find([[mesh.groups.iso20].id] == 10);
## locate a group of surface elements with id 12
grp_idx_clamp = find([[mesh.groups.quad8].id] == 12);
## assign the material number one to all elements in group 10
mesh.materials.iso20(mesh.groups.iso20(grp_idx_beam).elements) = 1;
## insert a new node used as modal node
cms_opt.nodes.modal.number = rows(mesh.nodes) + 2;
## define the name of the modal node used in MBDyn's input files
cms_opt.nodes.modal.name = "node_id_modal";
## insert a new node used as interface node
cms_opt.nodes.interfaces.number = rows(mesh.nodes) + 1;
## define the name of the interface node in MBDyn's input files
cms_opt.nodes.interfaces.name = "node_id_interface1";
## define the position of the modal node in the FEM reference frame
mesh.nodes(cms_opt.nodes.modal.number, 1:3) = [0, 0, 0];
## define the position of the interface node in the FEM reference frame
mesh.nodes(cms_opt.nodes.interfaces.number, 1:3) = [0.1, 0, 0];
## create an RBE3 element for the interface node
## all nodes inside group 13 will be coupled to the interface node
mesh.elements.rbe3 = fem_pre_mesh_rbe3_from_surf(mesh, 13, cms_opt.nodes.interfaces.number, "quad8");
## compute 20 normal modes
cms_opt.modes.number = 20;
## define the name of the modal element in MBDyn's input files
```

```
cms_opt.element.name = "elem_id_modal";
## allocate the DOF status for all nodes
load_case_dof.locked_dof = false(size(mesh.nodes));
## lock all degrees of freedom of the modal node
load_case_dof.locked_dof(cms_opt.nodes.modal.number, :) = true;
## lock all degrees of freedom of clamped nodes
load_case_dof.locked_dof(mesh.groups.quad8(grp_idx_clamp).nodes, :) = true;
## generate modal element data
[mesh_cms, ...
 mat_ass_cms, ...
 dof_map_cms, ...
 sol_eig_cms, ...
 cms_opt, ...
 sol_tau_cms] = fem_cms_create2(mesh, load_case_dof, cms_opt);
## write MBDyn's input files conrod_mbd.fem and conrod_mbd.elm
fem_cms_export("conrod_mbd", mesh_cms, dof_map_cms, mat_ass_cms, cms_opt);
```

A.5 Procedures for Other FEA Software

Currently, no (semi-)automatic procedures are available to extract modal element data from other FEA software. If you need to use any other software, and possibly a free software, you'll need to design your own procedure, which may go from a very simple script (shell, awk, python) for handling of textual output, to a fully featured modeling and translation interface. If you want to share what you developed with the MBDyn project, feel free to submit code, procedures or just ideas and discuss them on the `mbdyn-users@mbdyn.org` mailing list.

Appendix B

Modules

As a general rule, entities defined as run-time loadable modules by definition may not appear in the input manual, as they are temporary, experimental or in any case elusive by their very nature. They are expected to honor the `help` keyword as their first argument. If present it should trigger the printing on the screen of the input syntax. If not followed by any arguments, the simulation is expected to end with no error.

Example Print help on screen and continue with the simulation:

```
user defined: 10, wheel2, help,
    10, ...
```

Print help on screen and gently stop:

```
user defined: 10, wheel2, help;
```

B.1 Element Modules

Recall that element modules are invoked as `user defined` elements, whose syntax is

```
<elem_type> ::= user defined

<normal_arglist> ::= <name> [ , <module_data> ]
```

B.1.1 Module-aerodyn

Authors: Fanzhong Meng and Pierangelo Masarati
This module implements NREL's AeroDyn v 12.58 wind turbine aerodynamic loads.

B.1.2 Module-asynchronous_machine

Author: Reinhard Resch
This module implements an asynchronous electric motor.

B.1.3 Module-cyclocopter

Author: Mattia Mattaboni (turned into module by Pierangelo Masarati)
This module implements inflow models for cycloidal rotors.

```
<name> ::= cycloidal no inflow

<module_data> ::=
    <aircraft_node_label> ,
    [ orientation , (OrientationMatrix) <orientation> , ]
    <rotor_node_label>

<name> ::= cycloidal { uniform 1D | uniform 2D | Polimi }

<module_data> ::=
    <aircraft_node_label> ,
    [ orientation , (OrientationMatrix) <orientation> , ]
    <rotor_node_label>
    (bool) <average> ,
    <rotor_radius> ,
    <blade_span>
    [ , delay , (DriveCaller) <delay> ]
    [ , omegacut , <cut_frequency> ]
    [ , timestep , <time_step> ]
```

The optional `orientation` is required when axis 3 of the aircraft node is not normal to the rotor axis, it is defined with respect to the aircraft node; axis 3 of the rotor node must be aligned with the rotor axis; the choices of axes 1 and 2 only influence the orientation of the output.

Example 1.

```
user defined: CURR_DRUM,
    cyclocopter no inflow,
    AIRCRAFT,
    orientation,
        reference, node,
            1, 0., 0., 1.,
            3, 0., 1., 0.,
    AIRCRAFT;
```

The latter defines a cyclocopter with no inflow. It essentially is only useful to obtain the output data.

Example 2.

```
user defined: CURR_DRUM,
    cyclocopter uniform 2D,
    AIRCRAFT,
    orientation,
        reference, node,
        1, 0., 0., 1.,
        3, 0., 1., 0.,
    CURR_DRUM,
```

```
0, # no average
# 1, # average
ROTOR_DIAMETER/2.,
SPAN,
delay,
    const, 0.5,
omegacut, OMEGA_CUT,
kappa, 1.3, # hover empirical correction coefficient
timestep, DT,
output, yes;
```

While this one defines a cyclocopter with a two-dimensional inflow model.

Output

The output is obtained in plain text in the **.usr** file where the columns represent,

- 1: element label
- 2–4: rotor force on the aircraft in local x, y and z directions in the optionally transformed, if <orientation> is defined, aircraft reference frame
- 5–7: rotor moment about the aircraft in local x, y and z directions in the optionally transformed, if <orientation> is defined, aircraft reference frame
- 8: mean magnitude of the inflow velocity, based on momentum theory
- 9–16: depends on inflow model

B.1.4 Module-fab-electric

Author: Eduardo Okabe
This module implements several electric components.

BJT

```
<name> ::= bjt

<module_data> ::=
    { npn | pnp } ,
    <collector_electric_node_label> ,
    <base_electric_node_label> ,
    <emitter_electric_node_label> ,
    (real) <base_to_emitter_leakage_saturation_current> ,
    (real) <base_to_collector_leakage_saturation_current> ,
    (real) <ideal_maximum_forward_beta> ,
    (real) <ideal_maximum_reverse_beta> ,
    [ , thermal voltage , (real) <thermal_voltage> ]
```

Capacitor

```
<name> ::= capacitor

<module_data> ::=
    <electric_node_1_label> ,
    <electric_node_2_label> ,
    (real) <capacitance>
```

Output. The format is:

- the label of the element;
- the current in the element (I in NetCDF format);

Diode

```
<name> ::= diode

<module_data> ::=
    <electric_node_1_label> ,
    <electric_node_2_label> ,
    (real) <forward_saturation_current> ,
    (real) <forward_ideality_factor> ,
    (real) <breakdown_current> ,
    (real) <breakdown_voltage> ,
    (real) <reverse_ideality_factor>
    [ , thermal voltage , (real) <thermal_voltage> ]
```

Electrical source

```
<name> ::= electrical source

<module_data> ::=
    { current | voltage } ,
        [ control , { current | voltage } ,
            <input_electric_node_1_label> ,
            <input_electric_node_2_label> , ]
    <output_electric_node_1_label> ,
    <output_electric_node_2_label> ,
    (DriveCaller) <source_driver>
```

Ideal transformer

```
<name> ::= ideal transformer

<module_data> ::=
    <input_electric_node_1_label> ,
    <input_electric_node_2_label> ,
    <output_electric_node_1_label> ,
    <output_electric_node_2_label> ,
    (DriveCaller) <tranformer_ratio>
```

Inductor

```
<name> ::= inductor

<module_data> ::=
    <electric_node_1_label> ,
    <electric_node_2_label> ,
    (real) <inductance>
```

Output. The format is:

- the label of the element;
- the current in the element (I in NetCDF format);

Operational amplifier

```
<name> ::= operational amplifier

<module_data> ::=
    <input_electric_node_1_label> ,
    <input_electric_node_2_label> ,
    <reference_electric_node_1_label> ,
    <output_electric_node_2_label>
    [ , gain , (real) <gain> ]
    [ , input resistance , (real) <input_resistance> ]
```

Proximity sensor

```
<name> ::= proximity sensor

<module_data> ::=
    <struct_node_1_label> ,
        [ , position, (Vec3) <node_1_offset> , ]
    <struct_node_2_label> ,
        [ , position, (Vec3) <node_2_offset> , ]
    <electric_node_1_label> ,
    <electric_node_2_label> ,
    (ScalarFunction) <distance_to_voltage_conversion_function>
```

Resistor

```
<name> ::= resistor

<module_data> ::=
    <electric_node_1_label> ,
    <electric_node_2_label> ,
    (real) <resistance>
```

Output. The format is:

- the label of the element;
- the current in the element (I in NetCDF format);

Switch

```
<name> ::= switch

<module_data> ::=
    <electric_node_1_label> ,
    <electric_node_2_label> ,
    (DriveCaller) <switch_state>
```

where switch_state is a drive that determines the state of the switch (0: open, 1: closed).
NOTE: untested!

B.1.5 Module-fab-motion

Author: Eduardo Okabe
This module implements several joints.

Gear joint

```
<name> ::= gear joint
```

Linear transmission joint

```
<name> ::= linear transmission joint
```

Motion transmission joint

```
<name> ::= motion transmission joint
```

Smooth step

```
<name> ::= smooth step
```

B.1.6 Module-fab-sbearings

Author: Eduardo Okabe
This module implements several joints.

Hydrodynamic bearing

```
<name> ::= hydrodynamic bearing
```

Rolling bearing

```
<name> ::= rolling bearing
```

B.1.7 Module-hfelem

Author: Pierangelo Masarati
This element was sponsored by Hutchinson CdR

This element produces a harmonic excitation with variable frequency, which is used to control the time marching simulation to support the extraction of the harmonic response.

The element produces a harmonic signal, of specified frequency, and monitors a set of signals produced by the analysis. When the fundamental harmonic of the monitored signals' response converges, the frequency of the excitation is changed, according to a prescribed pattern. The time step is changed accordingly, such that a period is performed within a prescribed number of time steps of equal length. The (complex) magnitude of the fundamental harmonic of each signal is logged at convergence.

```
<name> ::= harmonic excitation

<arglist> ::=
    inputs number , (integer) <inputs_number> ,
        (DriveCaller) <input_signal>
                [ { test | output | target } , { true | false | (bool) <value> } ]
            [ , ... ] , # inputs_number occurrences
    [ output format , { complex | magnitude phase }, [ normalized , ] ]
    steps number , (integer) <steps_number> ,
    [ max delta t , (DriveCaller) <max_delta_t> ,]
    initial angular frequency , (real) <omega0> ,
        [ max angular frequency , { forever | (real) <omega_max> } , ]
        angular frequency increment ,
            { additive , (real) <omega_add_increment>
                | multiplicative , (real) <omega_mul_increment>
                | custom , (real) <omega1> [ , ... ] , last } ,
    [ initial time , (real) <initial_time> ,
        [ prescribed time step , (DriveCaller) <initial_time_step> , ] ]
    [ RMS test,
        [ RMS target, (DriveCaller) <RMS_target_value>
            [ , initial amplitude , (real) <initial_amplitude> ]
            [ , overshoot , (real) <overshoot> ] , ]
        [ RMS prev periods, (integer) <RMS_prev_periods> ] ]
    [ tolerance , (real) <tolerance> , ]
    min periods , (integer) <min_periods>
    [ , max periods , (integer) <max_periods>
        [ , no convergence strategy , { continue | abort } ] ]
    [ , timestep drive label , (unsigned) <timestep_label> ]
    [ , print all periods ]
    [ , write after convergence periods , (integer) <write_after_convergence_periods>
        [ , frequency , (integer) <write_frequency_after_convergence> ] ]
```

with

- `inputs_number` > 0; this field indicates the number of signals – or measures – that are used to evaluate the first harmonic of the response (they are the *inputs* to the harmonic analysis process); it is suggested that the `node` (see Section 2.6.22), `element` (see Section 2.6.13) drive callers, either alone or in combination with other drive callers, are used to extract relevant measures of the output

of the system; If test is true, the signal will be used to evaluate convergence, otherwise it will be ignored; If output is true, the fundamental harmonic of the signal will appear in the output, otherwise it will be ignored; by default, i.e. if neither of the keywords test nor output are used, the signal is used to test for convergence and appears in the output; If target is true, and option RMS target is an argument, the signal is used to change amplitude until the signal's RMS amplitude matches RMS_target_value, within the tolerance.

- output format; coose the output format: either complex (default) of magnitude phase; if normalized the output is normalized with respect to the amplitude (see B.1.7 for the output description);
- steps_number > 1; the number of time steps within a single period, such that

$$\Delta t = \frac{2\pi}{\omega \cdot \texttt{steps\_number}}$$

- max_delta_t; drive caller that contains a function of the excitation angular frequency (passed as Var) returning the maximum value of the time step Δt; if the time step computed using steps_number is greater than the value returned by max_delta_t then the time step is reduced according to $\Delta t = \Delta t / \text{ceil}(\Delta t / \texttt{max\_delta\_t})$. The value returned by max_delta_t must always be greater than 0.
- omega0 > 0, the initial excitation angular frequency;
- omega_max $>$ omega0; if the max angular frequency field is not present, or if the keyword forever is used, the simulation ends when the final time (as defined in the problem block) is reached.

  Since the amount of simulation time required to reach a certain excitation angular frequency cannot be determined a priori, it is suggested that final time (in the problem block) is set to forever, and termination is controlled using the omega_max field, or the custom variant of the angular frequency increment field;
- omega_add_increment > 0 when the angular frequency increment pattern is additive;
- omega_mul_increment > 1 when the angular frequency increment pattern is multiplicative;
- $\texttt{omega}_i >$ $\texttt{omega}_{i-1}$ when the angular frequency increment pattern is custom; in this latter case, the simulation ends either when the last angular frequency value is reached or when $\texttt{omega}_i >$ omega_max, if the latter is defined.

  The list of custom angular frequencies is terminated by the keyword last;
- initial_time; the simulation time after which the elements begins its action
- initial_time_step; optional drive caller that returns the prescribed time step for $t <=$ initial_time; initial_time_step(initial_time) needs to be equal to the time step computed by the element for the first forcing frequency.
- RMS test is a keyword to perform the test not on the difference between the values of the test functions, but on their RMSs; the relative difference of each test function RMS between two periods needs to be $<=$ tolerance.
- RMS target modifies the RMS convergence test: convergence is achieved if the target function RMS also matches, within tolerance, the value returned by the drive RMS_target_value, which is a function of the excitation angular frequency (passed as Var). This is accomplished by changing the module amplitude private data. This option makes sense if and only if there is a single target variable. If min_periods is specified then the periods counter is reset also for an amplitude change, and not only for a frequency change.

- `initial_amplitude` is the initial value of the private data `amplitude`.
- `overshoot`: multiplicated to `tolerance` gives the value by which the upper and lower values of the old RMSs are overestimated and underestimated, respectively, when computing the approximate gradient needed to compute the next value of the private data `amplitude` with the Newton's method.
- `RMS_prev_periods`: number of periods preceding the current period whose test signals RMS amplitude are compared to the current period test signals RMS amplitude to check convergence; default value is 2.
- `min_periods` > 1 is the minimum number of periods that must be performed before checking for convergence.
- `max_periods` $>$ `min_periods` is the maximum number of periods that can be performed for convergence. In case convergence is not achieved, if the value of `no convergence strategy` is
  - `continue`: the frequency is marked as not converged (the value of the number of periods for convergence is set to $-$`max_periods`) and the analysis moves to the subsequent frequency;
  - `abort`: the analysis is aborted.
- `timestep_label` is the label of the timestep drive caller, as defined in the `strategy: change` statement in the `initial value` block, and postponed until the definition of the user-defined element. Using this keyword replaces the need to manually define an `element` drive caller for the timestep (see the discussion of the `timestep` private datum).
- `print all periods` is a keyword to print a row in the *user-defined* elements output file at the end of each period, and not only at convergence for each specific excitation frequency.
- `write_after_convergence_periods` $>= 0$ is the number of periods to be run after convergence is achieved; default value is 0. This option makes sense if the output of the simulation is controlled by an `output meter` using the private data `output`, which is a boolean variable defined by this element that is equal to 1 when the output must be written to the `.usr` file or for `write_after_convergence_periods` periods after convergence.
- `write_frequency_after_convergence` causes the private data `output` to be equal to 1 every `write_frequency_after_convergence` timesteps during the `write_after_convergence_periods` periods; the default value is 1.

Output. Output in text format takes place in the *user-defined* elements output file (the one with the `.usr` extension).

The output of this element differs a bit from that of regular ones, since it only occurs at convergence for each specific excitation frequency (except for when `print all periods` is specified as a keyword), which in turn can only occur at the end of a period.

> *Beware that, in case other user-defined elements are present in the model, this might "screw up" any regularity in the output file.*

Each output row contains:

1) the label (unsigned)

2) the time at which convergence was reached (real)

3) the angular frequency (real)

4) how many periods were required to reach convergence (unsigned)

5–?) fundamental frequency coefficient of output signals; for each of them,

- the real and the imaginary part are output (pairs of reals) if `output format` is either not set or set equal to `complex`; they are both divided by `amplitude` if `output format` is `normalized`;
- the magnitude and phase are output (pair of reals) if `output format` is set equal to `magnitude phase`; the magnitude is divided by `amplitude` if `output format` is `normalized`.

last) in case that the argument `RMS target` was provided, the value of private data `amplitude` and the RMS of the target signal.

For example, to plot the results of the first output signal in Octave (or Matlab), one can use:

```
octave:1> data = load('output_file.usr');
octave:2> omega = data(:, 3);
octave:3> x = data(:, 5) + j*data(:, 6);
octave:4> figure; loglog(omega, abs(x));
octave:5> figure; semilogx(omega, atan2(imag(x), real(x))*180/pi);
```

No output in NetCDF format is currently available.

Private Data. This element exposes several signals in form of private data. The first two are fundamental for the functionality of this element.

- `timestep`, i.e. the time step to be used in the simulation; to this end, the analysis (typically, an `initial value` problem) must be set up with a `problem` block containing

  ```
  strategy : change , postponed , (unsigned) <timestep_drive_label> ;
  ```

  Then, after instantiating the `hfelem`, one must reference the time step drive, with the statement

  ```
  drive caller : (unsigned) <timestep_drive_label> ,
      element , (unsigned) <hfelem_label> , loadable ,
      string , "timestep" , direct ;
  ```

  Alternatively, one can use the optional keyword `timestep drive label` to pass the element the label of the timestep drive caller, which is directly instantiated by the element.

- `excitation`, i.e. the harmonic forcing term, which is a sine wave of given frequency and `amplitude` amplitude. This signal must be used to excite the problem (e.g. as the multiplier of a force or moment, or a prescribed displacement or rotation). For example, in the case of a force:

```
force : (unsigned) <force_label> , absolute ,
    (unsigned) <node_label> ,
        position, null ,
    1., 0., 0., # absolute x direction
    element , (unsigned) <hfelem_label> , loadable ,
        string , "excitation" , linear , 0., 100. ;
```

In the above example, the force is applied to node `node_label` along the (absolute) x direction, and scaled by a factor 100.

- `psi`, i.e. the argument of the sine function that corresponds to the `excitation`:

$$\texttt{excitation} ::= \sin(\texttt{psi}) \tag{B.1}$$

Its definition is thus

$$\texttt{psi} ::= \texttt{omega} \cdot (t - t_0) \tag{B.2}$$

It may be useful to have multiple inputs with different phases; for example, in the case of a force whose components have different phases:

```
force : (unsigned) <force_label> , absolute ,
    (unsigned) <node_label> ,
        position, null ,
    component,
        # absolute x direction
        element , (unsigned) <hfelem_label> , loadable ,
            string , "psi" , string , "100.*sin(Var)",
        # absolute y direction
        element , (unsigned) <hfelem_label> , loadable ,
            string , "psi" , string , "50.*sin(Var - pi/2)",
        # absolute z direction
        const, 0.;
```

In the above example, the force is applied to node `node_label` along the (absolute) x direction, scaled by a factor 100, and along the (absolute) y direction, scaled by a factor 50, with 90 deg phase delay.

- `omega`, i.e. the angular frequency of the excitation signal. This may be useful, for example, to modify the amplitude of the signal as a function of the frequency.
- `amplitude`, the amplitude of the excitation signal (equal to 1, unless `RMS target` is set).
- `count`, the number of frequencies that have been applied so far.
- `output`, boolean variable equal to 1 when the output must be written to the `.usr` file or for `write_after_convergence_periods` periods after convergence every `write_frequency_after_convergence` timesteps. This private data should be called by an `output meter`, and it is conceived to avoid the need very large output files (other than the `.usr` file) when only the history of the variables in the converged periods matters.

  To this end, the analysis (typically, an `initial value` problem) should be set up with a `control data` section containing

```
output meter : postponed , (unsigned) <outputmeter_drive_label> ;
```

Then, after instantiating the hfelem, one must define the drive caller outputmeter_drive_label with the following statement or any other statement calling the hfelem private data output.

