COMSOL MULTIPHYSICS®

REFERENCE GUIDE

VERSION 3.5a

COMSOL

How to contact COMSOL:

Benelux
COMSOL BV
Röntgenlaan 19
2719 DX Zoetermeer
The Netherlands
Phone: +31 (0) 79 363 4230
Fax: +31 (0) 79 361 4212
info@comsol.nl
www.comsol.nl

Denmark
COMSOL A/S
Diplomvej 376
2800 Kgs. Lyngby
Phone: +45 88 70 82 00
Fax: +45 88 70 80 90
info@comsol.dk
www.comsol.dk

Finland
COMSOL OY
Arabianranta 6
FIN-00560 Helsinki
Phone: +358 9 2510 400
Fax: +358 9 2510 4010
info@comsol.fi
www.comsol.fi

France
COMSOL France
WTC, 5 pl. Robert Schuman
F-38000 Grenoble
Phone: +33 (0)4 76 46 49 01
Fax: +33 (0)4 76 46 07 42
info@comsol.fr
www.comsol.fr

Germany
COMSOL Multiphysics GmbH
Berliner Str. 4
D-37073 Göttingen
Phone: +49-551-99721-0
Fax: +49-551-99721-29
info@comsol.de
www.comsol.de

Italy
COMSOL S.r.l.
Via Vittorio Emanuele II, 22
25122 Brescia
Phone: +39-030-3793800
Fax: +39-030-3793899
info.it@comsol.com
www.it.comsol.com

Norway
COMSOL AS
Søndre gate 7
NO-7485 Trondheim
Phone: +47 73 84 24 00
Fax: +47 73 84 24 01
info@comsol.no
www.comsol.no

Sweden
COMSOL AB
Tegnérgatan 23
SE-111 40 Stockholm
Phone: +46 8 412 95 00
Fax: +46 8 412 95 10
info@comsol.se
www.comsol.se

Switzerland
FEMLAB GmbH
Technoparkstrasse 1
CH-8005 Zürich
Phone: +41 (0)44 445 2140
Fax: +41 (0)44 445 2141
info@femlab.ch
www.femlab.ch

United Kingdom
COMSOL Ltd.
UH Innovation Centre
College Lane
Hatfield
Hertfordshire AL10 9AB
Phone:+44-(0)-1707 636020
Fax: +44-(0)-1707 284746
info.uk@comsol.com
www.uk.comsol.com

United States
COMSOL, Inc.
1 New England Executive Park
Suite 350
Burlington, MA 01803
Phone: +1-781-273-3322
Fax: +1-781-273-6603

COMSOL, Inc.
10850 Wilshire Boulevard
Suite 800
Los Angeles, CA 90024
Phone: +1-310-441-4800
Fax: +1-310-441-0868

COMSOL, Inc.
744 Cowper Street
Palo Alto, CA 94301
Phone: +1-650-324-9935
Fax: +1-650-324-9936

info@comsol.com
www.comsol.com

For a complete list of international representatives, visit
www.comsol.com/contact

Company home page
www.comsol.com

COMSOL user forums
www.comsol.com/support/forums

Version: November 2008 COMSOL 3.5a

Part number: CM020005

CONTENTS

Chapter 1: Command Reference

Chapter 3: The Finite Element Method

Chapter 4: Advanced Geometry Topics

Chapter 5: Advanced Solver Topics

Chapter 6: The COMSOL Multiphysics Files

Command Reference

This chapter contains reference information for all COMSOL Multiphysics commands with descriptions of purpose, syntax, properties, and example scripts. With these commands, you can perform command-line modeling, calling routines for creating geometries, assembling, meshing, solving, postprocessing, and other tasks when using COMSOL Multiphysics with MATLAB.

Summary of Commands

Commands Grouped by Function

User Interface Functions

FUNCTION	PURPOSE
`comsol`	Start the COMSOL Multiphysics graphical user interface or a COMSOL Multiphysics server

Solver Functions

FUNCTION	PURPOSE
`adaption`	Solve PDE problem using adaptive mesh refinement
`femeig`	Solve eigenvalue PDE problem
`femlin`	Solve linear stationary PDE problem
`femnlin`	Solve nonlinear stationary PDE problem
`femoptim`[a]	Optimize stationary PDE problem
`femstatic`	Solve stationary PDE problem
`femtime`	Solve time-dependent PDE problem

a. Requires the Optimization Lab.

Geometry Functions

FUNCTION	PURPOSE
`chamfer`	Create flattened corners in 2D geometry object
`drawgetobj`	Get geometry object from draw structure
`drawsetobj`	Change geometry object in draw structure
`elevate`	Elevate degrees of 2D geometry object Bézier curves
`embed`	Embed 2D geometry object as 3D geometry object
`extrude`	Extrude 2D geometry object to 3D geometry object
`fillet`	Create circular rounded corners in 2D geometry object
`flcontour2mesh`	Create boundary mesh from contour data
`flim2curve`	Create 2D curve object from image data
`flmesh2spline`	Create spline curves from mesh
`geomanalyze`	Compose and analyze geometry of FEM problem
`geomarrayr`	Create rectangular array of geometry object
`geomcoerce`	Compose and coerce geometry objects
`geomcomp`	Compose (analyze) geometry objects
`geomcsg`	General function for analyzing geometry objects
`geomdel`	Delete interior boundaries
`geomedit`	Edit geometry object
`geomexport`	Export geometry object to file
`geomfile`	Geometry M-file
`geomgetwrkpln`	Retrieve work plane information
`geomgroup`	Group geometry objects into an assembly
`geomimport`	Import geometry object from file
`geominfo`	Retrieve geometry information.
`geomobject`	Create geometry object
`geomplot`	Plot a geometry object
`geomposition`	Position 3D geometry object
`geomspline`	Spline interpolation
`geomsurf`	Surface interpolation
`get` (p. 229)	Get geometry object properties
`getparts`	Extract parts from an assembly object
`loft`	Loft 2D geometry sections to 3D geometry

FUNCTION	PURPOSE
`mirror`	Reflect geometry
`move`	Move geometry object
`revolve`	Revolve 2D geometry object to 3D geometry object
`rotate`	Rotate geometry object
`scale`	Scale geometry object
`split`	Split geometry object
`tangent`	Create a tangent line

Geometry Objects

FUNCTION	PURPOSE
`arc1, arc2`	Elliptical or circular arc/solid sector
`block2, block3`	Rectangular block face/solid object
`circ1, circ2`	Circle curve/solid object
`cone2, cone3`	Cone face/solid object.
`curve2, curve3`	2D/3D rational Bézier curve object
`cylinder2, cylinder3`	Cylinder face/solid object
`econe2, econe3`	Eccentric cone face/solid object
`ellip1, ellip2`	Ellipse curve/solid object
`ellipsoid2, ellipsoid3`	Ellipsoid face/solid object
`face3`	3D rational Bézier surface object
`gencyl2, gencyl3`	Straight homogeneous generalized cylinder face/solid object
`geom0, geom1, geom2, geom3`	0D/1D/2D/3D geometry object
`helix1, helix2, helix3`	Helix curve/face/solid object
`hexahedron2, hexahedron3`	Hexahedron face/solid object
`line1, line2`	Open curve/solid polygon
`point1, point2, point3`	1D/2D/3D point object
`poly1, poly2`	Closed curve/solid polygon
`pyramid2, pyramid3`	Pyramid face/solid object
`rect1, rect2`	Rectangle curve/solid object
`solid0, solid1, solid2, solid3`	0D/1D/2D/3D solid object
`sphere3, sphere2`	Sphere solid/face object
`square1, square2`	Square curve/solid object

FUNCTION	PURPOSE
`tetrahedron2`, `tetrahedron3`	Tetrahedron face/solid object
`torus2`, `torus3`	Torus face/solid object

Mesh Functions

FUNCTION	PURPOSE
`femmesh`	Create a mesh object
`flcontour2mesh`	Create boundary mesh from contour data
`get` (p. 142)	Get mesh object properties
`mesh2geom`	Create geometry from (deformed) mesh
`meshbndlayer`	Create boundary layer mesh
`meshcaseadd`	Add new mesh cases
`meshcasedel`	Delete mesh cases
`meshconvert`	Convert mesh to simplex mesh
`meshcopy`	Copy mesh between boundaries
`meshdel`	Delete elements in a mesh
`meshembed`	Embed a 2D mesh into 3D
`meshenrich`	Make mesh object complete
`meshexport`	Export meshes to file
`meshextend`	Extend a mesh to the desired finite element types
`meshextrude`	Extrude a 2D mesh into a 3D mesh
`meshimport`	Import meshes from file
`meshinit`	Create free mesh
`meshmap`	Create mapped quad mesh
`meshplot`	Plot mesh
`meshqual`	Mesh quality measure
`meshrefine`	Refine a mesh
`meshrevolve`	Revolve a 2D mesh into a 3D mesh
`meshsmooth`	Jiggle internal points of a mesh
`meshsweep`	Create swept mesh
`xmeshinfo`	Get extended mesh information

Utility Functions

FUNCTION	PURPOSE
`assemble`	Assemble the stiffness matrix, right-hand side, mass matrix, and constraints of a PDE problem
`asseminit`	Compute initial value
`femdiff`	Symbolically differentiate general form
`femsim`	Create Simulink structure
`femstate`	Create state-space model for PDE problem
`femstruct`	FEM structure information
`femwave`	Extend FEM structure to a wave equation problem
`flcompact`	Compact `equ`/`bnd`/`edg`/`pnt` fields
`flform`	Convert between PDE forms
`flload`	Load a COMSOL Multiphysics file
`flngdof`	Get number of global degrees of freedom
`flnull`	Compute null space of a matrix, its complement, and the range of the matrix
`flreport`	Globally turn off progress window or show it
`flsave`	Save a COMSOL Multiphysics file
`multiphysics`	Multiphysics function
`solsize`	Get number of solutions in a solution object

Postprocessing Functions

FUNCTION	PURPOSE
`femplot`	Description of properties common to all plot functions
`meshintegrate`	Compute integrals in arbitrary cross sections
`postanim`	Shorthand command for animation
`postarrow`	Shorthand command for arrow plot in 2D and 3D
`postarrowbnd`	Shorthand command for boundary arrow plot in 2D and 3D
`postcolormap`	Return a MATLAB colormap for a COMSOL color table
`postcont`	Shorthand command for contour plot in 2D
`postcoord`	Get coordinates in a model
`postcrossplot`	Cross-section plot

FUNCTION	PURPOSE
`posteval`	Evaluate expressions on subdomains, boundaries, edges, and vertices
`postflow`	Shorthand command for streamline plot in 2D and 3D
`postglobaleval`	Evaluate globally defined expressions, such as solutions to ODEs
`postglobalplot`	Plotting globally defined expressions, such as solutions to ODEs
`postgp`	Extract Gauss points and Gauss point weights
`postint`	Integrate expression over subdomains, boundaries, edges, and vertices
`postinterp`	Evaluate expressions in arbitrary points
`postiso`	Shorthand command for isosurface plot in 3D
`postlin`	Shorthand command for line plot
`postmax`	Compute maximum value for expression
`postmin`	Compute minimum value for expression
`postmovie`	Postprocessing animation function
`postplot`	Postprocessing plot function
`postprinc`	Shorthand command for subdomain principal stress/strain plot in 2D and 3D
`postprincbnd`	Shorthand command for boundary principal stress/strain plot in 2D and 3D
`postslice`	Shorthand command for slice plot in 3D
`postsum`	Sum expressions over nodes
`postsurf`	Shorthand command for surface plot in 2D and 3D
`posttet`	Shorthand command for subdomain plot in 3D
`postwriteinterpfile`	Create interpolation file

Shape Function Classes

FUNCTION	PURPOSE
`sharg_2_5`	Fifth-order Argyris shape function object in 2D
`shbub`	Bubble shape function object
`shcurl`	Vector shape function object
`shdens`	Density element shape function object
`shdisc`	Discontinuous shape function object
`shdiv`	Divergence shape function object
`shgp`	Gauss-point shape function object
`shherm`	Hermite shape function object
`shlag`	Lagrange shape function object
`shuwhelm`	Scalar plane wave basis function object

Element Syntax Classes

FUNCTION	PURPOSE
`elconst`	Global expression variable element
`elcontact`	Contact map operator element
`elcplextr`	Extrusion coupling variable element
`elcplgenint`	Destination-aware integration coupling variable element
`elcplproj`	Projection coupling variable element
`elcplscalar`	Integration coupling variable element
`elcplsum`	Summation coupling variable element
`elcurlconstr`	Vector constraint element
`elempty`	Empty element which defines basic syntax
`elepspec`	Evaluation and constraint point pattern declaration element
`eleqc`	Coefficient and general form equation element
`eleqw`	Weak form equation element
`elgeom`	Geometric variable element
`elgpspec`	Integration point pattern declaration element
`elinline`	Inline function declaration element
`elinterp`	Interpolation function declaration element
`elinv`	Inverse matrix component variable element

FUNCTION	PURPOSE
`elirradiation`	Irradiation coupling variable element
`elmapextr`	Extrusion map operator element
`elmesh`	Mesh variable element
`elode`	Global scalar variable and equation element
`elpconstr`	Pointwise constraint element
`elpiecewise`	Piecewise function declaration element
`elplastic`	Plastic strain variable element
`elpric`	Principal component and vector variable element
`elsconstr`	Coefficient and general form constraint element
`elshape`	Shape function declaration element
`elshell_arg2`	Shell equation element
`eluwhelm`	Ultraweak variational form equation element
`elvar`	Expression variable element

Mathematical Functions

FUNCTION	PURPOSE
`fldc1hs` (p. 210)	Smoothed Heaviside function with continuous first derivative
`fldc1hs` (p. 210)	Derivative of `flc1hs`
`flc2hs` (p. 210)	Smoothed Heaviside function with continuous second derivative
`fldc2hs` (p. 210)	Derivative of `flc2hs`
`flsmhs` (p. 223)	Smoothed Heaviside function
`fldsmhs` (p. 223)	Derivative of smoothed Heaviside function
`flsmsign` (p. 223)	Smoothed sign function
`fldsmsign` (p. 223)	Derivative of smoothed sign function

Obsolete Functions in 3.5a

FUNCTION	PURPOSE	REPLACEMENT
`dst`, `idst`	Discrete sine transform	
`meshpoi`	Make regular mesh on a rectangular geometry	`meshmap`, `meshconvert`
`pde2draw`	Convert a PDE Toolbox geometry description	
`pde2fem`	Convert a PDE Toolbox model description to an FEM structure	
`pde2geom`	Convert a PDE Toolbox decomposed geometry	
`poisson`	Fast solution of Poisson's equation on a rectangular grid	`femstatic`

Obsolete Functions in 3.5

FUNCTION	PURPOSE	REPLACEMENT
`meshhex2tet`	Convert hexahedral mesh to tetrahedral mesh	`meshconvert`
`meshquad2tri`	Convert quadrilateral mesh to triangular mesh	`meshconvert`

Obsolete Functions in 3.3

FUNCTION	PURPOSE	REPLACEMENT
`shvec`	First-order simplex vector shape element	`shcurl`

Obsolete Functions in 3.2

FUNCTION	PURPOSE	REPLACEMENT
`dxfread`	Import geometry from DXF file	`geomimport`
`dxfwrite`	Export geometry to DXF file	`geomexport`
`igesread`	Import 3D geometry from IGES file	`geomimport`
`stlread`	Import 3D geometry from STL file	`geomimport`
`vrmlread`	Import 3D geometry from VRML file	`geomimport`

Obsolete Functions in 3.1

FUNCTION	PURPOSE	REPLACEMENT
`flgetrules`	Import differentiation rules from FEMLAB 1.1	
`flgeomsf2`	Set 2D geometry object weights on standard form	
`flsde`	Indices of edges in a set of subdomains	
`flsdp`	Indices of points in a set of subdomains	
`flsdt`	Indices of elements in a set of subdomains	
`fltrg`	Triangle geometry data	

Obsolete Functions in FEMLAB 3.0

FUNCTION	PURPOSE	REPLACEMENT
`appl2fem`	Expand application mode data to FEM structure	`multiphysics`
`change`	Change 2D geometry object	
`elemdefault`	Return available default element types for an application mode	
`faceprim3`	Primitive 3D face object	
`femiter`	Solve stationary PDE problem by iterative methods	`fem{n}lin`
`fldae`	Implicit DAE solver	`femtime`
`fldaek`	Iterative implicit DAE solver	`femtime`
`fldaspk`	Direct or iterative implicit DAE solver	`femtime`
`fleeceng`	Energy norm error estimator function	`adaption`
`fleel2`	L^2 norm error estimator function	`adaption`

FUNCTION	PURPOSE	REPLACEMENT
`fleelfun`	Linear functional error estimator	`adaption`
`fleig`	Solve generalized sparse eigenvalue problem	`femeig`
`flgbit`	Good Broyden iterative solver	`fem{n}lin`
`flgmres`	GMRES iterative solver	`fem{n}lin`
`flisop2p1`	Matrix M-file for Navier-Stokes Iso P2-P1 element	
`fllrq`	Iterative real symmetric definite generalized eigenvalue solver	`femeig`
`flngbit`	Good Broyden iterative solver for use with `fldaek`	`femtime`
`flngmres`	GMRES iterative solver for use with `fldaek`	`femtime`
`flntfqmr`	TFQMR iterative solver for use with `fldaek`	`femtime`
`fltfqmr`	TFQMR iterative solver	`fem{n}lin`
`fltpft`	Minimize the error for a given number of elements	`adaption`
`fltpqty`	Refine a given fraction of the elements	`adaption`
`fltpworst`	Refine elements with error greater than a fraction of the worst error	`adaption`
`multigrid`	Linear or nonlinear (adaptive) multigrid solver	`fem{n}lin`
`solidprim3`	Primitive 3D solid object	

In FEMLAB 3.0, all FEMLAB 2.3 Element Class Methods, Element Library Low-Level Functions, and Shape Function Class Methods are obsolete.

Purpose

Solve PDE problem using adaptive mesh refinement.

Syntax

```
fem = adaption(fem,...)
[fem.sol,fem.mesh] = adaption(fem,...)
```

Description

`fem = adaption(fem)` solves a linear or nonlinear stationary PDE problem or eigenvalue PDE problem. In addition, `adaption` performs adaptive mesh refinement.

`[fem.sol,fem.mesh] = adaption(fem)` explicitly returns the solution structure and the adapted mesh object.

The function `adaption` accepts the following property/value pairs:

TABLE 1-1: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Adjppr`	`auto` | `on` | `off`	`auto`	Use recovery for adjoint solution error estimate
`Callback`	string		Function to call each callback
`Callblevel`	`refine` | `param` | `nonlin`	`refine`	Callback solverlevel (param/nonlin for solver=stationary)
`Callbparam`	cell array		Parameters to the callback function
`Eefun`	`l2` | `func`	`l2`	Error estimation function
`Eefunc`	string		Error estimate functional name (eefun=func)
`Eigselect`	vector of positive scalars	1	Weights for eigenmodes
`Geomnum`	integer	1	Geometry number
`Hauto`	positive integer	7	Mesh generation parameter for refinement method meshinit
`L2scale`	vector of positive scalars	1	Scale factors for the L2 error norm
`L2staborder`	vector of positive integers	2	Orders in the stability estimate for the L2 error estimate
`Maxt`	positive scalar	`Inf`	Maximum number of mesh elements

TABLE 1-1: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Ngen`	scalar integer	5 (1D) 2 (2D) 1 (3D)	Maximum number of refinements

`Out`	`fem` | `sol` | `u` | `lambda` | `mesh` | `solcompdof` | `Kc` | `Lc` | `Dc` | `Null` | `ud` | `Nnp` | `uscale` | `stop` | cell array of these strings	`fem`
		`[sol,mesh]`
			Output variables

`Resmethod`	`weak` | `coefficient`	`weak`	Residual computation method
`Resorder`	`auto` | `scalar` | `vector`	`auto`	Order of decrease of equation residuals
`Rmethod`	`regular` | `longest` | `meshinit`	`longest`	Refinement method
`Solver`	`stationary` | `eigenvalue`	`stationary`	Solver type
`Tpfun`	`fltpft` | `fltpworst` | `fltpqty`	`fltpft`	Element selection method
`Tppar`	Nonnegative real number		Parameter to the element selection method

In addition, the common solver properties listed under `femsolver` are available. Also, when using the `stationary` solver type, the properties listed under `femstatic` apply, and for the `eigenvalue` solver type the properties in `femeig` apply. See therefore the entries `femsolver`, `femstatic`, and `femeig` for more information about the property/values.

Algorithm

The algorithm solves a sequence of PDE problems using a sequence of refined meshes. The first mesh is obtained from the mesh field in the FEM structure. The following generations of meshes are obtained by solving the PDE problem, computing a mesh element error indicator based on the error estimate function, selecting a set of elements based on the element pick function, and then finally refining these elements. The solution to the PDE problem is then recomputed. The loop continues until the maximum number of element generations has been reached or until the maximum number of elements is obtained.

The PDE problem is stored in the FEM structure `fem`. See `femstruct` for details. The adaptive solver works in one geometry at a time. You specify the geometry number in the property `geomnum`. The solver only supports simplex meshes. The residual computation method `weak` support all solution forms. The residual computation method `coefficient` does not support the solution form weak, weak contributions, or constraints on subdomains.

First, the solver chosen by the property `Solver` is called.

Error Estimation

Then, the residuals in the equations are computed for all mesh elements. The *error estimation function* is given by the property `Eefun`. For solver type `stationary` there are two functions available: `l2` and `func`. The function `l2` computes the error indicator using the L^2 norm and the function `func` computes the error indicator using a functional. For the solver type `eigenvalue` only the `l2` function is available.

If the error estimate function `functional` is used, you must provide a functional variable name using the property `eefunc`. This variable must have global scope. The algorithm computes the adjoint solution related to the functional and estimates the error in this solution. This error estimate is used for the selection of elements to refine. Two methods are supported to estimate the error in the adjoint solution: a recovery technique and a gradient-based method. By selecting `adjppr=on` you can enforce using the recovery technique, and with `adjppr=off` the adaptive solver uses the gradient method. When using the `auto` option, the algorithm automatically detects if the model uses only Lagrange basis function. If so, the recovery technique is used. Otherwise, the algorithms chooses the gradient-based method. In the Solver Log you can inspect which method is selected. See "The Adaptive Solver Algorithm" on page 540 for more information.

The error estimator gives local error indicators $(f(i,j)h(j)^{\beta(i)})^{\alpha}\mathrm{Vol}(j)$, where i is the equation number, j is the mesh element number, h is the mesh element size, and Vol is the mesh element volume. $f(i,j)$ is the scaled absolute value of the ith equation residual on the jth mesh element. The mesh element error indicator is the sum of these local error indicators over the equation index i. The global error indicator is the α root of the sum of the mesh element error indicators over the mesh element index j. See "The Adaptive Solver Algorithm" on page 540 for more information.

If the eigenvalue solver is used, you can specify the weighting of the error indicators for the different eigenfunctions using the property `Eigselect`. The nth component of the `Eigselect` vector is the weight for the error indicator of the *n*th eigenfunction.

Mesh Refinement

Then, a refinement of the mesh is generated based on the local error indicators. The aim is to refine the mesh most where the errors are largest. The mesh refinements ratios are determined by the function given in the property `Tpfun`, with parameter `Tppar`. There are three predefined functions available: `fltpft`, `fltpworst`, and `fltpqty`. `fltpft` tries to minimize the total error for a prescribed mesh size, namely `Tppar` times the current number of mesh elements (the default `Tppar` is 1.7). Each element can be refined several times. `fltpworst` and `fltpqty` refine each element at most once. `fltpworst` refines the elements with an error greater than a fraction of the worst error, whereas `fltpqty` refines a given fraction of the elements (the fraction is given in the property `Tppar`, the default is 0.5).

The property `Resorder` is a scalar or vector that gives the order of decrease of the equation residuals as the mesh size `h` tends to 0. If it is a vector, the residual of the *n*th equation is $O(h^{\mathrm{Resorder}(n)})$. If `Resorder` equals `auto`, the software determines the order of decrease in equation residuals on the basis of the shape function orders in the model. Roughly speaking, the order is one less than the shape function order for the corresponding equation.

Once the mesh refinement ratios have been determined, the mesh is refined using the method given in the property `Rmethod`, and the algorithm starts a new iteration. The refinement method is either `longest`, `regular`, or `meshinit`. Details on the first two refinement methods can be found in the entry on `meshrefine`. `Meshinit` means that a new mesh is generated through a call to `meshinit` using the `Hmesh` property to control the element sizes.

Convergence Control

No more than `Ngen` successive refinements are attempted. Refinement is also stopped when the number of elements in the mesh exceeds `Maxt`.

The property `Stop` makes it possible to return a partially converged solution when the nonlinear, iterative, or eigenvalue solver fails at some point. If a failure occurs, the result from the previous iteration is returned. The output value `Stop` is 0 if a complete solution was returned, 1 if a partial solution was returned, and 2 if no solution was returned.

If the property `callback` is given, the solver makes interrupts (or callbacks) and calls a function with the given name. To control when the solver makes a callback, use the `callblevel` property. When `callblevel` is `refine`, the solver makes a callback after each mesh refinement step. When the `solver` is `stationary`, the `callblevel` values `param` and `nonlin` are supported as well; see the description under the entry for `femstatic` on page 182 for details.

When the `Report` property is `on`, a progress window is shown. Information about the progress of the adaptive process is printed after each adaptive step. You get a message on the number of mesh elements obtained by the adaptive step, and an error indicator. This error indicator is not an absolute error estimate. In favorable cases there is a constant C such that C times the error indicator is an upper bound of some norm of the error.

EXAMPLE: EIGENMODES OF A NONCONVEX GEOMETRY

Solve the eigenvalue problem

$$\begin{cases} -\nabla\cdot\nabla u = \lambda u & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases},$$

where the domain Ω is a polygon with some concave corners. Adapt the mesh for the first and second eigenpair and compare. Finally, adapt the mesh using a equally weighted sum of the error estimates of both these eigenpairs.

```
clear fem
fem.geom = poly2([-1,-1,-0.5,-0.5,1,1,0.5,0.5],...
                 [0,-0.4,-0.4,0,0,0.6,0.6,0.2]);
fem.shape = 2;
fem.equ.da = 1; fem.equ.c = 1;
fem.bnd.h = 1;
fem.solform = 'general';
```

First set the adaptive solver to eigenvalue and use the L^2-norm error estimator. Adapt for the first eigenvalue and solve the problem using a maximum number of 500 triangles:

```
fem.mesh = meshinit(fem);
fem.xmesh = meshextend(fem);
fem = adaption(fem,'solver','eigenvalue','eefun','fleel2',...
               'eigselect',1,'maxt',500);
clf
subplot(211), postsurf(fem,'u'), axis equal
subplot(212), meshplot(fem), axis equal
```

Now solve the same problem but adapt for the second eigenvalue:

```
fem.mesh = meshinit(fem);
fem.xmesh = meshextend(fem);
fem = adaption(fem,'solver','eigenvalue','eefun','fleel2',...
               'eigselect',2,'maxt',500);
clf
subplot(211), postsurf(fem,'u','solnum',2), axis equal
subplot(212), meshplot(fem), axis equal
```

Finally, generate a mesh adapted for both these eigenvalues. Do so by specifying a vector of weights for the errors in each eigenpair:

```
fem.mesh = meshinit(fem);
fem.xmesh = meshextend(fem);
fem = adaption(fem,'solver','eigenvalue','eefun','fleel2',...
               'eigselect',[1 2],'maxt',500);
clf
subplot(211), postsurf(fem,'u'), axis equal;
subplot(212), meshplot(fem), axis equal;
```

Cautionary

The change of solution form to weak in COMSOL Multiphysics 3.3 may make it necessary to add `fem.solform = 'general'`; to your script models before calling `meshextend`.

The Coefficient residual computation method does not support weak equations and does not take other weak contributions into account.

Diagnostics

Upon termination, one of the following messages is displayed:

- `Maximum number of elements reached`
- `Maximum number of refinements reached`

Compatibility

COMSOL Multiphysics 3.2: the change to weak solution may make it necessary to add `fem.solform = 'general'`; to your script models before calling `meshextend`.

FEMLAB 3.1: error estimators `fleelfun`, `fleeceng`, and property `Stop` are not supported.

The property `Variables` has been renamed `Const` in FEMLAB 2.3.

The properties `epoint` and `tpoint` are obsolete from FEMLAB 2.2. Use `fem.***.gporder` to specify integration order. See `assemble` for details.

The properties `toln` and `normn` has been made obsolete from FEMLAB 1.2. `Ntol` replaces `toln`.

See Also

`femstruct`, `meshinit`, `meshrefine`, `meshextend`, `femeig`, `femlin`, `femnlin`

Purpose

Create elliptical or circular arc.

Syntax

```
c = arc1(cx,cy,a,b,theta,phi1,phi2)
c = arc2(cx,cy,a,b,theta,phi1,phi2)
c = arc1(cx,cy,r,phi1,phi2)
c = arc2(cx,cy,r,phi1,phi2)
```

Description

`c = arc1(cx,cy,a,b,theta,phi1,phi2)` creates a 2D curve geometry object in the form of an elliptical arc, centered in the coordinates given by `cx` and `cy`. The lengths of the semi-axes are `a` and `b`, and they are rotated the angle `theta`. The start and end angles are `phi1` and `phi2`, respectively, and are specified with respect to the semi-axes of the ellipse. The valid range of these angles is `0<=phi1`, `phi2<2*pi`.

`c = arc2(cx,cy,a,b,theta,phi1,phi2)` creates a 2D solid geometry object in the form of an elliptical sector.

`c = arc1(cx,cy,r,phi1,phi2)` creates a 2D curve geometry object in the form of a circular arc, where `r` is the radius.

`c = arc2(cx,cy,r,phi1,phi2)` creates a 2D solid geometry object in the form of a circular sector.

Examples

The following commands create two circular arc objects, coerce them into one curve object, and plot the result:

```
c1 = arc1(0,-1,1,pi/2,3*pi/2);
c2 = arc1(0,1,1,3*pi/2,pi/2);
g = geomcsg({},{c1,c2});
c = curve2(g)
geomplot(c)
axis equal
```

Compatibility

The FEMLAB 2.3 syntax is obsolete but still supported.

See Also

`geom0`, `geom1`, `geom2`, `geom3`, `curve2`, `curve3`

Purpose

Assemble the stiffness matrix, right-hand side, mass matrix, damping matrix, and constraints of a PDE problem.

Syntax

```
[K,L,M,N] = assemble(fem,...)
[K,L,M,N,D] = assemble(fem,...)
[K,L,M,N,D,E] = assemble(fem,...)
[D,M,...] = assemble(fem,'Out',{'D' 'M' ...}, ...)
```

Description

`assemble` is a fundamental function in COMSOL Multiphysics. It assembles a PDE problem using a finite element discretization.

For time-dependent problems, the finite element discretization is the system of ODEs

$$\begin{aligned} 0 &= L(U, \dot{U}, \ddot{U}, t) - N_F(U, t)\Lambda \\ 0 &= M(U, t) \end{aligned}$$

where L is the residual vector, M is the constraint residual vector, U is the solution vector, and Λ is the Lagrange multiplier vector. The linearization of this system uses the stiffness matrix K, the damping matrix D, the mass matrix E, and the constraint Jacobian matrix N given by

$$K = -\frac{\partial L}{\partial U}, \qquad D = -\frac{\partial L}{\partial \dot{U}}, \qquad E = -\frac{\partial L}{\partial \ddot{U}}, \qquad N = -\frac{\partial M}{\partial U}$$

Here N_F is the constraint force Jacobian matrix. If only ideal constraints are used then

$$N_F = N^T .$$

All these matrices can depend on the solution vector U. The matrices K, D, and E can also depend on the time derivatives $\dot{U}$ and $\ddot{U}$.

For a stationary problem, the discretization is

$$\begin{aligned} 0 &= L(U) - N_F(U)\Lambda \\ 0 &= M(U) \end{aligned}$$

and the linearized problem is

$$\begin{aligned} K(U - U_0) &= L - N_F\Lambda \\ NU &= M \end{aligned}$$

where K, L, M, N, and N_F are evaluated for some linearization "point" $U = U_0$.

For an eigenvalue problem, the discretization reads

$$KU - (\lambda - \lambda_0)DU + (\lambda - \lambda_0)^2 EU = -N_F\Lambda$$
$$NU = M$$

where K, D, E, N, and N_F are evaluated for an equilibrium "point" $U = U_0$. The eigenvalue is denoted by λ and the linearization point for the eigenvalue by λ_0.

Table 1-2 lists the valid property/value pairs for the `assemble` function.

TABLE 1-2: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Assemtol`	scalar	`1e-12`	Assembly tolerance
`Blocksize`	positive integer | `auto`	`auto`	Assembly block size
`Complexfun`	`on` | `off`	`off`	Use complex-valued functions with real input
`Const`	cell array		Definitions of constants
`Eigname`	string		Eigenvalue name
`Eigref`	string	`0`	Linearization point for the eigenvalue
`Matherr`	`on` | `off`	`on`	Error for undefined operations
`Mcase`	non-negative integer	mesh case with largest number of DOFs	Mesh case
`Out`	K | L | M | N |NF | D | E |Ksp | A | AL | BE | C | DA | EA | F | G | GA | H | Q | R | cell array of these strings	`[K,L,M,N]` `[K,L,M,N,D]` `[K,L,M,N,D,E]`	Output matrices
`Solcomp`	cell array of strings		Degree of freedom names to solve for
`T`	scalar	`0`	Time for evaluation
`U`	solution object | numeric vector | scalar	`0`	Solution for evaluation

The property Assemtol affects the assembly process. If the local stiffness matrix elements result in a negligible global matrix entry, this element is replaced by a zero. These zeros are removed from the matrices after the assembly process, saving space and computational overhead (in a sparse matrix format the zero matrix entries does not have to be stored). The tolerance is used in a relative sense. Namely if the local matrix contribution (from one element) is $A_l = \{A_{l,ij}\}$ and the currently assembled global matrix is $A_l = \{A_{ij}\}$ then the entry A_{ij} is replaced by a zero if

$$\left|A_{ij} + [A_l]_{ij}\right| < \varepsilon_{\mathrm{a}} \mathrm{max}_{kl}\left|A_{l,\,kl}\right|$$

where $[\ ]_{ij}$ denotes the contribution from a local matrix to the global matrix entry ij, and where ε_{a} is the assembly tolerance controlled by the property Assemtol. For certain types of shape functions the procedure described above is not always safe to perform. This is the case for shape function elements with degrees of freedoms that are of different types, for example when a field variable and its spatial derivative is combined (as in the shherm or sharg_2_5 elements), or when the displacement and displacement angles are combined (as in some Euler Beam elements in the Structural Mechanics Module). For this reason, the above process is never used for local matrix contributions from these types of shape function elements. If the Assemtol is zero then no elements are neglected, but the removal of zeros are still performed. If Assemtol is negative, no elements are neglected, and zeros are not removed from the assembled matrices.

The property Blocksize determines the number of mesh elements that are assembled together in a vectorized manner. A low value gives a lower memory consumption, while a high value might give a better performance. If the default setting gives an unsatisfactory performance for the given problem, then it is recommended to test with Blocksize equal to 1000.

The properties Complexfun and Matherr are described in femsolver.

The property Const gives a list of definitions of constants to be used in evaluations. This list is a row cell array with alternating constant names and numeric values. This list is appended to the list given in fem.const. If there is a conflict, the definition in Const is used.

The properties Eigname and Eigref are described in femeig. Note that assemble has an empty default value for the property Eigname. So if you want to assemble the matrices for an eigenvalue problem formulated using and eigenvalue name, then the Eigname property must be given. If the variable name lambda is used and if

`Eigname` is not set (and if `lambda` is not defined in another way), then this variable is evaluated to zero.

The property `Out` determines which matrices to output. `Ksp` is the sparsity pattern of `K`. The matrices `A`, `AL`, `BE`, `C`, and `Q` are the contributions to the K matrix that come from the coefficients a, α, β, c, and q, respectively. The vectors `F`, `G`, and `GA` are the contributions to the L vector that come from the terms f (or F), g (or G), and γ (or Γ), respectively. The matrix `H` is the contribution to the N matrix that comes from the h coefficient. The vector `R` is the contribution to the M vector that comes from the r (or R) coefficient. The matrix `DA` is the contribution to the D matrix that comes from the d_a coefficient. The matrix `EA` is the contribution to the E matrix that comes from the e_a coefficient.

The property `Solcomp` is a cell array that specifies the names of the degrees of freedom for which to solve. This property correspondlingly restricts the sizes of the assembled matrices `K`, `L`, `M`, `N`, `D`, and `E`.

The property `T` determines for which time the matrices are evaluated.

The property `U` determines the values of the degrees of freedom for which the matrices are computed (that is, the linearization point), and also their first and second time derivatives if `U` is a time-dependent solution object. `U` can be a solution (`femsol`) object, a solution vector (this has to be a column vector with values for all the degrees of freedom in the discretized problem), or a scalar (which is expanded to a solution vector).

Examples

Sparsity Structure of Finite Element Discretization of Poisson's Equation

Assemble the stiffness matrix, right-hand side, and constraint matrices of Poisson's equation

$$\begin{cases} -\Delta u = 1 & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases}$$

where Ω is the unit disk.

```
clear fem
fem.geom = circ2;
fem.mesh = meshinit(fem);
fem.shape = 2;
fem.equ.c = 1; fem.equ.f = 1;
fem.bnd.h = 1;
fem.xmesh = meshextend(fem);
[K,L,M,N] = assemble(fem);
```

```
n = size(N,1);
```

The sparsity structure of the FEM formulation of the PDE problem is:

```
spy([K,N',L;N,sparse(n,n),M]);
```

The column to the right corresponds to the right-hand side. You can continue and solve the PDE problem by using the function `femstatic`:

```
fem.sol = femstatic('In',{'K' K 'L' L 'M' M 'N' N});
postsurf(fem,'u')
```

Compatibility

FEMLAB 3.0: the properties `Context` and `Sd` are not supported and output matrices `AS`, `ALS`, `BES`, `CS`, `DAS`, `FS`, and`GAS` not supported.

In FEMLAB 2.3, the size of the matrices `D`, `K`, and `L` was unaffected by `Solcomp`. In FEMLAB 3.0, the size of the matrices `D`, `K`, `L`, and `N` shrinks if `Solcomp` is a subset of all degree of freedom names.

The property `Variables` has been renamed `Const` in FEMLAB 2.3.

The properties `bdl`, `epoint`, `sdl`, `tpoint` are obsolete from FEMLAB 2.2. Use `fem.xxx.gporder` to specify integration order.

The outputs `KM`, `LM`, `MM`, `NM`, `DM`, `MC`, `NC`, `NCL`, `MU`, and `NU` are no longer available in FEMLAB 2.2 and later versions.

The default value for `u` and `t` is 0 in FEMLAB 1.1. In FEMLAB 1.0 it was an error to use `u` or `t` in a level 4 expression when the properties `u` or `t` were not passed to `assemble`.

See Also

`femstruct`, `femsolver`, `femlin`, `femnlin`, `femtime`, `femeig`

Purpose Compute initial value.

Syntax

```
sol = asseminit(fem,...)
sol = asseminit(fem,'u',femsrc,...)
sol = asseminit(fem,'init',femsrc,...)
```

Description `sol = asseminit(fem,...)` computes a solution object corresponding to the initial value expressions in the FEM structure `fem`.

`sol = asseminit(fem,'u',femsrc,...)` evaluates these initial value expressions using the solution `femsrc.sol` in the source FEM structure `femsrc`.

`sol = asseminit(fem,'init',femsrc,...)` transfers the solution `femsrc.sol` in the source FEM structure `femsrc` to the mesh in `fem`, using interpolation.

`fem` is an FEM structure or extended FEM structure. If `Init` is a solution object or if `Solnum` has length greater than 1, then the output solution object is of the same type as the source solution object (`Init` or `U`). Otherwise, the output solution object is of the time-dependent type (containing also first time derivatives).

The function `asseminit` accepts the following property/values:

TABLE 1-3: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Blocksize`	integer	`1000`	Assembly block size
`Complexfun`	`on` | `off`	`off`	Use complex functions with real input
`Const`	cell array		Definition of constants
`Framesrc`	string | cell array of strings	reference frame	Frame for source geometry
`Gmap`	integer vector	`0`	Geometry map
`Init`	solution object | cell array | solution vector | string | scalar | FEM structure		Initial value specification
`Mcase`	integer	lowest existing mesh case	Mesh case
`Mcasesrc`	integer	`Mcase`	Mesh case for source solution
`Out`	`fem` | `sol` | `u`	`sol`	Output

TABLE 1-3: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Outcomp`	cell array of strings		Solution components to output
`Solnum`	integer vector		Solution numbers to use in source solution
`T`	real vector		Time for evaluation
`U`	solution object | solution vector | scalar | FEM structure	`0`	Solution for evaluation
`Xmesh`	extended mesh object	`fem.xmesh`	Extended mesh for source solution

The properties `Blocksize`, `Complexfun`, `Const`, and `Outcomp` are described in `femsolver`.

The property `Framesrc` is needed when mapping a solution after remeshing in a moving mesh simulation. In such a case, the source geometry does not conform to the destination geometry. Rather, the deformed source mesh agrees with the destination geometry if the source mesh is viewed in a certain frame. The property `Framesrc` contains the name of this frame, or if there are several source geometries, a frame name for each source geometry.

The geometry map vector `Gmap` tells for each geometry `g` in the destination FEM structure `fem` the corresponding geometry number in the source FEM structure, namely `Gmap(G)`. If `Gmap(G)=0`, there is no corresponding source geometry. The default is a trivial `Gmap`, that is, the geometry numbers are unchanged.

The initial values are given by the property `Init`. This can be:

- A solution object or a solution vector, corresponding to the extended mesh `Xmesh`. That solution is mapped to the current extended mesh (`fem.xmesh`).
- A cell array of alternating DOF names and expressions, or a single expression. The DOFs will be given the values of the expressions, evaluated for the solution `U` on the extended mesh `Xmesh`. In this context, a DOF name can also be the name of the time derivative of a DOF. For example, `ut` is the time derivative of the DOF `u`.
- An FEM structure. The solution in that FEM structure will be mapped to the current extended mesh.
- A scalar. The scalar will be expanded to a solution vector.

If the property `Init` is not given, the initial value will be computed by evaluating the initial expressions in the FEM structure for the solution `U` on the extended mesh `Xmesh`.

If the source solution (`Init` or `U`) is a numeric vector, its corresponding mesh case number can be given in the `Mcasesrc` property.

The property `Solnum` gives the solution numbers to use in the source solution. If the source solution is time-dependent, interpolation at the times in `T` can be used instead. By default, only the last solution (first solution for eigensolutions) is used.

Compatibility

The properties `context`, `initmethod`, and `linsolver` are obsolete from FEMLAB 3.0.

The property `Variables` has been renamed `Const` in FEMLAB 2.3.

See Also

`assemble`, `femsolver`, `femlin`, `femnlin`, `femtime`, `adaption`, `meshextend`

Purpose

Create a right-angled block geometry object.

Syntax

```
obj = block3
obj = block2
obj = block3(lx,ly,lz,...)
obj = block2(lx,ly,lz,...)
```

Description

`obj = block3` creates a right-angled solid block object with all side lengths equal to `1`, one corner at the origin, and the local *z*-axis equal to the global *z*-axis. `block3` is a subclass of `solid3`.

`obj = block3(lx,ly,lz,...)` creates a right-angled solid block geometry object with positive side lengths `lx`, `ly`, and `lz`. `lx`, `ly`, and `lz` are positive real scalars, or strings that evaluate to positive real scalars, given the evaluation context provided by the property `const`.

The functions `block3/block2` accept the following property/values:

TABLE 1-4: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`axis`	Vector of reals or cell array of strings	`[0 0]`	Local z-axis of the object
`base`	`corner` | `center`	`corner`	Positions the object either centered about `pos` or with one corner in `pos`
`const`	Cell array of strings	`{}`	Evaluation context for string inputs
`pos`	Vector of reals or cell array of strings	`[0 0]`	Position of the object
`rot`	real or string	0	Rotational angle about `axis` (radians)

`axis` sets the local *z*-axis, stated either as a directional vector of length 3, or as a 1-by-2 vector of spherical coordinates. `axis` is a vector of real scalars, or a cell array of strings that evaluate to real scalars, given the evaluation context provided by the property `const`. See `gencyl3` for more information on `axis`.

`pos` sets the position of the object, either centered about the position or with one corner in the position. The corresponding values of `base` are `center` and `corner`. `pos` is a vector of real scalars, or a cell array of strings that evaluate to real scalars, given the evaluation context provided by the property `const`.

`rot` is an intrinsic rotational angle for the object, about its local *z*-axis provided by the property `axis`. `rot` is a real scalar, or a string that evaluate to a real scalar, given the evaluation context provided by the property `const`. The angle is assumed to be in radians if it is numeric, and in degrees if it is a string.

`obj = block2(...)` creates a right-angled surface block geometry object with properties as given for the `block3` function. `block2` is a subclass of `face3`.

Block objects have the following properties:

TABLE 1-5: BLOCK OBJECT PROPERTIES

PROPERTY	DESCRIPTION
`lx`, `ly`, `lz`	Side lengths
`base`	Base point
`x`, `y`, `z`, `xyz`	Position of the object. Components and vector forms
`ax2`	Rotational angle of symmetry axis
`ax3`	Axis of symmetry
`rot`	Rotational angle

In addition, all 3D geometry object properties are available. All properties can be accessed using the syntax `get(object,property)`. See `geom3` for details.

Examples

The following commands create a surface and solid block object, where the position is defined in the two alternative ways.

```
b1 = block2(1,2.1,0.5,'base','center','pos',[1 0 1],...
               'axis',[0 0 1],'rot',0)
get(b1,'xyz')
b2 = block3(1,1,1,'base','corner','pos',[-1 -1 -1])
get(b2,'xyz')
```

Compatibility

The FEMLAB 2.3 syntax is obsolete but still supported.

See Also

`face3`, `hexahedron2, hexahedron3`

Purpose Create flattened corners in 2D geometry object.

Syntax `g = chamfer(g1,...)`

Description `g = chamfer(g1,...)` creates flattened corners in 2D geometry object `g1` according to given property values.

The function `chamfer` accepts the following property/values:

TABLE 1-6: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`angles`	1-by-m vector		Angles (in radians) with respect to edges in first row of `edges`
`dist1`	1-by-m vector		Distances along edges in the first row of `edges`. A positive entry states the chamfered length and a negative entry states the remaining length of edge after chamfering
`dist2`	1-by-m vector		Distances along edges in second row of `edges`, using same format as `dist1`
`edges`	2-by-m matrix		Pairs of edge numbers
`lengths`	1-by-m vector		Lengths of line segments that make up the flattened corners
`out`	Cell array of strings		Determines the output
`point`	integers | `all` | `none`	`all`	`out`

The corners to chamfer is either specified with either the property `point` or `edges`. The default value is the all corners are chamfered.

The size of the chamfer is specified with any of the following combinations of properties: `dist1` and `dist2`; `dist1` and `angles`; `dist1` and `lengths`; and `length` and `angles`. If only `dist1` is supplied the chamfering distance is equal for both edges. All these properties can be given as a vector or as a single value.

Examples Chamfer a rectangle in different ways.

```
r = rect2;
s1 = chamfer(r,'dist1',0.1);
s2 = chamfer(r,'edges',[1 2;2 3],'angles',pi/4,'lengths',0.5);
```

Diagnostics

If a chamfer cannot be created according to the specified properties this corner is ignored.

When the chamfers generates intersections with other edges in the geometry, an error message is given.

Compatibility

FEMLAB 3.0: The property `trim` is no longer supported. Only pair of edges that have a common vertex can be chamfered. For edges that are not linear, the linear approximation of the edge in the corner is used to compute a chamfer.

See Also

`curve2`, `curve3`, `fillet`

Purpose Create circle geometry object.

```
obj = circ2
obj = circ1
obj = circ2(r,...)
obj = circ1(r,...)
```

Description `obj = circ2` creates a solid circle geometry object with radius 1, centered at the origin. `circ2` is a subclass of `ellip2` and `solid2`.

`obj = circ2(r,...)` creates a circle object with radius r, centered at the origin. `r` is a positive real scalar, or a string that evaluates to a positive real scalar, given the evaluation context provided by the property `const`.

The functions `circ2/circ1` accept the following property/values:

TABLE 1-7: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`base`	`corner` | `center`	`center`	Positions the object either centered about `pos` or with the lower left corner of surrounding box in `pos`
`const`	Cell array of strings	`{}`	Evaluation context for string inputs
`pos`	Vector of reals or cell array of strings	`[0 0]`	Position of the object
`rot`	real or string	0	Rotational angle about `pos` (radians)

`obj = circ1(...)` creates a curve circle geometry object with properties as given for the `circ2` function. `circ1` is a subclass of `ellip1` and `curve2`.

Circle objects have the following properties:

TABLE 1-8: CIRCLE OBJECT PROPERTIES

PROPERTY	DESCRIPTION
`r`	Radius
`base`	Base point
`x`, `y`	Position of the object
`rot`	Rotational angle

In addition, all 2D geometry object properties are available. All properties can be accessed using the syntax `get(object,property)`. See `geom2` for details.

Examples The commands below create a unit solid circle geometry object and plot it.

```
c1 = circ2(1,'base','center','pos',[0 0]);
get(c1,'base')
geomplot(c1)
```

Compatibility The FEMLAB 2.3 syntax is obsolete but still supported.

See Also `ellip1`, `ellip2`, `curve2`, `curve3`

Purpose

Start COMSOL software products.

Syntax

```
comsol
comsol server
```

Description

`comsol` starts the COMSOL Multiphysics graphical user interface from MATLAB.

`comsol server` starts a COMSOL Multiphysics server within the MATLAB process. You can connect to the COMSOL Multiphysics server from a COMSOL Multiphysics client. The COMSOL Multiphysics client must be started outside MATLAB.

There is also a `comsol` command available on the command prompt in Windows, UNIX/Linux, and Mac. Using the COMSOL command you can start COMSOL Multiphysics running stand alone. You can also start a COMSOL Multiphysics client for connecting to a COMSOL Multiphysics server. The options to this command are listed in the *COMSOL Installation and Operations Guide*.

Purpose

Create a circular cone geometry object.

Syntax

```
c3 = cone3
c2 = cone2
c3 = cone3(r,h)
c2 = cone2(r,h)
c3 = cone3(r,h,ang)
c2 = cone2(r,h,ang)
c3 = cone3(r,h,ang,...)
c2 = cone2(r,h,ang,...)
```

Description

`c3 = cone3` creates a solid circular cone geometry object with bottom radius and height equal to 1, top radius equal to 0.5, and the center of the bottom at the origin. `cone3` is a subclass of `econe3`.

`c3 = cone3(r,h)` creates a solid circular cone geometry object, with bottom radius `r`, height `h`, and top radius `r/2`.

`c3 = cone3(r,h,ang)` creates a solid circular cone geometry object, with bottom radius `r`, height `h`, and the angle `ang` between the local *z*-axis and a generator of the conical surface. `ang` is given in radians in the interval `[0,pi/2)`.

The functions `cone3/cone2` accept the following property/values:

TABLE 1-9: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`axis`	Vector of reals or cell array of strings	`[0 0]`	Local z-axis of the object
`const`	Cell array of strings	`{}`	Evaluation context for string inputs
`pos`	Vector of reals or cell array of strings	`[0 0]`	Position of the object
`rot`	real or string	0	Rotational angle about `axis` (radians)

For more information on input arguments and properties see `gencyl3`.

`c2 = cone2(...)` creates a surface circular cone geometry object without bottom and top faces, according to the arguments as described for `cone3`. `cone2` is a subclass of `econe2`.

Cone objects have the following properties:

TABLE 1-10: CONE OBJECT PROPERTIES

PROPERTY	DESCRIPTION
`r`	Radius
`h`	Height
`ang`	Semi-angle
`x, y, z, xyz`	Position of the object. Components and vector forms
`ax2`	Rotational angle of symmetry axis
`ax3`	Axis of symmetry
`rot`	Rotational angle

In addition, all 3D geometry object properties are available. All properties can be accessed using the syntax `get(object,property)`. See `geom3` for details.

See `geomcsg` and `geom` for more information on geometry objects.

Compatibility

The FEMLAB 2.3 syntax is obsolete but still supported. The numbering of faces, edges and vertices is different from the numbering in objects created in 2.3.

Examples

Create a cone with an apex

```
h = 2;
r = 1;
c3 = cone3(r,h,atan(r/h));
get(c3,'ang')
```

Created truncated and rotated cone

```
c2 = cone2(r,h,atan(0.7*r/h),'pos',[1 -2 4],...
           'axis',[1 -1 0.3],'rot',pi/3);
get(c2,'ax2')
```

See Also

`cylinder2`, `cylinder3`, `econe2`, `econe3`, `face3`, `gencyl2`, `gencyl3`, `geom0`, `geom1`, `geom2`, `geom3`, `geomcsg`

Purpose

Create a curve object.

Syntax

```
c3 = curve3(x,y,z)
c3 = curve3(x,y,z,w)
c3 = curve3(vtx,vtxpre,edg,edgpre,fac,mfdpre,mfd)
[c3,...] = curve3(g3,...)
c3 = curve3(g2)
c2 = curve2(x,y)
c2 = curve2(x,y,w)
c2 = curve2(vtx,edg,mfd)
[c2,...] = curve2(g2,...)
```

Description

`c3 = curve3(x,y,z)` creates a 3D curve object. The degree is determined from the number of control points given in the vectors, `x`, `y`, and `z`. Length 2 generates a straight line. Lengths 3 and 4 generates rational Bézier curves of degrees 2 and 3 respectively. Unit weights are used.

`c3 = curve3(x,y,z,w)` works similarly to the above, but also applies the positive weights w to the control points of the curve.

`c3 = curve3(vtx,vtxpre,edg,edgpre,fac,mfdpre,mfd)` creates 3D curve geometry object `c3` from the arguments `vtx`, `vtxpre`, `edg`, `edgpre`, `fac`, `mfdpre`, and `mfd`. The arguments must define a valid 3D curve object. See `geom3` for a description of the arguments.

`[c3,...] = curve3(g3,...)` coerces the 3D geometry object `g3` to a 3D curve object `c3`.

`c3 = curve3(g2)` coerces the 2D geometry object `g2` to a 3D curve object `c3`, by embedding it in the plane $z = 0$.

`c2 = curve2(x,y)` creates a 2D curve object in the form of a Bézier curve, with the control points given by the vectors x and y of the same lengths. Length 2 generates a straight line. Lengths 3 and 4 generates rational Bézier curves of degrees 2 and 3, respectively. Unit weights are used.

`c2 = curve2(x,y,w)` works similarly to the above, but also applies the positive weights w to the control points of the curve.

`c2 = curve2(vtx,edg,mfd)` creates a 2D curve object from the properties `vtx`, `edg`, and `mfd`. The arguments must define a valid 2D curve object. See `geom2` for a description of the arguments.

`[c2,...] = curve2(g2,...)` coerces the 2D geometry object `g2` to a 2D curve object.

The coercion functions `[c2,...] = curve2(g2,...)` and `[c3,...] = curve3(g3,...)` accept the following property/values:

TABLE 1-11: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`out`	`stx` | `ftx` | `ctx` | `ptx`	`{}`	Cell array of output names

See `geomcsg` and `geom` for more information on geometry objects.

The *n*D geometry object properties are available. The properties can be accessed using the syntax `get(object,property)`. See `geom` for details.

Examples

The commands below compute the union of a unit circle and a unit square, coerce the solid object to a curve object, and plot the result.

```
s = circ2+square2;
c = curve2(s);
geomplot(c)
```

The following commands generate and plot an elliptic 3D arc:

```
c = curve3([0 1 2],[0 1 0],[0 1 2],[1 1/sqrt(2) 1]);
geomplot(c)
```

Compatibility

The FEMLAB 2.3 syntax is obsolete but still supported.

See Also

`face3`, `geom0, geom1, geom2, geom3`, `geomcsg`, `point1, point2, point3`

Purpose

Create a cylinder geometry object.

Syntax

```
c3 = cylinder3
c2 = cylinder2
c3 = cylinder3(r,h)
c2 = cylinder2(r,h)
c3 = cylinder3(r,h,...)
c2 = cylinder2(r,h,...)
```

Description

`c3 = cylinder3` generates a solid cylinder object, with radius and height equal to 1, axis along the *z*-axis and bottom surface centered at the origin. `cylinder3` is a subclass of `cone3`.

`c3 = cylinder3(r,h)` generates a solid cylinder object with radius `r` and height `h`.

The functions `cylinder3/cylinder2` accept the following property/values:

TABLE 1-12: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`axis`	Vector of reals or cell array of strings	`[0 0]`	Local z-axis of the object
`const`	Cell array of strings	`{}`	Evaluation context for string inputs.
`pos`	Vector of reals or cell array of strings	`[0 0]`	Position of the bottom surface
`rot`	real or string	0	Rotational angle about `axis` (radians)

For more information on input arguments and properties, see `gencyl3`.

`c2 = cylinder2(...)` creates a surface cylinder object, from arguments as described for `cylinder3`. `cylinder2` is a subclass of `cone2`.

Cylinder objects have the following properties:

TABLE 1-13: CYLINDER OBJECT PROPERTIES

PROPERTY	DESCRIPTION
`r`	Radius
`h`	Height
`x`, `y`, `z`, `xyz`	Position of the object. Components and vector forms
`ax2`	Rotational angle of symmetry axis

TABLE 1-13: CYLINDER OBJECT PROPERTIES

PROPERTY	DESCRIPTION
ax3	Axis of symmetry
rot	Rotational angle

In addition, all 3D geometry object properties are available. All properties can be accessed using the syntax `get(object,property)`. See `geom3` for details.

See `geomcsg` and `geom` for more information on geometry objects.

Examples

The following commands generates a surface cylinder object and a solid cylinder object.

```
c2 = cylinder2(0.5,4,'pos',[1,1,0],'axis',[pi/2,0]);
c3 = cylinder3(20,40,'pos',[0,0,-100],'axis',[1,1,1]);
```

Compatibility

The FEMLAB 2.3 syntax is obsolete but still supported. The numbering of faces, edges and vertices is different from the numbering in objects created in 2.3.

See Also

`gencyl2`, `gencyl3`, `cone2`, `cone3`, `face3`, `geom0`, `geom1`, `geom2`, `geom3`, `geomcsg`

Purpose Get geometry object from draw structure.

Syntax `obj = drawgetobj(fem,name)`

Description `obj = drawgetobj(fem,name)` retrieves the geometry object(s) with name `name` in the FEM structure `fem`. All objects with names beginning with `name` are returned in the cell array `obj`. If only one object matches, it is returned without an enclosing cell array.

`objs = drawgetobj(fem)` returns all objects in the draw structure `fem.draw` in a cell array.

Example

```
clear fem
fem.draw.s.objs = {rect2 rect2(1,2,0,1)};
fem.draw.s.name = {'R1' 'R2'};
drawgetobj(fem,'R1')
```

See Also `drawsetobj`, `geomanalyze`

Purpose Change geometry object in draw structure.

Syntax `fem = drawsetobj(fem,name,obj)`

Description `fem = drawsetobj(fem,name,obj)` replaces the existing geometry object named `name` with `obj` in the draw structure `fem.draw`.

Example

```
  clear fem
  fem.draw.s.objs = {rect2 rect2(1,2,0,1)};
  fem.draw.s.name = {'R1' 'R2'};
  fem = drawsetobj(fem,'R1',scale(drawgetobj(fem,'R1'),2,2));
  fem.draw.s.objs{1}
```

See Also `drawgetobj`, `geomanalyze`

Purpose

Create eccentric cone geometry object.

Syntax

```
ec3 = econe3
ec2 = econe2
ec3 = econe3(a,b,h)
ec2 = econe2(a,b,h)
ec3 = econe3(a,b,h,rat)
ec2 = econe2(a,b,h,rat)
ec3 = econe3(a,b,h,rat,...)
ec2 = econe2(a,b,h,rat,...)
```

Description

`ec3 = econe3` creates a solid eccentric cone geometry object with height and semi-axes of the elliptical bottom surface equal to one, axis along the coordinate *z*-axis, and the center of the bottom surface at the origin. `econe3` is a subclass of `gencyl3`.

`ec3 = econe3(a,b,h)` creates a solid eccentric cone geometry object with semi-axes `a` and `b`, and height `h`.

`ec3 = econe3(a,b,h,rat)` creates a cone with the non-negative ratio `rat` between the top and bottom surface.

The functions `econe3/econe2` accept the following property/values:

TABLE 1-14: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`axis`	Vector of reals or cell array of strings	`[0 0]`	Local z-axis of the object
`const`	Cell array of strings	`{}`	Evaluation context for string inputs
`displ`	2-by-n_d matrix	`[0;0]`	Displacement of extrusion top
`pos`	Vector of reals or cell array of strings	`[0 0]`	Position of the bottom surface
`rot`	real or string	0	Rotational angle about `axis` (radians)

For more information on input arguments and properties see `gencyl3`.

`ec2 = econe2(...)` creates a surface eccentric cone geometry object, without bottom and top faces, according to the arguments described for `econe3`. `econe2` is a subclass of `gencyl2`.

Eccentric cone objects have the following properties:

TABLE 1-15: ECCENTRIC CONE OBJECT PROPERTIES

PROPERTY	DESCRIPTION
`a`, `b`	Semi-axes
`r`	Radius
`h`	Height
`rat`	Ratio
`x`, `y`, `z`, `xyz`	Position of the object. Components and vector forms
`ax2`	Rotational angle of symmetry axis
`ax3`	Axis of symmetry
`rot`	Rotational angle

In addition, all 3D geometry object properties are available. All properties can be accessed using the syntax `get(object,property)`. See `geom3` for details.

Compatibility

The FEMLAB 2.3 syntax is obsolete but still supported. The numbering of faces, edges and vertices is different from the numbering in objects created in 2.3.

Examples

Create a truncated eccentric cone with the basis surface in the *xy*-plane.

```
e = econe2(10,40,20,0.5)
```

Create an eccentric cone with an apex, that is, a singular patch, on top.

```
e = econe3(1,2,4,0,'displ',[1,1],'pos',[100 100 100],...
           'axis',[0 1 4],'rot',pi/4)
```

See Also

`cone2`, `cone3`, `gencyl2`, `gencyl3`, `face3`

Purpose Define coefficient or general form constraints.

Syntax

```
el.elem = 'elcconstr'
el.g{ig} = geomnum
el.form = 'coefficient' | 'general'
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.dim{idim} = dimvarname
el.geomdim{ig}{edim}.r{eldomgrp} = rvec
el.geomdim{ig}{edim}.h{eldomgrp} = hmat
el.geomdim{ig}{edim}.cpoints{eldomgrp}{ic} = cpind
```

Description The `elcconstr` element adds a set of constraints specified in coefficient or general form, as specified by the `el.form` field, to the FEM problem. For the syntax of the `ind` field, see `elempty`. The coefficient `rvec` has the same syntax as the `fem.bnd.r` field, while `hmat` corresponds to an `fem.bnd.h` entry. See further the chapter "Specifying a Model" on page 3 of the *COMSOL Multiphysics MATLAB Interface Guide*. The `cpoints` field differs from `fem.bnd.cporder` in that it contains pattern indices instead of orders, see `elepspec`.

Dirichlet boundary conditions are implemented using `elcconstr` elements if the solution form is Coefficient or General. When assembling in the Weak solution form, an `elpconstr` elements replaces the `elcconstr`.

Examples In a 2D model, add a Dirichlet boundary condition on `u` at boundary 1 and 2 using constraint point pattern 1:

```
el.elem = 'elcconstr';
el.g = {'1'};
el.form = 'coefficient';
gd.ind = {{'1','2'}};
gd.dim = {'u'};
gd.r = {{'0'}};
gd.h = {{'1'}};
gd.cpoints = {{'1'}};
el.geomdim{1} = {{},gd,{}};
fem.elem = [fem.elem {el}];
```

See Also `elempty`, `elpconstr`, `elcurlconstr`, `elepspec`, `eleqc`

Purpose

Define global expression variables.

Syntax

```
el.elem = 'elconst'
el.var{2*ivar-1} = varname
el.var{2*ivar} = varexpr
```

Description

The `elconst` element declares expression variables `varname` to be accessible across all geometries and dimensions. The defining expressions, `varexpr`, can contain any variables, including variables that are only present on some domains. Expressions are expanded in the context where evaluation is requested.

Examples

Add global expressions for the transformation between Cartesian and cylindrical polar coordinates.

```
  clear el;
  el.elem = 'elconst';
  el.var = {'r','sqrt(x^2+y^2)','phi','atan2(y,x)'};
  fem.elem = [fem.elem {el}];
```

See Also

`elempty`

Purpose

Define contact map operators.

Syntax

```
el.elem = 'elcontact'
el.g{ig} = geomnum
el.opname{iop} = opname
el.mphname{iop} = mphname
el.gapname{iop} = gapname
el.contname{iop} = contname
el.conttol{iop} = 'auto' | abstol
el.visname{iop} = visname
el.method{iop} = 'direct' | 'ball'
el.checkdist{iop} = chkdist

el.srcframe{iop} = frame
el.srcn{iop}{idim} = srcnx_i
el.dstx{iop}{idim} = dstx_i
el.dstn{iop}{idim} = dstnx_i

el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.src{iop} = eldomgrplist
```

Description

The `elcontact` element defines contact map operators and related gap distance, contact flag and visibility flag variables.When evaluated, a contact map operator searches for the closest source (or *master*) point found following a ray in the `dstn` direction from the point given by the `dstx` expressions evaluated at the destination (or *slave*) point.

For each operator name, a set of master domains is specified by listing one or more domain group indices in the corresponding `el.geomdim{ig}{edim}.src{iop}` field. Normal direction expressions for the master domains are specified in the `el.srcn{iop}{idim}` fields. Coordinate expressions for the master are obtained indirectly from the frame specified in the `el.srcframe{iop} field`.

The map operator allows evaluation of any expression at a corresponding master point, while the gap distance variable evaluates to the distance between master and slave point. The optional `mphname` field for each map operator gives the name of a corresponding multiphysics operator. This second operator evaluates to the same value as the main contact operator, but its Jacobian does not contain any contribution from the map (mesh position), only from the argument expression

The contact flag variable evaluates to a nonzero value if th.e master point is well defined and the gap distance less than the `conttol` treshold value. The visibility flag variable is nonzero if a corresponding master point was found for the slave point where the flag is evaluated.

There are two slightly different methods available to search for master points. The `'direct'` method is a clean and stable direct search algorithm while the `'ball'` method is faster by only treating master elements inside a given ball radius accurately. This ball radius can be set using the `checkdist` field, and should normally be larger than any mesh element taking part in the search.

Cautionary

The `elcontact` element is only implemented for boundaries. That is, no edge-edge, edge-boundary or similar contact can be detected and evaluated.

The computation of complete Jacobians rely on the availability of spatial derivatives of the mapped expression with respect to local coordinate directions. These local derivatives cannot be calculated for all variables, a notable example being any global spatial derivatives. Therefore, for expressions like $\mathrm{map}(uTx)$ some Jacobian contributions will be missing.

Examples

Evaluate and display the distance from a hard surface to the closest point of a cylinder lying on its side on the surface.

```
clear fem;
fem.geom = ...
rect2(2,0.2,'pos',[-1,-0.2])+circ2(0.8,'pos',[0,0.8]);
fem.mesh = meshinit(fem);
fem.sshape = 2;

clear el;
el.elem = 'elcontact';
el.g = {'1'};
el.opname = {'map'};
el.gapname = {'gap'};
el.visname = {'vis'};
el.method = 'ball';
el.checkdist = '1';
el.srcframe = {'xy'};
el.srcn = {{'nx','ny'}};
el.dstx = {{'x','y'}};
el.dstn = {{'nx','ny'}};

clear src11
src11.ind = {{'3','4'}};
src11.src = {{'1'}};
el.geomdim{1} = {{},src11};

fem.elem = {el};
fem.xmesh = meshextend(fem);
fem.sol = asseminit(fem);
```

```
postcrossplot(fem,1,[6 8],'lindata','if(vis,gap,0)',...
'linxdata','if(vis,map(x),sign(x))');

% compare to the theoretical value
hold on;
x=-1:0.05:1;
plot(x,sqrt(x.^2+0.8^2)-0.8,'ro')
```

See Also `elmapextr`

Purpose Define extrusion coupling variables.

Syntax

```
el.elem = 'elcplextr'
el.g{ig} = geomnum
el.var{ivar} = varname
el.map{imap} = linmap | genmap | unitmap
el.usenan = 'true' | 'false'
el.extttol = tol
el.src{ig}{edim}.ind{srcdomgrp} = domainlist
el.src{ig}{edim}.expr{ivar}{srcdomgrp} = srcexpr
el.src{ig}{edim}.map{ivar}{srcdomgrp} = imap
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.map{ivar}{eldomgrp} = imap

linmap.type = 'linear'
linmap.sg = srcig
linmap.sv{ivtx} = srcvtx
linmap.sframe = srcframe
linmap.dg = dstig
linmap.dv{ivtx} = dstvtx
linmap.dframe = dstframe

genmap.type = 'local'
genmap.expr{idim} = transexpr
genmap.frame = frame

unitmap.type = 'unit'
unitmap.frame = frame
```

Description The `elcplextr` element declares the extrusion coupling variable names listed in the `var` field to be accessible on domains where the corresponding destination map field `geomdim{ig}{edim}.map{ivar}{eldomgrp}` entry is nonempty. Both destination and source map fields contain indices into the `map` field which consists of a list of transformation specifications.

The available transformation types are `'linear'`, `'local'`, and `'unit'`, each with its own syntax. The unit transformation takes one optional argument specifying which frame is to be used for evaluating the mesh position. If not given, the reference frame is assumed. Unit transformations without a frame field actually never has to be specified explicitly, since using index zero in the source and destination map fields is interpreted as an implicit unit transformation.

The local transformation, which is called “general” in the COMSOL Multiphysics user interface, lets you specify an arbitrary expression for each source dimension. These expressions can contain spatial coordinate variables from any frame. The `frame` field decides which frame to use in the search operation when the local transformation is used as source transformation. Therefore, choose a frame such that the transformation is as linear as possible relative to the given frame.

Linear transformations are described in the section "Extrusion Coupling Variables" on page 275 of the *COMSOL Multiphysics User's Guide*. Note that the source geometry and vertex fields `linmap.sg` and `linmap.sv` refer to the vertices that are used as source for the *transformation*, which are usually related to the coupling variable destination domain. This is because linear maps are best used as destination maps, specifying a map from the destination domain into the source domain.The `frame` fields specify which coordinate set to use in evaluating the vertex positions.

The coupling variable source transformation and expression is set up using the `src` field. A separate domain grouping is specified for the source dimensions which does not contribute to the global domain group splitting. Source expressions in the `expr{ivar}{srcdomgrp}` field can be left as empty cell arrays to signify that the particular source domain group is not part of the source for a given variable.

Cautionary

Parameter or time dependency in the source transformation is not properly detected by the solvers, which means that the source transformation will not be updated between parameter or time steps in that case. Solution dependencies in the transformation are properly detected, but do not give any Jacobian contributions from the transformation.

Examples

Calculate the first ten eigenvalues of a 3-by-2 rectangle with periodic boundary conditions both left-right and top-bottom. Different map types are used.

```
fem.geom = rect2(3,2);
fem.mesh = meshinit(fem,'hmax',0.05);
fem.equ.c = 1;
fem.equ.da = 1;
fem.bnd.ind = [0 1 2 0];
fem.bnd.constr = {'ucx-u','ucy-u'};
fem.elem = {};

el.elem = 'elcplextr';
el.g = {'1'};
el.var = {'ucx','ucy'};

clear map1;
map1.type = 'linear';
map1.sg = '1';
map1.sv = {'2','3'};
map1.dg = '1';
map1.dv = {'1','4'};
```

```
clear map2;
map2.type = 'local';
map2.expr = {'x'};

el.map = {map1 map2};

clear src;
src.ind = {{'1'},{'4'}};
src.expr = {{{},'u'},{'u',{}}};
src.map = {{{},'0'},{'2',{}}};
el.src{1} = {{},src,{}};

clear dst;
dst.ind = {{'2'},{'3'}};
dst.map = {{'1',{}},{{},'2'}};
el.geomdim{1} = {{},dst,{}};

fem.elem = [fem.elem {el}];
fem.xmesh = meshextend(fem);
fem.sol = femeig(fem,'neigs',10,'shift',1);
postplot(fem,'tridata','u','triz','u','refine',3,'solnum',8);
```

See Also `elempty`, `elcplproj`

Purpose

Define destination-aware integration coupling variables.

Syntax

```
el.elem = 'elcplgenint'
el.g{ig} = geomnum
el.var{ivar} = varname
el.global = varlist
el.src{ig}{edim}.ind{srcdomgrp} = domainlist
el.src{ig}{edim}.expr{ivar}{srcdomgrp} = srcexpr
el.src{ig}{edim}.ipoints{ivar}{srcdomgrp} = ip
el.src{ig}{edim}.iorders{ivar}{srcdomgrp} = io
el.src{ig}{edim}.frame{ivar}{srcdomgrp} = frame
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.usage{ivar} = eldomgrplist
```

Description

The `elcplgenint` element accepts the same syntax as the `elcplscalar` element with the only notable exception that a destination operator `dest(subexpr)` can be used in the source expression. The destination operator's argument will be evaluated on the destination point instead of on the source domain. This can be used to evaluate convolution integrals.

Cautionary

The integral is evaluated for each destination point, whether the `dest()` operator is present in the source expression or not. Use an `elcplscalar` element if there is no destination dependence.

Examples

Plot part of the Fourier transform of `g=|x|<1`.

```
clear fem;
fem.geom = geom1([-10 10]);
fem.mesh = meshinit(fem,'report','off','hmax',0.1);
fem.equ.gporder = 4;
fem.elem = {};

clear el
el.elem = 'elcplgenint';
el.g = {'1'};
el.var = {'G'};
clear src;
src.expr = {{'(abs(x)<1)*exp(-i*dest(x)*x)'}};
src.iorders = {'4'};
el.src = {{{},src}};
clear dst;
dst.usage = {{'1'}};
el.geomdim = {{{},dst}};
fem.elem = [fem.elem {el}];

fem.xmesh=meshextend(fem);
postplot(fem,'lindata','G','liny','G');
```

See Also `elepspec`, `elcplscalar`, `elgpspec`

Purpose

Define projection coupling variables.

Syntax

```
el.elem = 'elcplproj'
el.g{ig} = geomnum
el.var{ivar} = varname
el.map{imap} = projmap | linmap | genmap | unitmap
el.src{ig}{srcdim}.ind{srcdomgrp} = domainlist
el.src{ig}{srcdim}.expr{ivar}{srcdomgrp} = srcexpr
el.src{ig}{srcdim}.iorder{ivar}{srcdomgrp} = intorder
el.src{ig}{srcdim}.map{ivar}{srcdomgrp} = imap
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.map{ivar}{eldomgrp} = imap

projmap.type = 'projection'
projmap.sg = srcig
projmap.sv{ivtx} = srcvtx
projmap.sframe = srcframe
projmap.dg = dstig
projmap.dv{ivtx} = dstvtx
projmap.dframe = dstframe

linmap.type = 'linear'
linmap.sg = srcig
linmap.sv{ivtx} = srcvtx
linmap.sframe = srcframe
linmap.dg = dstig
linmap.dv{ivtx} = dstvtx
linmap.dframe = dstframe

genmap.type = 'local'
genmap.expr{idim} = transexpr
genmap.frame = frame

unitmap.type = 'unit
unitmap.frame = frame'
```

Description

The `elcplproj` projection variable element is closely related to the `elcplextr` element. Both elements map the `srcdim`-dimensional source domain onto an intermediate `srcdim`-dimensional fictitious domain. While the destination transformation in the `elcplextr` element maps the destination domain into the intermediate domain, the `elcplproj` element destination transformation maps the destination only into the first `srcdim-1` dimensions. The last dimension of the fictitious domain is collapsed by integration onto the first `srcdim-1` dimensions. For more information about coupling variables, see “Using Coupling Variables” on page 269 of the *COMSOL Multiphysics User’s Guide*.

All map types available for `elcplextr` can be used also in projection coupling variables. The common combination of a `unit` source map and a `linear` destination map is not very useful, though. Instead, there is a map type `projection` which specifies `srcdim+1` vertices in the source geometry and `srcdim` vertices in the destination. The basis defined by vectors from the first source vertex to each of the remaining vertices is mapped onto a right-handed orthogonal system with unit axes. The basis described by the destination vertices is then mapped onto the first `srcdim`−1 dimensions of the same orthogonal basis. This means that the direction of integration is effectively from the first source vertex to the last.

In addition to the fields present in the `elcplextr` element, the `elcplproj` requires an integration order for the line integrals evaluated for each destination point. The `iorder` field specifies the order of polynomials that should be exactly integrated.

Cautionary

Projection coupling is only implemented for simplex meshes. When finding integration limits, the `elcplproj` element works directly on the basic polyhedral mesh. Therefore, results can be inaccurate if the mesh does not properly resolve the geometry.

Parameter or time dependency in the source transformation is not properly detected by the solvers, which means that the source transformation will not be updated between parameter or time steps in that case. Solution dependencies in the transformation are properly detected, but do not give any Jacobian contributions from the transformation.

The automatic detection of nonlinear and time-dependent or parameter-dependent problems does not work properly in that all problems containing projection coupling variables are considered to be nonlinear and time dependent.

Examples

Project the diagonal cross section distance on the left and bottom edges of a square.

```
clear fem
fem.geom = square2;
fem.mesh = meshinit(fem);
fem.elem = {};

el.elem = 'elcplproj';
el.g = {'1'};
el.var = {'d','d'};

clear map1;
map1.type = 'projection';
map1.sg = '1';
map1.sv = {'1','2','3'};
```

```
map1.dg = '1';
map1.dv = {'1','2'};

clear map2;
map2.type = 'projection';
map2.sg = '1';
map2.sv = {'1','4','3'};
map2.dg = '1';
map2.dv = {'1','4'};

el.map = {map1 map2};

clear src;
src.ind = {{'1'}};
src.expr = {{'1'},{'1'}};
src.iorder = {{'1'},{'1'}};
src.map = {{'1'},{'2'}};
el.src{1} = {{},{},src};

clear dst;
dst.ind = {{'1'},{'4'}};
dst.map = {{'1',{}},{{},'2'}};
el.geomdim{1} = {{},dst,{}};

fem.elem = [fem.elem {el}];
fem.xmesh = meshextend(fem);
postint(fem,'d/sqrt(2)','edim',1,'dl',[1 4])
```

See Also `elempty`, `elcplextr`

Purpose

Define integration coupling variables.

Syntax

```
el.elem = 'elcplscalar'
el.g{ig} = geomnum
el.var{ivar} = varname
el.global = varlist
el.maxvars = maxvarlist

el.src{ig}{edim}.ind{srcdomgrp} = domainlist
el.src{ig}{edim}.expr{ivar}{srcdomgrp} = srcexpr
el.src{ig}{edim}.ipoints{ivar}{srcdomgrp} = ip
el.src{ig}{edim}.iorders{ivar}{srcdomgrp} = io
el.src{ig}{edim}.frame{ivar}{srcdomgrp} = frame
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.usage{ivar} = eldomgrplist
```

Description

The `elcplscalar` element declares the integration coupling variable names listed in the `var` field to be accessible on domain groups specified as a, possibly empty, cell array for each variable in the `usage` field, or globally if the variable index is mentioned in the `global` field. The same variable cannot be defined as both global and local.