```
drive caller : (unsigned) <outputmeter_drive_label> ,
    element , (unsigned) <hfelem_label> , loadable ,
    string , "output" , direct ;
```

Notes.

- As illustrated in the **Private Data** timestep section, the time step must be controlled by this element.
- Only one instance of this element should be used in the analysis; currently, a warning is issued if it is instantiated more than once, and the two instances operate together, with undefined results (convergence is likely never reached, because the solution does not tend to become periodic with any of the expected periods).
- This element might terminate the simulation in the following cases:
  a) the excitation angular frequency becomes greater than omega_max;
  b) the angular frequency increment pattern is custom, and convergence is reached for the last value of excitation angular frequency.
  c) the no convergence strategy option for the max periods optional parameters is abort, and no steady value is achieved for the fundamental harmonic of the response within the specified maximum number of periods.

Example.

```
begin: initial value;
    # ...
    final time: forever;

    # the time step will be passed from outside
    set: const integer TIMESTEP_DRIVE = 99;
    strategy: change, postponed, TIMESTEP_DRIVE;

    # ...
end: initial value;

# ...

begin: elements;
    # ...

    set: const integer HARMONIC_EXCITATION = 97;
    user defined: HARMONIC_EXCITATION, harmonic excitation,
        inputs number, 2,
            node, 2, structural, string, "X[1]", direct,
            node, 2, structural, string, "XP[1]", direct,
        steps number, 128,
        initial angular frequency, 2.*2*pi,
            max angular frequency, 25.*2*pi, # otherwise forever (or until max time)
```

```
            angular frequency increment,
                # additive, 0.25*2*pi,
                multiplicative, 1.06,
                # custom, 4.*2*pi, 8.*2*pi, 15.*2*pi, 20.*2*pi, 30.*2*pi, last,
        # initial time, 5., # defaults to "always"
        tolerance, 1.e-8,
        min periods, 2;

    # make sure output occurs...
    output: loadable, HARMONIC_EXCITATION;

    # pass time step to solver (alternatively, use the "timestep drive label" option)
    drive caller: TIMESTEP_DRIVE, element, HARMONIC_EXCITATION, loadable,
        string, "timestep", direct;

    # ...
end: elements;
```

TODOs?

- Make it possible to continue the analysis instead of stopping when `omega_max` is reached?

B.1.8 Module-hydrodynamic_plain_bearing_with_offset

Author: Reinhard Resch

This module implements a hydrodynamic plain bearing according to Hans Jürgen Butenschön 1976 Das hydrodynamische zylindrische Gleitlager endlicher Breite unter instationärer Belastung.

```
    <name> ::= hydrodynamic_plain_bearing_with_offset

    <arglist> ::=
        shaft, (<label>) <shaft_node> ,
        [ offset , (Vec3) <o1> , ]
        bearing , (<label>) <bearing_node> ,
        [ offset , (Vec3) <o2> , ]
        bearing width , (real) <b> ,
        { , shaft diameter , (real) <d> | bearing_diameter , (real) <D> } ,
        relative clearance , (real) <Psi> ,
        oil viscosity , (real) <eta> ,
        initial assembly factor , (DriveCaller) <assembly_factor>
        [ , number of Gauss points , (integer) <num_Gauss_points> ]
        [ , output points , (integer) <num_output_points>
            [ , { Gauss point , (integer) <Gauss_index_1 > |
                custom , r , (real) <r_1> ,
                alpha , (real) <alpha_1> }
            ]
        ]
        [ , extend shaft to bearing center , { yes | no | (bool) <extend_axis_of_shaft> } ]
        [ , epsilon max , (real) <epsilon_max> ]
```

We assume that the axis of rotation is always axis three in the reference frame of the node. Only small misalignments of bearing and shaft are allowed.

$$\cos\varphi = \boldsymbol{e}_3{}^T \boldsymbol{R}_1{}^T \boldsymbol{R}_2 \boldsymbol{e}_3 \approx 1 \tag{B.3}$$

The effective angular velocity of the bearing rigidly connected to node <bearing_node> is $\bar{\omega}_2 = \boldsymbol{e}_3{}^T \boldsymbol{R}_2{}^T \boldsymbol{\omega}_2$ and the effective angular velocity of the shaft connected to node <shaft_node> is $\bar{\omega}_1 = \boldsymbol{e}_3{}^T \boldsymbol{R}_2{}^T \boldsymbol{\omega}_1$. <o1> and <o2> are the offsets between nodes and centre of the shaft and bearing respectively. The width of the bearing is <b> and the relative clearance of the bearing is defined as <Psi> = (<D> − <d>)/<D> where <D> is the inner diameter of bearing and <d> is the outer diameter of the shaft. <eta> is the dynamic viscosity of the oil and its SI unit is [Pa s]. It is possible to disable the hydrodynamic bearing force at the beginning of the simulation by providing some sort of ramp up function called <assembly_factor>. In order to take into account small misalignments between shaft and bearing, one can set the number of Gauss points <num_Gauss_points> to a value of two, three or six. In that case hydrodynamic bearing forces will be evaluated at multiple points along the width of the bearing and output will be generated for each Gauss point by default. Output can be customised by selecting only a subset of available Gauss points by means of <Gauss_index_1> or by specifying custom locations <r_1> and weighting factors <alpha_1> to be output. If the option <extend_axis_of_shaft> is enabled, axial movement of the shaft is allowed and bearing kinematics and reaction forces will be evaluated at the intersection point between the axis of the shaft and centre plane of the bearing. According to the theory of a rigid bearing, eccentricities of $\varepsilon \geq 1$ are not possible and the simulation would fail if ε becomes one or higher. In addition to that, the approximation of the Sommerfeld numbers is valid until a relative eccentricity of $\varepsilon = 0.999$. For that reason a linear extrapolation of the Sommferfeld numbers is used in case of $\varepsilon \geq$ <epsilon_max> which is equal to 0.999 by default.

B.1.9 Module-hydrodynamic plain bearing2

Author: Reinhard Resch

This module implements a hydrodynamic lubricated plain bearing based on a numerical solution of the so called Reynolds equation[43]. The following features are implemented:

- Finite Difference or Finite Element discretization
- mass conserving and non mass conserving cavitation models (e.g. Gümbel, Jakobsson-Floberg-Ollson).
- asperity contact and dry friction considering stick-slip effects (e.g. Coulomb, LuGre)
- cylindrical bearings with small imperfections (e.g. barrel shape, cone shape, ellipse shape, grooves ...)
- arbitrary lubrication grooves and boundary conditions (e.g. pressure, density or filling ratio)
- optional coupling with hydraulic networks
- deformable shaft and bearing surfaces (e.g. two way fluid structure interaction)

```
<name> ::= hydrodynamic plain bearing2

<arglist> ::=
        hydraulic fluid,
            (HydraulicFluid) <hydraulic_fluid>,
        mesh, { linear finite difference | quadratic finite element },
```

```
        enable mcp, (bool) <mixed_complementarity_problem>
    geometry, cylindrical,
        mesh position, { at shaft | at bearing },
        bearing width, (real) <bearing_width>,
        shaft diameter, (real) <diameter_shaft>,
        bearing diameter, (real) <diameter_bearing>,
        [ pockets, <pockets_definition>, ]
        shaft node,
            (StructNode) <shaft_node_id>,
            offset, (Vec3) <offset_shaft_center>
            orientation, (OrientationMatrix) <orientation_shaft_node>,
        bearing node,
            (StructNode) <bearing_node_id>,
            offset, (Vec3) <offset_bearing_center>
            orientation, (OrientationMatrix) <orientation_bearing_node>,
    grid spacing,
        <grid_data>
{   boundary conditions,
        <boundary_condition_1>,
        <boundary_condition_2>, |
        pressure coupling conditions axial,
        (HydraulicNode) <hydraulic_node_label_outlet1>,
        (HydraulicNode) <hydraulic_node_label_outlet2>,
}
[ lubrication grooves, <lubrication_groove_data>, ]
[ pressure coupling condition radial, <pressure_coupling_condition_radial>, ]
[ contact model, <contact_model>, ]
[ friction model, <friction_model>, ]
[ compliance model, <compliance_model>, ]
  [ , pressure dof scale, (real) <pressure_dof_scale> ]
  [ , equation scale, (real) <equation_scale> ]
  [ , output pressure, { yes | no } ]
  [ , output contact pressure, { yes | no } ]
  [ , output density, { yes | no } ]
  [ , output friction loss, { yes | no } ]
  [ , output clearance, { yes | no } ]
  [ , output clearance derivative, { yes | no } ]
  [ , output velocity, { yes | no } ]
  [ , output stress, { yes | no } ]
  [ , output reaction force, { yes | no } ]
  [ , output total deformation, { yes | no } ]
  [ , output deformation bearing, { yes | no } ]
  [ , output deformation shaft, { yes | no } ]
  [ , output mesh, yes, { yes | no } ]
```

Grid

```
<grid_data> :: = { <grid_data_uniform> | <grid_data_intervals> }
```

```
<grid_data_uniform> :: = uniform, { <grid_data_uniform_nodes> | <grid_data_uniform_elements> }

<grid_data_uniform_nodes> :: =
    number of nodes z, (real) <number_of_nodes_z>,
    number of nodes Phi, (real) <number_of_nodes_x>

<grid_data_uniform_elements> :: =
    number of elements z, (real) <number_of_elements_z>,
    number of elements Phi, (real) <number_of_elements_x>

<grid_data_intervals> :: =
    uniform intervals,
    number of intervals z, (integer) <number_intervals_axial>,
    <interval_data>,
    number of intervals x, (integer) <number_intervals_circumference>,
    <interval_data>,


<interval_data> :: =
   (real) <position>, (integer) <number_of_nodes_1>,
   (real) <position>, (integer) <number_of_nodes_2>,
   ... ,
   (real) <position>, (integer) <number_of_nodes_N>,
```

Pockets

```
<pockets_definition> :: =
    <pocket_definition_1>,
    ...
    <pocket_definition_N>

<pocket_definition> :: =
 { shaft | bearing },
      (integer) <number_of_pockets>,
        <pocket_shape_1>,
        ...
        <pocket_shape_N>,

<pocket_shape>
    position,
      (real) <pocket_position_x_1>,
      (real) <pocket_position_z_1>,
      <pocket_area_2D>,
      pocket height, { const | surface grid },
         <pocket_height_definition>

<pocket_area_2D> ::=
 { <pocket_area_circular> | <pocket_area_rectangular> | complete surface }
```

```
<pocket_area_circular> :: =
    circle,
    radius, (real) <pocket_radius>,

<pocket_area_rectangular> :: =
     rectangle,
     width, (real) <pocket_width>,
     height, (real) <pocket_height>

<pocket_height_definition> :: =
  { <pocket_height_definition_const> | <pocket_height_definition_grid> }

<pocket_height_definition_const> = const, (real) <pocket_height>

<pocket_height_definition_grid> = surface grid,
    x, (integer) <number_grid_points_x>, (real) <x_1>, ... , (real) <x_N>
    z, (integer) <number_grid_points_z>, (real) <z_1>, ... , (real) <z_N>
    delta y, (real) <y_11>, (real) <y_12>, ... , (real) <y_NM>
```

Boundary conditions

```
<boundary_condition> :: =
 { pressure | density | filling ratio },
    (DriveCaller) <boundary_condition_value>

<lubrication_groove_data> :: =
    (integer) <number_of_grooves>,
    <lubrication_groove_definition_1>,
     ...
    <lubrication_groove_definition_N>,

<lubrication_groove_definition> :: =
 { at shaft | at bearing },
    <boundary_condition>,
    position, (real) <position_x>, (real) <position_z>,
    <lubrication_groove_dimensions>

<lubrication_groove_dimensions> :: =
 { <lubrication_groove_dim_circular> | <lubrication_groove_dim_rectangular> }

<lubrication_groove_dim_circular> :: =
    circle,
    radius, (real) <lubrication_groove_radius>

<lubrication_groove_dim_rectangular> :: =
    rectangle,
    width, (real) <lubrication_groove_width>,
    height, (real) <lubrication_groove_height>
```

Coupling with hydraulic networks

```
pressure coupling conditions radial, (integer) <number_of_inlets>,
    <pressure_coupling_condition_1>,
    ...,
    <pressure_coupling_condition_N>

<pressure_coupling_condition> :: =
    <lubrication_groove_dimensions>,
    pressure node, (HydraulicNode) <hydraulic_node_label_inlet>,
```

Contact model

```
<contact_model> :: =
    greenwood tripp,
    E1, (real) <young_modulus_1>,
    nu1, (real) <poisson_ratio_1>,
    E2, (real) <young_modulus_2>,
    nu2, (real) <poisson_ratio_2>,
    M0, (real) <combined_spectral_moment_M0>,
    M2, (real) <combined_spectral_moment_M2>,
    M4, (real) <combined_spectral_moment_M4>,
```

Friction model - dry friction

```
<friction_model> :: =
 { coulomb | lugre },
    <friction_model_data>

<friction_model_data> :: =
 { <friction_model_data_coulomb> | <friction_model_data_lugre> }

<friction_model_data_coulomb> :: =
    mu, (real) <coulomb_friction_coefficient>
[ , sliding velocity threshold, (real) <sliding_velocity_threshold> ]

<friction_model_data_lugre> :: =
    method, { implicit euler | trapezoidal rule },
    coulomb friction coefficient, (real) <coulomb_friction_coefficient>,
  [ static friction coefficient, (real) <static_friction_coefficient>, ]
  [ sliding velocity coefficient, (real) <sliding_velocity_coefficient>, ]
    micro slip stiffness, (real) <micro_slip_stiffness_sigma0>,
  [ micro slip damping, (real) <micro_slip_damping_sigma1> ]
```

Compliance model

```
<compliance_model> :: =
 { <compliance_mod_nodal> | <compliance_mod_double_nodal> }

<compliance_mod_nodal> :: =
    matrix, 1, <compliance_matrix_data>,
```

```
    matrix, 2, <compliance_matrix_data>

<compliance_mod_double_nodal> :: =
    double nodal,
    matrix at shaft, <compliance_matrix_data>,
    matrix at bearing, <compliance_matrix_data>,
    axial displacement, { small | large }

<compliance_matrix_data> :: =
 { <compliance_matrix_half_space> | <compliance_matrix_from_file> }

<compliance_matrix_half_space> :: =
    elastic half space,
[ , E1, (real) <E1> ]
[ , nu1, (real) <nu1> ]

<compliance_matrix_from_file> :: =
    from file, " <compliance_matrix_file> "
[ , E1, (real) <E1> ]
[ , nu1, (real) <nu1> ]
[ , modal element, (ModalJoint) <elem_id_modal_joint> ]
```

Mass conserving versus non-mass conserving cavitation

In order to use a mass conserving cavitation model, a `linear compressible` hydraulic fluid must be selected. In that case the fluid density at the surface of the bearing may vary between zero and the liquid density, which is equal to the reference density of the hydraulic fluid, during the simulation. Otherwise, an `incompressible` hydraulic fluid should be used, and the density of the fluid will remain constant during the simulation.

Finite Difference versus Finite Element discretization

Mass conserving cavitation models are not yet supported by the Finite Element discretization.

MCP based solution

This option may be used if, and only if a mass conserving cavitation model is used and a proper nonlinear solver has been selected. See also section 4.1.1.

Bearing geometry

Currently only bearings with cylindrical surfaces are supported. The main dimensions are `<diameter_shaft>`, `<diameter_bearing>` and `<bearing_widht>` and the so called relative clearance Ψ is:

$$\Psi = \frac{\texttt{<diameter\_bearing>} - \texttt{<diameter\_shaft>}}{\texttt{<diameter\_shaft>}} \tag{B.4}$$

Small deviations from a perfectly cylindrical bearing (e.g. barrel shape, cone shape) may be specified by means of `<pockets>`. See also the technical manual for details.

Structural nodes

It is possible to specify an arbitrary offset and orientation between the center of the bearing and the related structural node. Actually, the bearing and shaft axes are aligned to the z-axis of of the orientation matrices <orientation_shaft_node> and <orientation_bearing_node>.

Computational grid

Regarding the mesh- or grid-size, it is possible to define either a fixed number of nodes or a fixed number of elements in axial or "z"-direction and in circumferential or "Phi"-direction. Also a non-constant grid size is supported.

Boundary conditions, lubrication grooves and coupling conditions

Boundary conditions at the edges of the bearing may be specified by means of drive callers or hydraulic nodes.

Contact model

The contact model is based on Greenwood and Tripp[44]. Input parameters are Young's modulus and Poisson's ratio of both surfaces and also the combined spectral moments of the surface roughness profiles. See the technical manual for further details.

Friction model

In case of solid contact inside the bearing, dry friction may be considered by means of a Coulomb or 2D-LuGre friction model[45]. Actually, the transition between sticking and sliding can be considered only by means of the LuGre model. See also the technical manual for further details.

Compliance model

If the deformation of the shaft and/or bearing surfaces should be considered (e.g. two way fluid structure interaction), a compliance model must be used. The following options are supported:

- Nodal compliance model: Only the bearing surface at the body attached to the mesh is considered as flexible. If a second matrix is provided, the values are just summed up. However, this is valid only if the component which is not attached to the mesh is axisymmetric. So, the actual relative orientation between shaft and bearing surface does not affect the flexibility.
- Double nodal compliance model: Both surfaces (e.g. shaft and bearing) are considered as flexible without any restrictions. In this case the pressure distribution and deformation will be interpolated between the shaft and bearing surface as required.

Elastic half space Based on the assumption of a semi-infinite body, the compliance matrix can be computed automatically without any additional data.

Compliance matrix from file In most situations, all the compliance matrices should be computed by `mboct-fem-pkg` and stored in a file <compliance_matrix_file>. The related function is called "fem_ehd_pre_comp_mat_unstruct" and the following options are available:

- Quasi static nodal compliance model (matrix type "nodal")

- Fully dynamic compliance model coupled to the mode shapes of a modal joint (matrix type "modal substruct total"): In this case the related modal joint (e.g. <elem_id_modal_joint>) must be provided in the input file. In addition to that, the related structural node (e.g. <shaft_node_id> or <bearing_node_id>) should be attached to the "modal node" of the same modal joint. See also 8.12.31.

Output

Actually, the data structure of all the output is quite complex. So, it is recommended to use `mboct-mbdyn-pkg` in order to load and display the output. See also the documentation of "mbdyn_post_ehd_load_output".

B.1.10 Module-imu

Author: Pierangelo Masarati
This module implements an element that provides the motion of a structural node in the form of the output of an Inertial Measurement Unit (3 components of acceleration, 3 components of angular velocity in body axes) and an element that prescribes the motion of a structural node in terms of acceleration and angular velocity.

IMU

This user-defined element emulates a IMU.

```
<name> ::= imu

<module_data> ::=
    <node_label>
        [ , position , (Vec3) <offset> ]
        [ , orientation , OrientationMatrix <orientation> ]
```

It makes the acceleration and the angular velocity of node `node_label` available in the node's reference frame as private data. The location and orientation of the IMU with respect to the node may be modified by `offset` and `orientation`.

Output. This element sends output to the `.usr` file. Each entry contains

1) the label

2–4) three components of angular velocity, in the reference frame of the node

5–7) three components of acceleration, in the reference frame of the node

Private Data. The following data are available:

1. `"wx"` angular velocity in local direction 1
2. `"wy"` angular velocity in local direction 2
3. `"wz"` angular velocity in local direction 3
4. `"ax"` acceleration in local direction 1
5. `"ay"` acceleration in local direction 2
6. `"az"` acceleration in local direction 3

IMU constraint

This user-defined element enforces IMU data as a constraint.