If a variable index is included in the `maxvars` list, the `elcplscalar` element computes an approximate maximum value of the source expression over the source domains, instead of the integral. The expression is evaluated and compared only in the same quadrature points as would otherwise have been used for integration; see below.

The source domain grouping specified in the `src{ig}{edim}.ind` field does not contribute to the global domain splitting. For each variable and source domain group, a possibly empty (no contribution) source expression is given in the `expr` field.

The `ipoint` field specifies an integration point pattern using indices referring to an `elgpspec` element. If the `ipoints` field is not present, the `iorders` field will be read instead and assumed to contain Gauss-Legendre quadrature orders for the source expressions. When specifying an `elcplscalar` element in a script, the latter syntax is more convenient.

The `frame` field selects the set of spatial variables with reference to which the integration is performed. For example, if the source expression is `'1'`, using the reference frame makes the variable evaluate to the undeformed volume of the source domains, while choosing a moving frame gives you the deformed volume.

Examples

Make the average and maximum values of the solution available on the boundary of a circle.

```
fem.geom = circ2;
fem.mesh = meshinit(fem);
fem.shape = 2;
fem.equ.c = 1; fem.equ.f = 1;
fem.bnd.h = 1;
fem.elem = {};

clear el;
el.elem = 'elcplscalar';
el.g = {'1'};
el.var = {'area' 'mean' 'max'};
el.global = {'1'};
el.maxvars = {'3'};

clear src;
src.ind = {{'1'}};
src.expr = {{'1'},{'u/area'},{'u'}};
src.iorders = {{'4'},{'4'},{'4'}};
el.src{1} = {{},{},src};

clear dst;
dst.ind = {{'1','2','3','4'}};
dst.usage = {{},{'1'},{'1'}};
el.geomdim{1} = {{},dst,{}};

fem.elem = [fem.elem {el}];
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem);

postint(fem,'u/pi')
postint(fem,'mean/(2*pi)','edim',1)
postint(fem,'max/(2*pi)','edim',1)
```

See Also

`elepspec`, `elgpspec`

Purpose

Define summation coupling variables.

Syntax

```
el.elem = 'elcplsum'
el.g{ig} = geomnum
el.var{ivar} = varname
el.global = varlist
el.src{ig}{edim}.ind{srcdomgrp} = domainlist
el.src{ig}{edim}.expr{ivar}{srcdomgrp} = srcexpr
el.src{ig}{edim}.nodes{ivar}{srcdomgrp} = nodes
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.usage{ivar} = eldomgrplist
```

Description

The `elcplsum` element declares the summation coupling variable names listed in the `var` field to be accessible on domain groups specified as a, possibly empty, cell array for each variable in the `usage` field, or globally if the variable index is mentioned in the `global` field. The same variable cannot be defined as both global and local.

The source domain grouping specified in the `src{ig}{edim}.ind` field does not contribute to the global domain splitting. For each variable and source domain group, a possibly empty (no contribution) source expression is given in the `expr` field.

The `nodes` field specifies the evaluation point pattern used to perform the sum over. If the nodes field is given as `-1` or `'all'` then all points that have a degree of freedom is used, otherwise the given integer is a Lagrange point order that specifies the Lagrange points to sum over.

Example

Define `res` as the sum of the node-wise constraint forces in all nodes at the boundary of a square domain.

```
fem.geom = rect2;
fem.mesh = meshinit(fem);
fem.equ.c = 1;
fem.equ.f = 1;
fem.bnd.h = 1;

fem.elem = {};
clear el;
el.elem = 'elcplsum';
el.g = {'1'};
el.var = {'res'};
el.nodes = {'all'};
el.global = {'1'};
clear src;
src.ind = {{'1','2','3','4'}};
```

```
  src.expr = {{'reacf(u)'}};
  el.src{1} = {{},src,{}};
  fem.elem = [fem.elem {el}];

  fem.xmesh = meshextend(fem);
  fem.sol = femstatic(fem,'reacf','on');
  postglobaleval(fem,{'res'})
```

See Also `elcplscalar`, `postsum`

Purpose Define constraints compatible with first order vector elements.

Syntax

```
el.elem = 'elcurlconstr'
el.g{ig} = geomnum
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.constr{eldomgrp}{ic} = vconstr
```

Description The `elcurlconstr` element adds constraints on vector expressions, `vconstr`, which are given as cell arrays with one entry for each space dimension. The projection of the vector expression onto element edges is constrained to zero at element edge center points.

For the syntax of the `ind` field, see `elempty`. The constr field has one entry for each domain group, each being a cell array of vector expressions to be constrained.

Cautionary This element is currently tailored to fit the `shcurl` vector shape functions by constraining the actual degrees of freedom. Other uses may be possible, but performance will be unpredictable.

Examples Add a PEC condition on boundaries 1 to 6 in a 3D electromagnetics model.

```
  el.elem = 'elcurlconstr';
  el.g = {'1'};
  gd.ind = {{'1','2','3','4','5','6'}};
  gd.constr = {{{'tEx','tEy','tEz'}}};
  el.geomdim{1} = {{},{},gd,{}};
  fem.elem = [fem.elem {el}];
```

When exporting the `fem` structure from a vector element electromagnetics model, similar `elcurlconstr` elements are added to the `fem.elemmph` field.

See Also `elempty`, `elpconstr`

elempty

Purpose

Define some basic functionality of the element syntax elements.

Syntax

```
el.elem = 'elempty'
el.g{ig} = geomnum
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
```

Description

The `elempty` element does not contribute anything directly to the problem description. It is described here because all element classes are derived from elempty and therefore share some basic syntax. This "element syntax" uses a limited subset of data structures to describe the complete FEM problem. All models have to be converted into element syntax before solving, a conversion handled by the `meshextend` function. Additional elements can be added in the `fem.elem` field. Unless there are conflicts, the additional elements are added to the global problem description.

The only building blocks allowed in the element syntax are strings, row cell arrays and structs. Note that pure numerical values are not allowed: both integers and decimal number have to be wrapped up as strings. Most elements accept empty cell arrays as placeholders to signify that the element does not wish to contribute anything for some variable or on some domain.

At the top level, all elements are structs with at least the field `elem` containing the element class name as a string. Most elements define contributions that are in some way local to one or more geometries. Such elements have a `g` field, which contains a row cell array of geometry numbers (quoted as strings). These geometries are in turn referred to internally in the element using the position in the `g` field as index, called `ig`.

Most element classes specify their contributions per geometry, dimension and domain group. See further the chapter "Specifying a Model" on page 3 of the *COMSOL Multiphysics MATLAB Interface Guide* for an explanation of the domain group concept. The `geomdim` field, where present, is a nested cell array where the outer level position corresponds to local geometry index, `ig`, and inner level position corresponds to space dimension plus one, called `edim` in the element context.

The `geomdim{ig}{edim}` entries are used as structs with field names and syntax depending on the particular element class. There are, however, some common principles. Whenever there is an `ind` field present, it is a cell array where the position corresponds to element domain group number, called `eldomgrp`, each entry being a domain list. The domain lists are in turn cell arrays of quoted domain numbers.

If the `ind` field is not present in an element structure for an element that accepts a domain grouping, it is defaulted as if all domains belong to group one. Other fields can usually be specified either per domain group or using one entry valid for all groups, whether explicitly specified or defaulted as one single group. Note that this behavior is not explicitly documented for each element type.

Cautionary

Adding element syntax contributions bypasses all high-level syntax checks, which can result in unintelligible error messages or even unexpected termination of a scripting session.

Examples

Create a simple model (Poisson's equation on unit circle) and extract the element syntax created by `meshextend`.

```
clear fem;
fem.geom = circ2;
fem.mesh = meshinit(fem);
fem.shape = 2;
fem.equ.c = 1; fem.equ.f = 1;
fem.bnd.h = 1;
fem.xmesh = meshextend(fem);

elstr = fem.xmesh.getElements;
clear elem;
for i=1:length(elstr)
  elem{i} = eval(elstr(i));
end

elstr = fem.xmesh.getInitElements;
clear eleminit;
for i=1:length(elstr)
  eleminit{i} = eval(elstr(i));
end
```

The element syntax can be studied, modified, and then fed back into the `fem.elem` and `fem.eleminit` fields. An additional call to `meshextend` with property `'standard'` set to `'off'` updates the `fem.xmesh` field before solving.

```
fem.elem = elem;
fem.eleminit = eleminit;
fem.xmesh = meshextend(fem,'standard','off');
fem.sol = femstatic(fem);
postsurf(fem,'u');
```

An `elempty` element can be used to force a domain group split, which can be necessary in some cases where subdomain variables are accessed on boundaries. To split the boundary into two domain groups, add the following commands to the element syntax created above:

```
  clear el;
  el.elem = 'elempty';
  gd.ind = {{'1','2'},{'3','4'}};
  el.geomdim{1} = {{},{},gd};
  fem.elem = [fem.elem {el}];
```

See Also

elcconstr, elconst, elcplextr, elcplgenint, elcplproj, elcplscalar, elcurlconstr, elepspec, eleqc, eleqw, elgeom, elgpspec, elinline, elinterp, elinv, elirradiation, elmesh, elpconstr, elplastic, elpric, elshape, elshell_arg2, elvar

Purpose

Declare constraint point patterns.

Syntax

```
el.elem = 'elepspec'
el.g{ig} = geomnum
el.geom{ig}.ep = meshcases | patterns

meshcases.default = patterns
meshcases.case{elmcase} = patterns
meshcases.mind{elmcase} = caselist

patterns{iptrn} = lagorder | ptrnlist

ptrnlist{2*itype-1} = bmtypename
ptrnlist{2*itype} = lagorder | ptrn

ptrn{ipnt} = lcoords
```

Description

The `elepspec` element defines local evaluation point patterns typically used for pointwise constraints. In contrast to most elements, the `elepspec` does not have a `geomdim` field. The `geom` field has one entry per geometry listed in the `g` field and one subfield, `ep`, which lists a number of patterns that other elements refer to by position.

If there are no alternate mesh cases specified, each entry in the `ep` field is either an integral number, which is interpreted as a Lagrange point order to be used on all basic mesh element types and dimensions, or a cell array of pairs of basic mesh element type and a Lagrange order or an explicit pattern. For a list of basic mesh element type names, see `elshape`. Explicit patterns are cell arrays of points in the local coordinate system on each element, each point being a cell array of the same dimension as the basic mesh element type.

If there are multiple mesh cases present in the model, the `ep` field is a struct with fields `default`, `case`, and `mind`. The `default` field has the same syntax as described above for the `ep` field itself. Multiple alternate cases which need the same evaluation point patterns can be grouped together using the `mind` field. This field is a cell array containing groups of mesh case numbers, each group, `caselist`, given as a cell array. For each element mesh case group, `elmcase`, an alternate pattern specification is given in the `case` field.

Cautionary

Currently, there can only be one `elepspec` element for each geometry, which is generated by default when converting the standard syntax. Therefore, no additional `elepspec` can be added in the `fem.elem` field unless `meshextend` is called with property `standard` set to `off`.

Examples

Given a single-geometry `fem` structure with an `xmesh` field, extract elements and add an additional pattern that can be used by other elements and update the `xmesh`.

```
elstr = fem.xmesh.getElements;
clear elem;
for i=1:length(elstr)
  elem{i} = eval(elstr(i));
  if strcmp(elem{i}.elem,'elepspec')
    iepspec = i;
  end
end

elstr = fem.xmesh.getInitElements;
clear eleminit;
for i=1:length(elstr)
  eleminit{i} = eval(elstr(i));
end

newptrn = length(elem{iepspec}.geom{1}.ep)+1;
elem{iepspec}.geom{1}.ep{newptrn} = {'s(1)' {{'0.5'}}};
fem.elem = elem;
fem.eleminit = eleminit;
```

Here, additional elements using the constraint pattern with index `newptrn` can be added to the `fem.elem` field.

```
fem.xmesh = meshextend(fem,'standard','off');
```

See Also

`elempty`, `elgpspec`, `elshape`, `elpconstr`, `elcconstr`

Purpose

Define coefficient form or general form equation contributions.

Syntax

```
el.elem = 'eleqc'
el.g{ig} = geomnum
el.form = 'coefficient' | 'general'
el.eqvars = 'on' | 'off'
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.dim{idim} = dimvarname
el.geomdim{ig}{edim}.ea{eldomgrp} = eacoeff
el.geomdim{ig}{edim}.da{eldomgrp} = dacoeff
el.geomdim{ig}{edim}.c{eldomgrp} = ccoeff
el.geomdim{ig}{edim}.al{eldomgrp} = alcoeff
el.geomdim{ig}{edim}.ga{eldomgrp} = gacoeff
el.geomdim{ig}{edim}.be{eldomgrp} = becoeff
el.geomdim{ig}{edim}.a{eldomgrp} = acoeff
el.geomdim{ig}{edim}.f{eldomgrp} = fcoeff
el.geomdim{ig}{edim}.ipoints{eldomgrp}{idim} = ipind
```

Description

The `eleqc` element adds equation contributions in coefficient form or general form as specified by the `el.form` field. It also defines variables that evaluate to various parts of the equations. These equation variables can be turned off using the `eqvars` field.

For the syntax of the `ind` field, see `elempty`. The `ea`, `da`, `c`, `al`, `ga`, `be`, `a`, and `f` coefficients have the same syntax as the corresponding `fem.equ` fields. See further the chapter “Specifying a Model” on page 3 of the *COMSOL Multiphysics MATLAB Interface Guide*. In contrast to the standard syntax, the `eleqc` coefficients have the same names on all dimensions. That is, the standard `fem.bnd.g` and `fem.bnd.q` fields correspond to a `geomdim{ig}{sdim-1}.f` and a `geomdim{ig}{sdim-1}.a`, respectively.

The `ipoints` field differs from the `gporder` fields in the standard syntax (for example, `fem.equ.gporder`) in that it always contains pattern indices instead of orders (see `elgpspec`).

The COMSOL Multiphysics user interface generates an `eleqc` element for geometries where the solution form is the coefficient or general form. When assembling using the weak solution form (the default), equations are converted to weak form and an `eleqw` element is generated instead.

Cautionary

Because of the naming convention for equation variables, at most one `eleqc` element per geometry can have `equvars` set to `on`. Because the default `eleqc` element has equation variables turned on, unless otherwise specified, it is proper procedure to turn them off for any additional `eleqc` elements added in the `fem.elem` field.

Make sure that the integration point pattern index you use really does exist and corresponds to a reasonable integration order. When adding an `eleqc` element to an existing model, it may be necessary to extract and modify also the default `elgpspec` element.

Examples

Because equation contributions are simply added, you can introduce additions to a single coefficient using the `fem.elem` field.

```
clear fem;
fem.geom = circ2;
fem.mesh = meshinit(fem);
fem.shape = 2;
fem.equ.c = 1; fem.equ.f = 0;
fem.bnd.h = 1;
fem.elem = {};

clear el;
el.elem = 'eleqc';
el.g = {'1'};
el.form = 'coefficient';
el.eqvars = 'off';
clear equ;
equ.dim = {'u'};
equ.ind = {{'1'}};
equ.f = {{'1'}};
equ.ipoints = {{'1'}};
el.geomdim{1} = {{},{},equ};

fem.elem = [fem.elem {el}];
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem);
postplot(fem,'tridata','u','triz','u','refine',3);
```

See Also

`elempty`, `eleqw`, `elgpspec`

Purpose Define weak form contributions.

Syntax

```
el.elem = 'eleqw'
el.g{ig} = geomnum
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.coeff{eldomgrp}{iequ} = weak
el.geomdim{ig}{edim}.tcoeff{eldomgrp}{iequ} = dweak
el.geomdim{ig}{edim}.ipoints{eldomgrp}{iequ} = ipind
el.geomdim{ig}{edim}.dvolname{eldomgrp}{iequ} = dvolname
```

Description The `eleqw` element adds weak form contributions to the FEM problem. For the syntax of the `ind` field, see `elempty`. The `weak` and `dweak` coefficients have the same syntax as the corresponding fields in the standard `fem` struct syntax. See further the chapter "Specifying a Model" on page 3 of the *COMSOL Multiphysics MATLAB Interface Guide*. The `ipoints` field differs from the `gporder` fields in the standard syntax (for example, `fem.equ.gporder`) in that it always contains pattern indices instead of orders (see `elgpspec`).

The field `dvolname` specifies the name of the differential volume factor to be used in integrating the particular equation. If you are using multiple frames, this name effectively decides in which frame the equation is defined. There is normally a unique volume factor name tied to each frame, with `dvol` being the default for fixed meshes.

The main difference between specifying equations using `eleqw` and using `eleqc` is that the former always gives a correct Jacobian if it is possible to automatically differentiate all functions called.

Cautionary Make sure that the integration point pattern index you use really does exist and corresponds to a reasonable integration order. When adding an `eleqw` element to an existing model, it may be necessary to extract and modify also the default `elgpspec` element.

Examples As equation contributions are simply added, it is easy to introduce additional weak form terms using the `fem.elem` field.

```
clear fem;
fem.geom = circ2;
fem.mesh = meshinit(fem);
fem.shape = 2;
fem.equ.c = 1; fem.equ.f = 0;
fem.bnd.h = 1;
fem.elem = {};

clear el;
```

```
  el.elem = 'eleqw';
  el.g = {'1'};
  clear equ;
  equ.ind = {{'1'}};
  equ.coeff = {{'u_test'}};
  equ.ipoints = {{'1'}};
  el.geomdim{1} = {{},{},equ};

  fem.elem = [fem.elem {el}];
  fem.xmesh = meshextend(fem);
  fem.sol = femstatic(fem);
  postplot(fem,'tridata','u','triz','u','refine',3);
```

See Also elempty, eleqc, elgpspec

Purpose

Elevate degrees of 2D geometry object Bézier curves.

Syntax

```
ge = elevate(g,en,d)
[ge,tl] = elevate(g,en,d)
ge = elevate(g,dl)
[ge,tl] = elevate(g,dl)
```

Description

`ge = elevate(g,en,d)` elevates the degrees of edges `en` in the 2D geometry object `g`, using the degree steps `d`. `en` is a vector that specifies the edge numbers of the curves to be degree elevated, and `d` is the corresponding vector that specifies the degrees of elevation, so that curve number `en(i)` is elevated by `d(i)` degrees.

`ge = elevate(g,dl)` degree elevates the Bézier curve defined by `g.rb{i}(:,k)` and `g.wt{i}(:,k)`, by the number of degrees specified in `dl{i}(:,k)`. `dl` is a cell array of the same size as `rb` and `wt`. See `geom2` for details on these properties. The first and last entries in `dl` must be empty, since there are no curves of degree 0, and curves of maximum degree cannot be degree elevated.

`[ge,tl] = elevate(g,...)` additionally returns the cell array `tl`, of length 3, containing permutation vectors for vertices, edges and subdomains, respectively. Entry `i` of such a vector contains the entity number `j` of the geometry object `g` from which the entity `i` in `ge` originates.

Examples

Elevate the degree of edge 1 and 3 in a circle, by one degree.

```
c1 = circ2;
figure, geomplot(c1,'edgelabels','on','ctrlmode','on');
axis equal
[c2,tl] = elevate(c1,[1 3],[1 1]);
figure, geomplot(c2,'edgelabels','on','ctrlmode','on');
axis equal
```

An alternative way of obtaining the same degree elevated circle, is to use the input argument `dl`, as is done below.

```
c3 = elevate(c1,{[] [] [1 0 1 0] []});
figure, geomplot(c3,'edgelabels','on','ctrlmode','on');
axis equal
```

See Also

`geom0, geom1, geom2, geom3`

Purpose

Define geometrical variables.

Syntax

```
el.elem = 'elgeom'
el.g{ig} = geomnum
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.frame = frame
el.sorder = sorder
el.method = {'Lenoir'} | 'fl33'
```

Description

The `elgeom` element evaluates geometrical variables for the (undeformed) geometry, notably coordinates, curve and face parameters, and tangential vectors. These variables are defined on the domains in `domainlist`. The variable names are derived from the space coordinate names of the frame `frame`. See further the chapter "Specifying a Model" on page 3 of the *COMSOL Multiphysics MATLAB Interface Guide*.

The Lenoir method (the default method) provides a continuous piecewise polynomial interpolation of order `sorder`. The `fl33` method, which was the default method until version 3.3a, provides a discontinuous nonpolynomial-based interpolation. The `sorder` field is only applicable when using the Lenoir method and has a default value of 1.

Examples

The default generated `elgeom` element for a 2D model is:

```
  el.elem = 'elgeom';
  el.g = {'1'};
  el.frame = 'xy';
  el.sorder = '1';
  el.method = 'Lenoir';
```

See Also

`elempty`, `elmesh`

Purpose

Declare integration point patterns.

Syntax

```
el.elem = 'elgpspec'
el.g{ig} = geomnum
el.geom{ig}.ep{iptrn} = order | ptrnlist

meshcases.default = patterns
meshcases.case{elmcase} = patterns
meshcases.mind{elmcase} = caselist

patterns{iptrn} = lagorder | ptrnlist

ptrnlist{2*itype-1} = bmtypename
ptrnlist{2*itype} = order | ptrn

ptrn{ipnt} = {lcoords,weight}
```

Description

The `elgpspec` element defines local integration point patterns and weights for the numerical quadrature needed when assembling equations on each mesh element. In contrast to most elements, the `elgpspec` does not have a `geomdim` field. The `geom` field has one entry per geometry listed in the `g` field and one subfield, `ep`, which lists a number of patterns which other elements refer to by position.

If there are no alternate mesh cases specified, each entry in the `ep` field is either an integral number, which is interpreted as the order of polynomials that should be integrated exactly on all basic mesh element types and dimensions, or a cell array of pairs of basic mesh element type and a polynomial order or an explicit pattern. For a list of basic mesh element type names, see `elshape`. Explicit patterns are cell arrays of points and weights in the local coordinate system on each element, each point-weight pair being represented as a single cell array where the weight follows directly after the coordinates.

If there are multiple mesh cases present in the model, the `ep` field is a struct with fields `default`, `case`, and `mind`. The `default` field has the same syntax as described above for the `ep` field itself. Multiple alternate cases which need the same integration point patterns can be grouped together using the `mind` field. This field is a cell array containing groups of mesh case numbers, each group, `caselist`, given as a cell array. For each element mesh case group, `elmcase`, an alternate pattern specification is given in the `case` field.

Cautionary

Currently, there can only be one `elgpspec` element for each geometry, which is generated by default when converting the standard syntax. Therefore, no additional `elgpspec` can be added in the `fem.elem` field unless `meshextend` is called with property `'standard'` set to `'off'`.

Note that the sum of weights in an explicit pattern specification are supposed to be equal to the element's volume in the element local coordinate system, that is, to `1/2` for triangles and `1/6` for tetrahedra.

Examples

Given an `fem` structure with an `xmesh` field, extract elements and add an additional pattern which uses fourth order integration on curved simplices and explicit zeroth order integration on other simplices. Finally, the extended mesh `xmesh` is updated.

```
elstr = fem.xmesh.getElements;
clear elem;
for i=1:length(elstr)
  elem{i} = eval(elstr(i));
  if strcmp(elem{i}.elem,'elgpspec')
    igpspec = i;
  end
end

elstr = fem.xmesh.getInitElements;
clear eleminit;
for i=1:length(elstr)
  eleminit{i} = eval(elstr(i));
end

newptrn = length(elem{igpspec}.geom{1}.ep)+1;
elem{igpspec}.geom{1}.ep{newptrn} = {'s(2)', '4', ...
  'ls(2)', {{'0.333333333','0.333333333','0.5'}}};
fem.elem = elem;
fem.eleminit = eleminit;
```

Here, additional elements using the integration point pattern with index `newptrn` can be added to the `fem.elem` field.

```
fem.xmesh = meshextend(fem,'standard','off');
```

See Also

`elempty`, `elepspec`, `elshape`, `eleqc`, `eleqw`

Purpose

Declare functions and corresponding symbolic derivatives.

Syntax

```
el.elem = 'elinline'
el.name = fname
el.args{iarg} = argname
el.expr = evalexpr
el.dexpr{iarg} = devalexpr
el.complex = 'true' | 'false'
el.linear = 'true' | 'false'
```

Description

The `elinline` element declares a new function with differentiation rules in terms of other built-in functions or MATLAB functions. COMSOL Multiphysics calls the function using the MATLAB interpreter if you run COMSOL Multiphysics with MATLAB.

Declared inline functions can be used with the syntax `fname(arg1,arg2,...)` anywhere except for in other inline function definitions. The `args` field contains a list of formal parameter names which can be used in the `expr` field defining the function itself and the `dexpr` field defining derivative expressions with respect to each of the formal arguments.

The `evalexpr` and `devalexpr` expressions can contain any valid expression in the formal arguments. Global constants and variables are not available with `pi` being the only noticeable exception. Note that differentiation with respect to some formal argument can be disabled by just specifying the corresponding `devalexpr` as `'0'`.

Functions which can generate complex values from real data must have the `complex` field set to `'true'`. The `linear` property decides if the function is treated as linear when deciding whether to reassemble the Jacobian at each time step/iteration or not.

Cautionary

Note that inline functions cannot depend on other inline functions, only on built-in functions and functions defined on your MATLAB path. Inline functions can be used to override built-in functions but can never override another inline function.

Examples

Use an inline function to redefine the derivative of the `sqrt` function in such a way that the Jacobian of `sqrt(u^2+v^2)` will exist for `u=v=0`.

```
el.elem = 'elinline';
el.name = 'sqrt';
el.args = {'a'};
el.expr = 'sqrt(a)';
el.dexpr = {'1/(2*sqrt(a)+eps)'};
el.complex = 'true';
el.linear = 'false';
```

```
fem.elem = [fem.elem {el}];
```

See Also elempty

Purpose Declare interpolation functions.

Syntax

```
el.elem = 'elinterp'
el.name = fname
el.x = xcoords
el.y = ycoords
el.z = zcoords
el.data = fdata
el.mesh = tridata
el.method = 'neighbor' | 'linear' | 'piecewisecubic' | 'cubicspline'
el.extmethod = 'const' | 'interior' | 'linear' | value
```

Description The `elinterp` element declares an interpolation function based on a 1D, 2D, or 3D data set provided by the user. Interpolation functions take one, two, or three arguments, depending on the dimension of the data set.

The `xcoords`, `ycoords`, and `zcoords` parameters are cell arrays of points. Depending on the dimensions, not all fields are used. For 1D interpolation, the `fdata` parameter is a cell array of values corresponding to the points in `el.x`. For structured 2D data, the size of fdata is `length(xcoords)*length(ycoords)`, with `x` increasing fastest, and similarly in 3D, with the entries sorted in increasing order. For unstructured data, `xcoords`, `ycoords`, `czcoords`, and `fdata` must be of the same length. In this case, it is also possible to supply a triangulation of the points in the `mesh` field. `mesh` should be a cell array of strings, where each string is a whitespace-separated list of indices representing one row in the elem matrix in the function syntax for the interpolation functions (see “User-Defined Functions” on page 22 in the *COMSOL Multiphysics MATLAB Interface Guide)*.

There are four interpolation methods to choose from. Nearest neighbor and linear interpolation are available in all dimensions, while `'piecewisecubic'` and `'cubicspline'` can only be used for interpolation in 1D data sets. The difference between the latter two is, generally speaking, that `'piecewisecubic'` preserves monotonicity and does not overshoot at the cost of discontinuous second derivatives at the tying points.

There are also four extrapolation methods to choose from. The method `'const'` gives the function the same value outside boundary as on the boundary of the defined data set. The method`'interior'` continues the interpolating function outside the defined data set. It can only be used with structured evaluation. The method `'linear'` makes the function linear outside the defined data set. Both the function and the derivative are continuous on the boundary of the data set. This method can only be used in 1D with the interpolation methods `'piecewisecubic'` and `'cubicspline'`. If `extmethod` is a string containing a real number (*`value`*), this value is used outside the defined data set. For unstructured interpolation, only `'const'` and *`value`* are supported.

Cautionary

Interpolation is provided only for real numbers. To interpolate complex numbers, real and imaginary parts have to be treated separately.

Examples

Given the matrices `x` (`1`-by-`m`), `y` (`1`-by-`n`) and `F` (`m`-by-`n`), create a corresponding interpolation element declaring a function `f(x,y)`.

```
cellX = cell(1,m);
for i=1:m
  cellX{i} = num2str(x(i));
end

cellY = cell(1,n);
for i=1:n
  cellY{i} = num2str(y(i));
end

cellF = cell(1,m*n);
for i=1:m*n
  cellF{i} = num2str(F(i));
end

el.elem = 'elinterp';
el.name = 'f';
el.x = cellX;
el.y = cellY;
el.data = cellF;
el.method = 'linear';
fem.elem = [fem.elem {el}];
```

See Also

`elempty`

Purpose Define matrix inverse component variables.

Syntax

```
el.elem = 'elinv'
el.g{ig} = geomnum
el.matrixdim = mdim
el.format = 'symmetric' | 'hermitian' | 'unsymmetric'
el.basename = bname
el.postname = pname
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.matrix{eldomgrp} = matexpr
```

Description The elinv `element` defines components of the inverse of an `mdim`-by-`mdim` matrix field. The `matexpr` is a cell array of expressions which specify the source matrix in column order. If the format is `'symmetric'` or `'hermitian'`, only the upper triangle has to be given.

Matrix inverse component names, that is, the variable names defined by the element, are created by appending row and column indices to the `bname` parameter and, if the `postname` field is present, append `'_pname'`. If the format is specified as `'symmetric'`, variables are generated only for the upper triangular part of the inverse, otherwise for all components.

Examples Define variables to evaluate the Jacobian inverse components for an explicit variable transformation `[X(x,y,z), Y(x,y,z), Z(x,y,z)]`.

```
el.elem = 'elconst';
el.var = {'X','2*x','Y','y+z','Z','z'};
fem.elem = [fem.elem {el}];

clear el;
el.elem = 'elinv';
el.g = {'1'};
el.matrixdim = '3';
el.format = 'unsymmetric';
el.basename = 'd';
gd.ind = {{'1'}};
gd.matrix = {{'diff(X,x)','diff(X,y)','diff(X,z)',...
              'diff(Y,x)','diff(Y,y)','diff(Y,z)',...
              'diff(Z,x)','diff(Z,y)','diff(Z,z)'}};
el.geomdim{1} = {{},{},gd};
fem.elem = [fem.elem {el}];
```

See Also `elempty`, `elpric`

Purpose

Define irradiation variables for radiative heat transfer.

Syntax

```
el.elem = 'elirradiation'
el.g{ig} = geomnum
el.method = 'area' | 'hemicube'
el.iorder = order
el.resolution = res
el.sectors = nsectors
el.cache = 'on' | 'off'
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.name{eldomgrp} = Gname
el.geomdim{ig}{edim}.ambname{eldomgrp} = Fname
el.geomdim{ig}{edim}.expr{eldomgrp} = Jexpr
el.geomdim{ig}{edim}.opexpr{eldomgrp} = opacity
el.geomdim{ig}{edim}.dnsign{eldomgrp} = normalsign
el.geomdim{ig}{edim}.cavity{eldomgrp} = cavitylist
```

Description

The `elirradiation` element defines one variable `Gname` representing the local irradiation from other surfaces and one `Fname`, known as *ambient view factor*, representing the fraction of the field of view not covered by other surfaces. The variable names can differ between domain groups.

The irradiation at each point depends on the radiosity at all other visible surface points. An expression for the radiosity is provided in the `expr` field. For interior boundaries, it has to be made clear in which direction the radiation goes. If the field `opexpr` is present, the `opacity` expression is evaluated on adjacent subdomains, and is expected to be nonzero on exactly one. Otherwise, the `dnsign` field is expected to contain a multiplier (either `+1` or `-1`) aligning the outward normal from the geometrical down side, `[dnx, dny, dnz]`, with the desired radiation direction.

To avoid unnecessary visibility checks, the surfaces can be manually assigned to one or more cavities, each of which exchanges radiation only with other surfaces in the same cavity. For each domain group, specify to which cavities the group belongs.

There are currently two view factor evaluation methods which can be selected using the `method` field. The `'area'` method implies direct area integration using a simple quadrature rule of order `order` and no visibility checks. Different convex cavities can be held apart using the `cavity` field. The `'hemicube'` method, and its generalizations to lower dimensions, uses techniques borrowed from computer graphics to handle surfaces obstructing each other. Essentially, images of resolution `res`-by-`res` are generated from each evaluation point in 3D.

Use the `sectors` field to specify that a 2D geometry shall be interpreted as axially symmetric and to set the azimuthal resolution when evaluating view factors. The value `nsectors` is the number of sectors to a full revolution in a virtual 3D geometry created by revolving the 2D mesh about the axis.

Due to the rather complex evaluation, it is usually beneficial to store the view factors between calls. This can, however, generate a lot of data, which can potentially be a limiting factor preventing a solution on a given system. Therefore, the `cache` field can be set to `'off'`, but this increases run times considerably.

Cautionary

Radiation is currently only possible between boundaries, that is, between entities of dimension one lower than the space dimension. Also, radiation only works within one geometry.

The `elirradiation` element is available only if your license includes the Heat Transfer Module.

Examples

Compare the irradiation calculated by the hemicube algorithm with an analytical solution in a known case.

```
clear fem;
fem.geom = geomcsg({rect2(1,1,'pos',[0 -1]),...
                    rect2(1,1,'pos',[-1 0]),...
                    rect2(.8,.8,'pos',[.2 .2])});
fem.mesh = meshinit(fem);
fem.expr = {'xb',   0.2,...
            'yb',   0.2,...
            'xc',   'xb+xb*yb/(1-yb)',...
            'y1',   '(x>xc)*(xb*yb/(x-xb)+yb)+(x<=xc)',...
            'Gref', '0.5-x/(2*sqrt(x^2+y1^2))'};
fem.elem ={};

clear el
el.elem = 'elirradiation';
el.g = {'1'};
el.method = 'hemicube';
el.resolution = '512';
clear gd;
gd.ind = {{'6','7'},{'8','9'}};
gd.name = 'G';
gd.ambname = 'F_amb';
gd.expr = {'1','0'};
gd.opexpr = '1';
el.geomdim{1} = {{},gd,{}};
fem.elem = [fem.elem {el}];

fem.xmesh = meshextend(fem);
postint(fem,'abs(G-Gref)','edim',1,'dl',7)/...
  postint(fem,'abs(Gref)','edim',1,'dl',7)
```

See Also	`elempty`

Purpose Create ellipse geometry object.

Syntax

```
obj = ellip2
obj = ellip1
obj = ellip2(a,b,...)
obj = ellip1(a,b,...)
```

Description `obj = ellip2` creates a solid ellipse geometry object with center at the origin and semi-axes equal to `1`. `ellip2` is a subclass of `solid2`.

`obj = ellip2(a,b,...)` creates an ellipse object with semi-axes equal to `a` and `b`, respectively, centered at the origin. `a` and `b` are positive real scalars, or strings that evaluate to positive real scalars, given the evaluation context provided by the property `const`.

The functions `ellip2/ellip1` accept the following property/values:

TABLE 1-16: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`base`	`corner` | `center`	`center`	Positions the object either centered about `pos` or with the lower left corner of surrounding box in `pos`
`const`	cell array of strings	`{}`	Evaluation context for string inputs
`pos`	vector of reals or cell array of strings	`[0 0]`	Position of the object
`rot`	real or string	0	Rotational angle about `pos` (radians)

`obj = ellip1(...)` creates a curve circle geometry object with properties as given for the `ellip2` function. `ellip1` is a subclass of `curve2`.

Ellipse objects have the following properties:

TABLE 1-17: ELLIPSE OBJECT PROPERTIES

PROPERTY	DESCRIPTION
`a`, `b`	Semi-axes
`x`, `y`	Position of the object
`rot`	Rotational angle

In addition, all 2D geometry object properties are available. All properties can be accessed using the syntax `get(object,property)`. See `geom2` for details.

Compatibility The FEMLAB 2.3 syntax is obsolete but still supported.

Examples The commands below create an ellipse object and plot it.

```
e1 = ellip2(1,0.3,'base','center','pos',[0,0],'rot',pi/4)
get(e1,'rot')
geomplot(e1)
```

See Also `circ1`, `circ2`, `curve2`, `curve3`, `geomcsg`

Purpose

Create an ellipsoid geometry object.

Syntax

```
obj = ellipsoid3
obj = ellipsoid2
obj = ellipsoid3(a,b,c)
obj = ellipsoid2(a,b,c)
obj = ellipsoid3(a,b,c,...)
obj = ellipsoid2(a,b,c,...)
```

Description

`obj = ellipsoid3` creates a solid ellipsoid geometry object with center at the origin and semi-axes equal to `1`. `ellipsoid3` is a subclass of `solid3`.

`obj = ellipsoid3(a,b,c,...)` creates a solid ellipsoid object with semi-axes `a`, `b`, and `c`. `a`, `b`, and `c` are positive real scalars, or strings that evaluate to positive real scalars, given the evaluation context provided by the property `Const`.

The functions `ellipsoid3/ellipsoid2` accept the following property/values:

TABLE 1-18: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Axis`	vector of reals or cell array of strings	`[0 0]`	Local z-axis of the object
`Const`	cell array of strings	`{}`	Evaluation context for string inputs
`Pos`	vector of reals or cell array of strings	`[0 0]`	Position of the object
`Rot`	real or string	0	Rotational angle about `Axis` (radians)

`Axis` sets the local *z*-axis, stated either as a directional vector of length 3, or as a 1-by-2 vector of spherical coordinates. `Axis` is a vector of real scalars, or a cell array of strings that evaluate to real scalars, given the evaluation context provided by the property `Const`. See `gencyl3` for more information on `Axis`.

`Pos` sets the center of the object. `Pos` is a vector of real scalars, or a cell array of strings that evaluate to real scalars, given the evaluation context provided by the property `Const`.

`Rot` is an intrinsic rotational angle for the object about its local *z*-axis provided by the property Axis. `Rot` is a real scalar, or a string that evaluates to a real scalar given the evaluation context provided by the property `Const`. The angle is assumed to be in radians if it is numeric, and in degrees if it is a string.

`obj = ellipsoid2(...)` creates a surface ellipsoid object with the properties as given for the `ellipsoid3` function. `ellipsoid2` is a subclass of `face3`.

Ellipsoid objects have the following properties:

TABLE 1-19: ELLIPSOID OBJECT PROPERTIES

PROPERTY	DESCRIPTION
a, b, c	Semi-axes
x, y, z, xyz	Position of the object. Components and vector forms
ax2	Rotational angle of symmetry axis
ax3	Axis of symmetry
rot	Rotational angle

In addition, all 3D geometry object properties are available. All properties can be accessed using the syntax `get(object,property)`. See `geom3` for details.

Examples

The following commands create a surface and solid ellipsoid object, where the position and semi-axis are defined in the two alternative ways.

```
e2 = ellipsoid2(1,1,1,'pos',[0 1 0],'axis',[0 0 1],'rot',0)
e3 = ellipsoid3(12,10,8)
```

Compatibility

The representation of the ellipsoid objects has been changed. The FEMLAB 2.3 syntax is obsolete but still supported. If you use the old syntax or open 2.3 models containing ellipsoids they are converted to general face or solid objects.

See Also

`face3`, `geom0, geom1, geom2, geom3`, `sphere3, sphere2`

Purpose

Fast evaluation of predefined convolution integrals

Syntax

```
el.elem = 'elkernel'
el.g{ig} = geomnum
el.name = opname | opnamelist
el.kernel = 'unit' | 'helmholtz2D' | 'helmholtz2Dinf' | 'helmholtz3D' |
  'helmholtz3Dinf' | 'helmholtz2Daxi' | 'helmholtz2Daxiinf' |
  'maxwell3Dinf' | 'maxwellTEinf' | 'maxwellTMinf' | 'maxwellTEaxiinf'
  | 'maxwellTMaxiinf'

el.frame = srcframe
el.iorder = gporder
el.k = wavenumber
el.symflags{idim} = '-1' | '0' | '1'

el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.srcn{idim}{eldomgrp} = nexpr
el.geomdim{ig}{edim}.srcu{eldomgrp} = uexpr
el.geomdim{ig}{edim}.srcnux{eldomgrp} = nuxexpr
el.geomdim{ig}{edim}.srcnxe{eldomgrp} = nxeexpr
el.geomdim{ig}{edim}.srcnxcurle{eldomgrp} = nxculreexpr
```

Description

The `elkernel` element is a wrapper which defines an operator that represents a predefined convolution integral. You can choose between plain integration of an arbitrary expression, Helmholtz-Kirchhoff integral solutions to Helmholtz' equation in various dimensions and a number of instances the Stratton-Chu formula for far-field evaluation of electromagnetic fields under various conditions. For Helmholtz equation, both the complete integral solution and the far-field limiting case are provided.

The type of integral is specified in the `kernel` field where `'unit'` gives you an operator which takes an expression as only argument, while the rest define operators which are functions of the evaluation point coordinates. The 3D Stratton-Chu formula requires a list of three operator names, one for each field component, while TM waves require two operator names and the remaining only one.

The integrals are evaluated on the mesh with the integration order specified in `iorder` and using the source coordinate names corresponding to the given frame. Both Helmholtz-Kirchhoff and Stratton-Chu integral evaluation require a free-space wave number, which is expected to be a global constant. The former, in addition, needs expressions for the normal vector, solution value and its normal derivative on the source domains, specified in `srcn`, `srcu`, and `srcnux`, respectively. The Stratton-Chu formula requires the cross products of source normal with electric field and source normal with curl of electric field, specified in `srcnxe` and `srcnxcurle`, respectively.

The source domains are expected to form a closed surface containing all sources and inhomogenities, and the specified normal vector must be facing into the domain enclosed by this surface. When exploiting symmetry to model only 1/2, 1/4, or 1/8 of the actual geometry, a closed surface can be recovered using the symmetry flags. In the `symflags` field, -1 in position `idim` is interpreted as antisymmetry with respect to the coordinate plane normal to the `idim`-axis, while +1 means that the plane is a symmetry plane. If the `symflags` field is not given, all entries are considered to be 0, which signifies that the model is neither symmetric nor antisymmetric.

Cautionary

The operators defined by `elkernel` are primarily intended for postprocessing and therefore do not define any Jacobian contributions.

Examples

The acoustic field from a baffled piston oscillating with specified velocity normally to an infinite rigid plane can be evaluated explicitly using the `helmholtz2Daxi` kernel with `srcu` set to zero since the terms proportional to *u* cancel out anyway.

```
fem.geom = circ2*rect2;
fem.mesh = meshinit(fem);

fem.expr = {'SPL' '10*log10(0.5*abs(p(x,y))^2/2e-5^2)'};

clear el;
el.elem = 'elkernel';
el.g = {'1'};
el.kernel = 'helmholtz2Daxiinf';
el.name = 'p';
el.iorder = '20';
el.k = '100';
clear src11
src11.ind = {{'2'}};
src11.srcn = {'0','-1'};
src11.srcu = '0';
src11.srcnux = {'1'};
el.geomdim{1} = {{},src11};
fem.elem = {el};

fem.xmesh = meshextend(fem);
fem.sol = asseminit(fem);

postcrossplot(fem,1,3,'lindata','SPL','linxdata',...
  '180/pi*atan2(y,x)','refine',10);
```

Purpose

Define extrusion map operators.

Syntax

```
el.elem = 'elmapextr'
el.g{ig} = geomnum
el.opname{iop} = opname
el.flagname{iop} = flagname
el.extttol = tol
el.usenan = 'true' | 'false'
el.map{imap} = linmap | genmap | unitmap
el.srcmap{iop} = imap
el.dstmap{iop} = imap
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.src{iop} = eldomgrplist

linmap.type = 'linear'
linmap.sg = srcig
linmap.sv{ivtx} = srcvtx
linmap.sframe = srcframe
linmap.dg = dstig
linmap.dv{ivtx} = dstvtx
linmap.dframe = dstframe

genmap.type = 'local'
genmap.expr{idim} = transexpr
genmap.frame = frame

unitmap.type = 'unit'
unitmap.frame = frame
```

Description

The `elmapextr` element defines extrusion map operators which can be used at any location where the source and destination transformations make sense. Each map operator takes its values from the source domain groups listed in the corresponding `el.geomdim{ig}{edim}.src{iop}` field.

For each operator, transformations are specified in the `srcmap` and `dstmap` fields in the form of indices into the `map` field that consists of a list of transformation specifications. The available transformation types are `'linear'`, `'local'`, and `'unit'`. They are described in detail under `elcplextr` on page 57.

For each operator name a global variable `flagname{iop}` is also defined, which evaluates to 1 for all destination points where the map operation can find a corresponding source point, otherwise it evaluates to 0. If the `flagname` field is not given, the flag variables will be given the same name as the corresponding operator. Therefore, statements like `if(my_map,my_map(u),0)` make perfect sense.

Cautionary

Parameter or time dependency in the source transformation is not properly detected by the solvers, which means that the source transformation will not be updated between parameter or time steps in that case. Solution dependencies in the transformation are properly detected but do not give any Jacobian contributions from the transformation.

Examples

Calculate the first ten eigenvalues of a 3-by-2 rectangle with periodic boundary conditions both left-right and top-bottom. Different map types are used. Note that this is the same example as used under `elcplextr`.

```
fem.geom = rect2(3,2);
fem.mesh = meshinit(fem,'hmax',0.05);
fem.equ.c = 1;
fem.equ.da = 1;
fem.bnd.ind = [0 1 2 0];
fem.bnd.constr = {'left2right(u)-u','lower2upper(u)-u'};
fem.elem = {};

el.elem = 'elmapextr';
el.g = {'1'};
el.opname = {'left2right','lower2upper'};

clear map1;
map1.type = 'unit';

clear map2;
map2.type = 'linear';
map2.sg = '1';
map2.sv = {'2','3'};
map2.dg = '1';
map2.dv = {'1','4'};

clear map3;
map3.type = 'local';
map3.expr = {'x'};

el.map = {map1 map2 map3};

el.srcmap = {'1','3'};
el.dstmap = {'2','3'};

clear src;
src.ind = {{'1'},{'4'}};
src.src = {{'2'},{'1'}};
el.geomdim{1} = {{},src,{}};

fem.elem = [fem.elem {el}];
fem.xmesh = meshextend(fem);
fem.sol = femeig(fem,'neigs',10,'shift',1);
postplot(fem,'tridata','u','triz','u','refine',3,'solnum',8);
```

See Also `elcplextr`

Purpose Define mesh variables and a frame.

Syntax

```
el.elem = 'elmesh'
el.g{ig} = geomnum
el.frame = frame
el.xvars = 'on' | 'off'
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.sizename{eldomgrp} = sname
el.geomdim{ig}{edim}.qualname{eldomgrp} = qname
el.geomdim{ig}{edim}.dvolname{eldomgrp} = dvolname
el.geomdim{ig}{edim}.meshtypename{eldomgrp} = meshtypename
el.geomdim{ig}{edim}.meshelemname{eldomgrp} = meshelemname
el.geomdim{ig}{edim}.sshape{eldomgrp} = sshape
sshape{2*i-1} = bmtypename
sshape{2*i} = params
params.type = 'fixed' | 'moving_abs' | 'moving_rel' | 'moving_expr'
params.sorder = sorder
params.sdimdofs{idim} = dofname
params.sdimexprs{idim} = expr
params.refframe = refframename
```

Description Concerning the syntax of the `ind` field, see `elempty`. For each domain group, the `elmesh` element defines the variable names `sname`, `qname`, `dvolname`, `meshtypename`, and `meshelemname`, which evaluate to local mesh element size, element quality, element volume, mesh type index, and mesh element number, respectively. Also, if `xvars='on'` the space coordinates, the space coordinate's reference time derivative, and the normal vector are defined. These variable names are derived from the space coordinates of the frame `frame`. The type of frame is determined by `sshape`.

Examples By default, the mesh size variable `h` is available only on the top dimension. If evaluated on a boundary, `h` returns the size of the adjacent subdomain element. An additional `elmesh` element can be used to define `h` on the boundary to represent the size of the boundary element.

```
el.elem = 'elmesh';
el.g = {'1'};
el.frame = 'xy'
el.xvars = 'off'
gd.sizename = 'h';
el.geomdim{1} = {{},gd,{}};
fem.elem = [fem.elem {el}];
```

See Also `elempty`

Purpose

Define global scalar dependent variables and equations.

Syntax

```
el.elem = 'elode'
el.dim{idim} = depvarname
el.f{idim} = rexpr
el.weak{iweak} = wexpr
```

Description

The `elode` element adds globally available scalar dependent variables (named degrees of freedom) and corresponding equations. The `dim` field lists unique variable names which are allocated on a fictitious 0D geometry and made available throughout the model. The optional `f` field has the same number of entries as the `dim` field, while the `weak` field, if present, can have any number of entries. These fields define scalar equations on the form `rexpr=0` and `wexpr=0`, respectively. The `f` field requires the presence of a `dim` field. See further "fem.ode—Global Variables and Equations" on page 45 of the *COMSOL Multiphysics MATLAB Interface Guide*.

The fictitious geometry mentioned above is for most purposes equivalent to a real 0D geometry with one domain, a point. This geometry can be explicitly referenced using geometry index `0`. Therefore, the expressions `rexpr` and `wexpr` can contain variables which are globally available or explicitly available on domain `1` of geometry `0`.

For further examples of use of scalar dependent variables and equations, see the *COMSOL Multiphysics User's Guide*.

Cautionary

Though the elode element applies to the ever-present fictitious geometry `0`, a real geometry also has to be defined for the solvers to work. Note also that `elode` can be used to define global weak contributions to existing equations. That is, the `weak` field may be used without the presence of a `dim` field.

Examples

Solve a simple scalar wave equation:

```
clear fem
fem.geom = geom0(zeros(0,1));
fem.mesh = meshinit(fem);

clear el
el.elem = 'elode';
el.dim = {'u'};
el.f = {'utt+u'};
fem.elem = {el};

clear elinit;
elinit.elem = 'elconst';
elinit.var = {'u','0','ut','1'};
fem.eleminit = {elinit};

fem.xmesh = meshextend(fem);
fem.sol = femtime(fem,'tlist',linspace(0,4*pi,100),...
                        'maxorder',2,'rtol',1e-8,'atol',1e-8);
postcrossplot(fem,0,1,'pointdata','u')
```

See Also `elempty`, `eleqc`, `eleqw`, `elshape`

Purpose

Define general pointwise constraints.

Syntax

```
el.elem = 'elpconstr'
el.g{ig} = geomnum
el.nname = Nname
el.nfname = NFname
el.mname = Mname

el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.constr{eldomgrp}{ic} = constrexpr
el.geomdim{ig}{edim}.constrf{eldomgrp}{ic} = constrfexpr
el.geomdim{ig}{edim}.cpoints{eldomgrp}{ic} = cpind
```

Description

The `elpconstr` element adds a set of pointwise constraints of type `constrexpr=0`. The `constr` field has the same syntax as the `fem.bnd.constr` field. See further the chapter "Specifying a Model" on page 3 of the *COMSOL Multiphysics MATLAB Interface Guide*. The optional `constrf` field controls the way constraint forces are applied. You enter an expression such that its Jacobian with respect to the test functions decides on which degrees of freedom the reaction force is applied for each constraint. If `constrf` is omitted, constraints are *ideal*, which corresponds to setting `constrf` to `test(constrexpr)`.

The constraint and constraint force Jacobians are by default assembled to matrices called `NT` and `NF`, and the constraint residual is called `M`. This can be changed by assigning different names to the optional `nname`, `nfname` and `mname` fields. Only certain names are recognized by functions like `assemble`, though, see page 27.

The constraints are enforced at the same local coordinates in all elements in one domain group. The constraint point pattern is specified as a pattern index in the `cpoints` field. Indices refer to patterns defined by an `elepspec` element.

Compared to the `elcconstr` element, the `elpconstr` can implement a wider range of constraints, as a correct constraint Jacobian is always calculated on the fly. This is in contrast to the user-specified Jacobian matrix `h`, used in `fem.bnd.h` and the `elcconstr` element.

Examples

Solve a 1D biharmonic equation (related to Euler beams) with constraints on both value and normal derivative at the endpoints.

```
clear fem;
fem.geom = solid1([0,1]);
fem.mesh = meshinit(fem);
fem.shape = 'shherm(1,3,''u'')';
fem.form = 'weak';
fem.equ.weak = 'uxx_test*uxx';
fem.bnd.dim = {'u'};
fem.bnd.cporder = 1;
fem.elem = {};

clear el;
el.elem = 'elpconstr';
el.g = {'1'};
clear gd;
gd.ind = {{'1'},{'2'}};
gd.constr = {{'-u','1-ux'},{'-u','1+ux'}};
gd.cpoints = {{'1'},{'1'}};
el.geomdim{1} = {gd,{}};
fem.elem = [fem.elem {el}];

fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem);
postplot(fem,'liny','u');
```

See Also `elempty`, `elcconstr`, `elcurlconstr`, `elepspec`

Purpose

Declare piecewise functions.

Syntax

```
el.elem = 'elpiecewise'
el.subtype = 'poly' | 'exppoly' | 'general'
el.name = fname
el.args = argname
el.intervals{ibnd} = interval_bound
el.expr{iexpr} = poly_spec | expr
el.extmethod = 'interior' | 'const' | double_value | 'none'
el.smoothzone = double_value
el.smoothorder = '0' | '1' | '2'
el.complex = 'true' | 'false'
el.linear = 'true' | 'false'

poly_spec{2*ipow} = exponent
poly_spec{2*ipow+1} = coefficent
```

Description

The `elpiecewise` element declares a function `fname` which is of type `subtype` in each interval, the boundaries of which are given in the `intervals` field. The polynomials or general expressions are given for each subinterval in the `expr` field, which contains one pair of polynomial exponent and coefficient or one expression for each interval. Derivatives are calculated by automatic symbolic differentiation. Outside the intervals, the value of the function is either extrapolated, taken from the nearest interval boundary, or given a fixed number, according to the `extmethod` field. `'none'` indicates that extrapolation is deactivated and results in errors or `NaN` values for out-of-range values, depending on how and where the element is used.

Because the given expressions can be discontinuous at the interval boundaries, `elpiecewise` includes an optional smoothing option. If given, the `smoothzone` field specifies a relative size of the smoothing zone, interpreted as the fraction of each interval length which should be smoothed at the intersections between intervals. The `smoothorder` field gives the number of continuos derivatives that must exist at the boundary between smoothing zone and interval.

Functions which can generate complex values from real data must have the `complex` field set to `'true'`. The `linear` property decides if the function is treated as linear when deciding whether to reassemble the Jacobian at each time step/iteration or not.

Examples

Setup a piecewise function element of the polynomials $0.2x^{-6} + 5.1x + 0.05x^6$ and $60x$, defined from 1.7 to 4 and 4 to 5.2, respectively, with continuos first derivatives at the intersection:

```
  el.elem = 'elpiecewise';
  el.name = 'myfun';
  el.subtype = 'poly';
  el.expr = {{'-6' '0.2' '1' '5.1' '6' '0.05'} {'1' '60'}}
  el.intervals = {'1.7' '4' '5.2'}
  el.smoothzone = '0.1';
  el.smoothorder = '1';
  fem.elem = [fem.elem {el}];
```

See Also `elempty`, `elinterp`, `elinline`

Purpose

Define plastic strain variables.

Syntax

```
el.elem = 'elplastic'
el.g{ig} = geomnum
el.vars{ivar} = varname
el.varsToCache = cachevarlist
el.varPairsToGpProcess{2*igpvar-1} = gpvarname
el.varPairsToGpProcess{2*igpvar} = gpvarexpr
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.Yield{eldomgrp} = yieldexpr
el.geomdim{ig}{edim}.EffStress{eldomgrp} = effstressexpr
el.geomdim{ig}{edim}.G{eldomgrp}{ivar} = gexpr
el.geomdim{ig}{edim}.gporder{eldomgrp} = iorder
```

Description

The `elplastic` element defines the plastic strain variable names specified in the `vars` field. For the syntax of the `ind` field, see `elempty`. The yield function is `effstressexpr−yieldexpr`. For each strain variable in `vars`, the right-hand side of the corresponding rate equation is `gexpr` times the plastic multiplier, `lambda`. See "Continuum Application Modes", section "Theory Background", in the *Structural Mechanics Module User's Guide*. The `gporder` field specifies a quadrature rule order, which should preferably be the same as the order used in the assembly of the main equation.

The `varsToCache` field contains a list of variables that can be assumed not to depend explicitly on the plastic strains. By specifying variable names representing complicated material property expressions or interpolated data independent of the plastic strains, it is possible to avoid repeated evaluation in the inner, plastic, loop.

In addition to the plastic strain variables, the `varPairsToGpProcess` field defines a number of postprocessing variable-expression pairs, which, when evaluated, are linearly extrapolated from the integration points. Use this feature, for example, to avoid problems with nonconvergent plastic strains at sharp geometry corners.

For each variable and integration point, the `elplastic` element declares an additional degree of freedom, which appears in the solution vector. However, consider these degrees of freedom to be internal data of the `elplastic` element. They are updated only by a special procedure in the nonlinear solver.

Cautionary

Note that some of the field names are mixed case, and case matters. Also, the domain-dependent fields do not accept empty entries for any domain group.

The `elplastic` element is available only if your license includes the Structural Mechanics Module or the MEMS Module.

Examples

By faking a single plastic strain variable, the `elplastic` element can be used also as a pure postprocessing element to define variables extrapolated from the integration points. The following example works for a 2D plane strain model.

```
el.elem = 'elplastic';
el.g = {'1'};
el.vars = {'foo'};
el.varPairsToGpProcess = {'ex','ex_smpn',...
                          'ey','ey_smpn',...
                          'exy','exy_smpn'};
gd.ind = {{'1'}};
gd.Yield = {'0'};
gd.EffStress = {'0'};
gd.G = {{'0'}};
gd.gporder = {'2'};
el.geomdim{1} = {{},{},gd};
fem.elem = [fem.elem {el}];
```

See Also

`elempty`

Purpose

Define variables which evaluate principal values and vector components.

Syntax

```
el.elem = 'elpric'
el.g{ig} = geomnum
el.basename = bname
el.postname = pname
el.sdim{idim} = dimname
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.tensor{eldomgrp} = matexpr
```

Description

The `elpric` element evaluates eigenvalues and eigenvectors of a `3`-by-`3` real symmetric matrix. The `tensor` field always has six components specifying the upper triangle of the source matrix as a cell array of expressions in column order. The `basename` field is compulsory and specifies a single name from which all output variable names are derived. The output variables are defined wherever the `tensor` field is nonempty.

Eigenvalue variable names are created by appending numbers `1` to `3` to `bname` and, if `postname` is present, append `'_pname'`. Eigenvector component names are then created by inserting the space variable names given in `sdim` directly after the component number. Eigenvalues are sorted in decreasing order.

Cautionary

No Jacobian contribution is calculated even if the `tensor` expressions contain dependent variables. The reason is the condition that the eigenvalues are sorted, which makes the eigenvector components discontinuous functions of the input matrix components.

Examples

Define postprocessing variables for principal strains and directions, given strain components `[ex,ey,ez,exy,exz,eyz]`.

```
el.elem = 'elpric';
el.g = {'1'};
el.basename = 'e';
el.sdim = {'x','y','z'};
gd.ind = {{'1'}};
gd.tensor = {{'ex','exy','ey','exz','eyz','ez'}};
el.geomdim{1} = {{},{},{},gd};
fem.elem = [fem.elem {el}];
```

See Also

`elempty`, `elinv`

Purpose

Define pointwise constraints controlled by shape functions.

Syntax

```
el.elem = 'elsconstr'
el.g{ig} = geomnum
el.nname = Nname
el.nfname = NFname
el.mname = Mname

el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.shelem = meshcases | shapelist

meshcases.default = shapelist;
meshcases.case{elmcase} = shapelist;
meshcases.mind{elmcase} = caselist;

shapelist{eldomgrp}{ishape}{3*i-2} = bmtypename
shapelist{eldomgrp}{ishape}{3*i-1} = shapename
shapelist{eldomgrp}{ishape}{3*i} = shapeparams

el.geomdim{ig}{edim}.constr{eldomgrp}{ic}{j} = constrexpr
el.geomdim{ig}{edim}.constrf{eldomgrp}{ic}{j} = constrfexpr
el.geomdim{ig}{edim}.cshape{eldomgrp}{ic} = ishape
```

Description

The `elsconstr` element defines pointwise constraints where the type of constraint is determined by a shape function object. The idea is that the constraint points are selected to be appropriate for variables having the corresponding shape function. For the syntax of the `ind` field, see `elempty`. For the syntax of the `shelem` field, see `elshape`. A difference to the syntax in `elshape` is that the cell array `shapelist` has three levels instead of two.

The expressions that are used to formulate the constraint are given in the `constr` field, and the index of the corresponding shape function object is given in the `cshape` field. More precisely, in element geometry `ig`, dimension `edim`, and element domain group `eldomgrp`, constraint number `ic` is defined by the shape function object with index `ishape=el.geomdim{ig}{edim}.cshape{eldomgrp}{ic}`. The expressions needed to formulate this constraint is given by the cell array `el.geomdim{ig}{edim}.constr{eldomgrp}{ic}`. The number of expressions `ne` in this cell array depends on the shape function object.

The optional `constrf` field controls the way constraint forces are applied. You enter an expression such that its Jacobian with respect to the test functions decides on which degrees of freedom the reaction force is applied for each constraint. If `constrf` is omitted, constraints are *ideal*, which corresponds to setting the components of `constrf` to `test(constrexpr)`.

The constraint and constraint force Jacobians are by default assembled to matrices called NT and NF, and the constraint residual is called M. This can be changed by assigning different names to the optional nname, nfname and mname fields. Only certain names are recognized by functions like assemble, though, see page 27.

The elsconstr constraint element is only implemented for the shape functions shlag, shcurl, and shdiv. For shlag, the number of expressions ne=1, and this expression is constrained to be zero in the node points of the shlag object. For shcurl, the number of expressions ne=sdim, and these expressions are considered as components of a vector. The tangential component of this vector is constrained to be zero in the node points for the shcurl shape function. For shdiv, the number of expressions ne=sdim, and these expressions are considered as components of a vector. The normal component of this vector is constrained to be zero in the node points for the shdiv shape function.

Example

Impose a constraint on a vector field E represented using shcurl shape functions of order 2. The constraint is that the tangential component of E-(2,3) is zero.

```
clear fem;
fem.geom = circ2;
fem.mesh = meshinit(fem);
fem.shape = 'shcurl(2,''E'')';
fem.dim = {'Ex' 'Ey'};
fem.equ.weak = '-(Ex*Ex_test+Ey*Ey_test+dExy_test*dExy)';

clear el gd;
el.elem = 'elsconstr';
el.g = {'1'};
gd.ind = {{'1','2','3','4'}};
gd.shelem{1}{1} = ...
  {'s(1)','shcurl',struct('fieldname','E','order','2')};
gd.constr{1}{1} = {'Ex-2','Ey-3'};
gd.cshape{1}{1} = '1';
el.geomdim{1} = {{},gd,{}};
fem.elem = {el};

fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem);
postarrow(fem,{'Ex' 'Ey'});
```

See Also

elempty, elpconstr, elshape, shdiv, shlag, shcurl

Purpose

Define dependent variables and select shape functions.

Syntax

```
el.elem = 'elshape'
el.g{ig} = geomnum
el.tvars = 'on' | 'off'
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.shelem = meshcases | shapelist

meshcases.default = shapelist;
meshcases.case{elmcase} = shapelist;
meshcases.mind{elmcase} = caselist;

shapelist{eldomgrp}{3*ishape-2} = bmtypename
shapelist{eldomgrp}{3*ishape-1} = shapename
shapelist{eldomgrp}{3*ishape} = shapeparams
```

Description

The `elshape` element is responsible for allocating degrees of freedom and defining dependent variables. For the syntax of the `ind` field, see `elempty`. The `tvars` field turns the generation of time derivative variables on or off (default on).

The `shelem` field has a rather complicated syntax. If no alternate mesh cases are defined, it is a cell array which for each domain group contains a cell array of triplets `bmtypename`–`shapename`–`shapeparams`. The string `bmtypename` is a unique identifier for a basic mesh element shape with certain additional properties, see the table below.

TABLE 1-20: BASIC MESH ELEMENT TYPE IDENTIFIERS FOR ELEMENT TYPES GENERATED IN MESHES

NAME	DESCRIPTION
ls(0)	0D simplex (all elements are equivalent in 0D)
s(1)	1D simplex, higher-order shape generated on boundaries in 2D and 3D
ls(1)	1D simplex, linear shape generated in 1D
s(2)	2D simplex (triangle), higher-order shape generated on boundaries in 3D and in a layer closest to boundaries in 2D
ls(2)	2D simplex (triangle), linear shape generated in the inner of 2D domains
b(2)	2D brick (quadrilateral, quad), higher-order shape generated on boundaries in 3D and in a layer closest to boundaries in 2D
lb(2)	2D brick (quadrilateral, quad), bilinear shape generated in the inner of 2D domains
s(3)	3D simplex (tetrahedron), higher-order shape generated in a layer closest to boundary surfaces and free edges

TABLE 1-20: BASIC MESH ELEMENT TYPE IDENTIFIERS FOR ELEMENT TYPES GENERATED IN MESHES

NAME	DESCRIPTION
ls(3)	3D simplex (tetrahedron), linear shape generated away from boundaries and edges in 3D
b(3)	3D brick (hexahedron, hex), higher-order shape generated in a layer closest to boundary surfaces and free edges
lb(3)	3D brick (hexahedron, hex), trilinear shape generated away from boundaries and edges in 3D
prism	3D prism (pentahedron, wedge), higher-order shape generated in a layer closest to boundary surfaces and free edges
lprism	3D prism (pentahedron, wedge), bilinear shape generated away from boundaries and edges in 3D

The `shapename` identifier selects a shape function for the base mesh type, and the format of the `shapeparams` parameter in turn depends on the particular shape function. Typically, the shape function expects a struct with fields specifying dependent variable name and order.

If there are multiple mesh cases present in the model, the `shelem` field is a struct with fields `default`, `case`, and `mind`. The `default` field has the same syntax as described above for the `shelem` field itself. Multiple alternate cases which use the same shape functions can be grouped together using the `mind` field. This field is a cell array containing groups of mesh case numbers, each group, `caselist`, given as a cell array. For each element mesh case group, `elmcase`, an alternate shape list is given in the `case` field.

Cautionary

Multiple shape functions can be specified by the same `elshape` element simply by repeating a basic mesh type name with different shape function and/or parameters in the `shelem` field. This means that there also has to be some conflict resolution. When multiple shape functions specify the same dependent variable name, the one with the highest interpolation order for the basic field prevails.

The field variables defined on a given domain are the union of the variables defined in the default case by shape functions on all basic mesh element types on that domain. This means that variables can at times be missing on certain mesh element types or for certain mesh cases.

Examples

Add a dependent variable `lm` on the boundary that can be used as a Lagrange multiplier in a weak constraint.

```
clear fem;
```

```
fem.geom = circ2;
fem.mesh = meshinit(fem);
fem.shape = 2;
fem.equ.c = 1; fem.equ.f = 1;
fem.bnd.weak = 'lm_test*u+lm*u_test';
fem.elem = {};

clear el;
el.elem = 'elshape';
el.g = {'1'};
el.tvars = 'off';
gd.ind = {{'1','2','3','4'}};
gd.shelem = ...
  {{'s(1)','shlag',struct('basename','lm','order','2')}};
el.geomdim{1} = {{},gd,{}};
fem.elem = [fem.elem {el}];

fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem);
postplot(fem,'tridata','u','triz','u');
```

See Also elempty, sharg_2_5, shbub, shdens, shdiv, shgp, shlag, shcurl

Purpose Create a linear flat faceted shell element.

Syntax

```
el.elem = 'elshell_arg2'
el.g{ig} = geomnum
el.dim = depvarnames
el.equation = equation type
el.omega = frequency
el.postname = postfix
el.geomdim{ig}{3}.ind{eldomgrp} = domainlist
el.geomdim{ig}{3}.E{eldomgrp} = E_expr
el.geomdim{ig}{3}.nu{eldomgrp} = nu_expr
el.geomdim{ig}{3}.rho{eldomgrp} = rho_expr
el.geomdim{ig}{3}.thickness{eldomgrp} = th_expr
el.geomdim{ig}{3}.height{eldomgrp} = height_expr
el.geomdim{ig}{3}.alphadM{eldomgrp} = alpha_expr
el.geomdim{ig}{3}.betadK{eldomgrp} = beta_expr
el.geomdim{ig}{3}.xlocalx{eldomgrp} = xlx_expr
el.geomdim{ig}{3}.xlocaly{eldomgrp} = xly_expr
el.geomdim{ig}{3}.xlocalz{eldomgrp} = xlz_expr
el.geomdim{ig}{3}.nsidex{eldomgrp} = nx_expr
el.geomdim{ig}{3}.nsidey{eldomgrp} = ny_expr
el.geomdim{ig}{3}.nsidez{eldomgrp} = nz_expr
```

Description The `elshell_arg2` element describes a linear Mindlin theory shell made up of essentially constant-strain triangles with added drilling rotations. The element lives on a 2D surface embedded in a 3D geometry. Its material properties, constraints and loads are specified directly in the element syntax structure.

An `elshell_arg2` element implements tasks which are handled by an `eleqc` or `eleqw` element when using the standard syntax. That is, it directly assembles contributions to the stiffness and mass matrices and to the residual vector. In addition, it defines a number of postprocessing variables. The shell element structure contains global properties, common to the entire shell, as well as local material properties on the boundary level. Note that the shell exists only on the boundary level and below.

Note: The `elshell_arg2` element requires a triangular mesh and will not work with a quadrilateral mesh.

Global properties: The `equation` field specifies whether to treat the problem as stationary, time harmonic or time-dependent. The three displacement and three rotation field variable names must be specified in the `dim` field, for example,

```
e.dim = {'u','v','w','thx','thy','thz'}
```

The displacement fields are most easily defined using 6 separate `shlag` objects of order 1.

TABLE 1-21: GLOBAL PROPERTIES OF THE ELSHELL_ARG2 SHELL ELEMENT STRUCTURE

FIELD	MEANING	SYNTAX	DEFAULT VALUE
`elem`	Shell element name	`elshell_arg2`	-
`g`	Geometry index	scalar number	1
`dim`	Field variable names for displacements and rotations	1-by-6 cell vector of strings	-
`equation`	Affects how matrices are assembled	`static` | `freq` | `time` | `eigen`	
`omega`	Frequency for the `freq` equation	string expression	0
`postname`	Name that is appended to postprocessing variables	string	empty string

Material, loads and constraints: The shell described by the `elshell_arg2` element can be considered a collection of discrete, homogeneous, flat triangles. The material properties, including damping factors as well as the element thickness, are taken to be constant within any triangle. The syntax of the material properties, loads and constraints is analogous to the syntax of the coefficient form *level 1 coefficients*. See further the chapter "Specifying a Model" on page 3 of the *COMSOL Multiphysics MATLAB Interface Guide*. However note that all values as well as expressions must be strings.

TABLE 1-22: BND LEVEL PROPERTIES IN THE ELSHELL_ARG2 SHELL ELEMENT STRUCTURE

FIELD	MEANING	SYNTAX
`E`	Elasticity modulus / Young's modulus	level 1 coefficient
`nu`	Poisson's ratio	level 1 coefficient
`rho`	Density	level 1 coefficient
`thickness`	Shell thickness	level 1 coefficient
`height`	Postprocessing level	level 1 coefficient
`alphadM`	Mass damping coefficient (submodes `time` and `freq` only)	level 1 coefficient
`betadK`	Stiffness damping coefficient (submodes `time` and `freq` only)	level 1 coefficient

TABLE 1-22: BND LEVEL PROPERTIES IN THE ELSHELL_ARG2 SHELL ELEMENT STRUCTURE

FIELD	MEANING	SYNTAX
`xlocalx`, `xlocaly`, `xlocalz`	Vector, whose projection on the shell defines the local x direction	level 1 coefficient
`nsidex`, `nsidey`, `nsidez`	Direction vector which defines the "up" side of the shell	level 1 coefficient

Postprocessing variables: The postprocessing variables defined by the `elshell_arg2` element have standard names that do not depend on the names of the space variables given in `fem.sdim`. The `postname` property of the shell element structure, if not the empty string, is appended to all postprocessing variables. For example, the direct x strain will be referenced as `exs` or `exs_postname`, depending on the value of the `postname` field.

TABLE 1-23: POSTPROCESSING VARIABLES DEFINED BY THE ELSHELL_ARG2 ELEMENT

VARIABLE	MEANING
`exs`, `eys`, `ezs`, `exys`, `exzs`, `eyzs`	Strain tensor components in global coordinates
`exls`, `eyls`, `ezls`, `exyls`, `exzls`, `eyzls`	Strain tensor components in local coordinates
`Nxls`, `Nyls`, `Nxyls`	In-plane forces in local coordinates
`Qxls`, `Qyls`	Out-of-plane forces in local coordinates
`Mxls`, `Myls`, `Mxyls`	In-plane moments in local coordinates
`exlxs`, `exlys`, `exlzs`	Local system x-axis expressed in global coordinates
`eylxs`, `eylys`, `eylzs`	Local system y-axis expressed in global coordinates

Theory: The `elshell_arg2` shell element is a combination of an isotropic version of the TRIC element proposed by Argyris and others (Ref. 1) and the constant strain triangle with drilling rotations due to Allman (Ref. 2). As such, the element is essentially a constant strain triangle whose displacement field vary linearly in the direction tangential to each edge, and as a restricted third order polynomial in the normal direction.

The material properties are considered to be constant within any triangle, and therefore symbolic integration can be used to describe an element stiffness matrix and a consistent element mass matrix in terms of element geometry and material data.

Cautionary

The `elshell_ar2` shell element is not multiphysics enabled. This means that there will be no contributions to the exact Jacobian from solution-dependent material data.

The `elshell_arg2` element is available only if your license includes the Structural Mechanics Module.

Compatibility

COMSOL Multiphysics 3.2: The `tdim` field and wave extension in the time-dependent case are no longer used.

See Also

`elempty`

References

[1] J. Argyris, L. Tenek, and L. Olofsson, “TRIC: a simple but sophisticated 3-node triangular element based on 6 rigid-body and 12 straining modes for fast computational simulations of arbitrary isotropic and laminated composite shells,” *Comput. Methods Appl. Mech. Engrg.*, vol. 145, pp. 11–85, 1997.

[2] D. J. Allman, “Evaluation of the constant strain triangle with drilling rotations,” *Int. J. Numer. Meth. Engrg.*, vol. 26, pp. 2645–2655, 1988.

Purpose

Assemble acoustic Helmholtz equation on ultraweak variational form.

Syntax

```
el.elem = 'eluwhelm'
el.g{ig} = geomnum
el.basename = fieldname
el.ndir = ndir
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.Rho{eldomgrp} = rho
el.geomdim{ig}{edim}.K{eldomgrp} = k
el.geomdim{ig}{edim}.Q{eldomgrp} = q
```

Description

The `eleqw` element uses the Ultraweak variational formulation (UWVF) to implement a Helmholtz equation for the acoustic pressure, p,

$$\nabla \cdot \left(-\frac{\nabla p}{\rho_0}\right) - \frac{k^2}{\rho_0} p = 0$$

with boundary conditions of the form

$$n \cdot \frac{\nabla p}{\rho_0} + i\frac{k}{\rho_0}p = q\left(-n \cdot \frac{\nabla p}{\rho_0} + i\frac{k}{\rho_0}p\right) + g$$

The density ρ_0, wave number k, and parameter q are supplied directly to the element, while the boundary term g and all volume, point, and edge sources must be implemented separately, outside the element.

The acoustic field variable `fieldname` must be represented by an `shuwhelm` shape function with fixed number of directions, `ndir`, throughout the domains where the `eluwhelm` element is active.

Cautionary

The eluwhelm element does not currently account for curved boundaries—all element edges and faces are assumed to be planar. This may change, which can possibly affect future element syntax.

See Also

`elempty`, `shuwhelm`

Purpose Define expression variables.

Syntax

```
el.elem = 'elvar'
el.g{'ig'} = geomnum
el.geomdim{ig}{edim}.ind{eldomgrp} = domainlist
el.geomdim{ig}{edim}.var{2*ivar-1} = varname
el.geomdim{ig}{edim}.var{2*ivar}{eldomgrp} = varexpr
```

Description The `elvar` element declares expression variables `varname` to be accessible on domain groups for which the defining expression `varexpr` is nonempty. For the syntax of the `ind` field, see `elempty`.

Examples Redefine the space derivatives of `u` on an interior boundary to be evaluated on the "up" side instead of being averaged.

```
clear fem;
fem.geom = geomcsg({rect2(1,1,'pos',[-1 0]),rect2});
fem.mesh = meshinit(fem);
fem.equ.ind = [1 2];
fem.equ.c = {1 2};
fem.bnd.ind = [1 0 0 0 0 0 2];
fem.bnd.h = 1;
fem.bnd.r = {0 1};
fem.elem = {};

clear el;
el.elem = 'elvar';
el.g = {'1'};
clear gd;
gd.ind = {{'4'}};
gd.var = {'ux',{'up(ux)'},'uy',{'up(uy)'}};
el.geomdim{1} = {{},gd,{}};
fem.elem = [fem.elem {el}];

fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem);
postplot(fem,'lindata','ux','linz','ux');
```

See Also `elempty`

Purpose

Embed a 2D geometry object as a 3D geometry object.

Syntax

```
g3 = embed(g2)
g3 = embed(g2,p_wrkpln)
```

Description

`g3 = embed(g2)` embeds the 2D geometry object as a 3D geometry object. A 2D solid object becomes a 3D face object, a 2D curve object becomes a 3D curve object, and a 2D point object becomes a 3D point object.

`g3 = embed(g2,p_wrkpln)` additionally, `p_wrkpln` specifies the position in the 3D space. See `geomgetwrkpln` for more information on `p_wrkpln`.

See also

`extrude`, `curve2`, `curve3`, `face3`, `geom0`, `geom1`, `geom2`, `geom3`, `point1`, `point2`, `point3`

Purpose Extrude a 2D geometry object into a 3D geometry object.

Syntax

```
g3 = extrude(g2,...)
```

Description `g3 = extrude(g,...)` extrudes the 2D geometry object `g` into a 3D geometry object `g3` according to given parameters.

The function `extrude` accepts the following property/values:

TABLE 1-24: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Displ`	2-by-n_d matrix	`[0;0]`	Displacement of extrusion top
`Distance`	k-by-n_d matrix	`1`	Extrusion distances
`Face`	string	`all`	Cross-sectional faces to delete
`Polres`	scalar	`50`	Polygon resolution
`Scale`	2-by-n_d matrix	`[1;1]`	Scale of extrusion top
`Twist`	1-by-n_d vector	`0`	Twist angle (in radians)
`Wrkpln`	3-by-3 matrix	`[0 1 0; 0 0 1; 0 0 0]`	Work plane for 2D geometry cross section

The 3D object `g3` is an extruded object, where `Distance` is the extrusion distance in the normal direction of the bottom plane, defined by the property `Wrkpln`.

The properties `Displ`, `Scale`, and `Twist` defines the translation displacements, scale factors and rotation of the top with respect to the bottom of the extruded object. They are defined in the local system of the work plane.

To define a piecewise linear extrusion, `Distance` is given as a row vector, of size 1-by-n_d, of displacements with respect to the bottom work plane. `Scale`, `Displ`, and `Twist` need to have the same number of columns as `Distance`.

To define a cubic extrusion `Distance` is given as a 3-by-n_d matrix where rows 2 and 3 contain weights of the extrusion segments. The weights are given in the interval [0 1] and specifies the influence of the tangential continuity at the junctions. The weights of rows 2 and 3 specifies the influence from the first- and second-junction, respectively, of each segment. If the weight is close to 0, the influence of the junction is weak, and if it is close to 1, the influence is strong.

`Polres` defines the resolution in the polygon representations of the edges.

Face specifies if cross-sectional faces are removed: `all` removes them, `none` keeps them.

Compatibility The numbering of faces, edges and vertices is different from the numbering in objects created in 2.3.

Examples Creation of a cylinder of height 1.3.

```
g3 = extrude(circ2,'distance',1.3);
```

Extrusion of rectangle from a *zx*-plane.

```
p_wrkpln = geomgetwrkpln('quick',{'zx',10});
g3 = extrude(rect2(1,2),'distance',1.3,'displ',[0.4;0],...
  'scale',[2;2],'wrkpln',p_wrkpln);
geomplot(g3);
```

Cubic extrusion of a circle.

```
g3 = extrude(circ2,'distance',[1 3 4;0.3 0.3 0.3;0.3 0.3 0.3],...
  'scale',[1 1.5 2;1 1 2],'twist',[0 pi/6 pi/6],...
  'displ',[0 0 0;0 1 1]);
```

See Also `geom0`, `geom1`, `geom2`, `geom3`, `geomcsg`, `geomgetwrkpln`

Purpose

Create 3D surface geometry object.

Syntax

```
f3 = face3(x,y,z)
f3 = face3(x,y,z,w)
f3 = face3(vtx,vtxpre,edg,edgpre,fac,mfdpre,mfd)
[f3,...] = face3(g3,...)
f3 = face3(g2)
```

Description

`f3 = face3(x,y,z)` creates a face3 object `f3`. The degree of the rational Bézier surfaces is determined from the size of the matrices `x`, `y`, and `z`. The arrays `x`, `y`, and `z` are always of equal size. If `size(x,2)>1` then a rectangular patch is created. If `size(x,2)==1` then a triangular patch is created.

The valid combinations of matrix sizes of `x`, `y`, and `z` are: 3-by-1 for creating triangular planar patches, 2-by-2, 3-by-2, 4-by-2, 3-by-3, 4-by-3, 4-by-4 for rectangular patches of degree (1,1), (2,1), (3,1), (2,2), (3,2), and (3,3) respectively.

`f3 = face3(x,y,z,w)` works similarly to the above, but also applies arbitrary positive weights to the points of the surface.

`f3 = face3(vtx,vtxpre,edg,edgpre,fac,mfdpre,mfd)` creates a 3D surface geometry object `f3` from the fields `vtx`, `vtxpre`, `edg`, `edgpre`, `fac`, `mfdpre`, and `mfd`. The arguments must define a valid face object. See `geom3` for a description of the arguments.

`[f3,...] = face3(g3,...)` coerces the 3D geometry object `g3` to a 3D face object f3.

`f3 = face3(g2)` coerces the 2D geometry object `g2` to a 3D face object f3. The object f3 is then embedded in the plane z=0 and is a trimmed planar patch since it is not in general representable as a rectangular or triangular Bézier patch.

The coercion function `[f3,...] = face3(g3,...)` a accepts the following property/values:

TABLE 1-25: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Out`	`stx` | `ftx` | `ctx` | `ptx`	`{}`	Cell array of output names

See `geomcsg` and `geom` for more information on geometry objects.

The 3D geometry object properties are available. The properties can be accessed using the syntax `get(object,property)`. See `geom3` for details.

Examples

Create an untrimmed triangular patch in the plane, $y = 1$.

```
f1 = face3([0 1 0]',[1 1 1]',[0 0 1]');
```

Create a circular face as a trimmed patch in the plane, $z = 0$.

```
f2 = face3(circ2(0,20,10));
```

A patch that can constitute the wall of a cylinder, is created by setting the control weights explicitly, as in the command below.

```
f3 = face3([-1 -1;-1 -1;0 0],[0 0;-1 -1;-1 -1],...
            [0 1; 0 1; 0 1],[1 1;1/sqrt(2) 1/sqrt(2);1 1]);
```

To generate a third degree rectangular patch, the following commands can be given.

```
[x,y] = meshgrid(-3:3:6,0:2:6);
z = rand(size(x));
f4 = face3(x,y,z);
```

Compatibility

The FEMLAB 2.3 syntax is obsolete but still supported.

See Also

`curve2`, `curve3`, `geom0`, `geom1`, `geom2`, `geom3`, `geomcsg`, `point1`, `point2`, `point3`

femdiff

Purpose Symbolically differentiate a PDE in general form.

Syntax

```
fem1 = femdiff(fem,...)
xfem1 = femdiff(xfem,...)
```

Description `fem1 = femdiff(fem,...)` symbolically differentiates the Γ, F, G, and R coefficients in a PDE given in general form. `xfem` can also be an extended FEM structure. In this case, `femdiff` differentiates all FEM structures in general form. The coefficients are obtained from the `ga`, `f`, and `g`, `r` fields from the `fem.equ` and `fem.bnd` structures, respectively. It returns an FEM structure where the fields `equ` and `bnd` have been updated with the fields `c`, `al`, `be`, `a`, and `q`, `h` according to "The Linear or Linearized Model" on page 386 in the *COMSOL Multiphysics User's Guide*.

The expressions in the coefficients Γ, F, G, and R can contain expressions containing the binary operators `+`, `-`, `*`, `/`, `^`, `==`, `~=`, `>`, `>=`, `<`, `<=`, `|`, and `&`; the unary operators `+`, `-`, and `~`; and the functions `abs`, `acos`, `acosh`, `acot`, `acoth`, `acsc`, `acsch`, `asec`, `asech`, `asin`, `asinh`, `atan`, `atanh`, `cos`, `cosh`, `cot`, `coth`, `csc`, `csch`, `erf`, `exp`, `lambw`, `log`, `log10`, `log2`, `sec`, `sech`, `sign`, `sin`, `sinh`, `sqrt`, `tan`, and `tanh`.

The function `femdiff` accepts the following property/value pairs:

TABLE 1-26: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Defaults`	`off` | `on`	`off`	Return default fields
`Diff`	`off` | `on` | cell array containing a selection of `ga`, `g`, `f`, `r`, `var`, and `expr`	`on`	List of fields which can be differentiated in order to evaluate the `Out` values. The default string `'on'` is equivalent to the cell array `{'ga' 'g' 'f' 'r' 'expr'}`. Note: not `'var'`
`Shrink`	`off` | `on`	`on`	List of differentiation rules
`Simplify`	`off` | `on`	`on`	Simplify differentiated expressions

The properties `Diff`, `Rules`, and `Simplify` can alternatively be given as fields in the FEM structure: `fem.diff`, `fem.rules`, and `fem.simplify`.

Use the field `fem.rules` to specify additional differentiation rules. The derivative of the inverse hyperbolic tangent function `atanh` can, for example, be specified as

```
{'atanh(x)','1/(1-x^2)'}.
```

It can also be stored as a field in the FEM structure.

Assume a user-defined function `foo(a,b)` has been written, implementing the analytical expression `a^2+a*sin(b^3)`. The derivatives of this function are specified as:

```
'rules', {'foo(a,b)', '2*a+sin(b^3),3*a*b^2*cos(b^3)'}
```

`femdiff` does not support functions with string arguments.

Cautionary

The relational and Boolean operators have been included for convenience only, and must be used with extreme caution. Their symbolic derivative is considered to be identically 0.

Coefficient M-files are not allowed in the input.

Compatibility

The function `flgetrules` for converting FEMLAB 1.0/1.1 differentiation rules is no longer available.

The properties `bdl`, `out`, `rules`, and `sdl` are obsolete in FEMLAB 3.0.

The fields `fem.equ.varu` and `fem.bnd.varu`, etc. are no longer generated in FEMLAB 3.0.

The precedence rules for the operators `|` and `&` have been changed to comply with MATLAB 6.0 precedence.

The differentiation algorithm is new in FEMLAB 1.2. The `@fldiffobj` class is obsolete.

See Also

`femnlin`

Purpose Solve eigenvalue PDE problem.

Syntax

```
fem.sol = femeig(fem,...)
[u,lambda] = femeig(fem,...)
fem = femeig(fem,'Out',{'fem'},...)
fem.sol = femeig('In',{'D' D 'K' K 'N' N},...)
```

Description

`fem.sol = femeig(fem,...)` assembles and solves the eigenvalue PDE problem described by the (possibly extended) FEM structure `fem`.

`fem.sol = femeig('In',{'D' D 'K' K 'N' N},...)` solves the eigenvalue problem given by the matrices `D`, `K`, and `N`.

For both linear and nonlinear PDE problems, the eigenvalue problem is that of the linearization about a solution U_0. If the eigenvalue appears nonlinearly, COMSOL Multiphysics reduces the problem to a quadratic approximation around a value λ_0 specified by the property `eigref`. The discretized form of the problem reads

$$KU - (\lambda - \lambda_0)DU + (\lambda - \lambda_0)^2 EU = -N_F \Lambda$$
$$NU = M$$

where K, D, E, N and N_F are evaluated for $U = U_0$ and $\lambda = \lambda_0$. Λ is the Lagrange multiplier vector, λ is the eigenvalue. The eigenvalue name can be given by the property `eigname`. The linearization point U_0 can be given with the property `U`. The shift, described below, is compensated according to the linearization point for the eigenvalue. Therefore, changing the linearization point has no effect at all for linear or quadratic eigenvalue problems.

The function `femeig` accepts the following property/value pairs:

TABLE 1-27: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Eigname`	string	`lambda`	Name of eigenvalue variable
`Eigref`	string	0	Linearization point for the eigenvalue
`Etol`	positive scalar	0	Eigenvalue tolerance
`In`	cell array of names and matrices `K` | `N` | `D` | `E`	`N` is empty `E`=0, `D`=0	Input matrices
`Krylovdim`	positive integer		Dimension of Krylov space

TABLE 1-27: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Neigs`	positive integer	`6`	Number of eigenvalues sought
`Out`	`fem` | `sol` | `u` | `lambda` | `stop` | `solcompdof` | `Kc` | `Dc` | `Ec` | `Null` | `Nullf` | `Nnp` | `uscale` | `nullfun` | `symmetric`	`sol`	Output variables
`Shift`	scalar	`0`	Eigenvalue search location

In addition, the properties described in the entry `femsolver` are supported.

Specify where to look for the desired eigenvalues with the property `shift`. Enter a real or complex scalar; the default value is 0, meaning that the solver tries to find eigenvalues close to 0.

For the tolerance parameter in the convergence criterion for linear systems, the number given by `Itol` is used.

Using the property `In` you can specify explicit values for the matrices in the eigenvalue problem. The value for this property is a cell array with alternating matrix names and matrix values. The matrix names can be `D`, `E`, `K`, or `N`. If the `N` matrix is not given, it is taken to be empty. If the `D` or `E` matrix is not given, it is taken to be 0.

The property `Out` defines the output variables and their order. The output `fem` means the FEM structure with the *solution object* `fem.sol` added. `sol` is a `femsol` object containing the fields `lambda` and `u`. `lambda` is a row vector containing the eigenvalues. `u` is a *solution matrix*. Each column in the solution matrix is the solution vector of the eigenfunction for the corresponding eigenvalue in `lambda`. The output value `stop` returns nonzero if the solution process was not completed. `stop` is `1` if a partial solution was returned, and `2` if no solution was returned. For the other outputs, see `femlin`.

For more information about the eigenvalue solver, see "The Eigenvalue Solver" on page 402 in the *COMSOL Multiphysics User's Guide*.

Example

Eigenmodes and Eigenvalues of the L-shaped Membrane

Compute eigenvalues corresponding to eigenmodes for the PDE problem

$$\begin{cases} -\Delta u = \lambda u & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases}$$

where Ω is the L-shaped membrane. Start by setting up the problem:

```
clear fem
fem.geom = poly2([-1 0 0 1 1 -1],[0 0 1 1 -1 -1]);
fem.mesh = meshinit(fem,'hmax',0.1);
fem.shape = 2;
fem.equ.c = 1; fem.equ.da = 1;
fem.bnd.h = 1;
fem.xmesh = meshextend(fem);
fem.sol = femeig(fem,'neigs',16);
```

Display the first and sixteenth eigenmodes. The `membrane` function is available in MATLAB.

```
postsurf(fem,'u')             % first eigenmode
membrane(1,20,9,9)            % the MATLAB function
postsurf(fem,'u','solnum',16) % sixteenth eigenmode
```

Cautionary

Consider the case of a linear eigenvalue problem, $E = 0$. Write the generalized eigenvalue problem as $(A - \lambda B)u = 0$. In the standard case the coefficients c and d_a are positive in the entire region. All eigenvalues are positive, and 0 is a good choice for the shift (eigenvalue search location). The cases where either c or d_a is zero are discussed below.

- If $d_a = 0$ in a subregion, the mass matrix B becomes singular. This does not cause any trouble, provided that $c > 0$ everywhere. The pencil (A, B) has a set of infinite eigenvalues.
- If $c = 0$ in a subregion, the stiffness matrix A becomes singular, and the pencil (A, B) has many zero eigenvalues. Choose a positive shift below the smallest nonzero eigenvalue.
- If there is a region where both $c = 0$ and $d_a = 0$, we get a singular pencil. The whole eigenvalue problem is undetermined, and any value is equally plausible as an eigenvalue.

Compatibility

The property `Variables` has been renamed `Const` in FEMLAB 2.3.

The properties `Epoint` and `Tpoint` are obsolete from FEMLAB 2.2. Use `fem.***.gporder` to specify integration order.

See Also

`femsolver`, `assemble`, `femlin`

Purpose Solve linear or linearized stationary PDE problem.

Syntax

```
fem.sol = femlin(fem,...)
fem = femlin(fem,'Out', {'fem'},...)
[Ke,Le,Null,ud] = femlin(fem,...)
[Kl,Ll,Nnp] = femlin(fem,...)
[Ks,Ls] = femlin(fem,...)
fem.sol = femlin('In',{'K' K 'L' L 'M' M 'N' N 'NF' NF},...)
```

Description

`fem.sol = femlin(fem)` solves a linear or linearized stationary PDE problem described by the (possibly extended) FEM structure `fem`. See `femstruct` for details on the FEM structure.

`fem.sol = femlin(fem, 'pname','P', 'plist',list,...)` solves a linear or linearized stationary PDE problem for several values of the parameter `P`. The values of the parameter `P` are given in the vector `list`.

`fem = femlin(fem,'out',{'fem'})` modifies the FEM structure to include the solution structure, `fem.sol`.

`[Ke,Le,Null,ud] = femlin(fem)` partially solves the PDE problem by eliminating the constraints. The solution of PDE problem can be obtained by the scripting command `u = u0+Null*(Ke\Le)+ud`, where `u0` is the linearization point.

`[Kl,Ll,Nnp] = femlin(fem)` partially solves the PDE problem by using the Lagrange method. The solution can then be obtained by `u = Kl\Ll`, and then `u = u0+u(1:Nnp)`.

`[Ks,Ls] = femlin(fem)` partially solves the PDE problem by approximating the constraints with stiff springs. The solution to the PDE problem is `u = u0+Ks\Ls`.

`fem.sol = femlin('in',{'K' K 'N' N 'NF' NF 'L' L 'M' M})` solves a pre-assembled PDE problem.

`u = femlin('in’,{'K' K 'L' L},'out','u')` is equivalent to solving the linear system using `u = K\L`, with the important difference that you have access to all linear system solvers (except Geometric multigrid) using the `Linsolver` property.

Consider the finite element discretization of a stationary PDE problem:

$$0 = \begin{bmatrix} L - N_F\Lambda \\ M \end{bmatrix}$$

where L, N_F, and M depend on the solution vector U. `femlin` solves the linearized form of this problem:

$$\begin{bmatrix} K & N_F \\ N & 0 \end{bmatrix} \begin{bmatrix} U - U_0 \\ \Lambda \end{bmatrix} = \begin{bmatrix} L \\ M \end{bmatrix}$$

where K, N_F, N, L, and M are evaluated for $U = U_0$. Thus, if the original problem is linear and K is the correct Jacobian, `femlin` computes the solution of the original problem. The linearization "point" U_0 can be specified with the property `U`.

`femlin` can also partially solve the eigenvalue problem:

$$\begin{aligned} KU - (\lambda - \lambda_0)DU + (\lambda - \lambda_0)^2 EU &= -N_F \Lambda \\ NU &= M \end{aligned}$$

in that it transforms the problem using one of the constraint-handling methods. Here λ is the eigenvalue, the name can be controlled by the property `eigname`. λ_0 is the eigenvalue linearization point, the value can be controlled by the property `eigref`.

The function `femlin` accepts the following property/values:

TABLE 1-28: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Eigname`	string	`lambda`	Eigenvalue name
`Eigref`	string	`0`	Linearization point for the eigenvalue
`In`	cell array of names and matrices `K` | `L` | `M` | `N` | `NF` | `D` | `E`	`N` and `M` are empty, `D=E=0`, `NF=N`^{T}	Input matrices
`Keep`	string containing `K`, `N` | `auto`	`auto`	Parameter-independent quantities
`Oldcomp`	cell array of strings		Old parameter components
`Out`	`fem` | `sol` | `u` | `plist` | `stop` | `solcompdof` | `Kc` | `Lc` | `Dc` | `Ec` | `Null` | `Nullf` | `Nnp` | `ud` | `uscale` | `nullfun` | `symmetric` | cell array of these strings	`sol`	Output variables
`Pinitstep`	positive real		Initial stepsize for parameter

TABLE 1-28: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Plist`	real vector		List of parameter values
`Pmaxstep`	positive real		Maximum stepsize for parameter
`Pminstep`	positive real		Minimum stepsize for parameter
`Pname`	cell array of strings		Parameter names
`Porder`	0	0	Predictor order for parameter stepping
`Stopcond`	string with expression		Stop parameter stepping before expression become negative

In addition, the properties described in the entry `femsolver` are supported.

The parametric solver properties `Oldcomp`, Pdistrib, `Pinitstep`, `Plist`, `Pmaxstep`, `Pminstep`, `Pname`, `Porder`, and `Stopcond` are described under `femnlin`.

The property `In` explicitly provides assembled matrices. Its value is a cell array with alternating matrix names and matrix values. The allowed matrix names are `K`, `L`, `M`, `N`, `NF`, `D`, and `E`.

The property `Out` explicitly sets output variables and their order. The output variable `fem` means the FEM structure with the solution object `fem.sol` added. The outputs `sol`, `u`, and `plist` are the solution object, the solution matrix (`sol.u`), and the parameter list (`sol.plist`), respectively. The output value `stop` is 0 if a complete solution was returned, 1 if a partial solution was returned, and 2 if no solution was returned. The output `solcompdof` is a vector containing the indices of the degrees of freedom solved for. The output matrix `Kc` and the vector `Lc` are the matrix and right-hand side of the linear system after constraint handling; see "Constraint Handling" on page 533. The matrices `Dc` and `Ec` are the corresponding damping matrix and mass matrix after constraint handling for an eigenvalue or time-dependent problem. The outputs `Null`, `Nullf` and `ud` are related to the eliminate constraint handling method. The outputs `Nnp` and `uscale` are the number of degrees of freedom solved for and the scale factors used in the rescaling of the degrees of freedom; see "Scaling of Variables and Equations" on page 531. The outputs `nullfun` and `symmetric` can be useful in finding out the result of the automatic null function or the automatic symmetric mechanisms.

Example

The L-Shaped Membrane with Three Subdomains

Take a look at the geometry of the L-shaped membrane for examples of what you can do. First create the L-shaped membrane and examine the subdomain labels and edge segment labels by plotting:

```
clear fem
sq1 = square2(0,0,1);
sq2 = move(sq1,0,-1);
sq3 = move(sq1,-1,-1);
fem.geom = sq1+sq2+sq3;
fem.mesh = meshinit(fem);
geomplot(fem,'edgelabel','on','sublabel','on')
```

Say you want to use $c = 1$, $1/2$, and $1/3$ and $f = x$, y, and x^2+1 in subdomains 1, 2, and 3, respectively. Use Dirichlet boundary conditions on the outer boundaries:

```
fem.shape = 2;
fem.equ.c = {1 1/2 1/3};
fem.equ.f = {'x' 'y' 'x^2+1'};
fem.bnd.h = 1;
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem);
postsurf(fem,'u')
```

You can set `fem.bnd.h = 1` because `fem.border` has not been set (and defaults to `off`). When `fem.border` is set to `on` you must type `fem.bnd.h = {1 1 1 0 1 1 0 1 1 1}`. Otherwise you get `u=0` also on interior boundaries.

Using anisotropic $c = \begin{bmatrix} 2 & x+y \\ x+y & 10 \end{bmatrix}$ and $f = 1$ in all subdomains can be done by typing

```
fem.equ.c = {{{2 'x+y' 10}}};
fem.equ.f = 1;
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem);
postsurf(fem,'u')
```

Cautionary

When using the general form it is assumed that the coefficients c, α, β, a, q, h have been computed using `fem=femdiff(fem)`. In the user interface, this is done automatically.

Compatibility

The property `Variables` has been renamed `Const` in FEMLAB 2.3.

If scaling is used, the matrix outputs from `femlin` are derived from the rescaled system. This means that the scale factors `uscale` have to be taken into account if a solution is computed from the matrices. See “Scaling of Variables and Equations” on page 531.

The properties `Epoint` and `Tpoint` are obsolete from FEMLAB 2.2. Use `fem.***.gporder` to specify integration order.

The properties `u` and `t` have been made obsolete in FEMLAB 1.1.

See Also `femsolver`, `femstruct`, `assemble`, `asseminit`, `femnlin`, `femeig`, `flnull`

Purpose Create a mesh object.

Syntax `fem.mesh = femmesh(p, el)`

Description `fem.mesh = femmesh(p, el)` creates a mesh object from the mesh data stored in `p` and `el`.

`p` is an `sdim`-by-`np` matrix containing the coordinates of the mesh vertices. The *x*-, *y*-, and *z*-coordinates are stored in the first, second, and third row, respectively. `np` is the number of mesh vertices.

`el` is a cell array of structures with mesh element information. Each structure stores information on elements of a specific type.

STRUCTURE FIELD	VALUE	DESCRIPTION
`type`	`vtx` | `edg` | `tri` | `quad` | `tet` | `prism` | `hex`	Element type. The valid element types are: vertex element (vtx), edge element (edg), triangular element (tri), quadrilateral element (quad), tetrahedral element (tet), prism element (prism), and hexahedral element (hex)
`elem`	matrix of size nNodes-by-nElem	Mesh vertex indices for the element points. nElem is the number of elements and nNodes is the number of element points
`dom`	matrix of size 1-by-nElem	Geometry domain numbers
`param`	matrix of size nParam-by-nElem	Geometry parameter values
`ud`	matrix of size 2-by-nElem	Up- and down-side subdomain numbers. The first row contains the up-side subdomain numbers and the second row the down-side subdomain numbers

The field `param` is only valid for elements of dimension 1, that is, edge elements, in 2D and 3D, and for elements of dimension 2, that is, triangular or quadrilateral elements, in 3D. For each edge element, the first and second row contain the starting and ending parameter value, respectively, in 2D, and starting and ending arc length value, respectively, in 3D. For each triangular and quadrilateral element, the rows contain the first and second parameter values for each element corner.

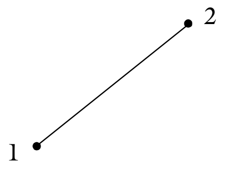
The field `ud` is only valid for elements of dimension sdim−1, also referred to as *boundary elements*. The direction of the normal vector of a boundary element defines the up-side and down-side of the boundary element. For a 1D boundary element, the normal points to the left, considering the direction of the boundary element. For a 2D boundary element, the normal is defined as the cross product of the vector going from the first to the second element corner and the vector going from the first to the third element corner.

When defining a mesh object using the `femmesh` command the domain numbering for each dimension must start from 1 and must not contain gaps.

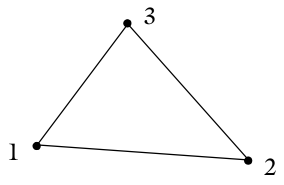
The properties `p` and `el` can be accessed using the syntax `get(object,property)`.

The (local) numbering of the corners of an element is defined according to the following.

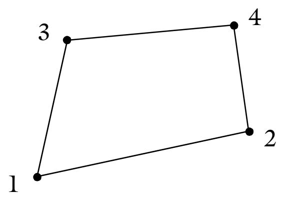
Edge element (`edg`):



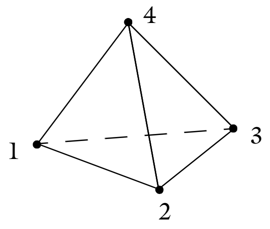
Triangular element (`tri`):



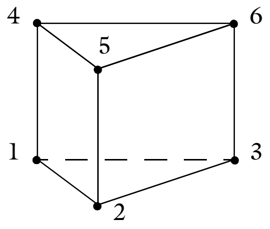
Quadrilateral element (`quad`):



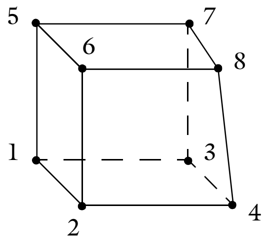
Tetrahedral element (`tet`):



Prism element (`prism`):

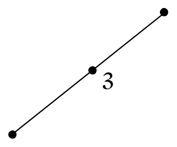


Hexahedral element (`hex`):

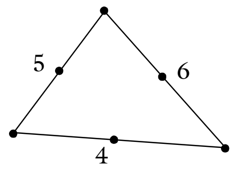


For second-order mesh element types, the strings `edg2`, `tri2`, `quad2`, `tet2`, `prism2`, and `hex2` are used. The second-ordered nodes are numbered after the corner vertices according to the following.

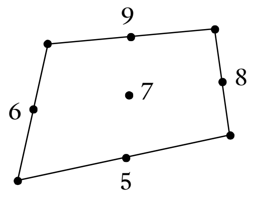
Edge element (`edg2`):



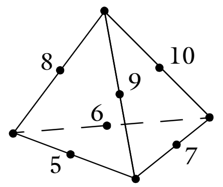
Triangular element (`tri2`):



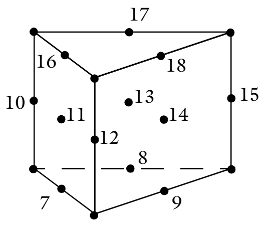
Quadrilateral element (`quad2`):



Tetrahedral element (`tet2`):



Prism element (`prism2`):

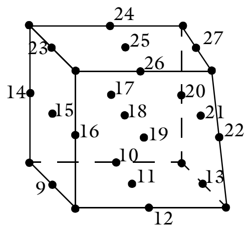


The mid node number for each quadrilateral face of the prism element can also be seen in the following table.

FACE (EDGE NODES)	FACE MID NODE
7,10,12,16	11

FACE (EDGE NODES)	FACE MID NODE
8,10,15,17	13
9,12,15,18	14

Hexahedral element (`hex2`):



The mid-node number for each quadrilateral face of the hexahedral element can also be seen in the following table.

FACE (EDGE NODES)	FACE MID NODE
9,10,12,13	11
9,14,16,23	15
10,14,20,24	17
12,16,22,26	19
13,20,22,27	21
23,24,26,27	25

The mid-node number for the hexahedral element is 18.

When importing meshes with reduced second-order elements, also called *serendipity elements*, the mid node of quadrilateral elements (or quadrilateral faces) and the mid node of hexahedral elements must be added manually. The coordinates for the mid node for a second order quadrilateral element (or quadrilateral face) is calculated from the surrounding nodes according to `0.5*edgNodes-0.25*vertNodes`, where `edgNodes` is the sum of the surrounding (4) edge mid nodes and `vertNodes` is the sum of the surrounding (4) vertex nodes. For a second order hexahedron, the coordinates of the mid node is calculated from the surrounding nodes according to `0.25*edgNodes-0.25*vertNodes`, where `edgNodes` is the sum of the surrounding (12) edge mid nodes and `vertNodes` is the sum of the surrounding (8) vertex nodes.

Degenerated elements (or *collapsed elements*), that is, elements where two or more nodes refer to the same mesh point, are not allowed.

Compatibility

The FEMLAB 2.3 (and earlier) mesh structure format is a valid input to `femmesh` as well.

See also

`meshinit`, `meshrefine`, `meshplot`

Purpose Get mesh object properties.

Syntax `get(m,prop)`

Description `get(m,prop)` returns the value of a property `prop` for a mesh object `m`.

`prop` is a string that contains a valid property name. The following tables list the valid property names for mesh objects:

TABLE 1-29: MESH OBJECT PROPERTY NAMES

PROPERTY NAME	DESCRIPTION
p	Mesh vertex coordinates
el	Element information

`p` is a matrix where each column contains the coordinates for the corresponding mesh vertex. For example, `p(:,34)` returns the coordinates for Vertex 34.

`el` is a cell array of structures with mesh element information. See `femmesh` on page 136 for information about the field in these structures.

For information about the formats for the `vtx` property, see “1D Geometry Object Properties” on page 228.

Example Create a triangular mesh and determin the vertices that form mesh element 100:

```
m = meshinit(rect2);
el = get(m,'el');
el{3}.elem(:,100);
```

See Also `femmesh`

Purpose Solve nonlinear stationary PDE problem.

Syntax

```
fem.sol = femnlin(fem,...)
fem = femnlin(fem,'Out',{'fem'},...)
```

Description `fem.sol = femnlin(fem)` solves a stationary PDE problem.

`fem.sol = femnlin(fem,'pname','P','plist',list,...)` solves a stationary PDE problem for several values of the parameter `P`. The values of the parameter `P` are given in the vector `list`.

The PDE problem is stored in the (possibly extended) FEM Structure `fem`. See `femstruct` for details.

The solver is an affine invariant form of the damped Newton method. The solver can optionally be combined with Uzawa iterations, often used to solve problems with the augmented Lagrangian technique.

The function `femnlin` accepts the following property/value pairs:

TABLE 1-30: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Augcomp`	cell array of strings		Augmented Lagrange components
`Augmaxiter`	positive integer	25	Max number of augmentation iterations
`Augsolver`	`umfpack` | `spooles` | `taucs_llt_mf` | `taucs_ldlt` | `luinc` | `taucs_llt` | `gmres` | `fgmres` | `cg` | `bicgstab` | `amg` | `gmg` | `ssor` | `ssoru` | `sor` | `soru` | `jac` | `lumped`	`umfpack`	Linear system solver for augmented Lagrange components
`Augtol`	positive real	`1e-6`	Tolerance for augmented Lagrange
`Damp`	`positive real`	1	Damping factor for the damped Newton method
`Dtech`	`const` | `autodamp` | `newton`	see below	Damping technique
`Hnlin`	`on` | `off`	`off`	Indicator of a highly nonlinear problem
`Initstep`	non-negative scalar	see below	Initial damping factor

TABLE 1-30: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Jtech`	`minimal` | `once` | `onevery`	see below	Jacobian update technique
`Keep`	string containing K, N | `auto`	`auto`	Parameter and iteration-independent quantities
`Maxiter`	positive integer	25	Maximum number of Newton iterations
`Minstep`	positive scalar	see below	Minimum damping factor
`Ntol`	positive scalar	`1e-6`, `1e-3` (segregated solver groups)	Relative tolerance for stationary problem
`Oldcomp`	cell array of strings	`{}`	Old parameter components
`Out`	`fem` | `sol` | `u` | `plist` | `stop` | `solcompdof` | `Kc` | `Lc` | `Null` | `Nnp` | `ud` | `uscale` | `nullfun` | `symmetric` | cell array of these strings	`sol`	Output variables
`Pdistrib`	`on` | `off`	`off`	If the solver should distribute the parameter sweep
`Pinitstep`	positive real		Initial stepsize for parameter
`Plist`	real matrix		List of parameter values
`Pmaxstep`	positive real		Maximum stepsize for parameter
`Pminstep`	positive real		Minimum stepsize for parameter
`Pname`	cell array of strings		Parameter names
`Porder`	`0` | `1`	1	Predictor order for parameter stepping
`Rstep`	real scalar > 1	10	Restriction for step size update
`Stopcond`	string with expression		Stop parameter stepping before expression becomes negative

In addition, the properties described in the entry `femsolver` are supported.

The property `Augcomp` control the augmentation components. If this property is set to a subset of the solution components, then the solution procedure is split into two substeps, that are repeated until a convergence criterion is met or until the maximum number of iterations is reached. The main components, which are the solution components excluding the augmentation components, are first solved for, while the augmentation components are held fix. After this, the augmentation components are solved for while the main components are held fix. In this second solution step, a linear solution approach is taken. Therefore, with the main components fixed, the augmentation components are assumed to fulfill a linear equation. The property `Augtol` control the tolerance for the augmentation components and the property `Ntol` the tolerance for the main components in the convergence criterion for the combined iteration (two substeps each). The convergence criterion is that the relative increment, from one iteration to the next, for the augmented components *and* the main components must not be larger than their tolerances. The maximum number of combined iterations is controlled by the property `Augmaxiter`. The linear system solver used for the solution of the augmentation components can be controlled by the property `Augsolver`. For a more detailed description of this solution procedure, see the "Nonlinear Solver Settings" on page 389 in the *COMSOL Multiphysics User's Guide*.

The property `Dtech` controls which damping factor to use in the damped Newton iterations. When `Dtech=const`, the constant damping factor specified in the property `Damp` is used. When `Dtech=autodamp`, the solver determines an appropriate damping factor. With `Dtech=newton`, a full Newton approach is used; that is, the constant damping factor one is used. For a stationary problem, the default is `autodamp`. When the problem is time-dependent (`femtime` is used), the default is `const`.

Setting `Hnlin` to `on` causes the solver to treat the problem as being highly nonlinear. This option can be tried if there is no convergence with `Hnlin` set to `off`. Depending on this parameter, certain standard values are selected for the `Initstep` and `Minstep` properties. Moreover, certain internal control structures are adapted. Especially, the error control is biased from a more absolute norm towards a relative norm. So this parameter is also useful if a solution with components of highly varying orders of magnitudes are present. In the context of parameter stepping, you can also try this option if the step sizes in the parameter seem to be too small.

`Initstep` is the initial damping factor for the step length. The default is `1` if `Hnlin` is off and `1e-4` if `Hnlin` is `on`.

When `Dtech=const`, the property `Jtech` can be used to control how often the Jacobian is updated. With `Jtech=minimal`, the Jacobian is updated as seldom as possible (only once for a stationary problem and at most once per time step for a time-dependent problem). For time-dependent problems, the choice `Jtech=once` makes the solver update the Jacobian once per time step. With `Jtech=onevery`, the Jacobian is updated on every Newton iteration. The default is `onevery` for stationary problems and `minimal` for time-dependent problems.

`Maxiter` and `Minstep` are safeguards against infinite Newton iterations. They bound the number of iterations and the damping factor used in each iteration. `Minstep` defaults to `1e-4` if `Hnlin` is `off` and `1e-8` if `Hnlin` is `on`.

The tolerance `Ntol` gives the criterion for convergence for a stationary problem, see "Nonlinear Solver Settings" on page 389 in the *COMSOL Multiphysics User's Guide*.

The property `Out` explicitly sets output variables and their order. The output variable `fem` means the FEM structure with the solution object `fem.sol` added. The solution object `sol` has a field `sol.u`, which is the solution vector for the FEM formulation of the PDE problem. The solution vector `u` is a column vector with one component for each degree of freedom of the discretized problem. If the parameter variation feature is used, then `sol.u` is a matrix, and there are additional fields `sol.pname` and `sol.plist`. The field `sol.pname` is the name of the parameter, and `sol.plist` is a matrix with parameter values for which a solution was computed. The corresponding solution vectors are stored as columns in the matrix `sol.u`. The output variable `Stop` is 0 if a complete solution was returned, 1 if a partial solution was returned, and 2 if no solution was returned. For the other outputs, see `femlin`.

`femlin` and `femnlin` can solve a stationary problem for a number of values of a parameter or several parameters at once. The name of the parameters are specified with the property `Pname`, and the values of the parameters are specified with the property `Plist`. The matrix in `Plist` can be an increasing or decreasing sequence in the first parameter. It is also possible to specify `Plist` as a vector containing pairs of group values. If more than two parameter values are given, then solutions are delivered for these parameter values (though the algorithm may internally compute the solution for intermediate values). If only two parameter values are given, the algorithm also delivers the solutions for the intermediate values determined by the algorithm. The algorithm tries to follow a continuous path of solutions when varying the parameter, and adjusts the step size in the parameter in order achieve this. If the algorithm detects that some sort of singularity or turning point is approached, then the stepsize is reduced, and the algorithm terminates. In this case, if the property `Stop` is set to `on`, the solutions for the visited parameter values are delivered.

If the property `Pdistrib` is set to `on` and the distributed version is used, the parameter sweep will be distributed between the computer nodes. It is assumed that the parameters are independent in the sense that the list can be split in any way without causing convergence problems for each separate problem. The distributed version only stores the solutions for the values in `Plist`.

When going from one parameter value to another, the initial guess at the new parameter value is by default obtained by following the tangent to the solution curve at the old parameter value. If the property `Porder` is set to `0`, then the initial guess is instead taken as the solution for the old parameter value. In very simple cases, `Porder = 0` may give better performance than the default `Porder = 1`. In the case of parameter sweeps with several parameters involved the last solution is always used as initial guess, i.e. `Porder` is always `0`.

The property `Pinitstep` specifies the initial parameter stepsize that will be tried. The algorithm terminates if the Newton method diverges and the parameter step is less than `Pminstep`. The property `Pmaxstep` provides an upper bound on the parameter step. If any of the properties `Pinitstep`, `Pminstep`, or `Pmaxstep` are `0` or not given, they are given default values.

For some applications the access to the solution at a previous parameter value is needed. Such an application is for example contact problems with friction in Structural Mechanics. The solution components controlled by the property `Oldcomp` are treated in a separate linear solution step, or updating step, performed after the solver for the parameter step has finished. These components are subtracted from the solution components and are not included in the main parametric solver step. The linear system solver used in the update solver step is UMFPACK.

For more information on the parameter-stepping feature, see "The Parametric Solver" on page 405 in the *COMSOL Multiphysics User's Guide*.

The property `Rstep` sets a restriction for the damping factor update in the Newton iteration. Each time the damping factor is updated, it is allowed to change at most by a factor `Rstep`.

If the property `Stop` is set to on, the solver gives an output even if the algorithm fails at some point. If the parameter stepping feature is used with more than one parameter value, the output contains the solutions for the parameters that were successfully computed. Otherwise, the output is the nonconverged solution corresponding to the iteration where the failure occurred. If `Stop` is set to `off`, the solver terminates with an error if the algorithm fails.

Use the property `Stopcond` to make sure the solver stops when a specified condition is fulfilled. You provide a scalar expression that is evaluated after each parameter step. The parameter stepping is stopped if the real part of the expression is evaluated to something negative. The corresponding solution, for which the expression is negative is not returned.

For more information about the nonlinear stationary solver, see "The Stationary Solver" on page 385 in the *COMSOL Multiphysics User's Guide*.

Diagnostics

If the Newton iteration does not converge, the error messages `Maximal number of iterations reached` or `Damping factor too small` are displayed. If during the solution process `NaN` or `Inf` elements are encountered in the solution even after reducing the damping factor to the minimum, the error message `Inf or NaN repeatedly found in solution` is printed. The message `Underflow of parameter step length` means that the Newton iterations did not converge, even after reducing the parameter step length to the limit given in `Pminstep`. This probably means that the curve of solutions has a turning point or bifurcation point close to the current parameter value and solution.

Compatibility

The property `Variables` has been renamed `Const` in FEMLAB 2.3.

The properties `Epoint` and `Tpoint` are obsolete from FEMLAB 2.2. Use `fem.***.gporder` to specify integration order.

The property/value `Jacobian/Lumped` has been made obsolete from FEMLAB 1.1.

The properties `Toln` and `Normn` have been made obsolete from FEMLAB 1.2. `Ntol` replaces `Toln`.

See Also

`assemble`, `asseminit`, `femlin`, `femsolver`, `femstatic`, `femstruct`

Purpose Solve a PDE-constrained optimization problem

Syntax

```
fem.sol = femoptim(fem,...);
[fem.sol obj] = femoptim(fem,...)
```

Description `fem.sol = femoptim(fem,...)` solves a PDE-constrained optimization problem, returning the PDE solution evaluated for the optimal set of design variables. When the gradient-evaluation method is analytic, `femoptim` also returns the adjoint solution. The complete problem description is given by the FEM structure and an OPT structure, whose fields are described in Table 1-32.

Note: `femoptim` requires the Optimization Lab.

`[fem.sol obj] = femoptim(fem,...)` in addition returns a vector `obj` containing the values of the objective function at all major iterations.

The function `femoptim` accepts the following property/value pairs:

TABLE 1-31: VALID PROPERTY/VALUE PAIRS FOR FEMOPTIM

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Callback`	string	Empty	Function to call after each evaluation of the objective function
`Callblevel`	`optim` | `param` | `nonlin`	`optim`	Callback solverlevel
`Callbparam`	cell array	Empty	Allows additional arguments to be passed to the callback function
`Gradient`	`analytic` | `numeric`	`analytic`	Gradient/Jacobian evaluation method
`Limitexpr`	string	Empty	Reduce SQP step length if limitexpr<limitval
`Limitval`	real scalar	`0`	Threshold value for limitexpr
`Nsolvemax`	integer	not set	Maximum number of PDE solutions
`Opt`	OPT structure	`fem.opt`	Problem definition
`Optcomp`	cell array	Empty	Active optimization variables

TABLE 1-31: VALID PROPERTY/VALUE PAIRS FOR FEMOPTIM

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Optprop`	cell array	See Table 1-33	Optimization solver parameters
`Out`	`sol` | `objfun` | cell array of these strings	`sol`	Output variables
`Report`	`on` | `off`	`off`	Report progress
`Solcomp`	cell array	`all`	Active PDE solution components and optimization variables
`Solprop`	cell array	See `femstatic`	Solver properties

The `callback` function is called with the FEM structure `fem` and optional arguments `callbparam`. A cell array is unpacked in the function call, that is, `callbparam = {a1,a2}` results in the function being called with `callback(fem,a1,a2)`. To control when the solver makes a callback, use the `callblevel` property. When `callblevel` is `optim`, the solver makes a callback after each optimization step. Note that one optimization step may require solving the PDE more than once. For the values `param` and `nonlin`, see the description under the entry for `femstatic` on page 182.

The property `Out` determines whether `femoptim` returns the PDE solution (if the value is `sol` or the property is not specified), the objective function (if the value is `objfun`), or both (if the value is a cell array of these two strings).

The property `Solprop` may contain most property/value pairs listed under `femstatic` and `femsolver`.

By default, `femoptim` looks for an optimization problem definition in the field `fem.opt`. If there is an OPT structure supplied in the argument list using the `Opt` property, `fem.opt` is ignored. Table 1-32 lists the valid fields in the OPT scructure.

TABLE 1-32: VALID FIELDS IN THE OPT STRUCTURE

FIELD	VALUE	DESCRIPTION
`names.obj`	string	Objective function name
`names.constr`	cell array of strings	Scalar constraint names
`names.constrlb`	cell array of strings	Scalar constraint lower bounds
`names.construb`	cell array of strings	Scalar cosntraint upper bounds
`names.nlinconstr`	`on` | `off`	Assumption about constraint linearity

The names given in the OPT structure must be global scalar variables in the FEM problem. In addition to the scalar constraints, which can be general nonlinear functions of both FEM state variables and parameters, the FEM structure can specify pointwise constraints on the parameters that are assembled to matrices named NP and MP. Corresponding lower and upper bounds are assembled as MLB and MUB, respectively.

Pointwise constraints are assumed to be linear in the parameters and independent of the PDE state. However, if the `names.nlinconstr` field is set to `on`, nonlinear constraints can also be applied, but convergence might be slow.

The value of the property `Optprop` is a cell array of property/value pairs for the optimization solver. Table 1-33 lists the allowed properties. In the table, `m` refers to the number of constraints and `n1` to the number of nonlinear variables (computed internally by the solver). For detailed descriptions of the various properties, see the section "Optimization Solver Properties" on page 574.

TABLE 1-33: ALLOWED PROPERTY/VALUE PAIRS IN THE OPTPROP CELL ARRAY

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`cendiff`	numeric	`6.0e-6`	Central difference interval
`checkfreq`	integer	`60`	Check frequency
`diffint`	numeric	`1.5e-8`	Difference interval
`elasticw`	numeric	`1.0e4`	Elastic weight
`expfreq`	integer	`10000`	Expand frequency
`facfreq`	integer	`50`	Factorization frequency
`feastol`	numeric	`1.0e-6`	Minor feasibility tolerance
`funcprec`	numeric	`3.8e-11`	Function precision
`hessdim`	integer	`min(1000,n1+1)`	Hessian dimension
`hessfreq`	integer	`9999999`	Hessian frequency
`hessmem`	`full` | `limited`	`limited` if `n1` > 75 or `qpsolver` is `cg`	Hessian memory
`hessupd`	integer	10 if `hessmem` is `limited`	Hessian updates
`infbound`	positive numeric	`1.0e20`	Infinite bound size
`itlim`	integer	`500`	Minor iteration limit

TABLE 1-33: ALLOWED PROPERTY/VALUE PAIRS IN THE OPTPROP CELL ARRAY

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`linesearch`	`derivative` | `nonderivative`	`derivative`	Linesearch method
`linestol`	numeric	`0.9`	Linesearch tolerance
`majfeastol`	numeric	`1.0e-6`	Major feasibility tolerance
`majitlim`	integer	`max(1000,m)`	Major iterations limit
`majprintlevel`	integer	`1`	Controls the amount of output to the print file for the major iterations
`majprintlevel`	integer	`1`	Controls the amount of output to the print file for the major iterations
`majsteplim`	numeric	`2.0`	Major step limit
`maximize`	`on` | `off`	`off`	`on` if objective should be maximized
`newsuplim`	integer	`99`	New superbasics limit
`opttol`	numeric	`1.0e-6`	Optimality tolerance
`parprice`	integer	`1`	Partial price
`pivtol`	numeric	`3.7e-11`	Pivot tolerance
`print`	filename	Empty (no printing)	Print information about the solver progress and solution to file
`printfreq`	integer	`100`	Print frequency for the minor iterations

TABLE 1-33: ALLOWED PROPERTY/VALUE PAIRS IN THE OPTPROP CELL ARRAY

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`printlevel`	integer	`1`	Controls the amount of output to the print file for the minor iterations
`proxmeth`	`1` | `2`	`1`	Proximal point method
`qpsolver`	`cholesky` | `cg` | `qn`	`cholesky`	Specifies the active-set algorithm used to solve the QP subproblem; see below for details
`scaleopt`	`0` | `1` | `2`	`1`	Scale option
`scaletol`	numeric	`0.9`	Scale tolerance
`stop`	`on` | `off`	`on`	When `on`, deliver partial solution when failing
`suplim`	integer	`n1+1`	Superbasics limit
`totitlim`	integer	`max(10000,20*m)`	Iterations limit (absolute limit on the total number of minor iterations
`verify`	`-1` | `0` | `1` | `2` | `3`	`0`	Verification level of derivatives through finite differences. Derivatives are checked at the first point that satisfies all bounds and linear constraints.
`viollim`	numeric	`10`	Violation limit

The property `qpsolver` specifies the active-set algorithm for solving the QP subproblem as follows: `cholesky` indicates the Cholesky solver; `qn` indicates the quasi-Newton method; and `cg` uses an active-set method similar to `qn` but it uses the conjugate-gradient method to solve all systems involving the reduced Hessian.

The value of the property `verify` determines the verification level as follows: `-1` indicates that derivative checking is disabled; `0` indicates that only a cheap test is performed, requiring two calls to user functions; `1` indicates that individual gradients are checked with a more reliable test; `2` indicates that individual columns of the problem Jacobian are checked; and `3` indicates that both options `2` and 1 occur (in that order).

See Also `femsolver`, `femstatic`

Purpose

Description of properties common to all plot functions.

Description

Valid property/value pairs:

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Axis`	numeric vector		Axis limits
`Axisequal`	`on` | `off`	`on`	Axis equal
`Axislabel`	cell array of strings		X-, Y- and Z-axis labels
`Axisvisible`	`on` | `off`	`on`	Axis visible
`Camlight`	`on` | `off`	`off`	Light at camera position
`Campos`	1-by-3 numeric vector		Position of camera
`Camprojection`	`orthographic` | `perspective`	`perspective`	Projection
`Camtarget`	1-by-3 numeric vector		Camera aiming point
`Camup`	1-by-3 numeric vector		Rotation of the camera
`Camva`	numeric between 0 and 180	90	Field of view in degrees
`Grid`	`on` | `off`	`off`	Grid visible
`Lightmodel`	`flat` | `gouraud` | `phong` | `none`	`phong`	Lighting algorithm
`Lightreflection`	`dull` | `shiny` | `metal` | `default` | 1-by-3, 4, or 5 numeric vector	`default`	Reflectance of surfaces
`Parent`	axes handle		Handle to axes object
`Renderer`	`auto` | `painters` | `zbuffer` | `opengl`	`auto`	Rendering algorithm
`Scenelight`	`on` | `off`	`off`	Create scene light
`Scenelightpos`	1-by-3 numeric vector		Location of scene light object
`Title`	string	empty	Plot title
`Titlecolor`	color	`k`	Title color

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Transparency`	number between 0 and 1	1	Transparency (only has effect when using OpenGL)
`View`	2 or 3 | numeric pair	2 or 3	3D view point

Purpose Create a Simulink structure.

Syntax `sct = femsim(fem,...)`

Description `sct = femsim(fem,...)` creates a Simulink structure `sct` for the FEM structure `fem`.

To use the exported Simulink structure in Simulink, open the Blocksets & Toolboxes library in Simulink, double-click on the COMSOL Multiphysics icon, and drag the COMSOL Multiphysics Subsystem block to your Simulink model. Double click on your copy of the block, and enter the name of your Simulink Structure. This sets up the input and the output ports of the block.

The function `femsim` accepts the following property/value pairs:

TABLE 1-34: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Input`	cell array of strings	`fem.const`	Input variable names
`Keep`	string containing K, L, M, N, D, E | `auto`	`auto`	Time-independent quantities
`Nonlin`	`off` | `on` | `auto`	`auto`	Use nonlinear stationary solver
`Outnames`	cell array of strings	`y1,y2,...`	Output data names
`Output`	cell array	`{}`	Output data (see below)
`Redcomp`	cell array of strings		Degrees of freedom in model reduction of wave equations
`Redmodes`	integer	`10`	Number of eigenmodes in reduction
`Redstatic`	`off` | `on`	`on`	Use static modes in reduction
`Reduction`	`off` | `on`	`off`	Model reduction
`State`	`off` | `on`	`off`	Use linearized state-space model
`Static`	`off` | `on`	`off`	Use static solver
`T`	scalar	`0`	Time for evaluation of linearized model

In addition, the common solver properties described in the entry `femsolver` are supported, with modifications described below.

The names of the input variables are given in the property Input. The default is all names in `fem.const`.

The output values are determined by the entries in the `Output` cell array. Each entry in this cell array should be a cell array of the following form:

```
{'expr' xx pvlist}
```

where `'expr'` is an expression, `xx` is a column vector containing global coordinates, and `pvlist` is a (possibly empty) list of property/value pairs (see `postinterp`). The function `postinterp` is used to evaluate the output value as

```
postinterp('expr', xx, pvlist)
```

The names of the output ports of the COMSOL Multiphysics Subsystem block are given in the property `Outnames`. The default names are `y1`, `y2`, etc.

The COMSOL Multiphysics Subsystem block can act in four different modes, chosen by the properties `State` and `Static`:

General Dynamic Export (State=off, Static=off)
Export a dynamic model, where the COMSOL Multiphysics degrees of freedom are part of the Simulink state vector. The COMSOL Multiphysics solver is called several times for each time step to compute the time derivative of the state vector, with inputs from Simulink. Only linear, time-independent constraints and the `eliminate` constraint handling method is supported. The `Itol` property is supported, see `femstatic`. The `Const` property is not supported.

General Static Export (State=off, Static=on)
Export a static model, where the COMSOL Multiphysics degrees of freedom are not part of the Simulink state vector. To compute the outputs of the COMSOL Multiphysics Subsystem block, the COMSOL Multiphysics linear or nonlinear stationary solver is called for each time step, with inputs from Simulink. If `Nonlin=off`, the linear solver is used and the property `Itol` is supported, see `femstatic`. If `Nonlin=on`, the nonlinear solver is used and the properties `Hnlin`, `Initstep`, `Maxiter`, `Minstep`, `Ntol`, and `Rstep` are supported, see `femnlin`.

Linearized Dynamic Export (State=on, Static=off)
Export a dynamic linearized model. The model is linearized about an equilibrium solution, and the matrices in the state-space form are computed. The COMSOL Multiphysics degrees of freedom are part of the Simulink state vector. At each time step, the matrices in the state-space form are used to compute the time derivative of the state vector, instead of calling the COMSOL Multiphysics solver. Only linear,

time-independent constraints and the `eliminate` constraint handling method is supported. The `Const` property is not supported.

The linearization point should be an equilibrium point (stationary solution) and is controlled by the property `U`, see `femsolver`. The inputs and the outputs are deviations from the equilibrium values.

Model reduction can be used to approximate the linearized model with a model that has fewer degrees of freedom, by using `Reduction=on`. A number of eigenmodes (given by the property `Redmodes`) and static modes (if `Restatic=on`) will then be computed, and the linearized model will be projected onto the corresponding subspace. The properties `Etol`, `Itol`, `Krylovdim`, and `Shift` of the eigenvalue solver are supported, see `femeig`.

When using model reduction on wave equation models that have been rewritten as a system of first-order equations (wave extension), the algorithm needs to know the names of the original (non time derivative) solution components. The names of the non-time derivative solution components should be specified using the property `Redcomp`. Since COMSOL Multiphysics 3.2, wave equations are usually formulated without wave extension; then the `Redcomp` property should not be used.

Linearized Static Export (State=on, Static=on)

Export a static linearized model. The model is linearized about an equilibrium solution, and a transfer matrix is computed. The COMSOL Multiphysics degrees of freedom are not part of the Simulink state vector. To compute the outputs of the COMSOL Multiphysics Subsystem block, the transfer matrix is used.

The linearization point should be an equilibrium point (stationary solution) and is controlled by the property `U`, see `femsolver`. The inputs and the outputs are deviations from the equilibrium values.

Example

Heat equation with heat source Q as input.

```
fem.geom = solid1([0 1]); fem.mesh = meshinit(fem);
fem.shape = 2; fem.equ.da = 1; fem.equ.c = 1; fem.equ.f = 'Q';
fem.xmesh = meshextend(fem);
% Temperature u at x = 0.5 is output
sct = femsim(fem, 'input',{'Q'}, 'outnames',{'Temp'}, ...
             'output',{{'u' 0.5}});
```

Compatibility

For backward compatibility, the `Input` property can also be a vector of indices into `fem.const`.

Most of the FEMLAB 2.3 data types in the `Output` cell array are still supported:

TABLE 1-35: FEMLAB 2.3 OUTPUT DATA TYPES

ENTRY IN OUTPUT CELL ARRAY	INTERPRETATION
Cell array `{N iName}`	The solution component `fem.dim{iName}` at mesh vertex number `N` in `fem.mesh` or `fem.fem{g}.mesh`, where `g` is the geometry given in the `Geomnum` property (default is `1`)
Integer `N`	Shortcut for `{N 1}`. That is, solution component `fem.dim{1}` at mesh vertex number `N`
Struct `lfun`	Linear functional, is no longer supported. Use an integration coupling variable instead
Cell array `{N 'expr'}`	Value of expression `'expr'` at mesh vertex number `N` in the geometry given in the `Geomnum` property
String `func`	Value of function `func(fem,u,t,indata)`, is no longer supported

The properties `Mass` and `Timescale` are no longer supported.

A Simulink structure with `State=off` can no longer be saved to file using the commands `save` or `flsave`.

See Also `femsolver`, `femlin`, `femnlin`, `femeig`, `femtime`, `femstate`

Purpose Create a solution object.

Syntax

```
fem.sol = femsol(u)
fem.sol = femsol(u,'tlist',tlist)
fem.sol = femsol({u ut},'tlist',tlist)
fem.sol = femsol(u,'plist',plist,'pname',plist)
fem.sol = femsol(u,'lambda',lambda)
```

Description

`fem.sol = femsol(u)` creates a stationary solution object from a column vector `u`. The length of u must equal number of degrees of freedoms in the extended mesh object, `fem.xmesh`, (see `flngdof`).

`fem.sol = femsol(u,...)` stores a matrix corresponding to a time-dependent, parametric, or eigenvalue solution in the solution object. The number of rows must equal the number of degrees of freedoms in the extended mesh object, `fem.xmesh`, (see `flngdof`) and the number of columns of `u` must equal the number of time steps, parameter values, or eigenvalues, respectively (see `solsize`).

`fem.sol = femsol({u ut},...)` creates a time-dependent solution object containing also the first time derivative. The matrix `u` is the usual solution matrix, and `ut` is its time derivative.

`fem.sol = femsol(u,'mcase',mcase)` sets the mesh case of the created solution object to `mcase`. The default mesh case is 0.

Access Functions

The following access functions lets you fetch properties from the solution object.

TABLE 1-36: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUES
`fem.sol.u`	Solution vector or matrix
`fem.sol.ut`	Matrix containing time derivative (time-dependent solutions)
`fem.sol.tlist`	List of time steps (time-dependent solutions)
`fem.sol.plist`	List of parameter values (parametric solutions)
`fem.sol.pname`	Parameter name (parametric solutions)
`fem.sol.lambda`	List of eigenvalues (eigenvalue solutions)
`fem.sol.mcase`	Mesh case
`fem.sol.reacf`	Reaction force vectors
`fem.sol.sens`	Forward sensitivity vectors
`fem.sol.adj`	Adjoint solution vector or matrix

TABLE 1-36: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUES
`fem.sol.fsens`	Functional sensitivity vectors
`fem.sol.sensidx`	Sensitivity variable indices in the solution vectors

If the time-derivatives have not been stored in the `femsol` object, `fem.sol.ut` is computed as the slope of the linear interpolation between the time steps. To store the time derivatives in the solution, use the `Outcomp` property, see `femsolver`.

When the solver property `reacf` is `on` (default), reaction forces are stored for degrees of freedom corresponding to nonzero rows in the constraint force Jacobian matrix, N_F. Remaining rows in `fem.sol.reacf` are `NaN`.

For a stationary solution with forward sensitivity analysis enabled, `fem.sol.sens` is an N-by-M-by-P array where N is the number of degrees of freedom, M is the number of sensitivity variables, and P is the number of parameter steps in the parametric solver output. The second index corresponds to positions in `fem.sol.sensidx`, which contains the sensitivity variable degree of freedom index for the corresponding column.

The adjoint sensitivity method creates the field `fem.sol.adj`, which has the same size as `fem.sol.u` and contains the corresponding adjoint solution. In addition, the adjoint method always generates functional sensitivities in `fem.sol.fsens`. Rows which correspond to sensitivity variables in this matrix contain derivatives of the sensitivity functional with respect to the sensitivity variable. Remaining rows are `NaN`. If the property `sensfunc` is specified, also the forward method generates `fem.sol.fsens`.

Example

Create a solution object:

```
fem.geom = rect2;
fem.mesh = meshinit(fem);
fem.shape = 2; fem.equ.c = 1; fem.equ.f = 1; fem.bnd.h = 1;
fem.xmesh = meshextend(fem);
fem.sol = femtime(fem,'tlist',0:0.1:1)
```

Fetch the solution vector and the list of time steps.

```
u = fem.sol.u;
tlist = fem.sol.tlist;
```

Multiply the solution by 2 and recreate a solution object.

```
fem.sol = femsol(2*u,'tlist',tlist);
```

Postprocess the solution.

```
  postplot(fem,'tridata','u')
```

Compatibility

In FEMLAB 2.3 the solution was represented with a MATLAB structure. The solution object does not allow exactly the same type access as the structure. The solution object has been designed to be compatible with the MATLAB structure.

See also

`asseminit`, `femeig`, `femlin`, `femnlin`, `femtime`

Purpose

Description of properties common to all solvers.

Description

In addition to the properties in the table below, the solvers accept properties controlling the linear system solvers, see the sections starting with "Linear System Solvers" on page 169.

TABLE 1-37: COMMON SOLVER PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Assemtol`	scalar	`1e-12`	Assembly tolerance
`Blocksize`	positive integer | `auto`	`auto`	Assembly block size
`Complexfun`	`off` | `on`	`off`	Use complex-valued functions with real input
`Conjugate`	`off` | `on`	`off`	Use complex conjugate, Hermitian transpose
`Const`	cell array of alternating strings and values, or a structure		Definition of constants
`Constr`	`auto` | `ideal` | `nonideal`	`auto`	Constraint force Jacobian
`Init`	solution object | numeric vector | scalar		Initial value
`Keep`	string containing K, L, M, N, D, E | `auto`	`auto`	Manual control of reassembly
`Linsolver`	`umfpack` | `spooles` | `pardiso` | `pardiso_ooc` | `taucs_llt_mf` | `taucs_ldlt` | `luinc` | `taucs_llt` | `gmres` | `fgmres` | `cg` | `bicgstab` | `amg` | `gmg` | `ssor` | `ssoru` | `sor` | `soru` | `jac` | `vanka`	`umfpack`	Linear system solver
`Matherr`	`off` | `on`	`on`	Error for undefined operations
`Mcase`	non-negative integer (or vector for GMG)	mesh case with largest number of DOFs	Mesh case to solve for

TABLE 1-37: COMMON SOLVER PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Method`	`eliminate` | `elimlagr` | `lagrange` | `spring`	`eliminate`	Constraint handling method
`Nullfun`	`flnullorth` | `flspnull` | `auto`	`auto`	Null space function
`Outcomp`	cell array of strings		Solution components to store in output
`Report`	`off` | `on`	`on`	Show progress dialog box
`Rowscale`	`off` | `on`	`on`	Equilibrate rows
`Solcomp`	cell array of strings		Solution components to solve for
`Solfile`	`off` | `on`	`off`	Store solution on file
`Solfileblock`	positive scalar	`16`	Max size of solution block (MB)
`Stop`	`off` | `on`	`on`	Deliver partial solution when failing
`Symmetric`	`on` | `off` | `auto`	`auto`	Symmetric matrices
`Symmtol`	non-negative scalar	`1e-10`	Symmetry detection tolerance
`U`	solution object | numeric vector | scalar		Values of variables not solved for and linearization point
`Uscale`	`auto` | `init` | `none` | cell array | solution vector	`auto`	Scaling of variables

In addition to the constants in `fem.const`, you can define constants using the `Const` property, see `assemble`.

The parameter `Constr` controls how the constraint force Jacobian is computed. For `auto` and `nonideal`, the constraint force Jacobian matrix N_F is assembled independently of the constraint Jacobian matrix N. When `auto` is selected, a comparison between N_F and N^T is performed. If these matrices are found equal (up to a tolerance), then N_F is cleared and $N_F = N^T$. For `ideal`, only the constraint Jacobian matrix N is assembled and $N_F = N^T$.

If the solver fails to find a complete solution, it returns a partial solution if the property `Stop` is `on` (this is the default).

You can use the property `Symmetric` to tell the solver that the model is symmetric or you can use the automatic feature to find out. The symmetry detection tolerance `Symmtol` is used for this automatic feature (see "Which Problems Are Symmetric?" on page 431 in the *COMSOL Multiphysics User's Guide*). If the model is Hermitian, you should set both the `Symmetric` and `Conjugate` properties to `on`.

PROPERTIES CORRESPONDING TO THE SOLVER MANAGER

The properties `Init`, `U`, `Solcomp`, and `Outcomp` correspond to settings in the Solver Manager (see "The Solver Manager" on page 436 in the *COMSOL Multiphysics User's Guide*).

The property `Init` determines the initial value for the solution components you solve for. For possible syntaxes, see `asseminit`. If you omit this property, the solver computes the initial value by evaluating the initial value expressions in `fem.equ.init`, `fem.equ.dinit`, `fem.bnd.init`, `fem.bnd.dinit`, etc. If any of these expressions depend on a solution component, the value 0 is used for that solution component.

The property `U` determines the value of solution component you do not solve for and the linearization point. A scalar value is equivalent to a solution vector containing that value in all its components.

The property `Solcomp` is a cell array containing the names of the degrees of freedom to solve for. The default is all degrees of freedom.

The property `Outcomp` is a cell array containing the names of the degrees of freedom to store in the output solution object. If the solution is time dependent, the property `Outcomp` can contain also the time derivatives of the DOF names. The default is all degrees of freedom (excluding the time derivatives).

THE PROGRESS WINDOW

By default, a progress window appears when a solver is called. This is similar to the progress window that appears in the COMSOL Multiphysics user interface (see "Solution Progress" on page 446 in the *COMSOL Multiphysics User's Guide*). The progress window gives you the possibility to cancel or stop the solver. Also, when running in MATLAB, it gives you the possibility to examine the convergence in a plot. Using the properties specified under Probe Plot Parameters below, in addition, the progress window gives you the possibility to plot values of certain quantities during the solution process for the time dependent and parametric solver. If you do not want the progress window, use the `Report` property or the `flreport` command (see `flreport`).

PROBE PLOT PARAMETERS

Properties to the time-dependent solver and the parametric solver:

PROPERTY NAME	PROPERTY VALUE	DESCRIPTION
`plotglobal`	String or cell array of strings	Global plot expressions
`plotglobalpar`	Cell array or cell array of cell arrays	Property/values to postglobaleval
`plotint`	String or cell array of strings	Integration plot expressions
`plotintpar`	Cell array or cell array of cell arrays	Property/values to postint
`plotinterp`	String or cell array of strings	Probe plot expressions
`plotinterppar`	Cell array or cell array of cell arrays	Property/values to postinterp
`plotsum`	String or cell array of strings	Summation plot expressions
`plotsumpar`	Cell array or cell array of cell arrays	Property/values to postsum

The properties `plotglobalpar`, `plotinpar`, `plotsumpar`, and `plotinterppar` must be a cell array containing property/values to the corresponding evaluating function `postglobaleval`, `postint`, `postsum`, and `postinterp`, respectively. In addition, the property `title` can be specified. Also, `plotinterppar` must contain the property `probecoord` with the value being an `sdim`-by-`n` coordinate matrix. If, for example, `plotint` is a cell array of expressions to plot, `plotintpar` can either be a cell array of property/values, or, if different property/values are to be specified for the different expressions, a cell array of cell arrays of property/values.

An example of a `femstatic` call for a parametric problem specifying one global expression, one interpolation expression, and two integration expressions.

```
fem.sol = femstatic(fem, ...
   'solcomp',{}, ...
   'outcomp',{}, ...
   'plotglobal',{'w1','w2'}, ...
   'plotglobalpar',{'phase',pi,'title','Global'}, ...
   'plotinterp','u', ...
   'plotinterppar',{'probecoord',[0.1; 0.2]}, ...
   'plotint',{'u','ux','uy'}, ...
   'plotintpar',{{'phase',0, 'edim',0,'dl',4}, ...
{'phase',pi,'edim',1,'intorder',5},...
{'phase',pi,'edim',2,'dl',[3,6,7]}});
```

ADVANCED SOLVER PARAMETERS

The section "Advanced Solver Settings" on page 526 describes the features corresponding to the properties `Blocksize`, `Complexfun`, `Conjugate`, `Keep`, `Method`, `Nullfun`, `Rowscale`, `Solfile`, and `Uscale`.

The property `Assemtol` affects the assembled matrices, see `assemble` for details.

By default, COMSOL Multiphysics gives an error message if the solver encounters an undefined mathematical operation when solving the model, for instance `0/0` or `log(0)`. If you instead want the solver to proceed, put the property `Matherr=off`. Then `0/0=NaN` (not a number) and `log(0)=-Inf`.

The `Symmetric` and `Conjugate` properties correspond to the **Solver Parameters** dialog box settings **Matrix symmetry** and **Use Hermitian transpose in constraint matrix and in symmetry detection** according to the following table:

MATRIX SYMMETRY	USE HERMITIAN TRANSPOSE	SYMMETRIC	CONJUGATE
Automatic	cleared	`auto`	`off`
Automatic	selected	`auto`	`on`
Nonsymmetric	cleared	`off`	`off`
Nonsymmetric	selected	`off`	`on`
Symmetric	n.a.	`on`	`off`
Hermitian	n.a.	`on`	`on`

The property `Keep` corresponds to the manual control of reassembly feature. Its value can be a string containing the letters `D`, `E`, `K`, `L`, `M`, `N`, or the string `auto`. These letters have the following meaning: `E`=constant mass, `D`=constant damping, `K`=constant Jacobian, `L`=constant load, `M`=constant constraint, `N`=constant constraint Jacobian (see "Manual Control of Reassembly" on page 530).

For the `Nullfun` property, `flnullorth` is the orthonormal null-space function, and `flspnull` is the sparse null-space function.

If `Solfile=on`, the solution is stored on a temporary file. The temporary file is stored in the temporary directory created at startup. You can decide the temporary directory with the `-tmpdir` switch; see page 53 of the *COMSOL Installation and Operations Guide* for further details). A part of the solution is stored in memory in a few *blocks* (usually 1–5 blocks reside in memory). The maximum block size (in megabytes) can be controlled with the property `Solfileblock`.

The property `Uscale` determines a scaling of the degrees of freedom that is applied in order to get a more well-conditioned system; see “Scaling of Variables and Equations” on page 531. The possible values are:

TABLE 1-38: VALUES FOR THE PROPERTY USCALE

VALUE	MEANING
`auto`	The scaling is automatically determined
`init`	The scaling is determined from the initial value. Use this if the sizes of the components of the initial value give a good estimate of the order of magnitude of the solution
`none`	No scaling is applied
cell array	A cell array with alternating degree of freedom names and positive numbers. The numbers specify the expected magnitude of the corresponding degree of freedom
solution vector	A numeric vector with positive components that specify the expected magnitude of the solution

The default is `auto`, except when using one of the syntaxes

```
[Ke,Le,Null,ud] = femstatic(fem,...)
[Kl,Ll,Nnp] = femstatic(fem,...)
[Ks,Ls] = femstatic(fem,...)
fem.sol = femstatic('In',{'K' K 'L' L 'M' M 'N' N},...)
```

which assume that the property `Out` is not given in the first three cases. In these cases the default is `none`. The resulting vector of scale factors is contained in the output variable `uscale`. The scaling of the degrees of freedom is applied symmetrically to the Jacobian matrix, that is, both the rows and columns are scaled.

LINEAR SYSTEM SOLVERS

The properties `Linsolver`, `Prefun`, `Presmooth`, `Postsmooth`, and `Csolver` select the linear system solver, preconditioner, presmoother, postsmoother, and coarse solver, according to the following table.

TABLE 1-39: LINEAR SYSTEM SOLVERS/PRECONDITIONERS/SMOOTHERS

NAME	ALGORITHM
`umfpack`	UMFPACK direct solver
`spooles`	SPOOLES direct solver
`pardiso`	PARDISO direct solver

TABLE 1-39: LINEAR SYSTEM SOLVERS/PRECONDITIONERS/SMOOTHERS

NAME	ALGORITHM
`pardiso_ooc`	PARDISO Out-of-core solver
`taucs_llt_mf`	TAUCS direct Cholesky solver
`taucs_ldlt`	TAUCS direct LDLT solver (not recommended)
`luinc`	Incomplete LU preconditioner/smoother
`taucs_llt`	TAUCS Incomplete Cholesky preconditioner
`gmres`	GMRES iterative solver
`fgmres`	FGMRES iterative solver
`cg`	Conjugate Gradients iterative solver
`bicgstab`	BiCGStab iterative solver
`amg`	Algebraic Multigrid iterative solver/preconditioner
`gmg`	Geometric Multigrid iterative solver/preconditioner
`ssor`	SSOR preconditioner/smoother
`ssoru`	SSORU preconditioner/smoother
`sor`	SOR preconditioner/smoother
`soru`	SORU preconditioner/smoother
`jac`	Jacobi (diagonal scaling) preconditioner/smoother
`ssorvec`	SSOR vector preconditioner/smoother
`sorvec`	SOR vector preconditioner/smoother
`soruvec`	SORU vector preconditioner/smoother
`ssorgauge`	SSOR gauge preconditioner/smoother
`sorgauge`	SOR gauge preconditioner/smoother
`sorugauge`	SORU gauge preconditioner/smoother
`vanka`	Vanka-type preconditioner/smoother

For a description of these solvers, see the section “The Linear System Solvers” on page 426 in the *COMSOL Multiphysics User’s Guide*.

DIRECT LINEAR SYSTEM SOLVER PROPERTIES

The umfpack, spooles, pardiso, pardiso_ooc, taucs_llt_mf, luinc, and taucs_llt direct linear solvers/preconditioners/smoothers have the following properties.

TABLE 1-40: DIRECT LINEAR SOLVERS PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
Droptol	scalar between 0 and 1	0.01 when used as pre-conditioner or smoother, 0 when used as solver	Drop tolerance (luinc, taucs_llt, umfpack, spooles)
Errorchk	on | off | auto	on	Check error estimate (pardiso)
Errorchkd	on | off	off	Check error estimate (umfpack, spooles)
Fillratio	non-negative scalar	2	Column fill-ratio (luinc)
Itol	positive real	1e-6 0.1 (coarse solver)	Error check tolerance (pardiso, umfpack, spooles)
Maxdepth	positive integer	10000	Maximum recursion depth (taucs_llt_mf)
Modified	on | off	off	Modified incomplete Cholesky (taucs_llt)
Oocmemory	positive real	512.0	Out-of-core memory (pardiso_ooc)
Pivotperturb	scalar between 0 and 1	1e-8	Pivot perturbation threshold (pardiso)
Pivotrefines	non-negative integer	0	Number of forced iterative refinements
Pivotstrategy	on | off	on	Use 2-by-2 Bunch-Kaufmann pivoting (pardiso)
Preorder	mmd | nd | ms | both	nd	Preordering algorithm (spooles)

TABLE 1-40: DIRECT LINEAR SOLVERS PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Pardreorder`	`mmd` | `nd`	`nd`	Preordering algorithm (`pardiso`)
`Pardrreorder`	`on` | `off`	`on`	Row preordering algorithm (`pardiso`)
`Respectpattern`	`on` | `off`	`on`	Do not drop original nonzeros (`luinc`)
`Rhob`	scalar > 1	400 1 (coarse solver)	Factor in linear error estimate (`pardiso`, `umfpack`, `spooles`)
`Thresh`	scalar between 0 and 1	`0.1` (`umfpack`, `spooles`) `1.0` (`luinc`)	Pivot threshold (`umfpack`, `spooles`, `luinc`)
`Umfalloc`	non-negative scalar	`0.7`	Memory allocation factor (`umfpack`)

ITERATIVE LINEAR SYSTEM SOLVER PROPERTIES

The iterative linear solvers/preconditioners/smoothers `luinc`, `gmres`, `fgmres`, `cg`, bicgstab, `amg`, `gmg`, `ssor`, `ssoru`, `sor`, `soru`, `jac`, `ssorvec`, `sorvec`, `soruvec`, `ssorgauge`, `sorgauge`, `sorugauge`, `vanka` have the following properties.

TABLE 1-41: ITERATIVE LINEAR SOLVERS PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Divcleantol`	positive real	`1e-6`	Divergence cleaning tolerance (SOR gauge algorithms)
`Iluiter`	non-negative integer	1	Fixed number of iterations (when used as preconditioner, smoother, or coarse solver) (`luinc`)
`Iter`	non-negative integer	2	Fixed number of iterations (when used as preconditioner, smoother, or coarse solver) (all except `luinc`)

TABLE 1-41: ITERATIVE LINEAR SOLVERS PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Itol`	positive real	`1e-6` `0.1` (coarse solver)	Relative tolerance (note that when used as preconditioner or smoother a fixed number of iterations is default)
`Itrestart`	positive integer	`50`	Number of iterations before restart (`gmres`, `fgmres`)
`Maxlinit`	positive integer	`10000` `500` (coarse solver)	Maximum number of linear iterations (when used with a tolerance)
`Prefun`	`luinc` | `taucs_llt` | `umfpack` | `spooles` | `gmres` | `fgmres` | `cg` | `bicgstab` | `amg` | `gmg` | `ssor` | `ssoru` | `sor` | `soru` | `jac` | `ssorvec` | `sorvec` | `soruvec` | `ssorgauge` | `sorgauge` | `sorugauge` | `vanka` | `none`	`luinc`	Preconditioner (`gmres`, `fgmres`, `cg`, `bicgstab`)
`Prefuntype`	`left` | `right`	`left` (`gmres`, cg) `right` (bicgstab)	Left or right preconditioning (`gmres`, `cg`, `bicgstab`)
`Prepar`	cell array of property/ value pairs or structure		Preconditioner properties (`gmres`, `fgmres`, `cg`, `bicgstab`)
`Relax`	scalar between 0 and 2	`1`	Relaxation factor (Jacobi, SOR-based algorithms, incomplete LU, and Vanka)
`Rhob`	scalar >= 1	`400` `1` (coarse solver)	Factor in linear error estimate (when used with a tolerance)

TABLE 1-41: ITERATIVE LINEAR SOLVERS PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Seconditer`	nonnegative integer	`1`	Number of secondary iterations (SOR vector and SOR gauge algorithms), number of SSOR updates (`vanka`)
`Sorblocked`	`on` | `off`	`on`	Blocked SOR method
`Sorvecdof`	cell array of strings		Vector element variables (SOR vector and SOR gauge algorithms)
`Vankablocked`	`on` | `off`	`on`	Blocked Vanka method
`Vankarelax`	scalar between 0 and 2	`0.8`	Relaxation factor for Vanka update
`Vankarestart`	positive integer	`100`	GMRES restart value (`vanka`)
`Vankasolv`	`gmres` | `direct`	`gmres`	Local block solver (`vanka`)
`Vankatol`	positive scalar	`0.02`	GMRES tolerance (`vanka`)
`Vankavars`	cell array of strings	`{}`	Lagrange multiplier variables (`vanka`)

The property `Divcleantol` is used in the inequality $|T^T b| < \mathrm{divcleantol} \cdot |b|$ to ensure that the numerical divergence after divergence cleaning is small enough, see "The SSOR Gauge, SOR Gauge, and SORU Gauge Algorithms" on page 569.

MULTIGRID SOLVER PROPERTIES

The multigrid solvers/preconditioners amg and gmg accept the following properties in addition to those in table Table 1-41.