```
<name> ::= imu constraint

<module_data> ::=
    <node_label>
        [ , position , (Vec3) <offset> ]
        [ , orientation , (OrientationMatrix) <orientation> ]
        (TplDriveCaller<Vec3>) <omega> ,
        (TplDriveCaller<Vec3>) <acceleration>
```

It imposes the angular velocity `omega` and the acceleration `acceleration`, namely the measurements that come from an IMU, to node `node_label`. The location and orientation of the IMU with respect to the node may be modified by `offset` and `orientation`.

Output. This element sends output to the `.usr` file. Each entry contains

1) the label

2–4) three angular velocity Lagrange multipliers, in the reference frame of the node

5–7) three acceleration Lagrange multipliers, in the reference frame of the node

8–10) three components of velocity, in the reference frame of the node

11–13) three components of velocity derivative, in the reference frame of the node

B.1.11 Module-mds

This module is a simple example of run-time loadable user-defined element, implements a scalar mass-damper-spring system.

B.1.12 Module-MFtire

```
<name> ::= MFtire

<arglist> ::=
    (integer) <hub_node_label>, # wheel structural node label
    (integer) <rim_node_label>, # rim structural node label
    (Vec3) <Vec3_axle> , # wheel axle direction (e.g. 0., 1., 0.)
    (real) <Ru>, # unloaded radius
    (real) <Re>, # effective rolling radius
    (real) <kt>, # tire vertical stiffness
    # longitudinal pacejka coefficients
    (real) <Dx>,
    (real) <Cx>,
    (real) <Bx>,
    (real) <Ex>,
    # lateral pacejka coefficients
    (real) <Dy>,
    (real) <Cy>,
```

```
    (real) <By>,
    (real) <Ey>
    # reference vx, damping coeff
    [low speed, (real) <vx_low>, (real) <vx_dmp>];
```

B.1.13 Module-nonsmooth-node

Author: Matteo Fancello

```
<name> ::= nonsmooth node

<module_data> ::=
    (StructDispNode) <NonsmoothNODELABEL> ,
    mass , (real) <mass> ,
    radius , (real) <radius> ,
    planes , (integer) <number_of_planes> ,
        <PlaneDefiningNODELABEL> ,
            position , (Vec3) <relative_plane_position> ,
            rotation orientation , (OrientationMatrix) <rel_rot_orientation_1> ,
            restitution , (real) <rest_coef>
            [ , friction coefficient , (real) <mu> ]
        [ , ... ] # as many blocks as number_of_planes
    [ , constraint type , { position | velocity | both } ] # default: both
    [ , theta , <theta> ]
    [ , gamma , <gamma> ]
    [ , LCP solver , <solver> ]
    [ , tolerance , <tolerance> ]
    [ , max iterations , <num_iter> ]
        # these options depend on LCP solver support, see
        # http://siconos.gforge.inria.fr/Numerics/LCPSolvers.html
    [ , limit iterations , <niterations> ]
    [ , limit LCP iterations , <niterations> ]
    [ , verbose , { yes | no | (bool) <verbose> } ]

<solver> ::=
    lexico lemke # the default
    | rpgs
    | qp
    | cpg
    | pgs
    | psor
    | nsqp
    | latin
    | latin_w
    | Newton_min
    | Newton_FB
```

Output

- 1: element label

- 2–4: impulse on nonsmooth node in global ref. frame
- 5–7: position of nonsmooth node in global ref. frame
- 8–10: velocity of nonsmooth node in global ref. frame
- 11–13: constraint reaction between multibody node and nonsmooth node
- 14: norm of the impulse reaction normal to the contact plane
- 15: number of active constraints during step

if verbose, also:

- 16–18: position constraint relaxation factor (only when `constraint type` is `both`)
- 19: LCP solver status: 0 indicates convergence, 1 indicates `iter` $\equiv$ `maxiter`, > 1 indicates failure (only for some solvers)
- 20: LCP solver `resulting_error` (only meaningful for some solvers)

B.1.14 Module-template2

Template of user-defined element.

B.1.15 Module-wheel2

Authors: Marco Morandini, Stefania Gualdi and Pierangelo Masarati
This module implements a simple tire model for aircraft landing and ground handling.

B.1.16 Module-wheel4

Authors: Louis Gagnon, Marco Morandini, and Pierangelo Masarati
This module implements a rigid ring tire model similar to the models commonly known as SWIFT. It is used as an element that will apply a 3D force and a 3D moment to the ring node of a wheel-ring multibody system. A wheel node is also necessary for inertia and contact forces calculations. It is intended to evaluate the transient behavior of the tire rolling on a deteriorated road profile. The equations are integrated implicitly except for the road profile, which is an input. This profile has to be previously filtered by a super ellipse function because the `wheel4` module will only apply the tandem-cam part of the profile filtering. It is tailored for, but not restricted to, applications at low camber angles, limited steering and velocity changes, and continuous contact with the road. It is expected to be accurate under excitation frequencies up to 100 Hz and road deformations up to 20% of the tire radius. A variable time-step algorithm is also embed and speeds up the simulation in cases where the road is flat.

Two runnable examples are available and will clarify the implementation in an actual multibody simulation:

- the simple example *axleExampleNoData* located within the `module-wheel4` directory of the MBDyn package
- the more complex *semitrailer* model is available on the example page of the MBDyn website; this example provides the script necessary to apply the super ellipse filter on the road profile; https://www.mbdyn.org/Documentation/Examples.html

```
<name> ::= wheel4

<module_data> ::=
    (StructDispNode) <WheelStructNodeLabel> , (StructBody) <WheelBodyLabel> ,
    (OrientationVector) <wheel_axle_direction> , (real) <tire_radius> ,
    swift ,
    (StructDispNode) <RingStructNodeLabel> , (StructBody) <RingBodyLabel> ,
    (vector) <patch_stiffness> , (drive) <stiffness_modifier> ,
    (vector) <patch_damping> , (drive) <damping_modifier> ,
    (vector) <initial_patch_velocity> , (real) <patch_mass> ,
    (DriveCaller) <road_profile_driver> ,
    (real) <patch_to_ellip_cam_ratio> ,
    (real) <r_a1_param> , (real) <r_a2_param> ,
    (real) <patchToTireCircumf_ratio> ,
    (real) <vert_wheel_ring_stiff>
    [ , loadedRadius ]
    [ , slip ,
    ginac , <longi_tire_force_funct> ,
    ginac , <lateral_tire_force_funct> ,
    ginac , <pneumatic_trail_funct> ,
    ginac , <aligning_residual_moment> ,
    (real) <S_ht> , (real) <S_hf> ,
    (real) <q_sy1> , (real) <q_sy3> ,
    (real) <dvao>
    [ , threshold, (real) <TRH> , (real) <TRHA> , (real) <TRHT> ,
    (real) <TRHTA> , (real) <TRHC> , (real) <TdLs> , (real) <TdReDiv> ,
    (real) <TdRe> , (real) <dtOn> , (real) <TmaxH> ,
    (real) <dtRes> , (real) <maxstep> , (real) <minstep> ,
    (real) <TmaxF> , (real) <TminF> , (integer) <TminS> , (real) <TdivF> ,
    (real) <TdivF3> , (real) <TdivF4> , (real) <RDA> ,
    (real) <RDB> , (real) <RDL> ] ]
```

where the keyword loadedRadius enables the use of the alternative, validated, loading radius instead of the more generally used definition. The keywords slip and threshold should always be present. Although the given examples are the best way to understand the model, the following tables clarify most input parameters,

	tire parameters
<wheel_axle_direction>	3D vector of the wheel axle direction in the absolute reference frame (care should be taken if set to a value other than *0.,1.,0.* because no elaborate testing has been carried for alternate axle initial orientations; any comments or on its functionality are welcome)
<tire_radius>	rigid ring radius (undeformed radius of the tire)
<patch_stiffness>	3D stiffness vector of the contact patch to ring connection (in the ring reference frame)
<stiffness_modifier>	modifier drive to allow, for example, a gradual application of the stiffness
<patch_damping>	3D damping vector of the contact patch to ring connection (usually about 6% to 10% of critical damping and distributed over ring and patch connections)
<damping_modifier>	modifier drive to allow, for example, a gradual application of the damping
<initial_patch_velocity>	3D vector for the initial velocity of the patch in absolute reference frame
<patch_to_ellip_cam_ratio>	patch contact-length to elliptical cam tandem base parameter (Schmeitz eq. 4.15, $l_s/(2a)$)
<r_a1_param>	r_{a1} contact length parameter from Besselink eq. 4.85
<r_a2_param>	r_{a2} contact length parameter from Besselink eq. 4.85
<patchToTireCircumf_ratio>	ratio of the contact patch length to tire circumference (to calculate how much mass contributes to the ring's centripetal forces)
<vert_wheel_ring_stiff>	vertical stiffness given to the viscoelastic connection between the ring and the wheel nodes
<longi_tire_force_func>	longitudinal force F_x/F_z given in GiNaC format
<lateral_tire_force_func>	lateral force F_y/F_z given in GiNaC format
<pneumatic_trail_func>	pneumatic trail divided by vertical force and given in GiNaC format
<aligning_residual_moment>	residual torque M_{zr}/F_z given in GiNaC format
<S_ht>	horizontal shift of pneumatic trail, for aligning moment angle modifier
<S_hf>	residual aligning moment angle modifier/shift
<q_sy1>	tire rolling resistance linear velocity coefficient (usually between 0.01 and 0.02)
<q_sy3>	rolling resistance velocity correction coefficient
<dvao>	reference velocity for rolling resistance velocity influence factor
	variable time step algorithm parameter
<dtOn>	boolean to enable or disable adjustable time step calculation (will greatly increase the rapidity of the solution only if you have few bumps on a very smooth road and will otherwise slow down the simulation)
<TmaxH>	maximum height change wanted on the road profile for one step
<dtRes>	resolution of bump search
<maxstep>	maximum time step imposed in the initial value section
<minstep>	minimum time step imposed in the initial value section
<TdivF3>	time step adjustment factor if force switched sign 3 times in the last TminS steps
<TdivF4>	time step adjustment factor if force switched sign 4 or more times in the last TminS steps
	road offset parameters
<RDA>	road offset (null before position reaches that value)
<RDB>	road offset (interpolated when position is between <RDA> and that value)
<RDL>	road loop condition (will loop after <RDB>+<RDL> over <RDL>)

	algorithm threshold parameters
<TRH>	prevents division by zero at null x-velocity at the price of losing validity for velocities near <TRH>
<TRHA>	buffer used to prevent division by zero
<TRHT>	prevents division by zero when computing the angle of the vehicle or wheels
<TRHTA>	buffer used on angle zero division prevention
<TRHC>	maximum value allowed for the longitudinal slip ratio
<TdLs>	minimum value that the half contact patch length may take
<TdReDiv>	minimum value that the wheel angular velocity may take in the calculation of the effective rolling radius
<TdRe>	maximum ratio $\left\|\frac{effectiverollingradius}{ringradius}\right\|$ which will be allowed to reach

Information about the informally cited works may be found in the following two theses,

- Schmeitz, A. J. C. (2004) *A Semi-Empirical Three-Dimensional Model of the Pneumatic Tyre Rolling over Arbitrarily Uneven Road Surfaces* available on request
- Besselink, I.J.M. (2000). *Shimmy of Aircraft Main Landing Gears* available at http://www.tue.nl/en/publication/ep/p/d/ep-uid/227775/

Output

The output can be obtained either in plain text in the **.usr** file or in NetCDF format in the **.nc** file. Section C.1.3 explains the NetCDF output whereas the plain text output is as follows,

- 1: element label
- 2: velocity of wheel in x-dir. (longitudinal, forward)
- 3: velocity of wheel in y-dir. (lateral)
- 4: relative speed between center of wheel and contact point on tire in the forward direction
- 5–7: moment applied on ring by this module
- 8–10: moment arm on ring
- 11: slip ratio
- 12: slip angle
- 13: longitudinal friction coefficient
- 14: lateral friction coefficient
- 15: road height
- 16–18: road normal
- 19–21: position of patch
- 22–24: velocity of patch
- 25–27: relative position of patch
- 28–30: relative velocity of patch

- 31–33: force between ring and patch acting on patch
- 34–36: force between ring and patch acting on ring and thus applied on ring by this module
- 37–39: forward direction vector of the wheel
- 40–42: forward direction vector of the ring
- 43–45: forward direction vector of the ring without the slope of the profile
- 46–48: point of contact on ring between ring and springs (contact patch viscoelastic elements)
- 49: normal force for Pacejka's formulas
- 50–52: relative velocity between patch and wheel
- 53: sum of wheel, ring, and patch kinetic energies
- 54: sum of wheel and ring potential energies (patch is not included because it is not subjected to gravity)
- 55: sum of wheel, ring, and patch total energies (kinetic + potential)
- 56: virtually calculated effective rolling radius
- 57: half length of the tandem elliptical cam follower
- 58: modified loaded radius (distance between ring center and patch center)
- 59–61: distance between ring contact point and patch as seen from the ring in its own reference frame (not rotated with road slope)
- 62: centrifugal force added to tire
- 63–65: rolling resistance force vector (this force is then applied as a moment only)
- 66–68: aligning moment
- 69: center point x-value of the road profile (this is not actual position, but only position on the input road file
- 70: front edge x-point of the tandem (front contact point of patch)
- 71: rear edge x-point of the tandem (rear contact point of patch)
- 72: centrifugally induced virtual displacement of tire in the radial direction (units of distance)
- 73: patch vertical velocity calculated using road displacement and time step
- 74: time step

B.1.17 Module-MFtire

Author: Andrea Fontana
This module implements a magic formula tire model.
Note:

- The Wheel Hub and the Wheel Rim structural nodes must be connected by a joint that allows relative rotations only about one axis
- The center of the wheel is assumed coincident with the position of the Wheel Hub and Rim structural nodes
- The model assumes a flat infinite plane as road
- Forces, moments and slips are defined according to ISO convention.

```
<name> ::= MFtire

<module_data> ::=
    (StructDispNode) <wheel structural node label> ,
    (vector) <wheel_axle_direction> ,
    (real) <unloaded_wheel_radius> ,
    (real) <effective_rolling_radius> ,
    (real) <vertical_stiffness> ,
    (real) <Dx>, (real) <Cx>, (real) <Bx>, (real) <Ex>,
    (real) <Dy>, (real) <Cy>, (real) <By>, (real) <Ey>
    [ , low speed , (real) <low speed threshold>, (real) <low speed damping> ]
```

where all the magic formula coefficients but <Dy> must be > 0; <Dy> should be < 0.

Output

The output can be obtained in plain text only, in the **.usr** output file.

- 1: element label
- 2-4: tire force in global reference frame
- 5-7: tire couple in global reference frame
- 8: slip ratio
- 9: slip angle
- 10: loaded radius

B.2 Constitutive Law Modules

B.2.1 Module-constlaw

Simple example of run-time loadable user-defined constitutive law.

B.2.2 Module-constlaw-f90

Simple example of run-time loadable user-defined constitutive law in Fortran 90.

B.2.3 Module-constlaw-f95

Simple example of run-time loadable user-defined constitutive law in Fortran 95.

B.2.4 Module-cont-contact

Author: Matteo Fancello
Implements various formulas of 1D continuous contact models. The same constitutive law can also be used as a 3D constitutive law, under the assumption that the 1D formulas are considered with respect to direction 3 of the 3D entity it is associated with. In such case, the component 3 of strain and strain rate are used as inputs, and the output force is applied along direction 3.

The syntax is

```
<drive_caller> ::= help ;
    | [ help , ]
        [ sign , { positive | negative | <sign_value> } , ]
        [ formulation , { flores | hunt crossley | lankarani nikravesh } , ]
        restitution , <restitution_coef> ,
        kappa , <stiffness> ,
        exp , <exponent>
        [ , EpsPrimeTol , <initial_eps_prime_tol> ]
```

- `<sign_value>` is a non-null real number, whose sign is used for the `sign` parameter.
- `<restitution_coef>` is the restitution coefficient ($[0 \to 1]$).
- `<stiffness>` is the stiffness coefficient (> 0).
- `<exp>` is the exponent (> 1).
- `<initial_eps_prime_tol>` is the tolerance on the initial contact velocity (> 0).

Three models are implemented. They all use the strain and the strain rate, either as are, when `sign` is `positive`, or their opposite, when `sign` is `negative`,

$$x = \texttt{sign\_value} \cdot \texttt{Eps} \tag{B.5a}$$
$$\dot{x} = \texttt{sign\_value} \cdot \texttt{EpsPrime} \tag{B.5b}$$

The expression of the force is

$$f = 0 \qquad x \leq 0 \tag{B.6a}$$
$$f = \texttt{sign\_value} \cdot x^{\texttt{exp}} \left(\texttt{stiffness} + \texttt{dissipation\_coef} \cdot \dot{x}\right) \qquad x > 0 \tag{B.6b}$$

with the dissipation coefficient `dissipation_coef` defined according to each of the supported models.

Their `formulation` is:

- `flores` [46]: this is the default model; the dissipation coefficient is

$$\texttt{dissipation\_coef} = \frac{8}{5} \cdot \texttt{stiffness} \cdot \frac{1 - \texttt{restitution\_coef}}{\texttt{restitution\_coef} \cdot \dot{x}_0} \tag{B.7}$$

  Note that in this case $0 < \texttt{restitution\_coef} \leq 1$.

- `hunt crossley` [47]: the dissipation coefficient is

$$\texttt{dissipation\_coef} = \frac{3}{2} \cdot \texttt{stiffness} \cdot \frac{1 - \texttt{restitution\_coef}}{\dot{x}_0} \tag{B.8}$$

- `lankarani nikravesh` [48]: the dissipation coefficient is

$$\texttt{dissipation\_coef} = \frac{3}{4} \cdot \texttt{stiffness} \cdot \frac{1 - \texttt{restitution\_coef}^2}{\dot{x}_0} \tag{B.9}$$

Note that, to avoid division by zero, $\texttt{dissipation\_coef} = 0$ when $|\dot{x}_0| < \texttt{initial\_eps\_prime\_tol}$; as a consequence, when the initial contact velocity is below the threshold no dissipation takes place.

Output

This constitutive law appends data to the output of the elements it is used with in both the textual and NetCDF formats:

- the initial velocity at contact (`xPi` in NetCDF format); when non-zero, it indicates that contact is taking place
- the dissipation coefficient that results from the initial velocity and approximately gives the desired restitution factor (`dc` in NetCDF format)

In the textual format, those fields appear in the file where the element data is output, appended to the element's line after the regular fields. In NetCDF, they appear among the fields of the element, prefixed by `constitutiveLaw.` (e.g. `elem.joint.<label>.constitutiveLaw.xPi`, where `label` is the label of the joint that makes use of the constitutive law, for example a `rod` joint).

Private Data

The same parameters output in the NetCDF format are also available as private data of the element the constitutive law is used with, prefixed by `constitutiveLaw.` (e.g., access to the related datum is granted by the string `"model::element::joint(<label>, \"constitutiveLaw.xPi\")"` where `label` is the label of the joint that makes use of the constitutive law, for example a `rod` joint).

B.2.5 Module-damper-graall

Author: Pierangelo Masarati, based on an original work of Gian Luca Ghiringhelli

This module implements a 1D constitutive law that models the behavior of a landing gear shock absorber. It requires the user to supply the name of the GRAALL-style input file that contains the data of the damper. It will be documented as soon as it reaches an appreciable level of stability. See also the `shock absorber` constitutive law.

B.2.6 Module-damper-hydraulic

Author: Pierangelo Masarati

This module implements a simple hydraulic damper with turbulent orifice and relief valve.

B.2.7 Module-muscles

Authors: Andrea Zanoni and Pierangelo Masarati

This module implements a family of simple muscle constitutive laws, based on the simplified approach proposed in [49].
They are supposed to be applied to Rod elements (see Section 8.12.42).