TABLE 1-42: MULTIGRID SOLVERS PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
amgauto	integer between 1 and 10	3	Quality of multigrid hierarchy (amg)
csolver	umfpack | spooles | pardiso | pardiso_ooc | taucs_llt_mf | taucs_ldlt | luinc | taucs_llt | gmres | fgmres | cg | bicgstab | amg | ssor | ssoru | sor | soru | jac | ssorvec | sorvec | soruvec | ssorgauge | sorgauge | sorugauge | vanka	umfpack	Coarse solver
csolverpar	cell array with property/ value pairs	{}	Coarse solver properties
maxcoarsedof	positive integer	5000	Maximum number of DOFs at coarsest level (amg)
meshscale	vector of positive numbers	2	Mesh scale factor (gmg)
mgassem	on | off | numeric vector	on	Assembly on coarse levels (gmg)
mgauto	off | explicit | meshscale | shape | both | meshrefine		Method for mesh case generation (gmg)
mgcycle	v | w | f	v	Cycle type
mggeom	vector of positive integers	all	Geometry numbers for multigrid hierarchy (gmg)
mgkeep	on | off	off	Keep generated mesh cases (gmg)

TABLE 1-42: MULTIGRID SOLVERS PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`mglevels`	integer > 1	6 (`amg`), 2 (`gmg`)	Maximum number of multigrid levels
`postsmooth`	`ssor` | `ssoru` | `sor` | `soru` | `jac` | `ssorvec` | `sorvec` | `soruvec` | `ssorgauge` | `sorgauge` | `sorugauge` | `luinc` | `gmres` | `fgmres` | `cg` | `bicgstab` | `amg` | `vanka`	`soru`	Postsmoother
`postsmoothpar`	cell array with property/ value pairs	`{}`	Postsmoother properties
`presmooth`	`ssor` | `ssoru` | `sor` | `soru` | `jac` | `ssorvec` | `sorvec` | `soruvec` | `ssorgauge` | `sorgauge` | `sorugauge` | `luinc` | `gmres` | `fgmres` | `cg` | `bicgstab` | `amg` | `vanka`	`sor`	Presmoother
`presmoothpar`	cell array with property/ value pairs	`{}`	Presmoother properties
`rmethod`	`regular` | `longest`	`regular`	Mesh refinement method (`gmg`)
`shapechg`	vector of integers	`-1`	Change in shape function orders (`gmg`)

For the Geometric multigrid solver/preconditioners, the construction of the multigrid hierarchy is controlled by the properties `Mgauto`, `Mcase`, `Mglevels`, `Meshscale`, `Shapechg`, and `Mgassem`:

- If `Mgauto=both`, `shape`, or `meshscale`, then the multigrid hierarchy is automatically constructed starting from the mesh case given in the property `Mcase`. This process is described in the section "Constructing a Multigrid Hierarchy" on page 560, where the methods are called **Coarse mesh and lower order** (`both`), **Lower element order first** (`shape`), and **Coarse mesh** (`meshscale`). The mesh coarsening factor is given in the scalar `Meshscale`, the shape function order change amount is given in the scalar `Shapechg`, and the number of multigrid levels (including the finest level) is given in the property `Mglevels` (default 2).

- If Mgauto=explicit, then the multigrid hierarchy is automatically constructed starting from the mesh case given in the property Mcase. The properties Meshscale and Shapechg should be vectors of the same length (however if one is scalar, it is expanded to the same length as the other). A number of coarse levels are constructed, where level i has a mesh that is coarsened with the factor Meshscale(i), and shape functions orders incremented with Shapechg(i) relative to mesh case Mcase. Shapechg(i) should be negative or zero.
- If Mgauto=meshrefine, then the multigrid hierarchy is automatically constructed by refining the mesh in mesh case Mcase repeatedly. The number of multigrid levels (including the original, coarsest level) is given in the property Mglevels (default 2). The refinement method can be specified using the property Rmethod, see meshrefine.
- If Mgauto=off, then only existing mesh cases are used in the hierarchy. If the property Mcase is a scalar, then all mesh cases that have fewer degrees of freedom than the mesh case Mcase are used as coarse levels. If Mcase is a vector with more than one component, the mesh cases in that vector are used. However, if the property Mglevels is given, no more than Mglevels levels are used. The solver sorts the list of mesh cases according to decreasing number of DOFs, and the solution is delivered for the mesh case with the largest number of DOFs. This corresponds to the **Manual** option in the COMSOL Multiphysics user interface.

The default for Mgauto is as follows: If the FEM structure has several mesh cases, then Mgauto=off, otherwise Mgauto is the same as the default in meshcaseadd.

The construction of coarse level matrices is controlled by the property Mgassem. If Mgassem is a vector, Mgassem(i) should be a 0 or 1. Mgassem(i)=1 means that matrices should be assembled in mesh case Mcase(i), rather than being projected from the next finer level. The length of Mgassem should be (at least) the number of mesh cases used, including the finest level. The value of Mgassem(i) for the finest level i is ignored, because matrices are always assembled on the finest level. A scalar Mgassem applies to all coarse mesh cases.

When an iterative solver is used as preconditioner, smoother, or coarse solver you can choose whether to solve using a tolerance or to perform a fixed number of iterations. When used as a coarse solver the default is to solve using a tolerance. When used as a preconditioner or smoother the default is to perform a fixed number of iterations. If both properties Itol and Iter (or Iluiter for luinc) are given, the program will solve using a tolerance.

Four Examples How to Construct the Geometric Multigrid Hierarchy

Assume that `fem` only contains the mesh case 0, and no extended mesh (`xmesh` field).

Alternative 1:

```
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem,'linsolver','gmg');
```

This alternative uses a temporary hierarchy that is constructed by the solver. Because the solver also constructs a temporary extended mesh, this alternative wastes some memory.

Alternative 2:

```
fem = femstatic(fem,'linsolver','gmg','out','fem');
```

Here, the solver uses a temporary hierarchy, but there is only one extended mesh. If another such solver call is made, first delete `fem.xmesh` to save some memory.

Alternative 3:

```
fem =
femstatic(fem,'linsolver','gmg','mgkeep','on','out','fem');
```

Now the generated hierarchy is kept, which means that you can reuse it in a subsequent call:

```
fem.sol = femstatic(fem,'linsolver','gmg');
```

Alternative 4:

```
fem = meshcaseadd(fem);
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem,'linsolver','gmg');
```

The `meshcaseadd` call adds mesh cases to the FEM structure. These mesh cases form the multigrid hierarchy in the solver.

Compatibility

COMSOL Multiphysics 3.2: The default of the `Conjugate` property has been changed to `off`.

The following FEMLAB 2.3 general solver properties are obsolete in FEMLAB 3.0:

TABLE 1-43: OBSOLETE PROPERTY/VALUE PAIRS

PROPERTY	VALUES	IMPLICATION
`Initmethod`	`weak` | `pointwise` | `local` | `dof`	No longer supported
`Itsolv`	`gbit` | `gmres` | `tfqmr`	Use `Linsolver` property

TABLE 1-43: OBSOLETE PROPERTY/VALUE PAIRS

PROPERTY	VALUES	IMPLICATION
`Jacobian`	`lumped` | `numeric`	No longer supported
`Linsolver`	`matlab` | `superlu`	Uses default direct solver
`Maxlinit`	vector with 2 components	Ignores second component and warns
`Nullfun`	name of user-defined function	No longer supported
`Sd`		Use streamline diffusion, see the chapter "Stabilization Techniques" on page 481 in the COMSOL Multiphysics Modeling Guide

The default of the property `Stop` has been changed to `On`.

Purpose

Compute state-space form of a time-dependent PDE problem.

Syntax

```
[A,B,C,D] = femstate(fem,...)
[M,MA,MB,C,D] = femstate(fem,...)
[M,MA,MB,C,D,Null,ud,x0] = femstate(fem,...)
state = femstate(fem,...)
```

Description

`[A,B,C,D] = femstate(fem,...)` calculates the linearized state-space form of the dynamic PDE model `fem` on the format

$$\begin{cases} \dfrac{dx}{dt} = Ax + Bu \\ y = Cx + Du \end{cases}$$

where x are the state variables, u are the input variables, and y are the output variables.

`[M,MA,MB,C,D] = femstate(fem,...)` calculates the state-space form on the format

$$\begin{cases} M\dfrac{dx}{dt} = MAx + MBu \\ y = Cx + Du \end{cases}$$

The matrices M and MA are usually much sparser than the matrix A.

`[M,MA,MB,C,D,Null,ud,x0] = femstate(fem,...)` also returns the null-space matrix `Null`, the constraint contribution `ud`, and the initial state `x0`. The full solution vector `U` can be obtained from the state variables by `U = Null*x+u0`, where `u0` is the linearization point.

`state = femstate(fem,...)` returns the structure `state` containing the fields `M`, `MA`, `MB`, `C`, `D`, `Null`, and `x0`.

`state = femstate(fem,'out','statenom',...)` returns the structure `state` containing the fields `A`, `B`, `C`, `D`, `Null`, and `x0`.

`s = femstate(fem,'out','ss',...)` returns the Control System Toolbox state-space object `s = ss(A,B,C,D)`.

The output from `femstate` is intended for use from Simulink or the Control System Toolbox. The function `femstate` with the output `state` is equivalent to `femsim` with the property `State=on`. In addition to the properties of `femsim`, `femstate` accepts the following property/value pairs:

TABLE 1-44: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Out`	`A` | `B` | `C` | `D` | `M` | `MA` | `MB` | `Null` | `ud` | `x0` | `state` | `statenom` | `ss` | cell array of these strings	`[A,B,C,D]` `[M,MA,MB,C,D]` `[M,MA,MB,C,D,Null,ud,x0]` `state`	Output variables
`Sparse`	`off` | `on`	`off`	Sparse matrices

The property `Sparse` controls whether the matrices `A`, `B`, `C`, `D`, `M`, `MA`, `MB`, and `Null` are stored in the sparse format. See `femsim` for a description of the other properties.

The matrices M and MA are produced by the same algorithms that do the finite-element assembly and constraint elimination in COMSOL Multiphysics. M and MA are the same as the matrices D_c (eliminated mass matrix) and $-K_c$ (K_c is the eliminated stiffness matrix), respectively, from a call to `femlin` (see `femlin` on page 131). The matrices are produced from an exact residual vector Jacobian calculation (that is, differentiation of the residual vector with respect to the degrees of freedoms x) plus an algebraic elimination of the constraints. The matrix C is produced in a similar way; that is, the exact output vector Jacobian matrix plus constraint elimination.

The matrices MB and D are produced by a numerical differentiation of the residual and output vectors, respectively, with respect to the input parameters (the algorithm systematically perturbs the input parameters by multiplying them by a factor $(1+10^{-8})$).

When exporting the A and B matrices, A and B are computed by $A = M \backslash MA$ and $B = M \backslash MB$ (that is, from an LU factorization of M using the UMFPACK solver.

Compatibility See the `femsim` entry.

See Also `femsim`

Purpose Solve stationary PDE problem with a nonlinear or linear solver.

Syntax

```
fem.sol = femstatic(fem,...)
fem = femstatic(fem,'Out',{'fem'},...)
```

Description `fem.sol = femstatic(fem)` solves a stationary PDE problem using either a linear or nonlinear solver.

`fem.sol = femstatic(fem,'pname','P', plist',list,...)` solves a stationary PDE problem for several values of the parameter `P`. The values of the parameter `P` are given in the vector `list`.

The PDE problem is stored in the (possibly extended) FEM Structure `fem`. See `femstruct` for details.

The function `femstatic` accepts the following property/value pairs:

TABLE 1-45: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Callback`	string		Function to call at each callback
`Callblevel`	`param` | `nonlin`		Callback solverlevel
`Callbparam`	cell array		Parameters to the callback function
`Llimitdof`	cell array of strings		Lower limit dofs
`Llimitval`	vector	`0`	Lower limit vals
`Maxsegiter`	positiv integer	`100`	Maximum number of segregated iterations
`Maxsubiter`	integer vector	`20`	Maximum number of substep iterations
`Nonlin`	`off` | `on` | `auto`	`auto`	Use the nonlinear solver
`Out`	`fem` | `sol` | `u` | `plist` | `stop` | `solcompdof` | `Kc` | `Lc` | `Null` | `Nnp` | `ud` | `uscale` | `nullfun` | `symmetric` | `nonlin` | `interrupted` | cell array of these strings	`sol`	Output variables
`Reacf`	`on` | `off`	`on`	Compute reaction forces

TABLE 1-45: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Segcomp`	cell array of strings		Segregated group components
`Seggrps`	cell array of cell array		Segregated group properties
`Segiter`	positive integer	`1`	Fixed number of segregated iterations
`Segorder`	integer vector		Segregated substep group numbers
`Segterm`	`iter` | `tol` | `itertol` | cell array of these strings	`tol`	Segregated solver termination technique
`Senscomp`	cell array of strings		Sensitivity components
`Sensfunc`	string		Sensitivity functional variable name
`Sensmethod`	`none` | `adjoint` | `forward`	`none`	Sensitivity analysis
`Subdamp`	real vector	`0.5`	Segregated substep damping factors
`Subdtech`	`const` | `autodamp` | cell array of these strings	`const`	Segregated substep damping technique
`Subhnlin`	`off` | `on` | cell array of these strings	`off`	Segregated substep indicators of highly nonlinear problems
`Subinitstep`	real vector	see below	Segregated substep initial damping factors
`Subiter`	integer vector	`1`	Segregated substep iterations
`Subjtech`	`minimal` | `once` | `onfirst` | `onevery` | cell array of these strings	see below	Segregated substep Jacobian update technique
`Subminstep`	real vector	see below	Segregated substep minimum damping factors
`Subntol`	real vector	`1e-2`	Segregated substep tolerances for stationary problem

TABLE 1-45: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Segcomp`	cell array of strings		Segregated group components
`Seggrps`	cell array of cell array		Segregated group properties
`Segiter`	positive integer	`1`	Fixed number of segregated iterations
`Segorder`	integer vector		Segregated substep group numbers
`Segterm`	`iter` | `tol` | `itertol` | cell array of these strings	`tol`	Segregated solver termination technique
`Senscomp`	cell array of strings		Sensitivity components
`Sensfunc`	string		Sensitivity functional variable name
`Sensmethod`	`none` | `adjoint` | `forward`	`none`	Sensitivity analysis
`Subdamp`	real vector	`0.5`	Segregated substep damping factors
`Subdtech`	`const` | `autodamp` | cell array of these strings	`const`	Segregated substep damping technique
`Subhnlin`	`off` | `on` | cell array of these strings	`off`	Segregated substep indicators of highly nonlinear problems
`Subinitstep`	real vector	see below	Segregated substep initial damping factors
`Subiter`	integer vector	`1`	Segregated substep iterations
`Subjtech`	`minimal` | `once` | `onfirst` | `onevery` | cell array of these strings	see below	Segregated substep Jacobian update technique
`Subminstep`	real vector	see below	Segregated substep minimum damping factors
`Subntol`	real vector	`1e-2`	Segregated substep tolerances for stationary problem

TABLE 1-45: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Segcomp`	cell array of strings		Segregated group components
`Seggrps`	cell array of cell array		Segregated group properties
`Segiter`	positive integer	`1`	Fixed number of segregated iterations
`Segorder`	integer vector		Segregated substep group numbers
`Segterm`	`iter` | `tol` | `itertol` | cell array of these strings	`tol`	Segregated solver termination technique
`Senscomp`	cell array of strings		Sensitivity components
`Sensfunc`	string		Sensitivity functional variable name
`Sensmethod`	`none` | `adjoint` | `forward`	`none`	Sensitivity analysis
`Subdamp`	real vector	`0.5`	Segregated substep damping factors
`Subdtech`	`const` | `autodamp` | cell array of these strings	`const`	Segregated substep damping technique
`Subhnlin`	`off` | `on` | cell array of these strings	`off`	Segregated substep indicators of highly nonlinear problems
`Subinitstep`	real vector	see below	Segregated substep initial damping factors
`Subiter`	integer vector	`1`	Segregated substep iterations
`Subjtech`	`minimal` | `once` | `onfirst` | `onevery` | cell array of these strings	see below	Segregated substep Jacobian update technique
`Subminstep`	real vector	see below	Segregated substep minimum damping factors
`Subntol`	real vector	`1e-2`	Segregated substep tolerances for stationary problem

TABLE 1-45: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Segcomp`	cell array of strings		Segregated group components
`Seggrps`	cell array of cell array		Segregated group properties
`Segiter`	positive integer	`1`	Fixed number of segregated iterations
`Segorder`	integer vector		Segregated substep group numbers
`Segterm`	`iter` | `tol` | `itertol` | cell array of these strings	`tol`	Segregated solver termination technique
`Senscomp`	cell array of strings		Sensitivity components
`Sensfunc`	string		Sensitivity functional variable name
`Sensmethod`	`none` | `adjoint` | `forward`	`none`	Sensitivity analysis
`Subdamp`	real vector	`0.5`	Segregated substep damping factors
`Subdtech`	`const` | `autodamp` | cell array of these strings	`const`	Segregated substep damping technique
`Subhnlin`	`off` | `on` | cell array of these strings	`off`	Segregated substep indicators of highly nonlinear problems
`Subinitstep`	real vector	see below	Segregated substep initial damping factors
`Subiter`	integer vector	`1`	Segregated substep iterations
`Subjtech`	`minimal` | `once` | `onfirst` | `onevery` | cell array of these strings	see below	Segregated substep Jacobian update technique
`Subminstep`	real vector	see below	Segregated substep minimum damping factors
`Subntol`	real vector	`1e-2`	Segregated substep tolerances for stationary problem

TABLE 1-45: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Segcomp`	cell array of strings		Segregated group components
`Seggrps`	cell array of cell array		Segregated group properties
`Segiter`	positive integer	`1`	Fixed number of segregated iterations
`Segorder`	integer vector		Segregated substep group numbers
`Segterm`	`iter` | `tol` | `itertol` | cell array of these strings	`tol`	Segregated solver termination technique
`Senscomp`	cell array of strings		Sensitivity components
`Sensfunc`	string		Sensitivity functional variable name
`Sensmethod`	`none` | `adjoint` | `forward`	`none`	Sensitivity analysis
`Subdamp`	real vector	`0.5`	Segregated substep damping factors
`Subdtech`	`const` | `autodamp` | cell array of these strings	`const`	Segregated substep damping technique
`Subhnlin`	`off` | `on` | cell array of these strings	`off`	Segregated substep indicators of highly nonlinear problems
`Subinitstep`	real vector	see below	Segregated substep initial damping factors
`Subiter`	integer vector	`1`	Segregated substep iterations
`Subjtech`	`minimal` | `once` | `onfirst` | `onevery` | cell array of these strings	see below	Segregated substep Jacobian update technique
`Subminstep`	real vector	see below	Segregated substep minimum damping factors
`Subntol`	real vector	`1e-2`	Segregated substep tolerances for stationary problem

TABLE 1-45: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Subntolfact`	real vector	1	Segregated substep tolerance factors for time-dependent problem
`Subrstep`	real vector	10	Segregated substep restrictions for stepsize updates
`Subterm`	`iter` | `tol` | `itertol` | cell array of these strings	`iter`	Segregated substep termination techniques

This solver uses the nonlinear solver described in `femnlin` if `nonlin` is `on`, and it uses the linear solver described in `femlin` if `nonlin` is `off`. If `nonlin` is set to `auto` an analysis is performed to automatically detect if the problem can be solved with the linear solver.

In addition to the properties listed above, also the properties for `femlin` or `femnlin` are supported, depending on which solver is used. For example, the linear solver does not support the properties `Augcomp`, `Augsolver`, `Augtol`, `Augmaxiter`, `Porder`, and `Ntol`. Similarly, the nonlinear solver does not support the property `In`. Furthermore, the properties described in the entry `femsolver` are supported.

If the property `callback` is given, the solver makes interrupts (or callbacks) and calls a function with the given name. To control when the solver makes a callback, use the `callblevel` property. When `callblevel` is `param`, the solver makes a callback after each parameter step that is exported to the output; this is the default for the parameter solver. When `callblevel` is `nonlin`, the solver makes a callback after each nonlinear (or segregated) iteration.; this is the default for the stationary solver. If the stationary problem is linear, no callback is done.

The automatic nonlinear/linear detection works in the following way. The linear solver is called if the residual Jacobian matrix (the stiffness matrix, K) and the constraint Jacobian matrix (the constraint matrix, N) are both found not solution dependent and if these matrices are detected as complete. In all other situations the nonlinear solver is used. The analysis is performed by a symbolic analysis of the expressions contributing to these matrices. Complete here means that in the residual and constraint vectors, only expressions where found for which COMSOL Multiphysics will compute the correct Jacobian contribution.

Therefore, if you want to solve a linearized (nonlinear) problem, you must select `nonlin` to `off`. Furthermore, there are variables for which COMSOL Multiphysics is conservative and will flag these, and their Jacobian contribution, as solution dependent even though they not always are. For these situations, the nonlinear solver will be used even though the linear solver could be used. This should only result in some extra computational effort, and should not influence the result. The opposite situation however, where the linear solver is used for a nonlinear problem is more dangerous. So, select `nonlin` to `off` with great care.

The property `Reacf` controls the computation and storage of constraint reaction forces. The value `Reacf=on` (default) means that the solver stores the FEM residual vector L in the solution object `fem.sol`. Because $L = N_F\Lambda$ for a converged solution, the residual is the same as the constraint force. Only the components of L that correspond to nonzero rows of N_F are stored. The value `Reacf=off` gives no computation or storage of the reaction force and saves some memory.

The property `Sensmethod` controls if also one of the two supported sensitivity analysis methods should be used. These methods reuses the Jacobian matrix from the primal solution step and performes a second linear solution step, by reusing the linear solvers. For the `adjoint` method the transpose of the matrix is solved for. The linear solver UMFPACK can solve for the transpose without extra initialization cost, while the SPOOLES and PARDISO methods performes a new factorization of the matrix. The method `adjoint` requires that a functional variable name is given by the property `Sensfunc`, while the method `forward` requires that at least one sensitivity parameter is given by the property `Senscomp`.

If the property `Augcomp` is given then the augmented lagrange solver is used, if the property `Seggrps` is given then the segregated solver is used and otherwise the standard nonlinear solver is used. These solvers are described in the *COMSOL Multiphysics User's Guide*; see "The Stationary Solver" on page 385, "The Parametric Solver" on page 405, "The Stationary Segregated Solver" on page 411, and "The Parametric Segregated Solver" on page 416.

The segregated solver group properties are given through the `Seggrps` property with one list for each group. The only mandatory property is the property `Segcomp` defining which solution components to be used for the group. Optional group properties are the standard solver properties; `Ntol`, `Linsolver`, `Prefun`, `Uscale`, and so on.

The segregated substeps are controlled by the property `Segorder` where the segregated group numbers should be given in the preferred solution order by an integer vector. The number of substeps is thereby determined by the length of the given integer vector. If this property is not given, the groups are solved for from first to last.

Termination of the segregated solver is controlled by the property `Segterm`. The default setting is `tol` in which case the segregated iterations are terminated when for each group, the estimated error is below the corresponding tolerance set through the group property `Ntol`, see "The Stationary Segregated Solver Algorithm" on page 539. However, a maximum number of allowed segregated iterations is chosen through the property `Maxsegiter`; if the maximum is reached, the iterations are terminated and an error message is displayed. Termination after a fixed number of segregated iterations is achieved by instead choosing `iter`. The number of segregated iterations is controlled by the property `Segiter`. The third available option for `Segterm` is `itertol`, which is a combination of the other two options; the segregated iterations are terminated when one of the two convergence criteria of `tol` and `iter` is met. Note that the property `Maxsegiter` is only supported when `tol` is used for termination. For both the settings `iter` and `itertol`, the number of iterations is controlled by the property `Segiter`.

Analogously, the property `Subterm` controls how each substep is terminated through the properties `Maxsubiter`, `Subiter`, and `Subntol/Subntolfact` for a stationary/time-dependent problem.

The damping technique used in each substep is controlled by the property `Subdtech`. The default setting is `const`, which means that damped Newton iterations with a fixed damping factor is used. The damping factor is set in the property `Subdamp`. The other available damping technique is `autodamp` in which case the damping factor is automatically adjusted. For substeps which uses `autodamp`, four other properties are supported: `Subhnlin`, `Subinitstep`, `Subminstep`, and `Subrstep`. For each substep, these properties set the properties `Hnlin`, `Initstep`, `Minstep`, and `Rstep` supported by the nonlinear solver, see `femnlin`.

In substeps with `Subdtech=const`, how often the Jacobian is updated is controlled by the property `Subjtech`. The values `minimal`, `once`, and `onevery` give the same Jacobian update techniques as they do when applied to the coupled solver through the property `Jtech`, see `femnlin` on page 143. The value `onfirst` makes the solver update the Jacobian of the substep on the first subiteration each time the substep is solved for. Default value is `onevery` for stationary problems and `minimal` for time-dependent problems.

The linear solver uses the property `Itol` for termination of iterative linear system solvers and for error checking for direct solvers (if enabled). The nonlinear solver uses an adaptive tolerance for termination of iterative linear system solvers. This adaptive tolerance is based on the maximum of `Ntol` and `Itol`. During the nonlinear iterations, it can, however, be larger or smaller than this number. The segregated solver uses the same tolerance as the linear solver when constant damping is used. However, when automatically adjusted damping is used, the adaptive tolerance of the nonlinear solver is used. The parametric solver uses the same tolerance as the corresponding stationary solver.

See Also `femlin`, `femnlin`, `femsolver`, `femstruct`, `assemble`, `asseminit`

Purpose FEM structure.

Syntax

```
help femstruct
```

Description The *FEM structure* is a container for the full description of a PDE problem. See "FEM Structure Overview" on page 4 in the *COMSOL Multiphysics MATLAB Interface Guide*.

Compatibility The fields `fem.equiv` and `fem.mat` are no longer supported. The `fem.rules` field is obsolete and replaced by `fem.functions`.

The field `fem.variables` has been renamed `fem.const` in FEMLAB 2.3.

Purpose Solve time-dependent PDE problem.

Syntax

```
fem.sol = femtime(fem,'Tlist',[t1 ... tn],...)
fem = femtime(fem,'Tlist',[t1 ... tn],'Out',{'fem'},...)
fem.sol = femtime('in',{'K' K 'N' N 'L' L 'M' M 'D' D 'E' E},...
                  'Tlist',[t1 ... tn],...)
```

Description `fem.sol = femtime(fem,...)` solves a time-dependent PDE problem.

The PDE problem is stored in the (possibly extended) FEM structure `fem`. See `femstruct` for details. The time interval and possible intermediate time values are given in the property `Tlist`. The output times are controlled by the property `Tout`.

`fem.sol = femtime('in',{'K' K 'N' N 'L' L 'M' M 'D' D 'E' E})` solves the pre-assembled linear problem

$$E\ddot{U} + D\dot{U} + KU = L - N_F\Lambda$$
$$NU = M$$

The function `femtime` accepts the following property/values:

TABLE 1-46: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Atol`	See below	`1e-3`	Absolute tolerance
`Callback`	string		Function to call at each callback
`Callbfreq`	`tout` | `tsteps`	`tout`	Callback frequency (callblevel=time)
`Callblevel`	`time` | `nonlin`	`time`	Callback solverlevel
`Callbparam`	cell array		Parameters to the callback function
`Complex`	`on` | `off`	`off`	Complex numbers
`Consistent`	`off` | `on` | `bweuler`	`bweuler`	Consistent initialization of DAE systems
`Estrat`	`0` | `1`	`0`	Error estimation strategy
`Eventout`	`on` | `off`	See below	Store both pre- and post-event solutions
`Eventtol`	numeric	`0.01`	Tolerance for event detection and location

TABLE 1-46: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`In`	cell array of names and matrices `K` | `L` | `M` | `N` | `D` | `E`	`N` and `M` are empty, `D=E=0`	Input matrices
`Incrdelay`	on | off	off	Use delay in time step increase
`Incrdelaysteps`	positive integer	15	Number of time steps to delay a time step increase
`Initialstep`	positive scalar		Initial time step
`Keep`	string containing K, L, M, N, D, E | `auto`	`auto`	Time-independent quantities
`MassSingular`	`yes` | `maybe`	`maybe`	Singular mass matrix
`Maxorder`	integer between 1 and 5	5	Maximum BDF order
`Maxreinit`	positive integer	`100`	Maximum number of reinitializations at an event
`Maxstep`	positive scalar		Maximum time step
`Minorder`	`1` | `2`	`1`	Minimum BDF order
`Nlsolver`	`automatic` | `manual`	`automatic`	Nonlinear solver settings
`Ntolfact`	positive scalar	`1`	Tolerance factor for solution of nonlinear system
`Odesolver`	`bdf_ida` | `genalpha` | `bdf_daspk`	`bdf_ida`	Time-stepping method
`Out`	`fem` | `sol` | `u` | `tlist` | `solcompdof` | `stop` | `Kc` | `Lc` | `Dc` | `Ec` | `Null` | `Nnp` | `ud` | `uscale` | `nullfun` | `symmetric` | cell array of these strings	`sol`	Output variables
`Predictor`	`linear` | `constant`	`linear`	Predictor
`Ratelimit`	positive scalar	`0.9`, `1` (segregated solver)	Limit on nonlinear convergence rate

TABLE 1-46: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Reacf`	`on` | `off`	`on`	Compute reaction forces
`Rhoinf`	numeric	`0.75`	Amplification factor for high frequencies
`Rtol`	numeric	`0.01`	Relative tolerance
`Stopcond`	string with expression | integer		Stop when expression becomes negative or when implicit event is triggered
`Timestep`	numeric scalar | numeric vector | string with expression	`0.01`	Time step when manual time stepping
`Tlist`	numeric vector		Time list
`Tout`	`tlist` | `tsteps`	`tlist`	Output times
`Tsteps`	`free` | `intermediate` | `strict` | `init` | `manual` (`genalpha`)	`free`	Time-stepping mode
`Uifscale`	`none` | `init` | cell array | solution vector	`none`	Scaling of indicator functions
`Useratelimit`	`on` | `off`	`on`	Use limit on nonlinear convergence rate

In addition, the properties described in the entry `femsolver` are supported.

When `Odesolver` equals `genalpha`, or when `Odesolver` equals `bdf_ida` and `Nlsolver` is set to `manual`, you can control the process of solving the linear or nonlinear system of equations in each time step manually. For a coupled problem, this is done through the properties `Damp`, `Dtech`, `Hnlin`, `Initstep`, `Jtech`, `Maxiter`, `Minstep`, and `Rstep` listed under `femnlin`. For a segregated problem, the properties listed under `femstatic` that are related to the segregated solver are available.

The maximum allowed relative error in each time step (the local error) is specified using `Rtol`. However, for small components of the solution vector U, the algorithm tries only to reduce the absolute local error in U below the tolerance given in `Atol`.

The absolute tolerance `Atol` can be given for each degree of freedom separately. The value for the property `Atol` can be:

- A scalar.
- A solution vector.
- A solution object.
- A row cell array with alternating degree of freedom names and definitions. The definitions can be numeric scalars or string expressions. The string expressions may only depend on constants defined in `fem.const` or `Const`. Unspecified degree of freedom names are given the default value 0.

There is no guarantee that the error tolerances are met strictly, that is, for hard problems they can be exceeded.

For the tolerance parameter in the convergence criterion for linear systems, the maximum of the numbers `Rtol` and `Itol` is used.

Use `Complex=on` if complex numbers occur in the solution process.

The property `Consistent` controls the consistent initialization of a *differential algebraic equation* (DAE) system. The value `Consistent=off` means that the initial values are consistent (this is seldom the case, because the initial value of the time derivative is 0). Otherwise, the solver tries to modify the initial values so that they become consistent. The value `Consistent=on` can be used (when `Odesolver=bdf_ida` and `Nlsolver=automatic` or when `Odesolver=bdf_daspk`) for index–1 DAEs. Then the solver fixes the values of the differential DOFs, and solve for the initial values of the algebraic DOFs and the time derivative of the differential DOFs. The value `Consistent=bweuler` can be used for both index-1 and index-2 DAEs. Then the solver perturbs the initial values of all DOFs by taking a backward Euler step.

For a DAE system, if `Estrat=1`, then the algebraic DOFs are excluded from the error norm of the time discretization error.

You can suggest a size of the initial time step using the property `Initialstep`.

By default, the solver determines whether the system is differential-algebraic by looking after zero rows or columns in the mass matrix. If you have a DAE where the mass matrix has no zero rows or columns, put `Masssingular=yes`.

The property `Maxorder` gives the maximum degree of the interpolating polynomial in the BDF method (when `Odesolver=bdf_ida` or `Odesolver=bdf_daspk`).

The property Maxstep puts an upper limit on the time step size (this property is not allowed when Tsteps=manual).

The property Nlsolver controls which nonlinear solver is used to solve the linear or nonlinear system of equations in each time step when Odesolver=bdf_ida. With Nlsolver=automatic, the nonlinear solver which is included in the solver IDA is used. With Nlsolver=manual, the nonlinear solver in COMSOL Multiphysics is used.

When the COMSOL Multiphysics nonlinear solver is used (when Odesolver=bdf_ida and Nlsolver=manual or when Odesolver=genalpha), the property Ntolfact controls how accurately the nonlinear system of equations is solved. The value given in Ntolfact is multiplied with the default tolerance in the convergence criteria. Also, the solution process is interrupted (and the Jacobian updated or the time step reduced) if the convergence is too slow. This can be disabled by setting Useratelimit=off. When Useratelimit=on, what is to be considered as too slow convergence can be controlled through the property Ratelimit. The solution process is interrupted if the estimated linear convergence rate (of all steps, when the segregated solver is used) becomes larger than the value given in Ratelimit.

The property Odesolver is used to select which time-stepping method to use. With Odesolver=bdf_ida, the solver IDA (which uses variable order backward differentiation formula) is used. A similar solver, DASPK, is used if Odesolver=bdf_daspk. With Odesolver=genalpha, the method generalized-α is used. With generalized-α, the numerical damping can be controlled by giving a value, $0 \le \rho_\infty \le 1$, by which the amplitude of the highest possible frequency is multiplied each time step (hence, a small value corresponds to large damping while a value close to 1 corresponds to little damping). This is done through the property Rhoinf. Also, the initial guess for the solution at the next time step (needed by the nonlinear solver) can be controlled through the property Predictor when generalized-α is used. With Predictor=linear, linear extrapolation using the current solution and time-derivative is used. With Predictor=constant, the current solution is used as initial guess.

If the property callback is given, the solver makes interrupts (or callbacks) and calls a function with the given name. To control when the solver makes a callback, use the callblevel property. When callblevel is time, the solver makes callbacks at the time-loop level. If callbfreq is tout, the callback is made at the times that are exported to the output, and when callbfreq is set to tsteps, the callback is made at times taken by the solver. When `callblevel` is `nonlin`, the solver makes a callback after each nonlinear (or segregated) iteration. `Nonlin` is only supported when `odesolver` is `genalpha` or when `odesolver` is `bdf_ida` with `nlsolver` set to `manual`. The `callbfreq` property does not apply when `callblevel` is `nonlin`.

The property `Out` determines the output arguments and their order. The solution object `fem.sol` contains the output times and the corresponding solutions, see `femsol`. By default, the time derivatives are not stored in the solution. To store them, use the `Outcomp` property, see `femsolver`. This will also give a more accurate value in postprocessing of values interpolated in time. The output `u` is a matrix whose columns are the solution vectors for the output times. The output `tlist` is a row vector containing the output times. The output variable `Stop` is 0 if a complete solution was returned, 1 if a partial solution was returned, and 2 if no solution was returned. For the other outputs, see `femlin`.

The property `Reacf` controls the computation and storage of the constraint reaction force. The value `Reacf=on` (default) means that the solver stores the FEM residual vector L in the solution object `fem.sol`. Because $L = N_F\Lambda$ for a converged solution, the residual is the same as the constraint force. Only the components of L that correspond to nonzero rows of N_F are stored. For each time for which the solution is requested an extra residual vector assembly is performed. The value `Reacf=off` gives no computation or storage of the reaction force and can therefore save some computational time.

The property `Stop` makes it possible to return a partial solution when the time stepping fails at some point. If a failure occurs, the computed time steps are returned in `sol`.

Use the property `Stopcond` to make sure the solver stops when a specified condition is fulfilled. When you provide a scalar expression, then the expression is evaluated after each time step. The time stepping is stopped if the real part of the expression is evaluated to something negative. The corresponding solution, for which the expression is negative is not returned. When you provide an integer the solver stops when the corresponding implicit event is triggered.

You can use the property `Keep` to tell `femtime` that certain quantities are constant in time, which sometimes can speed up the computation, see “Manual Control of Reassembly” on page 530. The corresponding value is a string or a cell array of strings.

The property `Tlist` must be a strictly monotone vector of real numbers. Commonly, the vector consists of a start time and a stop time. If more than two numbers are given, the intermediate times can be used as output times, or to control the size of the time-steps (see below). If just a single number is given, it represents the stop time, and the start time is 0.

The property `Tout` determines the times that occur in the output. If `Tout=tsteps`, then the output contains the time steps actually taken by the solver. If `Tout=tlist`, then the output contains interpolated solutions for the times in the `Tlist` property. The default is `Tout=tlist`.

The property `Tsteps` controls the selection of time steps. If `Tsteps=free`, then the solver selects the time steps according to its own logic, disregarding the intermediate times in the `Tlist` vector. If `Tsteps=strict`, then time steps taken by the solver contain the times in `Tlist`. If `Tsteps=intermediate`, then there is at least one time step in each interval of the `Tlist` vector. With `Tsteps=init`, the solver only computes consistent initial values (for the start time, as defined by the property `Tlist`) and then stops. Note that time derivatives of algebraic variables and indicator functions might still be uninitialized after this operation. Such uninitialized quantities will be represented by `NaN` (not a number) in the solution object. If `Tsteps=manual` (only possible when `Odesolver=genalpha`), the solver follows the time step specified in the property `Timestep`. If `Timestep` is a scalar value, this time step is taken in the entire simulation. When `Timestep` is a (strictly monotone) numeric vector, the solver computes the solution at the times in the vector. The start time and stop time is still obtained from `Tlist`; the vector given in `Timestep` is truncated and/or expanded using the first and/or last time step in the vector so that the start time and stop time agrees with the values in `Tlist`. Finally, an expression using variables with global scope and which results in a scalar can be used as `Timestep`.

For problems of wave-type, the logic by which the solver selects the time step can sometimes result in a time step which oscillates in an inefficient manner. When `Odesolver=genalpha` (the solver typically used for problems of wave-type), such oscillations in the time step can be avoided through the property `Incrdelay`. When `Incrdelay=on`, a counter keeps track of the number of consecutive time steps for which a time step increase has been warranted. When this counter exceeds the number given in the property `Incrdelaysteps`, the time step is increased and the counter is set to zero.

The properties Eventout, Eventtol, Maxreinit, and Uifscale only have an effect when used in setups containing events which is only possible with Odesolver=bdf_ida. The property Eventtol determines how accurately implicit event times should be detected. Typically, the value of this property should be similar to the value of rtol. When an event occurs, the property Eventout determines whether both the solution before reinitialization and after reinitialization should be saved. When Tout=tsteps the default value is on, and when Tout=tlist the default value is off. If Tout=tsteps and Eventout=off only the solution prior to reinitialization will be saved. The update at an event and the subsequent reinitialization might trigger new events. The property Maxreinit controls how many times triggered events are allowed to trigger new events at one particular time. Finally, the property Uifscale works just like the property Uscale, but is only applicable to indicator functions. See the entry femsolver for a description of the property Uscale.

For more information about the time-dependent solver; see "The Time-Dependent Solver" on page 391 of the *COMSOL Multiphysics User's Guide*.

Example

Solve the heat equation

$$\frac{\partial u}{\partial t} - \Delta u = 0$$

on a square geometry $-1 \le x, y \le 1$. Choose $u(0) = 1$ on the disk $x^2 + y^2 < 0.4^2$, and $u(0) = 0$ otherwise. Use Dirichlet boundary conditions $u = 0$. Compute the solution at times linspace(0,0.1,20).

```
clear fem
fem.geom = square2(2,'pos',[-1 -1])+circ2(0.4);
fem.mesh = meshinit(fem);
fem.shape = 2;
fem.equ.c = 1; fem.equ.da = 1;
fem.bnd.h = 1;
fem.equ.init = {0 1};
fem.xmesh = meshextend(fem);
fem.sol = femtime(fem,'report','on','tlist',linspace(0,0.1,20));
postanim(fem,'u')
```

Cautionary

In structural mechanics models, the displacements are often quite small, and it is critical that the Atol property is chosen to be smaller than the actual displacements.

Compatibility

The property Variables has been renamed Const in FEMLAB 2.3.

The properties `Epoint` and `Tpoint` are obsolete from FEMLAB 2.2. Use `fem.***.gporder` to specify integration order.

See Also `femsolver`, `assemble`, `asseminit`, `femstruct`, `femlin`, `femnlin`

Purpose

Extend FEM structure to a wave equation problem.

Syntax

```
fem1 = femwave(fem)
xfem1 = femwave(xfem)
```

Description

`fem1 = femwave(fem)` extends the coefficients of the PDE problem to a wave equation problem. `xfem` can also be an extended FEM structure. In the latter case, use `geomnum` to specify the geometry number for the wave extension.

Note: Since COMSOL Multiphysics 3.2, wave equations can be more easily and efficiently formulated using the e_a coefficient or the second time derivative variable.

When `fem` is given in coefficient form, `fem1` contains extended PDE and boundary coefficients to solve the wave equation problem

$$d_a\frac{\partial^2 u}{\partial t^2} - \nabla\cdot(c\nabla u + \alpha u - \gamma) + \beta\nabla u + au = f$$

where the d_a coefficient was stored in the `fem.equ.da` field of the FEM structure `fem`. In the same way, when `fem` is given in general or weak form, `fem1` contains extended PDE and boundary coefficients to solve the wave equation problem obtained by replacing the first time derivative term in the standard problem by a second time derivative.

The function introduces a set of new variables, v, such that

$$v = \frac{\partial u}{\partial t}$$

and then transforms the PDE problem by doubling the size of the system, and rewriting the coefficients according to the table below:

TABLE 1-47: TRANSFORMATION DONE BY FEMWAVE

PDE COEFFICIENT	COEFFICIENT FORM	GENERAL FORM
c	$\begin{bmatrix} 0 & 0 \\ c & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 \\ c & 0 \end{bmatrix}$
α	$\begin{bmatrix} 0 & 0 \\ \alpha & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 \\ \alpha & 0 \end{bmatrix}$
γ	$\begin{bmatrix} 0 \\ \gamma \end{bmatrix}$	$\begin{bmatrix} 0 \\ \gamma \end{bmatrix}$
a	$\begin{bmatrix} 0 & -I \\ a & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & -I \\ a & 0 \end{bmatrix}$
f	$\begin{bmatrix} 0 \\ f \end{bmatrix}$	$\begin{bmatrix} v \\ f \end{bmatrix}$
d_a	$\begin{bmatrix} I & 0 \\ 0 & d_a \end{bmatrix}$	$\begin{bmatrix} I & 0 \\ 0 & d_a \end{bmatrix}$
e_a	$\begin{bmatrix} e_a & 0 \\ 0 & 0 \end{bmatrix}$	$\begin{bmatrix} e_a & 0 \\ 0 & 0 \end{bmatrix}$
q	$\begin{bmatrix} 0 & 0 \\ q & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 \\ q & 0 \end{bmatrix}$
g	$\begin{bmatrix} 0 \\ g \end{bmatrix}$	$\begin{bmatrix} 0 \\ g \end{bmatrix}$

TABLE 1-47: TRANSFORMATION DONE BY FEMWAVE

PDE COEFFICIENT	COEFFICIENT FORM	GENERAL FORM
h	$\begin{bmatrix} \frac{\partial h}{\partial t} & h \\ h & 0 \end{bmatrix}$	$\begin{bmatrix} \frac{\partial h}{\partial t} & h \\ h & 0 \end{bmatrix}$
r	$\begin{bmatrix} \frac{\partial r}{\partial t} \\ r \end{bmatrix}$	$\begin{bmatrix} \frac{\partial r}{\partial t} - hv \\ r \end{bmatrix}$

When the property `Tdiff` is `off`, the following modifications to the table applies:

TABLE 1-48: TRANSFORMATION DONE BY FEMWAVE

h	$\begin{bmatrix} 0 & 0 \\ h & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 \\ h & 0 \end{bmatrix}$
r	$\begin{bmatrix} 0 \\ r \end{bmatrix}$	$\begin{bmatrix} 0 \\ r \end{bmatrix}$

For a PDE problem in general form, `femwave` produces h using `femdiff`, when the `Diff` property is `on` and the field `h` does not exist.

If the coefficients `weak`, `dweak`, and `constr` are present, either in addition to the above coefficients, or on their own because the problem is in weak form, they also become doubled in size, but their treatment is very special because they may contain explicit references to the n dependent variables u, their time derivatives u`_time` and the test functions u`_test`, ux`_test`, etc.

`Weak` coefficients are cell arrays of length n. They become cell arrays of length $2n$, by moving the existing entries down to the second half, then replacing all references therein to u`_test`, ux`_test`, etc. with references to v`_test`, vx`_test`, etc. The first half of the vector contains n entries of the form v_i*u_i`_test`.

`Dweak` coefficients are cell arrays of length n. They become cell arrays of length $2n$, by moving the existing entries down to the second half, then replacing all references therein to u`_time`, ux`_time`, u`_test`, ux`_test`, etc. with references to v`_time`, vx`_time`, v`_test`, vx`_test`, etc. The first half of the vector contains n entries of the form u_i`_time`*u_i`_test`.

`Constr` coefficients are cell arrays of length n. They become cell arrays of length $2n$, by moving the existing entries down to the second half. If `Tdiff` is `on`, then the top half of the vector is filled by the following entries. If one of the coefficients is c_i, one of the new entries in the top half of the new vector is

$$\frac{\partial c_i}{\partial t} + v_j \frac{\partial c_i}{\partial u_j} + (v_j)_{x_k} \frac{\partial c_i}{\partial (u_j)_{x_k}}$$

where $(v_j)_{x_k}$ represents the partial derivative of v_j with respect to space coordinate x_k, and where there is implicit summation over repeated indices.

The function `femwave` accepts the following property/value pairs:

TABLE 1-49: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Defaults`	`off` | `on`	`off`	Return default fields
`Diff`	cell array with strings that may contain the strings `r`, `var`, and/or `expr` or the strings `off` or `on`	`on` (for general form)	Differentiate constraints. Describes which fields that are differentiated if `Tdiff` is `on`. See also `femdiff`.
`Geomnum`	integer	`1`	Geometry number
`Shrink`	`off` | `on`	`off`	Shrinks coefficients to most compact form.
`Simplify`	`off` | `on`	`on`	Simplify differentiated expressions
`Tdiff`	`off` | `on`	`on`	Differentiate constraints with respect to time

Use `fem.rules` to specify additional differentiation rules. The derivative of the Inverse hyperbolic tangent function `atanh` can, for example, be specified as `{'atanh','1./(1-x.^2)'}`. It can also be stored as a field in the `fem` structure.

Cautionary

The properties `bdl`, `out`, `rules`, and `sdl` are obsolete in FEMLAB 3.0.

The h and r coefficients at level 4 of the syntax must be given either as a scalar numeric value, or a string containing an expression.

You should set the ODE Suite parameter `maxorder` to 2 for the solver `ode15s` for wave type problems. This is automatically done by the graphical user interface.

Compatibility

In FEMLAB 1.0, when using general form, you had to apply `femdiff` before `femwave`. This was because the h coefficient in `fem` affects the result of the r coefficient in the output `fem1`, and h had to be computed by symbolic differentiation by `femdiff`. In FEMLAB 1.1, h is automatically computed by `femwave` if not provided. Therefore `femdiff` can be applied after the `femwave` call in FEMLAB 1.1.

See Also `femtime`

Purpose Create circular rounded corners in geometry object.

Syntax `g = fillet(g1,...)`

Description `g = fillet(g1,...)` creates rounded corners in 2D geometry object.

The function `fillet` accepts the following property/values:

TABLE 1-50: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`out`	Cell array of strings	`none`	Determines the output
`point`	integers | `all` | `none`	`all`	Specifies which vertices are filleted
`radii`	1-by-m vector		Curvature radii of the fillet

The corners to fillet is either specified with either the property `point` or `edges`. The default value is the all possible corners are filleted.

If there is only one radius but more than one corner then the single radius is used for all corners.

Examples Fillet a rectangle object:

```
r = rect2;
s1 = fillet(r,'radii',0.1);
s2 = fillet(r,'edges',[1 2;2 3],'radii',0.2);
```

Diagnostics If `fillet` does not succeed in creating a rounded corner according to the specified radius, the corner is skipped.

When a fillet intersects another edge, the function generates an error message.

Compatibility The FEMLAB 2.3 property `Trim` is no longer supported. Only pair of edges that have a common vertex can be filleted. For edges that are not linear, the linear approximation of the edge in the corner is used to compute an approximate fillet.

See Also `chamfer`, `curve2, curve3`

Purpose Compact `equ/bnd/edg/pnt` fields.

Syntax

```
fem = flcompact(fem)
```

Description `fem = flcompact(fem)` removes unused and duplicated coefficients in the `fem.equ`, `fem.bnd`, `fem.edg`, and `fem.pnt` fields. The resulting structures always have numeric `ind` fields. Coefficients are considered equal if they represent the same expression, that is, equivalent short-hand and expanded forms are compacted.

The function `flcompact` accepts the following property/value pairs:

TABLE 1-51: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`defaults`	`off` | `on`	`off`	Return default fields
`shrink`	`off` | `on`	`off`	Shrinks coefficients to most compact form.

Compatibility The syntaxes

```
equ = flcompact(equ,'equ',nsd)
bnd = flcompact(bnd,'bnd',nbnd)
edg = flcompact(edg,'edg',nedg)
pnt = flcompact(pnt,'pnt',npnt)
field = flcompact(field,fldnames,nelem)
```

are no longer supported in FEMLAB 3.1.

See Also `multiphysics`

Purpose

Create boundary mesh from contour data.

Syntax

```
m = flcontour2mesh(c)
```

Description

`m = flcontour2mesh(c)` creates a boundary mesh `m` with fields `m.p` and `m.e` from the contour data `c`. The contour matrix `c` is a two-row matrix of contour lines. Each contiguous drawing segment contains the value of the contour, the number of (x,y) drawing pairs, and the pairs themselves. The segments are appended end-to-end as

```
c = [level1 x1 x2 x3 ... level2 x2 x2 x3 ...;
     pairs1 y1 y2 y3 ... pairs2 y2 y2 y3 ...]
```

The contour matrix format is used by the MATLAB function `contourc`.

By using the contour matrix format, you can convert geometry data defined by a point set, to a COMSOL Multiphysics geometry object. Firstly, define a contour matrix `c` corresponding to your point set and use `flcontour2mesh` to convert the contour matrix `c` to a 2D boundary mesh `m`. Then, use `flmesh2spline` to convert the mesh object `m` to a `curve2` object.

Examples

Create a mesh from contour data.

```
[x,y] = meshgrid(linspace(-3,3,50));
z = (x.^2+y.^2).*exp(-x.^2-y.^2)+cos(y)+sin(x);
figure
c = contour(z);
m = flcontour2mesh(c);
figure
meshplot(m);
```

See Also

`contourc`, `flmesh2spline`, `flim2curve`

Purpose

Smoothed step functions.

Syntax

```
y = flc1hs(x,scale)
y = flc2hs(x,scale)
y = fldc1hs(x,scale)
y = fldc2hs(x,scale)
```

Description

`y = flc1hs(x,scale)` and `y = flc2hs(x,scale)` compute the values of a smoothed version of the Heaviside function `y = (x>0)`. The function is `0` for `x<-scale`, and `1` for `x>scale`.

In the interval `-scale<x<scale`, `flc1hs` is a smoothed Heaviside function with a continuous first derivative without overshoot. It is defined by a fifth-degree polynomial.

In the interval `-scale<x<scale`, `flc2hs` is a smoothed Heaviside function with a continuous second derivative without overshoot. It is defined by a a sixth-degree polynomial.

The input `x` can be an array. The input `scale` must be positive scalar.

`yp = fldc1hs(x,scale)` and `yp = fldc2hs(x,scale)` compute the derivative of the functions `flc1hs` and `flc2hs`, respectively.

See Also

`flsmhs, flsmsign, fldsmhs, fldsmsign`

Purpose Convert between PDE forms.

Syntax

```
fem1 = flform(fem,'outform',form,...)
[equ,bnd] = flform(fem,'outform',form,...)
```

Description `fem1 = flform(fem,'outform',form,...)` converts the FEM structure `fem` to an FEM structure `fem1` on form. The fields in `fem1.equ` and `fem1.bnd` contain the corresponding fields from `fem.equ` and `fem.bnd` converted to the form `form`. `fem1.form` is set to `form`. All other fields in `fem` are copied to `fem1`.

```
[equ,bnd] = flform(fem,'outform',form,...)
```

is an alternative syntax, returning only the `equ` and `bnd` fields of the FEM structure.

Conversion from coefficient to general form is performed according to

$$\Gamma_{lj} = -c_{lkji}\frac{\partial u_k}{\partial x_i} - \alpha_{lkj}u_k + \gamma_{lj}$$

$$F_l = f_l - \beta_{lki}\frac{\partial u_k}{\partial x_i} - a_{lk}u_k$$

$$G_l = g_l - q_{lk}u_k$$

$$R_m = r_m - h_{ml}u_l$$

using a notation where there is an implicit summation over the k (or l) and i indices in each product. Affected fields are therefore `ga`, `c`, `al`, `f`, `be`, and `a` from `equ` and `g`, `q`, `r`, and `h` from `bnd`, with `c`, `al`, `be`, `a`, `g`, and `q` removed and `ga`, `f`, `r`, and `g` remaining. Other fields within `equ` and `bnd`, such as `shape`, `weak`, `init`, `var`, etc., remain unchanged.

Conversion from general form to weak form is performed according to

$$W_l^{(n)} = W_l^{(n)} + \Gamma_{lj}\frac{\partial v_l}{\partial x_j} + F_l v_l$$

$$W_l^{(nt)} = W_l^{(nt)} + d_{alk}\frac{\partial u_k}{\partial t}v_l + e_{alk}\frac{d^2 u_k}{dt^2}v_l$$

$$W_l^{(n-1)} = W_l^{(n-1)} + G_l v_l$$

$$R_m^{(n)} = R_m$$

where there is an implicit summation over the k and i indices in each product. n is the space dimension. Affected fields are therefore `ga`, `f`, `weak`, `da`, `ea`, and `dweak` from `equ` and `g`, `weak`, `r`, and `constr` from `bnd`, with `weak`, `dweak`, and `constr` the only fields remaining. Other fields within `equ` and `bnd`, such as `shape`, `init`, `var`, etc., remain unchanged.

In addition, when converting to `weak` form, `flform` tries to take `fem.border` into account. That is to say that if `fem.border` is not `1` or `on`, there may be interior boundaries on which boundary conditions should not be applied. This process is carried out because `meshextend` and the solvers pay no attention to `fem.border` when considering `weak`, `dweak`, and `constr`, unlike `ga`, `c`, `f`, `q`, `r`, `h`, etc.

The function `flform` accepts the following property/value pairs:

TABLE 1-52: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`defaults`	`off` | `on`	`off`	Return default fields
`outform`	`coefficient` | `general` | `weak`	`coefficient`	Output form
`out`	`fem` | `equ` | `bnd` | `edg` | `pnt`	`fem`	Output variables
`shrink`	`off` | `on`	`off`	Shrinks coefficients to most compact form.
`simplify`	`off` | `on`	`on`	Simplify expressions

Cautionary

Conversion from general form to coefficient form, or from weak form to general or coefficient form is not supported.

Example

The following code shows how the convergence can be improved for a stationary solution of the model “Resistive Heating”. The system is converted to general form, the symbolic derivatives are computed using `femdiff`, and the system is solved with `femstatic`.

```
% !!! First run the example under the multiphysics entry
fem = flform(fem,'outform','general');
fem = femdiff(fem);
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem,'report','on');
postsurf(fem,'T');
```

Alternatively, change the `outform` to `weak` and remove the `femdiff` call.

See Also

`multiphysics`, `meshextend`

Purpose Create 2D curve object from image data.

Syntax `[c,r] = flim2curve(I,fmt,...)`

Description `[c,r] = flim2curve(I,fmt,...)` creates a `curve2` object `c` and small `curve2` objects `r` from the image `I` (gray-scale or RGB) operated upon by parameters contained in the cell-array `fmt`. `c` is a `curve2` object that approximates the contours of `I`. `r` is a cell-array of `curve2` objects containing small curves detected by the argument. This is very useful for images containing noise. `I` is either an m-by-n intensity image matrix or an m-by-n-by-3 RGB-image matrix as typically obtained from the function `imread`. `fmt` is a cell-array of length 2 used to create contours from `I`. If `fmt{2}` is empty then the scalar `fmt{1}` is used as a threshold value. If instead, `fmt{1}` is empty, then the vector `fmt{2}` is used to specify the contour levels of interest. See the function `contourc` for an explanation of the contour level syntax in `fmt{2}`. All `flmesh2spline` properties are supported.

`c = flim2curve(i,fmt,...)` is an alternative syntax and is equivalent to `c = geomcsg({},{c,r{:}})` where the arguments `c` and `r` are those obtained from the other call. This is less stable whenever `i` contain small structures.

Examples Create contour-curves from function.

```
[x,y] = meshgrid(linspace(-3,3,50));
z = (x.^2+y.^2).*exp(-x.^2-y.^2)+cos(y)+sin(x);
figure
imagesc(z)
g = flim2curve(z,{[],[-1.5:0.5:2]});
figure
geomplot(g,'Pointmode','off');
```

Create curves from noisy picture.

```
load mri
pic = D(:,:,1,10);
figure
image(pic)
v = axis;
[c,r] = flim2curve(pic,{[],1:30:91},'KeepFrac',0.10);
figure
geomplot(c,'Pointmode','off');
```

Plot all small curves in a green color.

```
for j = 1:length(r)
  hold on
  geomplot(r{j},'Pointmode','off','edgecolor','g')
end
```

```
axis(v)
axis ij
```

See Also `flcontour2mesh`, `flmesh2spline`

Purpose Load a COMSOL Multiphysics file.

Syntax

```
flload filename
fem = flload('filename')
```

Description `flload(filename)` retrieves FEM structures, geometry objects, or mesh objects from a COMSOL Multiphysics file. If `filename` has no extension, it is assumed to be a Model MPH-file.

`flload` supports Model MPH-files (`.mph`) for retrieving complete FEM structures and COMSOL Multiphysics text and binary files (`.mphtxt`, `.mphbin`) for retrieving geometry and mesh objects.

See Also `flsave`

Purpose

Create spline curves from mesh.

Syntax

`[g2,r2] = flmesh2spline(msh,...)`

Description

`[g2,r2] = flmesh2spline(msh,...)` creates spline curves `g2` and filtered small curves `r2`. The structure `msh` is a valid mesh, where only the fields `msh.p` and `msh.e` are needed. The object `g2` is a `curve2` object containing spline curves approximating the edge of `msh`. The variable `r2` is a cell-array containing small curves filtered away by the algorithm. This is a useful feature when trying to generate curves from meshes that originate from noisy contour data.

`g2 = flmesh2spline(msh,...)` is an alternative syntax and is equivalent to `g2 = geomcsg({},{g2,r2{:}})` where the arguments `g2` and `r2` are those obtained from the other call. This is less stable whenever `msh` contain small (ill-conditioned) structures.

TABLE 1-53: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`KeepFrac`	real scalar >0, <=1	`0.2`	Fraction of points to keep
`Smooth`	`on` | `off`	`on`	Curve smoothing on or off
`SplineMethod`	`uniform` | `chordlength` | `centripetal` | `foley`	`chordlength`	Method for spline parameterization

The property `KeepFrac` provides a useful way to reduce the complexity of the resulting geometry object. If the algorithm fails to produce the desired result, try to lower the value of this property. The smoothing algorithm used is a simple anti-aliasing filter, and is controlled by the property `Smooth`. For an explanation of the property `SplineMethod`, see `geomspline`.

Note: You might need to refine the boundary mesh data using `meshrefine` to be able to get reasonable results using this function. Alternatively, you can create a finer mesh using `meshinit` by manipulating the mesh parameters.

Examples

Create spline curves from a full mesh:

```
msh = meshinit(circ2+rect2(1,1,'pos',[0.5 0.5]));
```

```
figure
meshplot(msh)
c = flmesh2spline(msh,'keepfrac',0.3)
figure
geomplot(c)
```

Create spline curves from contour data:

```
[x,y] = meshgrid(linspace(-3,3,50));
z = (x.^2+y.^2).*exp(-x.^2-y.^2)+cos(y)+sin(x);
figure
c = contour(z);
m = flcontour2mesh(c);
figure
meshplot(m)
g = flmesh2spline(m);
figure
geomplot(g)
```

See Also `flcontour2mesh`, `flim2curve`, `geomspline`

flngdof

Purpose Get number of global degrees of freedom.

Syntax

```
n = flngdof(fem)
n = flngdof(fem, mcase)
```

Description The returned number `n` is the number of degrees of freedom in the FEM structure `fem`. This is the same as the length of the solution vector. If the mesh case `mcase` is not given, it is taken to be the mesh case with the greatest number of degrees of freedom in the extended mesh.

See Also `meshextend`

Purpose Compute null space of a matrix, its complement, and the range of the matrix.

Syntax

```
Range = flnull(N,...)
[Null,Compl] = flnull(N,...)
[Null,Compl,Range] = flnull(N,...)
[...] = flnull('in',{...},'out',{...},...)
```

Description `Range = flnull(N,...)` computes the range of N.

`[Null,Compl] = flnull(N,...)` computes the null space of N and its complement.

`[Null,Compl,Range] = flnull(N,...)` computes null space, its complement, and the range of N.

`[...] = flnull('in',{...},'out',{...},...)` compute null space, its complement, and the range of N.

The function `flnull` accepts the following property/value pairs:

TABLE 1-54: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`In`	`N` | `NT`	`N`	Input matrices
`Nullfun`	`flnullorth` | `flspnull` | `auto`	`auto`	Null-space function
`Out`	`null` | `range` | `compl`		Output variables

The property `Nullfun` selects the null-space algorithm. The algorithm `flnullorth` computes a orthonormal basis for the null space by using singular value decomposition in a block-wise pattern. The method `flspnull` handles constraint matrices with non-local couplings by employing a sparse algorithm. The `auto` method automatically selects the most appropriate of `flnullorth` and `flspnull`.

See the sections “Advanced Solver Settings” on page 526 and “Constraint Handling” on page 533 for further information on the use of these matrices.

Example *The Poisson Equation on the Unit Disk*

Solve this problem by elimination. The example illustrates the way `femstatic` handles the constraints internally by the default constraint handling method: `eliminate`.

```
clear fem
fem.geom = circ2;
fem.mesh = meshinit(fem);
fem.shape = 2;
fem.equ.c = 1; fem.equ.f = 1;
fem.bnd.h = 1;
fem.xmesh = meshextend(fem);
[K,L,M,N] = assemble(fem);
[Null,Compl,Range] = flnull(N);
ud = Compl*((Range'*N*Compl)\(Range'*M));
Ke = Null'*K*Null;
Le = Null'*(L-K*ud);
vn = Ke\Le;
u = Null*vn+ud;
fem.sol = femsol(u);
postplot(fem,'tridata','u')
```

See Also `femlin`, `assemble`

Purpose Globally turn off the report progress window or show it.

Syntax

```
flreport('off')
flreport('on')
flreport('show')
flreport('file')
flreport('file',filename)
```

Description `flreport('off')` disables the use of the progress window that is normally shown during meshing and solution.

`flreport('on')` enables the use of the progress window again. This means that COMSOL Multiphysics uses the value of the `'report'` property to determine if the progress window should be shown.

`flreport('show')` shows the progress window if it has been closed.

`flreport('file')` prints progress information to file. If you have not provided a filename in an earlier call to `flreport`, it prints progress information to the standard output (stdout).

`flreport('file',filename)` prints progress information to the file `filename`. If no filename is given, it prints progress information to the standard output (stdout).

Purpose Save a COMSOL Multiphysics file.

Syntax `flsave filename fem`

Description `flsave filename arg1 arg2 ...` saves FEM structures, geometry objects, and mesh objects to a COMSOL Multiphysics file. `flsave` supports Model MPH-file (`.mph`) and the COMSOL Multiphysics text and binary formats (`.mphtxt`, `.mphbin`).

`flsave filename` saves all valid COMSOL data in the workspace.

`flsave('filename',arg1,...)` is an alternative syntax.

Compatibility Since FEMLAB 3.0, `flsave` is obsolete for saving a MAT-file. Use `save` to save an FEM structure or any part of the FEM structure.

See Also `flload`

Purpose

Smoothed step functions and their derivatives.

Syntax

```
y = flsmhs(x,scale)
y = flsmsign(x,scale)
yp = fldsmhs(x,scale)
yp = fldsmsign(x,scale)
```

Description

`y = flsmhs(x,scale)` computes the values of a smoothed version of the Heaviside function `y = (x>0)`. The function is `0` for `x<-scale`, and `1` for `x>scale`.

`y = flsmsign(x,scale)` computes the values of a smoothed version of the sign function `y = sign(x)`. The function is `-1` for `x<-scale`, and `1` for `x>scale`.

In the interval `-scale<x<scale`, the functions `flsmhs` and `flsmsign` are defined by a seventh-degree polynomial, which is chosen so that the second derivative is continuous. Moreover, the moments of order 0, 1, and 2 agree with those for the Heaviside function and the sign function, respectively. This implies that the functions have small overshoots.

`yp = fldsmhs(x,scale)` and `yp = fldsmsign(x,scale)` compute the derivative of the functions `flsmhs` and `flsmsign`, respectively.

The input `x` can be an array. The input `scale` must be positive scalar.

See Also

`flc1hs, flc2hs, fldc1hs, fldc2hs`

Purpose

Create straight homogeneous generalized cylinder geometry object.

Syntax

```
s3 = gencyl3
s2 = gencyl2
s3 = gencyl3(base)
s2 = gencyl2(base)
s3 = gencyl3(base,h)
s2 = gencyl2(base,h)
s3 = gencyl3(base,h,rat)
s2 = gencyl2(base,h,rat)
s3 = gencyl3(base,h,rat,...)
s2 = gencyl2(base,h,rat,...)
```

Description

`s3 = gencyl3` creates a solid straight homogeneous generalized cylinder geometry object `s3`, with a solid circle base surface, cylinder axis of length 1 along the z-axis, and size of top surface equal to base surface. `gencyl3` is a subclass of `solid3`.

`s3 = gencyl3(base)` creates a solid straight homogeneous generalized cylinder geometry object with base surface `base`.

`s3 = gencyl3(base,h)` also sets the height of the generalized cylinder to `h`.

`s3 = gencyl3(base,h,rat)` additionally specifies top surface with the scale factor `rat` with respect to the origin, that is, all 2D points in the top plane are obtained by multiplying the points in the base plane with `rat`.

The functions `gencyl3` and `gencyl2` accept the following property/values:

TABLE 1-55: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Axis`	Vector of reals or cell array of strings	`[0 0]`	Local z-axis of the object.
`Const`	Cell array of strings	`{}`	Evaluation context for string inputs.
`Displ`	2-by-nd matrix	`[0;0]`	Displacement of extrusion top
`Pos`	Vector of reals or cell array of strings	`[0 0]`	Position of the bottom surface.
`Rot`	real or string	0	Rotational angle about `Axis` (radians).

s2 = gencyl2(...) creates a surface straight homogeneous generalized cylinder, from the same arguments as described for gencyl3. gencyl2 is a subclass of face3.

Generalized cylinder objects have the following properties:

TABLE 1-56: GENERALIZED CYLINDER OBJECT PROPERTIES

PROPERTY	DESCRIPTION
base	Base 2D geometry object
h	Height
rat	Ratio
dx, dy	Semi-axes
x, y, z, xyz	Object position—components and vector form
ax2	Rotational angle of symmetry axis
ax3	Axis of symmetry
rot	Rotational angle

In addition, all 3D geometry object properties are available. All properties can be accessed using the syntax get(object,property). See geom3 for details.

Compatibility

The FEMLAB 2.3 syntax is obsolete but still supported. The numbering of faces, edges and vertices is different from the numbering in objects created in 2.3.

Examples

Creation of a 3D solid with two circular edges, and with a top face that is smaller than the bottom face.

```
base = solid2(geomdel(rect2(2,6,'pos',[-1 -3])+...
                   circ2(1,'pos',[0 -3])+circ2(1,'pos',[0 3])));
g3 = gencyl3(base,2,0.75);
geomplot(g3)
```

See Also

econe2, econe3, extrude, face3

Purpose

Low-level constructor functions for geometry objects.

Syntax

```
g = geom3(vertex,pvertex,edge,pedge,face,mfd,pcurve,...)
[g,...] = geom3(g3,...)
g = geom2(vertex,edge,curve)
[g,...] = geom2(g2,...)
g = geom1(p,ud)
[g,...] = geom1(g1,...)
g = geom0(p)
```

Description

`g3 = geom3(vertex,pvertex,edge,pedge,face,mfd,pcurve,...)` creates a `geom3` object.

`[g,...] = geom3(g3,...)` coerces any 3D geometry object `g3` to a `geom3` object.

`c = geom2(vertex,edge,curve,...)` creates a `geom2` object.

`[g,...] = geom2(g2,...)` coerces any 2D geometry object to a `geom2` object.

`c = geom1(vtx)` creates a 1D geometry object from the property `vtx`.

`[g,...] = geom1(g1,...)` coerces any 1D geometry object to a `geom1` object.

`g = geom0(p)` creates a 0D geometry object, where `p` is a matrix of size 0-by-1.

`g = geom0(g1,...)` coerces any 0D geometry object to a `geom0` object.

The coercion functions accept the following property/values:

TABLE 1-57: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Out`	`stx` | `ftx` | `ctx` | `ptx`	`{}`	Cell array of output names

3D Geometry Object Properties

`vertex` is a 5-by-nv matrix representing the vertices of the 3D geometry. Rows 1, 2, and 3 provide the 3D coordinates of the vertices. Row 4 provides the subdomain number. Row 5 contains a relative local tolerance for the entity. For nontolerant entities the tolerance is NaN.

`pvertex` is a 6-by-`npv` matrix containing embeddings of vertices in faces. Row 1 contains the vertex index (i.e. column in VERTEX), rows 2 and 3 contain (s, t) coordinates of the vertex on the face, row 4 contains a face index, and row 5 contains the manifold index into `mfd`. Row 6 contains a relative local tolerance for the entity.

`edge` is a 7-by-`ne` matrix representing the edges of the 3D geometry. Rows 1 and 2 contain the start and end vertex indices of the edge (0 if they do not exist). Rows 3 and 4 give the parameter values of the these vertices. Row 5 gives the index of a subdomain if the edge is not adjacent to a face. Row 6 gives a sign and an index to the underlying manifold. The sign indicates the direction of the edge relative the curve. Finally, row 7 contains a relative local tolerance for the entity.

`pedge` is a 10-by-`npe` matrix representing the embeddings of the edges in faces. The first row gives the index of the edge in `edge`. Rows 2 and 3 contain the start and end vertex indices in `pvertex`. Rows 4 and 5 give the parameter values of the these vertices. Row 6 and 7 give the indices of the faces to the left and right of the edge, respectively. Row 8 gives a sign and index to the parameter curve (if any), and row 9 gives the index to the surface. Row 10 contains a relative local tolerance for the entity.

`face` is a 4-by-`nf` matrix representing the faces of the 3D geometry. Rows 1 and 2 contain the up and down subdomain index of the face, and row 3 contains the manifold index of the face. Row 4 contains a relative local tolerance for the entity.

`mfd` is a cell array or Java array of 3D manifolds.

`pcurve` is a cell array or Java array of parameter curves.

TABLE 1-58: JAVA 3D MANIFOLD CLASSES

MANIFOLD	USAGE	DESCRIPTION
`MfdBezierCurve`	`(xyzw)`	Rational Bezier curve
`MfdBezierTri`	`(xyzw)`	Rational Bezier triangular surface
`MfdBezierSurf`	`(xyzw)`	Rational Bezier tensor-product surface
`MfdBSplineCurve`	`(deg,knots,P,w)`	B-spline curve
`MfdBSplineSurf`	`(uDeg,vDeg,uKnots, vKnots,P,w)`	B-spline surface
`MfdMeshCurve`	`(coord,par)`	Mesh curve
`MfdMeshSurface`	`(coord,par,tri)`	Mesh surface
`MfdPolChain`	`(pol)`	Polygon chain manifold

All properties can be accessed using the syntax `get(object,property)`.

2D Geometry Object Properties

`vertex` is a 4-by-nv matrix representing the vertices of the 3D geometry. Rows 1 and 2 provide the 2D coordinates of the vertices. Row 3 provides the subdomain number. Row 4 contains a relative local tolerance for the entity. For non-tolerant entities the tolerance is NaN.

`edge` is a 8-by-ne matrix representing the edges of the 3D geometry. Rows 1 and 2 contain the start and end vertex indices of the edge, 0 if they do not exists. Rows 3 and 4 give the parameter values of the these vertices. 5 and 6 contain the left and right subdomain number of the edge. Row 7 gives a sign and an index to the array of underlying curves. The sign indicates the direction of the edge relative the curve. Row 8 contains a relative local tolerance for the entity.

`curve` is a cell array or Java array of 2D curves.

TABLE 1-59: JAVA 2D MANIFOLD CLASSES

MANIFOLD	ARGUMENTS	DESCRIPTION
`MfdBezierCurve`	`(xyzw)`	Rational Bézier curve manifold
`MfdBSplineCurve`	`(deg,knots,P,w)`	B-spline curve
`MfdFileCurve`	`(name,ind,s1,s2)`	Geometry M-file manifold
`MfdMeshCurve`	`(coord,par)`	Mesh curve
`MfdPolChain`	`(pol)`	Polygon chain manifold

All properties can be accessed using the syntax `get(object,property)`.

1D Geometry Object Properties

`vtx` is a 3-by-`nvtx` matrix representing the vertices of the 2D geometry. Row 1 provides the 1D coordinates of the vertices. Rows 2 and 3 provides the up and down subdomain.

All properties can be accessed using the syntax `get(object,property)`.

0D Geometry Object Properties

A 0D geometry object `g` has the property `p`, a 0-by-`ns` double of empty coordinates. `ns` can be either 1 or 0, for a nonempty and empty object, respectively.

All properties can be accessed using the syntax `get(object,property)`.

Compatibility

The FEMLAB 3.0 syntax is obsolete but still supported.

See Also

`geom0/get`, `geom1/get`, `geom2/get`, `geom3/get`, `geomobject`, `geomedit`, `geominfo`, `point1`, `point2`, `point3`, `curve2`, `curve3`, `face3`

Purpose

Get geometry object properties.

Syntax

```
get(g,prop)
```

Description

`get(g,prop)` returns the value of the property `prop` for a geometry object `g`, which can be a geometry object of type `geom1` (1D geometry object), `geom2` (2D geometry object), or `geom3` (3D geometry object).

`prop` is a string that contains a valid property name. The following tables list the valid property names for `geom1`, `geom2`, and `geom3` objects:

TABLE 1-60: GEOM1 PROPERTY NAMES

PROPERTY NAME	DESCRIPTION
`vtx`	Vertices
`nv`	Number of vertices
`ns`	Number of subdomains
`mp`	Vertex coordinates
`sd`	Vertex subdomain numbers

For information about the formats for the `vtx` property, see “1D Geometry Object Properties” on page 228.

TABLE 1-61: GEOM2 PROPERTY NAMES

PROPERTY NAME	DESCRIPTION
`vertex`	Vertices
`edge`	Edges
`curve`	2D manifolds
`nv`	Number of vertices
`ne`	Number of edges
`ns`	Number of subdomains
`mp`	Vertex coordinates
`sd`	Vertex subdomain numbers

For information about the formats for the properties `vertex`, `edge`, and `curve`, see “2D Geometry Object Properties” on page 227.

TABLE 1-62: GEOM3 PROPERTY NAMES

PROPERTY NAME	DESCRIPTION
`vertex`	Vertices
`pvertex`	Parameter vertices

TABLE 1-62: GEOM3 PROPERTY NAMES

PROPERTY NAME	DESCRIPTION
`edge`	Edges
`pedge`	Parameter edges
`face`	Faces
`mfd`	3D manifolds
`pcurve`	Parameter curves
`nv`	Number of vertices
`ne`	Number of edges
`nf`	Number of faces
`ns`	Number of subvolumes
`mp`	Vertex coordinates
`sd`	Vertex subdomain numbers

For information about the formats for the properties `vertex`, `pvertex`, `edge`, `pedge`, `face`, `mfd`, and `pcurve`, see “3D Geometry Object Properties” on page 226.

You can also use `get` to retrieve specific properties for primitives such as `circ2`, `rect2`, `block3`, `cylinder3`, `ellipsoid3`, and `sphere3`. For example, `get(sph,'r')` returns the radius of the sphere `sph`. In this case, type

```
help sphere3/get
```

to get a list of available property names.

For `geom0` objects, the property `ns` returns either 1 or 0, for a nonempty and empty object, respectively (see “0D Geometry Object Properties” on page 228).

Example

Create a cylinder object and return the number of subvolumes (1), the number of vertices (8), and the number of faces (6):

```
cyl = cylinder3;
ns = get(cyl,'ns')
nv = get(cyl,'nv')
nf = get(cyl,'nf')
```

See Also

`geom0, geom1, geom2, geom3, geominfo`

Purpose Decompose and analyze geometry of FEM problem.

Syntax

```
fem = geomanalyze(fem, ...)
fem = geomanalyze(fem, draw, ...)
[fem, map] = geomanalyze(fem, ...)
```

Description `fem = geomanalyze(fem,...)` analyzes and updates the geometry data for the model defined by `fem`.

`fem = geomanalyze(fem, draw, ...)` analyzes and updates the geometry data in the model defined by `fem`. `draw` is one or several input arguments given in the same ways as in `geomcsg`.

`[fem, map] = geomanalyze(fem, ...)` additionally returns a cell array of vectors representing the mappings for the different domains. The vector `map{k+1}` describes the mapping of `k`-dimensional domains. Each element in this vector is the associated index of that domain in `fem.geom` before the call.

The function supports the following property/values:

TABLE 1-63: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Geomnum`	integer	`1`	Geometry number
`Imprint`	`on` | `off`	`on`	Make imprints when creating pairs
`Ns`	cell array of strings	`{}`	Names of input solids
`Paircand`	`all` | `none` | cell array of strings	not specified	Specifies the geometries among which pairs are created
`Repairtol`	positive scalar	`1e-6`	Repair tolerance, relative to size of union of input objects
`Sf`	text expression	union of objects	Set formula for Boolean operation
`Solidify`	`on` | `off`	`off`	Create subdomains from empty regions

If the property `paircand` is not specified, the geometry data in the model is the result of the boolean operation specified in the property `sf`. See `geomcsg` for more details.

If the property `paircand` is specified, an assembly geometry is created. In addition identity pairs are created using the property `imprint`. See `geomgroup` for details.

The above properties are explained in `geomcsg` and `geomgroup`.