```
<constitutive_law> ::= muscle ,
    [ model, { pennestri | erf } , ]
    [ initial length , <Li> , ]
    reference length , <L0> ,
    [ reference velocity , <V0> , ]
    reference force , <F0> ,
    activation , (DriveCaller) <activation>
        [ , activation check , (bool) <activation_check>
          , warn, (bool) <warn_user>]
    [ , ergonomy , { yes | no } ]
    [ , reflexive , # only when ergonomy == no
        proportional gain , <kp> ,
        derivative gain , <kd> ,
        reference length , (DriveCaller) <lref> ]
    [ , short range stiffness ]
    [     , model , { exponential | linear } ]
    [     , gamma , <gamma> ]
    [     , delta , <delta> ]
```

The general expression of the muscle force is

$$f_m(x, v, a) = F_0 \left(f_1(x) f_2(v) a(t) + f_3(x) \right) \tag{B.10}$$

with $x = \ell/\ell_0$, $v = -\dot{\ell}/v_0$, and the following meaning of the symbols:

1. ℓ: current length of the muscle, i.e. the instantaneous length as provided by the Rod element the constitutive is attached to;
2. l_0: optimal muscle length for isometric contraction, set through <L0>;
3. v_0: reference contraction velocity, optionally set through <V0>;
4. F_0: maximum contractile force expressed by the muscle in isometric conditions, set trough <F0>;
5. $a(t)$: muscular activation parameter, indicating the fraction of the maximum force that the muscle active component is producing at time t, $0 \leq a \leq 1$;
6. $f_1(x)$ is the active force-length relationship of the muscle (more on this function later);
7. $f_2(v)$ is the active force-velocity relationship of the muscle (more on this function later);
8. $f_3(x)$ is the passive force-length relationship of the muscle (again, more on this function later);

The two model differ in the implementation of the active force-length function $f_1(x)$. The Pennestri model is the one originally proposed in [49]:

$$f_1(x) = \mathrm{e}^{\left(-40(x-0.95)^4+(x-0.95)^2\right)} \tag{B.11}$$

Symbol	Value
a_1	0.4
b_1	10.0
c_1	-0.55
a_2	0.2
b_2	5.0
c_2	-0.85
a_3	-0.6
b_3	2.4
c_3	-1.35

Table B.1: Constants hard-coded in the `erf` muscle model $f_1(x)$ function.

while the second one, the `erf` model, is an original fitting of data published in [50], based on error functions:

$$f_1(x) = a_1\text{erf}(b_1(x + c_1)) + a_2\text{erf}(b_2(x + c_2)) + a_3\text{erf}(b_3(x + c_3)) \tag{B.12}$$

The constants a_i, b_i, c_i, d are hard-coded, with the values of Table B.1.

The $f_2(v)$ and $f_3(x)$ functions are the same for the two models:

$$f_2(v) = 1.6 - 1.6\text{e}^{\left(\frac{0.1}{(v-1)^2} - \frac{1.1}{(v-1)^4}\right)} \tag{B.13}$$

$$f_3(x) = 1.3\tan^{-1}(0.1(x - 0.22)^{10}) \tag{B.14}$$

$$\tag{B.15}$$

The `ergonomy` flag, when active, indicates that the equivalent damping must not be used. It is used to indicate that the constitutive law acts as an "ergonomy" spring in an inverse kinematics analysis.

The `reflexive` keyword indicates that reflexive behavior is being defined; its use is mutually exclusive with the active status of the `ergonomy` flag.

The `initial length` is equal to the `reference length` unless specified. It is used to differentiate the length of the rod as computed from the input from the length used to formulate the constitutive law.

The `reference velocity` is set to 4<L0> m/s, unless specified.

The second occurrence of the `reference length` is related to the reflexive contribution to muscular activation, which is defined as

$$a = \texttt{activation} + \texttt{kp} \cdot \left(\frac{\ell}{\texttt{L0}} - \frac{\texttt{lref}}{\texttt{L0}}\right) + \texttt{kd} \cdot \frac{\dot{\ell}}{\texttt{V0}} \tag{B.16}$$

Be careful that the resulting requested activation can be outside of the range $0 \leq a \leq 1$. If `activation_check` is `yes`, the compliance of the requested activation with the bounds is verified, and the actual value of the activation will be reset at 0 in case of underflow and 1 in case of overflow. If `warn` is set to yes, a warning will be printed in `stderr` when over/underflow conditions are reached.

The contribution of the muscle short-range stiffness (SRS) can be added with `short range stiffness`. The optional parameter `delta` represents the maximum non-dimensional stretch perturbation, above which the short-range stiffness contribution to the muscle force gradient vanishes. A default value of $5.7 \cdot 10^{-3}$ is set. The optional parameter `gamma` represents the SRS constant, according to the model proposed by De Groote et al. [51]. The default value is 280. The optional keyword `model` activates the selection of the implementation:

- `linear`, which is the default, defines the SRS contribution through a piecewise linear relationship with the muscle stretch (as proposed in [51]):

$$\begin{cases} F_{\text{SRS}} = 0 & \Delta x < 0 \\ F_{\text{SRS}} = \texttt{gamma} \cdot \texttt{F0} \cdot \texttt{activation} \cdot f_1(x) \cdot \Delta x & 0 \leq \Delta x < \texttt{delta} \\ F_{\text{SRS}} = \texttt{gamma} \cdot \texttt{F0} \cdot \texttt{activation} \cdot f_1(x) \cdot \texttt{delta} & \Delta x \geq \texttt{delta} \end{cases} \tag{B.17}$$

  where f_1 is the active force-stretch relationship defined according to the model presented in [49], $x = \texttt{lref}/\texttt{L0}$ and $\Delta x = \frac{\ell}{\texttt{L0}} - \frac{\texttt{lref}}{\texttt{L0}}$;

- `exponential` defines the SRS contribution through a smooth model, preserving the initial gradient in the reference configuration:

$$F_{\text{SRS}} = \text{step}(\Delta x) \cdot \texttt{gamma} \cdot \texttt{F0} \cdot \texttt{activation} \cdot f_1(x) \cdot \texttt{delta} \cdot (1 - \exp(1 - \exp(\Delta x/\texttt{delta}))) \tag{B.18}$$

Output. The module appends additional output to the row of the element it is attached to in the `.jnt` file. The following columns are added:

- `a`: the actual total activation of the muscle;
- `aReq`: the *requested* activation of the muscle, before saturation is applied to confine it in the $0 \leq a \leq 1$ range;
- `f1`: the active force-stretch relationship [49], evaluated at the current stretch;
- `f2`: the active force-contraction velocity relationship [49], evaluated at the current contraction velocity;
- `f3`: the passive force-stretch relationship [49], evaluated at the current stretch;
- `df1dx`: the active force-stretch relationship derivative with respect to the stretch, evaluated at the current stretch;
- `df2dv`: the active force-contraction velocity relationship derivative with respect to the contraction velocity, evaluated at the current contraction velocity;
- `df3dx`: the passive force-stretch relationship derivative with respect to the stretch, evaluated at the current stretch;
- (if `reflexive`) `Kp`: the current value of the proportional gain, in the reflexive contribution;
- (if `reflexive`) `Kd`: the current value of the derivative gain, in the reflexive contribution;
- (if `reflexive`) `Lref`: the current value of `lref`;
- (if `short range stiffness`): `SRSf`: the short-range stiffness force contribution;
- (if `short range stiffness`): `SRSdfdx`: the gradient of the short-range stiffness force contribution with respect to the non-dimensional stretch.

The same parameters are also added to the NetCDF output of the element, with the names indicated above.

B.3 Drive Caller Modules

B.3.1 Module-drive

Simple example of run-time loadable user-defined drive caller.

B.3.2 Module-randdrive

Generates normally distributed pseudo-random numbers with given mean and variance (standard deviation) using the random number handling capabilities provided by Boost[1].

```
<drive_caller> ::= boost random ,
    (real) <mean> , (real) <variance>
```

The module itself supports load-time input parameters:

```
module load : "libmodule-randdrive"
    [ , { seed , (integer) <seed> ]
        | seed input file name , " <seed_input_file_name> " } ]
    [ , seed output file name , " <seed_output_file_name> " ]
```

where the integer `seed` is used to seed the random number generator, whereas `seed_input_file_name` and `seed_output_file_name` are used to load/save the random number generator's state in Boost's internal format.

B.4 Template Drive Caller Modules

B.4.1 Module-eu2phi

This module implements a `TplDriveCaller<Vec3>` drive caller that converts three Euler angles into the corresponding Euler vector.

Syntax:

```
eu2phi ,
    [ help , ]
    [ format , { euler123 | euler313 | euler321 } , ]
    (TplDriveCaller<Vec3>) <drive>
```

Example:

```
# assuming that file drive 1 provides Euler angles 1, 2 and 3 in channels 1, 2, 3:
joint: 10, total joint,
    1001,
    1002,
    orientation constraint, active, active, active,
        eu2phi, format, euler123,
            component,
                file, 1, 1,
                file, 1, 2,
                file, 1, 3;
```

[1] http://www.boost.org/

B.5 Driver Modules

Recall that driver modules are invoked as `file` drivers, whose syntax is

```
<driver_type> ::= file

<normal_arglist> ::= <name> [ , <module_data> ]
```

B.5.1 Module-HID

This module provides input from joysticks in the form of a stream, using the Human Interface Device (HID) standard. It is the user's responsibility to map the device's output to stream channels.

```
<name> ::= joystick

<module_data> ::= <file> ,
    <number_of_buttons> ,
    <number_of_linear_controls>
        [ , scale , <scale_factor> [ , ... ] ] ; # scale_factors default to 1.0
```

This stream makes available `number_of_buttons` + `number_of_linear_controls` channels. It is assumed that linear controls come after buttons; `scale_factor`s apply to linear controls.

The value of buttons is 0 when not pressed, and 1 when pressed.

The value coming from linear controls is (assumed to be) comprised between -`UINT16_MAX` and `UINT16_MAX`; it is multiplied by its `scale_factor` and divided by `UINT16_MAX`, so that the value provided by the channel is comprised between -`scale_factor` and `scale_factor`. In some cases, the value coming from linear controls can be comprised between 0 and `UINT16_MAX`; it is suggested to handle this occurrence, if needed, by means of an additional linear drive caller before using the value provided by this stream driver.

Example:

```
file: 999, joystick,
    "/dev/input/js0",        # joystick device
    8,                       # number of buttons
    6,                       # number of linear controls
        scale, 1., 10., 100., 256., 1., 1.;
```

Note that HID provide events when they occur, whereas the logic of file drivers is that of gathering information and making it consistently available through the solution of a time step.

For buttons, the indication that the button was pressed across two time steps is returned. If the button was released, or pressed and released multiple times, only the indication that is was pressed (at least once) is actually returned.

For linear controls, the last measured value across two time steps is returned.

B.6 Scalar Function Modules

B.6.1 Module-scalarfunc

Simple example of run-time loadable user-defined scalar function.

B.7 Miscellaneous Modules

B.7.1 Module-chrono-interface

Author: Runsen Zhang (supported by Google Summer of Code 2020)
This module is to implement cosimulation between MBDyn and Project Chrono[2].

Usage

To use this module, chrono libraries should be installed first. Basically, libraries `libChronoEngine.so` and `libChronoEngine_multicore.so` should be preinstalled.
This module is composed of three parts,

- `ChronoInterface` element: this element is defined in MBDyn side, to send nodes' kinematic data to the other subsystems;
- MBDyn-Chrono interface functions: these functions are provided to achieve cosimulation between these two softwares;
- User-defined chrono subsystem: this subsystem should be defined by users in the function named `MBDyn_CE_CEModel_Create()`.

Before using the `ChronoInterface` element, the second and third parts above should be complied together as a dynamic library, which is called `libce_model.so` in the following. This library is linked to MBDyn throung module-chrono-interface, and its location should be clearly indicated in file "`Makefile.inc`", as `MODULE_LINK=-L$LIBPATH,-libce_model.so`. To use the shared library in execution process, here `$LIBPATH` is added to `/etc/ld.so.conf`,

```
$ vim /etc/ld.so.conf
$ ldconfig
```

ChronoInterface Element

```
<user defined> ::= <user defined type>, <normal_arglist>;
<user defined type> ::= ChronoInterface,
<normal_arglist> ::=
        <cosimulation_platform>,
        <coupling_variables>,
        <coupling_bodies>,
        <other_settings>;
<cosimulation_platform> ::=
        coupling, {none | <loose_coupling> | <tight_coupling> },
        <loose_coupling> ::=
                loose, [{embedded | jacobian | gauss}],
        <tight_coupling> ::=
                tight, (int) <num_iterations>, {tolerance, (real) <tolerance>},
<coupling_variables> ::=
        [, force type, {reaction | contact}]
        [, motor type, {pos | vel | acc, <acc_parameters> | origin}]
        [, length scale, (real) <length_scale>],
```

[2]last accessed, `http://projectchrono.org/`

```
        [, mass scale,(real) <math_scale>],
        <acc_parameters> ::=
                (real) <alpha>, (real) <beta>, (real) <gamma>,
<coupling_bodies> ::=
        nodes number, (int) <num_coupling_nodes>,
        <mbdyn_nodes_info>, <chrono_bodies_infor>,
        [<coupling_constraint>], [<chbody_outputs>]... <chrono_ground>,
        <mbdyn_nodes_info> ::=
                mbdyn node, (int) <node_label>,
                [, offset, (Vec3) <offset_coupling_point>,
                rotation, (Mat3x3) <relative_orientation>],
        <chrono_bodies_info> ::=
                chrono body, (int) <chbody_label>,
                [, offset, (Vec3) <offset_coupling_point>],
        <coupling_constraints> ::=
                [, position constraint, (int) <bool_x>, (int) <bool_y>, (int) <bool_z>],
                [, rotation constraint, (int) <bool_x>, (int) <bool_y>, (int) <bool_z>],
        <chbody_output> ::=
                [, output chbody, yes | no],
        <chrono_ground> ::=
                ground, (int) <chbody_label>,
                [, position, (Vec3 <absolute_position>)],
                [, orientation, (Mat3x3 <absolute_rotation>)],
                [<chbody_output>],
<other_settings> ::=
        [output all chbodies],
        [coupling forces filename, (filename) <filename>],
        [verbose];
```

Example

```
# ...
begin: elements
# ...
        module load: "libmodule-chrono-interface";
        user defined: 1, ChronoInterface,
                # coupling, tight, 4, tolerance, 0.001,
                coupling, loose, embedde, # jacobian,
                force type, reaction, # contact,
                motor type, acc, 2.0/(time_step^2), 2.0/time_step, 1.0,
                # pos, vel, origin,
                length scale, 1000.0,
                mass scale, 1,0,
                nodes number, 1,
                        mbdyn node, BEAM_ROOT_NODE+2*BEAM_N, offset,
                                reference, REF_POINT_E, 0.0, 0.0, 0.0,
                        chrono body, 1, offset, 0.0, 0.0, 0.0,
                        position constraint, 1, 1, 1,
                        orientation constraint, 1, 1, 1,
                        output chbody, yes,
```

```
                    ground, 0,
                            position, reference,  REF_POINT_E, 0.0, 0.0, 0.0,
                            orientation, reference, REF_POINT_E, eye,
                    coupling forces filename, "test/force.dat",
                    verbose
                    ;
    end:elements
```

B.7.2 Module-flightgear

Author: Luca Conti (supported by Google Summer of Code 2017)

This module implements...
It consists of three components:

- a stream modifier for FlightGear flight dynamics data
- a stream modifier for FlightGear controls data
- a customization of the file drive caller to support retrieval of FlightGear specific data from stream drives (see Section 2.6.15)

The stream modifiers are used in conjunction with stream output elements for communication to FlightGear (see Section 8.14.1), and with stream drivers for communication from FlightGear (see Section 7.1.5).

Stream Output Element

```
<user_defined> ::= <user_defined_type> ,
    { NetFDM | NetCtrls }
    [ , print options ]
    , <var_name> , <value>
    [ , ...]

<user_defined_type> ::= flightgear
```

where <value> is currently either drive or nodedof, followed by the related arguments as illustrated in Streamed output (see Sec. 8.14.1). The supported values for <var_name> are listed below with reference to flight dynamics (NetFDM) or controls (NetCtrls).

Stream Drive

```
<user_defined> ::= <user_defined_type> ,
    { NetFDM | NetCtrls }
    [ , print options ]

<user_defined_type> ::= flightgear
```

Flight Dynamics Available Channels

-

Controls Available Channels

-

File Drive Customization

```
<user_defined> ::= <user_defined_type> , <var_name>

<user_defined_type> ::= flightgear
```

The supported names are those listed above for flight dynamics and controls. The correct subset is chosen based on the type of stream drive the label refers to, i.e. when the file drive caller points to a stream driver related to flight dynamics (NetFDM), the flight dynamics related values for var_name must be used. Similarly, the controls related values for var_name must be used when the file drive caller points to a stream driver related to controls (NetCtrls).

Example

```
# ...
\begin{drivers}
# ...
    file: CTRLS, stream,
        name, "CTRLS",
        create, yes,
        port, 9011,
        socket type, udp,
        non blocking,
        flightgear, NetCtrls, print options;
# ...
\end{drivers}
\begin{elements}
# ...
    stream output: FDM,
        stream name, "FG_FDM",
        create, no,
        port, 9012,
        host, "0.0.0.0",
        socket type, tcp,
        non blocking,
        flightgear, NetFDM, print options,
            longitude, drive, node, AIRCRAFT, structural, string, "X[1]", direct,
            latitude, drive, node, AIRCRAFT, structural, string, "X[2]", direct;

    ..., file, CTRLS, flightgear, aileron;                      # use as is
    ..., file, CTRLS, flightgear, throttle_01, amplitude, 100.;  # scale
\end{elements}
```

B.7.3 Module-octave

Author: Reinhard Resch
This module implements support for octave-based user-defined entities, including drive callers, constitutive laws, scalar functions, and elements.

General octave flags

Every GNU Octave based entity supports the following flags:

```
<octave_flags> ::= [ update octave variables, { yes | no | (bool) <status> }, ]
    [ update mbdyn variables, { yes | no | (bool) <status> }, ]
    [ update global variables, { yes | no | (bool) <status> }, ]
    [ pass data manager, { yes | no | (bool) <status> }, ]
    [ embed octave, { yes | no | (bool) <status> }, ]
    [ octave search path, (string) <path1> [ , ... ] ]
```

Global variables If the flag `update octave variables` is enabled, MBDyn's symbol table is copied to the global Octave name space before each function call. This flag should be used only if the Octave function has to access things like plug in variables which depend on the state of the simulation. If this flag is not enabled (the default) MBDyn's symbol table is copied only before the first call of any Octave function. If the flag `update mbdyn variables` is enabled, MBDyn's symbol table is updated by Octave's global variables after each function call. Only those variables that exist in MBDyn's symbol table are updated. The flag `update global variables` is a combination of `update octave variables` and `update mbdyn variables`.

Additional arguments If the flag `pass data manager` is enabled, a pointer to MBDyn's data manager is passed to the Octave function as the last argument.

Embed octave The flag `embed octave` makes it possible to write Octave functions directly inside the MBDyn input file. This is especially useful for small models. If this flag is enabled, the MBDyn input file where `embed octave` appears will be sourced into Octave before the first function call. In order to work it is necessary to put all MBDyn specific commands inside a #{ ... #} comment, and to put all Octave specific code inside a #/* ... #*/ comment.

Octave's search path Directories which should be appended to Octave's search path should appear after `octave search path`. This is rather a global option and is not specific to a particular element or drive caller.

Element

An Octave element is a full featured user defined MBDyn element written in GNU Octave scripting language. From Octave's point of view an Octave element is a special class.

```
<elem_type> ::= user defined

<normal_arglist> ::= octave , " <class_name> "
    [ , <octave_flags> ]
    [ , <elem_data> ]
```

All member functions of this class must reside in a directory named "`@<class_name>`". The parent directory of "`@<class_name>`" must be in Octave's search path. Additional Element specific data in `<elem_data>` like node labels, vectors and rotation matrices may be accessed from within the constructor of the Octave class. If the `octave-ad` package is installed, the `D` function can be used to in order to compute the Jacobian matrix from the residual.

Octave element member functions At the moment the following member functions are supported:

- `[elem] = <class_name>(pMbElem, pDM, HP)`
- `[iRows, iCols] = WorkSpaceDim(elem)`
- `[iNumDof] = iGetNumDof(elem) # optional; iNumDof == 0 if missing`
- `SetValue(elem, XCurr, XPrimeCurr) # optional`
- `[f, ridx] = AssRes(elem, dCoef, XCurr, XPrimeCurr)`
- `[Jac, ridx, cidx, bSparse] = AssJac(elem, dCoef, XCurr, XPrimeCurr) # optional`
- `[elem] = Update(elemin, XCurr, XPrimeCurr) # optional`
- `[iRows, iCols] = InitialWorkSpaceDim(elem) # optional`
- `[iNumDof] = iGetInitialNumDof(elem) # optional`
- `SetInitialValue(elem, XCurr) # optional`
- `[f, ridx] = InitialAssRes(elem, XCurr) # optional`
- `[Jac, ridx, cidx, bSparse] = InitialAssJac(elem, XCurr) # optional`
- `[order] = GetDofType(elem, i) # optional`
- `[order] = GetEqType(elem, i) # optional`
- `[elem] = AfterPredict(elemin, XCurr, XPrimeCurr) # optional`
- `[elem] = AfterConvergence(elemin, XCurr, XPrimeCurr) # optonal`
- `[iNumPrivData] = iGetNumPrivData(elem) # optional`
- `[iPrivDataIdx] = iGetPrivDataIdx(elem, name) # optional`
- `[dPrivData] = dGetPrivData(elem, i) # optional`
- `[iNumConnectedNodes] = iGetNumConnectedNodes(elem) # optional`
- `[connectedNodes] = GetConnectedNodes(elem) # optional`
- `Output(elem, outStream) # optional`
- `DescribeDof(elem, out, prefix, bInitial) # optional`
- `DescribeEq(elem, out, prefix, bInitial) # optional`
- `Restart(elem, out) # optional`

Semantic and parameters of these functions are almost the same like in C++.
TODO: Describe the meaning of the parameters!

Drive Caller

An Octave drive caller could be used as an alternative to string drives or GiNaC drives. If the `octave-ad` package is installed, Octave based drive callers are differentiable by default. An arbitrary number of additional arguments can be passed to the Octave function <function_name> after the arguments keyword.

```
<drive_caller> ::= octave , " <function_name> "
    [ , <octave_flags> ]
    [ , arguments, (integer) <count>, <arg_1> [ , ... , <arg_n> ] ]
```

Template Drive Caller

The syntax of a template drive caller is the same like the scalar drive caller. Of course the Octave function must return a matrix or a vector according to the dimension of the drive caller. The template drive caller is also differentiable by default.

```
<tpl_drive_caller> ::= octave , " <function_name> "
    [ , <octave_flags> ]
    [ , arguments, (integer) <count>, <arg_1> [ , ... , <arg_n> ] ]
```

Scalar Function

```
<scalar_function> ::= octave , " <function_name> "
    [ , <octave_flags> ]
    [ , arguments, (integer) <count>, <arg_1> [ , ... , <arg_n> ] ]
```

Constitutive Law

Octave constitutive laws are full featured user defined constitutive laws. Additional constitutive law specific data in <const_law_data> may be accessed from within the constructor of the Octave class.

```
<specific_const_law> ::= octave , " <class_name> "
    [ , <octave_flags> ]
    dimension , <dimension>
    [ , <const_law_data> ]
```

Octave constitutive law member functions At the moment the following member functions are supported:

- `[cl] = <class_name>(pDM, HP)`
- `[cl, F, FDE, FDEPrime] = Update(clin, Eps, EpsPrime)`
- `[constLawType] = GetConstLawType(cl)`

If the `octave-ad` package is installed, the `D` function can be used to compute `FDE` and `FDEPrime` from function `F`. Semantic and parameters of these functions are almost the same like in C++.
TODO: Describe the meaning of the parameters!

B.7.4 Module-tclpgin

This module implements the Tcl plugin in form of run-time loadable module.

B.7.5 Module-template

Loadable element template (deprecated).

B.7.6 Module-udunits

Author: Pierangelo Masarati
This module implements a namespace called `units` that provides unit-conversion capabilities to the mathematical parser, based on the UDUnits library. The namespace provides only the function `convert`, which is thus invoked as `units::convert`. This function's prototype is

```
(real) units::convert( (string) <from> , (string) <to> [ , (real) <val> ] )
```

where

- `from` is a string indicating what units to convert from;
- `to` is a string indicating what units to convert to;
- `val` is an optional real argument that indicates what value should be converted.

When called with just the two mandatory arguments, the function returns the scale factor between the two units; for example

```
units::convert("ft", "m")
```

returns 0.3048.

When called with the third optional argument `val` as well, the function returns the optional argument `val` converted according to the conversion arguments; for example

```
units::convert("ft", "m", 10.)
```

returns 3.048, corresponding to

```
0.3048 m/ft * 10 ft
```

The first form may not be enough, for example, for those conversions that include an "intercept"; for example, when converting from degrees Celsius to degrees Fahrenheit, the actual conversion would be

```
F = 32 + 9 / 5 * C
```

In those cases, the conversion fails, unless the third argument is specified; for example

```
units::convert("Celsius", "Fahrenheit")
```

would fail, while

```
units::convert("Celsius", "Fahrenheit", 0.)
```

would return 32.

This module requires UNIDATA's `libudunits`; get it from

`http://www.unidata.ucar.edu/software/udunits/`.

Some Linux distributions provide it packaged; for example, recent Ubuntu releases provide a library `libudunits2`; to use it, tweak the configuration variables in `modules/module-udunits/Makefile.inc`.

Appendix C

NetCDF Output Format

NetCDF support has been initially contributed by Patrick Rix.

NetCDF is a format to efficiently store and retrieve data to and from a database on file in a portable, platform-independent manner. Further details can be found at

`http://www.unidata.ucar.edu/software/netcdf/`

and in [52]. The link reported above also describes some of the common tools that can be used to read and manipulate the contents of the database.

The output in NetCDF format consists in a single binary file written by the NetCDF library and intended to be read by tools exploiting the library itself. This document does not describe the details of NetCDF low-level format, since this is not intended to be accessed directly by MBDyn users. Interested readers can consult the specific documentation [52].

The output in NetCDF format is a work-in-progress, so it may be subjected to changes during the development of MBDyn. This chapter is intended to document its current status, so it may be incomplete, and occasionally outdated as well.

C.1 NetCDF Output

Following the convention of NetCDF data, each datum is defined by a variable, whose name indicates the type of datum and the entity that generated it, organized in a tree-like fashion.

For example, the vector containing the three components of the position of the `structural` node labeled `label` is

`node.struct.<label>.X`

Each variable usually has few attributes:

- a `description` is usually given;
- the `units` are specified, unless variables are non-dimensional;
- the `type` is specified, if relevant; it contains the name of the C/C++ structure the data was taken from.

Each level contains some datum that is intended to make the contents of the database as self-explanatory as possible. For example, the level `node` contains an array whose values are the strings indicating the node types available; each level `node.<type>` contains the labels of the available nodes for that type, and so on.

C.1.1 Base Level

Currently, the following basic levels are available:

- `run`, for general simulation-related data;
- `node`, for nodes;
- `elem`, for elements.

C.1.2 Run Level

There is no variable `run`, with the name of the run level. This level contains general, simulation-related data:

- `run.step`, the step number;
- `time` (previously `run.time`), the value of the time;
- `run.timestep`, the value of the timestep.

Note: the `run.time` variable has been renamed `time` because many NetCDF manipulation tools automatically recognize this name as the ordering variable for the rest of the data.

C.1.3 Node Level

There variable `node` contains the number of node types that are present in the output. Its attribute `types` is a semicolon-separated string of the node types.

This level contains as many sublevels as the node types that are present in the model. Each node type consists in a variable that contains an array with the labels (integers) of the nodes defined for that type. A variable named `node.<type>.<label>` may be present (it is, for example, for `structural nodes`); however, useful data are usually contained in specific variables, whose names describe the contents.

Currently supported node types are:

- Abstract nodes, indicated as `abstr`.
- Electric nodes, indicated as `elec`.
- Pressure nodes, indicated as `pres`.
- Structural nodes, indicated as `struct`.
- Thermal nodes, indicated as `thermal`.

Abstract Node

The following variables are defined.

- `node.abstr.<label>` is actually empty; its `type` attribute contains `differential`, indicating that the time derivative of the state is present;
- `node.elec.<label>.X` contains the state of the node;
- `node.elec.<label>.XP` contains the time derivative of the state of the node.

Electric Node

The following variables are defined.

- node.elec.<label> is actually empty; its type attribute contains differential, indicating that the time derivative of the state (of the voltage) is present;
- node.elec.<label>.V contains the voltage of the node;
- node.elec.<label>.VP contains the time derivative of the voltage of the node.

Pressure Node

The following variables are defined.

- node.pres.<label> is actually empty; its type attribute contains algebraic, indicating that the time derivative of the state (of the pressure) is not present;
- node.pres.<label>.p contains the pressure of the node;

Structural Node

The following variables are defined.

- node.struct.<label> is actually empty; it contains the type of the structural node in the type attribute;
- node.struct.<label>.X contains the position of the node, in the appropriate reference frame;
- node.struct.<label>.R contains the orientation matrix of the node, in the appropriate reference frame;
- node.struct.<label>.Phi contains the orientation vector describing the orientation of the node, in the appropriate reference frame;
- node.struct.<label>.E contains the Euler angles describing the orientation of the node, in the appropriate reference frame;
- node.struct.<label>.XP contains the velocity of the node, in the appropriate reference frame;
- node.struct.<label>.Omega contains the angular velocity of the node, in the appropriate reference frame.

Note that only one of R, Phi, or E are present, depending on the requested description of the orientation of the node (see structural node, Section 6.5, and default orientation, Section 5.3.12).

The dynamic and modal node types, if requested, can output extra variables.

- node.struct.<label>.XPP contains the acceleration of the node, in the appropriate reference frame;
- node.struct.<label>.OmegaP contains the angular acceleration of the node, in the appropriate reference frame.

If requested, the inertia associated with dynamic nodes is output within the namespace of the node, although actually handled by the corresponding automatic structural element. As a consequence, extra variables can appear in the output of dynamic nodes.

- node.struct.<label>.B contains the momentum of the inertia associated with the node, in the appropriate reference frame;
- node.struct.<label>.G contains the momenta moment of the inertia associated with the node, in the appropriate reference frame;
- node.struct.<label>.BP contains the derivative of the momentum of the inertia associated with the node, in the appropriate reference frame;
- node.struct.<label>.GP contains the derivative of momenta moment of the inertia associated with the node, in the appropriate reference frame.

Thermal Node

The following variables are defined.

- node.thermal.<label> is actually empty; its type attribute contains differential, indicating that the time derivative of the state (of the temperature) is present;
- node.thermal.<label>.T contains the temperature of the node;
- node.thermal.<label>.TP contains the time derivative of the temperature of the node.

C.1.4 Element Level

There variable elem contains the number of element types that are present in the output. Its attribute types is a semicolon-separated string of the element types.

This level contains as many sublevels as the element types that are present in the model. Each element type consists in a variable that contains an array with the labels (integers) of the elements defined for that type. A variable named elem.<type>.<label> may be present; however, useful data are usually contained in specific variables, whose names describe the contents.

Currently supported element types are:

- Automatic structural elements, indicated as autostruct.
- Beam elements, indicated as beam2 and beam3 for 2- and 3-node beam elements, respectively.
- Force and couple elements, indicated as force and couple.
- Joint elements, indicated as joints.

Aerodynamic Elements

NetCDF output of aerodynamic elements has been sponsored by REpower Systems AG.

The aerodynamic elements allow to define a broad set of variables. Values are output at the integration points. Each element allows to define n integration points. The aerodynamic body outputs at $N = n$ points. The aerodynamic beam2 outputs at $N = 2 \cdot n$ points. The aerodynamic beam3 outputs at $N = 3 \cdot n$ points.

Output.

- elem.aerodynamic.<label>.alpha_<i> contains the angle of attack, in degrees
- elem.aerodynamic.<label>.gamma_<i> contains the sideslip angle, in degrees
- elem.aerodynamic.<label>.Mach_<i> contains the Mach number
- elem.aerodynamic.<label>.cl_<i> contains the lift coefficient
- elem.aerodynamic.<label>.cd_<i> contains the drag coefficient
- elem.aerodynamic.<label>.cm_<i> contains the moment coefficient
- elem.aerodynamic.<label>.X_<i> contains the location of the integration point i, in the global reference frame;
- elem.aerodynamic.<label>.R_<i> contains the orientation matrix of the airfoil at the integration point i, in the global reference frame;
- elem.aerodynamic.<label>.Phi_<i> contains the rotation vector that describes the orientation of the airfoil at the integration point i, in the global reference frame;
- elem.aerodynamic.<label>.E_<i> contains the set of Euler angles that describes the orientation of the airfoil at the integration point i, in the global reference frame. The sequence of the angles is detailed in the variable's description. It can be 123 (the default), 313 or 321;
- elem.aerodynamic.<label>.V_<i> contains the velocity of the integration point i, in the global reference frame;
- elem.aerodynamic.<label>.Omega_<i> contains the angular velocity of the integration point i, in the global reference frame;
- elem.aerodynamic.<label>.F_<i> contains the force at the integration point i, in the global reference frame;
- elem.aerodynamic.<label>.M_<i> contains the moment at the integration point i, in the global reference frame, referred to the location of the integration point.

where i $\in [0, N)$.

Note: variables R_<i>, Phi_<i> and E_<i> are mutually exclusive.

Note: by default, only variables alpha_<i>, gamma_<i>, Mach_<i>, cl_<i>, cd_<i>, cm_<i>, F_<i>, and M_<i> are output. See Section 8.1.1 for indications about how to customize aerodynamic element output in NetCDF format.

Induced Velocity Elements

The following variables are defined.

- elem.inducedvelocity.<label>, actually empty, it contains the type of the induced velocity element in the type attribute.

only rotor induced velocity elements are currently supported.

Rotor

- elem.inducedvelocity.<label>.f contains the rotor force in x, y and z directions (longitudinal, lateral and thrust components);
- elem.inducedvelocity.<label>.m contains the rotor moment about x, y and z directions (roll, pitch and torque components);
- elem.inducedvelocity.<label>.UMean contains the mean inflow velocity, based on momentum theory;
- elem.inducedvelocity.<label>.VRef contains the rotor center reference velocity, sum of aistream and craft_node velocity;
- elem.inducedvelocity.<label>.Alpha contains the rotor disk angle;
- elem.inducedvelocity.<label>.Mu contains the advance parameter μ;
- elem.inducedvelocity.<label>.Lambda contains the inflow parameter λ;
- elem.inducedvelocity.<label>.Chi contains the advance/inflow parameter $\chi = \tan^{-1}(\mu/\lambda)$;
- elem.inducedvelocity.<label>.Psi0 contains the reference azimuthal direction ψ_0, related to rotor yaw angle;
- elem.inducedvelocity.<label>.UMeanRefConverged contains a boolean flag indicating convergence in reference induced velocity computation internal iterations;
- elem.inducedvelocity.<label>.Iter contains the number of iterations required for convergence;

In the case the inflow model is dynamic inflow, the following additional output is written

- elem.inducedvelocity.<label>.VConst constant inflow angle;
- elem.inducedvelocity.<label>.VSine sine inflow state (lateral);
- elem.inducedvelocity.<label>.VCosine cosine inflow state (longitudinal);

Automatic Structural

No variables are explicitly defined for the automatic structural element; on the contrary, their specific data, if requested, is appended to the corresponding dynamic structural node.

Beam

The beam elements allow to define a broad set of variables.

Two-Node Beam Element. The two-node beam element allows to define the variables

- elem.beam.<label>.X contains the location of the evaluation point, in the global reference frame;
- elem.beam.<label>.R contains the orientation matrix of the beam section at the evaluation point, in the global reference frame;
- elem.beam.<label>.Phi contains the rotation vector that describes the orientation of the beam section at the evaluation point, in the global reference frame;

- elem.beam.<label>.E contains the set of Euler angles that describes the orientation of the beam section at the evaluation point, in the global reference frame. The sequence of the angles is detailed in the variable's description. It can be 123 (the default), 313 or 321;
- elem.beam.<label>.F contains the internal force at the evaluation point, in the reference frame of the beam section;
- elem.beam.<label>.M contains the internal moment at the evaluation point, in the reference frame of the beam section;
- elem.beam.<label>.nu contains the linear strain at the evaluation point, in the reference frame of the beam section;
- elem.beam.<label>.k contains the angular strain at the evaluation point, in the reference frame of the beam section;
- elem.beam.<label>.nuP contains the linear strain rate at the evaluation point, in the reference frame of the beam section;
- elem.beam.<label>.kP contains the angular strain rate at the evaluation point, in the reference frame of the beam section.

Note: variables R, Phi and E are mutually exclusive.
Note: variables muP and kP are only available for viscoelastic beam elements.
Note: by default, only variables F and M are output. See Section 8.3.2 for indications about how to customize beam element output in NetCDF format.

Three-Node Beam Element. The three-node beam element allows to define the same set of variables of the two-node one, postfixed with either _I or _II to indicate the first (between nodes 1 and 2) or the second (between nodes 2 and 3) evaluation point.

Driven

When an element is driven, a scalar field driven is added to the driven element's entry. Its value is 1 when the driven element is active, and 0 otherwise. The content of the driven element's output should be disregarded when this flag is 0.

Force

The various flavors of force and couple elements allow different variables.

Absolute displacement force: the type is absolute displacement.

- elem.force.<label>.F contains the three components of the force, in the absolute reference frame.

Internal absolute displacement force: the type is internal absolute displacement.

- elem.force.<label>.F contains the three components of the force, in the absolute reference frame.

Absolute force: the type is absolute.

- elem.force.<label>.F contains the three components of the force, in the absolute reference frame.
- elem.force.<label>.Arm contains the three components of the arm, in the absolute reference frame.

Follower force: the type is `follower`.

- `elem.force.<label>.F` contains the three components of the force, in the absolute reference frame.
- `elem.force.<label>.Arm` contains the three components of the arm, in the absolute reference frame.

Absolute couple: the type is `absolute`.

- `elem.couple.<label>.M` contains the three components of the couple, in the absolute reference frame.

Follower couple: the type is `follower`.

- `elem.couple.<label>.M` contains the three components of the couple, in the absolute reference frame.

Internal absolute force: the type is `internal absolute`.

- `elem.force.<label>.F` contains the three components of the force, in the absolute reference frame.
- `elem.force.<label>.Arm1` contains the three components of the arm with respect to node 1, in the absolute reference frame.
- `elem.force.<label>.Arm2` contains the three components of the arm with respect to node 2, in the absolute reference frame.

Internal follower force: the type is `internal follower`.

- `elem.force.<label>.F` contains the three components of the force, in the absolute reference frame.
- `elem.force.<label>.Arm1` contains the three components of the arm with respect to node 1, in the absolute reference frame.
- `elem.force.<label>.Arm2` contains the three components of the arm with respect to node 2, in the absolute reference frame.

Internal absolute couple: the type is `internal absolute`.

- `elem.couple.<label>.M` contains the three components of the force, in the absolute reference frame.

Internal follower couple: the type is `internal follower`.

- `elem.couple.<label>.M` contains the three components of the force, in the absolute reference frame.

Joint

All the joint elements write the following variables

- elem.joint.<label>.f contains the three components of the reaction force, in the relative reference frame.
- elem.joint.<label>.m contains the three components of the reaction moment, in the relative reference frame.
- elem.joint.<label>.F contains the three components of the reaction force, in the absolute reference frame.
- elem.joint.<label>.M contains the three components of the reaction moment, in the absolute reference frame.

Additional variables can be added, depending on the joint type.

Angular acceleration:

- elem.joint.<label>.wP contains the three components of the imposed angular acceleration, in the joint reference frame.

Angular velocity:

- elem.joint.<label>.w contains the three component of the axis about which the angular velocity is imposed, in the absolute reference frame;
- elem.joint.<label>.dOmega contains the absolute value of the imposed angular velocity.

Axial rotation:

- elem.joint.<label>.Phi contains the relative orientation of the connected nodes, expressed in the joint reference frame attached to node 1;
- elem.joint.<label>.Omega contains the relative angular velocity of the two connected nodes, expressed in the joint reference frame attached to node 1.

If friction is present:

- elem.joint.<label>.MR contains the friction moment;
- elem.joint.<label>.fc contains the friction coefficient.

Beam slider:

- elem.joint.<label>.Beam contains the label of the current beam;
- elem.joint.<label>.sRef contains the current curvilinear abscissa;
- elem.joint.<label>.l contains the three components of the local direction unit vector, in the absolute reference frame.

If friction is present:

- elem.joint.<label>.FF contains the tangential force component;
- elem.joint.<label>.fc contains the friction coefficient;
- elem.joint.<label>.v contains the sliding velocity.

Brake:

- elem.joint.<label>.Phi contains the orientation vector expressing the relative orientation of the connected nodes, with respect to node 1;
- elem.joint.<label>.Omega contains the relative angular velocity, expressed in the reference frame of node 2.

If friction is present:

- elem.joint.<label>.fc contains the friction coefficient;
- elem.joint.<label>.Fb contains the brake force.

Cardano rotation/pin:

- elem.joint.<label>.Phi contains the orientation vector expressing the relative orientation of the connected nodes, expressed in the reference frame of node 2.

Cardano hinge:

- elem.joint.<label>.E contains the Euler angles expressing the relative orientation of the connected nodes, expressed in the reference frame of node 2;
- elem.joint.<label>.Phi contains the orientation vector expressing the relative orientation of the connected nodes, expressed in the reference frame of node 2;
- elem.joint.<label>.R contains the orientation matrix expressing the relative orientation of the connected nodes, expressed in the reference frame of node 2;

Deformable axial joint:

- elem.joint.<label>.Theta contains the relative angle between the connected nodes;
- elem.joint.<label>.Omega contains the relative angular velocity between the connected nodes.

Deformable displacement joint:

- elem.joint.<label>.d contains the relative position of the connected nodes, in the joint local frame;
- elem.joint.<label>.dPrime contains the relative velocity of the connected nodes, in the joint local frame;
- elem.joint.<label>.D contains the relative position of the connected nodes, in the global frame;
- elem.joint.<label>.DPrime contains the relative velocity of the connected nodes, in the global frame.

Deformable hinge:

- `elem.joint.<label>.E` contains the Euler angles expressing the relative orientation of the connected nodes, expressed in the joint reference frame;
- `elem.joint.<label>.Phi` contains the orientation vector expressing the relative orientation of the connected nodes, expressed in the joint reference frame;
- `elem.joint.<label>.R` contains the orientation matrix expressing the relative orientation of the connected nodes, expressed in the joint reference frame;
- `elem.joint.<label>.Omega` contains the relative angular velocity of the connected nodes, expressed in the joint reference frame.

Deformable displacement:

- `elem.joint.<label>.d` contains the relative position of the connected nodes, in the joint local frame;
- `elem.joint.<label>.dPrime` contains the relative velocity of the connected nodes, in the joint local frame;
- `elem.joint.<label>.D` contains the relative position of the connected nodes, in the global frame;
- `elem.joint.<label>.DPrime` contains the relative velocity of the connected nodes, in the global frame.
- `elem.joint.<label>.E` contains the Euler angles expressing the relative orientation of the connected nodes, expressed in the joint reference frame;
- `elem.joint.<label>.Phi` contains the orientation vector expressing the relative orientation of the connected nodes, expressed in the joint reference frame;
- `elem.joint.<label>.R` contains the orientation matrix expressing the relative orientation of the connected nodes, expressed in the joint reference frame;
- `elem.joint.<label>.Omega` contains the relative angular velocity of the connected nodes, expressed in the joint reference frame.

Distance:

- `elem.joint.<label>.V` contains the three components of the unit vector along which the distance is constrained, expressed in the global reference frame;
- `elem.joint.<label>.d` contains the magnitude of the constrained distance;

Drive displacement/Drive displacement pin:

- `elem.joint.<label>.d` contains the three components of the imposed relative displacement, in the global reference frame;

Drive hinge:

- `elem.joint.<label>.Phi` contains the orientation vector expressing the relative orientation of the connected nodes, expressed in the joint reference frame;

Gimbal rotation:

- `elem.joint.<label>.Theta` contains the relative rotation angle about joint axis 1;
- `elem.joint.<label>.Phi` contains the relative rotation angle about joint axis 2;

In line: If friction is present:

- `elem.joint.<label>.FF` contains the friction force along the constraint direction;
- `elem.joint.<label>.fc` contains the friction coefficient.

Linear acceleration:

- `elem.joint.<label>.a` contains the imposed relative linear acceleration.

Linear velocity

- `elem.joint.<label>.v` contains the imposed relative linear velocity.

Revolute hinge:

- `elem.joint.<label>.E` contains the Euler angles expressing the relative orientation of the connected nodes, expressed in the global reference frame;
- `elem.joint.<label>.Phi` contains the orientation vector expressing the relative orientation of the connected nodes, expressed in the global reference frame;
- `elem.joint.<label>.R` contains the orientation matrix expressing the relative orientation of the connected nodes, expressed in the global reference frame;
- `elem.joint.<label>.Omega` contains the relative angular velocity of the connected nodes, expressed in the joint reference frame.

If friction is present:

- `elem.joint.<label>.MRF` contains the friction moment exchanged by the two constrained nodes about the joint axis 3;
- `elem.joint.<label>.fc` contains the friction coefficient.

Revolute rotation:

- `elem.joint.<label>.E` contains the Euler angles expressing the relative orientation of the connected nodes, expressed in the global reference frame;
- `elem.joint.<label>.Phi` contains the orientation vector expressing the relative orientation of the connected nodes, expressed in the global reference frame;
- `elem.joint.<label>.R` contains the orientation matrix expressing the relative orientation of the connected nodes, expressed in the global reference frame;
- `elem.joint.<label>.Omega` contains the relative angular velocity of the connected nodes, expressed in the joint reference frame.

Rod:

- elem.joint.<label>.l contains the length of the element;
- elem.joint.<label>.lP contains the lengthening velocity of the element;
- elem.joint.<label>.v contains the three components of the unit vector representing the element line of action, expressed in the global reference frame;
- elem.joint.<label>.constitutiveLaw.<item> may contain any constitutive law-specific output, if any. See the related constitutive law's NetCDF output options for possible values of item.

Spherical joint:

- elem.joint.<label>.E contains the Euler angles expressing the relative orientation of the connected nodes, expressed in the joint reference frame;
- elem.joint.<label>.Phi contains the orientation vector expressing the relative orientation of the connected nodes, expressed in the joint reference frame;
- elem.joint.<label>.R contains the orientation matrix expressing the relative orientation of the connected nodes, expressed in the joint reference frame;

Spherical pin joint:

- elem.joint.<label>.E contains the Euler angles expressing the relative orientation of the connected nodes, expressed in the joint reference frame;

Total joint:

- elem.joint.<label>.X contains the three components of the relative position of the connected nodes, expressed in the joint reference frame;
- elem.joint.<label>.Phi contains the orientation vector expressing the relative orientation of the connected nodes, expressed in the joint reference frame;
- elem.joint.<label>.V contains the three components of the relative velocity of the connected nodes, expressed in the joint reference frame;
- elem.joint.<label>.Omega contains the three components of the relative angular velocity of the connected nodes, expressed in the joint reference frame.

Viscous body:

- elem.joint.<label>.V contains the three components of the relative velocity of the connected nodes, expressed in the joint reference frame;
- elem.joint.<label>.Omega contains the three components of the relative angular velocity of the connected nodes, expressed in the joint reference frame.

Total pin joint:

- `elem.joint.<label>.Phi` contains the orientation vector expressing the relative orientation of the connected nodes, expressed in the joint reference frame;
- `elem.joint.<label>.X` contains the three components of the relative position of the connected nodes, expressed in the joint reference frame;
- `elem.joint.<label>.V` contains the three components of the relative velocity of the connected nodes, expressed in the joint reference frame;
- `elem.joint.<label>.Omega` contains the three components of the relative angular velocity of the connected nodes, expressed in the joint reference frame.

C.1.5 Eigenanalysis

The binary output of eigenanalysis is structured as follows.

Base level

If at least one eigenanalysis is requested, a base level is added, called `eig`. This level contains general data:

- `eig.idx`, a vector containing the reference row index of each structural node (excluding any dummy node) in the eigenvectors;
- `eig.labels`, a vector containing the labels of each structural node, if any (excluding any dummy node);
- `eig.<elem_type>.idx`, the indices of the degrees of freedom in the eigenvector associated with specific elements (e.g. Lagrange multipliers of joints);
- `eig.<elem_type>.labels`, a vector containing the corresponing labels of the elements;

Analysis level

For each analysis, a level `eig.<index>` is created. The requested outputs are then added to that level.

- `eig.<index>.step`, the step number at which each eigenanalysis was performed;
- `eig.<index>.time`, the time in seconds at which each eigenanalysis was performed;
- `eig.<index>.dCoef`, the coefficient used to build the problem matrices;
- `eig.<index>.X0`, a vector containing the reference position and orientation of the structural nodes, if any (excluding any dummy node); the vector contains, for each node, the three components of the reference position and the three components of the Euler vector corresponding to the reference orientation; all data are referred to the absolute reference frame;

Full matrices. If `output full matrices` was specified, the following variables are added to the `eig.<index>` level:

- `eig.<index>.Aplus` : contains the $\mathbf{J}_{(+c)} = \boldsymbol{f}_{/\dot{\boldsymbol{x}}} + \texttt{dCoef} \cdot \boldsymbol{f}_{/\boldsymbol{x}}$ matrix;
- `eig.<index>.Aminus` : contains the $\mathbf{J}_{(-c)} = \boldsymbol{f}_{/\dot{\boldsymbol{x}}} - \texttt{dCoef} \cdot \boldsymbol{f}_{/\boldsymbol{x}}$ matrix.

Both matrices are stored in full format. The corresponding attributes "matrix type" of `eig.<index>.Aplus` and `eig.<index>.Aminus` are set to "dense".

Sparse matrices. If `output sparse matrices` was specified, the following variables are added to the `eig.<index>` level:

- `eig.<index>.Aplus` : contains the $\mathbf{J}_{(+c)} = \boldsymbol{f}_{/\dot{\boldsymbol{x}}} + \texttt{dCoef} \cdot \boldsymbol{f}_{/\boldsymbol{x}}$ matrix;
- `eig.<index>.Aminus` : contains the $\mathbf{J}_{(-c)} = \boldsymbol{f}_{/\dot{\boldsymbol{x}}} - \texttt{dCoef} \cdot \boldsymbol{f}_{/\boldsymbol{x}}$ matrix.

and the following dimensions specified

- `eig_<index>_Aplus_sp_iSize` : represents the number of nonzero elements in matrix `Aplus`;
- `eig_<index>_Aminus_sp_iSize` : represents the number of nonzero elements in matrix `Aminus`;

Both matrices are stored in sparse triplet format and may be passed to Matlab's spconvert function. The corresponding attributes "matrix type" of `eig.<index>.Aplus` and `eig.<index>.Aminus` are set to "sparse".

Eigenvalues and Eigenvectors. If `output eigenvectors` was specified, the following variables are added to the `eig.<index>` level:

- `eig.<index>.VR` : contains the right eigenvectors matrix;
- `eig.<index>.VL` : contains the left eigenvectors matrix, only written if the chosen method computes it;
- `eig.<index>.alpha` : contains the `alpha` matrix, in which the first row contains the α_r coefficients, the second row the α_i coefficients and the third row the β coefficients, so that the k-th discrete time eigenvalue is

$$\Lambda_k = \frac{\alpha_r(k) + \mathrm{i}\alpha_i(k)}{\beta(k)}. \tag{C.1}$$

  while the corresponding continuous time domain eigenvalue λ_k is

$$\lambda_k = \frac{1}{\texttt{dCoef}} \frac{\Lambda_k - 1}{\Lambda_k + 1} = \frac{1}{\texttt{dCoef}} \frac{\alpha_r(k)^2 + \alpha_i(k)^2 - \beta(k)^2 + \mathrm{i}2\alpha_i(k)\beta(k)}{\left(\alpha_r(k) + \beta(k)\right)^2 + \alpha_i(k)^2}. \tag{C.2}$$

  Note that this means that the transposed of the `alpha` matrix as described in Section 4.1.1 is written in the NetCDF file;

Both matrices `VR` and `VL` (when available) are stored as 3D matrices, where the first index of the third dimension refers to the real part of the eigenvectors and the second index of the third dimension to the imaginary part. So to store the matrices in complex shape in software packages that support is, for example in Octave, a procedure like the following must be used (here making use of the Octave package `netcdf`, see Section C.2.1):

Example 1.