Example

The following is an example of a circle containing the source for the model, moving through two different subdomains.

```
% Circle moving through two rectangles
clear fem
draw{1} = rect2(.5,1,'pos',[0 0]);
draw{2} = rect2(.6,1,'pos',[0.5 0]);
draw{3} = circ2(.1,'pos',[0.2 0.5]);
% Create analyzed geometry
fem = [];
fem = geomanalyze(fem,draw,'ns',{'R1','R2','C1'});
% Create mesh
fem.mesh = meshinit(fem,'report','off');
% Set source in circle
fem.appl.mode = 'FlPDEC';
fem.appl.equ.f = {0 0 1};
fem.appl.equ.c = 1;
fem.appl.bnd.h = 1;
fem = multiphysics(fem);
% Assemble and solve
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem,'report','off');
% Plot solution
postplot(fem,'tridata','u')
% Start loop and move geometry
nSteps = 5;              % number of steps in loop
dist = .75/nSteps;       % distance to move every step
for i = 1:nSteps
  % Modify geometry in draw structure
  c1 = drawgetobj(fem,'C1');
  fem = drawsetobj(fem,'C1',move(c1, dist, 0));
  % Re/analyze geometry, and update boundary conditions
  fem = geomanalyze(fem);
  % Create mesh for new geometry
  fem.mesh = meshinit(fem,'report','off');
  % Update equation system
  fem = multiphysics(fem);
  % Assemble and solve new equation system
  fem.xmesh = meshextend(fem);
  fem.sol = femstatic(fem,'report','off');
  % Plot solution
  figure, postplot(fem,'tridata','u')
end
```

Compatibility

The default for `Repairtol` has changed from 1e-10 to 1e-6 in version 3.5.

See Also

`geomcsg`, `geomgroup`, `geomedit`

Purpose

Create rectangular array of geometry objects.

Syntax

```
cg1 = geomarrayr(g1,dx)
cg1 = geomarrayr(g1,dx,n)
cg2 = geomarrayr(g2,dx,dy)
cg2 = geomarrayr(g2,dx,dy,n)
cg2 = geomarrayr(g2,dx,dy,nx,ny)
cg3 = geomarrayr(g3,dx,dy,dz)
cg3 = geomarrayr(g3,dx,dy,dz,n)
cg3 = geomarrayr(g3,dx,dy,dz,nx,ny,nz)
```

Description

`cg1 = geomarrayr(g1,dx)` distributes copies of the 1D geometry object `g1` with absolute displacements, `dx` with respect to geometry object `g1`. `cg1` is a cell array with the distributed geometry objects. The `cg1` cell array has the same size as `dx`.

`cg1 = geomarrayr(g1,dx,n)` distributes copies of the 1D geometry object `g1`, `n` times with the relative scalar displacements `dx`.

`cg2 = geomarrayr(g2,dx,dy)` distributes copies of the 2D geometry object `g2` with absolute displacements, `dx` and `dy` with respect to geometry object `g2`. The displacements `dx` and `dy` are matrices of equal size. `cg2` is a cell array with the distributed geometry objects. The `cg2` cell array has the same size as `dx` and `dy`.

`cg2 = geomarrayr(g2,dx,dy,n)` distributes copies of the 2D geometry object `g2`, `n` times with the relative scalar displacements `dx` and `dy`.

`cg2 = geomarrayr(g2,dx,dy,nx,ny)` distributes copies of the 2D geometry object `g2`, `nx`, and `ny` times in corresponding directions, with the relative displacements `dx` and `dy`. The geometry `g2` is included as the first item in the output cell array `cg2`.

`cg3 = geomarrayr(g3,dx,dy,dz)` distributes copies of the geometry object `g3` with absolute displacements, `dx`, `dy`, and `dz` with respect to geometry object `g3`. The displacements `dx`, `dy`, and `dz` are matrices of equal size. `cg3` is a cell array with the distributed geometry objects. The `cg3` cell array has the same size as `dx`, `dy`, and `dz`.

`cg3 = geomarrayr(g3,dx,dy,dz,n)` distributes copies of the geometry object `g3`, `n` times with the relative scalar displacements `dx`, `dy`, and `dz`.

`cg3 = geomarrayr(g3,dx,dy,dz,nx,ny,nz)` distributes copies of the geometry object `g3`, `nx`, `ny`, and `nz` times in corresponding directions, with the relative displacements `dx`, `dy`, and `dz`. The geometry `g3` is included as the first item in the output cell array `cg3`.

The input argument g1, g2, or g3 could also be a cell array of geometry objects. In that case the corresponding output argument cg1, cg2, or cg3 is a cell array of cell arrays.

Example

The following commands are used to create a block object with four equally sized holes.

```
g=geomcsg(geomarrayr(cylinder3,4,4,0,2,2,1));
g2=block3(10,14,5,'corner',[-3 -5 -4])-g;
geomplot(g2)
```

See Also

geom0, geom1, geom2, geom3, move

Purpose Compose and coerce geometry objects.

Syntax `[g,...]=geomcoerce(class,ol,...)`

Description `g=geomcoerce(class,ol)` forms the union of the geometry objects in the cell array `ol`, coerces the composite object to the class `class`, and returns the coerced geometry object in `g`. `Class` is one of the strings: `solid`, `face`, `curve`, or `point`.

The function `geomcoerce` accepts the following property/values:

TABLE 1-64: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Out`	`stx` | `ftx` | `ctx` | `ptx` | cell array of these		Outputs (except the output geometry)
`Repairtol`	positive scalar	`1e-6`	Repair tolerance, relative to size of union of inputs

For information on the geometry tables `stx`, `ftx`, `ctx`, and `ptx`, see `geomcsg`.

Compatibility The default for `Repairtol` has changed from 1e-10 to 1e-6 in version 3.5.

See Also `geomcomp`, `geomcsg`, `geomanalyze`

Purpose

Compose (analyze) geometry objects.

Syntax

```
[g,...]=geomcomp(ol,...)
```

Description

`[g,...]=geomcomp(ol,...)` composes the geometry objects in the cell array `ol` and returns the composite (analyzed) geometry in `g`.

The function `geomcomp` accepts the following property/values:

TABLE 1-65: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Edge`	`all` | `none`	`none`	Delete isolated edges on a face (3D). Delete interior edges and edges not adjacent to a subdomain (2D).
`Face`	`all` | `none`	`none`	Delete interior faces (3D)
`Ns`	cell array of strings		Names of input solids
`Out`	`stx` | `ftx` | `ctx` | `ptx` | cell array of these		Outputs (except the output geometry)
`Point`	`all` | `none`	`none` in 3D `none` in 2D `all` in 1D	Delete isolated vertices on a face (3D). Delete isolated vertices in a subdomain (2D).
`Repairtol`	positive scalar	`1e-6`	Repair tolerance, relative to size of union of inputs
`Sf`	text expression	union of objects	Set formula for Boolean operation

The function works as follows:

1 The solids among the input objects are combined using the set formula in the property `Sf` (the default is union). The operators `+`, `*`, and `-` correspond to the set operations union, intersection, and difference. The names of the solids are given by the property `Ns`.

2 The union of the resulting solid with the input non-solids is formed.

3 Faces, edges, and vertices are deleted according to the properties `Face`, `Edge`, `Point`, see `geomdel`.

4 If all input objects are of the same class (solid, face, curve, point), the result is an object of the same class.

5 If requested, the outputs `stx`, `ftx`, `ctx`, and `ptx` are computed, see `geomcsg`.

Compatibility The default for `Repairtol` has changed from 1e-10 to 1e-6 in version 3.5.

See Also `geomcsg`, `geomdel`, `geomanalyze`, `geomcoerce`

Purpose

Analyze geometry model.

Syntax

```
g = geomcsg(fem,...)
g = geomcsg(draw,...)
g = geomcsg(sl,...)
g = geomcsg(sl,fl,...)
g = geomcsg(sl,fl,cl,...)
g = geomcsg(sl,fl,cl,pl,...)
g = geomcsg(sl,cl,...)
g = geomcsg(sl,cl,pl,...)
g = geomcsg(sl,pl,...)
[g,st] = geomcsg(sl,...)
[g,st,ft] = geomcsg(sl,fl,...)
[g,st,ft,ct] = geomcsg(sl,fl,cl,...)
[g,st,ft,ct,pt] = geomcsg(sl,fl,cl,pl,...)
[g,st,ct] = geomcsg(sl,cl,...)
[g,st,ct,pt] = geomcsg(sl,cl,pl,...)
[g,st,pt] = geomcsg(sl,pl,...)
[g,...] = geomcsg(sl,...,'Out',{'g' ...},...)
```

Description

`g = geomcsg(fem)` analyzes the *geometry model* in `fem.draw`, by performing *Boolean operations* on the solid objects and superimposing the objects of lower dimension on top of the result of the Boolean operations. The result is an *analyzed geometry* object `g`.

`g = geomcsg(draw)` analyzes the *geometry model* `draw` and returns the geometry object `g`.

`g = geomcsg(ol)` decomposes the geometry objects in the cell array `ol` into the analyzed geometry object `g`.

`g = geomcsg(sl,fl,...)` decomposes the 3D solid objects `sl` and the 3D face objects `fl` into the analyzed 3D geometry `g`.

`g = geomcsg(sl,fl,cl,...)` decomposes the 3D solid objects `sl`, the 3D face objects `fl`, and the 3D curve objects `cl` into the analyzed 3D geometry `g`.

`g = geomcsg(sl,fl,cl,pl,...)` decomposes the 3D solid objects `sl`, the 3D face objects `fl`, the 3D curve objects `cl`, and the 3D point objects `pl` into the analyzed 3D geometry `g`.

`g = geomcsg(sl,cl,...)` decomposes the 2D solid objects `sl` and the 2D curve objects `cl` into the analyzed 2D geometry `g`.

`g = geomcsg(sl,cl,pl,...)` decomposes the 2D solid objects `sl`, the 2D curve objects `cl`, and the 2D point objects `pl` into the analyzed 2D geometry `g`.

`g = geomcsg(sl,pl,...)` decomposes the 1D solid objects `sl` and the 1D point objects `pl` into the analyzed 1D geometry `g`.

`[g,st] = geomcsg(sl)` additionally returns a *solid table*, `st`, which relates the original solid objects in `sl` to the subdomains in `g`.

`[g,st,ft] = geomcsg(sl,fl,...)` additionally returns a face table, `ft`, which relates the original face objects in `fl` to the face segments in `g`.

`[g,st,ft,ct] = geomcsg(sl,fl,cl,...)` additionally returns the curve table, `ct`, which relates curve objects in `cl` to edge segments in `g`.

`[g,st,ft,ct,pt] = geomcsg(sl,fl,cl,pl,...)` additionally returns the point table, `pt`, which relates point objects in `pl` to vertices in `g`.

`[g,ct] = geomcsg(sl,cl,...)` additionally returns the curve table, `ct`, for 2D geometry objects.

`[g,ct,pt] = geomcsg(sl,cl,pl,...)` additionally returns the point table, `pt`, for 2D geometry objects.

`[g,pt] = geomcsg(sl,pl,...)` additionally returns the point table, `pt`, for 1D geometry objects.

`sl`, `fl`, `cl`, and `pl` are cell arrays containing solid objects, face objects, curve objects, and point objects, respectively.

`st`, `ft`, `ct`, and `pt` are sparse matrices where column number *i* corresponds to the *i*th object in `sl`, `fl`, `cl`, or `pl`, and row number *j* corresponds to the object index of the corresponding geometric entity in `g`.

The function `geomcsg` accepts the following property/values:

TABLE 1-66: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Ns`	cell array of strings		Names of input solids
`Out`	`g` | `st` | `ft` | `ct` | `pt` | `stx` | `ftx` | `ctx` | `ptx` | cell array of these	`g`	Output variables
`Repairtol`	positive scalar	`1e-6`	Repair tolerance, relative to the size of the union of the input objects

TABLE 1-66: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Sf`	text expression	union of objects	Set formula for boolean operation
`Solidify`	`off` | `on`	`off`	Create subdomains from empty regions

The function works as follows:

1. The solids among the input objects are combined using the set formula in the property `Sf` (the default is union). The operators `+`, `*`, and `-` correspond to the set operations union, intersection, and difference. The names of the solids are given by the property `Ns`.
2. The union of the resulting solid with the input non-solids is formed.
3. If `Solidify='on'`, empty regions are converted to solid subdomains.
4. If requested, the outputs `stx`, `ftx`, `ctx`, `ptx`, `st`, `ft`, `ct`, and `pt` are computed.

The outputs `stx`, `ftx`, `ctx`, and `ptx` correspond to `st`, `ft`, `ct`, and `pt` but contain more detailed information about the relations between the original solid objects, face objects, curve objects, and point objects and the subdomains, faces, edge segments, and vertices in the geometry object `g`. `stx`, `ftx`, `ctx`, and `ptx` are cell arrays of the same length as the cell array `ol = {sl{:},fl{:},cl{:},pl{:}}`. Each cell corresponds to a geometry object in `ol` and contains a sparse matrix whose elements encode maps from the geometric entities in that object to those in the output object `g`.

`stx` is a cell array of sparse matrices, each corresponding to a geometry object in `ol`. In each matrix, column number i corresponds to subdomain number $i - 1$ in the corresponding object in `ol`, and row number j corresponds to the final subdomain index $j - 1$ in `g`. In both cases, subdomain index 0 refers to the complement of the union of all subdomains. If a geometry object in `ol` contains no subdomains, the single column of the corresponding matrix in `stx` shows how subdomain 0 is mapped to `g`.

`ftx` (in 3D only) is a cell array of sparse matrices, each corresponding to a geometry object in `ol`. In each matrix, column number i corresponds to face number i in the corresponding object in `ol`, and row number j corresponds to the final face index j in `g`. If a geometry object in `ol` contains no faces, the corresponding matrix in `ftx` will be empty.

ctx (in 2D and 3D only) is a cell array of sparse matrices, each corresponding to a geometry object in ol. In each matrix, column number *i* corresponds to edge number *i* in the corresponding object in ol, and row number *j* corresponds to the final edge index *j* in g. If a geometry object in ol contains no edges, the corresponding matrix in ctx will be empty.

ptx is a cell array of sparse matrices, each corresponding to a geometry object in OL. In each matrix, column number *i* corresponds to vertex number *i* in the corresponding object in ol, and row number *j* corresponds to the final vertex index *j* in g. If a geometry object in ol contains no vertices, the corresponding matrix in ptx will be empty.

Ns is a cell array of variable names that relates the elements in sl to variable names in sf. Each element in ns contains a variable name. Each such variable assigns a name to the corresponding solid object in sl. This way you can refer to a solid object in sl in the set formula sf.

Sf represents a *set formula* with variable names from ns. The operators +, *, and - correspond to the set operations union, intersection, and set difference, respectively. The precedence of the operators + and - are the same. * has higher precedence. You can control the precedence with parentheses.

Geometry Model

The geometry model fem.draw contains the following fields:

TABLE 1-67: GEOMETRY MODEL OR DRAW STRUCTURE

FIELD	1D	2D	3D	DESCRIPTION
s.objs	√	√	√	Cell array of solid objects
s.name	√	√	√	Cell array of names (default for the property ns)
s.sf	√	√	√	String with Boolean expression (default for the property sf)
f.objs			√	Cell array of face objects
f.name			√	Cell array of names (ignored by geomcsg)
c.objs		√	√	Cell array of curve objects
c.name		√	√	Cell array of names (ignored by geomcsg)

TABLE 1-67: GEOMETRY MODEL OR DRAW STRUCTURE

FIELD	1D	2D	3D	DESCRIPTION
`p.objs`	√	√	√	Cell array of point objects.
`p.name`	√	√	√	Cell array of names (ignored by `geomcsg`)

Examples

3D Geometries

Perform a solid operation on two intersecting cylinders:

```
s1=cylinder3(2,2,[0.5 0.5 -1],[0 0 1]);
s2=cylinder3(1,1,[0.5 0.5 -0.5],[0 0 1]);
[g,st,stx]=geomcsg({s1,s2},'out',{'g','st','stx'}, ...
    'ns',{'Cyl1','Cyl2'},'sf','Cyl1-Cyl2');
```

To easily create solid objects, use the overloaded operators `+`, `*`, and `-`, instead of calling `geomcsg`:

```
s=s1-s2;
geomplot(s,'facemode','off')
```

`s` and `g` are equivalent, except that `g` is a `geom3` object while `s` is a `solid3` object.

2D Geometries

Create a unit circle solid object and a unit square solid object:

```
c1 = circ2;
sq1 = square2;
g = geomcsg({c1 sq1},{},'ns',{'a' 'b'},'sf','a-b');
```

Using object arithmetic for solid objects, the same result can be obtained by typing.

```
g = c1-sq1;
```

You can plot the geometry object by

```
geomplot(g,'sublabel','on','edgelabel','on')
```

or just

```
geomplot(g)
```

You can obtain the number of subdomains and edge segment by just typing `g` or by explicitly getting the object properties.

```
g
get(g,'nmr')
get(g,'nbs')
```

There is one subdomain, with five edge segments, three circle edge segments, and two line edge segments.

1D Geometries

Create a simple 1D geometry by composing two 1D solids:

```
g1 = geomcsg({solid1([0 0.1 4]),solid1([3 4])});
geomplot(g1,'pointlabel','on','sublabel','on')
```

The resulting geometry consists of three subdomains and four vertices.

Compatibility The default for `Repairtol` has changed from 1e-10 to 1e-6 in version 3.5.

See Also `geomanalyze`, `geomgroup`, `geomdel`, `geomcoerce`, `geomcomp`, `geominfo`, `geomplot`

Purpose

Delete points, edges, or faces in a geometry.

Syntax

```
[g,...] = geomdel(g1,...)
g = geomdel(g1)
```

Description

`[g,...] = geomdel(g1,...)` deletes points, edges, or faces in the geometry object `g1` according to the specified properties. The resulting object is of the same type (solid, face, curve, or point) as the original one.

`g = geomdel(g1)` deletes all interior boundaries.

The function `geomdel` accepts the following property/values:

TABLE 1-68: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Edge`	integer vector | `all` | `none`	`all`	Specifies which edges that are deleted
`Face`	integer vector | `all` | `none`	`all`	Specifies which faces that are deleted
`Out`	`stx` | `ftx` | `ctx` | `ptx`	`none`	Output variables (only in 3D). For more information, see the entry geomcsg
`Point`	integer vector | `all` | `none`	`all` in 3D `none` in 2D `all` in 1D	Specifies which vertices that are deleted
`Subdomain`	integer vector | `all` | `none`	none	Specifies which subdomains that are deleted

In 1D, `Point` can either be an array of integers specifying which points that are deleted or one of the strings `all` or `none`. `all` means that all interior points are deleted and `none` that no points are deleted. The default value is `all`.

In 2D, `Edge all` means that all interior boundaries and edges outside any subdomain are deleted and `none` that no edges are deleted. The default value is `all`. `Point all` means that all vertices lying inside a subdomain are deleted and `none` that no vertices are deleted. The default value is `none`. Only isolated vertices can be deleted.

In 3D, `Face all` means that all faces inside or between subdomains are deleted and `none` that no faces are deleted. The default value is `all`. `Edge all` means that all edges lying inside faces are deleted and none that no edge segments are deleted. The default value is `all`. Only edge segments that are not face boundaries can be deleted. `Point all` means that all vertices lying in faces are deleted and `none` that no vertices are deleted. The default value is `none`. Only vertices that are not adjacent to an edge can be deleted.

Examples

The following command generates a block with a face inside, partitioning the subdomain into two parts:

```
g = geomcsg({block3},...
{face3([0.5 0.5;0.5 0.5],[0 1;0 1],[0 0;1 1])});
```

Remove the inner face and all interior edge segments:

```
g1=geomdel(g);
```

See Also

`geomcsg`

Purpose

Edit geometry object.

Syntax

```
[g,...] = geomedit(g0,...)
g1 = geomedit(g)
g2 = geomedit(g, g1)
```

Description

`g1 = geomedit(g, ...)` splits the geometry object `g` into primitive objects `g1` that can be edited. Each object in `g1` is associated with the object `g` so that it is possible to recreate the composite object again.

`g2 = geomedit(g, g1)` creates the geometry `g2` by using the geometry objects in the cell array `g1`, with associative information to `g`, to create a new composite geometry object that is as similar as possible to `g`.

`geomedit` only works for 2D geometries.

The function `geomedit` accepts the following property/values:

TABLE 1-69: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Out`	`stx` | `ctx` | `ptx`	none	Output variables. For more information, see the entry geomcsg

Examples

The following commands create a geometry containing eight curves, then splits the geometry into primitive objects, and finally recreates the geometry with one primitive object omitted:

```
g = curve2(rect2+circ2);
gg = geomedit(g);
[g2, ctx] = geomedit(g, gg([1:4 6:end]), 'out', {'ctx'});
```

See Also

`geomcsg`, `geom0`, `geom1`, `geom2`, `geom3`, `geomanalyze`

Purpose Export geometry objects to file.

Syntax `geomexport(filename, geoms,...)`

Description `geomexport(filename, geoms...)` exports the geometry data in the cell array `geoms` of geometry objects to a file.

`filename` can be any of the following formats:

TABLE 1-70: VALID FILE FORMATS

FILE FORMAT	NOTE	FILE EXTENSIONS
COMSOL Multiphysics Binary		`.mphbin`
COMSOL Multiphysics Text		`.mphtxt`
Parasolid Binary	1	`.x_b`
Parasolid Text	1	`.x_t`
DXF		`.dxf`

Note 1: This format requires a license for the COMSOL CAD Import Module. Only geometries imported using the CAD Import Module (cad3part objects) can be exported to Parasolid format.

The following properties are supported

TABLE 1-71: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Report`	`on` | `off`	`on`	Display a progress window

`Report` determines if a progress window appears during the call.

Examples

```
r = rect2;
geomexport('foo.dxf',{r})
b = block3;
geomexport('bar.mphtxt',{b})
```

Diagnostics `geomexport` replaces the functionality of the 3.1 function `dxfwrite`.

See Also `geom0, geom1, geom2, geom3, meshexport, geomimport`

Purpose

Geometry M-file.

Syntax

```
ne = geomfile
d = geomfile(bs)
[x,y] = geomfile(bs,s)
```

Description

The Geometry M-file is a template format for a user-specified M-file that contains the complete geometry for a model. You can specify both 1D and 2D geometries by using the Geometry M-file format. The `geomfile` format is not supported in 3D.

1D Geometry

For 1D geometries, the Geometry M-file essentially contains a set of points, and geometry information on the intervals between these point.

`ne = geomfile` returns the number of boundary points `ne`.

`d = geomfile(bs)` returns a matrix `d` with one column for each boundary point specified in `bs`, with the following contents:

- Row 1 contains the x-coordinate of the boundary point
- Row 2 contains the label of the "up" subdomain ("up" is the positive direction)
- Row 3 contains the label of the "down" subdomain ("down" is the negative direction)

The complement of the union of all subdomains is assigned the subdomain number 0.

2D Geometry

2D subdomains are represented by parameterized edge segments. Both the subdomains and edge segments are assigned unique positive numbers as labels. The edge segments cannot overlap. The full 2D problem description can contain several nonintersecting subdomains, and they can have common interior boundary segments. The boundary of a subdomain can consist of several edge segments. Each subdomain boundary need to consist of at least two edge segments. All edge segment junctions must coincide with edge segment endpoints.

`ne = geomfile` returns the number of edge segments `ne`.

`d = geomfile(bs)` returns a matrix `d` with one column for each edge segment specified in `bs`, with the following contents:

- Row 1 contains the start parameter value
- Row 2 contains the end parameter value

- Row 3 contains the label of the left-hand subdomain (left with respect to direction induced by start and end from row 1 and 2)
- Row 4 contains the label of the right-hand subdomain

The complement of the union of all subdomains is assigned the subdomain number 0.

`[x,y] = geomfile(bs,s)` produces coordinates of edge segment points. `bs` specifies the edge segments and `s` the corresponding parameter values. `bs` can be a scalar.

Examples

The function `cardg` defines the geometry of a cardioid:

$$r = 2(1 + \cos(\phi))$$

```
function [x,y]=cardg(bs,s)
%CARDG Geometry File defining the geometry of a cardioid.
nbs=4;

if nargin==0
  x=nbs;
  return
end
dl=[  0      pi/2   pi       3*pi/2
      pi/2   pi     3*pi/2   2*pi;
      1      1      1        1
      0      0      0        0];

if nargin==1
  x=dl(:,bs);
  return
end

x=zeros(size(s));
y=zeros(size(s));
[m,n]=size(bs);
if m==1 & n==1
  bs=bs*ones(size(s));  % expand bs
elseif m~=size(s,1) & n~=size(s,2),
  error('bs must be scalar or of same size as s');
end

r=2*(1+cos(s));
x(:)=r.*cos(s);
y(:)=r.*sin(s);
```

You can test the function by typing:

```
clear fem
fem.geom = 'cardg'
geomplot(fem), axis equal
fem.mesh = meshinit(fem);
meshplot(fem), axis equal
```

Then solve the PDE problem $-\Delta u = 1$ on the geometry defined by the cardioid. Use Dirichlet boundary conditions $u = 0$ on $\partial\Omega$. Finally plot the solution.

```
fem.equ.c = 1;
fem.equ.f = 1;
fem.bnd.h = 1;
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem);
postsurf(fem,'u')
```

Cautionary

The Geometry M-file format is not supported in 3D.

In 2D, each subdomain boundary must consist of at least two edge segments.

See Also

`geom0`, `geom1`, `geom2`, `geom3`, `geominfo`, `meshinit`, `meshrefine`

Purpose Retrieves coordinates for a work plane.

Syntax `[p_wrkpln,localsys] = geomgetwrkpln(type,args)`

Description `[p_wrkpln,localsys] = geomgetwrkpln(type,args)` returns the coordinate matrix `p_wrkpln`, spanning a plane, and the local coordinate system `localsys` of that plane, according to the string `type` and the cell array `args`.

The columns of the 3-by-3 coordinate matrix `p_wrkpln` contain point coordinates for 3 non co-linear points spanning a work plane with a local *z* direction defined by `lz = cross(p(:,2)-p(:,1),(p(:,3)-p(:,1)))`. The vector `lx = p(:,2)-p(:,1)` is defined to be the local *x*-axis, and the local *y*-axis is defined as `ly = cross(lz,p(:,2)-p(:,1))`. The corresponding normalized unit vectors are `nlx`, `nly`, and `nlz`, respectively.

The local coordinate system `localsys` is formed as `localsys = [p(:,1),nlx,nly,nlz]`, where `p(:,1)` is the position of the local origin and `nlx`, `nly`, and `nlz` specifies unit vectors in the direction of the positive local coordinate axes.

Work Plane of Type Explicit

```
[p_wrkpln,localsys] = geomgetwrkpln('explicit',{p_wrkpln1})
```

copies the coordinates `p_wrkpln1` and forms the corresponding local coordinate system.

Work Plane of Type Quick

```
[p_wrkpln,localsys] = geomgetwrkpln('quick',{coordplane,offset})
```

forms a work plane parallel to the coordinate plane defined by `coordplane`, which can have any of the values `xy`, `yz`, or `zx`. The real scalar `offset` specifies the signed offset from the coordinate plane.

Work Plane of Type FaceParallel

```
[p_wrkpln,localsys] =
     geomgetwrkpln('faceparallel',{g,fn,dir,offset})
```

creates a work plane, parallel to face `fn` in 3D geometry object `g`. The direction `dir` takes the values `+1` or `-1` and specifies if the local *z*-axis, `localsys(:,4)`, should be in the direction of the face's normal, or reversed normal, respectively. The scalar `offset` specifies the displacement along the local *z*-axis for the work plane with respect to the face. The face, with number `fn`, must be planar. If the face is not planar, within the system tolerance, an error message occurs.

Work Plane of Type EdgeAngle

```
[p_wrkpln,localsys] =
     geomgetwrkpln('edgeangle',{g,en,angle,fn,dir})
```

creates a work plane rotated `angle` radians about the edge `en` in the 3D geometry object `g`. The zero-angle is defined by the tangent plane of the face with face number `fn`. The face `fn` must be adjacent to the edge `en` and the face must have a single tangent plane common to all points of the edge `en`. The direction `dir` takes the values `+1` or `-1` and specifies if the rotation should be in positive or negative direction, with respect to the direction of the edge `en`, respectively. The matrix `p_wrkpln`, and the system `localsys`, referred to above, are formed based on the coordinate system induced by the selected edge, where the edge becomes the positive x-axis.

Work Plane of Type Vertices

```
[p_wrkpln,localsys] =
     geomgetwrkpln('vertices',{gl,vn,dir,offset})
```

creates a work plane spanned by the three vertices `vn` in the 3D geometry objects `gl`. `vn` is an index vector of length 3, corresponding to the entries in the cell array `gl`. The direction `dir` takes the values `+1` or `-1` and specifies if the local z-axis, `localsys(:,4)`, should be in the direction of the positive normal, or reversed normal, respectively. The positive normal is defined as the cross product of the vectors in the direction from `vn(1)` to `vn(2)`, and from `vn(1)` to `vn(3)`, respectively. The scalar `offset` specifies the displacement along the local z-axis for the work plane with respect to the plane containing the vertices `vn`. The matrix `p_wrkpln`, and the system `localsys`, referred to above, are formed based on the chosen vertices such that, for `offset = 0`, the vector from `vn(1)` to `vn(2)` forms the local x-axis. The local y-axis is formed based on this vector and the local z-axis, as to produce a right-handed local coordinate system.

See Also `geomposition`

Purpose Groups geometry objects into an assembly.

Syntax

```
[g, ...] = geomgroup(gl, ...)
[g, ...] = geomgroup(draw, ...)
[g, ...] = geomgroup(fem, ...)
```

Description `[g, ...] = geomgroup(gl, ...)` creates an assembly object `g` from the geometry objects in the cell array `gl`.

`[g, ...] = geomgroup(draw, ...)` creates an assembly object `g` from the geometry objects in the draw struct `draw`.

`[g, ...] = geomgroup(fem, ...)` creates an assembly object `g` from the geometry objects in the draw struct `fem.draw`.

Note that the parts of assembly `g` are not identical to `gl`. They are canonized and may have additional domains due to imprints.

The function supports the following properties:

TABLE 1-72: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Imprint`	`on` | `off`	`on`	Make imprints, when creating pair information
`Ns`	cell array of strings		Names of input solids
`Out`	cell array of strings: `g`, `gt`, `st`, `ft`, `ct`, `pt`, `stx`, `ftx`, `ctx`, `ptx`, `pairs`	`{'g'}`	Outputs
`Paircand`	`all` | `none` | cell array of strings	`all`	Specifies the geometries which are used to create the pair information
`Repairtol`	positive scalar	`1e-6`	Repair tolerance, relative to size of union of input objects

Note: If the first syntax from above is used and the property `paircand` is a cell array of strings, also the property `ns` has to be specified.

The output `pairs` is a cell array of sparse matrices containing the pair information of the operation. Element i contains information for pairs of domains of dimension $i-1$. The column refers to the source and the row to the destination.

The output gt is a sparse matrix that relates the parts in the assembly to the original geometry objects. If an object has not been modified during the operation the value in gt is 1, otherwise 2.

Examples

```
[g pairs] = geomgroup({rect2 move(rect2,[1 0.5])},'out',...
{'g' 'pairs'});
[gg,stx,ctx,ptx] = getparts(g,'out',{'stx','ctx','ptx'})
```

Compatibility

The default for Repairtol has changed from 10^{-10} to 10^{-6} in version 3.5.

See Also

geomcsg, geomanalyze, getparts

Purpose Import geometry objects from a file.

Syntax

```
gl = geomimport(filename,...)
```

Description `gl = geomimport(filename,...)` reads the geometry file `filename` and translates the geometry data using the specified properties into a cell array of geometries `gl`.

`filename` can be of any of the following formats:

TABLE 1-73: SUPPORTED FILE FORMATS

FILE FORMAT	NOTE	FILE EXTENSIONS	OBJECT TYPE IN COMSOL MULTIPHYSICS
Autodesk Inventor	2	`.ipt`, `.iam`	CAD object
CATIA V4	2	`.model`	CAD object
CATIA V5	2	`.CATPart`, `.CATProduct`	CAD object
COMSOL Multiphysics Binary		`.mphbin`	CAD and/or COMSOL objects
COMSOL Multiphysics Text		`.mphtxt`	CAD and/or COMSOL objects
DXF		`.dxf`	COMSOL object
GDS	3	`.gds`	COMSOL object
IGES	1	`.igs`, `.iges`	CAD object
NETEX-G	3	`.asc`	COMSOL object
ODB++(X)	3	`.xml`	COMSOL object
Parasolid	1	`.x_t`, `.x_b`	CAD object
Pro/ENGINEER	2	`.prt`, `.asm`	CAD object
SAT	1	`.sat`, `.sab`	CAD object
STEP	1	`.step`, `.stp`	CAD object
STL		`.stl`	COMSOL object
VDA-FS	2	`.vda`	CAD object
VRML		`.wrl`, `.vrml`	COMSOL object

Note 1: This format requires a license for the COMSOL CAD Import Module.

Note 2: This format requires a license for a format-specific module from COMSOL.

Note 3: This format requires a license for any of the following modules: AC/DC Module, MEMS Module, or the RF Module.

This function supports the following properties:

TABLE 1-74: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Check`	`on` | `off`	`on`	Check imported data for errors. (Only when importing CAD Import Module formats.)
`Coercion`	`solid` | `face` | `curve` | `point` | `off`	`solid`	Coerce the imported geometry
`Keepbnd`	`on` | `off`	`on`	Keep boundary entities
`Keepfree`	`on` | `off`	`off`	Keep free edge/point entities
`Keepsolid`	`on` | `off`	`on`	Keep solid entities
`Repair`	`on` | `off`	`on`	Repair imported data
`Repairtol`	positive scalar	`1e-4`	Repair tolerance
`Importtol`	positive scalar	`1e-5`	Absolute repair tolerance used when importing CAD Import Module formats
`Report`	`on` | `off`	`on`	Display a progress window
`Layers`	`cell array`		Determines which layers to be imported when importing DXF, GDS, ODB++, and NETEX-G.

When importing COMSOL Multiphysics files the function ignores all properties, except for `Report`.

For DXF import, the default for `Coercion` is `curve`.

For STL and VRML imports, the command supports all properties in the function `meshenrich`.

`Coercion` can force the import process to knit boundary segments together and possibly try to form solid entities.

`Keepbnd`, `Keepfree`, and `Keepsolid` indicate which type of entities the module should consider in the imported data.

`Repair` determines if the module should process the imported data to improve the quality. These operations include snapping of points, removal of small entities, and improvement of geometric data.

`Repairtol` is a relative tolerance. It indicates the size of entities to remove, which points to snap together, and similar features.

`Importtol` is an absolute tolerance. It indicates the size of entities to remove, which points to snap together, and similar features. It is used when importing a file using the CAD Import module, replacing `Repairtol`.

`Report` determines if a progress window should appear during the call.

The property `Layers` controls which DXF layers are imported. The value can be either a cell array of strings, containing the names of the layers that are to be imported, or an array of integers, where the integers refer to the order in which the DXF layers occur in the DXF file. This property is also used for ODB++, NETEX-G, and GDS import to identify which layers to import, but then only a cell array of strings is allowed as the value. In addition, the ODB++, NETEX-G, and GDS file formats all support the set of properties listed in Table 1-75.

TABLE 1-75: VALID PROPERTY/VALUE PAIRS FOR GDS, ODB++, AND NETEX-G

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Edge`	`on` | `off`	`on`	Keeps interior edges of nets
`Cell`	string		Name of top net or cell to import for GDS and NETEX-G files
`Importtype`	`full3d` | `shell`	`full3d`	Determines if metal layers are imported as solid or faces
`Grouping`	`all` | `layers` `none`	`all`	The grouping of the imported layers, where `all` returns one single object, and `layers` gives you one object per layer. In 2D, `all` is the same as `layer`.
`Bondtype`	`edge` | `block` | `cylinder`	`edge`	Specifies how bond wires are modeled for NETEX-G files
`Importdielectrics`	`on` | `off`	`on`	Import dielectric regions
`Table`	cell matrix		Table representing the settings for layer specific changes (rows x 4). Columns are row number, name, type, and thickness.
`Leftmargin`	positive scalar	0	Distance to the left edge of the dielectric region
`Rightmargin`	positive scalar	0	See `Leftmargin`
`Topmargin`	positive scalar	0	See `Leftmargin`
`Bottommargin`	positive scalar	0	See `Leftmargin`

TABLE 1-75: VALID PROPERTY/VALUE PAIRS FOR GDS, ODB++, AND NETEX-G

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Abovemargin`	positive scalar	`0`	Add extra layer above the imported geometry
`Belowmargin`	positive scalar	`0`	See `Abovemargin`
`Sdim`	`2` | `3`	`2`	Import to 2D or 3D
`Findarcs`	`on` | `off`	`on`	Turns on arc recognition for GDS and NETEX-G files
`Arcradiustol`	positive scalar	`0.4`	Tolerance for arc curvature
`Arcminangle`	positive scalar	`pi/50`	Minimum angle for segments to part of an arc (in radians).
`Arcmaxangle`	positive scalar	`pi/5`	Maximum angle for segments
`Findlines`	`on` | `off`	`on`	Only available when `findarcs` is turned on. Also tries to find segments that lies on a straight line.
`Ignoretext`	`on` | `off`	`on`	Ignore objects marked as text in ODB++ files

For more details, see the "ECAD Import" section in the User's Guide of the AC/DC Module, MEMS Module, or the RF Module.

Examples

The following example imports the DXF file `demo1.dxf` as a curve2 object:

```
filepath = which('demo1.dxf');
g = geomimport(filepath);
geomplot(g{1})
```

Diagnostics

`geomimport` replaces the functionality of the COMSOL Multiphysics 3.0 functions `dxfread`, `gdsread`, `igesread`, `stlread`, and `vrmlread`. We no longer support those functions and their properties, and therefore do not document them.

See Also

`geom0`, `geom1`, `geom2`, `geom3`, `meshimport`, `geomexport`

Purpose Retrieve geometry information.

Syntax

```
[xx,...] = geominfo(geom,'Out', {'xx' ...},...)
```

Description `[xx,...] = geominfo(geom,'Out', {'xx',...},...)` retrieves geometry information specified in property `Out` from the *analyzed geometry* `geom`.

`geom` is an analyzed geometry, which is a geometry object, a mesh object or a Geometry M-file. The two latter formats are not supported in 3D. The Decomposed Geometry matrix of the PDE Toolbox is supported as well, but this alternative may be eliminated in future releases. For details on analyzed geometries, see the chapter "Geometry Modeling and CAD Tools" on page 23 of the *COMSOL Multiphysics User's Guide* or the entries `geomcsg`, `geomfile`, and `meshinit` in this manual.

In the following description, a *geometric entity* refers to a *vertex*, an *edge segment*, a *face segment*, or a *subdomain*.

- If `geom` is a 1D analyzed geometry, the geometric entities are vertices and 1D subdomains that are bounded by vertices.
- If `geom` is a 2D analyzed geometry, the geometric entities are vertices, 1D edge segments bounded by vertices, that are assumed to be smooth in the interior, that is, sufficiently differentiable, and 2D subdomains bounded by edge segments.
- If `geom` is a 3D analyzed geometry, the geometric entities are vertices, 1D edge segments bounded by vertices, 2D smooth face segments bounded by edges, and 3D subdomains bounded by face segments.

The function `geominfo` accepts the following property/values:

TABLE 1-76: VALID PROPERTY/VALUE PAIRS.

PROPERTY	1D	2D	3D	DESCRIPTION
`Od`	√	√	√	Vector that contains geometric entity dimension numbers
`Odp`	√	√	√	Matrix with columns that contain geometric entity dimension number pairs
`Out`	√	√	√	Output arguments, cell array containing strings specifying the output arguments
`Par`	√	√	√	Cell array, where each element is a cell array containing two matrices defining geometric entity number and parameter values

Out specifies the geometry information to retrieve and return as output arguments. It is a cell array that can contain string equal to the entries given in Table 1-77.

Par is a cell array containing parameter values and corresponding numbers of geometric entities. Par{m} is a cell containing the matrix Bm and the matrix Sm. Bm is of size nm1-by-nm2, and gives the numbers of entities of dimension d for which the parameters, in the nm1-by-nm2-by-d array Sm, are valid. Bm can also be a scalar or a vector, and as such it is expanded to the size of Sm. Note that the size of the third dimension in Sm, that is, d, defines if the entity number refers to a vertex (d = 0 or Par{m} = {Bm}), an edge (d = 1), or a face (d = 2).

The following table lists the valid outputs to geominfo.

TABLE 1-77: OUTPUT ARGUMENTS

OUT	1D	2D	3D	DESCRIPTION	INPUT PROPERTY
gd	√	√	√	Geometry dimension	
no	√	√	√	Number of objects of the dimensions specified in Od	Od
adj	√	√	√	Adjacency relations of entities in Odp	Odp
xx	√	√	√	Coordinate information	Par
dx		√	√	First-order derivative information	Par
ddx		√	√	Second-order derivative information	Par
nor		√	√	Normal vector information	Par
ff1			√	First fundamental matrices	Par
ff2			√	Second fundamental matrices	Par
crv		√	√	Curvature information	Par
rng		√	√	Parameter range of geometric entities	Od
ud	√	√	√	Up and down subdomains	
se	√	√	√	Start and end vertices of all 1D primitive objects	
nmr	√	√	√	Number of subdomains	
nbs	√	√	√	Number of boundary segments	
mp		√		Coordinates of vertices	
sd		√		Vertex subdomain numbers	

no is a vector of the same size as Od, containing the number of primitive objects of the dimension as specified in Od.

adj is a cell array of adjacency matrices, where adj{k} corresponds to Odp(:,k), and is a sparse matrix where abs(sign(adj{k}(i,j))) = 1 iff object i of dimension Odp(1,k) is adjacent to object j of dimension Odp(2,k). If the relation Odp(1,k) and Odp(2,k) can be given an orientation, the matrix entries +1 and −1 denotes positive or negative orientation, respectively. If both oriented and non orientable relations exist, −1, +1, and +2 are used, where +2 indicates a non oriented relation. If Odp is a vector of length 2, then adj is a sparse matrix. For subdomain information, the 0-domain is represented as output domain number 1. Thus, there is always an offset of 1 for subdomains.

xx is a cell array of same size as Par containing coordinate information, where xx{m} is an nm1-by-nm2-by-gd array, where gd is the geometry dimension, and nm1 and nm2 are given from the size of Par{m}{2}. If the outer curly brackets in Par are not present, then xx is an n1-by-n2-by-gd array.

dx is a cell array of same size as Par containing first order derivative information for edges or faces. For edges, the dx{m} has the same format as xx{m} above. For faces dx{m} is a nm1-by-nm2-by-3-by-2 array, where the last dimension refers to the two vectors, formed by the derivatives of u and v respectively, spanning the tangent plane.

ddx is a cell array of same size as Par containing second order derivative information for edges or faces. For edges, ddx{m} has the same format as xx{m}. For faces ddx is a nm1-by-nm2-by-3-by-2-by-2 array, where the last two dimensions refer to the 2-by-2 matrix of second order derivatives in the parameters u and v.

nor is a cell array of same size as Par, where the contents are the normalized normal vectors. They are given on the same format as the contents in xx.

ff1 is a cell array of same size as Par containing the first fundamental matrices of faces, where ff1{m} is an array of size nm1-by-nm2-by-2-by-2. For a parameter point given by the indices im1 and im2, the first fundamental matrix is given by GG = reshape(ff1{m}(im1,im2,:,:),2,2) and the corresponding Jacobian is given by J = reshape(dx{m}(im1,im2,:,:),3,2). It then holds that GG = J'*J.

ff2 is a cell array of same size as Par containing the second fundamental matrices of faces, where ff2{m} is an array of size nm1-by-nm2-by-2-by-2. For a parameter point given by the indices im1 and im2, the second fundamental matrix is given by

DD = reshape(ff2{m}(im1,im2,:,:),2,2). If the corresponding normal derivative DNN and Jacobian J are obtained as above, then DD = -DNN'*J.

crv is a cell array of same size as Par containing curvature information of edges and faces. crv{m} is of size nm1-by-nm2-by-2 in 3D, where for a parameter point defined by the indices im1 and im2, crv{m}(im1,im2,1) is the curvature and crv{m}(im1,im2,2) is the torsion, when referring to an edge. The corresponding values obtained for a face is the Gaussian curvature and the mean curvature, respectively. In 2D, crv{m} is of size nm1-by-nm2 where crv{m}(im1,im2) contains the curvature of an edge for a given parameter.

rng is a cell array of same length as Od, containing parameter range information for edges or faces. For edges, the first row in a matrix corresponds to the starting parameter value at the starting point, and the second row corresponds to the end parameter value at the end point. For faces, the first and third row contains the lower bounds on parameter values for the *u* and *v* parameters respectively. The second and fourth row contains the upper bounds on parameter values for the *u* and *v* parameters respectively. The range for geometry edges is from zero to the arc-length of each edge. If no Od is specified, rng is a matrix of range information for all edge curves, in 2D, or all faces, in 3D.

ud is a matrix containing up (left) and down (right) subdomain numbering for boundary segments, in the first and second row, respectively. One column of ud corresponds to one boundary segment.

sd is a vector containing the subdomain numbering of the vertices of mp. If a vertex is adjacent to more than one subdomain, the contents are NaN.

There is a family of low-level geometry functions used by geominfo, for obtaining the geometric data described above. These can be called directly, which in some cases can be preferred. Their names and descriptions are given in the table below.

TABLE 1-78: LOW-LEVEL GEOMETRY FUNCTIONS

FUNCTION	DESCRIPTION
flgeomadj	Get geometry adjacency matrices
flgeomec	Get curvature information from curve derivatives
flgeomed	Get coordinates and derivatives for geometry edges
flgeomes	Get parameter space size of geometry edge
flgeomfc	Get curvature from fundamental forms
flgeomfd	Get coordinates and derivatives for geometry faces

TABLE 1-78: LOW-LEVEL GEOMETRY FUNCTIONS

FUNCTION	DESCRIPTION
`flgeomff1`	Get first fundamental form from derivatives
`flgeomff2`	Get second fundamental form from derivatives
`flgeomfn`	Get normals from face derivatives
`flgeomfs`	Get parameter space size of geometry face
`flgeomnbs`	Get number of geometry boundary segments
`flgeomnes`	Get number of geometry edge segments
`flgeomnmr`	Get number of subdomains
`flgeomnv`	Get number of vertices
`flgeomsdim`	Get space-dimension of geometry object
`flgeomse`	Get end-point indices of geometry edges
`flgeomud`	Get up-down subdomain numbering of geometry faces
`flgeomvtx`	Get coordinates for geometry vertices

For details on the syntaxes for calling these functions, write `help` followed by the function name on the command line.

Examples

3D Geometries

To demonstrate the `geominfo` command, create a solid block object with a circular curve object on top, using the following commands.

```
g3 = geomcsg({block3},{},...
             {move(embed(circ1(0.3,'pos',[0.5 0.5])),[0 0 1])})
geomplot(g3,'facelabels','on')
```

The generated object `g3` is a solid 3D object consisting of 1 subdomain, 7 faces, 16 edges and 12 vertices. These can be obtained using `geominfo` with the arguments given below.

```
[gd,no,rng,ud,nbs] = geominfo(g3,...
             'out',{'gd' 'no' 'rng' 'ud' 'nbs'},'od',0:3);
```

From the arguments `gd` and `no`, it is clear that `g3` is a 3D object with the number of entities as above. The number of faces is also given in `nbs`, that is, the number of boundary segments. The parameter range of both faces and edges are given in `rng`. These are of importance when setting up parameter arrays for edge/face information evaluation below.

The following commands set up parameter matrices in two different formats, for faces 4, 5, and 7. The parameter range of these faces is $0<u<0.5$, $0<v<0.5$, as given by `rng{3}(:,[4 5 7])`.

```
[u,v] = meshgrid(0:0.1:0.5,0:0.1:0.5);
S1 = reshape([u(:) v(:)],1,36,2);
B1 = 7;
S2(1,:,:) = deal([u(:) v(:)]);
S2(2,:,:) = deal([u(:) v(:)]);
B2 = [4;5];
```

Appropriate parameter values for the bounding edges of face 7, can be obtained by first creating the face-edge adjacency matrix with `geominfo`, and then using this information together with the argument `rng` to set up the parameter vectors. This is done with the following commands.

```
adj = geominfo(g3,'out','adj','odp',[1;2]);
B3 = find(adj{1}(:,7));
for i=1:length(B3)
  S3(i,:,1) = linspace(rng{2}(1,B3(i)),rng{2}(2,B3(i)),10);
end
```

Now, coordinate values of the faces and edges given above, together with coordinates for vertex 3, are obtained as follows.

```
[xx] = geominfo(g3,'out',{'xx'},...
                'par',{{B1 S1} {B2 S2} {B3 S3} {3}})
```

To see the obtained results, simply give the following commands.

```
hold on
plot3(xx{1}(:,:,1),xx{1}(:,:,2),xx{1}(:,:,3),'r.')
plot3(xx{3}(:,:,1),xx{3}(:,:,2),xx{3}(:,:,3),'b.')
```

Finally, derivatives and curvatures, for both faces and edges are with the command below. Note that, both curvature measures for all points at face 7 are 0, as is the torsion for the surrounding curves. The curvature of these curves is however nonzero.

```
[dx,crv] = geominfo(g3,'out',{'dx' 'crv'},...
                    'par',{{B1 S1} {B3 S3}})
```

2D Geometries

Create a solid ellipse, and retrieve coordinates and curvatures for all four edge segments, by the following commands.

```
e = ellip2(0,0,1,2)
[xy,c] = geominfo(e,'out',{'xx','crv'},'par',...
                  {ones(11,1)*[1 2 3 4],(0:0.1:1)'*ones(1,4)})
```

Plot the obtained coordinates of the ellipse by the command.

```
plot(xy(:,:,1),xy(:,:,2),'b-')
```

The curvature of one of the edges are obtained via

```
figure,plot(0:0.1:1,c(:,1))
```

The command below retrieves the number of primitive objects (vertices, edges and subdomains) from geometry file (`geomfile`) `cardg`:

```
no = geominfo('cardg','out',{'no'},'od',[0 1 2])
```

1D Geometries

A 1D geometry consisting of two subdomains, is created by the following command.

```
g1 = solid1([-1 0.2 1])
```

Since no parameter domain exist only coordinates of vertices can be retrieved. The up and down subdomain of every vertex is given in `ud`, and the vertex-subdomain adjacency information is given in `adj`.

```
[xx,ud,adj] = geominfo(g1,'out',{'xx' 'ud' 'adj'},...
                       'par',{2},'odp',[0;1])
```

Note that the same information is given in `ud` and `adj`. The matrix `adj` is directly obtained from `ud` via the command:

```
adj = sparse(repmat(1:3,2,1),ud+1,[ones(1,3);-1*ones(1,3)])
```

Compatibility The FEMLAB 2.3 function `flgeomepol` is obsolete.

See Also `geomcsg`, `geomedit`, `meshinit`

geomobject

Purpose Create geometry object.

Syntax `obj = geomobject(input)`

Description `obj = geomobject(input)` creates a geometry object from `input`.

`input` can be any of the following

- A geometry object. See `geom0, geom1, geom2, geom3`.
- A mesh object. See `femmesh`.
- A geometry M-file name. See `geomfile`.

Note that in 3D, `input` cannot be a Geometry M-file.

See Also `geom0, geom1, geom2, geom3`, `femmesh`, `geomfile`

Purpose

Plot geometry.

Syntax

```
geomplot(fem,...)
geomplot(geom,...)
h = geomplot(fem,...)
h = geomplot(geom,...)
```

Description

`geomplot(fem)` plots the analyzed geometry `fem.geom`. For an extended FEM structure, `xfem.fem{geomnum}.geom` is plotted, where geomnum is 1 by default.

`geomplot(geom)` plots the analyzed geometry `geom`.

`h = geomplot(...)` additionally returns handles to the plotted axes objects.

The *analyzed geometry* can be any of the following geometry representations: a geometry object, a Geometry M-file, or a mesh. The geometry object and Geometry M-file are described in the entries `geomcsg` and `geomfile`, respectively. The mesh data structure is described in the entry `meshinit`.

In 3D, the default plot is a patch plot of the faces with the edge segments and isolated vertices plotted as lines and markers respectively. The face, edge segment, and vertex parts of the plot can be controlled by the property/values starting with `face`, `edge`, and `point` respectively. Subdomains cannot be plotted directly, only indirectly through their adjacent faces.

In 2D, the default plot is a patch plot of the subdomains with the edge segments and vertices plotted as lines and markers, respectively. The subdomain, edge segment, and vertex parts of the plot can be controlled by the properties starting with `sub`, `edge`, and `point`, respectively. You can turn on indication of curve parameter direction by using the property `edgearrows`.

In 1D, the default plot is a line plot of the subdomains with vertices plotted as markers. The subdomain and vertex parts of the plot can be controlled by the properties starting with `sub` and `point`, respectively.

The following table shows the property/value pairs for the `geomplot` command. The interpretation of the properties in 1D, 2D and 3D varies with dimension. The design philosophy has been to keep property interpretation constant over space dimension, but to plot these properties as plot objects of different types.

TABLE 1-79: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Boxcolor`		√		color	`k`	Control polygon color
`Boxstyle`		√		line style	`--`	Control polygon line style
`Ctrlmarker`		√		marker symbol	`o`	Control polygon marker style
`Ctrlmode`		√		`on` | `off`	`off`	Show control polygon
`Detail`		√	√	`fine` | `normal` | `coarse`	`normal`	Geometry resolution
`Edgearrows`		√		`on` | `off`	`off`	Show edge directions with arrows
`Edgecolor`		√	√	color	`k`	Edge color data
`Edgelabels`		√	√	`on` | `off` | list of strings	`off`	Edge label list
`Edgemode`		√	√	`on` | `off`	`on`	Show edges
`Edgestyle`		√	√	line style	`-`	Edge line style
`Facelabels`			√	`on` | `off` | list of strings	`off`	Face label list
`Facemode`			√	`on` | `off`	`on`	Show faces
`Labelcolor`	√	√	√	color	`k`	Label color data
`Linewidth`		√	√	numeric	1	Line width
`Linewidth`	√			numeric	2	Line width
`Markersize`	√	√	√	numeric	6	Marker size
`Mesh`		√	√	mesh	new special mesh	Mesh used to render geometry
`Pointcolor`	√	√	√	color	`b`	Point color data

TABLE 1-79: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Pointlabels`	√	√	√	`on` | `off` | list of strings	`off`	Point label list
`Pointmarker`	√	√	√	marker symbol	`o`	Point marker
`Pointmode`	√	√	√	`on` | `off` | isolated	`on`	Show points
`Sublabels`	√	√	√	`on` | `off` | list of strings	`off`	Subdomain label list
`Submode`	√	√		`on` | `off`	`on`	Show subdomains

In addition, the common plotting properties listed under `femplot` are available.

The properties `sublabels`, `facelabels`, `edgelabels`, and `pointlabels` control the display of subdomain labels, face labels, edge segment labels, and point labels, respectively.

The properties that control marker type or coloring can handle any standard marker or color type in MATLAB. See, for example, the `plot` command in the MATLAB documenation.

Examples

3D Example

Create a simple 3D geometry:

```
c1 = cylinder3(0.5,2,[-1,0,0],[1,0,0]);
c2 = cylinder3(0.2,2,[0,-1,0],[0,1,0]);
g = c1-c2;
```

Plot edges and face labels.

```
geomplot(g,'facemode','off','facelabels','on')
axis equal
```

Plot faces with lighting and without edges and axis in high quality.

```
geomplot(g,'edgemode','off','detail','fine')
light, lighting phong
axis equal, axis off
```

Both faces and edges are plotted by default.

2D Example

Start by creating a simple geometry.

```
clear fem
c1 = circ2;
l1 = curve2([-1,-1,1,1],[-1,1,-1,1]);
p1 = point2(0,0.5);
fem.draw.s.objs = {c1};
fem.draw.c.objs = {l1};
fem.draw.p.objs = {p1};
fem.geom = geomcsg(fem);
```

Plot the standard geometry plot with subdomains indicated as patches, and edge segments and vertices indicated by lines and markers, respectively.

```
geomplot(fem), axis equal
```

Remove patch plot of subdomains, add parameter direction for curves, subdomain numbers, and control polygons.

```
geomplot(fem,'submode','off','edgearrow','on','pointmode',...
         'isolated','sublabels','on','ctrlmode','on')
```

1D Example

Start by creating a simple geometry.

```
clear fem
s1 = solid1([0 0.1 1]);
p1 = point1(2);
fem.draw.s.objs = {s1};
fem.draw.p.objs = {p1};
fem.geom = geomcsg(fem);
```

The standard geometry plot with subdomains indicated as lines, and vertices indicated by markers.

```
geomplot(fem), axis equal
```

Change the color of the vertices to red, add vertex labeling, and change the vertex markers to diamonds.

```
geomplot(fem,'pointcolor','r','pointlabels','on',...
         'pointmarker','diamond')
axis equal
```

Compatibility

The properties `pt`, `ct`, `ft`, and `st` have been removed in FEMLAB 3.1.

Cautionary

The value `numeric` of the `sublabels`, `edgelabels`, and `pointlabels` properties was replaced by `on` in FEMLAB 1.1. The value `numeric` is still supported however, and is equivalent to `on`.

The default for the `ctrlmode` property was changed to `off` in FEMLAB 1.1.

See Also `geomcsg`, `geomedit`

geomposition

Purpose Position 3D geometry object in space using work plane info.

Syntax `g3 = geomposition(g32,p_wrkpln)`

Description `g3 = geomposition(g32,p_wrkpln)` positions the 3D geometry object `g32` in space by transforming the point matrix according to the work plane information in `p_wrkpln`. The geometry `g32` is thus assumed to be defined in the local coordinate system of the work plane.

See `geomgetwrkpln` for more information on work planes and the work plane points representation `p_wrkpln`.

See Also `geomgetwrkpln`

Purpose Spline interpolation.

Syntax `c = geomspline(p,...)`

Description `c = geomspline(p,...)` creates a `curve2` or `curve3` object from point data `p` by spline interpolation. The object generated is a closed or open, C^1 or C^2 continuous, spline.

`p` is a 2-by-`np` (in 2D) or 3-by-`np` (in 3D) matrix that specifies interpolation points.

The function `geomspline` accepts the following property/value pairs:

TABLE 1-80: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Closed`	`auto` | `on` | `off`	`auto`	Closed or open curve
`SplineDir`	2-by-`np` matrix 3-by-`np` matrix		Tangent vectors for the corresponding points in `p`
`SplineMethod`	`uniform` | `chordlength` | `centripetal` | `foley`	`chordlength`	Method for global parameterization

The property `SplineDir` is used to specify a tangent vector for the corresponding point in `p`. This means that the first control point is given and the curve thus generated is only guaranteed to be C^1 (continuous first derivatives). If this property is not given, however, the curve generated is guaranteed to be C^2 (continuous second derivatives). The `SplineMethod` property does not affect the curve if the `SplineDir` property is used.

The property `SplineMethod` controls the method for how to compute the global parameterization of the curve. The global parameterization is a parameter that varies from 0 to 1, from the first interpolated point to the last. For a closed curve the last point is equivalent to the first. The value `uniform` means that the global parameterization is `[0, 1,..., np]/np`. The default value `chordlength` means that the global parameterization is `[0, norm(p(:,2)-p(:,1)), norm(p(:,3)-p(:,2)),..., norm(p(:,np)-p(:,np-1))]/ sum(sqrt(sum((diff(p')').^2)'))`, where the denominator is the total chord length. The values `centripetal` and `foley` are two additional methods that handle irregular point sets `p` more effectively.

The property `Closed` controls the closure of the spline. If `Closed` is `on` the first point is regarded as the last point. The value auto for the property `Closed` generates a closed curve whenever the first and last points in a scaled version of the point set p agree to within 1000*eps in Euclidean distance. Otherwise, an open curve is generated.

On success, c is a curve2, or curve3 object that passes through the points defined by p. If p does not define a spline curve properly, either an error occurs or a line1, curve2, curve3, or circ1 object is created that meets the requirements in some way.

Example

```
% Interpolate irregularly distributed point on a circle.
% First create circle data
phi=0:0.2:2*pi; phi(end)=[];
% Remove some of the points.
phi([1 3 6 7 10 20 21 25 28])=[];
p=[cos(phi);sin(phi)];
% Add some noise.
randn('state',17)
p=p+0.05*randn(size(p));
plot(p(1,:),p(2,:),'r.')
% Interpolate using uniform parameterization.
c=geomspline(p,'splinemethod','uniform','closed','on')
hold on
geomplot(c,'pointmode','off')
% Interpolate using centripetal parameterization.
c=geomspline(p,'splinemethod','centripetal','closed','on')
hold on
geomplot(c,'pointmode','off','edgecolor','b')
axis equal
```

See Also

curve2, curve3

Purpose

Create 3D geometry surface using height data defined on a grid.

Syntax

```
f = geomsurf(x,y,z)
f = geomsurf(z)
s = geomsurf(x,y)
```

Description

`f = geomsurf(x,y,z)` creates a 3D face object `f` on the grid defined by `x` and `y`, using `z` as height data.

`f = geomsurf(z)` is equivalent to `f = geomsurf(1:nx,1:ny,z)`, where `[nx,ny]=size(z)`.

`s = geomsurf(x,y)` creates a 2D solid object `s` corresponding to the syntax `f = geomsurf(x,y,z)` with `z` all zeros.

The function supports the following property:

TABLE 1-81: VALID PROPERTY/VALUE PAIR

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Geomrep`	`bezier` | `mesh` | `spline`	`bezier`	Representation of surface

If `Geomrep` is `bezier`, the returned face object consists of several faces, each using a bilinear parameterization. If `Geomrep` is `mesh`, the object contains a single face that is parameterized using piecewise quadratic interpolation on a triangular mesh. If `Geomrep` is `spline`, there is also a single face, and it is parameterized using a quadratic spline surface. The surface has a continuous normal vector when `Geomrep` is `mesh` or `spline`.

Example

```
% Create randomly generated surface
% Create rectangular grid
[x,y]=meshgrid(-0.1:0.2:1.1,-0.4:0.2:0.4);
% Initialize random generator
randn('state',1);
% Create random height data
z=0.1*randn(size(x));
% Create 3D surface
f=geomsurf(x,y,z);
% Plot the surface
geomplot(f)

% Create approximation to a catenoidal surface
% Create grid in spherical coordinates
[theta,phi]=meshgrid(pi/8:pi/32:3*pi/8,pi/4:pi/32:pi/2);
% The conical surface is expressed in spherical coordinates
r=1;
```

```
x=r.*cos(theta)./sin(phi);
y=r.*sin(theta)./sin(phi);
z=r.*log((1+sin(phi))./sin(phi));
% Now, create the piecewise bilinear
% approximative surface
catenoid=geomsurf(x,y,z);
% Plot the surface
geomplot(catenoid)
axis equal
```

See Also `face3`, `meshgrid`

Purpose Extract parts from an assembly object.

Syntax

```
[gl, ...] = getparts(g, ...)
```

Description `[gl, ...] = getparts(g, ...)` returns a cell array where each element contains a part.

The function supports the following properties:

TABLE 1-82: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Out`	cell array of strings	`{}`	Cell array of strings: `stx`, `ftx`, `ctx`, `ptx`
`Part`	`all` | `none` | vector of integers	`all`	Specifies which parts to extract

Example

```
g = geomgroup({rect2 move(rect2,[1 0])});
[gg,stx,ctx,ptx] = getparts(g,'out',{'stx','ctx','ptx'});
```

See Also `geomgroup`, `geomcsg`

Purpose Create helix geometry object.

Syntax

```
h1 = helix1(r,dh,h)
h2 = helix2(dr,r,dh,h,n)
h3 = helix3(dr,r,dh,h,n)
```

Description `h1 = helix1(r,dh,h)` creates a helix-shaped `curve3` object with radius `r`, distance between consecutive turns `dh`, and total height `h`. The helix is centered at the origin with main axis in the *z* direction. All arguments are optional; when arguments are omitted, the following default values are used: `r` = `1.0`, `dh` = `1.0`, and `h` = `1.0`.

`h2 = helix2(dr,r,dh,h,n)` creates a helix-shaped `face3` object with cross-section radius `dr`, radius `r`, distance between consecutive turns `dh`, total height `h`, and resolution `n`. The resolution `n` is an integer that specifies the number of curved sections for every turn; a higher resolution yields a smoother-looking helix. The helix is centered at the origin with main axis in the *z* direction. All arguments are optional; when arguments are omitted, the following default values are used: `dr` = `0.1`, `r` = `1.0`, `dh` = `1.0`, `h` = `1.0`, and `n` = `12`.

`h3 = helix3(dr,r,dh,h,n)` creates a helix-shaped `solid3` object with cross-section radius `dr`, radius `r`, distance between turns `dh`, total height `h`, and resolution `n`. The resolution `n` is an integer that specifies the number of curved sections for every turn; a higher resolution yields a smoother-looking helix. The helix is centered at the origin with main axis in the *z* direction. All arguments are optional; when arguments are omitted, the following default values are used: `dr` = `0.1`, `r` = `1.0`, `dh` = `1.0`, `h` = `1.0`, and `n` = `12`.

Example The following command generates a solid helix-shaped object:

```
h3 = helix3(1,5,1,5,12);
```

See Also `extrude`, `loft`, `revolve`

Purpose

Create bilinear hexahedron geometry object.

Syntax

```
h2 = hexahedron2(p)
h3 = hexahedron3(p)
```

Description

`h3 = hexahedron3(p)` creates a solid hexahedron object with corners in the 3D coordinates given by the eight columns of `p`. `hexahedron3` is a subclass of `solid3`.

`h2 = hexahedron2(p)` creates a surface hexahedron object with corners in the 3D coordinates given by the eight columns of `p`. `hexahedron3` is a subclass of `face3`.

For a hexahedron approximately aligned to the coordinate planes, the points in `p` are ordered as follows. The first four points and the last four points projected down to the (x, y)-plane defines two negatively oriented quadrangles. The corresponding plane for the second quadrangle must lie above the plane of the first quadrant in the z direction. Generally oriented hexahedra have the points of `p` ordered in a similar way, except for a rigid transformation of the defining point set.

The default value of `p` is

```
p=[0 0 1 1 0 0 1 1;
   0 1 1 0 0 1 1 0;
   0 0 0 0 1 1 1 1]
```

The 3D geometry object properties are available. The properties can be accessed using the syntax `get(object,property)`. See `geom3` for details.

Example

The following command generates a solid hexahedron object.

```
h3 = hexahedron3([0 0   1 1 0 0 1   1;...
                  0 0.8 1 0 0 1 1.2 0;...
                  0 0.1 0 0.2 1 1 2 1]);
```

See Also

`face3`, `geom0, geom1, geom2, geom3`

Purpose Create polygons.

Syntax

```
c = line1(x,y)
s = line2(x,y)
```

Description `s = line2(x,y)` creates a 2D solid object `s` in the form of a solid polygon with vertices given by the vectors `x` and `y`.

`c = line1(x,y)` creates a 2D curve object `c` in the form of an open polygon with vertices given by the vectors `x` and `y`.

Examples The commands below create an open regular *n*-gon (*n*=11) and plot it.

```
n = 11
xy = exp(i*2*pi*linspace(0,1-1/n,n));
l = line1(real(xy),imag(xy));
geomplot(l)
```

See Also `arc1, arc2`, `circ1, circ2`, `ellip1, ellip2`, `geomcsg`, `poly1, poly2`

Purpose Loft 2D geometry sections to 3D geometry.

Syntax `g3 = loft(gl,...)`

Description `g3 = loft(gl,...)` lofts the 2D geometry sections in `gl` to a 3D geometry object `g3`.

`gl` is a cell array of size 1-by-`ng` of 2D geometry objects that belongs to one of the subclasses `solid2` or `curve2`. That is, `gl{i}` contains the geometry object of section number `i`.

The function `loft` accepts the following property/value pairs:

TABLE 1-83: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DESCRIPTION
`LoftEdge`	1-by-ng cell array of integer vectors	Permutation vectors for edges
`LoftSgnEdge`	1-by-ng cell array of integer vectors	Signed permutation vector for edges
`LoftVtxPair`	1-by-ng cell array of integer matrices with two rows	Permutation vector for vertex pairs
`LoftSecPos`	1-by-3 cell array	Positioning for 2D geometry sections
`Wrkpln`	1-by-ng cell array of 3-by-3 matrices	Work planes for 2D geometry sections
`LoftWeights`	Matrix of size 2-by-(ng-1)	Cubic lofting weights
`LoftMethod`	`linear` | `cubic`	Lofting method

The properties `LoftEdge`, `LoftSgnEdge`, or `LoftVtxPair` are needed to make the connection between edges and vertices in different sections unique.

The property `LoftEdge` with the value `{e1,e2,...}` means that the edge with number `e1(1)` in `gl{1}` should be lofted to match the edge with number `e2(1)` in `gl{2}` and so on for all elements in `e1` and `e2`.

Likewise, the property `LoftSgnEdge` with the value `{e1,e2,...}` means the same thing, except that edges with different directions is indicated by using negative signs. This is often more reliable than `LoftEdge` above.

The property `LoftVtxPair` is used in the same way, but uses pairs of vertices instead. Thus, `LoftVtxPair` with the value `{v1,v2,...}` means that the vertex with number `v1(1,1)` in `gl{1}` is to be matched with vertex number `v2(1,1)` in `gl{2}` and so on. It is required that `v1([1 2],1)` are the end points on the same curve.

Only one of these properties is allowed. If, however, none is specified then the property/value-pair `LoftSgnEdge` with the value `{[1:nbs],[1:nbs],...}` is used as default, where `nbs` is the number of edges in `gl{1}`. This means that the edges are considered in order, and is useful for lofting between sections that are simple or similar to each other.

The properties `LoftSecPos` or `Wrkpln` are used to specify the geometrical data of each section.

The property `LoftSecPos` with the value `{D,V,R}` has the following meaning:

`D` is either a 1-by-(ng–1) vector or a 3-by-ng matrix that specifies the position for each geometry section. If `D` is a vector, it contains real numbers that specifies the relative displacement in the local z direction between each pair of consecutive sections in `gl`, where it is assumed that `gl{1}` is positioned at $z = 0$. If `D` is a matrix, then each column specifies the 3D displacements for each of the sections in `gl`. Rows 1, 2, and 3 specifies the displacements in the x, y, and z direction, respectively.

`V` is either a 2-by-ng or a 3-by-ng matrix specifying the tilt-rotations of the geometry objects. If `V` is a 2-by-ng matrix, then each column specifies rotational angles in spherical coordinates. `V(1,:)` are the polar angles, that is, the angles between directional normals of each object and the positive z-axis, and `V(2,:)` are the azimuthal angles of the directional normals. If `V` is a 3-by-ng matrix, then each column specifies a directional normal vector for each section.

`R` is a 1-by-ng vector that specifies the intrinsic rotation of the geometry sections. Every element of `R` is a rotational angle, in radians, with respect to the local z-axis.

The alternative syntax is the property `Wrkpln` with the value `{T1,T2,...}`, where `Ti` is a matrix of size 3-by-3. Here `Ti` is understood to specify the work plane for section `gl{i}`. See `geomgetwrkpln` for more information on work planes.

Only one of these properties is allowed. If none is specified then the property `LoftSecPos` with the value `{ones(1,ng-1),zeros(2,ng),zeros(1,ng)}` is used as default. Moreover, if any of the cells `D`, `V`, or `R` is left empty, the default value is used for that cell.

The property `LoftWeights` specifies the relative significance of the geometry sections with respect to tangential continuity. This argument has no meaning for linear lofting and is then ignored.

The property `LoftMethod` can have the values `linear` or `cubic`, specifying if the lofting should be linear/ruled or bicubic, respectively. The default method is cubic, with the LoftWeights property set to the value `[0.3*ones(1,ng-1); 0.7*ones(1,ng-1)]`.

Examples

Create a loft between a circle and a square.

```
gl1 = cell(1,2);
gl1{1} = circ2;
gl1{2} = rect2(2,2,'pos',[-1 -1]);
```

Let the edge between vertices number one and two in the circle correspond to the edge between vertex number one and two in the square.

```
tl1 = cell(1,2); tl1{1} = [1;2]; tl1{2} = [1;2];
g1 = loft(gl1,'LoftVtxPair',tl1);
figure, geomplot(g1)
```

Create a loft between edge number 1 in the circle and edge number 1 in the square.

```
tl2 = cell(1,2); tl2{1} = 1; tl2{2} = 1;
```

Also, rotate the square.

```
g2 = loft(gl1,'LoftEdge',tl2,...
             'LoftSecPos',{1,zeros(2,2),[0 -pi/4]});
figure, geomplot(g2)
```

Create a more complicated example.

```
gl2 = cell(1,3);
gl2{1} = arc2(0,0,2,0,pi/2)-circ2;
gl2{2} = rect2(2,2,'pos',[-1 -1]);
gl2{3} = gl2{1};
```

Specify reversed direction of edges by using negative signs.

```
tl3 = cell(1,3); tl3{1} = 4; tl3{2} = -2; tl3{3} = -3;
```

Also, rotate the last two sections.

```
g3 = loft(gl2,'LoftSgnEdge',tl3,'LoftSecPos',...
     {[-1 -1 0; 0 0 1; 1 1 2]',zeros(2,3),[0 pi/4 pi]});
figure, geomplot(g3)
```

In the latter case, since there are no ambiguities, you could also use unsigned edge numbers by specifying the property `LoftEdge`.

```
tl4 = cell(1,3); tl4{1} = 4; tl4{2} = 2; tl4{3} = 3;
g4 = loft(gl2,'LoftEdge',tl4,'LoftSecPos',...
      {[-1 -1 0; 0 0 1; 1 1 2]',zeros(2,3),[0 pi/4 pi]});
figure, geomplot(g4)
```

See Also extrude, geomgetwrkpln, revolve

Purpose Create an analyzed geometry and/or a draw object from a (deformed) mesh.

Syntax

```
[xfem,g] = mesh2geom(xfem,args)
```

Description `[xfem,g] = mesh2geom(xfem,args)` returns a new extended fem structure `xfem` with the fields specified in the `destfield` property filled, generated from the source specified in the `srcdata` property. If `draw` is specified in `destfield`, the created draw object `g` is also returned.

The source can be either the deformed geometry from solving an ALE or parameterized geometry problem (`deformed`), or a mesh (`mesh`).

The destination can be any nonempty subset of `{'draw','geom','mesh'}`, indicating that a mesh, an analyzed geometry, and/or a draw object should be created. If `draw` is specified, you can use the `drawtag` property to specify the tag of the new Draw-mode object.

The destination geometry `destfem` can be an existing geometry or the next undefined geometry, in which case a new geometry is created.

TABLE 1-84: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Srcdata`	`deformed` | `mesh`	`mesh`	Source data: deformed geometry or mesh
`Destfield`	cell array of strings: `mesh` | `geom` | `draw`	`{'geom', 'mesh'}`	Destination: mesh, analyzed geometry, or draw object
`Srcfem`	positive integer	1	Source geometry
`Destfem`	positive integer	1	Destination geometry
`MCase`	integer	0	Source and/or destination mesh case
`Drawtag`	string		Draw tag to use when creating draw object
`Frame`	string		Frame to use when retrieving deformed geometry
`Solnum`	positive integer	last solution	Solution to use when retrieving deformed geometry

Examples *Creating an Analyzed Geometry From a Mesh*

Create a mesh.

```
clear fem;
```

```
fem.mesh=meshinit(rect2);
```

Create an analyzed geometry from the mesh, into the new geometry Geom2.

```
xfem=mesh2geom(fem,'destfem',2);
```

Creating an Analyzed Geometry From a Deformed Mesh

Draw two rectangles, one inside the other, and mesh

```
clear fem
g1=rect2(1.8,1.2,'base','corner','pos',[-0.8,-0.8]);
g2=rect2(0.2,0.4,'base','corner','pos',[-0.2,-0.4]);
fem.geom=geomcsg({g1,g2});
fem.mesh=meshinit(fem);
```

Set the inner rectangle to move along the x-axis.

```
clear appl
appl.mode.class = 'MovingMesh';
appl.sdim = {'X','Y','Z'};
appl.assignsuffix = '_ale';
appl.prop.analysis='transient';
appl.prop.weakconstr.value = 'off';
appl.bnd.defflag = {{1;1}};
appl.bnd.deform = {{0;0},{'t';0}};
appl.bnd.ind = [1,1,1,2,2,2,2,1];
appl.equ.type = {'free','pres'};
appl.equ.presexpr = {{0;0},{'t';0}};
appl.equ.ind = [1,2];
fem.appl{1} = appl;
fem.sdim = {{'X','Y'},{'x','y'}};
fem.frame = {'ref','ale'};
fem=multiphysics(fem);
fem.xmesh=meshextend(fem);
```

Solve the problem:

```
fem.sol=femtime(fem,'tlist',[0:0.01:0.1]);
```

Create an analyzed geometry from the deformed mesh.

```
fem=mesh2geom(fem,'srcdata','deformed','frame','ale');
```

Remesh the created geometry and continue solving.

```
fem.mesh=meshinit(fem);
fem.xmesh=meshextend(fem);
fem.sol=femtime(fem,'tlist',[0:0.01:0.1]);
```

Purpose Create boundary layer mesh

Syntax

```
fem.mesh = meshbndlayer(fem,...)
fem.mesh = meshbndlayer(fem.geom,...)
fem = meshbndlayer(fem,'out',{'fem'},...)
```

Description

`fem.mesh = meshbndlayer(fem,...)` returns a boundary layer mesh derived from the geometry in `fem.geom`.

`fem.mesh = meshbndlayer(geom,...)` returns a boundary layer mesh derived from the geometry `geom`.

`fem = meshbndlayer(fem,'Out','fem',...)` modifies the `fem` structure to include a boundary layer mesh in `fem.mesh`.

A boundary layer mesh is a mesh with dense element distribution in the normal direction along specific boundaries. This type of mesh is typically used for fluid flow problems to resolve the thin boundary layers along the no-slip boundaries. In 2D, a layered quadrilateral mesh is used along the specified no-slip boundaries. In 3D, a layered prism mesh or a layered hexahedral mesh is used depending on if the corresponding boundary layer boundaries contain a triangular mesh or a quadrilateral mesh.

The boundary layer mesher inserts boundary layer elements into an existing mesh. If the starting mesh is empty the free mesher is automatically used to create a starting mesh.

Boundary layers are not allowed on isolated boundaries, that is, boundaries with the same subdomain on each side of the boundary.

The function `meshbndlayer` accepts the following property/value pairs.

TABLE 1-85: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Blbnd`	array	all exterior boundaries	Boundary layer boundaries
`Blhmin`	numeric | cell array		Initial boundary layer thickness
`Blhminfact`	numeric | cell array	1	Factor that the default Blhmin is multiplied by
`Blnlayers`	numeric | cell array	8	Number of boundary layers

TABLE 1-85: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Blstretch`	numeric | cell array	`1.2`	Boundary layer stretching factor
`Hauto`	numeric	`5`	Predefined mesh element size
`Mcase`	numeric	`0`	Mesh case number
`Meshstart`	mesh object	empty	Starting mesh
`Out`	`fem` | `mesh`	`mesh`	Output variables
`Report`	`on` | `off`	`on`	Display progress
`Subdomain`	numeric array | `auto` | `all` | `none`	`auto`	Specifies the subdomains that are meshed

`meshbndlayer` accepts all property/values that `meshinit` does. The `meshinit` command is used to create the starting mesh for the subdomains to be processed by the boundary layer mesher if these are not already meshed.