```
# load the VR matrix in 3D format
octave:1> VR_3D = ncread('mbdyn_output.nc', 'eig.0.VR');
octave:2> VR_cpx = VR_3D(:, :, 1) + 1i*VR_3D(:, :, 2);
octave:3> clear VR_3D
octave:4> VR('component', 'mode') % access 'component' of 'mode'
```

The same using the Octave package `octcdf`, see Section C.2.1:

```
# load the VR matrix in 3D format
octave:1> nc = netcdf('mbdyn_output.nc', 'r');
octave:2> VR_3D = nc{'eig.0.VR'}(:);
octave:3> VR_cpx = VR_3D(1, :, :) + 1i*VR_3D(2, :, :);
octave:4> clear VR_3D
octave:5> VR_cpx(component, mode); % access 'component' of 'mode'
```

The following example shows how to reconstruct the discrete time eigenvalues vector `Lambda` and the corresponding continuous time eigenvalues vector `lambda` (see Section 4.1.1):

Example 2.

```
# load alpha and dCoef of eigensolution 0
octave:1> alpha = ncread('mbdyn_output.nc', 'eig.0.alpha');
octave:2> dCoef = ncread('dbp.nc', 'eig.dCoef', 1, 1);

# compute Lambda (discrete time) and lambda (continuous time)
octave:3> Lambda = (alpha(1,:) + 1i*alpha(2,:))./alpha(3,:);
octave:4> lambda = 1/dCoef*((Lambda - 1)./(Lambda + 1));
```

Geometry. As opposed to the `.m` text file output, the reference configuration is not written separately in the NetCDF output, since it is already available from the standard structural nodes' output.

For example, to access the reference configuration of node `10`, during the first calculated eigensolution, the following procedure can be used in Octave (here, again, making use of the Octave package `netcdf`, see Section C.2.1):

Example 1.

```
# access the position and orientation of node 10 during eigensolution 0
octave:1> ts = ncread('mbdyn_output.nc', 'eig.step', 1, 1);
octave:2> X0_10 = ncread('mbdyn_output.nc', 'node.struct.10.X', [1, ts(1)], [3, 1])
octave:3> Phi0_10 = ncread('mbdyn_output.nc', 'node.struct.10.Phi', [1, ts(1)], [3, 1])
```

The following function can be used to recover the complete reference configuration

Example 2.

```
function X0 = eig_geom(ncfile, eig, orient)
    %% EIG_GEOM -- Extracts the reference configuration of the eigensolution of
    %              index eig from MBDyn NetCDF output file ncfile, that is written
    %              with default orientation orient
    %
    %              INPUTS:
    %                 - ncfile -- filename (full path) of MBDyn NetCDF output file
    %                 - eig    -- index of eigensolution of interest
    %                 - orient -- string identifying the orientation used for
    %                             the structural nodes output. Please note that this
    %                             function does not handle multiple output
    %                             orientation descriptions
    %
    %              X0 = eig_geom(ncfile, eig, orient)
```

```
  pkg load netcdf

  step = ncread(ncfile, 'eig.step', eig + 1, 1);
  nodes = ncread(ncfile, 'node.struct');

  if orient == 'R'
    ncoord = 12;
  else
    ncoord = 6;
  end

  X0 = zeros(length(nodes)*6, 1);
  for ii = 1:length(nodes)
    nodevar = ['node.struct.' num2str(nodes(ii)) '.'];
    x_bg = (ii - 1)*ncoord + 1;
    r_bg = x_bg + 3;
    X0(x_bg:x_bg + 2) = ncread(ncfile, [nodevar 'X'], [1 step], [3 1]);
    if orient == 'R'
      X0(r_bg:r_bg + 2) = ncread(ncfile, [nodevar orient], [1 1 step], [3 1 1]);
      X0(r_bg + 3: r_bg + 5) = ncread(ncfile, [nodevar orient], [1 2 step], [3 1 1]);
      X0(r_bg + 6: r_bg + 8) = ncread(ncfile, [nodevar orient], [1 3 step], [3 1 1]);
    else
      X0(r_bg:r_bg + 2) = ncread(ncfile, [nodevar orient], [1 step], [rN 1]);
    end
  end

end
```

The reference configuration of eigensolution 0, with `default orientation` set to `orientation vector` (see Section 5.3.12 and C.1.3) is extracted with

```
octave:1> X0 = eig_geom('mbout.nc', 0, 'Phi')
```

In the same fashion, the reference configuration of eigensolution 2, with `default orientation` set to `orientation matrix` is extracted with

```
octave:1> X0 = eig_geom('mbout.nc', 2, 'R')
```

The orientation matrix of the node 3 (supposing it is also the third node in the model) in the reference configuration is, then

```
octave:2> R3 = zeros(3,3);
octave:3> R3(:) = X0(2*12 + 4, 3*12);
```

C.2 Accessing the Database

The database can be accessed using any of the tools listed at the web site

`http://www.unidata.ucar.edu/software/netcdf/software.html`

(yes, including MBDyn itself...)

C.2.1 Octave

Octave used to provide access to NetCDF databases using the (possibly outdated) `octcdf` package. The preferred package is the `netcdf` package.

OctCDF (WARNING: possibly outdated). The `octcdf` package, available from

`http://ocgmod1.marine.usf.edu/`,

provides a clean interface to using NetCDF databases from within the popular math environment Octave. Results from MBDyn can be easily handled once the data structure is known.

To access the database, a handler needs to be obtained by calling

```
octave:1> nc = netcdf('mbout.nc', 'r');
```

Variable descriptions are accessed as

```
octave:2> nc{'node.struct.1000.X'}
```

Their values are accessed as

```
octave:3> nc{'node.struct.1000.X'}(10, 3)
```

So, for example, the z component of the position of node 1000 can be plot with

```
octave:4> plot(nc{'time'}(:), nc{'node.struct.1000.X'}(:,3))
```

NetCDF. The `netcdf` package appears to have superseded the `octcdf` package in recent distributions of Octave. It is no longer necessary to obtain a handler to the MBDyn output file.

Variable properties are accessed with

```
octave:1> X10 = ncinfo('mbout.nc', 'node.struct.10.X');
```

A single component (in this example the y-position of node `10`) at all time steps is accessed with

```
octave:2> X10_y = ncread('mbout.nc', 'node.struct.10.X', [2, 1], [1, Inf]);
```

Single records (in this example the position of node `10` at step `15`) are accessed with

```
octave:3> X10_s15 = ncread('mbout.nc', 'node.struct.10.X', [1, 15], [3, 1]);
```

Single values (in this example the z-position of node `10` at step `15`) with

```
octave:4> X10_s15_z = ncread('mbout.nc', 'node.struct.10.X', [3, 15], [1, 1]);
```

An orientation matrix (in this example the orientation of node `10` at step `15`) is accessed as

```
octave:5> R10_s15 = ncread('mbout.nc', 'node.struct.10.R', [1, 1, 15], [3, 3, 1]);
```

Element `(2, 3)` is accessed as

```
octave:6> R10_s15_r2_c3 = ncread('mbout.nc', 'node.struct.10.R', [2, 3, 15], [1, 1, 1]);
```

Column `2` is accessed as

```
octave:7> R10_s15_c2 = ncread('mbout.nc', 'node.struct.10.R', [1, 2, 15], [3, 1, 1]);
```

Appendix D

Results Visualization

This section describes the different ways the raw output from MBDyn can be arranged to work with third-party software for visualization. In most of the cases, the preparation of the results can be done as a post-processing, starting from raw MBDyn output files. This is true, for instance, for EasyAnim (see `http://mecara.fpms.ac.be/EasyDyn/` for details).

In previous versions of MBDyn, the `output results` statement allowed to generate output compatible with MSC.ADAMS; provisions for Altair Motion View has been under preparation for long time, but it has never been truly supported. Anyone interested in that interface should contact MBDyn developers. The `output results` statement is now deprecated; support for all visualization tools will be reworked in form of postprocessing of MBDyn's raw output, either in textual or binary form. Right now, that statement is used to enable the experimental support for NetCDF.
Some post-processing preparation instructions are available for those packages that require special handling and thus are built-in.

D.1 EasyAnim

MBDyn exploits a special version of EasyAnim, based on 1.3 sources and patched by MBDyn developers; the patch is known to work under UN*X (GNU/Linux, essentially); it has not been tested with Windows, except with `cygwin`. It is available from the MBDyn web site, and it has been submitted to EasyAnim developers for evaluation. Check out MBDyn's web site for more information.

The preparation of the output is done via `awk(1)`, which is invoked by the `mbdyn2easyanim.sh` script. It uses the `.log` and `.mov` files, and generates a pair of `.vol` and `.van` files which contain the model and the motion, respectively. Use

```
mbdyn2easyanim.sh <mbdyn_output_filename>
```

where `mbdyn_output_filename` is the name of the output file from MBDyn, with no extension. This script generates a model based on the topology information contained in the `mbdyn_output_filename.log` file, plus the animation information according to the contents of the `mbdyn_output_filename.mov` file.

To check the model during preparation, run MBDyn with

```
abort after: input;
```

in the `problem_name` section (page 134), so that the model is output in the input configuration, and EasyAnim can visualize it.

The model can be customized by supplying the `mbdyn2easyanim.sh` script additional `awk(1)` code that adds nodes, edges and sides whose movement is associated with that of existing nodes.

For this purpose, one needs to write an `awk(1)` script like

```
BEGIN {
    # add a node property
    # node prop name, color, radius
    nodeprop_add("dummy_node", 4, 1.);

    # add a node
    # node name, base node name, x, y, z in base node frame, prop name
    node_add("added_node", "base_node", 1., 2., 3., "dummy_node");

    # add an edge property
    # edge prop name, color, radius
    edgeprop_add("dummy_edge", 15, 2.);

    # add an edge
    # edge name, node 1 name, node 2 name, prop name
    edge_add("added_edge", "node_1", "node_2", "dummy_edge");

    # add a side property
    # side prop name, color
    sideprop_add("dummy_side", 8);

    # add a side
    nodes[1] = "node_1";
    nodes[2] = "node_2";
    nodes[3] = "node_3";
    nodes[4] = "node_4";
    # side name, number of nodes, node labels, prop name
    side_add("added_side", 4, nodes, "dummy_side");
}
```

place it into a file, say `custom.awk`, and then invoke the interface script as

```
mbdyn2easyanim.sh -f custom.awk <mbdyn_output_filename>
```

D.2 Output Manipulation

This section notes how output can be manipulated in useful manners.

D.2.1 Output in a Relative Reference Frame

In many cases it may be useful to represent the output of `structural` nodes in a reference frame relative to one node. For example, to investigate the deformation of a helicopter rotor, it may be useful to refer the motion of the blades to the reference frame of the rotor hub, so that the reference rotation of the rotor is separated from the local straining.

For this purpose, MBDyn provides an `awk(1)` script that extracts the desired information from the `.mov` file.

The script is `$SRC/var/abs2rel.awk` and is usually installed in `$PREFIX/share/awk`.

To invoke the script run

```
$ awk -f abs2rel.awk \
    -v RefNode=<label> \
```

```
[-v RefOnly={0|1}] \
[-v InputMode=<imode>] \
[-v OutputMode=<omode>] \
infile.mov > outfile.mov
```

It requires the specification of the reference node label, which is given by the `RefNode` parameter.

The optional parameter `RefOnly` determines whether the output is transformed in a true reference frame, or it is simply re-oriented according to the orientation of the reference node.

The command also allows to specify the input and output formats of the orientations, under the assumption that rotations of all nodes are generated with the same format. The optional parameters `imode` and `omode` can be:

- `euler123` for the Euler-like angles, in degrees, according to the 123 sequence;
- `euler313` for the Euler-like angles, in degrees, according to the 313 sequence;
- `euler321` for the Euler-like angles, in degrees, according to the 321 sequence;
- `vector` for the orientation vector, in radians;
- `matrix` for the orientation matrix.

When `RefOnly` is set to 0 (the default), this script basically computes the configuration of each node as function of that of the reference one, namely

$$\mathbf{R}'_n = \mathbf{R}_0^T \mathbf{R}_n \tag{D.1}$$
$$\boldsymbol{x}'_n = \mathbf{R}_0^T (\boldsymbol{x}_n - \boldsymbol{x}_0) \tag{D.2}$$
$$\boldsymbol{\omega}'_n = \mathbf{R}_0^T (\boldsymbol{\omega}_n - \boldsymbol{\omega}_0) \tag{D.3}$$
$$\dot{\boldsymbol{x}}'_n = \mathbf{R}_0^T (\dot{\boldsymbol{x}}_n - \dot{\boldsymbol{x}}_0 - \boldsymbol{\omega}_0 \times (\boldsymbol{x}_n - \boldsymbol{x}_0)) \tag{D.4}$$
$$\dot{\boldsymbol{\omega}}'_n = \mathbf{R}_0^T (\dot{\boldsymbol{\omega}}_n - \dot{\boldsymbol{\omega}}_0 - \boldsymbol{\omega}_0 \times (\boldsymbol{\omega}_n - \boldsymbol{\omega}_0)) \tag{D.5}$$
$$\ddot{\boldsymbol{x}}'_n = \mathbf{R}_0^T (\ddot{\boldsymbol{x}}_n - \ddot{\boldsymbol{x}}_0 - \dot{\boldsymbol{\omega}}_0 \times (\boldsymbol{x}_n - \boldsymbol{x}_0) - \boldsymbol{\omega}_0 \times \boldsymbol{\omega}_0 \times (\boldsymbol{x}_n - \boldsymbol{x}_0) - 2\boldsymbol{\omega}_0 \times (\dot{\boldsymbol{x}}_n - \dot{\boldsymbol{x}}_0)) \tag{D.6}$$

where subscript 0 refers to the reference node, subscript n refers to the generic n-th node, and the prime refers to the relative values. Accelerations are only computed when present in the `.mov` file for both nodes.

When `RefOnly` is set to 1, the script simply re-orients the velocity and the acceleration of each node with respect to that of the reference node, namely

$$\mathbf{R}'_n = \mathbf{R}_0^T \mathbf{R}_n \tag{D.7}$$
$$\boldsymbol{x}'_n = \mathbf{R}_0^T (\boldsymbol{x}_n - \boldsymbol{x}_0) \tag{D.8}$$
$$\boldsymbol{\omega}'_n = \mathbf{R}_0^T \boldsymbol{\omega}_n \tag{D.9}$$
$$\dot{\boldsymbol{x}}'_n = \mathbf{R}_0^T \dot{\boldsymbol{x}}_n \tag{D.10}$$
$$\dot{\boldsymbol{\omega}}'_n = \mathbf{R}_0^T \dot{\boldsymbol{\omega}}_n \tag{D.11}$$
$$\ddot{\boldsymbol{x}}'_n = \mathbf{R}_0^T \ddot{\boldsymbol{x}}_n \tag{D.12}$$

Appendix E

Log File Format

MBDyn by default writes a file with `.log` extension that contains generic, one time info about the analysis and the model.

This file is experimental, since it is mainly intended to be used by external software, like post-processing tools, to extract or somehow infer information about the model, without the complexity of parsing the entire input.

It is used, for example, by the script `mbdyn2easyanim.sh`, along with the corresponding `.mov` output file, to generate model files for `EasyAnimm`, as illustrated in Section D.1.

E.1 Generic Format

The format is typically made of a keyword, starting in the first column, made of alphanumeric chars (may include spaces, though), terminated by a colon, and possibly followed by arbitrary, context-dependent data. Data may span multiple rows, where a continuation row is marked by a blank in the first column.

E.2 Model Description Entries

Some of the keywords mark model entities, described in the following.

E.2.1 acceleration, velocity

The `angular acceleration`, the `linear acceleration` the `angular velocity` and the `linear velocity` joints.

```
<joint_type>: <label>
    <node_label> (Vec3) <relative_direction>
<joint_type> ::=
        { angularacceleration | linearacceleration | angularvelocity |
    linearvelocity }
```

The `relative direction` vectors indicate the (not normalized) axis of the imposed acceleration/velocity. All data are on one line, without continuation.

E.2.2 aero0

The `aerodynamic body`.

```
aero0: <label> <node_label>
    (Vec3) <trail_left> (Vec3) <lead_left>
    (Vec3) <trail_right> (Vec3) <lead_right>
```

The points are expressed as 3D vectors Vec3, whose origin and orientation is expressed in the reference frame of the node. All data are on one line, without continuation.

E.2.3 aero2

The two-node aerodynamic beam.

```
aero2: <label>
    <node_1_label>
    (Vec3) <trail_left> (Vec3) <lead_left>
    <node_2_label>
    (Vec3) <trail_right> (Vec3) <lead_right>
```

The points are expressed as 3D vectors Vec3, whose origin and orientation is expressed in the reference frame of the respective nodes. All data are on one line, without continuation.

E.2.4 aero3

The three-node aerodynamic beam.

```
aero3: <label>
    <node_1_label>
    (Vec3) <trail_left> (Vec3) <lead_left>
    <node_2_label>
    (Vec3) <trail_center> (Vec3) <lead_center>
    <node_3_label>
    (Vec3) <trail_right> (Vec3) <lead_right>
```

The points are expressed as 3D vectors Vec3, whose origin and orientation is expressed in the reference frame of the respective nodes. All data are on one line, without continuation.

E.2.5 axial rotation

The axialrotation joint

```
axialrotation: <label>
    <node_1_label> (Vec3) <position_1> (Mat3x3) <orientation_1>
    <node_2_label> (Vec3) <position_2> (Mat3x3) <orientation_2>
```

The position_* vectors indicates the location of the joint in the reference frame of the respective node, while the orientation_* matrices indicate the orientation of the joint in the reference frame of the respective node. All data are on one line, without continuation.

E.2.6 beam2

The two-node beam.

```
beam2: <label>
    <node_1_label> (Vec3) <offset_1>
    <node_2_label> (Vec3) <offset_2>
```

The beam label is followed by the label and the offset of each node. All data are on one line, without continuation.

E.2.7 beam3

The three-node beam.

```
beam3: <label>
    <node_1_label> (Vec3) <offset_1>
    <node_2_label> (Vec3) <offset_2>
    <node_3_label> (Vec3) <offset_3>
```

The beam label is followed by the label and the offset of each node. All data are on one line, without continuation.

E.2.8 body

The body element.

```
body: <label>
    <node_label>
    (real) <mass>
    (Vec3) <x_cg>
    (Mat3x3) <J>
```

The x_cg vector indicates the location of the center of mass with respect to the reference frame of the respective node. The J matrix is the inertia matrix.

E.2.9 beam slider

The beam slider joint.

```
<beamslider>: <label>
        <slider_node_label>
    (Vec3) <relative_offset>
    (Mat3x3) <relative_orientation>
    <beam_number>
    <3_node_beam_1>
            (Vec3) <first_node_offset>
            (Mat3x3) <first_node_orientation>
            (Vec3) <mid_node_offset>
            (Mat3x3) <mid_node_orientation>
            (Vec3) <end_node_offset>
    (Mat3x3) <end_node_orientation>
    [...]
```

All data on one line, without continuation.

E.2.10 brake

The brake joint

```
brake: <label>
    <node_1_label> (Vec3) <position_1> (Mat3x3) <orientation_1>
    <node_2_label> (Vec3) <position_2> (Mat3x3) <orientation_2>
```

The position_* vectors indicates the location of the joint in the reference frame of the respective node, while the orientation_* matrices indicate the orientation of the joint in the reference frame of the respective node. All data are on one line, without continuation.

E.2.11 clamp

The clamp joint

```
clamp: <label>
    <node_label> (Vec3) <position> (Mat3x3) <orientation>
    <node_label> (Vec3) <position> (Mat3x3) <orientation>
```

The format is quite obscure; the position and the orientation are repeated twice. Moreover, the position is always a vector of zeros, and the orientation is the identity matrix. Basically, the location of the clamp is assumed to be that of the node. All data are on one line, without continuation.

E.2.12 deformable joints

The deformable hinge, the deformable displacement joint and the deformable joint, including the invariant versions, where defined

```
<joint_type>: <label>
    <node_1_label> (Vec3) <position_1> (Mat3x3) <orientation_1>
    <node_2_label> (Vec3) <position_2> (Mat3x3) <orientation_2>

<joint_type> ::=
    [ invariant ] deformable { hinge | [ displacement ] joint }
```

The position_* vectors indicates the location of the joint in the reference frame of the respective node, while the orientation_* matrices indicate the orientation of the joint in the reference frame of the respective node. All data are on one line, without continuation.

E.2.13 distance

The distance joint

```
distance: <label>
    <node_1_label> (Vec3) <offset_1>
    <node_2_label> (Vec3) <offset_2>
```

The label of the joint, followed by the label of each node and the offset of the respective joint extremity, in the reference frame of the node. All data are on one line, without continuation. Both distance and distance with offset joints are logged in this form.

E.2.14 drive displacement

The drive displacement joint

```
drivedisplacement: <label>
    <node_1_label> (Vec3) <offset_1>
    <node_2_label> (Vec3) <offset_2>
```

All data in one line, without continuation.

E.2.15 drive displacement pin

The `drive displacement pin` joint

```
drivedisplacementpin: <label>
    <node_1_label> (Vec3) <offset_1>
    (Vec3) <position>
```

All data in one line, without continuation.

E.2.16 gimbal rotation

The `gimbal rotation` joint

```
gimbalrotation: <label>
    <node_1_label> (Vec3) <position_1> (Mat3x3) <orientation_1>
    <node_2_label> (Vec3) <position_2> (Mat3x3) <orientation_2>
```

The `position_*` vectors indicates the location of the joint in the reference frame of the respective node, while the `orientation_*` matrices indicate the orientation of the joint in the reference frame of the respective node. All data are on one line, without continuation.

E.2.17 inline

The `inline` joint

```
inline: <label>
    <node_1_label> (Vec3)<position_1> (Mat3x3) <orientation_1>
    <node_2_label> (Vec3)<position_2> (Mat3x3) <orientation_2>
```

The `label` of the joint, followed by the label of the node that carries the reference line, the reference point `position_1` on the line and the orientation `orientation_1` of the line, such that axis 3 is aligned with the line. The second node label and the position of the point on the plane follow; `orientation_2` is the identity matrix. All data are on one line, without continuation.

E.2.18 inplane

The `inplane` joint

```
inplane: <label>
    <node_1_label> (Vec3) <position_1> (Mat3x3) <orientation_1>
    <node_2_label> (Vec3) <position_2> (Mat3x3) <orientation_2>
```

The `label` of the joint, followed by the label of the node that carries the reference plane, the reference point `position_1` on the plane and the orientation `orientation_1` of the plane, such that axis 3 is normal to the plane. The second node label and the position of the point on the plane follow; `orientation_2` is the identity matrix. All data are on one line, without continuation.

E.2.19 prismatic

The prismatic joint

```
prismatic: <label>
    <node_1_label> (Vec3) <position_1> (Mat3x3) <orientation_1>
    <node_2_label> (Vec3) <position_2> (Mat3x3) <orientation_2>
```

The position_* vectors are zero, while the orientation_* matrices indicate the orientation of the joint in the reference frame of the respective nodes. All data are on one line, without continuation.

E.2.20 relative frame structural node

The relative frame variant of dummy structural nodes is logged separately, as it cannot be intermixed with regular structural nodes and the offset variant of dummy structural nodes. The syntax is the same of the structural node (Section E.2.27), namely

```
relative frame structural node: <label> (Vec3) <X> <orientation_description>
```

E.2.21 revolute hinge

The revolute hinge joint

```
revolutehinge: <label>
    <node_1_label> (Vec3) <position_1> (Mat3x3) <orientation_1>
    <node_2_label> (Vec3) <position_2> (Mat3x3) <orientation_2>
```

The position_* vectors indicates the location of the joint in the reference frame of the respective node, while the orientation_* matrices indicate the orientation of the joint in the reference frame of the respective node. All data are on one line, without continuation.

E.2.22 revolute rotation

The revolute rotation joint

```
revoluterotation: <label>
    <node_1_label> (Vec3) <position_1> (Mat3x3) <orientation_1>
    <node_2_label> (Vec3) <position_2> (Mat3x3) <orientation_2>
```

The position_* vectors indicates the location of the joint in the reference frame of the respective node, while the orientation_* matrices indicate the orientation of the joint in the reference frame of the respective node. All data are on one line, without continuation.

E.2.23 rod

The rod joint

```
rod: <label>
    <node_1_label> (Vec3) <offset_1>
    <node_2_label> (Vec3) <offset_2>
```

The label of the joint, followed by the label of each node and the offset of the respective joint extremity, in the reference frame of the node. All data are on one line, without continuation. Both rod and rod with offset joints are logged in this form.

E.2.24 spherical hinge

The spherical hinge joint

```
sphericalhinge: <label>
    <node_1_label> (Vec3) <position_1> (Mat3x3) <orientation_1>
    <node_2_label> (Vec3) <position_2> (Mat3x3) <orientation_2>
```

The position_* vectors indicates the location of the joint in the reference frame of the respective node, while the orientation_* matrices indicate the orientation of the joint in the reference frame of the respective node. All data are on one line, without continuation.

E.2.25 spherical pin

The spherical pin joint

```
sphericalpin: <label>
    <node_label> (Vec3) <position> (Mat3x3) <orientation>
    <node_label> (Vec3) <position> (Mat3x3) <orientation>
```

The format is quite obscure; the position and the orientation are repeated twice. The vector position indicates the location of the joint, while the matrix orientation is the identity matrix. All data are on one line, without continuation.

E.2.26 structural forces

The structural forces elements

```
<force_type>: <force_label>
    <node1_label> (Vec3)<arm1>
    [ <node2_label> (Vec3)<arm2> ]

<force_type> ::= structural [ internal ] { absolute | follower } { force | couple }
```

When internal is present node2_label and arm2 must be present as well, otherwise they must be omitted.

E.2.27 structural node

The structural node

```
structural node: <label> (Vec3) <X> <orientation_description>

<orientation_description> ::=
    { euler123 (Vec3) <euler123_angles>
        | euler313 (Vec3) <euler313_angles>
        | euler321 (Vec3) <euler321_angles>
        | phi (Vec3) <orientation_vector>
        | mat (Mat3x3) <orientation_matrix> }
```

The label of the node, the position X and the orientation parameters that express the orientation are given. When orientation is any of euler* parametrizations, the angles are in degrees.

E.2.28 total joint

The `total` joint

```
total joint: <label>
    <node_1_label>
    (Vec3) <relative_offset_1>
    (Mat3x3) <rel_pos_orientation_1>
    (Mat3x3) <rel_rot_orientation_1>
    <node_2_label>
    (Vec3) <relative_offset_2>
    (Mat3x3) <rel_pos_orientation_2>
    (Mat3x3) <rel_rot_orientation_2>
    (bool) <position_status_1>
    (bool) <position_status_2>
    (bool) <position_status_3>
    (bool) <velocity_status_1>
    (bool) <velocity_status_2>
    (bool) <velocity_status_3>
    (bool) <orientation_status_1>
    (bool) <orientation_status_2>
    (bool) <orientation_status_3>
    (bool) <angular_velocity_status_1>
    (bool) <angular_velocity_status_2>
    (bool) <angular_velocity_status_3>
```

All data on one line, without continuation.

E.2.29 total pin joint

The `total pin` joint

```
total joint: <label>
    <node_label>
    (Vec3) <relative_offset>
    (Mat3x3) <rel_pos_orientation>
    (Mat3x3) <rel_rot_orientation>
    (Vec3) <absolute_position>
    (Mat3x3) <abs_pos_orientation>
    (Mat3x3) <abs_rot_orientation>
    (bool) <position_status_1>
    (bool) <position_status_2>
    (bool) <position_status_3>
    (bool) <velocity_status_1>
    (bool) <velocity_status_2>
    (bool) <velocity_status_3>
    (bool) <orientation_status_1>
    (bool) <orientation_status_2>
    (bool) <orientation_status_3>
    (bool) <angular_velocity_status_1>
    (bool) <angular_velocity_status_2>
    (bool) <angular_velocity_status_3>
```

All data on one line, without continuation.

E.2.30 universal hinge

The `universal hinge` joint

```
universalhinge: <label>
    <node_1_label> (Vec3) <position_1> (Mat3x3) <orientation_1>
    <node_2_label> (Vec3) <position_2> (Mat3x3) <orientation_2>
```

The `position_*` vectors indicates the location of the joint in the reference frame of the respective node, while the `orientation_*` matrices indicate the orientation of the joint in the reference frame of the respective node. All data are on one line, without continuation.

E.2.31 universal pin

The `universal pin` joint

```
universalpin: <label>
    <node_label> (Vec3) <position> (Mat3x3) <orientation>
    <node_label> (Vec3) <relative_position> (Mat3x3) <relative_orientation>
```

The `position` vector and the `orientation` matrix indicate the location and the orientation of the joint in the global reference frame, while the `relative_position` vector and the `relative_orientation` matrix indicate the location and the orientation of the joint in the reference frame of the node. All data are on one line, without continuation.

E.2.32 universal rotation

The `universal rotation` joint

```
universalrotation: <label>
    <node_1_label> (Vec3) <position_1> (Mat3x3) <orientation_1>
    <node_2_label> (Vec3) <position_2> (Mat3x3) <orientation_2>
```

The `position_*` vectors are zero, while the `orientation_*` matrices indicate the orientation of the joint in the reference frame of the respective node. All data are on one line, without continuation.

E.3 Analysis Description Entries

Some of the keywords mark analysis entities, described in the following.

E.3.1 inertia

An aggregate inertia.