The property `blbnd` is an array specifying the boundaries for which boundary layers are created. By default boundary layers are created for all exterior boundaries.

Use the properties `blhmin`, `blstretch`, and `blnlayers` to specify the distribution of the boundary layers. The value of each of these properties is a scalar value for all boundaries or an even numbered cell array where the odd entries contain boundary indices, either as scalar values, or as vectors with boundary indices, and the even entries contain the corresponding parameters. `blhmin` specifies the thickness of the initial boundary layer, `blstretch` a stretching factor, and `blnlayers` the number of boundary layers. This means that the thickness of the mth boundary layer (m=1 to blnlayers) is blstretch$^{(m-1)}$blhmin. The default value of `blhmin` is 1/50 of the size of the elements for the corresponding boundary layer boundaries. Note that the number of boundary layers and the thickness of the boundary layers might be automatically reduced in thin regions.

It is also possible to specify the thickness of the initial layer by using the `blhminfact` property. If you use this property the initial layer thickness is defined as blhminfact * blhmindef, where blhmindef is the default value of the property `blhmin`. The value of this property is a scalar value for all boundaries or an even numbered cell array where the odd entries contain boundary indices, either as scalar values, or as vectors with boundary indices, and the even entries contain the corresponding parameters.

hauto is an integer between 1 and 9 that controls the element size in the starting mesh. The default value is 5. For more information on this property see meshinit.

The meshstart property is used when meshing a geometry interactively. The value of this property is the starting mesh of the meshing operation. If meshstart does not contain a mesh of the subdomains to be processed a starting mesh is automatically created using the meshinit command before inserting the boundary layer elements.

Use the property subdomain to specify the subdomains to be meshed. If you use this property together with the meshstart property, the value auto means that all subdomains that are not meshed in the starting mesh are meshed, none means that no further subdomains are meshed, and all means that all subdomains are meshed. It is also possible to specify the subdomains to be meshed using a vector of subdomain indices.

Examples

Specify the boundary layer boundaries and the number of boundary layers

```
fem.geom = rect2(10,5) - circ2(1,'pos',[3 2.5]);
fem.mesh = meshbndlayer(fem,'blbnd',[2:3 5:8],...
                        'blnlayers',{5:8 8});
figure, meshplot(fem)
```

Insert boundary layers to an existing mesh containing both quadrilateral elements and triangular elements.

```
fem.geom = rect2 + rect2(1,1,'pos',[1 0]) - circ2(1.5,0.5,0.2);
fem.mesh = meshmap(fem,'subdomain',1);
fem.mesh = meshinit(fem,'meshstart',fem.mesh);
figure, meshplot(fem)
fem.mesh = meshbndlayer(fem,'blbnd',[2:3 5:6 8:11],...
                        'meshstart',fem.mesh,...
                        'subdomain','all');
figure, meshplot(fem)
```

Create a boundary layer mesh consisting of prism elements along the boundary layer boundaries and tetrahedral elements in the interior

```
fem.geom = block3(10,5,5) - sphere3(1,'pos',[3 2.5 2.5]);
fem.mesh = meshbndlayer(fem,'blbnd',[2:13]);
figure, meshplot(fem)
figure, meshplot(fem,'ellogic','x<3')
```

See Also

meshinit, meshmap, meshsweep

Purpose Add new mesh cases

Syntax `fem = meshcaseadd(fem, ...)`

Description `fem = meshcaseadd(fem)` adds one or several new mesh cases to the FEM structure FEM. The mesh cases are typically used as hierarchy in the Geometric multigrid solver. The new mesh cases are constructed by coarsening or refining the mesh (or keeping the same mesh), and possibly changing the order of the shape functions. If the order is changed, the integration point order and the constraint point order is changed accordingly.

The function `meshcaseadd` accepts the following property/value pairs:

TABLE 1-86: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Mcasekeep`	array of nonnegative integers | `all`	`Mcaseorig`	Mesh cases to keep
`Mcaseorig`	array of nonnegative integers	`lowest existing mesh case`	Original mesh case(s)
`Meshscale`	array of positive numbers	`2`	Scale factors for mesh size h
`Mgauto`	`meshscale` | `shape` | `anyshape` | `both` | `meshrefine` | `explicit`		Method for generating mesh cases
`Mggeom`	array of positive integers	`all`	Geometry numbers
`Nmcases`	positive integer	`1`	Number of new mesh cases to generate
`Report`	`on` | `off`	`on`	Display progress
`Rmethod`	`regular` | `longest`	`regular`	Mesh refinement method
`Shapechg`	array of integers	`-1`	Change in shape function orders

The function `meshcaseadd` operates on the FEM structures corresponding to the geometries `Mggeom`. The FEM structures for other geometries are left unaffected.

Before creating new mesh cases, all existing mesh cases except `Mcasekeep` are deleted.

The mesh case generation method is determined by the property `Mgauto`. The new mesh cases will be given the numbers mcmax+1, ..., mcmax+*n*, where mcmax is the current highest mesh case number.

- If `Mgauto=both`, `shape`, `anyshape`, or `meshscale`, then the new mesh cases are constructed starting from the mesh case given in the property `Mcaseorig` (this should be a single nonnegative integer). This process is described in the section "Constructing a Multigrid Hierarchy" on page 560, where the methods are called **Coarse mesh and lower order** (`both`), **Lower element order first (all)** (`shape`), **Lower element order first (any)** (`anyshape`), and **Coarse mesh** (`meshscale`). The mesh coarsening factor is given in the scalar `Meshscale`, the shape function order change amount is given in the scalar `Shapechg`, and the number of new mesh cases to create is given by the property `Nmcases`.
- If `Mgauto=explicit`, then new mesh cases are constructed starting from the mesh case(s) given in the property `Mcaseorig`. The properties `Mcaseorig`, `Meshscale`, and `Shapechg` should be vectors of the same length *n* (however if one is scalar, it is expanded to the same length as the other). Mesh case mcmax+*i* will have a mesh that is coarsened with the factor `Meshscale(i)`, and shape function orders incremented with `Shapechg(i)` relative to mesh case `Mcaseorig(i)`.
- If `Mgauto=meshrefine`, then the new mesh cases are constructed by refining the mesh in mesh case `Mcaseorig` (this should be a single nonnegative integer) repeatedly. The number of new mesh cases to create is given by the property `Nmcases`. The refinement method can be specified using the property `Rmethod`, see `meshrefine`.

The default value of `Mgauto` is as follows. Let *n* be the length of the longest among the vectors `Meshscale`, `Mcaseorig`, and `Shapechg`. The default for `Mgauto` is `shape` if $n = 1$, and `explicit` if $n > 1$.

The following fields in the FEM structure are affected by `meshcaseadd`:

`mesh`, `shape`, `gporder`, `***.gporder`, `cporder`, and `***.gporder`, where `***` is `equ`, `bnd`, `edg`, or `pnt`. Also, the corresponding fields in the `appl` field are affected.

See Also `femsolver`, `meshcasedel`

Purpose Delete mesh cases

Syntax

```
fem = meshcasedel(fem)
fem = meshcasedel(fem,mcases)
```

Description `fem = meshcasedel(fem)` deletes all mesh cases except 0 from the FEM structure `fem`.

`fem = meshcasedel(fem,mcases)` deletes the mesh cases in the integer array `mcases` from the FEM structure `fem`.

The following fields in the FEM structure are affected by `meshcasedel`:

`mesh`, `shape`, `gporder`, `***.gporder`, `cporder`, and `***.gporder`, where `***` is `equ`, `bnd`, `edg`, or `pnt`. Also, the corresponding fields in the `appl` field are affected.

See Also `meshcaseadd`

Purpose Convert mesh to simplex mesh

Syntax

```
fem.mesh = meshconvert(fem,...)
fem.mesh = meshconvert(mesh,...)
```

Description `fem.mesh = meshconvert(fem,...)` converts nonsimplex elements in the mesh object stored in `fem.mesh` to simplex elements using the geometry object stored in `fem.geom`.

`fem.mesh = meshconvert(mesh,...)` converts nonsimplex elements in the mesh object stored in `mesh` to simplex elements.

The function `meshconvert` accepts the following property/value pairs.

TABLE 1-87: VALID PROPERTY/VALUE PAIRS

PROPERTY	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Face`		√	integer array | `all` | `none`	`all`	Faces where the quadrilateral elements are converted to triangular elements
`Out`	√	√	`fem` | `mesh`	`mesh`	Output data structure
`Splitmethod`	√	√	`diagonal` | `center`	`diagonal`	Split method for quadrilateral and hexahedral elements
`Subdomain`	√	√	integer array | `all` | `none`	`all`	Subdomains where the nonsimplex elements are converted to simplex elements

Use the property `subdomain` in 2D and 3D and the property `face` in 3D to specify for which subdomains and faces, respectively, the `meshconvert` function converts nonsimplex elements to simplex elements.

Use the property `splitmethod` to specify how to split quadrilateral and hexahedral elements into triangular and tetrahedral elements, respectively. Use the `diagonal` option to split each quadrilateral element into two triangular elements and each hexahedral element into five tetrahedral element. Use the `center` option to split each quadrilateral element into four triangular elements and each hexahedral element into 28 tetrahedral elements. To be able to use the `center` option in 3D you have to provide a geometry object corresponding to the mesh object. Prism elements are not affected by this property because each prism element is always split into three tetrahedral elements. Note that the conversion also affects quadrilateral elements on the boundaries of the specified subdomains in 3D, which are converted into two triangular elements (when the option `diagonal` is used) or four triangular elements (when the option `center` is used).

Algorithm

Each quadrilateral element that is located either on a face in 3D specified by the property `face`, or in a subdomain in 2D specified by the property `subdomain`, is split into two or four triangular elements.

The property `splitmethod` specifies the technique that is used to convert the quadrilateral elements.

- `Diagonal` means that each quadrilateral element is split along a diagonal into two triangles.
- `Center` means that an extra mesh vertex is placed in the centroid of each quadrilateral, and the element is then split into four triangles.

Each prism element that is located in a subdomain specified by the property `subdomain` is converted into 3 tetrahedral elements. Any hexahedral element adjacent to a prism element is first converted into two prism elements by splitting it along the diagonal of a face. A hexahedral element split in this way is ultimately converted into 6 tetrahedral elements. Note that because the algorithm introduces new prisms by splitting the hexahedron, more hexahedrons might be split in this way.

Each hexahedral element that is located in a subdomain specified by the property `subdomain` is converted into 5 or 28 tetrahedral elements, unless it was split into a prism as described above.

The property `splitmethod` specifies the technique that is used to convert the hexahedral elements.

- `Diagonal` means that each hexahedral element is converted into 5 tetrahedral elements by splitting each face of the element along the diagonal.
- `Center` means that each hexahedral element is converted into 28 tetrahedral elements. Seven extra mesh vertices are added for each hexahedral element, placed in the centroid of each face and in the center of the element.

Note that the quadrilateral elements on the boundaries of the specified subdomains are also converted, either into two triangular elements (when option `diagonal` is used) or into four triangular elements (when option `center` is used). It is therefore necessary to convert all adjacent subdomains at the same time and with the same technique.

Examples

Create a mapped quad mesh on a unit rectangle and convert each quadrilateral element into four triangular elements:

```
fem.geom = rect2;
fem.mesh = meshmap(fem);
fem.mesh = meshconvert(fem, 'splitmethod', 'center');
meshplot(fem);
```

Create a prism mesh and convert each prism into three tetrahedral elements.

```
fem.geom = block3;
fem.mesh = meshinit(fem, 'face', 1, 'subdomain', []);
fem.mesh = meshsweep(fem, 'meshstart', fem.mesh);
fem.mesh = meshconvert(fem);
meshplot(fem);
```

See also

`meshinit`, `meshsweep`, `meshplot`

Purpose Copy mesh between boundaries

Syntax

```
fem.mesh = meshcopy(fem,...)
fem = meshcopy(fem,'out',{'fem'})
```

Description `fem.mesh = meshcopy(fem,...)` copies the mesh between boundaries in the mesh object `fem.mesh`.

`fem = meshcopy(fem,'out',{'fem'})` modifies the FEM structure to include the new mesh object in `fem.mesh`.

Copy the mesh from one or several source boundaries to one target boundary. The source boundary (or in the case of several source boundaries, the combined source boundaries) and the target boundary must be of the exact same shape. However, a scaling factor between the boundaries is allowed.

In 3D, the edges around the source and target boundaries are allowed to be partitioned differently, but only in such a way that several edges of the source boundary map to one edge of the target boundary, not the other way around.

The function `meshcopy` accepts the following property/value pairs:

PROPERTY	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Direction`	√	√	`auto` | `same` | `opposite`	`auto`	Direction between edges
`Mcase`	√	√	integer	0	Mesh case number
`Source`	√	√	integer array		Source boundaries
`Sourceedg`		√	integer array		Source edges
`Target`	√	√	integer		Target boundary
`Targetedg`		√	integer		Target edge

Use the properties `source` and `target` to specify the source and target boundaries. Note that `source` can be either a scalar value or a vector. The property `target` is always a scalar value. This means that copying from several boundaries is allowed, but you can only copy the mesh to a single boundary.

In 3D, use the properties `sourceedg`, `targetedg`, and `direction` to specify the edge mapping from the source to the target boundary. The property `sourceedg` can be either a single edge index or a vector of edge indices. The property `targetedg` is always a single edge index. The property `direction` specifies the direction between `sourceedg` and `targetedg`. The possible values are `same`, `opposite`, and `auto`, where the last option means that the direction between the edges is automatically determined by the algorithm. If `sourceedg` is a vector, then `direction` refers to the direction between `targetedg` and the edge with the lowest edge index in `sourceedg`.

In 2D, use the property `direction` to specify the direction between the edges given in the properties `source` and `target`. The properties `sourceedg` and `targetedg` are not used in 2D.

If you do not specify how to orient the source mesh on the target boundary through the `sourceedg`, `targetedg`, and `direction` properties, the algorithm attempts to determine the orientation automatically.

Copying a mesh is only possible if the target boundary is not adjacent to any meshed subdomain. If the target boundary is already meshed, the current mesh is first deleted and the source mesh is then copied to the target boundary.

In 3D, copying a mesh to a target boundary that is adjacent to a meshed boundary is allowed if the edge between these boundaries has the same number of elements as the corresponding source edges. In this case, the mesh on the target edge is kept, and the copied boundary elements are modified to fit with this edge mesh.

Examples

Mesh Face 1 of a block and copy the mesh to the opposite Face 4.

```
fem.geom = block3;
fem.mesh = meshinit(fem,'point',[],'edge',[], ...
                    'face',1,'subdomain',[]);
fem.mesh = meshcopy(fem,'source',1,'target',4);
```

Mesh Boundaries 1 and 3 of a rectangle and then copy the mesh to Boundary 5.

```
g1 = rect2;
g2 = point2(0,0.3);
fem.geom = geomcsg({g1 g2});
fem.mesh = meshinit(fem,'hnumedg',{1 8 3 4},'point',[], ...
                    'edge',[1,3],'subdomain',[]);
fem.mesh = meshcopy(fem,'source',[1 3],'target',5);
```

See also

`femmesh`, `meshinit`, `meshplot`

Purpose

Delete elements in mesh.

Syntax

```
fem.mesh = meshdel(fem,...)
mesh = meshdel(mesh,...)
fem = meshdel(fem,'Out',{'fem'},...)
```

Description

`fem.mesh = meshdel(fem,...)` deletes elements from the mesh object `fem.mesh` belonging to domains according to the specified properties.

`mesh = meshdel(mesh,...)` deletes elements from the mesh object `mesh`.

`fem = meshdel(fem,'out',{'fem'},...)` modifies the `fem` structure to include the new mesh object in `fem.mesh`.

The function `meshdel` accepts the following property/values:

TABLE 1-88: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Deladj`	`on` | `off`	`on`	Specifies if elements belonging to adjacent domains of lower dimensions are deleted as well
`Edge`	integer array | `all` | `none`	`none`	Specifies the edge domains for which the elements are deleted
`Face`	integer array | `all` | `none`	`none`	Specifies the face domains for which the elements are deleted
`Out`	`fem` | `mesh`	`mesh`	Output variables
`Point`	integer array | `all` | `none`	`none`	Specifies the vertex domains for which the elements are deleted
`Subdomain`	integer array | `all` | `none`	`none`	Specifies the subdomains for which the elements are deleted

Deleting elements corresponding to a specific domain, all elements on adjacent domains of higher dimension are deleted as well.

Examples

Create a mesh of a 2D geometry with 3 subdomains.

```
fem.geom = rect2+circ2;
geomplot(fem,'sublabels','on','edgelabels','on')
fem.mesh = meshinit(fem);
```

Delete the elements belonging to subdomain 3 only.

```
fem.mesh = meshdel(fem.mesh,'subdomain',3,'deladj','off');
figure, meshplot(fem)
```

Delete the elements belonging to subdomain 1 and all adjacent domains of lower dimensions that can be deleted.

```
fem.mesh = meshdel(fem.mesh,'subdomain',1,'deladj','on');
figure, meshplot(fem)
```

Delete the edge elements belonging to edge 1. Note that the elements belonging to the adjacent subdomain (subdomain 2) are deleted as well.

```
fem.mesh = meshdel(fem.mesh,'edge',1);
figure, meshplot(fem)
```

See Also `femmesh`, `meshenrich`, `meshinit`

Purpose Embed a 2D mesh object as a 3D mesh object.

Syntax

```
fem1 = meshembed(fem0,...)
[mesh,geom]= meshembed(fem,'Out',{'mesh','geom'},...)
```

Description `fem1 = meshembed(fem0,...)` embeds the 2D geometry object in `fem0.geom` and the 2D mesh object in `fem0.mesh`, as a 3D geometry object and a 3D mesh object stored in `fem1.geom` and `fem1.mesh`, respectively.

`[geom,mesh]= meshembed(fem,'Out',{'geom','mesh'},...)` returns the embedded 3D geometry object in `geom` and the embedded 3D mesh object in `mesh`.

Valid property/value pairs for the `meshembed` function are given in the following table. In addition, all `embed` parameters are supported and are passed to `embed` for creating the embedded 3D geometry object.

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Mcase`	integer	`0`	Mesh case number
`Out`	`fem` | `mesh` | `geom`	`fem`	Output variables

Embedding a 2D mesh object as a 3D mesh object, the 2D vertex elements, the 2D boundary elements, the 2D triangular elements, and the 2D quadrilateral elements, are embedded as 3D vertex elements, 3D edge elements, 3D triangular boundary elements, and 3D quadrilateral boundary elements, respectively.

See also `embed`, `meshextrude`, `meshrevolve`, `femmesh`

Purpose Make mesh object complete.

Syntax

```
fem.mesh = meshenrich(fem,...)
mesh = meshenrich(mesh,...)
fem = meshenrich(fem,'Out',{'fem'},...)
```

Description `fem.mesh = meshenrich(fem,...)` completes the mesh object `fem.mesh` with element information necessary for using the mesh object in a simulation or for converting into a geometry object.

`mesh = meshenrich(mesh,...)` completes the mesh object `mesh`.

`fem = meshenrich(fem,'out',{'fem'},...)` modifies the `fem` structure to include the new mesh object in `fem.mesh`.

The function `meshenrich` accepts the following property/value pairs.

TABLE 1-89: VALID PROPERTY/VALUE PAIRS

PROPERTY	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Extrangle`		√	numeric	`0.01`	Maximum angle between boundary element normal and extrusion plane that will cause the element to be a part the extruded face if possible
`Faceangle`		√	numeric	`1.8`	Maximum angle between any two boundary elements in the same face
`Facecleanup`		√	numeric	`0.01`	Avoid creating small faces. Faces with an area less than Facecleanup * the mean face area, are merged with adjacent faces
`Facecurv`		√	numeric	`0.2`	Maximum relative curvature deviation between any two boundary elements in the same face
`Faceparam`		√	`on` | `off`	`on`	Specifies if faces are parameterized
`Minareaecurv`		√	numeric	1	Minimum relative area of face to be considered as a face with constant curvature
`Minareaextr`		√	numeric	`0.05`	Minimum relative area of face to be considered extruded

TABLE 1-89: VALID PROPERTY/VALUE PAIRS

PROPERTY	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Minareaeplane`		√	numeric	`0.005`	Minimum relative area of face to be considered planar
`Neighangle`	√	√	numeric	`0.35`	Maximum angle between a boundary element and a neighbor that will cause the elements to be part of the same boundary domain if possible
`Out`	√	√	`fem` | `mesh`	`mesh`	Output variables
`Planarangle`		√	numeric	`0.01`	Maximum angle between boundary element normal and a neighbor that will cause the element to be a part the planar face if possible

Algorithm

These are the main steps of the 3D algorithm:

1. If the domain information (`dom` field) for the subdomain elements is missing, all subdomain elements are assigned the same domain label.
2. Missing boundary elements are added. Boundary elements are required at the boundaries of the subdomains.
3. The up-down subdomain information (`ud` field) for the boundary elements is made complete.
4. If the domain information (`dom` field) for the boundary elements is missing, the face domain partitioning is determined according to the following steps.
   - Search for planar faces according to `Planarangle` and `Minareaeplane`.
   - Search for extruded faces according to `Extrangle` and `Minareaextr`.
   - Search for faces with constant curvature according to `Facecurv` and `Minareacurv`. This search is only done for second order elements.
   - The remaining boundary elements are divided into face domains according to `Neighangle` and `Faceangle`.
5. Exceedingly small faces are merged with neighboring faces according to `Facecleanup`.
6. The faces are parameterized (`param` field).

7 Missing edge elements are added. Edge elements are required at the boundaries of the faces. Domain and parameter information (`dom` and `param` fields) for the edge elements is also added.

8 Missing vertex elements are added. Vertex elements are required on the boundaries of the edges.

The 1D and 2D algorithms work in a similar way.

Example

Create an initial mesh object from mesh point coordinates and tetrahedral element information.

```
load coord.txt;
load tet.txt;
el = cell(1,0);
tet = tet+1; % Lowest mesh point index is zero in tet.txt
el{1} = struct('type','tet','elem',tet);
m = femmesh(coord',el);
```

Use `meshenrich` to create a complete mesh object, that is, a mesh object with boundary elements, edge elements and vertex elements with necessary geometry information.

```
m = meshenrich(m);
meshplot(m)
```

See Also

`geominfo`, `meshinit`, `femmesh`

Purpose Export meshes to file.

Syntax `meshexport(filename,ml,...)`

Description `meshexport(filename,ml,...)` exports the meshes in the cell array `ml` to a file. `ml` can also be one single mesh object.

The function `meshexport` supports the following mesh formats:

FORMAT	FILE EXTENSION
COMSOL Multiphysics text file	`.mphtxt`
COMSOL Multiphysics binary file	`.mphbin`

Example Create a 3D mesh and export in the text file format.

```
m = meshinit(block3+cone3,'hauto',9)
meshexport('meshfile.mphtxt',m);
```

See Also `femmesh`, `meshimport`

Purpose Extend a mesh to the desired finite element types.

Syntax

```
fem.xmesh = meshextend(fem, ...)
[fem.xmesh, cv] = meshextend(fem, ...)
```

Description `fem.xmesh = meshextend(fem)` extends the (possibly extended) FEM structure `fem` with the `xmesh` field. The `xmesh` object contains the full finite element mesh for the model, and also the full description of the model using an internal syntax (the *element syntax*).

`[fem.xmesh, cv] = meshextend(fem)` also outputs a cell array `cv` containing names of variables that were multiply defined.

The function `meshextend` reads the field `fem.solform` and generates the extended using this solution form. The default value for `fem.solform` is `weak` meaning that the weak solution form will be used.

The function `meshextend` accepts the following property/values:.

TABLE 1-90: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Blocksize`	positive integer	`1000`	Block size
`Eqvars`	`on` | `off` | cell array	`on`	Generate equation variables
`Cplbndeq`	`on` | `off` | cell array	`on`	Generate boundary-coupled equation variables
`Cplbndsh`	`on` | `off` | cell array	`off`	Generate boundary-coupled shape variables
`Geoms`	integer vector	All meshed geometries	Extend the mesh on these geometries
`Interiorbnd`	`on` | `off`	`on` if fem.equ.bnd is present	Assemble on interior mesh boundaries
`Linshape`	integer array	All meshed geometries	Use linear geometry shape order for inverted elements on these geometries

TABLE 1-90: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Linshapetol`	scalar or vector	`0.1`	Use linear geometry shape order for inverted elements on these geometries
`Mcase`	integer array	All mesh cases	Extend the mesh for these mesh cases
`Report`	`on` | `off`	`on`	Show progress window
`Standard`	`on` | `off`	`on`	Convert standard syntax to element syntax

Use the properties `linshape` and `linshapetol` to avoid problems with inverted elements in the extended mesh. `linshape` is an integer array specifying the geometries where the software avoids inverted elements by using linear geometry shape order for the corresponding elements. `linshapetol` is the tolerance or a vector of tolerances of the same length as `linshape`. The tolerance values specify the minimum allowed value of the variable `reldetjacmin` for elements not being considered inverted.

Compatibility

If the FEM structure has a version field `fem.version` and the version is older than COMSOL Multiphysics 3.2, then the default for `fem.solform` is equal to `fem.form`.

See Also

`xmeshinfo`

Purpose Extrude a 2D mesh object into a 3D mesh object.

Syntax

```
fem1 = meshextrude(fem0,...)
[mesh,geom]= meshextrude(fem,'Out',{'mesh','geom'},...)
```

Description `fem1 = meshextrude(fem0,...)` extrudes the 2D geometry object in `fem0.geom` and the 2D mesh object in `fem0.mesh`, into a 3D geometry object and a 3D mesh object stored in `fem1.geom` and `fem1.mesh`, respectively, according to the given parameters.

`[geom,mesh]= meshextrude(fem,'Out',{'geom','mesh'},...)` returns the 3D geometry object in `geom` and the 3D mesh object in `mesh`.

Valid property/value pairs for the `meshextrude` function are given in the following table. In addition, all `extrude` parameters are supported and are passed to `extrude` for creating the extruded 3D geometry object.

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`Elextlayers`	1-by-nd cell array		Distribution of mesh element layers in extruded mesh
`Mcase`	integer	`0`	Mesh case number
`Out`	`fem` | `mesh` | `geom`	`fem`	Output variables

The property `Elextlayers` defines the distribution of mesh element layers in the extruded mesh. The value of `Elextlayers` is a cell array where each entry corresponds to a section in the extruded geometry object. If a cell entry is a scalar, it defines the number of equally distributed mesh element layers that is generated for the corresponding extruded section. Alternatively, if a cell entry is a vector, it defines the distribution of the mesh element layers for the corresponding extruded section. The values in the vector, that are sorted and starts with 0, specify the placements, in relative arc length, of the mesh element layers. Note that more element layers might be introduced due to the division of the revolved geometry into sections. By default, the number of element layers is determined such that the distance of each layer is equal to the mean element size in the original 2D mesh.

Extruding a 2D mesh object into a 3D mesh object, the 2D vertex elements, the 2D boundary elements, the 2D triangular elements, and the 2D quadrilateral elements, are extruded into 3D edge elements, 3D quadrilateral boundary elements, 3D prism elements, and 3D hexahedral elements, respectively.

Examples Create an extruded prism mesh on a cylinder of height 1.3.

```
fem.geom = circ2;
fem.mesh = meshinit(fem);
fem1 = meshextrude(fem,'distance',1.3);
```

Create a hexahedral mesh by extruding a quad mesh on a rectangle.

```
fem.geom = rect2(1,2,'pos',[0 0]);
fem.mesh = meshmap(fem);
fem1 = meshextrude(fem,'distance',[1.3 2],...
                   'displ',[0.4 0;0 -0.2],...
                   'scale',[2 1;2 1.5],...
                   'elextlayers',{5 [0 0.2 0.8 1]});
meshplot(fem1);
```

Cautionary Extruding a mesh with any scale factor equal to zero is not supported.

See also `extrude`, `meshembed`, `meshrevolve`, `femmesh`

Purpose Import meshes from file.

Syntax

```
meshes = meshimport(filename,...)
```

Description `meshes = meshimport(filename,...)` reads the file with name `filename` using the specified properties and returns a cell array of meshes.

The function `meshimport` supports the following mesh formats:

FORMAT	FILE EXTENSION
COMSOL Multiphysics text file	`.mphtxt`
COMSOL Multiphysics binary file	`.mphbin`
NASTRAN file	`.nas` | `.bdf` | `.dat`

Valid property/value pairs for the NASTRAN format include.

TABLE 1-91: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`elemsplit`	`on` | `off`	`off`	Specifies if mesh elements of different element forms get different subdomain labels.
`enrichmesh`	`on` | `off`	`on`	Specifies if the imported meshes are enriched.
`linearelem`	`on` | `off`	`off`	Specifies if extended node points are ignored.
`materialsplit`	`on` | `off`	`on`	Specifies if material data in the file is used to determine the domain partitioning of the subdomain elements.
`report`	`on` | `off`	`on`	Determines if a progress window is displayed.

`meshimport` accepts all property/values that `meshenrich` does.

`elemsplit` specifies if mesh elements of different element forms—that is, tetrahedral, pentahedral, or hexahedral—get different subdomain labels. The default value is `off`.

`enrichmesh` specifies if the meshes are enriched with domain information—that is, boundary elements, edge elements, and vertex elements. The domain partitioning is controlled by the properties of `meshenrich`. If the value is `off` the output meshes are not complete meshes. The default value is `on`.

`linearelem` determines if extended node points are ignored. If the value is `on` all imported elements are linear. Otherwise, the order of the imported elements is determined from the order of the elements in the file. The default value is `on`.

`materialsplit` determines if material data in the file is used (if available) to determine the domain partitioning of the subdomain elements. If the value is `off` all subdomain elements in the imported mesh belongs to the same subdomain if possible. The default value is `on`.

`report` specifies if a progress window is displayed. The default value is `on`.

The table below specifies the NASTRAN bulk data entries that are parsed in `meshimport`.

BULK DATA ENTRY
CBAR
CHEXA
CORD1C
CORD1R
CORD1S CORD2C CORD2R CORD2S CPENTA
CQUAD4 CQUAD8
CTETRA
CTRIA3
CTRIA6
GRID

The NASTRAN bulk data format uses reduced second-order elements; that is, the center node on quadrilateral mesh faces (`quadNode`) and the center node of hexahedral elements (`hexNode`) are missing. Importing a NASTRAN mesh with second-order elements, COMSOL Multiphysics interpolates the coordinates of these missing node points from the surrounding node points using the following formulas: `quadNode = 0.5*quadEdgeNodes-0.25*quadCornerNodes`, where `quadEdgeNodes` is the sum of the coordinates of the surrounding 4 edge nodes and `quadCornerNodes` is the sum of the coordinates of the surrounding 4 corner nodes, and `hexNode = 0.25*hexEdgeNodes-0.25*hexCornerNodes`, where `hexEdgeNodes` is the sum of the coordinates of the surrounding 12 edge nodes and `hexCornerNodes` is the sum of the coordinates of the surrounding 8 corner nodes.

Cautionary

`meshimport` does not handle NASTRAN files in free field format where the data fields are separated by blanks.

See Also

`femmesh`, `meshenrich`, `meshexport`

Purpose Create free mesh

Syntax

```
fem.mesh = meshinit(fem,...)
fem.mesh = meshinit(geom,...)
fem = meshinit(fem,'out',{'fem'},...)
```

Description

`fem.mesh = meshinit(fem,...)` returns a mesh object derived from the geometry object `fem.geom`. The mesh size is determined from the shape of the geometry object and various property/value pairs.

`fem.mesh = meshinit(geom,...)` returns a mesh object derived from the geometry object `geom`.

The mesh object `fem.mesh` is the data structure for the mesh. See `femmesh` for a full description of the mesh object.

`fem = meshinit(fem,'Out',{'fem'},...)` modifies the FEM structure to include the mesh object `fem.mesh`.

The function `meshinit` accepts the following property/value pairs:

TABLE 1-92: VALID PROPERTY/VALUE PAIRS

PROPERTY	0D	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`edge`			√	√	numeric array | `auto` | `all` | `none`	`auto`	Specifies the edges that are meshed
`edgelem`		√	√	√	cell array		Edge element distribution
`face`				√	integer array | `auto` | `all` | `none`	`auto`	Specifies the faces that are meshed
`hauto`			√	√	numeric	5	Automatic setting of several mesh parameters
`hcurve`			√		numeric	`0.3`	Curvature mesh size
`hcurve`				√	numeric	`0.6`	Curvature mesh size
`hcurveedg`			√	√	numeric array	`hcurve`	Curvature mesh size for edges
`hcurvefac`				√	numeric array	`hcurve`	Curvature mesh size for faces

TABLE 1-92: VALID PROPERTY/VALUE PAIRS

PROPERTY	0D	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
hcutoff			√		numeric	0.001	Curvature resolution cutoff
hcutoff				√	numeric	0.03	Curvature resolution cutoff
hcutoffedg			√	√	numeric array	hcutoff	Curvature resolution cutoff for edges
hcutofffac				√	numeric array	hcutoff	Curvature resolution cutoff for faces
hgrad		√	√		numeric	1.3	Element growth rate
hgrad				√	numeric	1.5	Element growth rate
hgradvtx		√	√	√	numeric array	hgrad	Element growth rate for vertices
hgradedg			√	√	numeric array	hgrad	Element growth rate for edges
hgradfac				√	numeric array	hgrad	Element growth rate for faces
hgradsub		√	√	√	numeric array	hgrad	Element growth rate for subdomains
hmax		√	√	√	numeric	estimate	Global maximum element size
hmaxvtx		√	√	√	numeric array	hmax	Maximum element for vertices
hmaxedg			√	√	numeric array	hmax	Maximum element for edges
hmaxfac				√	numeric array	hmax	Maximum element for faces
hmaxsub		√	√	√	numeric array	hmax	Maximum element for subdomains
hmaxfact		√	√	√	numeric	1	A factor that the default hmax is multiplied by
hmesh		√	√	√	numeric		Element size given on mesh

TABLE 1-92: VALID PROPERTY/VALUE PAIRS

PROPERTY	0D	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`hnarrow`			√		numeric	`1`	Resolution of narrow regions
`hnarrow`				√	numeric	`0.5`	Resolution of narrow regions
`hnumedg`			√	√	cell array		Number of elements for edges
`hnumsub`		√			cell array		Number of elements for subdomains
`hpnt`			√		numeric	`10`	Global number of resolution points
`hpnt`				√	numeric	`20`	Global number of resolution points
`hpntedg`			√		numeric array	`Hpnt`	Number of resolution points for edges
`hpntfac`				√	numeric array	`Hpnt`	Number of resolution points for faces
`jiggle`			√	√	`on` | `off`	`on`	Improve mesh quality
`mcase`	√	√	√	√	integer	`0`	Mesh case number
`mesh`		√	√	√	mesh object		Mesh for `hmesh`
`meshstart`	√	√	√	√	mesh object		Starting mesh
`methodsub`			√		cell array | `tri` | `triaf` | `quad`	`triaf`	Specify triangle (Delaunay), triangle (advancing front), or quad mesh
`minit`			√	√	`on` | `off`	`off`	Boundary triangulation
`mlevel`		√			`vtx` | `sub`	`sub`	Meshing level
`mlevel`			√		`vtx` | `edg` | `sub`	`sub`	Meshing level
`mlevel`				√	`vtx` | `edg` | `fac` | `sub`	`sub`	Meshing level

TABLE 1-92: VALID PROPERTY/VALUE PAIRS

PROPERTY	0D	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`point`		√	√	√	integer array | `auto` | `all` | `none`	`auto`	Specifies the vertices that are meshed
`out`	√	√	√	√	`fem` | `mesh` | `p` | `e` | `t` | `vg` | `eg` | `vg`	`mesh`	Output variables
`report`	√	√	√	√	`on` | `off`	`on`	Display progress
`subdomain`	√	√	√	√	integer array | `auto` | `all` | `none`	`auto`	Specifies the subdomains that are meshed
`xscale`			√	√	numeric	1	Scale geometry in x direction before meshing
`yscale`			√	√	numeric	1	Scale geometry in y direction before meshing
`zscale`				√	numeric	1	Scale geometry in z direction before meshing

Use the properties `point`, `edge`, `face`, and `subdomain` to specify the domains to be meshed. If you use these properties together with the `meshstart` property, the value `auto` means that all domains that are not meshed in the starting mesh are meshed and `none` that no further domains are meshed. `all` means that all domains not already meshed in the starting mesh and all meshed domains that are not adjacent to a meshed domain of higher dimension are meshed (or remeshed). It is also possible to specify the domains to be meshed (or remeshed) using a vector of domain indices.

The `meshstart` property is used when meshing a geometry interactively. The value of this property is the starting mesh of the meshing operation.

Use the property `methodsub` in 2D to specify if subdomains should be meshed with triangles, using either a Delaunay based method or an advancing front based method, or quads. The property is set to `tri`, `triaf`, or `quad`, respectively. The default option is `triaf`. In general, the advancing front algorithm produces a mesh of higher quality.

The value of the property can also be an even numbered cell array, where the odd entries contain subdomain indices, either as scalar values, or as vectors with subdomain indices, and the even entries are either `quad`, `tri`, or `triaf`, specifying which method to use for each subdomain. Default is `tri` for all subdomains.

Use the property `methodfac` in 3D to specify if faces should be meshed with triangles, using either a Delaunay based method or an advancing front based method, or quads. The property `methodfac` in 3D is equivalent to the property `methodsub` in 2D, and is specified in the same way, but for faces instead of subdomains.

The property `edgelem` is used to explicitly control the distribution of the edge elements in the mesh. The value of this property is an even numbered cell array where the odd entries contain edge indices, either as scalar values, or as a vectors with edge indices, and the even entries contain scalar values or vectors specifying the edge element distribution on the corresponding edge(s). If the edge element distribution is specified by a scalar, the edge elements on the corresponding edge(s) are equally distributed in arc length and the number of edge elements equals the value of the scalar. To get full control over the edge element distribution on an edge, the vector form is used. The values in the vector, that are sorted and starts with 0, specify the relative placement of the mesh vertices along the direction of the corresponding edge(s). The `edgelem` property can also be used on subdomains in 1D to control the element distribution.

The `minit` property is related to the way the mesh algorithm works. By turning on `minit` you can see the initial discretization of the boundaries. This property is only valid for `mlevel sub`.

`Hauto` is available in 2D and 3D and is an integer between 1 and 9. This integer is used to set several mesh parameters in order to get a mesh of desired size. Smaller values of `hauto` generate finer meshes with more elements.

TABLE 1-93: MESH PARAMETERS SET BY THE PROPERTY HAUTO IN 2D

HAUTO	HMAXFACT	HCURVE	HGRAD	HCUTOFF
1	0.15	0.2	1.1	0.0001
2	0.3	0.25	1.2	0.0003
3	0.55	0.25	1.25	0.0005
4	0.8	0.3	1.3	0.001

TABLE 1-93: MESH PARAMETERS SET BY THE PROPERTY HAUTO IN 2D

HAUTO	HMAXFACT	HCURVE	HGRAD	HCUTOFF
5	1	0.3	1.3	0.001
6	1.5	0.4	1.4	0.005
7	1.9	0.6	1.5	0.01
8	3	0.8	1.8	0.02
9	5	1	2	0.05

TABLE 1-94: MESH PARAMETERS SET BY THE PROPERTY HAUTO IN 3D

HAUTO	HMAXFACT	HCURVE	HGRAD	HCUTOFF	HNARROW
1	0.2	0.2	1.3	0.001	1
2	0.35	0.3	1.35	0.005	0.85
3	0.55	0.4	1.4	0.01	0.7
4	0.8	0.5	1.45	0.02	0.6
5	1	0.6	1.5	0.03	0.5
6	1.5	0.7	1.6	0.04	0.4
7	1.9	0.8	1.7	0.05	0.3
8	3	0.9	1.85	0.06	0.2
9	5	1	2	0.07	0.1

`Hcurve` is a scalar numeric value that relates the mesh size to the curvature of the geometry boundaries. The radius of curvature is multiplied by the `hcurve` factor to obtain the mesh size along the boundary.

`hcurveedg` and `hcurvefac` are matrices with two rows where the first row contains edge indices and face indices respectively, and the second row contains corresponding values of `hcurve`. If several faces are represented in one patch, the value of `hcurvefac` for the faces in the patch is set to the minimum value of the `hcurvefac` values for the corresponding faces.

`hcutoff` is used to prevent the generation of many elements around small curved parts of the geometry. The interpretation is that when the radius of curvature is smaller than `hcutoff*maxdist` the radius of curvature is taken as `hcutoff*maxdist`, where `maxdist` is the longest axis parallel distance in the geometry.

`hcutoffedg` and `hcutofffac` are matrices with two rows where the first row contains edge indices and face indices respectively, and the second row contains corresponding values of `hcutoff`. If several faces are represented in one patch, the value of hcutofffac for the faces in the patch is set to the minimum value of the hcutofffac values for the corresponding faces.

The property hgrad tells how fast the element size—measured as the length of the longest edge of the element—can grow from a region with small elements to a region with larger elements. If two elements lie one unit length apart, the difference in element size can be at most hgrad.

hgradvtx, hgradedg, hgradfac, and hgradsub are matrices with two rows where the first row contains vertex indices, edge indices, face indices, and subdomain indices respectively, and the second row contains corresponding values of hgrad. If several faces are represented in one patch, the value of hgradfac for the faces in the patch is set to the minimum value of the hgradfac values for the corresponding faces.

The hmax parameter controls the size of the elements in the mesh. meshinit creates a mesh where no element size exceeds hmax. The default hmax value is one fifteenth of the longest axis parallel distance in the geometry in 1D and 2D and one tenth of the longest axis parallel distance in the geometry in 3D.

hmaxvtx, hmaxedg, hmaxfac, and hmaxsub are matrices with two rows where the first row contains vertex indices, edge indices, face indices, and subdomain indices respectively, and the second row contains corresponding values of hmax.

The hmaxfact property is used to scale the defaulted hmax value.

hmesh is a vector with one entry for every mesh vertex or element in the mesh given in the mesh property. This can be used to specify the size of the elements using the mesh provided with the property mesh.

The hnarrow parameter controls the size of the elements in narrow regions. Increasing values of this property decrease the size of the elements in narrow regions. If the value of hnarrow is less than one, elements that are anisotropic in size might be generated in narrow regions.

hnumedg and hnumsub are cell arrays where the odd entries contain edge indices and subdomain indices respectively, and even entries contain number of elements.

The hpnt property controls the number of points that are placed on each edge in 2D and in each parametric direction on each geometry patch in 3D to resolve the geometry.

hpntedg is a matrix with two rows where the first row contains edge indices and the second row contains corresponding values of hpnt.

hpntfac is a matrix with two rows where the first row contains face indices and the second row contains corresponding number of resolution points in each parametric direction of the underlying geometry patch. If several faces are represented in one patch the value of hpntfac for the faces in the patch is set to the maximum value of the hpntfac values for the corresponding faces.

The jiggle property determines if the quality of the mesh is improved before the mesh is returned.

Use the mesh property to specify a mesh for the property hmesh.

The property mlevel determines to which level the mesh is generated. If it is vtx, only the vertices in the geometry are returned. If it is edg, the edges in the geometry are resolved. If it is fac the edges and faces in the geometry are resolved. If it is sub elements in the subdomains are generated as well.

The properties xscale, yscale, and zscale specify scalar factors in each axis direction that the geometry is scaled by before meshing. The resulting mesh is then scaled back to fit the original geometry. The values of other properties correspond to the scaled geometry. By default, no scaling is done.

Examples

3D Example

Create a 3D mesh of a cylinder.

```
clear fem
fem.geom=cylinder3;
fem.mesh=meshinit(fem);
meshplot(fem)
```

Use advancing front meshing on the boundary.

```
fem.mesh = meshinit(fem,'methodfac','triaf');
meshplot(fem)
```

2D Example

Make a simple mesh of a unit square.

```
clear fem
fem.geom = geomcsg({square2(0,1,1)});
fem.mesh = meshinit(fem);
meshplot(fem), axis equal
```

Make the mesh finer than the default.

```
fem.mesh = meshinit(fem,'hmax',0.02);
meshplot(fem), axis equal
```

Now, make the mesh denser only near the edge segment to the left.

```
fem.mesh = meshinit(fem,'hmaxedg',[1; 0.02]);
meshplot(fem), axis equal
```

Use instead the advancing front method to create the mesh.

```
fem.mesh = meshinit(fem,'methodsub','triaf');
meshplot(fem), axis equal
```

Make a free quad mesh of a circle

```
clear fem
fem.geom = geomcsg({circ2});
fem.mesh = meshinit(fem,'methodsub','quad');
meshplot(fem), axis equal
```

1D Example

Create a mesh on the interval [0,1] that is finer near the point 0 and grows toward 1.

```
fem.geom = geomcsg({solid1([0 1])});
fem.mesh = meshinit(fem,'hmax',0.1,'hmaxvtx',[1; 0.001]);
meshplot(fem)
```

2D Example Dealing with Interactive Meshing

Create a boundary mesh of a geometry

```
clear fem;
fem.geom = rect2+circ2;
fem.mesh = meshinit(fem,'subdomain','none');
meshplot(fem)
```

Mesh subdomain 2 using the boundary mesh as starting mesh

```
fem.mesh = meshinit(fem,'subdomain',2,'meshstart',fem.mesh);
meshplot(fem)
```

Mesh the remaining subdomains using the previous mesh as starting mesh

```
fem.mesh = meshinit(fem,'meshstart',fem.mesh);
meshplot(fem)
```

Compatibility

The second and third row in the `vg` field as well as the second row in the `v` field will be removed in future versions.

Cautionary

To achieve compatibility with FEMLAB 2.3, the geometry input is automatically converted to a geometry object using the function `geomobject`. The geometry input can be any analyzed geometry. See `geomobject` for details.

If you create a mesh with `methodsub` set to `quad` in 2D, or `methodfac` set to `quad` in 3D, the generated mesh is not guaranteed to contain only quadrilateral elements. If the algorithm for some reason fails to mesh the entire domain with quad elements, or if the quality of a quad element is very low, some triangular elements are generated instead.

See Also `femmesh`, `geomcsg`, `meshplot`, `meshrefine`

Reference George, P. L., *Automatic Mesh Generation—Application to Finite Element Methods*, Wiley, 1991.

Purpose Integrate over arbitrary cross section

Syntax

```
I = meshintegrate(p,t,d)
I = meshintegrate(p,d)
I = meshintegrate(p)
```

Description `I = meshintegrate(p,t,d)` computes the integral `I` over the mesh given by `p` and `t`, with values (for each point) in `d`. `d` is of size `1`-by-`np`, where `np` is the number of points in `p` (`=size(p,2)`). The elements are considered to be linear.

`I = meshintegrate(p,d)` assumes `t=[1,2,3,... (np-1) ; 2,3,4,... np]`, (`where np=size(p,2)`), i.e., that the mesh is a line and that the points in `p` are sorted.

`I = meshintegrate(p)` calls `meshintegrate(p(1,:), p(2,:))`.

This function is useful for computing integrals along cross sections plotted with `postcrossplot`, in which case `p`, `t`, and `d` are extracted from the output when the property `outtype` is set to `postdata`.

Examples Line integral in 2D:

```
% Just set up a problem:
clear fem
fem.geom = circ2+rect2;
fem.mesh = meshinit(fem);
fem.shape = 2; fem.equ.c = 1; fem.equ.f = 1; fem.bnd.h = 1;
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem);

% Make a cross-section plot, with output being a postdata
% structure
pd = postcrossplot(fem,1,[0 1;0 1],'lindata','u',...
                   'npoints',100,'outtype','postdata');

% Call meshintegrate:
I = meshintegrate(pd.p);
```

Line integral in 3D:

```
% Just set up a problem:
clear fem, fem.geom = block3;
fem.mesh = meshinit(fem,'hmax',0.15);
fem.shape = 2;
fem.equ.c = 1; fem.equ.f = 1;
fem.bnd.h = {1 1 0 0 1 1};
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem);
```

```
% Make cross-section plot:
pd = postcrossplot(fem,1,[0 1;0 1;0 1],'lindata','u',...
                   'npoints',100,'outtype','postdata');

% Call meshintegrate:
I = meshintegrate(pd.p)
```

Surface integral in 3D using the same problem as above:

```
pd = postcrossplot(fem,2,[0 0 0;0 1 0;1 0 1]','surfdata','u',...
                   'outtype','postdata');
I = meshintegrate(pd.p, pd.t, pd.d)
```

This function only works for lines and surfaces actually intersecting the geometry. For plots along geometry boundaries or edges (or 1-D subdomains), better results are achieved using `postint`.

Cautionary This function is not implemented for 3-D elements, i.e., when T has four rows.

See also `postcrossplot`, `postint`

Purpose

Create mapped quad mesh.

Syntax

```
fem.mesh = meshmap(fem,...)
fem.mesh = meshmap(geom,...)
fem = meshmap(fem,'out',{'fem'},...)
```

Description

`fem.mesh = meshmap(fem,...)` returns a mapped quad mesh derived from the geometry `fem.geom`. For a 3D geometry, only the faces are meshed.

`fem.mesh = meshmap(geom,...)` returns a mapped quad mesh derived from the geometry `geom`.

`fem = meshmap(fem,'out',{'fem'},...)` modifies the `fem` structure to include a mesh in `fem.mesh`.

The quad mesh is generated by a mapping technique. For each subdomain in 2D and face in 3D, a logical mesh is generated on a square geometry and is then mapped onto the real geometry by transfinite interpolation.

The following criteria must be met by the input geometry object for the mapping technique to work:

- Each subdomain/face to be meshed must be bounded by one connected boundary component only.
- Each subdomain/face to be meshed must be bounded by at least four edges.
- The subdomains/faces to be meshed must not contain isolated vertices or edges.
- The shape of each subdomain/face to be meshed must not differ too much from rectangular shape.

The function `meshmap` accepts the following property/values:

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`edgegroups`	cell array of size 1-by-ns		Determines the grouping of the edges, per subdomain/face, into four edge groups, corresponding to the edges of the logical square
`edgelem`	cell array		Edge element distribution
`face`	integer array | `auto` | `all` | `none`	`auto`	Specifies the faces in 3D that are meshed
`hauto`	numeric	5	Predefined mesh element size
`mcase`	integer	0	Mesh case number

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`meshstart`	mesh object	empty	Starting mesh
`report`	`on` | `off`	`on`	Display progress
`out`	`fem` | `mesh`	`mesh`	Output variables
`subdomain`	integer array | `auto` | `all` | `none`	`auto`	Specifies the subdomains in 2D that are meshed

The property `edgegroups` is a cell array where each cell entry, corresponding to each subdomain/face, determines the relation between the edges defining the boundary of the corresponding subdomain/face, and the four edges of the logical square. If a cell entry is left empty, the meshing algorithm splits the edges bounding the subdomain/face into the four edge groups at the vertices corresponding to the four sharpest corners. The relation between the edges of each subdomain/face and the edges of the logical square is specified as a cell array, where each cell entry contains the indices to the edges in the real geometry that correspond to one edge of the logical square.

The property `edgelem` determines the distribution of the edge elements in the mesh. The value of this property is an even numbered cell array where the odd entries contain edge indices, either as scalar values, or as a vectors with edge indices, and the even entries contain scalar values or vectors specifying the edge element distribution on the corresponding edge(s). If the edge element distribution is specified by a scalar, the edge elements on the corresponding edge(s) are equally distributed in arc length and the number of edge elements equals the value of the scalar. To get full control over the edge element distribution on an edge, the vector form is used. The values in the vector, that are sorted and starts with 0, specify the placements, in arc length, of the mesh vertices along the direction of the corresponding edge(s).

`hauto` is an integer between 1 and 9 that controls the element size in the generated mesh. The default value is 5 which means that the element size is set to 1/15 in 2D and 1/10 in 3D of the size of the geometry for the elements not affected by the `edgelem` property. By changing the value of this property, the default element size is multiplied by the following factors.

HAUTO	SCALE FACTOR
1	0.2
2	0.35

HAUTO	SCALE FACTOR
3	0.55
4	0.8
5	1
6	1.5
7	1.9
8	3
9	5

Use the property `subdomain` in 2D and `face` in 3D to specify the subdomains/faces to be meshed. If you use this property together with the `meshstart` property, the value `auto` means that all subdomains/faces that are not meshed in the starting mesh are meshed, `none` means that no further subdomains/faces are meshed, and `all` means that all subdomains/faces are meshed (or remeshed). It is also possible to specify the subdomains/faces to be meshed (or remeshed) using a vector of subdomain/face indices.

The `meshstart` property is used when meshing a geometry interactively. The value of this property is the starting mesh of the meshing operation.

Note that for the mapping technique to work, opposite edges require the same number of edge elements. If this requirement is not met by the specified values in `edgelem`, an error is generated.

Examples

Create a mapped quad mesh on a geometry where all subdomains are topologically equivalent with a rectangle.

```
clear fem
fem.geom = rect2(2,0.98)+rect2(2,0.04,'pos',[0 0.98])+...
           rect2(2,0.98,'pos',[0 1.02]);
fem.mesh = meshmap(fem,'edgelem',{[2 7] 12 [1 8] ...
                                  [0 0.2:0.2:0.8 ...
                                   0.86:0.04:0.98]});
figure, meshplot(fem);
```

Create a mapped quad mesh on a geometry with two subdomains.

```
fem.geom = ...
   rect2+rect2(1,1,'pos',[1 0])-circ2(0.4,'pos',[1.1 -0.1]);
figure, geomplot(fem,'edgelabels','on')
fem.mesh = meshmap(fem,'edgegroups',{{1 3 2 [4 8]},...
                                      {4 5 7 [9 10 6]}});
figure, meshplot(fem);
```

Create a mesh with both triangle and quad elements

```
fem.geom = geomcomp({circ2(0.5,'pos',[0 0.5]),rect2,...
                    circ2(0.5,'pos',[1 0.5])},'edge',7:10);
figure, geomplot(fem)
fem.mesh = meshmap(fem,'subdomain',2,'hauto',3);
fem.mesh = meshinit(fem,'meshstart',fem.mesh,'hauto',3);
figure, meshplot(fem);
```

See also meshdel, meshinit, meshplot

Purpose

Plot mesh.

Syntax

```
meshplot(fem,...)
meshplot(mesh,...)
h = meshplot(...)
```

Description

`meshplot(fem,...)` plots the mesh object `fem.mesh`.

`meshplot(mesh,...)` plots the mesh object `mesh`.

`h = meshplot(...)` additionally returns handles to the plotted axes objects.

The mesh of the PDE problem is specified by the mesh object. Details on the representation of the mesh can be found in the entry `femmesh`.

There is a multitude of options that enables you to plot the mesh in virtually any conceivable way. For 2D and 3D meshes, there are basically two types of mesh plots; the *wireframe plot* and the *patch plot.*

In 3D, the default type is a *patch plot*, where both the triangular faces of the elements and the boundary elements are rendered. Only the visible faces of the elements are included in the plot, and the boundary elements and other mesh faces are plotted in different colors. The characteristics of this plot type are controlled by the properties that start with `edge` and `bound`. The `edge` properties control the plot of the element faces, and the `bound` properties control the boundary element plot (the name `edge` has historic reasons; it was first used in 2D). The color of the edges of the element faces is determined by the property `eledgecolor`.

The alternative type of 3D plot is a *wireframe plot*, consisting of the (1D) edges of the elements and the boundary elements. The properties that start with `dedge` and `dbound` control the plot characteristics of this plot type. The `dedge` properties control the plot of the element edges, and the `dbound` properties control the plot of the edges of the boundary elements. See the 3D example below for how to obtain a patch plot and a wireframe plot of a mesh. You can also plot the mesh edges that lie on geometry edges (curves) with a special color. The characteristics of this plot is controlled by the properties that begin with `curve`.

In 2D, there are basically two types of mesh plots. The default type is a *wireframe plot* of the edges of the elements, where the boundary elements are plotted in a different color. The properties that start with `edge` and `bound` control the plot characteristics of this plot type. The `edge` properties control the plot of the element edges, and the `bound` properties control the boundary element plot.

The alternative type of 2D plot is a *patch plot* of the triangular elements, where the edges of the elements can have a different color. The properties that start with `el` control the characteristics of this plot type. See the 2D example below for how to obtain the two plot types. The two plot types can be combined, but doing this is not always useful.

In 1D, the default is to combine the above two types of plots into one plot type. Thus, by default you can see both the elements, the boundary elements, and the intermediate mesh vertices. The plot of the elements is controlled by the properties that begin with `el`, and the plot of the boundary elements is controlled by the `bound` properties. The plot of the mesh vertices is controlled by the `edge` properties.

In all dimensions, you may plot the mesh vertices (sometimes called node points) in a special color. This plot is controlled by the properties that start with `node`.

The table shows the valid property/value pairs for the `meshplot` command. The design philosophy has been to keep property interpretation constant over space dimension, but to plot these properties as plot objects of different types. The mode properties that turn the different visualization types on and off, have been marked with the type of the plot produced in the different space dimensions: `m` for marker plot, `l` for line/wireframe plot, and `p` for patch plot.

TABLE 1-95: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`bdl`		√	√	integer array	all	Boundary list
`boundcolor`	√	√		color	`r`	Boundary color
`boundcolor`			√	color | `qual`	`r`	Boundary color
`boundmarker`	√			marker	`o`	Boundary marker
`boundmode`	m	l	p	`on` | `off` | `isolated` (1D)	`on` `off` (3D)	Show boundary elements
`curvecolor`			√	color	`g`	Curve (edge) coloring data
`curvemode`			l	`on` | `off`	`off`	Show mesh edges on curves (edges)
`dboundcolor`			√	color	`r`	Boundary wireframe color
`dboundmode`			l	`on` | `off`	`off`	Show boundary elements as wireframe

TABLE 1-95: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
dedgecolor			√	color	b	Element wireframe color
dedgemode			l	on | off	off	Show elements as wireframe
edgecolor	√			color	b	Color of mesh vertices
edgecolor		√		color	b	Color of mesh edges
edgecolor			√	color | qual	b	Color of mesh faces
edgemarker	√			marker	x	Mesh vertex marker
edgemode	m			on | off	on	Show mesh vertices
edgemode		l		on | off	on	Show wireframe plot of mesh edges
edgemode			p	on | off	on	Show triangular mesh faces as patch plot
edl			√	integer array	all	Edge list
elcolor	√			color	k	Element color
elcolor		√		color	gray	Element color
eledgecolor		√	√	color	k	Mesh edge color in patch plot
elkeep		√	√	number between 0 and 1	1	Fraction of elements to keep
elkeeptype		√	√	min | max | random	random	Which elements to keep
ellabels	√	√		on | off	off	Mesh element labels
ellogic			√	logical expression	1	Select elements using a logical expression
ellogictype			√	all | any | xor	all	Interpretation of the logical expression
elmode	l			on | off	on	Show elements
elmode		p		on | off	off	Show elements
markersize	√	√	√	scalar	6	Marker size
nodecolor	√	√	√	color	k	Mesh vertex (node) color

TABLE 1-95: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`nodelabels`	√	√	√	`on` | `off`	`off`	Mesh vertex (node) labels
`nodemarker`	√	√	√	marker	`.`	Mesh vertex (node) marker
`nodemode`	m	m	m	`on` | `off`	See below	Show mesh vertices (node points)
`parent`	√	√	√	axes handle		Handle to axes object
`pointcolor`	√	√	√	color	`b`	Point color data
`pointlabels`	√	√	√	`off` | `on` | list of strings	`off`	Point label list
`pointmarker`	√	√	√	marker symbol	`o`	Point marker
`pointmode`	√	√	√	`on` | `off` | `isolated`	`on`	Show points
`qualbar`		√	√	`on` | `off`	`on`	Show color legend
`qualdlim`		√	√	1-by-2 numeric	`[0,1]`	Color limits
`qualmap`			√	color table	`Rainbow`	Color table
`qualmapstyle`			√	`auto` | `reverse`	`auto`	Color table style
`sdl`		√	√	integer array	all	Subdomain list

In 3D, the properties `Sdl`, `Bdl`, `Elkeep`, `Elkeeptype`, `Ellogic`, and `Ellogictype` determine what part of the mesh is displayed. Only elements lying in subdomains in the list `Sdl`, and boundary elements lying on boundaries in the list `Bdl`, are shown. The set of elements is restricted further by the properties `Elkeep`, `Elkeeptype`, `Ellogic`, and `Ellogictype`. These affect the patch and wireframe plots of the elements (controlled by the `Edge` and `Dedge` properties). Only the mesh elements whose corners satisfy the logical expression `Ellogic` are shown. `Ellogictype` determines whether all, some, or some but not all of the corners are required to satisfy the condition. Only a fraction `Elkeep` of the mesh elements are shown. The property `Elkeeptype` determines whether this fraction is the elements of worst quality, or if it is a random set of elements.

The `El` property group controls the display of the actual element. This is only possible in 1D and 2D. The `Edge` property group controls the display of the boundaries of the elements. This visualization type is available in all space dimensions. In 1D it is done by displaying marker symbols, in 2D it is done by displaying a wireframe plot of the element edges, and in 3D it is done by displaying the element edges as patches. In 3D, the property `dedge` makes it possible to obtain wireframe plots of the elements.

The `bound` property group controls the display of the boundary elements. In 1D it displays marker symbols, in 2D it displays the boundary elements as a wireframe, and in 3D it displays the boundary elements as a patch plot. In 3D, the property `dbound` makes it possible to get wireframe plots of the boundary elements.

In 3D, the `curve` property group controls the display of mesh elements on geometry edges.

The `point` property group displays the mesh vertex elements as marker symbols in all space dimensions.

The `node` property group displays the mesh vertices (node points) as marker symbols in all space dimensions.

The `ellabels` property is available in 1D and 2D and controls the display of mesh element labels. If there are more than one type of mesh elements, for example both triangular and quadrilateral elements in a 2D mesh, the different types are labeled individually. This means that the triangles will be labeled from 1 to the number of triangles, and the quads will be labeled from 1 to the number of quads.

The properties that control marker type or coloring can handle any standard scripting marker or color type (see the `plot` command). In 3D, the patch coloring can be made according to the element quality, by specifying the color as `'qual'`.

`meshplot` can display meshes where there are no elements on a certain element dimension. In these cases, the default values for `curvemode` and `nodemode` change to make the best possible mesh visualization.

Examples

3D Example

Start by creating a 3D geometry and a mesh.

```
c1 = cylinder3(0.2,1,[0.5,0.5,0]);
b1 = block3;
geom = b1-c1;
mesh = meshinit(geom);
```

Plot the mesh as a quality patch plot with parts of the elements excluded by a logical expression. These types of options make it easy to study the mesh inside the geometry.

```
meshplot(mesh,'ellogic','x+y>0.8',...
         'edgecolor','qual','boundcolor',[0.7 0.7 0.7],...
```

```
          'qualbar','on')
```

You can get a wireframe plot of the same mesh with only a fraction of the tetrahedra visible, by the command

```
meshplot(mesh,'ellogic','x+y>0.8','elkeep',1/100,...
         'edgemode','off','boundmode','off',...
         'dedgemode','on','dboundmode','on','curvemode','on')
```

The plot shows only a small fraction of the elements. It is not possible to get a mesh quality plot by using only wireframes.

2D Example

Start by creating the geometry and a coarse mesh.

```
clear fem
sq1 = square2;
sq2 = move(sq1,0,-1);
sq3 = move(sq1,-1,-1);
fem.geom = sq1+sq2+sq3;
fem.mesh = meshinit(fem,'hmax',0.4);
```

Then plot the mesh as a line plot of the edges of the elements. The element edges are blue except on the boundary elements, where they are red. This is the default mesh plot type.

```
meshplot(fem)
```

You can change the colors of the element edges to yellow and green.

```
meshplot(fem,'edgecolor','y','boundcolor','g')
```

Now, plot the mesh as a patch plot. You need to disable both the element edge and boundary element plots, and enable the element plot.

```
meshplot(fem,'edgemode','off','boundmode','off','elmode','on')
```

You can change the color of the elements to red, with white edges, and add mesh vertex labels by

```
meshplot(fem,'edgemode','off','boundmode','off','elmode',...
         'on','elcolor','r','eledgecolor','w')
```

1D Example

Start by creating the geometry and mesh.

```
clear fem
s1 = solid1([0 0.1 4]);
s2 = solid1([3 4]);
fem.geom = s1+s2;
fem.mesh = meshinit(fem,'hmax',0.4);
```

The standard mesh plot in 1D is the following plot.

```
meshplot(fem)
```

You can turn on node labeling, change the element color to red, and change the element edge boundary coloring to green by

```
meshplot(fem,'elcolor','r','boundcolor','g')
```

Compatibility

`meshplot` no longer supports the properties `boundlabels`, `curvelabels`, `ellabels`, and `labelcolor` from FEMLAB 2.3.

See Also

`femmesh`, `geomplot`, `postplot`

Purpose Mesh quality measure.

Syntax

```
q = meshqual(mesh)
```

Description `q = meshqual(mesh)` returns the mesh element quality for all elements in the mesh object `mesh`.

The mesh element quality is a number between 0 and 1. The quality is 1 for a perfect element.

Details on the mesh object can be found in the entry on `femmesh`.

The triangle quality is given by the formula

$$q = \frac{4\sqrt{3}a}{h_1^2 + h_2^2 + h_3^2},$$

where a is the area and h_1, h_2, and h_3 the side lengths of the triangle. If $q > 0.3$ the mesh quality should not affect the quality of the solution. $q = 1$ when $h_1 = h_2 = h_3$.

For a quadrilateral,

$$q = \frac{4A}{h_1^2 + h_2^2 + h_3^2 + h_4^2},$$

where h_1, h_2, h_3, and h_4 are the side lengths. q=1 for a square.

The tetrahedron mesh quality measure is given by

$$q = \frac{72\sqrt{3}V}{(h_1^2 + h_2^2 + h_3^2 + h_4^2 + h_5^2 + h_6^2)^{3/2}},$$

where V is the volume, and h_1, h_2, h_3, h_4, h_5, and h_6 are the side lengths of the tetrahedron. If $q > 0.1$ the mesh quality should not affect the quality of the solution.

For a hexahedron,

$$q = \frac{24\sqrt{3}V}{\left(\sum_{i=1}^{12} h_i^2\right)^{3/2}},$$

where h_i are the edge lengths. q=1 for a cube. For a prism,

$$q = \frac{36\sqrt{3}V}{\left(\sum_{i=1}^{9} h_i^2\right)^{3/2}},$$

where h_i are the edge lengths. q=1 for a right-angled prism where all edge lengths are equal.

The element quality is always 1 in 1D.

See Also

`meshrefine`, `femmesh`, `meshsmooth`

Reference

Bank, Randolph E., *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations, User's Guide 6.0*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1990.

Purpose Refine a mesh.

Syntax

```
fem.mesh = meshrefine(fem,...);
fem = meshrefine(fem,'Out',{'fem'},...);
```

Description `fem.mesh = meshrefine(fem,...)` returns a refined version of the triangular mesh specified by the geometry, `fem.geom`, and the mesh, `fem.mesh`.

`fem = meshrefine(fem,'Out',{'fem'},...)` modifies the FEM structure to include the refined mesh in `fem.mesh`.

`meshrefine` is supported for meshes containing simplex elements only, that is, lines, triangles, and tetrahedra.

The function `meshrefine` accepts the following property/values:

TABLE 1-96: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`mcase`	integer	`0`	Mesh case number
`out`	`fem` | `mesh`	`mesh`	Output variables
`rmethod`	`longest` | `regular`	see below	Refinement method
`subdomain`	integer array | `all` | `none`	`all`	Specifies the subdomains that are refined
`tri`	one row vector or two row matrix	all elements once	List of elements to refine (second row is number of refinements)

The default refinement method in 2D is regular refinement, where all of the specified triangles are divided into four triangles of the same shape. Longest edge refinement, where the longest edge of each specified triangle is bisected, can be demanded by giving `longest` as `Rmethod`. Using `regular` as `Rmethod` results in regular refinement. Some triangles outside of the specified set may also be refined, in order to preserve the triangulation and its quality.

In 3D, the default refinement method is `longest`. The regular refinement method is only implemented for uniform refinements.

In 1D, regular refinement, where each element is divided into two elements of the same shape, is always used.

Examples Refine the mesh of the L-shaped membrane several times and plot the mesh for the geometry of the L-shaped membrane:

```
clear fem
fem.geom = square2 + move(square2,0,-1) + move(square2,-1,-1);
fem.mesh = meshinit(fem,'hmax',0.4);
subplot(2,2,1), meshplot(fem)
fem.mesh = meshrefine(fem,'tri',[1:50;ones(1,50)]);
subplot(2,2,2), meshplot(fem)
fem.mesh = meshrefine(fem,'subdomain',...
                      [1 2],'rmethod','longest');
subplot(2,2,3), meshplot(fem)
fem.mesh = meshrefine(fem,'subdomain',1);
subplot(2,2,4), meshplot(fem)
```

Algorithm

The 2D algorithm is described by the steps below:

1. Pick the initial set of elements to be refined.
2. Either divide all edges of the selected elements in half (regular refinement), or divide the longest edge in half (longest edge refinement).
3. Divide the longest edge of any element that has a divided edge.
4. Repeat step 3 until no further edges are divided.
5. Introduce new points on all divided edges, and replace all divided entries in `e` by two new entries.
6. Form the new elements. If all three sides are divided, new elements are formed by joining the side midpoints. If two sides are divided, the midpoint of the longest edge is joined with the opposing corner and with the other midpoint. If only the longest edge is divided, its midpoint is joined with the opposing corner.

Compatibility

To achieve compatibility with FEMLAB 2.3, the geometry input is automatically converted to a geometry object using the function `geomobject`. The geometry input can be any analyzed geometry. See `geomobject`for details.

The output `u` is obsolete from FEMLAB 2.2. Use `asseminit` to interpolate the solution to a new mesh.

See Also

`femmesh`, `geomcsg`, `meshinit`

Purpose Revolve a 2D mesh object into a 3D mesh object.

Syntax

```
fem1 = meshrevolve(fem0,...)
[mesh,geom]= meshrevolve(fem,'Out',{'mesh','geom'},...)
```

Description `fem1 = meshrevolve(fem0,...)` revolves the 2D geometry object in `fem0.geom` and the 2D mesh object in `fem0.mesh`, into a 3D geometry object and a 3D mesh object stored in `fem1.geom` and `fem1.mesh`, respectively, according to the given parameters.

`[geom,mesh]= meshrevolve(fem,'Out',{'geom','mesh'},...)` returns the 3D geometry object in `geom` and the 3D mesh object in `mesh`.

Valid property/value pairs for the `meshrevolve` function are given in the following table. In addition, all `revolve` parameters are supported and are passed to `revolve` for creating the revolved 3D geometry object.

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`elrevlayers`	`scalar` | `vector`		Distribution of mesh element layers in revolved mesh
`mcase`	integer	`0`	Mesh case number
`out`	`fem` | `mesh` | `geom`	`fem`	Output variables

The property `elrevlayers` defines the distribution of the mesh element layers in the revolved mesh. If the value of `elrevlayers` is a scalar, it defines the number of equally distributed mesh element layers in the revolved mesh. Alternatively, if `elrevlayers` is a vector, it defines the distribution of the mesh element layers in the revolved mesh. The values in the vector, that are sorted and starts with 0, specifies the placements, in relative arc length, of the mesh element layers. By default, the number of element layers is determined such that the distance of each layer is equal to the mean size of the elements in the original 2D mesh.

Revolving a 2D mesh object into a 3D mesh object, the 2D vertex elements, the 2D boundary elements, the 2D triangular elements, and the 2D quadrilateral elements, are revolved into 3D edge elements, 3D quadrilateral boundary elements, 3D prism elements, and 3D hexahedral elements, respectively.

Examples Create a revolved prism mesh on a torus:

```
fem.geom = circ2(1,'pos',[2 0]);
fem.mesh = meshinit(fem);
fem1 = meshrevolve(fem);
```

Create a revolved hex mesh from the *zx* plane:

```
p_wrkpln = geomgetwrkpln('quick',{'zx',10});
ax = [0 1;0.5 2]';
fem.geom = rect2(1.5,1,'pos',[0.5 0]);
fem.mesh = meshmap(fem);
fem1 = meshrevolve(fem,'angles',[-pi/3 pi/3],...
                   'revaxis',ax,'wrkpln',p_wrkpln);
meshplot(fem1);
```

Cautionary

Revolving a triangular mesh adjacent to the revolution axis or a mesh containing a quadrilateral element with only one corner adjacent to the revolution axis is not supported.

See also

`meshembed`, `meshextrude`, `femmesh`, `revolve`

Purpose Smooth interior mesh vertices and improve quality of a mesh.

Syntax

```
fem.mesh = meshsmooth(fem,...)
mesh = meshsmooth(mesh,...)
fem = meshsmooth(fem,'out',{'fem'},...)
```

Description `mesh = meshsmooth(fem,...)` improves the quality of the elements in the mesh `fem.mesh` by adjusting the mesh vertex positions and by swapping mesh edges.

`mesh = meshsmooth(mesh,...)` improves the quality of the elements in the mesh object `mesh`.

`fem = meshsmooth(fem,'out',{'fem'},...)` modifies the `fem` structure to include the new mesh object in `fem.mesh`.

In 3D, `meshsmooth` is supported for meshes containing tetrahedral elements only.

The function `meshsmooth` accepts the following property/values:

TABLE 1-97: VALID PROPERTY/VALUE PAIRS

PROPERTY	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Jiggleiter`	√		numeric	5	Maximum number of jiggling iterations
`Qualoptim`	√		`off` | `mean` | `min` | `optim`	`min`	Optimization method
`Out`	√	√	`fem` | `mesh`	`mesh`	Output parameters
`Subdomain`	√	√	integer array | `all` | `none`	`all`	Specifies the subdomains that are smoothed

Algorithm *2D*

Each mesh vertex that is not located on the boundary is moved such that the quality of the surrounding element increases. This process is controlled via the properties `Jiggleiter` and `Qualoptim`.

`Jiggleiter` specifies the maximum number of jiggling iterations. The default value is 5.

`Qualoptim` specifies the technique that is used to improve the quality of the mesh.

- `Off` means that no improvement operations are performed.
- `Mean` means that jiggling is repeated until the mean element quality does not significantly increase, or until the bound `Jiggleiter` is reached. Furthermore, after every third jiggling iteration, edge swapping operations are performed.

- `Min` means that jiggling is repeated until the minimum element quality does not significantly increase, or until the bound `Jiggleiter` is reached. Furthermore, after every third jiggling iteration, edge swapping operations are performed.
- `Optim` means that a mesh quality optimizer is used. This tries to increase the minimum quality to at least 0.8. The `Jiggleiter` parameter has no effect in this case.

3D

Relocation of points similar to the 2D case is combined with edge swapping operations to improve the quality of the tetrahedra.

Examples

Create a triangular mesh of the L-shaped membrane without quality improvement and improve the quality by calling `meshsmooth`:

```
clear fem
sq1 = square2(0,0,1);
sq2 = move(sq1,0,-1);
sq3 = move(sq1,-1,-1);
fem.geom = sq1+sq2+sq3;
fem.mesh = meshinit(fem,'jiggle','off');
q = meshqual(fem.mesh);
minQual1 = min(q)
fem.mesh = meshsmooth(fem);
q = meshqual(fem.mesh);
minQual2 = min(q)
```

See Also

`meshqual`, `femmesh`, `meshinit`

Purpose Create swept mesh.

Syntax

```
fem.mesh = meshsweep(fem,...)
fem.mesh = meshsweep(fem.geom,...)
```

Description

`fem.mesh = meshsweep(fem,...)` returns a swept mesh derived from the 3D geometry in `fem.geom`.

`fem.mesh = meshsweep(geom,...)` returns a swept mesh derived from the 3D geometry `geom`.

A swept mesh is created for each subdomain by meshing the corresponding source face, if this face is not already meshed, and sweeping the resulting face mesh along the subdomain to the opposite target face. For straight and circular sweep paths it is allowed to use several connected faces as source faces.

If the source face for a subdomain is not meshed prior to the `meshsweep` operation, the source face is automatically meshed with triangles using the free triangle mesher (`meshinit`). Then, the swept mesh consists of prism elements. To create a swept mesh with hexahedral elements, you need to mesh the source face with quadrilateral elements using either the mapped quad mesher (`meshmap`), or the free quad mesher (`meshinit`), prior to the `meshsweep` operation.

The function `meshsweep` accepts the following property/value pairs.

TABLE 1-98: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Elsweeplayers`	cell array		Distribution of mesh element layers in the sweep direction
`Hauto`	numeric	5	Predefined mesh element size
`Mcase`	integer	`0`	Mesh case number
`Meshstart`	mesh object	empty	Starting mesh
`Out`	`fem` | `mesh`	`mesh`	Output variables
`Report`	`on` | `off`	`on`	Display progress
`Sourceface`	cell array		Source faces
`Subdomain`	integer array | `auto` | `all` | `none`	`auto`	Specifies the subdomains that are meshed

TABLE 1-98: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Sweeppath`	string | cell array	`auto`	Sweep path
`Targetface`	cell array		Target faces
`Targetmesh`	string | cell array	`auto`	Target mesh method

The property `elsweeplayers` determines the distribution of the mesh element layers in the sweep direction. The value of this property is an even numbered cell array where the odd entries contain subdomain indices, either as scalar values, or as vectors with subdomain indices, and the even entries contain scalar values or vectors specifying the element layer distribution for the corresponding subdomains. If the element layer distribution is specified by a scalar, the element layers for the corresponding subdomains are equally distributed and the number of element layers equals the value of the scalar. To get full control over the element layer distribution for a subdomain, the vector form is used. The values in the vector, that are sorted and starts with 0, specifies the distances of the element layers along the direction of the sweep for the corresponding subdomains.

`Hauto` is an integer between 1 and 9 that controls the element size in the generated mesh. The default value is 5 which means that the element size is set to $1/10$ of the size of the geometry for the elements not affected by the `elsweeplayers` property.

Use the property `subdomain` to specify the subdomains to be meshed. If you use this property together with the `meshstart` property, the value `auto` means that all subdomains that are not meshed in the starting mesh are meshed, `none` means that no further subdomains are meshed, and `all` means that all subdomains are meshed (or remeshed). It is also possible to specify the subdomains to be meshed (or remeshed) using a vector of subdomain indices.

The `meshstart` property is used when meshing a geometry interactively. The value of this property is the starting mesh of the meshing operation.

The property `sourceface` is a cell array specifying the source faces for the subdomains. The value of this property is an even numbered cell array where the odd entries contain subdomain indices, either as scalar values, or as vectors with subdomain indices, and the even entries contain the source face indices for the corresponding subdomains.

The property `targetface` defines the target faces for the subdomains. The syntax of this property is the same as for the `sourceface` property.