```
inertia: <label> (<name>)
    mass:       (real) <mass>
    J:          (Mat3x3) <J>
    Xcg:        (Vec3) <x_cg>
    Jcg:        (Mat3x3) <J_cg>
    Vcg:        (Vec3) <v_cg>
```

```
Wcg:          (Vec3) <omega_cg>
Xcg-X:        (Vec3) <relative_x_cg>
R^T*(Xcg-X):  (Vec3) <relative_orientation_x_cg>
Rp:           (Vec3) <principal_axes_orientation_matrix>
Thetap:       (Vec3) <principal_axes_orientation_vector>
Jp:           (Mat3x3) <principal_inertia_matrix_cg>
```

E.3.2 output frequency

An indication of how often output occurs with respect to time steps.

`output frequency`: <how_often>

where how_often can be:

- `custom` if the output is specially crafted by the `output meter` statement (Section 5.3.9);
- a positive integer if the output occurs every how_often time steps, as dictated by the `output frequency` statement (Section 5.3.8).

E.3.3 struct node dofs

The list of structural nodes (excluding dummy nodes).

`struct node dofs`: <idx> [...]

where idx is the off-by-one index of the location of the first state associated with a node (for example, $X(\texttt{idx} + i)$, with $i = 1, 2, 3$, is the component of the position along axis i).

E.3.4 struct node momentum dofs

The list of structural nodes (excluding static and dummy nodes).

`struct node momentum dofs`: <idx> [...]

where idx is the off-by-one index of the location of the first state associated with a node's momentum and momenta moment (for example, $X(\texttt{idx} + i)$, with $i = 1, 2, 3$, is the component of the position along axis i).

E.3.5 struct node labels

The list of structural nodes labels (excluding dummy nodes).

`struct node label`: <label> [...]

E.4 Output Elements

E.4.1 RTAI stream output

The `stream output` element in case an RTAI mailbox is used:

```
outputelement: <label>
        stream
        RTAI
        (string) <stream_name>
        (string) <host>
        (string) <name>
        (bool) <create>
        { values | motion }
        <content>
```

The `create` flag can take 1 or 0 value. The `content` formatting is specified in section E.4.4. All data in one line, without continuation.

E.4.2 INET socket stream output

the `stream output` element in case an INET socket is used:

```
outputelement: <label>
        stream
        INET
        (string) <stream_name>
        (string) <host>
        (integer) <port>
        { tcp | udp }
        (bool) <create>
        (bool) <signal>
        (bool) <blocking>
        (bool) <send_first>
        (bool) <abort_if_broken>
        (integer) <output_frequency>
        { values | motion }
        <content>
```

The `content` formatting is specified in section E.4.4. All data in one line, without continuation.

E.4.3 local socket stream output

the `stream output` element in case an local socket is used:

```
outputelement: <label>
        stream
        UNIX
        (string) <path>
        { tcp | udp }
        (string) <stream_name>
        (bool) <create>
        (bool) <signal>
        (bool) <blocking>
        (bool) <send_first>
        (bool) <abort_if_broken>
        (integer) <output_frequency>
```

```
        { values | motion }
        <content>
```

The content formatting is specified in section E.4.4. All data in one line, without continuation.

E.4.4 output element content

The content of a stream output element. In case of output of type values

```
  <content>::
          <values> (integer) <num_channels>
```

While in case of output of type motion

```
  <content>::
          <motion>
        (integer) <num_nodes>
        (bool) <position_output_flag>
        (bool) <orientation_matrix_output_flag>
        (bool) <orientation_matrix_transpose_output_flag>
        (bool) <velocity_output_flag>
        (bool) <angular_velocity_output_flag>
```

All data in one line, without continuation.

E.5 File drivers

E.5.1 INET socket stream file driver

The stream file driver in case an INET socket is used:

```
filedriver:
        <label>
        stream
        INET
        (string) <stream_name>
        (string) <host>
        (integer) <port>
        { tcp | udp }
        (bool) <create>
        (bool) <signal>
        (bool) <blocking>
        (integer) <input_frequency>
        (bool) <receive_first>
        (real) <timeout>
        (integer) <columns_number>
        (real) <initial_value_1>
        (real) <initial_value_2>
        [...]
```

Appendix F

Frequently Asked Questions

This section contains answers to frequently asked questions related to input and output formatting, and execution control. It is an attempt to fill the knowledge gap between the user's needs and the description of each input statement. In fact, this manual assumes that the user already knows what statement is required to solve a given problem or to model a given system, so that all it is missing is the details about the syntax. This is typically not true, especially for inexperienced users, whose main problem consists in understanding how to solve a problem, or model a system. Of course an input manual is not the right document to start with but, better than nothing, this section is trying to fill this gap, allowing to approach the manual "the other way round".

If one's question is listed below, then it's likely that the answer will point to reading the right sections of the manual.

F.1 Input

F.1.1 What is the exact syntax of element *X*?

The exact syntax of each input card is illustrated in the input manual.

The input manual is regularly updated, but omissions may occur, and outdated stuff and bugs may always slip in. Please feel free to notify errors and submit patches, if you think there is anything wrong in it, using the `mbdyn-users@mbdyn.org` mailing list (you need to subscribe first; instructions can be obtained on the website `https://www.mbdyn.org/Mailing.html`).

F.1.2 What element should I use to model *Y*?

To answer this question, the complement of the input manual, namely a modeling manual, is required. Unfortunately, such document does not exist, and it is not even foreseen. Right now, modeling style and capabilities are grossly addressed in the tutorials; for specific needs you should ask on the `mbdyn-users@mbdyn.org` mailing list (you need to subscribe first; instructions can be obtained on the web site `https://www.mbdyn.org/Mailing.html`).

F.2 Output

F.2.1 How can I reduce the amount of output?

There are different possibilities: one can only output a portion of the results, or produce output only at selected time steps, or both.

To selectively produce output there are different means:

- **by category:** use the `default output` statement (Section 5.3.6) to indicate what entities should be output. By default, all entities output data, so the suggested use is:

  ```
  default output: none, structural nodes, beams;
  ```

  so that the first value, `none`, turns all output off, while the following values re-enable the desired ones only.

- **entity by entity, directly:** all entities allow an output specification as the last argument (nodes: Section 6; elements: Section 8); so, at the end of an entity's card just put

  ```
  ..., output, no;    # to disable output
  ..., output, yes;   # to enable output
  ```

- **entity by entity, deferred:** all entities allow to re-enable output after they have been instantiated by using the `output` statement (nodes: Section 6.7.1; elements: Section 8.20.4).

So a typical strategy is:

- use `default output` to select those types that must be on or off; also disable the types of those entities that must be partially on and partially off;
- use the direct option to respectively set the output of those entities that must be absolutely on or off; this overrides the `default output` value.
- use the `output` statement to enable the output of those entities that must be on for that simulation only; this overrides both the `default output` and the direct values.

The direct option goes in the model, so it may be hidden in an include file of a component that is used in different models; its value should not be modified directly. On the contrary, the `output` statement can appear anywhere inside the node/element blocks, so the same subcomponents can have different output behavior by using different `output` statements.

Example.

```
# ...
begin: control data;
    default output: none, joints;
    structural nodes: 2;
    joints: 2;
end: control data;
begin: nodes;
    structural: 1, static,
        null, eye, null, null,
        output, no;  # this is the ground; never output it
```

```
    structural: 2, dynamic,
        null, eye, null, null;
        # defer decision about output

    # uncomment to output
    # output: structural, 2;
end: nodes;
begin: elements;
    joint: 1, clamp, 1, node, node,
        output, no;  # this is the ground; never output it
    joint: 2, revolute hinge,
        1, null,
            hinge, eye,
        2, null,
            hinge, eye;
         # this is output by "default output"

    # make sure that no matter what "default output"
    # is set, this joint is output
    output: joint, 2;
end: elements;
```

To reduce output in time, one can use the `output meter` statement (Section 5.3.9). It consists in a drive that causes output to be produced only when its value is not 0.

For example, to produce output only after a certain time, use

```
output meter: string, "Time > 10.";
```

to produce output every 5 time steps starting from 0., use

```
output meter: meter, 0., forever, steps, 5;
```

to produce output only when a certain variable reaches a certain value, use

```
output meter: string, "Omega > 99.";
```

if `Omega` is the third component of the angular velocity of a certain node (say, node 1000), later on, after the node is defined, the related definition must appear:

```
set: "[dof,Omega,1000,structural,6,differential]
```

i.e. the sixth component of the derivative state of node 1000, the angular velocity about axis 3, is assigned to the variable `Omega`. Any time `Omega` is evaluated in a string expression, its value gets updated.

F.3 Execution Debugging

F.3.1 How can I find out why the iterative solution of a nonlinear problem does not converge?

One needs to:

1. *find out the equations whose residual does not converge to zero.*

   The residual is printed using the statement

```
output: residual;
```

in the problem control block described in Section 4.1.4 of Chapter 4. Each coefficient is preceded by its number and followed by a brief description of which node/element instantiated the equation and what its purpose is, if available; for example, the output

```
Eq     1:        0 ModalStructNode(1): linear velocity definition vx
Eq     2:        0 ModalStructNode(1): linear velocity definition vy
Eq     3:        0 ModalStructNode(1): linear velocity definition vz
Eq     4:        0 ModalStructNode(1): angular velocity definition wx
Eq     5:        0 ModalStructNode(1): angular velocity definition wy
Eq     6:        0 ModalStructNode(1): angular velocity definition wz
Eq     7:        0 ModalStructNode(1): force equilibrium Fx
Eq     8:        0 ModalStructNode(1): force equilibrium Fy
Eq     9:        0 ModalStructNode(1): force equilibrium Fz
Eq    10:        0 ModalStructNode(1): moment equilibrium Mx
Eq    11:        0 ModalStructNode(1): moment equilibrium My
Eq    12:        0 ModalStructNode(1): moment equilibrium Mz
```

corresponds to a residual whose first 12 equations refer to a modal node labeled "1" (and the residual is exactly zero);

2. *find out who instantiated the offending equation or equations.*

   If this is not already indicated in the previously mentioned description, one should look up the offending equation index in the output originated by adding the statement

   ```
   print: equation description;
   ```

   in the `control data` block, as described in Section 5.3.2. In the above example, it would generate

   ```
   Structural(1): 12 1->12
           1->3: linear velocity definition [vx,vy,vz]
           4->6: angular velocity definition [wx,wy,wz]
           7->9: force equilibrium [Fx,Fy,Fz]
           10->12: moment equilibrium [Mx,My,Mz]
   ```

3. *find out who could contribute to that equation or equations.*

   If the equation was instantiated by a node, one should look at the elements connected to that node. This information is obtained by adding the statement

   ```
   print: node connection;
   ```

   in the `control data` block, as described in Section 5.3.2. In the above example, it would generate

   ```
   Structural(1) connected to
           Joint(1)
           Joint(2)
   ```

   so one should try to find out which of the connected elements is generating the offending contribution to that equation.

   If the equation was instantiated by an element, usually the element itself is the sole contributor to that equation.

Bibliography

[1] David J. Laino and A. Craig Hansen. User's guide to the wind turbine aerodynamics computer software AeroDyn. Technical report, NREL, 2002. Version 12.50.

[2] Pierre Dupont, Vincent Hayward, Brian Armstrong, and Friedhelm Altpeter. Single state elastoplastic friction models. *IEEE Transactions on Automatic Control*, June 2002.

[3] Gabriel Arslan Waltersson and Yiannis Karayiannidis. Planar friction modeling with lugre dynamics and limit surfaces. *IEEE Transactions on Robotics*, 40:3166–3180, 2024.

[4] John C. Houbolt and George W. Brooks. Differential equations of motion for combined flapwise bending, chordwise bending, and torsion of twisted nonuniform rotor blades. TN 3905, NACA, 1957. Updated in 1975.

[5] Marco Borri and Teodoro Merlini. A large displacement formulation for anisotropic beam analysis. *Meccanica*, 21:30–37, 1986.

[6] Klaus-Jürgen Bathe. *Finite Element Procedures*. K.J. Bathe, Watertown, MA, 2016.

[7] Richard Schwertassek and Oskar Wallrapp. *Dynamik flexibler Mehrkörpersysteme*. Friedrich Vieweg und Sohn, Braunschweig/Wiesbaden, 1998.

[8] Lars Kübler. *Simulation und Sensitivitätsanalyse flexibler Mehrkörpersysteme mit großen Deformationen*. 2005.

[9] Jeremy Bleyer. *Numerical Tours of Computational Mechanics with FEniCS*, 2018.

[10] Thomas Helfer, Bruno Michel, Jean-Michel Proix, Maxime Salvo, Jérôme Sercombe, and Michel Casella. Introducing the open-source mfront code generator: Application to mechanical behaviours and material knowledge management within the PLEIADES fuel element modelling platform. 70(5):994–1023.

[11] Pierangelo Masarati, Massimiliano Lanz, and Paolo Mantegazza. Multistep integration of ordinary, stiff and differential-algebraic problems for multibody dynamics applications. In *XVI Congresso Nazionale AIDAA*, pages 71.1–10, Palermo, 24–28 Settembre 2001.

[12] Huimin Zhang, Runsen Zhang, and Pierangelo Masarati. Improved second-order unconditionally stable schemes of linear multi-step and equivalent single-step integration methods. *Computational Mechanics*, 67(1):289–313, 2021. doi:10.1007/s00466-020-01933-y.

[13] Huimin Zhang, Runsen Zhang, Yufeng Xing, and Pierangelo Masarati. On the optimization of n-sub-step composite time integration methods. *Nonlinear Dynamics*, 102(3):1939–1962, November 2020.

[14] Huimin Zhang, Runsen Zhang, Andrea Zanoni, and Pierangelo Masarati. Performance of implicit A-stable time integration methods for multibody system dynamics. *Multibody System Dynamics*, 54(3):263–301, March 2022.

[15] Christopher A. Kennedy and Mark H. Carpenter. Diagonally implicit Runge–Kutta methods for stiff ODEs. *Applied Numerical Mathematics*, 146:221–244, December 2019.

[16] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992.

[17] Vincent Acary, Maurice Bremond, Olivier Huber, Franck Perignon, and Roger Pissard-Gibollet. *Siconos: nonsmooth numerical simulation*, 2022 (acccessed July 16, 2022).

[18] The Trilinos Project Team. *The Trilinos Project Website*, 2020 (acccessed May 22, 2020).

[19] C.T. Kelley. *Iterative methods for linear and nonlinear equations*. SIAM, Philadelphia, PA, 1995.

[20] P. Masarati. Direct eigenanalysis of constrained system dynamics. *Proc. IMech. E Part K: J. Multi-body Dynamics*, 223(4):335–342, 2009. doi:10.1243/14644193JMBD211.

[21] Marco Morandini and Paolo Mantegazza. Using dense storage to solve small sparse linear systems. *ACM Trans. Math. Softw.*, 33(1, Article 5), March 2007. doi:10.1145/1206040.1206045.

[22] Olaf Schenk and Klaus Gärtner. Solving unsymmetric sparse systems of linear equations with pardiso. *Future Generation Computer Systems*, 20(3):475–487, 2004. Selected numerical algorithms.

[23] Anshul Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Trans. Math. Softw.*, 28(3):301–324, sep 2002.

[24] Pascal Hénon, Pierre Ramet, and Jean Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, 2002.

[25] Timothy A. Davis. Suitesparseqr: Multifrontal multithreaded rank-revealing sparse qr factorization. *ACM Transactions on Mathematical Software*, 2011.

[26] R. Sinkhorn and P. Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics, 21(2)*, 1967.

[27] Amestoy P.R., Duff I.S., Ruiz D., and Ucar B. A parallel matrix scaling algorithm. *High Performance Computing for Computational Science - VECPAR 2008*, 2008.

[28] G. Pasinetti and Paolo Mantegazza. A single finite states modeling of the unsteady aerodynamic forces related to structural motions and gusts. *AIAA Journal*, 37(5):604–612, May 1999.

[29] I.C. Cheeseman and N.E. Bennett. The effect of ground on a helicopter rotor in forward flight. TR 3021, NASA, 1955.

[30] Dale M. Pitt and David A. Peters. Theoretical prediction of dynamic-inflow derivatives. *Vertica*, 5:21–34, 1981.

[31] Gian Luca Ghiringhelli, Pierangelo Masarati, and Paolo Mantegazza. A multi-body implementation of finite volume beams. *AIAA Journal*, 38(1):131–138, January 2000.

[32] Dewey H. Hodges. Review of composite rotor blade modeling. *AIAA Journal*, 28(3):561–565, March 1990.

[33] Vittorio Giavotto, Marco Borri, Paolo Mantegazza, Gian Luca Ghiringhelli, V. Caramaschi, G. C. Maffioli, and F. Mussi. Anisotropic beam theory and applications. *Computers & Structures*, 16(1–4):403–413, 1983.

[34] Pierangelo Masarati. On the semi-analytical solution of beams subjected to distributed loads. In *XVI Congresso Nazionale AIDAA*, pages 72.1–10, Palermo, 24–28 Settembre 2001.

[35] David A. Peters, Mnaouar ChouChane, and Mark Fulton. Helicopter trim with flap-lag-torsion and stall by an optimized controller. *J. of Guidance, Control, and Dynamics*, 13(5):824–834, September-October 1990.

[36] Stefania Gualdi, Marco Morandini, and Pierangelo Masarati. A deformable slider joint for multibody applications. In *XVII Congresso Nazionale AIDAA*, Rome, Italy, September 15–19 2003. https://home.aero.polimi.it/masarati/Publications/SliderAIDAA2003.pdf.

[37] Pierangelo Masarati and Marco Morandini. An ideal homokinetic joint formulation for general-purpose multibody real-time simulation. *Multibody System Dynamics*, 20(3):251–270, October 2008. doi:10.1007/s11044-008-9112-8.

[38] A. Fumagalli, P. Masarati, M. Morandini, and P. Mantegazza. Control constraint realization for multibody systems. *J. of Computational and Nonlinear Dynamics*, 6(1):011002 (8 pages), January 2011. doi:10.1115/1.4002087.

[39] Pierangelo Masarati. A formulation of kinematic constraints imposed by kinematic pairs using relative pose in vector form. *Multibody System Dynamics*, 29(2):119–137, 2013. doi:10.1007/s11044-012-9320-0.

[40] Wojciech Witkowski. 4-node combined shell element with semi-EAS-ANS strain interpolations in 6-parameter shell theories with drilling degrees of freedom. *Computational Mechanics*, 43(2):307–319, 2009. doi:10.1007/s00466-008-0307-x.

[41] Guido Dhondt. *The Finite Element Method for Three-Dimensional Thermomechanical Applications*. Wiley, 2004.

[42] EDF R&D. *Code_Aster - Shape functions and integration points of finite elements*. Electricite de France, 2001.

[43] Dirk Bartel. *Simulation von Tribosystemen*. Vieweg+Teubner Research, Wiesbaden, 2010.

[44] J. H. Tripp J. A. Greenwood. The contact of two nominal flat rough surfaces. 1970.

[45] Efstathios Velenis et al. Dynamic tire friction models for combined longitudinal and lateral vehicle motion. 2003.

[46] P. Flores, M. Machado, M. T. Silva, and J. M. Martins. On the continuous contact force models for soft materials in multibody dynamics. *Multibody System Dynamics*, 25(3):357–375, March 2011. doi:10.1007/s11044-010-9237-4.

[47] K. H. Hunt and F. R. E. Crossley. Coefficient of restitution interpreted as damping in vibroimpact. *Journal of Applied Mechanics, Transactions ASME*, 42(2):440–445, 1975. doi:10.1115/1.3423596.

[48] Hamid M. Lankarani and Parviz E. Nikravesh. Continuous contact force models for impact analysis in multibody systems. *Nonlinear Dynamics*, 5(2):193–207, 1994. doi:10.1007/BF00045676.

[49] E. Pennestrì, R. Stefanelli, P. P. Valentini, and L. Vita. Virtual musculo-skeletal model for the biomechanical analysis of the upper limb. *Journal of Biomechanics*, 40(6):1350–1361, 2007. doi:10.1016/j.jbiomech.2006.05.013.

[50] Taylor M. Winters, Mitsuhiko Takahashi, Richard L. Lieber, and Samuel R. Ward. Whole muscle length-tension relationships are accurately modeled as scaled sarcomeres in rabbit hindlimb muscles. *Journal of Biomechanics*, 44(1):109–115, January 2011.

[51] Friedl De Groote, Jessica L. Allen, and Lena H. Ting. Contribution of muscle short-range stiffness to initial changes in joint kinetics and kinematics during perturbations to standing balance: A simulation study. *Journal of Biomechanics*, 55:71–77, April 2017.

[52] *The NetCDF Users Guide*, June 2006. NetCDF Version 3.6.1.