Use the property `sweeppath` if you want to specify the shape of the sweep path. The value of this property is a string specifying the sweep path for all subdomains or a an even numbered cell array where the odd entries contain subdomain indices, either as scalar values or as vectors with subdomain indices, and the even entries contain a string specifying the sweep path for the corresponding subdomains. The string is either `auto`, `straight`, `circular`, or `general`. `Straight` means that all interior mesh points are located on straight lines between the corresponding source and target points. `Circular` means that all interior mesh points are located on circular arcs between the corresponding source and target points. `General` means that the positions of the interior mesh points are determined by a general interpolation procedure. `Auto`, which is default, means that the `meshsweep` algorithm automatically tries to determine if the sweep path is straight or circular. If this is the case `sweeppath` is set to `straight` or `circular`, respectively. Otherwise, `sweeppath` is set to `general`.

Use the property `targetmesh` if you want to specify the method to be used for transferring the source mesh to the target face. The value of this property is a string specifying the target mesh method for all subdomains or a an even numbered cell array where the odd entries contain subdomain indices, either as scalar values or as vectors with subdomain indices, and the even entries contain a string specifying the target mesh method for the corresponding subdomains. The string is either `auto`, `morph`, or `rigid`. `Morph` means that the target mesh is created from the source mesh by a morphing technique, and `rigid` means that the target mesh is created by a rigid transformation of the source mesh. `Auto`, which is default, means that the `meshsweep` algorithm automatically tries to determine a suitable method for creating the target mesh.

If the source face or target face is not specified for a subdomain, the software automatically tries to determine these faces.

The following criteria must be met by the input geometry object for the sweeping technique to work.

- Each subdomain must be bounded by one shell, that is, a subdomain must not contain holes that do not penetrate the source and target face.
- There can only be one target face per subdomain. For straight and circular sweep paths several connected source faces are allowed. For other more general sweep paths there can only be one source face per subdomain.

- The source and target for a subdomain must be opposite one another in the subdomain's topology.
- The cross section along the direction of the sweep for a subdomain must be topologically constant.

Each face about a subdomain to be swept is classified as either a source face, a target face, or a boundary face. The boundary faces are the faces linking the source and target face. The sweep algorithm can handle subdomains with multiple boundary faces in the sweep direction.

If any of the faces about a subdomain is meshed prior to the `meshsweep` operation, the following must be fulfilled.

- If the source and target faces are meshed, these meshes must match.
- Mapped quad meshes must be applied to the boundary faces.

Cautionary

The `sweeppath` property cannot be set to `general` when sweeping from several source faces or when the `targetmesh` property is set to `rigid`. Furthermore, when sweeping from several source faces, the `targetmesh` property cannot be set to `morph`.

Algorithm

The subdomains to be meshed are processed in the following order.

1. The subdomains where the source and/or target faces are specified are swept first. These subdomains are swept in the order of increasing subdomain number.
2. The remaining subdomains, with no specified source face or target face but with one adjacent meshed face, are swept in the order of increasing subdomain number.
3. Finally, the remaining subdomains, with no specified source face or target face and with none or several adjacent meshed faces, are swept in the order of increasing subdomain number.

When a swept has been generated for a subdomain, the source face of the next subdomain, that is, the subdomain adjacent to the target face of the current subdomain, is set to the target face of the current subdomain if the source and target faces for the next subdomain are not specified.

For a subdomain with no specified source and target face, the source face is determined according to the following.

1. If not any face about the subdomain is meshed, the software determines the opposite face pairs for the subdomain. An opposite face pair is a pair of faces about

the subdomain that are not adjacent to each other but to all other faces about the subdomain. The face in these face pairs with lowest geometric degree and face index is used as source face and the opposite face is used as target face. If no opposite face pairs exist for the subdomain, an error is thrown and the user is asked to specify the source face.

2 If there is at least one meshed face about the subdomain, the source face is determined according to the following.

  a The face with lowest geometric degree and face index of the meshed faces about the subdomain that is not a boundary face of another subdomain.

  b The face with lowest geometric degree and face index of the unmeshed faces.

  c The face with lowest geometric degree and lowest face index of all faces about the subdomain.

Examples

Create a swept mesh on a cylinder geometry specifying the element layer distribution in the sweep direction.

```
fem.geom = cylinder3;
fem.mesh = meshsweep(fem,'sourceface',{1 3},...
                     'elsweeplayers',{1 logspace(0,1,11)-1});
meshplot(fem);
```

Create a swept mesh on a helix shaped geometry.

```
fem.geom = helix3;
fem.mesh = meshsweep(fem,'sourceface',{1 1});
meshplot(fem);
```

Create a mesh with both tetrahedra and prisms using `meshinit` and `meshsweep`, respectively.

```
fem.geom = block3 + cone3(0.3,1,pi/20,'pos',[1 0.5 0.5],...
           'axis',[1 0 0]) + cone3(0.3,1,pi/20,...
           'pos',[3 0.5 0.5],'axis',[-1 0 0]) + ...
            block3(1,1,1,'pos',[3 0 0]);
fem.mesh = meshsweep(fem,'subdomain',[2 3],'sourceface',{2 7});
fem.mesh = meshinit(fem,'meshstart',fem.mesh);
meshplot(fem);
```

Create a swept mesh with hexahedrons for a geometry with two subdomains by meshing the source faces prior to the `meshsweep` operation using the free quad mesher.

```
fem.geom = block3 + cylinder3(0.25,1,'pos',[0.5 0.5 1]);
fem.mesh = meshinit(fem,'subdomain',[],'face',...
                    [4 8],'edge',[],'point',[],...
```

```
                              'methodfac','quad','hauto',3);
  meshplot(fem);
```

See Also meshdel, meshinit, meshmap

Purpose Reflect geometry.

Syntax

```
[gm,...] = mirror(g,pt,vec,...)
```

Description `[gm,...] = mirror(g,pt,vec,...)` creates a mirrored copy of the geometry object `g`, as reflected in the plane with normal vector `vec`, centered at `pt`.

Property value list for `mirror`.

TABLE 1-99: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Out`	`stx` | `ftx` | `ctx` | `ptx`	`{}`	Cell array of output names

Examples In 2D:

```
  g = rect2;
  gm = mirror(g,[1 1],[1 1]);
  figure,geomplot(g),hold on,geomplot(gm),axis equal
```

In 3D:

```
  g = block3;
  gm = mirror(g,[1 1 1],[1 1 1]);
  figure, geomplot(g), hold on, geomplot(gm), axis equal
```

See Also `geom0, geom1, geom2, geom3, scale`

Purpose

Move geometry object.

Syntax

```
[g,...] = move(g3,x,y,z,...)
[g,...] = move(g2,x,y,...)
[g,...] = move(gn,x,...)
```

Description

`[g,...] = move(g3,x,y,z,...)` moves a 3D geometry object by the vector `(x,y,z)`.

`[g,...] = move(g2,x,y,...)` moves a 2D geometry object by the vector `(x,y)`.

`[g,...] = move(gn,x,...)` moves an nD geometry object by the vector `x` of length n.

The function `move` accepts the following property/values:

TABLE 1-100: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Out`	`stx` | `ftx` | `ctx` | `ptx`	`{}`	Output parameters

Examples

The commands below move the circle from the origin to (2,3) and plot the result.

```
c1 = circ2;
c2 = move(c1,2,3);
geomplot(c2)
```

See Also

`geomcsg`

Purpose Multiphysics function.

Syntax

```
fem1 = multiphysics(fem,...)
xfem1 = multiphysics(xfem,...)
```

Description `fem1 = multiphysics(fem)` combines the application modes in `fem.appl` to the composite system `fem1`.

`xfem1 = multiphysics(xfem)`, where `xfem` is a structure with a field `fem`, performs the above call for all of the structures `xfem.fem{:}`. The results are placed in `xfem1.fem`. In addition the fields `elemmph` and `eleminitmph` created for each structure in `xfem.fem` are concatenated and placed as fields in `xfem1`.

The function `multiphysics` accepts the following property/values:

TABLE 1-101: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Bdl`	cell array of integers | integer vector | `NaN`	all boundaries	Only affect the indicated boundaries in `fem.bnd`
`Defaults`	`on` | `off`	`off`	Return default fields
`Diff`	cell array of `ga`, `f`, `r`, or `g` | `on` | `off`	`on` for general output form, otherwise `off`	Differentiate these coefficients
`Edl`	cell array of integers | integer vector | `NaN`	all edges	Only affect the indicated edges in `fem.edg`
`Outform`	`coefficient` | `general` | `weak`	the most general output form of the application modes in `fem.appl`	Output form
`Outsshape`	positive integer | `NaN`	generated from the application modes in `fem.appl`	Output sshape. `NaN` is the same as not providing `Outsshape`. Can also be given for each structure in `xfem.fem`, by giving a cell array of values
`Pdl`	cell array of integers | integer vector | `NaN`	all points	Only affect the indicated points in `fem.pnt`
`Sdl`	cell array of integers | integer vector | `NaN`	all subdomains	Only affect the indicated subdomains in `fem.equ`
`Shrink`	`on` | `off`	`on`	Shrink coefficients to most compact form
`Simplify`	`on` | `off`	`on`	Simplify expressions

The properties `Diff`, `Outform`, `Outsshape`, `Rules`, and `Simplify` can also be specified by using fields in the `fem` structure. The default value for `Diff` is `'off'` if the resulting model has coefficient or weak form and `'on'` if it has general form.

For calls of the form `xfem1 = multiphysics(xfem,...)`, The properties `Sdl/Bdl/Edl/Pdl/Outsshape/Outform/Diff/Simplify` can be given as cell arrays indicating the properties' values for the different elements of `xfem.fem`. In this case the number of elements of the cell arrays giving the values must be the same as the number of elements of `xfem.fem`. For `Sdl/Bdl/Edl/Pdl/Outsshape`, the value `NaN` may be given if no value is actually intended to be passed.

For calls of the form `xfem1 = multiphysics(xfem,...)` Where the property `Out` is specified, the value `'fem'` means `xfem1` as described above, but `'equ'/'bnd'/'edg'/'pnt'/'dim'/'form'/'shape'/'sshape'/'border'/'var'` are returned as cell arrays containing these properties for each of the elements of `xfem.fem`, and `'elemmph'/'eleminitmph'` are concatenations of the individual resulting properties.

Algorithm

The description below relates to the call `fem1 = multiphysics(fem)` unless otherwise stated. The applications are specified as a cell array in the field `appl` in the `fem` structure: `fem.appl={a1 a2 ...}`.

The notations `***`, `XXX` and `xxx` used below means the fields `equ`, `bnd`, `edg`, and/or `pnt`. `***` denotes any of these fields, `XXX` is used to denote the field which corresponds to the largest dimension (usually `equ`), while `xxx` denotes the fields corresponding to lower dimensions (usually `bnd`, `edg`, and `pnt`). Not all of them are always present, rather their presence is dependent on the geometry of the domain and the type of problem being solved.

The table below describes the fields in the `appl` structure.

TABLE 1-102: APPL STRUCTURE FIELDS

FIELD	INTERPRETATION
`appl.assign`	Application mode variable name assignments
`appl.bnd`	Boundary coefficients/application data
`appl.border`	Cell array with strings `on/off` or a corresponding logical vector, one for each solution component
`appl.dim`	Cell array with names of the solution components or an integer specifying the number of solution components
`appl.edg`	Edge coefficients/application data

TABLE 1-102: APPL STRUCTURE FIELDS

FIELD	INTERPRETATION
`appl.elemdefault`	A string indicating what kind of element is the default in this application
`appl.equ`	Subdomain coefficients/application data
`appl.form`	Problem form: `coefficient`, `general`, or `weak`
`appl.mode`	Application mode
`appl.pnt`	Point coefficients/application data
`appl.shape`	Shape functions. A cell array of shape function objects.
`appl.sshape`	Element geometry order. An integer.
`appl.***.shape`	Pointers to `appl.shape`
`appl.XXX.usage`	Activate/deactivate domain
`appl.***.gporder`	Order of numerical quadrature
`appl.***.cporder`	Constraints discretization order
`appl.XXX.init`	Initial conditions
`appl.var`	Application mode specific variables

`appl.assign` contains the application mode variable name assignments. It is a cell array of alternating fixed names and assigned names for the application mode variables. The default is an empty cell array.

`appl.bnd` is a structure with fields describing the boundary data. The structure contains one field for each of the application-specific boundary parameters, with the field name equal to the parameter name. Each field should be a cell array containing data for the boundaries. `appl.bnd` also contains a field `type`, which is a cell array with strings describing the boundary type for each boundary. If only one boundary type is available in the application mode, the `type` field may be omitted.

`appl.dim` provides the dependent variable names. The default is obtained from the application mode for physics modes. For PDE modes with no mode field and with a numeric `appl.dim`, default variable names are substituted according to the global position in the system.

`appl.elemdefault` contains a string indicating what kind of element is the default in this application. This method is used to generate the defaults for `appl.XXX.gporder`, `appl.XXX.cporder`, `appl.shape`, `appl.XXX.shape`, and `appl.sshape`. See `elemdefault` for a list of valid strings.

`appl.edg` is a structure with fields describing the edge data. The structure contains one field for each of the application-specific edge parameters, with the field name equal to the parameter name. Each field should be a cell array containing data for the edges.

`appl.equ` is a structure with fields describing the PDE data on subdomains. The structure contains one field for each of the application-specific PDE parameters, with the field name equal to the parameter name. Each field should be a cell array containing data for the subdomains.

`appl.mode` is a string with the name of the application mode or an application mode object. If the `appl.mode` field is omitted, the application structure can be used to describe ordinary coefficient/general/weak form PDE problem. This is useful in the definition of multiphysics problems, where `coefficient/general/weak` form models can be combined with application mode models. See the *COMSOL Modeling Guide* for a description of the application-specific data for each application mode.

`appl.pnt` is a structure with fields describing the point data. The structure contains one field for each of the application-specific point parameters, with the field name equal to the parameter name. Each field should be a cell array containing data for the points.

`appl.shape` is a cell array of shape function objects.

`appl.sshape` is an integer giving the order of geometry approximation.

The fields `fem.XXX.shape` and `fem.xxx.shape` are ind-based cell arrays of vectors pointing to elements of `appl.shape`. Indicates which shape functions to use in each domain group. An empty vector indicates that no shape functions are used in this domain group. Zero indicates that the affected domain group should use defaults or inherit shape functions. `Appl.XXX.shape` takes defaults, whereas `appl.xxx.shape` inherits from `appl.XXX.shape`. Where there is a conflict over which domain group in `appl.XXX.shape` to inherit from, the first appropriate group is used.

The fields `fem.appl.xxx.usage` are `ind`-based cell arrays of ones and zeros indicating domain group usage. Wherever a zero entry exists, the information in `appl.xxx.shape` is ignored when forming `fem.xxx.shape`.

The fields `appl.XXX.gporder` and `appl.xxx.gporder` indicate the order of quadrature formula to use in the different domain groups. In fully expanded form it is a cell array where each element is a cell array (of positive integers) of length equal to the number of dependent variables (excluding submode variables) in this mode. Defaulting and inheritance can be induced by using 0. The inherited order is the maximum order used in the objects from the XXX level in contact with the group at the xxx level. Where different elements within a domain group would inherit different orders, some domain group splitting takes place. This field is not present in the appl.pnt field.

The fields appl.XXX.cporder and appl.xxx.cporder indicate domain group constraints discretization order. Behaves exactly as gporder.

The fields appl.XXX.init indicate domain group initial conditions. For format see asseminit.

appl.var contains the application scalar variables. The default is obtained from the application mode for physics modes.

The composite system is created by appending the subsystems in the order they are specified in fem.appl. The affected fields in the fem1 structure are dim, form, equ, bnd, edg, pnt, var, elemmph, eleminitmph, shape, sshape, and border. All the other fields in fem are copied to fem1. In the description below the notation *** is used to represent any or all of the fields equ, bnd, edg, and/or pnt.

The dim field of the composite system, fem1.dim, is a cell array of the dependent variable names:

```
fem1.dim={fem.appl{1}.dim{:}, fem.appl{2}.dim{:}, ...}
```

The default names ('u1', 'u2', ...) Are used for subsystems with integer dim fields and no mode field.

The form of each subsystem may be converted by using flform, in the direction 'coefficient' -> 'general' -> 'weak'. The default form of the composite system, fem1.form, is the first form which all the subsystems may attain, possibly applying flform to some subsystems. The output form can be forced by using the property outform.

First, each application structure is converted to an FEM structure with all the above-mentioned fields. Then the fields in fem1.*** are computed from the corresponding fields in the subsystems according to the table below. The numbers after the coefficient names refer to the subsystems in the order they are specified in `fem.appl`.

QUANTITY	COMPOSITE SYSTEM
$f, \gamma, g, r,$ init weak, dweak, constr gporder, cporder	$\begin{bmatrix} f_1 \\ f_2 \\ \ldots \end{bmatrix}$
$c, d_a, e_a, \alpha, \beta, a, q, h$	$\begin{bmatrix} c_1 & 0 & 0 \\ 0 & c_2 & 0 \\ 0 & 0 & \ldots \end{bmatrix}$
var, varu, vart	$\begin{bmatrix} \mathrm{var}_1 & \mathrm{var}_2 & \ldots \end{bmatrix}$

The fields `fem.***.expr` are kept unchanged as far as possible; the only difference being the permutation of `ind`-groups to suit a new `ind` vector if one is generated. For `ind` groups where a particular `expr` variable is undefined, the entry `[]` is used.

Note that the coefficients in the second row of the table are *weakly* coupled in the sense that the corresponding coefficients in the composite system are block diagonal. This puts some limitations on the coupling between the subsystems. By using `general` form, however, there are no limitations on the composite system except for the `da` coefficient. The resulting stronger couplings are obtained by using a call to `femdiff` with the full system resulting from the composition of the subsystems. The properties `Diff`, `Rules`, and `Simplify` control or supplement the call to `femdiff`. Further couplings between applications can be introduced by the method `global_compute` which is called for each application before `femdiff`.

Elements of `fem1.***.(f/ga/g/r/weak/dweak/constr/c/da/ea/al/be/a/q/h)` which correspond to subsystems for which the corresponding field does not exist are `'0'`. Elements of `fem1.***.init` in such cases are empty strings. Elements in `fem1.***.(gporder/cporder)` in such cases are `1`. The fields `fem1.elemmph` and `fem1.eleminitmph` are obtained by concatenating the contents of the results of calling `elem_compute` for each application. The field `fem1.shape` is obtained by concatenating the contents of the fields `fem.appl{:}.shape`. Duplicate shape functions are removed and the fields `fem.***.shape` are adjusted to take account of this. The field `fem1.sshape` is obtained by taking the maximum of the fields `fem.appl{:}.sshape`. The resulting value is overridden if the property `Outsshape` is given. The `fem1.border` field is always `1` because coefficients for boundary conditions that are not used due to border being on or off in the application mode are set to zero and can be “applied”.

Suppose multiphysics has been called previously, and fem is the result of such a call. If changes are made to fem.equ and it is wished to keep them (that is not allow changes to be written over when multiphysics is called again), it is possible to restrict the set of subdomains for which multiphysics writes over fem.equ. Thus giving the property Sdl the value [1 2] in a call to multiphysics results in the coefficients for subdomains 1 and 2 being “refreshed” from the applications, but all the coefficients in fem.equ relating to other subdomains being kept and copied into fem1.equ. The same principle holds for bnd/edg/pnt using the properties Bdl/Edl/Pdl.

Example

The model “Resistive Heating” uses the multiphysics function. Note that the structure fem contains the data that is *common* for the subsystems, that is, the geometry and the mesh. The electrical subsystem and the heat transfer subsystem are specified in the application structures a1 and a2, and the multiphysics function is used to combine them.

```
clear fem a1 a2
fem.geom = geomcsg({square2(0,0,1)});
fem.mesh = meshinit(fem);
a1.mode = 'ConductiveMediaDC';
a1.assignsuffix = '_dc';
a1.bnd.V0 = {0.1 0 0};
a1.bnd.type = {'V' 'nJ0' 'V0'};
a1.bnd.ind = [1 2 2 3];
a1.equ.init = '0.1*(1-x)';
a1.equ.T0 = 293;
a1.equ.T = 'T';
a1.equ.alpha = 0.0039;
a1.equ.res0 = 1.754e-8
a1.equ.sigtype = 'heat';
a2.mode = 'HeatTransfer';
a2.assignsuffix = '_ht';
a2.bnd.T0 = {300 0};
a2.bnd.type = {'T' 'q0'};
a2.bnd.ind = [1 2 2 1];
a2.equ.rho = 8930;
a2.equ.C = 340;
a2.equ.k = 384;
a2.equ.Q = 'Q_dc';
a2.equ.init = 300;
fem.appl = {a1 a2};
fem = multiphysics(fem);
fem.xmesh=meshextend(fem);
fem.sol = femtime(fem,'tlist',linspace(0,3000,41), ...
                  'report','on');
```

Diagnostics

All variables in the `appl{:}.dim` fields must be unique. If any `appl{:}.dim` has not been provided, no other `appl{:}.dim` may collide with the defaults. If they do, an error message is generated.

If any form `appl{:}.form` is more general than `Outform`, an error message is generated.

Compatibility

To simplify the output of the `multiphysics` functions in FEMLAB 3.0, its defaults has been changed by the introduction of the properties `Shrink` and `Defaults`. The compatibility problems typically occur when you perform changes to the data generated by `multiphysics`. For example, modifying the α coefficient

```
fem=multiphysics(fem);
fem.equ.al{1}{2,1}=...;
fem.equ.al{1}{3,1}=...;
```

may not work as in FEMLAB 2.3 because alpha is often just empty. The code above assumes that `fem.equ.al{1}` is a 3-by-3 cell array. To obtain fully backward compatible output, use

```
fem=multiphysics(fem,'shrink','off','defaults','on');
```

which makes the above example work.

The properties out and rules are obsolete from FEMLAB 3.0.

The property `Idl` is obsolete in FEMLAB 2.2 and later versions.

The fields `fem.init` and `fem.usage` are no longer constructed. They are superseded by the fields `fem.***.init` and `fem.***.shape`, respectively. `Fem.***.init` is constructed for all fields `fem.***`, but many entries contain just empty strings.

See also

`femdiff`, `flform`, `multiphysics`

Purpose

Constructor functions for point objects.

Syntax

```
p3 = point3(x,y,z)
p3 = point3(vtx,vtxpre,edg,edgpre,fac,mfdpre,mfd)
[p3,...] = point3(g3,...)
p2 = point2(x,y)
p2 = point2(vtx,edg,mfd)
[p2,...] = point2(g,...)
p1 = point1(x)
p1 = point1(vtx)
[p1,...] = point1(g,...)
```

Description

`p3 = point3(x,y,z)` creates a 3D single point object with coordinate (x, y, z).

`p3 = point3(vtx,vtxpre,edg,edgpre,fac,mfdpre,mfd)` creates a 3D point geometry object `p3` from the arguments `vtx`, `vtxpre`, `edg`, `edgpre`, `fac`, `mfdpre`, `mfd`. The arguments must define a valid 3D point object. See `geom3` for a description of the arguments.

`p3 = point3(g3)` coerces the 3D geometry object `g3` to a 3D point object `p3`.

`p2 = point2(x,y)` creates a 2D point object consisting of a single point with coordinates `(x,y)`.

`p2=geom2(vtx,edg,mfd,…)` creates a 2D point object from the properties `vtx`, `edg`, and `mfd`. The arguments must define a valid 2D point object. See `geom2` for information on `vtx`, `edg`, and `mfd`.

`p2 = point2(g2)` coerces the 2D geometry object to a point object.

`p1 = point1(x)` creates a 2D point object consisting of a single point with coordinate `x`.

`p1 = point1(vtx,…)` creates a 1D point object from `vtx`. The arguments must define a valid 1D point object. See `geom1` for information on `vtx`.

`[p1,…] = point1(g,…)` coerces the 1D geometry object to a point object.

The coercion functions `[p1,…] = point1(g1,…)`, `[p2,…] = point2(g2,…)`, and `[p3,…] = point3(g3,…)` accept the following property/values:

TABLE 1-103: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Out`	`stx` | `ftx` | `ctx` | `ptx`	`{}`	Cell array of output names.

See `geomcsg` for more information on geometry objects.

The nD geometry object properties are available. The properties can be accessed using the syntax `get(object,property)`. See `geom` for details.

Example

The commands below create a 2D point object with four points and plot the result.

```
c1 = circ1;
p1 = point2(c1);
geomplot(p1)
```

Compatibility

The FEMLAB 2.3 syntax is obsolete but still supported.

See Also

`curve2, curve3`, `face3`, `geom0, geom1, geom2, geom3`, `geomcsg`

Purpose

Create polygons.

Syntax

```
c = poly1(x,y)
s = poly2(x,y)
```

Description

`s = poly2(x,y)` creates a 2D solid object `s` in the form of an solid polygon with vertices given by the vectors `x` and `y`.

`c = poly1(x,y)` creates a 2D curve object `c` in the form of an closed polygon with vertices given by the vectors `x` and `y`.

See `geomcsg` for more information on geometry objects.

Example

The commands below create a regular n-gon (n=11) and plot it.

```
  n = 11
  xy = exp(i*2*pi*linspace(0,1-1/n,n));
  p = poly1(real(xy),imag(xy));
  geomplot(p)
```

Cautionary

`poly1` and `poly2` always creates closed polygon objects. To create open polygon curves, use `line1` and `line2`.

See Also

`arc1`, `arc2`, `circ1`, `circ2`, `ellip1`, `ellip2`, `geomcsg`, `line1`, `line2`

Purpose

Shorthand command for animation in 1D, 2D and 3D.

Syntax

```
postanim(fem,expr,...)
M = postanim(fem,expr,...) % MATLAB only
```

Description

`postanim(fem,expr,...)` plots an animation of the expression `expr`. The function accepts all property/value pairs that `postmovie` does. In 1D, this command is just shorthand for the call

```
postmovie(fem,'liny',expr,...
              'linstyle','bginv',...)
```

and in 2D, it is shorthand for

```
postmovie(fem,'tridata',expr,...
              'tribar','on',...
              'geom','on',...
              'axisequal','on',...)
```

and in 3D, this command is just shorthand for

```
postmovie(fem,'slicedata',expr,...
              'slicebar','on',...
              'geom','on',...
              'axisequal','on',...)
```

`M = postanim(fem,expr,...)` additionally returns a matrix in MATLAB movie format.

If you want to have more control over your animation, use `postmovie` instead of `postanim`.

Cautionary

When you are replaying a movie that has been stored in a matrix `M`, you should explicitly provide a figure handle to the `movie` command.

```
M = postanim(fem,expr,...)
movie(gcf,M)
```

Otherwise the animation does not look good.

Compatibility

The syntax of the command is not compatible with its corresponding FEMLAB 2.1 syntax.

See Also

`postplot`, `postsurf`, `postcont`, `postlin`, `postarrow`, `postarrowbnd`, `postflow`, `postslice`, `postiso`, `posttet`

Purpose Shorthand command for subdomain arrow plot in 2D and 3D.

Syntax

```
postarrow(fem,expr,...)
h = postarrow(fem,expr,...)
```

Description `postarrow(fem,expr,...)` plots a subdomain arrow plot for the expressions in the cell array `expr`. In 2D, `expr` has length 2 or 3, and in 3D, it has length 3. The function accepts all property/value pairs that `postplot` does. This command is just shorthand for the call

```
  postplot(fem,'arrowdata',expr,...
               'geom','on',...
               'axisequal','on',...)
```

`h = postarrow(fem,expr,...)` additionally returns handles to the plotted handle graphics objects.

If you want to have more control over your arrow plot, use `postplot` instead of `postarrow`.

See Also `postplot`, `postanim`, `postsurf`, `postcont`, `postlin`, `postarrowbnd`, `postflow`, `postprinc`, `postprincbnd`, `postslice`, `postiso`, `posttet`

Purpose

Shorthand command for boundary arrow plot in 2D and 3D.

Syntax

```
postarrowbnd(fem,expr,...)
h = postarrowbnd(fem,expr,...)
```

Description

`postarrowbnd(fem,expr,...)` plots a boundary arrow plot for the expressions in the cell array `expr`. In 2D, `expr` has length 2 or 3, and in 3D, it has length 3. The function accepts all property/value pairs that `postplot` does. This command is just shorthand for the call

```
postplot(fem,'arrowbnd',expr,...
         'geom','on',...
         'axisequal','on',...)
```

`h = postarrowbnd(fem,expr,...)` additionally returns handles to the plotted handle graphics objects.

If you want to have more control over your arrow plot, use `postplot` instead of `postarrowbnd`.

See Also

`postplot`, `postanim`, `postsurf`, `postcont`, `postlin`, `postarrow`, `postflow`, `postprinc`, `postprincbnd`, `postslice`, `postiso`, `posttet`

Purpose Return a MATLAB colormap for a COMSOL color table.

Syntax

```
m = postcolormap(name)
```

Description `m = postcolormap(name)` returns the color table (of 1024 colors) for `name`, where `name` can be one of the following strings:

TABLE 1-104: THE COMSOL COLOR TABLES

NAME	DESCRIPTION
`Cyclic`	A color table that varies the hue component of the hue-saturation-value color model, keeping the saturation and value constant (equal to 1). The colors begin with red, pass through yellow, green, cyan, blue, magenta, and finally return to red. This table is particularly useful for displaying periodic functions and has a sharp color gradient.
`Disco`	This color table spans from red through magenta and cyan to blue.
`DiscoLight`	Similar to Disco but uses lighter colors.
`GrayScale`	A color table that uses no color, only the gray scale varying linearly from black to white.
`GrayPrint`	Varies linearly from dark gray (0.95, 0.95, 0.95) to light gray (0.05, 0.05, 0.05). This color table overcomes two disadvantages that the GrayScale color table has when used for printouts on paper, namely that it gives the impression of being dominated by dark colors, and that white cannot be distinguished from the background.
`Rainbow`	The color ordering in this table corresponds to the wavelengths of the visible part of the electromagnetic spectrum: beginning at the small-wavelength end with dark blue, the colors range through shades of blue, cyan, green, yellow, and red.
`RainbowLight`	Similar to Rainbow, this color table uses lighter colors.
`Thermal`	Ranges from black through red and yellow to white, which corresponds to the colors iron takes as it heats up.
`ThermalEquidistant`	Similar to Thermal but uses equal distances from black to red, yellow, and white, which means that the black and red regions become larger.
`Traffic`	Spans from green through yellow to red.

TABLE 1-104: THE COMSOL COLOR TABLES

NAME	DESCRIPTION
`TrafficLight`	Similar to Traffic but uses lighter colors.
`Wave`	Ranges linearly from blue to light gray, and then linearly from white to red. When the range of the visualized quantity is symmetric around zero, the color red or blue indicates whether the value is positive or negative, and the saturation indicates the magnitude.
`WaveLight`	Similar to Wave and ranges linearly from a lighter blue to white (instead of light gray) and then linearly from white to a lighter red.

Example

Calling `postcolormap` is equivalent to calling the corresponding color-table function directly.

```
m = postcolormap('Rainbow');
m = rainbow;
```

Purpose Shorthand command for contour plot in 2D.

Syntax

```
postcont(fem,expr,...)
h = postcont(fem,expr,...)
```

Description `postcont(fem,expr,...)` plots a contour plot for the expression `expr`. The function accepts all property/value pairs that `postplot` does. This command is just shorthand for the call

```
postplot(fem,'contdata',expr,...
             'contbar','on',...
             'geom','on',...
             'axisequal','on',...)
```

`h = postcont(fem,expr,...)` additionally returns handles to the plotted handle graphics objects.

If you want to have more control over your contour plot, use `postplot` instead of `postcont`.

Example Plot the contours of the solution to the equation $-\Delta u = 1$ over a unit circle. Use Dirichlet boundary conditions $u = 0$ on $\partial\Omega$.

```
clear fem
fem.geom = circ2;
fem.mesh = meshinit(fem);
fem.mesh = meshrefine(fem);
fem.equ.c = 1; fem.equ.f = 1;
fem.bnd.h = 1;
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem);
postcont(fem,'u')
```

Compatibility The syntax of the command is not compatible with its corresponding FEMLAB 2.1 syntax.

See Also `postplot`, `postanim`, `postsurf`, `postlin`, `postarrow`, `postarrowbnd`, `postflow`, `postprinc`, `postprincbnd`, `postslice`, `postiso`, `posttet`

Purpose Get coordinates in a model.

Syntax `coord = postcoord(fem,...)`

Description `coord = postcoord(fem,...)` returns global coordinates in a model by specifying, for example, a boundary and the number of points on the boundary.

To specify start points for particle tracing in `postplot`, property/values to `postcoord` can be specified in the `postplot` property `partstart`.

Valid property/value pairs for `postcoord` are given in the following table.

TABLE 1-105: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`coord`	cell array of double vectors		Coordinates where scalar expansion is used
`dl`	integer vector	all domains	Domain list
`edim`	`-1` | `1`	`-1`	Element dimension
`frame`	string	spatial frame	Coordinate frame
`geomnum`	positive integer	1	Geometry number
`grid`	positive integer or vector of domain parameters		Domain parameters
`mcase`	non-negative integer		Mesh case
`npoints`	non-negative integer		Number of subdomain points, picked from mesh vertices (`edim=-1`)
`solnum`	integer vector | `all` | `end`	See below	Solution number
`T`	double vector	See below	Time for evaluation
`U`	solution object | solution vector | scalar	`fem.sol` or 0	Solution(s) for evaluation

The properties `Frame`, `Solnum`, `T`, and `U` are only used when the model `fem` contains moving meshes. The default behavior of `Solnum` and `T` are described in `postinterp`.

Examples

```
% Set up a 2D geometry with a mesh and an extended mesh
clear fem
fem.geom = circ2+rect2;
fem.mesh = meshinit(fem);
fem.xmesh = meshextend(fem);

% Get the coordinates of 17 evenly spaced points on
% boundaries 1,2,4
coord = postcoord(fem,'edim',1,'grid',17,'dl',[1,2,4]);
```

Compatibility This function was introduced in COMSOL Multiphysics 3.3.

See Also `postinterp`, `postplot`

Purpose Cross-section plot.

Syntax

```
postcrossplot(fem,cdim,dom,...)
h = postcrossplot(fem,cdim,dom,...)
[h,data] = postcrossplot(fem,cdim,dom,...)
```

Description `postcrossplot(fem,cdim,dom,...)` displays a plot of an expression, including an FEM solution, in one or several cross sections of the geometry, with space dimension `cdim` and defined by `dom` (coordinates) or on mesh elements of space dimension `cdim`, specified by `dom` (domain list). The argument `dom` is therefore either a number of points to specify a cross section or a list of domains. To specify a cross section, `dom` must be one of the following, where `sdim` denotes the space dimension of the geometry:

- An `sdim`-by-2 matrix to specify a cross-section line (the line between the two points in the columns of `dom`) in 2D and 3D. Used with properties `lindata` and `surfdata`. Here, `cdim` must be `1`.
- A 3-by-3 matrix to specify a cross-section plane (the plane containing all three points in the columns of `dom`) in 3D. Used with property `surfdata`. Here, `cdim` must be `2`.
- An `sdim`-by-`np` matrix to specify `np` points for point plots. Used with property `pointdata`. Here, `cdim` must be `0`.

To specify a list of domains (geometry vertices, geometry edges, geometry boundaries, or geometry subdomains), `dom` must be:

- A 1-by-`nd` integer matrix, where `nd` is the number of domains with space dimension `cdim` (vertices, edges, boundaries, or subdomains) to plot on. When used with `surfdata` in 3D, `nd` must be one.

In 1D, if all entries in `dom` are integers greater than or equal to 1, they are interpreted as indices to geometry vertices. Otherwise, they are interpreted as coordinates. To specify a coordinate which is an integer greater than or equal to 1, use the property `pointtype` set to `coord`.

The expressions that are plotted can be COMSOL Multiphysics expressions involving variables, in particular *application mode variables.*

The following plot types can be made:

Point plots (1D, 2D, 3D) Plot of an expression on any geometry vertex or arbitrary point in the subdomains of the geometry. This is most useful when there are several solutions, in which case this plot shows the value of the expression in the selected points for the different solutions. If there is only one solution, the values in the specified points are displayed in the x-axis range [0 1].

Line plots on domains (1D, 2D, 3D) Plot of an expression on a set of connected 1D domains (edges in 3D, boundaries in 2D and subdomains in 1D). If `Linxdata` is not specified, this is done by folding out the arc length of the 1D domain to the *x*-axis of the resulting plot, and letting the value of the expression be set on the *x*-axis. If `dom` contains more than one domain, the different arc lengths are just added to each other on the *x*-axis. In this case, the domains have to be connected and so that not more than two selected domains meet in the same vertex.

If Linxdata is specified, this is the quantity on the *x*-axis in the resulting plot. Using `Linxdata`, you can project cross sections to, e.g., the *x*-axis by setting `Linxdata` to *x*.

The direction of the path is so that the start point is the point along the path with lowest geometry vertex number. If the selected domains form a closed curve, so that this point is also the end point, the direction is in the direction of the domain with lowest number.

If there are several solutions (that is, `Solnum` or `T` is a vector), the curves for the different solutions can be either plotted in the same *x*-*y*-plot or can be extruded along the third axis to generate a surface. This is controlled by specifying either the property `Lindata` or `Surfdata`.

Line plots on cross sections (2D, 3D) In 2D and 3D, plot of an expression along a straight line, defined between the two points in `dom`, in the geometry. The points in `dom` are regarded as the end points of the cross-section line and are the *x*-axis limits in the resulting plot, hence if the line between the two specified points do not intersect the geometry, the resulting plot will be empty.

Surface plots on domains (2D, 3D) Plot of an expression on a boundary in 3D or on a set of subdomains in 2D. In 3D, if `Surfxdata` and `Surfydata` are not specified, the boundary is plotted in an *xy*-plane where *x* and *y* correspond to the (s,t)-parameters of the boundary. If `Surfxdata` and `Surfydata` are specified, these represent the quantities on the *x*- and *y*-axis in the resulting plot. Using `Surfxdata` and `Surfydata`, you can project a cross section to, for example, the *xy*-plane of the geometry by setting `Surfxdata` to *x* and `Surfydata` to *y*. If there are several solutions, all plots for the different solutions are displayed along the third axis, as a slice plot. In 3D, dom must be a single integer.

Surface plots on cross sections (3D) Plot of an expression on one or more 2D cross-section planes, defined by the three points in dom, of the 3D geometry. If more than one cross section is selected, or only one cross section is selected but there are several solutions, all plots for the different cross sections/solutions are displayed along the third axis, as a slice plot. The cross-section plane is the plane containing the three points in the columns of dom.

For line plots, if more than one curve is plotted (either one cross section and several solutions or vice versa), the different curves can be either plotted in the same *x*-*y*-plot or can be extruded along the third axis to generate a surface. This is controlled by specifying either the property Lindata or Surfdata.

h = postcrossplot(fem,cdim,dom,...) additionally returns handles or a postdata structure (depending on the value of the property Outtype) to the plotted objects.

Valid property/value pairs for postcrossplot are given in the following table, where the columns S, L, and P denote if the property has effect on surface, line and point plots, respectively.

TABLE 1-106: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	S	L	P	PROPERTY VALUE	DEFAULT	DESCRIPTION
Axistype	√	√	√	cell array of strings lin or log		X-, Y- and Z-axis types
Complexfun	√	√	√	off | on	on	Definition of constants
Const	√	√	√	cell array		Definition of constants
Cont	√	√	√	off | on	off	Make output continuous
Crosslicecs	√			local | global	global	Coordinate system to plot cross-section slices in
Geom	√			off | on		Show geometry contour
Frame	√	√	√	string	spatial frame	Coordinate frame
Geomnum	√	√	√	integer (or vector of integers when cdim=2 and dom is 3-by-3)	1	Geometry number
Lincolor		√	√	colorspec | cell array of colorspec	cycle	Line color
Lindata		√		string		Expression to plot

TABLE 1-106: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	S	L	P	PROPERTY VALUE	DEFAULT	DESCRIPTION
Linewidth		√	√	numeric	0.5	Line width
Linlegend		√	√	off | on	off	Color legend
Linmarker		√	√	marker specifier | cell array of marker specifiers	none	Line marker
Linstyle		√	√	symbol | cell array of symbols	'-'	Line style
Linxdata		√		string		Line x-axis expression
Markersize		√	√	integer	6	Size of markers
Matherr	√	√	√	off | on	off	Error for undefined operations
Npoints		√		integer	200	Number of points on each line, when dom is a cross section
Outtype	√	√	√	handle | postdata | postdataonly	handle	Output type
Phase	√	√	√	scalar	0	Phase angle
Pointdata			√	string		Expression to plot
Pointtype			√	coord | vertex	vertex	Point plot type when cdim=0 and dom is integer(s)
Pointxdata			√	string		Point plot x-axis expression
Recover	√	√	√	off | ppr | pprint	off	Accurate derivative recovery
Refine	√	√		integer | auto	See posteval	Refinement of element in evaluation
Sdl	√			integer vector | cell array of integer vectors | all	all	Subdomain list for cross-section slice plots
Solnum	√	√	√	integer vector | all | end	all	Solution numbers
Spacing	√	√		integer	1	Number of planes/lines or vector with distances, when dom is a cross section
Surfbar	√			off | on	off	Color legend
Surfdata	√	√		string		Expression to plot

TABLE 1-106: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	S	L	P	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Surfdlim`	√			`[`*`min max`*`]`	full range	Surface plot color limits
`Surfedgestyle`	√			`flat` | `interp` | `none` | `bg` | `bginv` | colorspec	`none`	Triangle edge style
`Surffacestyle`	√			`flat` | `interp` | `none` | `bg` | `bginv` | colorspec	`interp`	Triangle face style
`Surfmap`	√			color table		Color table
`Surfmapstyle`	√			`auto` | `reverse`	`auto`	Color table style
`Surfxdata`	√			string		Surface x-axis expression
`Surfydata`	√			string		Surface y-axis expression
`T`	√	√	√	vector		Times for evaluation
`U`	√	√	√	solution object or vector	`fem.sol` or 0	Solution for evaluation

In addition, the common plotting properties listed under `femplot` are available.

If the field `fem.sol.u` does not exist and the property `U` is not specified, expressions not depending on the solution can still be plotted.

The notation colorspec in the value column denotes a *color specification*. See postplot for a description of this.

The property `Phase` is described in `posteval`.

Examples

3D Example

```
clear fem
fem.geom = geomcsg({cylinder3});
fem.mesh = meshinit(fem);
fem.equ.c = 1; fem.equ.f = 1; fem.equ.da = 1;
fem.bnd.h = 1;
fem.shape = 2;
fem.xmesh = meshextend(fem);
fem.sol = femtime(fem,'tlist',0:0.01:0.1);
```

Plot solutions on a cross section:

```
crosspts = [0 0 1;0 1 1;0 0 1];
postcrossplot(fem,2,crosspts,'surfdata','u','solnum',1:2:10,...
              'crosslicecs','local')
```

Plot fourth solution on five cross sections with geometry boundaries:

```
postcrossplot(fem,2,crosspts,'surfdata','u','solnum',4,...
              'geom','on','refine',3,'axisequal','on',...
              'spacing',5)
```

Plot on a boundary 6 for the last time value:

```
postcrossplot(fem,2,6,'surfdata','ux','cont','on','solnum',11)
```

Compare this with

```
postplot(fem,'tridata','ux','cont','on','bdl',6,'geom','on',...
         'solnum',11)
```

Plot on a boundary 6 for some time values with overlaid mesh:

```
postcrossplot(fem,2,6,'surfdata','ux','cont','on',...
              'surfedgestyle','k','refine',1,...
              'solnum',[1,4,7,11])
```

Plot along a line intersecting the geometry:

```
linpts = [-1 1;-1 1;0 1];
postcrossplot(fem,1,linpts,'lindata','u','npoints',100)
```

Same but with time extrusion:

```
postcrossplot(fem,1,linpts,'surfdata','u','npoints',100,...
              'camlight','on')
```

Plot along some connected edges:

```
postcrossplot(fem,1,[4 5 8 11],'lindata','t1x')
% Compare this with the postplot call
postplot(fem,'lindata','t1x','edl',[4 5 8 11],'linbar','on',...
         'geom','off')
```

Point plot on `n` points in geometry:

```
n = 30;
pts = [linspace(-1,1,n);linspace(0,1,n);linspace(0,1,n)];
postcrossplot(fem,0,pts,'pointdata','u')
```

2D Examples

Time-dependent problem (Heat equation)

```
clear fem
fem.geom = geomcsg({rect2});
fem.mesh = meshinit(fem);
fem.equ.c = 1; fem.equ.f = 1; fem.equ.da = 1;
fem.bnd.h = 1;
fem.shape = 2;
fem.xmesh = meshextend(fem);
fem.sol = femtime(fem,'tlist',0:0.01:0.1);
```

Plot solution along the diagonal for all time-steps:

```
postcrossplot(fem,1,[0 1;1 0],'lindata','u')
```

Plot solution at time step 4 in several parallel cross sections:

```
postcrossplot(fem,1,[0 1;1 0],'lindata','u*x','solnum',4,...
              'spacing',5,'lincolor','r')
```

Plot along three boundaries for the first five time steps:

```
postcrossplot(fem,1,[1 2 3],'lindata','ux','cont','on',...
              'solnum',1:5)
```

Same but with time-extrusion:

```
postcrossplot(fem,1,[1 2 3],'surfdata','ux','cont','on',...
              'solnum',1:5)
```

Make point plot of square of solution on three points in geometry:

```
pts = [0.2 0.3 0.6;0.1 0.7 0.2];
postcrossplot(fem,0,pts,'pointdata','u^2')
```

Compatibility

In FEMLAB 3.0a, extrusion plots, i.e., when plotting for several solutions (`Solnum` or `T` is a vector), `cdim` is 1, and `Surfdata` is used, can only be made for plots where the extrusion axis represents the solution. Extrusions cannot be made between parallel lines for cross-section line plots. Also, all line plots are all plotted in the x-y plane, also for several solutions and for several parallel cross-section lines.

The property `Variables` has been renamed `Const` in FEMLAB 2.3.

See Also

`postplot`, `postinterp`

Purpose Plot a post data structure.

Syntax

```
postdataplot(pd,...)
h = postdataplot(pd,...)
```

Description `postdataplot(pd,...)` displays a plot of a *post data* structure, typically returned by `posteval`. The function supports plotting postdata structures with element dimension 1 or 2, corresponding to the `posteval` property `Edim`.

`h = postdataplot(pd,...)` additionally returns handles to the plotted objects.

The function `postdataplot` accepts the following property/values:

TABLE 1-107: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Colorbar`	`off` | `on`	`on`	Display a color legend
`Colormap`	color table	`Rainbow`	The color table
`Colormapstyle`	`auto` | `reverse`	`auto`	Color table style
`Edgestyle`	`flat` | `interp` | `none` | `bg` | `bginv` | colorspec	`none`	Surface edge style
`Facestyle`	`flat` | `interp` | `none` | `bg` | `bginv` | colorspec	`interp`	Surface face style

In addition, the common plotting properties listed under `femplot` are available.

The notation colorspec in the value column denotes a *color specification*. See `postplot` for details.

Compatibility The function `postdataplot` was introduced in COMSOL Multiphysics 3.4.

See Also `posteval`

Purpose

Evaluate expressions in subdomains, boundaries, edges or vertices.

Syntax

```
[v1,v2,...,vn] = posteval(fem,e1,e2,...,en,...)
```

Description

`[v1,v2,...,vn] = posteval(fem,e1,e2,...,en,...)` returns values `v1,v2,...,vn` of the expressions `e1,e2,...,en`. The expressions can be evaluated on any domain type: subdomain, boundary, edge, and vertex, using one or several solutions.

The values `vi` are *post data*, a structure with fields `p`, `t`, `q`, `d`, and `elind`. The field `p` contains node point coordinate information. The number of rows in `p` is the number of space dimensions. The field `t` contains the indices to columns in `p` of a simplex mesh, each column in `t` representing a simplex. The field `q` contains the indices to columns in `p` of a quadrilateral mesh, each column in `q` representing a quadrilateral. The field `d` contains data values. The columns in `d` correspond to node point coordinates in columns in `p`. There is one row in `d` for each solution (see the properties `Solnum` and `T` below). The data contains the real part of complex-valued expressions. The field `elind` contains indices to mesh elements for each point.

The string expressions can be any COMSOL Multiphysics expressions involving variables, in particular *application mode variables.*

The function `posteval` accepts the following property/values:

TABLE 1-108: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Bpoint`	double matrix		Local coordinates for quadrilateral and block elements
`Complexfun`	`off` | `on`	`on`	Use complex-valued functions with real input
`Const`	cell array		Definition of constants
`Cont`	`off` | `on` | `internal`	`off`	Smoothing
`Dl`	integer vector or cell array of integer vectors	all domains	Domain lists
`Edim`	integer	full	Element dimension
`Frame`	integer	spatial frame	Coordinate frame
`Geomnum`	positive integer	1	Geometry number

TABLE 1-108: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Matherr`	`off` | `on`	`off`	Error for undefined operations
`Phase`	scalar	0	Phase angle
`Prpoint`	double matrix		Local coordinates for prism elements
`Refine`	integer | `auto`	3	Refinement of element for evaluation points
`Solnum`	integer vector | `all` | `end`	See below	Solution number
`Spoint`	double matrix		Local coordinates for simplex elements
`T`	double vector		Time for evaluation
`Triangulate`	`off` | `on`	`off`	Divide quad elements into triangles
`U`	solution object or vector	`fem.sol` or 0	Solution for evaluation

The property `Refine` constructs evaluation points by making a regular refinements of each element. Each mesh edge is divided into `Refine` equal parts. If `auto` is used, an automatic refinement value is computed internally and used, which depends on the maximum element order and the number of elements evaluated on. This value is most useful in `postplot`.

Use the properties `Spoint`, `Bpoint`, and `Prpoint` to specify arbitrary local element evaluation points for simplex elements (triangular, tetrahedral, and edge elements), quadrilateral/block elements, and prism elements, respectively. If you specify any of these properties, the fields `t` and `q` in the output postdata structure are empty, and the property `Cont` is neglected.

The property `Edim` decides which elements to evaluate on. Evaluation takes place only on elements with space dimension `Edim`. If not specified, `Edim=sdim` is used, where `sdim` is the space dimension of the geometry. For example, in a 3D model, if evaluation is done on edges (1D elements), `Edim` is `1`. Similarly, for boundary evaluation (2D elements), `Edim` is `2`, and for subdomain evaluation (3D elements), `Edim` is `3` (default in 3D).

The property `Dl` controls on which domains (subdomains, boundaries, etc.) evaluation should take place. If `Geomnum` is a vector, `Dl` must be a cell array of the same length as `Geomnum` containing domain lists for each geometry.

The property `Cont` controls if the post data is forced to be continuous on element edges. When `Cont` is set to `internal`, only elements not on interior boundaries are made continuous.

The expressions `e`*`i`* are evaluated for one or several solutions. Each solution generates an additional row in the `d` field of the post data output structure. The properties `Solnum` and `T` control what solutions are used for the evaluations. If `Solnum` is provided, the solution indicated by the indices provided with the `Solnum` property are used. It `T` is provided, solutions are interpolated The property `T` can only be used for time dependent solutions. If nether Solnum nor `T` is provided, a single solution is evaluated. For parametric and time-dependent solutions, the final solution is used. For eigenvalue solution the first solution is used.

For time-dependent problems, the variable `t` can be used in the expressions `e`*`i`*. The value of `t` is the interpolation time when the property `T` is provided, and the time for the solution, when Solnum is used. Similarly, `lambda` and the parameter are available as eigenvalues for eigenvalue problems and as parameter value for parametric problems, respectively.

When the property `Phase` is used, the solution vector is multiplied with `exp(i*phase)` before evaluating the expression.

Example

Solve Poisson's equation on two rectangles and evaluate the solution on one of them and the negative solution on the other.

```
clear fem
fem.geom = square2(1,'pos',[0 -1])+square2;
fem.mesh = meshinit(fem);
fem.equ.c = 1; fem.equ.f = 1;
fem.bnd.h = 1;
fem.equ.expr = {'uu' {'u','-u'}};
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem);
pd = posteval(fem,'uu');
```

Compatibility

The properties `Spoint` and `Bpoint` was re-introduced and `Prpoint` was introduced in COMSOL Multiphysics 3.2a.

The FEMLAB 3.0 output type has been changed to a structure containing data suitable for further postprocessing. The new output format is incompatible with FEMLAB 2.3 and earlier versions.

The properties `Context`, `Contorder`, `Posttype`, and `Spoint` are obsolete from FEMLAB 3.0.

The property `Variables` has been renamed `Const` in FEMLAB 2.3.

The properties `Bdl`, `Epoint`, `Sdl`, and `Tpoint`, are obsolete from FEMLAB 2.2. Use the `Dl` property to specify domain lists. The post data format has changed in FEMLAB 2.2 and later versions.

The variable name `lambda` introduced in FEMLAB 1.2 can introduce a variable name conflict for old models.

See Also `postdataplot`, `postglobaleval`, `postint`, `postinterp`, `postsum`

Purpose Shorthand command for streamline plot in 2D and 3D.

Syntax

```
postflow(fem,expr,...)
h = postflow(fem,expr,...)
```

Description `postflow(fem,expr,...)` plots a streamline plot for the expressions in the cell array `expr`. In 2D, `expr` has length 2, and in 3D, it has length 3. The function accepts all property/value pairs that `postplot` does. This command is just shorthand for the call

```
postplot(fem,'flowdata',expr,...
             'geom','on',...
             'axisequal','on',...)
```

`h = postflow(fem,expr,...)` additionally returns handles to the plotted handle graphics objects.

If you want to have more control over your streamline plot, use `postplot` instead of `postflow`.

See Also `postplot`, `postanim`, `postsurf`, `postcont`, `postlin`, `postarrow`, `postarrowbnd`, `postprinc`, `postprincbnd`, `postslice`, `postiso`, `posttet`

Purpose Evaluate globally defined expressions, such as solutions to ODEs.

Syntax

```
data = postglobaleval(fem,...)
data = postglobaleval(fem,expr,...)
```

Description `postglobaleval(fem,expr,...)` is the evaluation function for globally defined expressions, such as solution variables for ODEs and other space-independent equations.

The input `expr` contains the expressions to plot. It must be a cell array of strings. If omitted, the expressions in `fem.ode.dim` are evaluated.

`data = postglobaleval(fem,...)` returns a structure with fields `x`, `y`, and `legend`. The values can be plotted with

```
plot(data.x, data.y)
legend(data.legend)
```

Valid property/value pairs for `postglobaleval` are given in the following table.

TABLE 1-109: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Complexfun`	`off` | `on`	`on`	Use complex-valued functions with real input
`Const`	cell array		Definition of constants
`Matherr`	`off` | `on`	`off`	Error for undefined operations
`Phase`	scalar	`0`	Phase angle
`Solnum`	integer vector | `all` | `end`	`all`	Solution number
`T`	vector		Times for evaluation
`U`	solution object or vector	`fem.sol` or 0	Solution for evaluation

The property `Phase` is described in `posteval`.

Examples Example: Solve the Lotka-Volterra equations for two populations, *r* and *f*:

```
clear fem
fem.ode.dim={'r','f'};
fem.ode.f={'r*(1-2*f)-rt','-f*(3-r)-ft'};
fem.ode.init={'10','1'};
fem.ode.dinit={'0','0'};
fem.geom=solid1([0,1]);
fem.mesh = meshinit(fem);
fem.xmesh = meshextend(fem);
fem.sol = femtime(fem,'tlist',[0,1]);

% Evaluate 'r' and 'f' for all time steps
data = postglobaleval(fem);

% Evaluate 'r+f' and 'r*f' and plot the result
data = postglobaleval(fem,{'r+f','r*f'})
plot(data.x, data.y)
legend(data.legend)
```

Compatibility This function was introduced in COMSOL Multiphysics 3.2a.

See Also `postglobalplot`, `postinterp`

Purpose Plot globally defined expressions, such as solutions to ODEs.

Syntax

```
postglobalplot(fem,expr,...)
h = postglobalplot(fem,expr,...)
```

Description `postglobalplot(fem,expr,...)` is the plot function for globally defined expressions, such as solution variables for ODEs and space-independent equations.

The input `expr` contains the expressions to plot. It must be a string or a cell array of strings.

`h = postglobalplot(fem,expr,...)` additionally returns handles or a postdata structure (depending on the value of the property `Outtype`) to the plotted objects.

Valid property/value pairs for `postglobalplot` are given in the following table.

TABLE 1-110: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Complexfun`	`off` | `on`	`on`	Use complex-valued functions with real input
`Const`	cell array		Definition of constants
`Lincolor`	colorspec | cell array of colorspec	`cycle`	Line color
`Linewidth`	numeric	`0.5`	Line width
`Linlegend`	`off` | `on`	`off`	Color legend
`Linmarker`	marker specifier | cell array of marker specifiers	`none`	Line marker
`Linstyle`	symbol | cell array of symbols	`'-'`	Line style
`Linxdata`	string		Line x-axis expression
`Markersize`	integer	`6`	Size of markers
`Matherr`	`off` | `on`	`off`	Error for undefined operations
`Outtype`	`handle` | `postdata` | `postdataonly`	`handle`	Output type
`Phase`	scalar	`0`	Phase angle
`Solnum`	integer vector | `all` | `end`	`all`	Solution number

TABLE 1-110: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
T	vector		Times for evaluation
U	solution object or vector	`fem.sol` or 0	Solution for evaluation

In addition, the common plotting properties listed under `femplot` are available.

If Outtype is `'handle'`, `postglobalplot` returns a vector of handles to the plots. If Outtype is `'postdata'` or `'postdataonly'`, the function returns a post data structure. The post data structure has the same format as the output from `posteval`. When `'postdataonly'` is used, no plot is generated.

The notation colorspec in the value column denotes a *color specification*. See `postplot` for a description of this specification.

The property `Phase` is described in `posteval`.

Examples

Example: Solve the Lotka-Volterra equations for two populations r and f

```
clear fem
fem.ode.dim={'r','f'};
fem.ode.f={'r*(1-2*f)-rt','-f*(3-r)-ft'};
fem.ode.init={'10','1'};
fem.ode.dinit={'0','0'};
fem.geom=solid1([0,1]);
fem.mesh = meshinit(fem);
fem.xmesh = meshextend(fem);
fem.sol = femtime(fem,'tlist',[0,1]);

% Plot the solutions r and f with legend
postglobalplot(fem,{'r','f'},'linlegend','on');

% Plot the r population versus the f population
postglobalplot(fem,'r','linxdata','f')
```

Compatibility

This function was introduced in COMSOL Multiphysics 3.2a.

See Also

`postcrossplot`, `postinterp`

Purpose Extract Gauss points and Gauss point weights.

Syntax

```
gp = postgp(type,order)
[gp,gpw] = postgp(type,order)
```

Description `postgp(type,order)` returns the gauss points of order `order` for an element of type `type`.

`[gp,gpw] = postgp(type,order)` additionally returns the Gauss point weights.

The Gauss points and their weights are the ones used in `postint` when computing integrals.

The input `type` must be one of the following: `vtx`, `edg`, `tri`, `quad`, `tet`, `prism`, or `hex` corresponding to a vertex element, an edge element, a triangular element, a quadrilateral element, a tetrahedral element, a prism element, and a hexahedral element, respectively.

Examples

```
% The second order Gauss points for a triangular element
gp = postgp('tri',2);

% The third order Gauss points and their weights for a hexahedral
% element
[gp,gpw] = postgp('hex',3);
```

Compatibility This function was introduced in COMSOL Multiphysics 3.2a.

See Also `posteval`, `postint`

Purpose Integrate expressions in domains with arbitrary space dimension.

Syntax `[v1,v2,...,vn] = postint(fem,e1,e2,...,en,...)`

Description `[v1,i2,...,vn] = postint(fem,e1,e2,...,en,...)` returns the integrals `v1,v2,...,vn` of the expressions `e1,...,en`. The integrals can be evaluated on any domain type: subdomain, boundary, edge, and vertex, using one or several solutions. When the several solutions are provided, each `vi` is a vector with values corresponding to the solutions.

The expressions that are integrated can be expressions involving variables, in particular *application mode variables*.

`postint` accepts the following property/value pairs:

TABLE 1-111: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Complexfun`	`off` | `on`	`on`	Use complex-valued functions with real input
`Const`	cell array		List of assignments of constants
`Dl`	integer vector	all domains	Domain list
`Edim`	integer	full	Element dimension
`Frame`	string	spatial frame	Coordinate frame
`Geomnum`	positive integer	1	Geometry number
`Intorder`	positive integer	4	Integration order
`Matherr`	`off` | `on`	`off`	Error for undefined operations
`Phase`	integer vector	0	Phase angle
`Recover`	`off` | `ppr` | `pprint`	`off`	Accurate derivative recovery
`Solnum`	integer vector | `all` | `end`	See below	Solution numbers
`T`	double vector	See below	Time for evaluation
`U`	solution object or vector	`fem.sol` or 0	Solutions for evaluation

The expressions `ei` are integrated for one or several solutions. Each solution generates an element in the output vectors `vi`. The properties `Solnum` and `T` control what solutions are used for the evaluations. If `Solnum` is provided, the solution indicated by the indices provided with the `Solnum` property are used. It `T` is provided, solutions are interpolated The property `T` can only be used for time dependent solutions. If nether Solnum nor `T` is provided, a single solution is evaluated. For parametric and time-dependent solutions, the final solution is used. For eigenvalue solution the first solution is used.

For time-dependent problems, the variable `t` can be used in the expressions `e`*`i`*. The value of `t` is the interpolation time when the property `T` is provided, and the time for the solution, when Solnum is used. Similarly, `lambda` and the parameter are available as eigenvalues for eigenvalue problems and as parameter value for parametric problems, respectively.

Examples

Compute the integral of the solution to Poisson's equation on the unit disk using weak constraints. Use weak constraint to obtain accurate flux.

```
clear fem
fem.dim = {'u' 'lm'};
fem.geom = circ2;
fem.mesh = meshinit(fem);
fem.shape={'shlag(2,''u'')' 'shlag(2,''lm'')'};
fem.equ.c = {{1 0}};
fem.equ.f = {{1 0}};
% make shape function for u active on subdomain
fem.equ.shape={1};
fem.bnd.weak = {{'test(u)*lm' 'test(lm)*(-u)'}};
% make shape functions for u and lm active on boundary
fem.bnd.shape={[1 2]};
fem.solform = 'general';
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem);
postint(fem,'u')
```

Verify that the integral of the source term in Poisson's equation on the unit disk cancels the integral of the flux over the boundary. To have access to the variables `f1` and `ncu1`, you must use the General solution form.

```
postint(fem,'f1')
postint(fem,'lm','edim',1)
```

You can also use the variable `ncu` to compute the flux, but it is much less accurate.

```
postint(fem,'-ncu1','edim',1)
```

Compatibility

The properties `Context`, `Cont`, and `Contorder` are obsolete from FEMLAB 3.0.

The property `Variables` has been renamed `Const` in FEMLAB 2.3.

See Also `posteval`, `postsum`

Purpose

Evaluate expressions in arbitrary points.

Syntax

```
[v1,v2,...,vn,pe] = postinterp(fem,e1,e2,...,en,xx,...)
[pio,pe] = postinterp(fem,xx,...)
[v1,v2,...,vn] = postinterp(fem,e1,e2,...,en,pio,...)
```

Description

`[v1,v2,...,vn,pe] = postinterp(fem,e1,e2,...,en,xx,...)` returns the values `v1,v2,...,vn` of the expressions `e1,e2,...,en` in the points `xx`.

`[pio,pe] = postinterp(fem,xx,...)` computes a *PostInterp object* `pio`, which contains information about where the points `xx` are located.

`[v1,v2,...,vn] = postinterp(fem,e1,e2,...,en,pio,...)` returns the values `v1,v2,...,vn` of the expressions `e1,e2,...,en` in the points given by the PostInterp object `pio`.

The columns of the matrix `xx` are the coordinates for the evaluation points. If the number of rows in `xx` equals the space dimension, then `xx` are global coordinates, and the property `Edim` determines the dimension in which the expressions are evaluated. For instance, `Edim=2` means that the expressions are evaluated on boundaries in a 3D model. If `Edim` is less than the space dimension, then the points in `xx` are projected onto the closest point on a domain of dimension `Edim`. If, in addition, the property `Dom` is given, then the closest point on domain number `Dom` in dimension `Edim` is used.

If the number of rows in `xx` is less than the space dimension, then these coordinates are parameter values on a geometry face or edge. In that case, the domain number for that face or edge must be specified with the property `Dom`.

The expressions that are evaluated can be expressions involving variables, in particular *application mode variables*.

The matrices `v1,v2,...,vn` are of the size `k`-by-`size(xx,2)`, where `k` is the number of solutions for which the evaluation is carried out, see below. The value of expression `ei` for solution number `j` in evaluation point `xx(:,m)` is `vi(j,m)`.

The vector `pe` contains the indices `m` for the evaluation points `xx(:,m)` that are outside the mesh, or, if a domain is specified, are outside that domain.

postinterp accepts the following property/value pairs:

TABLE 1-112: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
Blocksize	positive integer	1000	Block size
Complexfun	off | on	on	Use complex-valued functions with real input
Const	cell array		List of assignments of constants
Dom	positive integer		Domain number
Edim	0 | 1 | 2 | 3	size(xx,1)	Element dimension for evaluation
Ext	number between 0 and 1	0.1	Extrapolation distance
Frame	string	spatial frame	Coordinate frame
Geomnum	positive integer	1	Geometry number
Matherr	off | on	off	Error for undefined operations
Mcase	non-negative integer		Mesh case
Phase	scalar	0	Phase angle
Recover	off | ppr | pprint	off	Accurate derivative recovery
Solnum	integer vector | all | end	See below	Solution numbers
T	double vector	See below	Time for evaluation
U	solution object | solution vector | scalar	fem.sol or 0	Solutions for evaluation

The properties Blocksize and Const are described in assemble.

The property Ext determines how far the extrapolation reaches. A positive value Ext means that for points outside the mesh, the evaluation is carried out by extrapolation from the nearest mesh element, provided that the distance to the mesh element is at most ext times the element diameter, roughly. Other (more distant) points outside the mesh give the value NaN in the value matrices v*i*.

The property Matherr is described in `femsolver`.

If the property `U` does not specify the mesh case number, it is given by the property `Mcase`. The default is the mesh case that has the greatest number of degrees of freedom.

The property `Phase` is described in `posteval`.

The property `U` specifies the solution for which the evaluation is carried out. If `U` is not specified, then `it` is taken from `fem.sol` if it exists; otherwise `it` is the zero vector.

The expressions `ei` are interpolated for one or several solutions. The properties `Solnum` and `T` control what solutions are used for the evaluations. If `Solnum` is provided, the solution indicated by the indices provided with the `Solnum` property are used. It `T` is provided, solutions are interpolated at the given times. The property `T` can only be used for time dependent solutions. If neither Solnum nor `T` is provided, a single solution is evaluated. For parametric and time-dependent solutions, the final solution is used. For eigenvalue solution the first solution is used.

For time-dependent problems, the variable `t` can be used in the expressions ei. The value of `t` is the interpolation time when the property `T` is provided, and the time for the solution, when `Solnum` is used. Similarly, `lambda` and the parameter are available as eigenvalues for eigenvalue problems and parameter value for parametric problems, respectively.

A subsequent evaluation with `[v1,v2,...,v`*n*`] = postinterp(fem,e1,e2,...,e`*n*`,pio,...)` is faster than using `xx` instead of `pio`. In this form of the call, only the properties `Const`, `Phase`, `Solnum`, `T`, and `U` are used.

Compatibility

The properties `Context`, `Cont`, and `Contorder` are obsolete from FEMLAB 3.0.

In FEMLAB 3.0, the interpolation structure is as a Java object.

The property `Variables` has been renamed `Const` in FEMLAB 2.3.

The syntax and capabilities of this function has changed since FEMLAB 2.1.

See Also

`posteval`

Purpose

Shorthand command for isosurface plot in 3D.

Syntax

```
postiso(fem,expr,...)
h = postiso(fem,expr,...)
```

Description

`postiso(fem,expr,...)` plots an isosurface plot for the expression `expr`. The function accepts all property/value pairs that `postplot` does. This command is just shorthand for the call

```
postplot(fem,'isodata',expr,...
             'isobar','on',...
             'geom','on',...
             'axisequal','on',...)
```

`h = postiso(fem,expr,...)` additionally returns handles to the plotted handle graphics objects.

If you want to have more control over your isosurface plot, use `postplot` instead of `postiso`.

See Also

`postplot`, `postanim`, `postsurf`, `postcont`, `postlin`, `postarrow`, `postarrowbnd`, `postflow`, `postprinc`, `postprincbnd`, `postslice`, `posttet`

Purpose

Shorthand command for line plot in 1D, 2D and 3D.

Syntax

```
postlin(fem,expr,...)
h = postlin(fem,expr,...)
```

Description

`postlin(fem,expr,...)` generates a line plot for the expression `expr`. The function accepts all property/value pairs that `postplot` does. In 1D, this command is just shorthand for the call

```
postplot(fem,'liny',expr,...
             'linstyle','bginv',...)
```

and in 2D, it is shorthand for

```
postplot(fem,'lindata',expr,...
             'linz',expr,...
             'linbar','on',...
             'axisequal','on',...)
```

and in 3D, it is shorthand for

```
postplot(fem,'lindata',expr,...
             'linbar','on',...
             'axisequal','on',...)
```

`h = postlin(fem,expr,...)` additionally returns handles to the plotted handle graphics objects.

If you want to have more control over your line plot, use `postplot` instead of `postlin`.

See Also

`postplot`, `postanim`, `postsurf`, `postcont`, `postarrow`, `postarrowbnd`, `postflow`, `postprinc`, `postprincbnd`, `postslice`, `postiso`, `posttet`

Purpose

Compute maximum value of an expression.

Syntax

```
m = postmax(fem,expr,...)
[m,p] = postmax(fem,expr,...)
```

Description

`m = postmax(fem,expr,...)` returns the maximum value of the expression `expr`. The function accepts all property/value pairs that `posteval` does, except `cont`. In addition, the following property/value pairs are accepted:

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Out`	`all` | `sollist`	`all`	Return min over all solutions, or min per solution, specified with solnum or t
`Useinf`	`on` | `off`	`on`	Allow infinity to be maximum value

`[m,p] = postmax(fem,expr,...)` additionally returns the `sdim`-by-1 matrix `p` containing the coordinate for which the maximum value occurs, where `sdim` is the space dimension of the geometry.

Note that the property `Refine` (see `posteval`) specifies the refinement used for finding the element in which the maximum value occurs. This element is then refined further to find the maximum value within the element. Therefore, the coordinate for which the maximum value of `expr` is attained, is not necessarily a node in the mesh.

Cautionary

When `expr` is evaluated to complex numbers, the real part is used in the maximum value calculation.

See Also

`posteval`, `postmin`

Purpose Compute minimum value of an expression.

Syntax

```
m = postmin(fem,expr,...)
[m,p] = postmin(fem,expr,...)
```

Description `m = postmin(fem,expr,...)` returns the minimum value of the expression `expr`. The function accepts all property/value pairs that `posteval` does, except `cont`. In addition, the following property/value pairs are accepted:

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Out`	`all` | `sollist`	`all`	Return min over all solutions, or min per solution, specified with solnum or t
`Useinf`	`on` | `off`	`on`	Allow -infinity to be minimum value

`[m,p] = postmin(fem,expr,...)` additionally returns the `sdim`-by-1 matrix `p` containing the coordinate for which the minimum value occurs, where `sdim` is the space dimension of the geometry.

Note that the property `Refine` (see `posteval`) specifies the refinement used for finding the element in which the minimum value occurs. This element is then refined further to find the maximum value within the element. Therefore, the coordinate for which the minimum value of `expr` is attained, is not necessarily a node in the mesh.

Cautionary When `expr` is evaluated to complex numbers, the real part is used in the minimum value calculation.

See Also `posteval`, `postmax`

Purpose Postprocessing animation function.

Syntax

```
postmovie(fem,...)
postmovie({fem1,fem2,fem3,...},...)
M = postmovie(fem,...) % MATLAB only
```

Description `postmovie(fem,...)` is the general solution animation function. It supports all property/value pairs that `postplot` supports, and in addition to that, it supports a set of property/value pairs that is exclusive for animation.

The input `fem` must be an FEM structure or a cell array of FEM structures. When it is a cell array, the properties `solnum` and `t` must, if specified, be cell arrays of the same size as `fem`. In this case, the FEM structures must have the same solution type, for example, all time-dependent solutions.

`M = postmovie(fem,...)` additionally returns a matrix in the MATLAB movie format.

The command can generate a sequence of image files containing all images in the movie. In addition, the command can generate an AVI movie file.

Valid property/value pairs for the `postmovie` function are given in the following table. In addition, all `postplot` parameters are supported and are passed to `postplot`. See the entry on `postplot` for a description of the post data formats.

TABLE 1-113: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Aviautoplay`	`on` | `off`	`on`	In MATLAB on Windows, try to launch program associated with AVI-extension and play generated AVI-movie
`Avicompression`	string		Compression used for AVI-movie
`Aviquality`	integer between 0 and 100	`75`	Quality of AVI-movie
`Filename`	string		Output file name
`Filetype`	`avi` | `jpg` | `tiff` | `png` | `gif` | `animgif`	`avi`	Output file type
`Fps`	integer	`12`	Frames per second
`Height`	integer	`480`	Height of image/movie files
`Repeat`	integer	`5`	Number of repeats

TABLE 1-113: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Reverse`	`on` | `off`	`off`	Make movie backwards
`Resol`	integer	`150`	Resolution
`Solnum`	integer vector | `all` | `end`	`all`	Solution numbers
`Statfunctype`	string	`full` | `half` | `linear`	Static plot function
`Statnframes`	integer	`11`	Number of frames in animation of static solution
`Width`	integer	`640`	Width of image/movie files

Example Run examples in `postplot`, and replace the `postplot` command with `postmovie`.

Cautionary When you are replaying a movie that has been stored in a matrix `M`, you should explicitly provide a figure handle to the `movie` command.

```
M = postmovie(fem,'tridata','u');
movie(gcf,M)
```

Otherwise the animation does not look good.

Compatibility The option `mov` for property `Filetype` as well as the properties `Qtrate`, `Qtqual`, and `Qtcomp` on Mac are removed in FEMLAB 3.0a. The option `avi` works on Mac. When using `mov`, it is translated internally to `avi`.

See Also `posteval`, `postplot`

Purpose Postprocessing plot function.

Syntax

```
postplot(fem,...)
h = postplot(fem,...)
```

Description `postplot(fem,...)` is the general solution plot function. It can display an FEM solution in several different ways. The command works for both 1D, 2D, and 3D geometries.

`h = postplot(fem,...)` additionally returns handles or postdata corresponding to the drawn axes objects. See properties `Out` and `Outtype`.

The function `postplot` accepts the following property/value pairs:

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Arrowbnd`		√	√	vector post spec		Boundary arrow data
`Arrowbndz`		√		post spec		Boundary arrow height data
`Arrowcolor`		√	√	colorspec	`red`	Subdomain arrow color
`Arrowcolorbnd`		√	√	colorspec	`blue`	Boundary arrow color
`Arrowcoloredg`			√	colorspec	`black`	Edge arrow color
`Arrowdata`		√	√	vector post spec		Arrow data
`Arrowedg`			√	vector post spec		Edge arrow data
`Arrowscale`		√	√	numeric	`auto`	Subdomain arrow scale
`Arrowscalebnd`		√	√	scalar	`auto`	Boundary arrow scale
`Arrowscaleedg`			√	scalar	`auto`	Edge arrow scale
`Arrowstyle`		√	√	`proportional` | `normalized`	`proportional`	Subdomain arrow style
`Arrowstylebnd`		√	√	`proportional` | `normalized`	`proportional`	Boundary arrow style
`Arrowstyleedg`			√	`proportional` | `normalized`	`proportional`	Edge arrow style
`Arrowtype`			√	`arrow` | `cone` | `arrow3d`	`cone`	Subdomain arrow type
`Arrowtypebnd`		√	√	`arrow` | `cone` | `arrow3d`	`cone`	Boundary arrow type
`Arrowtypeedg`			√	`arrow` | `cone` | `arrow3d`	`cone`	Edge arrow type

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Arrowxspacing`		√	√	number of arrows or vector specifying x-coordinates	15 (in 2D) 7 (in 3D)	Arrow x-spacing
`Arrowyspacing`		√	√	number of arrows or vector specifying y-coordinates	15 (in 2D) 7 (in 3D)	Arrow y-spacing
`Arrowz`		√		post spec		Arrow height data
`Arrowzspacing`			√	number of arrows or vector specifying z-coordinates	7 (in 3D)	Arrow z-spacing
`Bdl`		√	√	list of boundary numbers	`all`	Boundary list
`Bndmarker`		√	√	marker specifier	`square`	Boundary max/min marker type
`Bndmarkersize`		√	√	integer	6	Size of boundary max/ min markers
`Complexfun`	√	√	√	`off` | `on`	`on`	Use complex-valued functions with real input
`Const`	√	√	√	cell array		Definition of constants
`Cont`	√	√	√	`off` | `on` | `internal`	`off`	Make output continuous
`Contbar`		√		`off` | `on`	`on`	Show color legend for contours
`Contcolorbar`		√		`off` | `on`	`on`	Show color legend for contour colors
`Contcolordata`		√		post spec		Contour color data
`Contcolordlim`		√		`[`*`min max`*`]`	full range	Contour color limits
`Contcolormap`		√		color table		Color table for contour color data
`Contcolormapstyle`		√		`auto` | `reverse`	`auto`	Color table style for contour color data
`Contdata`		√		post spec		Contour data
`Contdlim`		√		`[`*`min max`*`]`	full range	Contour limits
`Contfill`		√		`off` | `on`	`off`	Filled contours

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Contlabel`		√		`off` | `on`	`off`	Show contour labels
`Contlevels`		√		number of levels or a vector specifying levels	20	Contour levels
`Contmap`		√		color table		Color table for contour plot
`Contmapstyle`		√		`auto` | `reverse`	`auto`	Color table style for contour plot
`Contrefine`		√		integer | `auto`	`auto`	Refinement of elements for contour plots
`Contstyle`		√		`bg` | `bginv` | `interp` | `cycle`	`interp`	Contour style
`Contz`		√		post spec		Contour height data
`Deformauto`		√	√	`on` | `off`	`on`	Auto scaling
`Deformbnd`		√	√	vector post spec		Deform data for boundaries
`Deformdata`		√	√	vector post spec		Deformation data
`Deformedg`			√	vector post spec		Deform data for edges
`Deformscale`	√	√	√	numeric		Deformation scale factor for subdomains
`Deformscalebnd`		√	√	numeric		Deformation scale factor for boundaries
`Deformscaleedg`			√	numeric		Deformation scale factor for edges
`Deformscalesub`			√	numeric		Deformation scale factor for subdomains
`Edgmarker`			√	marker specifier	`square`	Edge max/min marker type
`Edgmarkersize`			√	integer	6	Size of edge max/min markers
`Edl`			√	integer vector	`all`	Edge list
`Ellogic`	√	√	√	logical expression	1	Logical expression for elements to include

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Arrowxspacing`		√	√	number of arrows or vector specifying x-coordinates	15 (in 2D) 7 (in 3D)	Arrow x-spacing
`Arrowyspacing`		√	√	number of arrows or vector specifying y-coordinates	15 (in 2D) 7 (in 3D)	Arrow y-spacing
`Arrowz`		√		post spec		Arrow height data
`Arrowzspacing`			√	number of arrows or vector specifying z-coordinates	7 (in 3D)	Arrow z-spacing
`Bdl`		√	√	list of boundary numbers	`all`	Boundary list
`Bndmarker`		√	√	marker specifier	`square`	Boundary max/min marker type
`Bndmarkersize`		√	√	integer	6	Size of boundary max/min markers
`Complexfun`	√	√	√	`off` | `on`	`on`	Use complex-valued functions with real input
`Const`	√	√	√	cell array		Definition of constants
`Cont`	√	√	√	`off` | `on` | `internal`	`off`	Make output continuous
`Contbar`		√		`off` | `on`	`on`	Show color legend for contours
`Contcolorbar`		√		`off` | `on`	`on`	Show color legend for contour colors
`Contcolordata`		√		post spec		Contour color data
`Contcolordlim`		√		`[`*`min max`*`]`	full range	Contour color limits
`Contcolormap`		√		color table		Color table for contour color data
`Contcolormapstyle`		√		`auto` | `reverse`	`auto`	Color table style for contour color data
`Contdata`		√		post spec		Contour data
`Contdlim`		√		`[`*`min max`*`]`	full range	Contour limits
`Contfill`		√		`off` | `on`	`off`	Filled contours

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Contlabel`		√		`off` | `on`	`off`	Show contour labels
`Contlevels`		√		number of levels or a vector specifying levels	`20`	Contour levels
`Contmap`		√		color table		Color table for contour plot
`Contmapstyle`		√		`auto` | `reverse`	`auto`	Color table style for contour plot
`Contrefine`		√		integer | `auto`	`auto`	Refinement of elements for contour plots
`Contstyle`		√		`bg` | `bginv` | `interp` | `cycle`	`interp`	Contour style
`Contz`		√		post spec		Contour height data
`Deformauto`		√	√	`on` | `off`	`on`	Auto scaling
`Deformbnd`		√	√	vector post spec		Deform data for boundaries
`Deformdata`		√	√	vector post spec		Deformation data
`Deformedg`			√	vector post spec		Deform data for edges
`Deformscale`	√	√	√	numeric		Deformation scale factor for subdomains
`Deformscalebnd`		√	√	numeric		Deformation scale factor for boundaries
`Deformscaleedg`			√	numeric		Deformation scale factor for edges
`Deformscalesub`			√	numeric		Deformation scale factor for subdomains
`Edgmarker`			√	marker specifier	`square`	Edge max/min marker type
`Edgmarkersize`			√	integer	`6`	Size of edge max/min markers
`Edl`			√	integer vector	`all`	Edge list
`Ellogic`	√	√	√	logical expression	`1`	Logical expression for elements to include

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
Ellogictype	√	√	√	all | any | xor	all	Interpretation of logical expression
Flowback		√	√	off | on	on	Integrate streamlines backwards
Flowbar		√	√	off | on	on	Color legend for streamline color data
Flowcolor		√	√	interp | bg | bginv | colorspec	interp	Streamline color
Flowcolordata		√	√	post spec		Streamline color data
Flowcolordlim		√	√	[*min max*]	full range	Streamline color data limits, works only when flowtype is line
Flowdata		√	√	vector post spec		Streamline velocity field
Flowdens		√	√	none | uniform | velocity	none	Type of streamline density
Flowdist		√		numeric	0.05	Separating distance factor
Flowdist			√	numeric	0.15 or [0.05, 0.15]	Separating distance factor
Flowdistdel		√	√	numeric	0.2	Minimum Delaunay distance
Flowdistend		√	√	numeric	0.5	Terminating distance factor
Flowinitref		√	√	integer	1	Boundary element refinement
Flowdignoredist		√	√	numeric	0.5	Fraction of streamline length to ignore
Flowlines		√	√	integer	20	Number of streamlines
Flowlooptol		√	√	numeric	0.01	Streamline loop tolerance
Flowmap		√	√	color table	Rainbow	Color table for streamline color data
Flowmapstyle		√	√	auto | reverse	auto	Color table style for streamline color data

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Arrowxspacing`		√	√	number of arrows or vector specifying x-coordinates	15 (in 2D) 7 (in 3D)	Arrow *x*-spacing
`Arrowyspacing`		√	√	number of arrows or vector specifying y-coordinates	15 (in 2D) 7 (in 3D)	Arrow *y*-spacing
`Arrowz`		√		post spec		Arrow height data
`Arrowzspacing`			√	number of arrows or vector specifying z-coordinates	7 (in 3D)	Arrow *z*-spacing
`Bdl`		√	√	list of boundary numbers	`all`	Boundary list
`Bndmarker`		√	√	marker specifier	`square`	Boundary max/min marker type
`Bndmarkersize`		√	√	integer	6	Size of boundary max/min markers
`Complexfun`	√	√	√	`off` | `on`	`on`	Use complex-valued functions with real input
`Const`	√	√	√	cell array		Definition of constants
`Cont`	√	√	√	`off` | `on` | `internal`	`off`	Make output continuous
`Contbar`		√		`off` | `on`	`on`	Show color legend for contours
`Contcolorbar`		√		`off` | `on`	`on`	Show color legend for contour colors
`Contcolordata`		√		post spec		Contour color data
`Contcolordlim`		√		`[`*`min max`*`]`	full range	Contour color limits
`Contcolormap`		√		color table		Color table for contour color data
`Contcolormapstyle`		√		`auto` | `reverse`	`auto`	Color table style for contour color data
`Contdata`		√		post spec		Contour data
`Contdlim`		√		`[`*`min max`*`]`	full range	Contour limits
`Contfill`		√		`off` | `on`	`off`	Filled contours

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Contlabel`		√		`off` | `on`	`off`	Show contour labels
`Contlevels`		√		number of levels or a vector specifying levels	`20`	Contour levels
`Contmap`		√		color table		Color table for contour plot
`Contmapstyle`		√		`auto` | `reverse`	`auto`	Color table style for contour plot
`Contrefine`		√		integer | `auto`	`auto`	Refinement of elements for contour plots
`Contstyle`		√		`bg` | `bginv` | `interp` | `cycle`	`interp`	Contour style
`Contz`		√		post spec		Contour height data
`Deformauto`		√	√	`on` | `off`	`on`	Auto scaling
`Deformbnd`		√	√	vector post spec		Deform data for boundaries
`Deformdata`		√	√	vector post spec		Deformation data
`Deformedg`			√	vector post spec		Deform data for edges
`Deformscale`	√	√	√	numeric		Deformation scale factor for subdomains
`Deformscalebnd`		√	√	numeric		Deformation scale factor for boundaries
`Deformscaleedg`			√	numeric		Deformation scale factor for edges
`Deformscalesub`			√	numeric		Deformation scale factor for subdomains
`Edgmarker`			√	marker specifier	`square`	Edge max/min marker type
`Edgmarkersize`			√	integer	`6`	Size of edge max/min markers
`Edl`			√	integer vector	`all`	Edge list
`Ellogic`	√	√	√	logical expression	`1`	Logical expression for elements to include

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Ellogictype`	√	√	√	`all` | `any` | `xor`	`all`	Interpretation of logical expression
`Flowback`		√	√	`off` | `on`	`on`	Integrate streamlines backwards
`Flowbar`		√	√	`off` | `on`	`on`	Color legend for streamline color data
`Flowcolor`		√	√	`interp` | `bg` | `bginv` | colorspec	`interp`	Streamline color
`Flowcolordata`		√	√	post spec		Streamline color data
`Flowcolordlim`		√	√	`[`*`min max`*`]`	full range	Streamline color data limits, works only when flowtype is line
`Flowdata`		√	√	vector post spec		Streamline velocity field
`Flowdens`		√	√	`none` | `uniform` | `velocity`	`none`	Type of streamline density
`Flowdist`		√		numeric	`0.05`	Separating distance factor
`Flowdist`			√	numeric	`0.15` or `[0.05, 0.15]`	Separating distance factor
`Flowdistdel`		√	√	numeric	`0.2`	Minimum Delaunay distance
`Flowdistend`		√	√	numeric	`0.5`	Terminating distance factor
`Flowinitref`		√	√	integer	`1`	Boundary element refinement
`Flowdignoredist`		√	√	numeric	`0.5`	Fraction of streamline length to ignore
`Flowlines`		√	√	integer	`20`	Number of streamlines
`Flowlooptol`		√	√	numeric	`0.01`	Streamline loop tolerance
`Flowmap`		√	√	color table	`Rainbow`	Color table for streamline color data
`Flowmapstyle`		√	√	`auto` | `reverse`	`auto`	Color table style for streamline color data

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Flowmaxsteps`		√	√	integer	`400`	Maximum number of integration steps
`Flowmaxtime`		√	√	integer	`100`	Maximum integration time
`Flownormal`		√	√	`off` | `on`	`off`	Normalize velocity field
`Flowradiusdata`		√	√	post spec		Streamline radius data
`Flowrefine`		√	√	integer | `auto`	`auto`	Refinement of elements for streamline plots
`Flowsat`		√	√	numeric	`1.3`	Streamline saturation factor
`Flowseed`		√	√	1-by-sdim numeric		First start point for streamlines when flowdens is not none
`Flowstart`		√	√	`centers` | `ginput` | `edges` | cell array `{x y z}`	`centers`	Starting points for streamlines
`Flowstattol`		√	√	positive scalar	`1e-2`	Streamline stationary point stop tolerance
`Flowtol`		√		positive scalar	`1e-3`	Streamline integration tolerance
`Flowtol`			√	positive scalar	`1e-2`	Streamline integration tolerance
`Flowtuberes`		√	√	numeric	8	Tube resolution for streamlines
`Flowtubescale`		√	√	numeric		Tube radius scale for streamlines
`Flowtype`		√	√	`line` | `tube`	`tube` (line in 2D)	Type of streamline
`Flowz`		√		post spec		Streamline height data
`Frame`	√	√	√	string	spatial frame	Coordinate frame
`Geom`	√	√	√	`off` | `on`	`on`	Show geometry contours
`Geomcolor`	√	√	√	`bg` | `bginv` | colorspec	`bginv`	Geometry contours color
`Geomnum`	√	√	√	integer	1	Geometry number

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Isobar`			√	`on` | `off`	`on`	Isosurface color legend
`Isocolorbar`			√	`on` | `off`	`on`	Color legend for isosurfaces color data
`Isocolordata`			√	post spec		Isosurface color data
`Isocolordlim`			√	[*min max*]	full range	Isosurface color limits
`Isocolormap`			√	color table	`Rainbow`	Color table for isosurface color data
`Isocolormapstyle`			√	`auto` | `reverse`	`auto`	Color table style for isosurface color data
`Isodata`			√	Post spec		Isosurface data
`Isodlim`			√	[*min max*]	full range	Isosurface limits
`Isoedgestyle`			√	`flat` | `interp` | `none` | `bg` | `bginv` | colorspec	`none`	Isosurface edge style
`Isofacestyle`			√	`flat` | `interp` | `none` | `bg` | `bginv` | colorspec	`interp`	Isosurface face style
`Isolevels`			√	number of levels or a vector specifying levels	5	Isosurface levels
`Isomap`			√	color table		Color table for isosurface plot
`Isomapstyle`			√	`auto` | `reverse`	`auto`	Color table style for isosurface plot
`Isostyle`			√	`bg` | `bginv` | `color`	`color`	Isosurface style
`Linbar`	√	√	√	`off` | `on`	`on`	Line color legend
`Lindata`	√	√	√	post spec		Line data
`Lindlim`	√	√	√	[*min max*]	full range	Line limits
`Linmap`	√	√	√	color table		Line color table
`Linmapstyle`	√	√	√	`auto` | `reverse`	`auto`	Line color table style
`Linrefine`	√	√	√	integer| `auto`	`auto`	Refinement of elements for line plots
`Linstyle`	√	√	√	`flat` | `interp` | `none` | `bg` | `bginv` | colorspec	`interp`	Line style

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
Liny	√			post spec		Line y data
Linz	√	√		post spec		Line z data
Matherr	√	√	√	off | on	off	Error for undefined operations
Maxminbnd		√	√	post spec		Max/min marker on boundaries
Maxminedg			√	post spec		Max/min marker on edges
Maxminsub	√	√	√	post spec		Max/min marker on subdomains
Out	√	√	√	cell array of strings | all	all	Output
Outtype	√	√	√	handle | postdata | postdataonly	handle	Output type
Partatol		√	√	numeric vector		Absolute tolerances for particle tracing
Partbar		√	√	off | on	on	Show color legend for particle tracing
Partbnd		√	√	stick | disappear	stick	Particle point boundary behavior
Partcolordata		√	√	post spec		Particle tracing line color
Partdropfreq		√	√	numeric		Time between each particle release
Partdroptimes		√	√	numeric vector		Time values to release particles
Partedgetol		√	√	numeric | auto	0.001	Edge tolerance for particle tracing
Partdata		√	√	cell-array of strings		Particle force
Parthmax		√	√	numeric | auto		Maximum time step for particle tracing
Parthstart		√	√	numeric | auto		Initial time step for particle tracing
Partlinecolor		√	√	colorspec	blue	Particle tracing line color

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
Partmap		√	√	colortable		Color table for particle tracing
Partmapstyle		√	√	auto | reverse	auto	Color table style for particle tracing
Partmass		√	√	string	1	Particle mass
Partmaxsteps		√	√	integer | auto	1000	Maximum number of steps for particle tracing
Partplotas		√	√	lines | points | both | along	lines	Particle tracing plot type
Partpointcolor		√	√	colorspec	red	Particle tracing point color
Partpointscale		√	√	numeric		Point radius scale
Partradiusdata			√	post spec		Particle tracing tube radius
Partres		√	√	integer	5	Resolution of pathline for particle tracing
Partrtol		√	√	numeric		Relative tolerance for particle tracing
Partstart		√	√	numeric matrix, cell array of double vectors, or cell array of property/values to postcoord		Start points for particle tracing
Partstatic		√	√	off | on	off	Use instantaneous flow field
Partstatictend		√	√	numeric | auto	auto	End time for stationary flows
Parttstart		√	√	numeric | auto		Initial time for particle tracing
Parttuberes		√	√	numeric	8	Tube resolution
Parttubescale		√	√	numeric		Tube radius scale
Parttvar		√	√	string		Particle integration time variable name
Partvelstart		√	√	cell-array of strings	zero velocity	Initial velocity for particle tracing

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Partvelvar`		√	√	cell-array of strings		Particle velocity variable names
`Phase`	√	√	√	scalar	0	Phase angle
`Princbnd`			√	vector post spec		Boundary principal plot data
`Princcolor`		√	√	colorspec	`blue`	Subdomain principal plot color
`Princcolorbnd`			√	colorspec	`blue`	Boundary principal plot color
`Princdata`		√	√	vector post spec		Subdomain principal plot data
`Princscale`		√	√	numeric	`auto`	Subdomain principal plot scale
`Princscalebnd`			√	scalar	`auto`	Boundary principal plot scale
`Princstyle`		√	√	`proportional` | `normalized`	`proportional`	Subdomain principal plot style
`Princstylebnd`			√	`proportional` | `normalized`	`proportional`	Boundary principal plot style
`Princtype`			√	`arrow` | `cone` | `arrow3d`	`cone`	Subdomain principal plot type
`Princtypebnd`			√	`arrow` | `cone` | `arrow3d`	`cone`	Boundary principal plot type
`Princxspacing`		√	√	number of arrows or vector specifying x-coordinates	8 (in 2D) 5 (in 3D)	Arrow x-spacing for subdomain principal plot
`Princyspacing`		√	√	number of arrows or vector specifying y-coordinates	8 (in 2D) 5 (in 3D)	Arrow y-spacing for subdomain principal plot
`Princz`		√		post spec		Subdomain principal plot height data
`Princzspacing`			√	number of arrows or vector specifying z-coordinates	8 (in 3D)	Arrow z-spacing for subdomain principal plot
`Recover`	√	√	√	`off` | `ppr` | `pprint`	`off`	Accurate derivative recovery

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
`Refine`	√	√	√	integer | `auto`	`auto`	Refinement of elements for all plot types
`Sdl`	√	√	√	list of subdomain numbers	`all`	Subdomain list
`Slicebar`			√	`off` | `on`	`on`	Show color legend for slice plot
`Slicedata`			√	post spec		Slice plot data
`Slicedlim`			√	`[`*min max*`]`	full range	Slice plot limits
`Sliceedgestyle`			√	`flat` | `interp` | `none` | `bg` | `bginv` | colorspec	`none`	Slice plot edge style
`Slicefacestyle`			√	`flat` | `interp` | `none` | `bg` | `bginv` | colorspec	`interp`	Slice plot face style
`Slicemap`			√	color table		Color table for slice plot
`Slicemapstyle`			√	`auto` | `reverse`	`auto`	Color table style for slice plot
`Slicerefine`			√	integer | `auto`	`auto`	Refinement of elements for slice plots
`Slicexspacing`			√	number of slices or vector specifying x-coordinates	`5`	Slice plot x-positions
`Sliceyspacing`			√	number of slices or vector specifying y-coordinates	`[]`	Slice plot y-positions
`Slicezspacing`			√	number of slices or vector specifying z-coordinates	`[]`	Slice plot z-positions
`Solnum`	√	√	√	integer | `end`	`1`	Solution number
`Submarker`	√	√	√	marker specifier	`square`	Subdomain max/min marker type
`Submarkersize`	√	√	√	integer	`6`	Size of subdomain max/min markers
`T`	√	√	√	scalar		Time for evaluation

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
Tetbar			√	off | on	on	Show color legend for subdomain plot
Tetdata			√	Post spec		Subdomain plot data
Tetdlim			√	[*min max*]	full range	Subdomain plot limits
Tetedgestyle			√	flat | interp | none | bg | bginv | colorspec	none	Subdomain plot edge style
Tetfacestyle			√	flat | interp | none | bg | bginv | colorspec	interp	Subdomain plot face style
Tetkeep			√	number between 0 and 1	1	Fraction of elements to keep
Tetkeeptype			√	min | max | random	random	Which elements to keep
Tetmap			√	color table		Subdomain plot color table
Tetmapstyle			√	auto | reverse	auto	Subdomain plot color table style
Tetmaxmin			√	on | off	off	Show subdomain plot max/min markers
Tetrefine			√	integer | auto	auto	Refinement of elements for subdomain plots
Tribar		√	√	off | on	off	Surface color legend
Tridata		√	√	post spec		Surface data
Tridlim		√	√	[*min max*]	full range	Surface limits
Triedgestyle		√	√	flat | interp | none | bg | bginv | colorspec	none	Surface edge style
Trifacestyle		√	√	flat | interp | none | bg | bginv | colorspec	interp	Surface face style
Trimap		√	√	color table		Surface color table
Trimapstyle		√	√	auto | reverse	auto	Surface color table style
Trimaxmin		√	√	on | off	off	Show surface max/min markers

TABLE 1-114: VALID PROPERTY/VALUE PAIRS

PROPERTY	1D	2D	3D	VALUE	DEFAULT	DESCRIPTION
Trirefine		√	√	integer | auto	auto	Refinement of elements for surface plots
Triz		√		post spec		Triangle height
U	√	√	√	solution vector	fem.sol.u	Solution for evaluation

The properties Out and Outtype control the format of the output h when the syntax h = postplot(fem,...) is used.

If Out is 'all' (default), output corresponding to all plotted objects are returned. The property Out can also be a cell array containing any of the strings 'geom', 'slice', 'iso', 'tet', 'tri', 'cont', 'lin', 'flow', 'partline', 'partpoint', 'arrow', 'arrowbnd', 'arrowedg', 'maxminsub', 'maxminbnd', 'maxminedg', and 'light'. These correspond to the plots made using the properties 'geom', 'slicedata', 'isodata', 'tetdata', and so on. When Out is a 1-by-*n* cell-array, the output h is a cell array of the same size, matching the strings in Out.

If Outtype is 'handle', handle-graphics handles to the plots (as specified with Out) are returned in a vector of handles if Out is 'all', otherwise in a cell array.

If Outtype is 'postdata' or 'postdataonly', post data structures are returned in a cell array. The post data structures have the same format as the output from posteval. In addition, for particle tracing plots (using 'partdata'), the postdata structure contains the fields parttime and partvel, containing the time and velocity vector, respectively, associated to each point on the path. Also, for both particle tracing line plots and streamline plots, the fields startpts and endpts are included, containing the coordinates of each plotted line's start and end point, respectively. When 'postdataonly' is used, no plot is generated.

If the property Refine is specified, its value is used for all specified plot types; that is, it overrides all other properties ending with refine. See posteval.

The properties Princdata and Princbnd can either be specified as the names of the three principal stress or strain values in a 1-by-3 cell array of strings, for example, {'s1','s2','s3'}, or as the expressions for the value and then the vector components for each principal direction, for example, {'e1','e1x','e1y','e1z','e2','e2x','e2y','e2z','e3','e3x','e3y', 'e3z'}.

The camera properties (`Campos`, `Camtarget`, etc.) override the setting of `view` if both are used.

The notation colorspec in the value column denotes a *color specification*, that is a single letter string: `y`, `m`, `c`, `r`, `g`, `b`, `w`, and `k`, meaning yellow, magenta, cyan, red, green, blue, white, and black, respectively (also `'yellow'`, `'magenta'`, etc. are acceptable as color specification), or a 1-by-3 numeric array with RGB values.

Post spec is one of the following.

- A string with an expression to be evaluated. It can be a COMSOL Multiphysics expression involving variables, in particular *application mode variables*.
- A cell array, where the first entry is a string with an expression to be evaluated or a cell array of such strings, and the other entries are parameters passed to `posteval`.

Vector post spec is a cell array of Post specs.

The properties can be grouped in terms of what plot entity it refers to. The table below shows this grouping.

TABLE 1-115: PROPERTY GROUPING

PLOT ENTITY	1D	2D	3D	PROPERTY NAMES STARTING WITH
Arrows		Ω	Ω	`arrow`
Arrows		$\partial\Omega$	$\partial\Omega$	`arrowbnd`
Arrows			$\partial^2\Omega$	`arrowedg`
Contours		Ω		`cont`
Isosurfaces			Ω	`iso`
Lines	Ω	$\partial\Omega$	$\partial^2\Omega$	`lin`
Principal stress/strain plots		Ω	Ω	`princ`
Principal stress/strain plots			$\partial\Omega$	`princbnd`
Slices			Ω	`slice`
Particle tracing		Ω	Ω	`part`
Streamlines		Ω	Ω	`flow`
3D subdomains			Ω	`tet`
Surfaces		Ω	$\partial\Omega$	`tri`

The symbol $\partial\Omega$ indicates the boundary of the domain, and the symbol Ω indicates the domain itself. For the boundary of the domain, post data evaluated on the boundary is plotted. For the domain itself, post data evaluated on the domain is plotted.

Examples

3D Example

Solve the Poisson equation on a unit square:

```
clear fem
fem.geom = block3;
fem.mesh = meshinit(fem,'hmax',0.15);
fem.equ.c = 1; fem.equ.f = 1;
fem.bnd.h = {1 1 0 0 1 1};
fem.xmesh = meshextend(fem); fem.sol = femstatic(fem);
```

Plot the solution as a slice plot

```
postplot(fem,'slicedata','u')
```

Plot the solution using isosurfaces

```
postplot(fem,'isodata','u','scenelight','on')
```

Plot lighted cones showing the gradient together with geometry edges

```
postplot(fem,'arrowdata',{'ux','uy','uz'},...
             'geom','on','camlight','on','arrowtype','cone')
```

2D Example

Solve the Poisson equation on the unit circle

```
clear fem
fem.geom = circ2; fem.mesh = meshinit(fem);
fem.equ.c = 1; fem.equ.f = 1; fem.bnd.h = 1;
fem.xmesh = meshextend(fem); fem.sol = femstatic(fem);
```

Plot the solution as triangle color and z-height, and `u.*x` as contour lines

```
postplot(fem,'tridata','u','contdata','u*x',...
             'triz','u','contz','u');
```

Plot 30 streamlines for the field `(-uy, x*ux)` with color data `u`.

```
postplot(fem,'flowdata',{'-uy','x*ux'},...
             'flowlines',30,'flowcolordata','u')
```

Cautionary

Some default values have changed from FEMLAB 2.3 resulting in slightly different plots.

Compatibility

The properties `contlabel`, `context`, `contorder`, and `tetmarker` are no longer supported in FEMLAB 3.0.

Properties ending with `maxmin` are no longer supported. To plot max/min markers, use the properties `maxminsub`, `maxminbnd`, and `maxminedg` to plot markers on subdomains, boundaries, and edges, respectively.

The support for outputs from `posteval` as Post spec, has only a limited support and is not recommended.

The properties starting with `Princ` are added in FEMLAB 3.1.

The property `contlabel` is added in COMSOL Multiphysics 3.2a.

See Also

`geomplot`, `meshplot`, `postanim`, `postarrow`, `postarrowbnd`, `postcont`, `postcrossplot`, `posteval`, `postflow`, `postiso`, `postlin`, `postmovie`, `postprinc`, `postprincbnd`, `postslice`, `postsurf`, `posttet`

Purpose

Shorthand command for subdomain principal stress/strain plot in 2D and 3D.

Syntax

```
postprinc(fem,expr,...)
h = postprinc(fem,expr,...)
```

Description

`postprinc(fem,expr,...)` plots a subdomain principal stress/strain plot for the expressions in the cell array `expr`, which can have length 3 or 12. See property `Princdata` in `postplot`. The function accepts all property/value pairs that `postplot` does. This command is just shorthand for the call

```
postplot(fem,'princdata',expr,...
              'geom','on',...
              'axisequal','on',...)
```

`h = postprinc(fem,expr,...)` additionally returns handles to the plotted handle graphics objects.

If you want to have more control over your principal stress/strain plot, use `postplot` instead of `postprinc`.

See Also

`postplot`, `postanim`, `postsurf`, `postcont`, `postlin`, `postarrowbnd`, `postflow`, `postprincbnd`, `postslice`, `postiso`, `posttet`

Compatibility

This function was added in FEMLAB 3.1.

Purpose Shorthand command for boundary principal stress/strain plot in 2D and 3D.

Syntax

```
postprincbnd(fem,expr,...)
h = postprincbnd(fem,expr,...)
```

Description `postprincbnd(fem,expr,...)` plots a boundary principal stress/strain plot for the expressions in the cell array `expr`, which can have length 3 or 12. See property `Princdata` in `postplot`. The function accepts all property/value pairs that `postplot` does. This command is just shorthand for the call

```
postplot(fem,'princbnd',expr,...
             'geom','on',...
             'axisequal','on',...)
```

`h = postprincbnd(fem,expr,...)` additionally returns handles to the plotted handle graphics objects.

If you want to have more control over your principal stress/strain plot, use `postplot` instead of `postprincbnd`.

See Also `postplot`, `postanim`, `postsurf`, `postcont`, `postlin`, `postarrowbnd`, `postflow`, `postprinc`, `postslice`, `postiso`, `posttet`

Compatibility This function was added in FEMLAB 3.1.

Purpose

Shorthand command for slice plot in 3D.

Syntax

```
postslice(fem,expr,...)
h = postslice(fem,expr,...)
```

Description

`postslice(fem,expr,...)` plots a slice plot for the expression `expr`. The function accepts all property/value pairs that `postplot` does. This command is just shorthand for the call

```
postplot(fem,'slicedata',expr,...
             'slicebar','on',...
             'geom','on',...
             'axisequal','on',...)
```

`h = postslice(fem,expr,...)` additionally returns handles to the plotted handle graphics objects.

If you want to have more control over your slice plot, use `postplot` instead of `postslice`.

See Also

`postplot`, `postanim`, `postsurf`, `postcont`, `postlin`, `postarrow`, `postarrowbnd`, `postflow`, `postiso`, `postprinc`, `postprincbnd`, `posttet`

Purpose Compute the sum of expressions in nodes.

Syntax `[v1,v2,...,vn] = postsum(fem,e1,e2,...,en,...)`

Description `[v1,v2,...,vn] = postsum(fem,e1,e2,...,en,'nodes',order)` returns the sums `v1,v2,...,vn` of the expressions `e1,...,en` in the Lagrange nodes of order *order*. When order is equal to `'all'` or the property `'nodes'` is not given, the sums are taken over all nodes for which there exists a degree of freedom. The sums can be evaluated on any domain type: subdomain, boundary, edge, and vertex, using one or several solutions. When several solutions are provided, each `vi` is a vector with values corresponding to the solutions.

The expressions that are summed can be expressions involving variables, in particular application mode variables.

`postsum` accepts the following property/value pairs:

TABLE 1-116: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Blocksize`	positive integer	`1000`	Vectorization block size
`Complexfun`	`off` | `on`	`on`	Use complex-valued functions with real input
`Const`	cell array		List of assignments of constants
`Dl`	integer vector	all domains	Domain list
`Edim`	integer	full	Element dimension
`Geomnum`	positive integer	1	Geometry number
`Matherr`	`off` | `on`	`off`	Error for undefined operations
`Nodes`	`all` | positive integer	`all`	Nodes to sum
`Phase`	integer vector	0	Phase angle
`Recover`	`off` | `ppr` | `pprint`	`off`	Accurate derivative recovery
`Solnum`	integer vector | `all` | `end`	See below	Solution numbers

TABLE 1-116: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
T	double vector	See below	Time for evaluation
U	solution object or vector	`fem.sol` or 0	Solutions for evaluation

The expressions `e`*`i`* are summed for one or several solutions. Each solution generates an element in the output vectors `v`*`i`*. The properties `Solnum` and `T` control what solutions are used for the evaluations. If `Solnum` is provided, the solutions indicated by the indices provided with the `Solnum` property are used. If `T` is provided, solutions are interpolated. The property `T` can only be used for time-dependent solutions. If neither `Solnum` nor `T` is provided, a single solution is evaluated. For parametric and time-dependent solutions, the final solution is used. For eigenvalue solution the first solution is used.

For time-dependent problems, the variable `t` can be used in the expressions `e`*`i`*. The value of `t` is the interpolation time when the property `T` is provided and the time for the solution when `Solnum` is used. Similarly, `lambda` and the parameters are available as eigenvalues for eigenvalue problems and as parameter values for parametric problems, respectively.

Example

Calculate the sum of the node-wise constraint forces in all nodes at the boundary of a square domain for the solution to Poisson's equation.

```
fem.geom = rect2;
fem.mesh = meshinit(fem);
fem.equ.c = 1;
fem.equ.f = 1;
fem.bnd.h = 1;
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem,'reacf','on');
res = postsum(fem,'reacf(u)','edim',1);
```

See Also

`postint`, `posteval`

Purpose Shorthand command for surface plot in 2D and 3D.

Syntax

```
postsurf(fem,expr1,...)
postsurf(fem,expr1,expr2...)
h = postsurf(fem,...)
```

Description `postsurf(fem,expr1,expr2...)` plots a surface plot on subdomains in 2D colored according to the expression `expr1` and with height according to `expr2`. For a 3D model, it plots a colored surface plot on the boundaries, colored according to `expr1`. The function accepts all property/value pairs that `postplot` does. In 2D, this command is just shorthand for the call

```
  postplot(fem,'tridata',expr1,...
               'triz',expr2,...
               'tribar','on',...
               'axisequal','on',...)
```

and in 3D, it is shorthand for

```
  postplot(fem,'tridata',expr1,...
               'tribar','on',...
               'geom','on',...
               'axisequal','on',...)
```

`h = postsurf(fem,...)` additionally returns handles to the plotted handle graphics objects.

If you want to have more control over your surface plot, use `postplot` instead of `postsurf`.

Example Surface plot of the solution to the equation $-\Delta u = 1$ over the geometry defined by the L-shaped membrane. Use Dirichlet boundary conditions $u = 0$ on $\partial\Omega$.

```
  sq1 = square2(0,0,1);
  sq2 = move(sq1,0,-1);
  sq3 = move(sq1,-1,-1);
  clear fem
  fem.geom = sq1+sq2+sq3;
  fem.mesh = meshinit(fem);
  fem.equ.c = 1;
  fem.equ.f = 1;
  fem.bnd.h = 1;
  fem.xmesh = meshextend(fem);
  fem.sol = femstatic(fem);
  postsurf(fem,'u')
```

See Also `postplot`, `postanim`, `postcont`, `postlin`, `postarrow`, `postarrowbnd`, `postflow`, `postprinc`, `postprincbnd`, `postslice`, `postiso`, `posttet`

Purpose Shorthand command for subdomain plot in 3D.

Syntax

```
posttet(fem,expr,...)
h = posttet(fem,expr,...)
```

Description `posttet(fem,expr,...)` plots a subdomain plot for the expression `expr`. The function accepts all property/value pairs that `postplot` does. This command is just shorthand for the call

```
postplot(fem,'tetdata',expr,...
             'tetbar','on',...
             'geom','on',...
             'axisequal','on',...)
```

`h = postcont(fem,expr,...)` additionally returns handles to the plotted handle graphics objects.

If you want to have more control over your subdomain plot, use `postplot` instead of `posttet`.

See Also `postplot`, `postanim`, `postsurf`, `postcont`, `postlin`, `postarrow`, `postarrowbnd`, `postflow`, `postprinc`, `postprincbnd`, `postslice`, `postiso`

Purpose

Create interpolation file

Syntax

```
postwriteinterpfile(filename,x,data)
postwriteinterpfile(filename,x,y,data)
postwriteinterpfile(filename,x,y,z,data)
```

Description

`postwriteinterpfile(filename,x,data)` creates the file `filename` in the format of a 1D interpolation function. `x` and `data` must be vectors of equal length.

`postwriteinterpfile(filename,x,y,data)` creates the file `filename` in the format of a 2D interpolation function. `data` must have size (`length(x)`, `length(y)`).

`postwriteinterpfile(filename,x,y,z,data)` creates the file `filename` in the format of a 3D interpolation function. `data` must have size (`length(x)`, `length(y)`, `length(z)`).

Examples

To create a 1D interpolation function of x^2:

```
  x = 1:10;
  postwriteinterpfile('fun.txt',x,x.^2);
```

To create a 2D interpolation function of $x^2 + 2y$:

```
  x = 1:3;
  y = 1:5;
  [xx,yy] = meshgrid(x,y);
  postwriteinterpfile('fun.txt',x,y,(xx').^2+2*yy');
```

See Also

`elinterp`

Purpose Create rectangular pyramid geometry object.

Syntax

```
rp3 = pyramid3
rp2 = pyramid2
rp3 = pyramid3(a,b,h)
rp2 = pyramid2(a,b,h)
rp3 = pyramid3(a,b,h,rat)
rp2 = pyramid2(a,b,h,rat)
rp3 = pyramid3(a,b,h,rat,...)
rp2 = pyramid2(a,b,h,rat,...)
```

Description

`ec3 = pyramid3` creates a solid rectangular pyramid geometry object with height and side lengths of bottom surface equal to one, axis along the coordinate *z*-axis, and the center of the bottom surface at the origin. `pyramid3` is a subclass of `gencyl3`.

`ec3 = pyramid3(a,b,h)` creates a solid rectangular pyramid geometry object with side lengths `a` and `b`, and height `h`.

`ec3 = pyramid3(a,b,h,rat)` creates a pyramid with the non-negative ratio `rat` between the top and bottom surface.

The functions `pyramid3` and `pyramid2` accept the following property/values:

TABLE 1-117: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Axis`	Vector of reals or cell array of strings	`[0 0]`	Local z-axis of the object
`Const`	Cell array of strings	`{}`	Evaluation context for string inputs
`Displ`	2-by-*nd* matrix	`[0;0]`	Displacement of extrusion top
`Pos`	Vector of reals or cell array of strings	`[0 0]`	Position of the bottom surface
`Rot`	real or string	0	Rotational angle about `Axis` (radians)

For more information on input arguments and properties see `gencyl3`.

`ec2 = pyramid2(...)` creates a surface rectangular pyramid geometry object, without bottom and top faces, according to the arguments described for `pyramid3`. `pyramid2` is a subclass of `gencyl2`.

Pyramid objects have the following properties:

TABLE 1-118: PYRAMID OBJECT PROPERTIES

PROPERTY	DESCRIPTION
`a`, `b`	Side lengths
`h`	Height
`rat`	Ratio
`dx`, `dy`	Semi-axes
`x`, `y`, `z`, `xyz`	Position of the object. Components and vector forms
`ax2`	Rotational angle of symmetry axis
`ax3`	Axis of symmetry
`rot`	Rotational angle

In addition, all 3D geometry object properties are available. All properties can be accessed using the syntax `get(object,property)`. See `geom3` for details.

Compatibility

The FEMLAB 2.3 syntax is obsolete but still supported. The numbering of faces, edges and vertices is different from the numbering in objects created in version 2.3.

See Also

`econe2, econe3`, `gencyl2, gencyl3`

Purpose

Create rectangle geometry object.

Syntax

```
obj = rect2
obj = rect1
obj = rect2(lx,ly,...)
obj = rect1(lx,ly,...)
```

Description

`obj = rect2` creates a solid rectangle geometry object with all side lengths equal to `1`, and the lower left corner at the origin. `rect2` is a subclass of `solid2`.

`obj = rect2(lx,ly,...)` creates a solid rectangle object with side lengths equal to `lx` and `ly`, respectively, and the lower left corner at the origin. `lx` and `ly` are positive real scalars, or strings that evaluate to positive real scalars, given the evaluation context provided by the property `Const`.

The function `rect1` similarly creates curve rectangle objects.

The functions `rect2` and `rect1` accept the following property/values:

TABLE 1-119: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`base`	`corner` | `center`	`corner`	Positions the object either centered about `pos` or with the lower left corner at `pos`
`const`	Cell array of strings	`{}`	Evaluation context for string inputs
`pos`	Vector of reals or cell array of strings	`[0 0]`	Position of the object
`rot`	Real or string	0	Rotational angle about `pos` (radians)

`obj = rect1(...)` creates a curve circle geometry object with properties as given for the `rect2` function. `rect1` is a subclass of `curve2`.

Rectangle objects have the following properties:

TABLE 1-120: RECTANGLE OBJECT PROPERTIES

PROPERTY	DESCRIPTION
`lx`, `ly`	Side lengths
`base`	Base point
`x`, `y`	Position of the object
`rot`	Rotational angle

In addition, all 2D geometry object properties are available. All properties can be accessed using the syntax `get(object,property)`. See `geom2` for details.

Example

The commands below create a geometry object corresponding to the L-shaped membrane using the union of three rectangles and plot the result.

```
sq1 = rect2(1,1);
sq2 = move(sq1,0,-1);
sq3 = move(sq1,-1,-1);
lshape = sq1+sq2+sq3
geomplot(lshape);
```

Compatibility

The FEMLAB 2.3 syntax is obsolete but still supported.

See Also

`geomcsg`, `curve2`, `curve3`, `square1`, `square2`

Purpose

Revolve a 2D geometry object into a 3D geometry object.

Syntax

```
g3 = revolve(g2,..)
```

Description

`g3 = revolve(g2,...)` revolves the 2D geometry object `g2` into a 3D geometry object `g3` according to given parameters.

The function `revolve` accepts the following property/values:

TABLE 1-121: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUES	DEFAULT	DESCRIPTION
`angles`	1-by-2 vector	`[0 2*pi]`	Revolution angle
`polres`	scalar	50	Polygon resolution
`revaxis`	2-by-2 matrix	`[0 0; 0 1]`	Revolution axis
`wrkpln`	3-by-3 matrix	`[0 1 0; 0 0 1; 0 0 0]`	Work plane for 2D geometry cross section

The 3D object `g3` is a revolved object, where the 2D geometry object `g2` lying in the plane defined by the property `wrkpln` is revolved about the revolution axis between the angles defined by the property `angles`. `angles` can also be given as a scalar, in which case the first angle is assumed to be 0.

The property `revaxis` is a 2-by-2 matrix where the first column specifies a point on the axis, and the second column specifies the direction of the revolution axis.

`polres` defines the number of parameter value pairs in the polygon representations of the edges.

Examples

Create torus about the *y*-axis:

```
re = revolve(circ2(1,'pos',[2 0]));
```

Create revolved object from *zx*-plane:

```
p_wrkpln = geomgetwrkpln('quick',{'zx',10});
ax = [0 1;0.5 2]';
g3 = revolve(circ1(0.4,'pos',[1 0]),'angles',[-pi/3 pi/3],...
             'revaxis',ax,'wrkpln',p_wrkpln);
geomplot(g3);
```

See Also

`extrude`, `geom0`, `geom1`, `geom2`, `geom3`, `geomcsg`, `geomgetwrkpln`

Purpose Rotate geometry object.

Syntax

```
[g,...] = rotate(g,r,...)
[g3,...] = rotate(g,r,v,c,...)
[g3,...] = rotate(g,r,vx,vy,vz,cx,cy,cz,...)
[g3,Q,c] = rotate(g,...)
[g,...] = rotate(g,r,c,...)
[g,...] = rotate(g,r,cx,cy,...)
```

Description

`[g,...] = rotate(g,r,...)` rotates the 2D or 3D geometry object `g` by `r` radians about the *z*-axis.

`[g3,...] = rotate(g,r,v,c,...)` rotates the 3D geometry object `g` by `r` radians about the axis `v=(vx,vy,vz)`, with center of rotation `c=(cx,cy,cz)`. `v` can also be a vector of spherical coordinates, where `v(1)` is the polar angle, that is, the angle between the axis of rotation and the positive *z*-axis, and `v(2)` is the azimuthal angle of the axis of rotation.

`[g3,...] = rotate(g,r,vx,vy,vz,cx,cy,cz,...)` is the same as above, but the components of the axis and center of rotation are explicitly given.

`[g3,Q,c] = rotate(g,...)` additionally returns a rotation matrix `Q` corresponding to rotation given by `r` and `v` centered at the origin. The translation vector `c` is also returned for convenience. This means that a point set `p`, of size 3-by-*n*, containing 3D point coordinates, that is to be rotated in the same way as `g`, is transformed according to `prot = Q*(p-cp)+cp`, where `cp = repmat(c(:),1,size(p,2))` represents the center of rotation.

`[c,...] = rotate(g,r,c,...)` rotates a 2D geometry object about the point `c=(cx,cy)`.

`[c,...] = rotate(g,r,cx,cy,...)` is the same as above, but the center coordinates are explicitly given.

The function `rotate` accepts the following property/values:

TABLE 1-122: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Out`	`stx` | `ftx` | `ctx` | `ptx`	empty	Output parameters

See `geomcsg` for more information on geometry objects.

Example The command below rotates the ellipse by 1 radian about (2,3) and plots the result.

```
e1 = ellip2(0,0,1,3);
e2 = rotate(e1,1,2,3);
geomplot(e2)
```

See Also `geomcsg`

Purpose Scale geometry object.

Syntax

```
[g,...] = scale(g3,fx,fy,fy,...)
[g,...] = scale(g3,fx,fy,fy,x,y,z,...)
[g,...] = scale(g3,fxyz,xyz,...)
[g,...] = scale(g2,fx,fy,...)
[g,...] = scale(g2,fx,fy,x,y,...)
[g,...] = scale(g1,fx,...)
[g,...] = scale(g1,fx,x,...)
```

Description `[g,...] = scale(g3,xscale,yscale,zscale,...)` scale the 3D geometry object `g3` by (`xscale`, `yscale`, `zscale`) about the origin.

`[g,...] = scale(g,xscale,yscale,zscale,x,y,z,...)` scale the 3D geometry object `g3` by (`xscale`, `yscale`, `zscale`) about (`x`, `y`, `z`).

`[g,...] = scale(g,xyzscale,xyz,...)` scale the 3D geometry object `g3` by

the vector `fxyz` about the vector `xyz`.

`[g,...] = scale(g2,fx,fy,...)` scale the 2D geometry object by (`fx`, `fy`) about the origin.

`[g,...] = scale(g2,fx,fy,x,y,...)` scale the 2D geometry object by (`fx`, `fy`) about `(x,y)`.

`[g,...] = scale(g2,fx,...)` scale the 1D geometry object by `fx` about the origin.

`[g,...] = scale(g2,fx,x,...)` scale the 1D geometry object by `fx` about `x`.

The function `scale` accepts the following property/values:

TABLE 1-123: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Out`	`stx` | `ftx` | `ctx` | `ptx`	empty	Output parameters

See `geomcsg` for more information on geometry objects.

Examples The commands below scale the unit circle by (1,2) about (2,3) and plot the result.

```
c1 = circ2;
c2 = scale(c1,1,2,2,3);
geomplot(c2)
```

See Also `geomcsg`

Purpose Create an Argyris shape function object.

Syntax `obj = sharg_2_5(basename)`

Description The Argyris shape function object is used to implement the Argyris element of order 5 on triangles in 2D.

`obj = sharg_2_5(basename)`, where `basename` is a string.

For more information, see "The Argyris Element" on page 489.

See Also `shbub`, `shcurl`, `shdens`, `shdisc`, `shdiv`, `shgp`, `shherm`, `shlag`, `shuwhelm`

Purpose Create a bubble element shape function object.

Syntax `obj = shbub(mdim,basename)`

Description The bubble element shape function object is used to implement finite elements of bubble type of order `mdim` + 1 on a simplex.

`mdim` is the maximum dimension of the bubble and `basename` is a string.

For more information, see "Bubble Elements" on page 492.

See Also `sharg_2_5`, `shcurl`, `shdens`, `shdisc`, `shdiv`, `shgp`, `shherm`, `shlag`, `shuwhelm`

Purpose

Create a vector shape function object.

Syntax

`obj = shcurl(order,fieldname)`

Description

The vector shape function object is used to implement finite elements of curl (edge) type of order `order` on all types of mesh elements (also called Nédélec elements). `fieldname` is a string with the field name or a cell array containing the names of the field components.

For more information, see "The Curl Element" on page 494.

Compatibility

COMSOL Multiphysics 3.3: Replaces the `shvec` vector shape function object. `shvec` still works for backward compatibility reasons.

See Also

`sharg_2_5`, `shbub`, `shdens`, `shdisc`, `shdiv`, `shgp`, `shherm`, `shlag`, `shuwhelm`

Purpose Create a density element shape function object.

Syntax `obj = shdens(order,basename)`

Description The density element shape function object is used to implement finite elements of density type on any mesh element type. `order` is the element order, and `basename` is a string. For more information, see “Density Elements” on page 498.

See Also `sharg_2_5`, `shbub`, `shcurl`, `shdisc`, `shdiv`, `shgp`, `shherm`, `shlag`, `shuwhelm`

Purpose Create a discontinuous element shape function object.

Syntax

```
obj = shdisc(mdim,order,basename)
```

Description The discontinuous element shape function object is used to implement finite elements of discontinuous type on any mesh element type.

`mdim` is the maximum dimension of the element, `order` is the default element order, and `basename` is a string.

For more information, see “Discontinuous Elements” on page 496.

Compatibility Since COMSOL Multiphysics 3.2 the meaning of the degrees of freedom has changed. This means that you have to re-solve models made in earlier versions of COMSOL Multiphysics that include discontinuous elements.

See Also `sharg_2_5`, `shbub`, `shcurl`, `shdens`, `shdiv`, `shgp`, `shherm`, `shlag`, `shuwhelm`

Purpose

Create a divergence shape function object.

Syntax

```
obj = shdiv(order,fieldname)
```

Description

The divergence shape function object is used to implement finite elements of divergence type of order `order` on any type of mesh element. `fieldname` is a string with the field name or a cell array containing the names of the field components.

For more information, see "Divergence Elements" on page 499.

Compatibility

Since COMSOL Multiphysics 3.2 the meaning of the degrees of freedom has changed. This means that you have to re-solve models made in earlier versions of COMSOL Multiphysics that include divergence elements.

The syntax `obj = shdiv(fieldname)` is obsolete but still works in COMSOL Multiphysics 3.3.

See Also

`sharg_2_5`, `shbub`, `shcurl`, `shdens`, `shdisc`, `shgp`, `shherm`, `shlag`, `shuwhelm`

Purpose Create a Gauss-point shape function object.

Syntax `obj = shgp(mdim, order, basename)`

Description The Gauss-point shape function object is used to implement finite elements of Gauss-point type of any order on any type of mesh element. The shape function have the degrees of freedoms in the points determined by the Gauss-point pattern for the element type. `mdim` is the maximum dimension of the element, `order` is a positive integer and determines the number of points used through the order of the Gauss-point pattern. `basename` is a string. The variable `basename` is evaluated as the degree of freedom value in the nearest Gauss point.

See Also `sharg_2_5`, `shbub`, `shcurl`, `shdens`, `shdisc`, `shdiv`, `shherm`, `shlag`, `shuwhelm`

Purpose Create a Hermite shape function object.

Syntax

```
obj = shherm(order, basename)
```

Description The Hermite shape function object is used to implement finite elements of Hermite type of any order on mesh elements of any type. `order` is a positive integer greater than 2, and `basename` is a string. The variable `basename` is represented as a polynomial of degree (at most) `order` in the local coordinates.

For more information, see "The Hermite Element" on page 490.

See Also `sharg_2_5`, `shbub`, `shcurl`, `shdens`, `shdisc`, `shdiv`, `shgp`, `shlag`, `shuwhelm`

Purpose Create a Lagrange shape function object.

Syntax `obj = shlag(order, basename)`

Description The Lagrange shape function object is used to implement finite elements of Lagrange type of any order on any type of mesh element. `order` is a positive integer and `basename` is a string. The variable `basename` is represented as a polynomial of degree (at most) `order` in the local coordinates.

For more information, see “The Lagrange Element” on page 487.

Examples The following three sequences set up shape functions for the variables *u* and *v* of order 1 and 2, respectively, using the standard syntax:

```
fem.dim = {'u' 'v'};
fem.shape = [1 2];

fem.dim = {'u' 'v'};
fem.shape = {'shlag(1,''v'')' 'shlag(2,''v'')'}};

fem.dim = {'u' 'v'};
fem.shape = {'shlag(''basename'',''u'',''order'',1)' ...
             'shlag(''basename'',''v'',''order'',2)'}
```

See Also `sharg_2_5`, `shbub`, `shcurl`, `shdens`, `shdisc`, `shdiv`, `shgp`, `shherm`, `shuwhelm`

shuwhelm

Purpose

Create a scalar plane-wave basis function object.

Syntax

```
obj = shuwhelm(ndir,basename,'kvar')
obj = shuwhelm(ndir,basename,'kvar',{'xvar','yvar'})
obj = shuwhelm('ndir',ndir,'basename',basename,'kexpr','kvar',...
    'xexpr',{'xvar','yvar'})
```

Description

The scalar plane-wave basis function object, `shuwhelm`, implements scalar plane-wave basis functions for solving scalar wave equations of Helmholtz type using an *ultraweak variational formulation*. The plane-wave basis functions are discontinuous between mesh elements. `ndir` is a positive integer for the number of directions of the plane-wave basis functions and `basename` is a string. `'kvar'` is a variable representing the wave number. You can also add expressions for the transformation of the spatial coordinates. The default values are the global x, y, and z (in 3D) directions, typically `'x'`, `'y'`, and `'z'`. For PML domains (perfectly matched layers), where the spatial coordinates are mapped to a complex domain, the spatial coordinates in the PML domain provide the coordinate transformation, for example, `'PMLx_acpr'` and `'PMLy_acpr'` (in 2D), where `acpr` is the name of the application mode.

See Also

`sharg_2_5`, `shbub`, `shcurl`, `shdens`, `shdisc`, `shdiv`, `shgp`, `shherm`, `shlag`

Purpose

Constructor functions for solid objects.

Syntax

```
p3 = solid3(vtx,vtxpre,edg,edgpre,fac,mfdpre,mfd)
[s3,...] = solid3(g3,...)
s3 = solid3(g2)
p2 = solid2(vtx,edg,mfd)
[s2,...] = solid2(g,...)
s1 = solid1(x)
s1 = solid1(vtx)
[s1,...] = solid1(g,..)
s0 = solid0(full)
[s0,...] = solid0(p,...)
```

Description

`s3 = solid3(vtx,vtxpre,edg,edgpre,fac,mfdpre,mfd)` creates 3D solid geometry object `s3` from the arguments `vtx`, `vtxpre`, `edg`, `edgpre`, `fac`, `mfdpre`, `mfd`. The arguments must define a valid 3D solid object. See `geom3` for a description of these arguments.

`[s3,...] = solid3(g3,...)` coerces the 3D geometry object `g3` to a 3D solid object `s3`.

`s3 = solid3(g2)` coerces the 2D geometry object `g2` to a 3D solid object `s3`, by embedding `g2` in the plane, `z = 0`.

`p2 = solid2(vtx,edg,mfd)` creates a 2D solid geometry object from the properties `vtx`, `edg`, and `mfd`. The arguments must define a valid 2D solid object. See `geom2` for a description of these arguments.

`[s2,...] = solid2(g,...)` coerces the 2D geometry object to a 2D solid object.

`s1 = solid1(x)` creates a 1D solid object that spans all the coordinate values in the vector `x`.

`s1 = solid1(vtx)` creates a 1D solid geometry object from `vtx`. The arguments must define a valid 2D solid object. See `geom1` for a description of this argument.

`[s1,...] = solid1(g,...)` coerces the 1D geometry object to a 1D solid object.

`g = solid0(full)` creates a 0D solid geometry object, where the Boolean `full` determines if the object is empty or not.

`g = solid0(p)` creates a 0D solid geometry object, where `p` is a matrix of size 0-by-1.

`[s0,...] = solid0(g,...)` coerces the 0D geometry object to a 0D solid object.

The coercion functions `[s0,...] = solid0(g1,...)`, `[s1,...] = solid1(g1,...)`, `[s2,...] = solid2(g2,...)`, and `[s3,...] = solid3(g3,...)` accept the following property/values:

TABLE 1-124: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Out`	`stx` | `ftx` | `ctx` | `ptx`	`{}`	Cell array of output names

See `geomcsg` for more information on geometry objects.

The *n*D geometry object properties are available. The properties can be accessed using the syntax `get(object,property)`. See `geom` for details.

Examples

The following commands create a unit curve circle object, coerce it to a curve object, and then back to a solid object.

```
c1 = circ2
c2 = curve2(c1)
c3 = solid2(c2)
```

Compatibility

The FEMLAB 2.3 syntax is obsolete but still supported.

See Also

`curve2`, `curve3`, `face3`, `geomcsg`, `geom0`, `geom1`, `geom2`, `geom3`, `point1`, `point2`, `point3`

solsize

Purpose Get number of solutions in a solution object.

Syntax

```
sz = solsize(fem.sol)
```

Description `sz = solsize(fem.sol)` returns the number of solutions in the `femsol` object `fem.sol`.

Purpose

Create a spherical geometry object.

Syntax

```
obj = sphere3
obj = sphere2
obj = sphere3(r)
obj = sphere2(r)
obj = sphere3(r,...)
obj = sphere2(r,...)
```

Description

`obj = sphere3` creates a solid sphere geometry object with center at the origin and semi-axes equal to 1. `sphere3` is a subclass of `ellipsoid3`.

`obj = sphere3(r,...)` creates a solid sphere object with radius `r`. `r` is a positive real scalar, or a string that evaluates to a positive real scalar, given the evaluation context provided by the property `const`.

The functions `sphere3/sphere2` accept the following property/values:

TABLE 1-125: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`axis`	Vector of reals or cell array of strings	`[0 0]`	Local z-axis of the object
`const`	Cell array of strings	`{}`	Evaluation context for string inputs
`pos`	Vector of reals or cell array of strings	`[0 0]`	Position of the object
`rot`	real or string	0	Rotational angle about `axis` (radians)

`axis` sets the local *z*-axis, stated either as a directional vector of length 3, or as a 1-by-2 vector of spherical coordinates. `axis` is a vector of real scalars, or a cell array of strings that evaluate to real scalars, given the evaluation context provided by the property `const`. See `gencyl3` for more information on `axis`.

`pos` sets the position of the center of the object. `pos` is a vector of real scalars, or a cell array of strings that evaluate to real scalars, given the evaluation context provided by the property `const`.

`rot` is an intrinsic rotational angle for the object, about its local *z*-axis provided by the property `axis`. `pos` is a real scalar, or a string that evaluate to a real scalar, given the evaluation context provided by the property `const`. The angle is assumed to be in radians if it is numeric, and in degrees if it is a string.

`obj = sphere2(...)` creates a surface sphere object with the properties as given for the `sphere3` function. `sphere2` is a subclass of `ellipsoid2`.

Sphere objects have the following properties:

TABLE 1-126: SPHERE OBJECT PROPERTIES

PROPERTY	DESCRIPTION
`r`	Radius
`x`, `y`, `z`, `xyz`	Position of the object. Components and vector forms
`ax2`	Rotational angle of symmetry axis
`ax3`	Axis of symmetry
`rot`	Rotational angle

In addition, all 3D geometry object properties are available. All properties can be accessed using the syntax `get(object,property)`. See `geom3` for details.

Examples

The following commands create a surface and solid sphere object, where the position and radius are defined in the two alternative ways.

```
s2 = sphere2(1,'pos',[0 0 0],'axis',[0 0 1],'rot',0)
s3 = sphere3(4)
```

Compatibility

The representation of the sphere objects has been changed. The FEMLAB 2.3 syntax is obsolete but still supported. If you use the old syntax or open 2.3 models containing spheres they are converted to general face or solid objects.

See Also

`geom0, geom1, geom2, geom3, ellipsoid2, ellipsoid3`

Purpose Split a geometry object.

Syntax `[gg,...] = split(g,...)`

Description `[gg,...] = split(g,...)` returns a cell array where each cell entry contains a geometry object. When g is solid, face, curve, and point objects, the output `gg` contains object of the respective type. When `g` is a geometry object, the output contains a combination of solid, face, curve, and point objects.

The function `scale` accepts the following property/values:

TABLE 1-127: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Out`	`stx` | `ftx` | `ctx` | `ptx`	empty	Output parameters

Examples Split union of a solid circle and a solid rectangle.

```
g = solid2(geomcsg({rect2,circ2}));
gg = split(g);
```

See Also `geom0, geom1, geom2, geom3`

Purpose Create square geometry objects.

Syntax

```
obj = square2
obj = square1
obj = square2(l,...)
obj = square1(l,...)
```

Description `obj = square2` creates a solid square geometry object with all side lengths equal to `l`, and the lower left corner at the origin. `square2` is a subclass of `rect2` and `solid2`.

`obj = square2(l,...)` creates a solid square object with side lengths equal to `l`. `l` is a positive real scalar, or a string that evaluates to a positive real scalar, given the evaluation context provided by the property `const`.

The function `square1` similarly creates curve square objects.

The functions `square2/square1` accept the following property/values:

TABLE 1-128: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`base`	`corner` | `center`	`corner`	Positions the object either centered about `pos` or with the lower left corner in `pos`
`const`	Cell array of strings	`{}`	Evaluation context for string inputs
`pos`	Vector of reals or cell array of strings	`[0 0]`	Position of the object
`rot`	Real or string	0	Rotational angle about `pos` (radians)

`obj = square1(...)` creates a curve circle geometry object with properties as given for the `rect2` function. `square1` is a subclass of `rect1` and `curve2`.

Square objects have the following properties:

TABLE 1-129: SQUARE OBJECT PROPERTIES

PROPERTY	DESCRIPTION
`l`	Side length
`base`	Base point
`x, y`	Position of the object
`rot`	Rotational angle

In addition, all 2D geometry object properties are available. All properties can be accessed using the syntax `get(object,property)`. See `geom2` for details.

Example The commands below create a unit solid square geometry object and plot it.

```
sq1 = square2(1);
geomplot(sq1)
```

Compatibility The FEMLAB 2.3 syntax is obsolete but still supported.

See Also `geomcsg`, `rect1, rect2`

Purpose Create a tangent to a 2D geometry object.

Syntax

```
g = tangent(g1,g2,...)
g = tangent(g1,p1,...)
```

Description `g = tangent(g1,g2,...)` creates a common tangent line between geometry object `g1` and geometry object `g2`.

`g = tangent(g1,p1,...)` creates a tangent line from geometry object `g1` to a point `p1`.

The function `tangent` accepts the following property/values pairs:

TABLE 1-130: VALID PROPERTY/VALUE PAIRS

PROPERTY NAME	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Edim1`	0 or 1	geometry dependent	Starting point element dimension: 0 for vertex, 1 for edge
`Edim2`	0 or 1	geometry dependent	Ending point element dimension: 0 for vertex, 1 for edge
`Dom1`	positive integer	1	Starting point domain number
`Dom2`	positive integer	1	Ending point domain number
`Start1`	number between 0 and 1	`0.5`	Starting point parameter value on specified edge
`Start2`	number between 0 and 1	`0.5`	Ending point parameter value on specified edge
`Out`	cell array of strings	`{}`	Additional output data (see Table 1-131)

The following properties are valid in the `Out` cell array:

TABLE 1-131: OUTPUT DATA TYPES

ENTRY IN OUT CELL ARRAY	DESCRIPTION
Dom1	Domain number of starting point
Dom2	Domain number of ending point
Param1	Parameter value of starting point
Param2	Parameter value of ending point

TABLE 1-131: OUTPUT DATA TYPES

ENTRY IN OUT CELL ARRAY	DESCRIPTION
Coord1	Coordinate of starting point
Coord2	Coordinate of ending point

Examples

The following commands generate a tangent from the unit circle to the point (2, 0) and plot the result:

```
c=circ2;
t=tangent(c,[2 0]);
geomplot(c); hold on; geomplot(t);
```

The following commands generate a common tangent between two circles and plot the result:

```
c1=circ2;
c2=circ2(1,'pos',[2 2]);
t=tangent(c1,c2,'dom1',4,'dom2',4);
geomplot(c1); hold on; geomplot(c2); geomplot(t);
```

Purpose Create a tetrahedron geometry object.

Syntax

```
t2 = tetrahedron2(p)
t3 = tetrahedron3(p)
```

Description `t3 = tetrahedron3` creates a solid tetrahedron object with the corners at the origin and at the distance 1 from the origin along each positive coordinate axis. `tetrahedron3` is a subclass of `solid3`.

`t3 = tetrahedron3(p)` creates a solid tetrahedron object with the corners given by the four columns of `p`.

`t2 = tetrahedron2(...)` creates a surface tetrahedron object. `tetrahedron2` is a subclass of `face3`.

The 3D geometry object properties are available. The properties can be accessed using the syntax `get(object,property)`. See `geom3` for details

Examples The following command generates a solid tetrahedron object.

```
  t3 = tetrahedron3([0 0   1 1;...
                     0 0.8 1 0;...
                     0 0.1 0 0.2]);
```

See Also `face3`, `geom0, geom1, geom2, geom3`

Purpose

Create torus geometry object.

Syntax

```
t3 = torus3
t2 = torus2
t3 = torus3(rmaj,rmin)
t2 = torus2(rmaj,rmin)
t3 = torus3(rmaj,rmin,phi)
t2 = torus2(rmaj,rmin,phi)
t3 = torus3(rmaj,rmin,phi,...)
t2 = torus2(rmaj,rmin,phi,...)
```

Description

`t3 = torus3` creates a solid torus object with directrix radius 1 and generatrix radius 0.5 about the *z*-axis. `torus3` is a subclass of `solid3`.

`t3 = torus3(rmaj,rmin)` creates a solid torus with directrix radius `rmaj` and generatrix radius `rmin`, where `rmaj>rmin`.

`t3 = torus3(rmaj,rmin,phi)` additionally sets the revolution angle `phi` of the torus.

The functions `torus3/torus2` accept the following property/values:

TABLE 1-132: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Axis`	Vector of reals or cell array of strings	`[0 0]`	Local z-axis of the object
`Const`	Cell array of strings	`{}`	Evaluation context for string inputs
`Pos`	Vector of reals or cell array of strings	`[0 0]`	Position of the bottom surface
`Rot`	real or string	0	Rotational angle about `Axis` (radians)

`t3 = torus2(...)` works similarly to above, but creates a surface torus object. `torus2` is a subclass of `face3`.

Torus objects have the following properties:

TABLE 1-133: TORUS OBJECT PROPERTIES

PROPERTY	DESCRIPTION
`rmaj`	Directrix
`rmin`	Generatrix

TABLE 1-133: TORUS OBJECT PROPERTIES

PROPERTY	DESCRIPTION
`revang`	Revolution angle
`x`, `y`, `z`, `xyz`	Position of the object. Components and vector forms
`ax2`	Rotational angle of symmetry axis
`ax3`	Axis of symmetry
`rot`	Rotational angle

In addition, all 3D geometry object properties are available. All properties can be accessed using the syntax `get(object,property)`. See `geom3` for details.

For more information on geometry objects, see `geom` and `geomcsg`.

Compatibility

The FEMLAB 2.3 syntax is obsolete but still supported. The numbering of faces, edges and vertices is different from the numbering in objects created in 2.3.

Examples

The following commands generate a surface and solid torus, respectively.

```
t2 = torus2(2,1,pi,'pos',[0 0 0],'axis',[0 0 1]);
t3 = torus3(10,2,pi/2,'pos',[1 1 1],'axis',[0 0 -100],...
        'rot',pi/3);
```

See Also

`face3`, `geom0`, `geom1`, `geom2`, `geom3`

Purpose Get extended mesh information.

Syntax

```
out = xmeshinfo(fem,...);
out = xmeshinfo(xmesh,...);
```

Description The `xmeshinfo` function provides information about the numbering of elements, nodes, and degrees of freedom (DOFs) in the extended mesh and in the matrices returned by `assemble` and the solvers.

TABLE 1-134: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Dofname`	string | cell array of strings	`all`	DOF names
`Edim`	integer vector	`all`	Element dimensions
`Geomnum`	integer vector	`all`	Geometry numbers
`Ldof`	cell array	`all`	Local DOFs
`Lnode`	real matrix	`all`	Local coordinates of nodes
`Mcase`	integer	mesh case with largest number of DOFs	Mesh case
`Meshtype`	`vtx` | `edg` | `tri` | `quad` | `tet` | `hex` | `prism` | `edg2` | `tri2` | `quad2` | `tet2` | `hex2` | `prism2` | cell array of these strings	`all`	Mesh element types
`Null`	sparse matrix	identity	Null-space matrix for constraint elimination
`Out`	`mcases` | `femindex` | `dofs` | `nodes` | `edims` | `types` | `dofnames` | `ndofs` | `elements` | cell array of these strings	`dofs`	Output names
`Solcomp`	string | cell array of strings	`all`	DOFs solved for

The properties `Mcase`, `Geomnum`, `Edim`, and `Meshtype` determine the part of the extended mesh that information is requested for. The properties `Lnode`, `Dofname`, and `Ldof` determine the local nodes, DOF names, and local DOFs that information is requested for. The property `Ldof` is a cell array where the first row contains DOF names, and the remaining rows contain local coordinates.

NUMBERING CONVENTIONS

The numbering provided by `xmeshinfo` corresponds to the numbering in the mesh data structure (see `femmesh`). The extended mesh uses a different numbering internally. All numberings are 1-based.

- **Elements**. For each mesh element type, the element numbering of `femmesh` is used.
- **Node points**. The node points in `femmesh` have the same numbers in the extended mesh. Additional node points have higher numbers (these are arbitrarily ordered).
- **Local node points**. The numbering of the local node points within a mesh element is different from the numbering in `femmesh`. However, the same definition of the local coordinate system is used. In the extended mesh, the local node points are ordered in lexicographical order of their local coordinates. In `femmesh`, the mesh vertices come first, in lexicographical order, and then come the other node points in lexicographical order (the latter are only present for a second-order mesh).
- **DOFs**. By default, the DOF number is the index in the complete set of degrees of freedom of the model. If the property `Solcomp` is given, the DOF number is the index in the set of DOFs solved for. If the property `Null` is given, it is assumed that the Eliminate constraint handling method is used, and the DOF number is the index in the set of unconstrained DOFs. This assumes a simple form of the constraints, where each constraint only constrains one DOF. In other words, each column of the `Null` matrix has a single nonzero element. If `Null` does not have this form, an error message is given. The `Null` matrix is an output from the solvers (see `femlin`).

OUTPUTS WITH GLOBAL SCOPE

`mcases = xmeshinfo(xmesh,'out','mcases')` returns an integer vector containing all mesh cases in the extended mesh `xmesh`.

`femindex = xmeshinfo(xmesh,'out','femindex')` returns an integer vector containing indices into `xfem.fem` for all geometries in `xmesh`. That is, geometry `geomnum` in `xmesh` is `xfem.fem{femindex(geomnum)}.geom`.

`dofs = xmeshinfo(xmesh,'out','dofs')` returns information about DOFs in `xmesh` for the mesh case given by the property `Mcase`. The return value `dofs` is a structure with the following fields:

TABLE 1-135: DOFS STRUCT

FIELD	CONTENTS
`mcase`	Mesh case number
`geomnums`	Geometry numbers for all DOFs (vector)
`nodes`	Node numbers for all DOFs (vector)
`coords`	Global coordinates for all DOFs. The `k`th column of this matrix contains the coordinates of DOF number `k`
`names`	DOF names. Note that this is a subset of the property `Dofname` (if given)
`nameinds`	Indices into `names` for all DOFs (vector). The value `0` means that the DOF is not present in `names`
`solcompinds`	Indices into set of DOFs solved for (determined by property `Solcomp`) for all DOFs (vector). This field is only present if the `Null` property is given
`alldofinds`	Indices into total set of DOFs in the model for all DOFs (vector). This field is only present if the `Solcomp` property is given

OUTPUTS RELATED TO GEOMETRIES

`nodes = xmeshinfo(xmesh,'out','nodes')` returns information about nodes in the part of `xmesh` determined by the properties `Mcase` and `Geomnum`. The return value `nodes` is a struct or a cell array of structs with the following fields:

TABLE 1-136: NODES STRUCT CORRESPONDING TO A GEOMETRY

FIELD	CONTENTS
`mcase`	Mesh case number
`geomnum`	Geometry number
`names`	DOF names in this geometry. Note that this is a subset of the property `Dofname` (if given)

TABLE 1-136: NODES STRUCT CORRESPONDING TO A GEOMETRY

FIELD	CONTENTS
`dofs`	DOF numbers for all nodes in this geometry. `dofs(k,n)` is the DOF number for DOF name `names{k}` at node point `n`. A value 0 means that there is no DOF with this name at the node
`coords`	Global coordinates for all nodes. The `n`th column of the matrix `coords` contains the coordinates of node point number `n`

OUTPUTS RELATED TO MESH ELEMENT TYPES

`edims = xmeshinfo(xmesh,'out','edims')` returns a vector containing the element dimensions in the part of `xmesh` determined by the properties `Mcase`, `Geomnum`, `Edim`, and `Meshtype`.

`types = xmeshinfo(xmesh,'out','types')` returns a cell array of strings containing the mesh element types in the part of `xmesh` determined by the properties `Mcase`, `Geomnum`, `Edim`, and `Meshtype`.

`dofnames = xmeshinfo(xmesh,'out','dofnames')` returns a cell array of strings containing the DOF names in the part of `xmesh` determined by the properties `Mcase`, `Geomnum`, `Edim`, `Meshtype`, `Lnode`, `Dofname`, and `Ldof`.

`ndofs = xmeshinfo(xmesh,'out','ndofs')` returns the number of DOFs in the part of `xmesh` determined by the properties `Mcase`, `Geomnum`, `Edim`, `Meshtype`, `Lnode`, `Dofname`, and `Ldof`.

`elements = xmeshinfo(xmesh,'out','elements')` returns information about mesh elements in the part of `xmesh` determined by the properties `Mcase`, `Geomnum`, `Edim`, and `Meshtype`. The return value `elements` is a struct or a cell array of structs with the following fields:

TABLE 1-137: ELEMENTS STRUCT CORRESPONDING TO A MESH ELEMENT TYPE

FIELD	CONTENTS
`mcase`	Mesh case number
`geomnum`	Geometry number
`edim`	Element dimension
`type`	Mesh element type
`lnodes`	Local coordinates of nodes. The `k`th column of the matrix `lnodes` contains the coordinates of local node point number `k`. Note that `lnodes` is a subset of the property `Lnode` (if given)

TABLE 1-137: ELEMENTS STRUCT CORRESPONDING TO A MESH ELEMENT TYPE

FIELD	CONTENTS
`nodes`	Node point indices for all mesh elements of type `type`. `nodes(k,el)` is the node point number within geometry `geomnum` (see the output `nodes`) for local node point `k` within mesh element `el`. A value 0 means that there is no node point at this location
`ldofs`	A cell array containing the local DOFs. The first row contains DOF names, and the remaining rows contain local coordinates. If the property `Ldof` is given, `ldofs` is restricted to a subset. Otherwise, `ldofs` is restricted by the properties `Lnode` and `Dofname` (if given)
`dofs`	DOF numbers for all mesh elements of type `type`. `dofs(k,el)` is the DOF number for local DOF `ldofs(:,k)` within mesh element `el`. A value 0 means that there is no DOF at this location

Examples

Assume that `fem.mesh` is an imported NASTRAN mesh with second-order tetrahedral elements, where node point numbering starts at 1. Use second-order Lagrange elements:

```
m= meshimport('nastrandemo1.nas');
fem.mesh = m{1};
fem.dim = 'u';
fem.shape = 2;
fem.equ.c = 1;
fem.bnd.h = 1;
fem.xmesh = meshextend(fem);
```

To get the DOF number corresponding to node point number 22 in the NASTRAN mesh, type

```
nodes = xmeshinfo(fem,'out','nodes');
nodes.dofs(1,22)
```

Compute an eliminated stiffness matrix and a null-space matrix by

```
[Kc,Null]=femstatic(fem,'out',{'Kc' 'Null'});
```

To find the node point number corresponding to column 30 of `Kc`, and its coordinates, type

```
dofs = xmeshinfo(fem,'out','dofs','null',Null);
n = dofs.nodes(30)
```

```
dofs.coords(:,30) % alternatively:
nodes.coords(:,n)
```

To find the six DOF numbers in tetrahedron element 10 of the mesh, type

```
elements = xmeshinfo(fem,'out','elements',...
                     'meshtype','tet2');
elements.dofs(:,10)
```

To find the total number of DOFs on the boundary, type

```
xmeshinfo(fem,'out','ndofs','edim',2)
```

See Also `femmesh`, `meshextend`

2

Diagnostics

This chapter contains lists of the most common error messages that might occur in COMSOL Multiphysics. The lists also include an explanation of the error and possible causes and workarounds.

Error Messages

This section summarizes the most common error messages and solver messages generated by COMSOL Multiphysics. All error messages are numbered and sorted in different categories according to the following table.

TABLE 2-1: ERROR MESSAGE CATEGORIES

NUMBERS	CATEGORY
1000–1999	Importing Models
2000–2999	Geometry Modeling
3000–3999	CAD Import
4000–4999	Mesh Generation
5000–5999	Point, Edge, Boundary, and Subdomain Specification
6000–6999	Assembly and Extended Mesh
7000–7999	Solvers
8000–8999	Postprocessing
9000–9999	General

For error messages that do not appear in the following lists, contact COMSOL's support team for help.

2000–2999 Geometry Modeling

TABLE 2-2: GEOMETRY MODELING ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
2118	Negative output from empty input	Incorrect Geometry M-file.
2119	Non scalar output from empty input	Incorrect Geometry M-file.
2120	Normal directions are inconsistent	Incorrect input data from STL/VRML import.
2138	Self intersections not supported	Curves resulting in self-intersections are not supported.
2140	Singular extrusions not supported	Error in input parameters.

TABLE 2-2: GEOMETRY MODELING ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
2141	Singular revolutions not supported	The revolved mesh has a singularity at the z axis. If possible, create the cylinder using a 3D primitive or by revolving the geometry before meshing.
2146	Subdomain must bounded at least four boundary segments	Incorrect geometry for mapped mesh.
2147	Subdomain must bound one connected edge component only	Incorrect geometry for mapped mesh.
2190	Invalid radius or distance	Incorrect input parameters to fillet/chamfer.
2197	Operation resulted in empty geometry object	Geometry operation resulted in an empty geometry object which is not allowed. Make sure an empty geometry object is not created.
2209	Geometry to revolve may not cross axis of revolution	The axis of revolution and the geometry intersect. Check the dimension of the geometry and the definition of the axis for the revolution.

4000–4999 Mesh Generation

TABLE 2-3: MESH GENERATION ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
4002	A degenerated tetrahedron was created	The mesh generator ran into numerical difficulties while creating tetrahedrons with a size based on user-controlled parameters. Causes could be too small and narrow subdomains relative to the rest of the geometry or exceedingly short boundary segments. Try to avoid creating small and narrow subdomains and very short boundary segments that are adjacent to longer boundary segments.
4003	A degenerated triangle was created	The mesh generator ran into numerical difficulties while creating triangles with a size based on user-controlled parameters. Causes could be too small and narrow subdomains relative to the rest of the geometry or exceedingly short boundary segments. Try to avoid creating small and narrow subdomains and very short boundary segments that are adjacent to longer boundary segments.
4012	Cannot create mapped mesh for this geometry	The geometry does not fulfill the topological requirements for a mapped mesh. Changes in input parameters or further subdomain division can possibly help this.
4026	Failed create matching edge discretizations	Cannot make mapped mesh with the given input parameters.
4029	Failed to insert point	Problems inserting point at given coordinate. Manually inserting a point there may help.

TABLE 2-3: MESH GENERATION ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
4031	Failed to respect boundary element on geometry edge	The mesh generator failed in making the elements compatible with the geometry object's edges. The reason for this could be that the face mesh is too coarse or contains adjacent elements with large differences in scale. Another reason can be that some subdomains in the geometry are too narrow with respect to the rest of the geometry.
4032	Failed to respect boundary element on geometry face	See Error message 4031.
4044	Internal error boundary respecting	See Error message 4031.
4054	Invalid topology of geometry	The geometry object cannot be used for creating a mapped mesh. It must be subdivided.
4055	Isolated entities found	Entities that are not connected to the boundaries of a geometry objects is found. The mapped mesh generator does not support such isolated entities.
4119	Singular edge detected	The geometry object contains an edge of zero length.

6000–6999 Assembly and Extended Mesh

TABLE 2-4: ASSEMBLY AND EXTENDED MESH ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
6008	Circular variable dependency detected	A variable has been defined in terms of itself, possibly in a circular chain of expression variables. Make sure that variable definitions are sound. Be cautious with equation variables in equations.
6063	Invalid degree of freedom name	The software does not recognize the name of a degree of freedom. Check the names of dependent variables that you have entered for the model. See also Error 7192.

TABLE 2-4: ASSEMBLY AND EXTENDED MESH ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
6139	Wrong number of DOFs in initial value	The current solution or the stored solution has for some reason the wrong number of degrees of freedom, sometimes due to a change of the implementation of elements between two versions of the software. To overcome the problem, go to the **Initial value** area in the **Solver Manager**, and select **Initial value expression**. Then the initial value expressions is evaluated without using the current or stored solution.
6140	Wrong number of dofs in linearization point	The current solution or the stored solution has for some reason the wrong number of degrees of freedom, sometimes due to a change of the implementation of elements between two versions of the software. To overcome the problem, go to the **Value of variables not solved for and linearization point** area in the **Solver Manager**, and click the **Use setting from Initial value frame** button or the **Zero** button.
6163	Divide by zero	A property in the model contains a divisor that becomes zero. Check to make sure that division by zero does not occur in any expression or coefficient.
6164	Duplicate variable name	A variable name has two different definitions. For instance, the same variable name appears two or more times for a dependent variable, a constant, an expression variable, or a coupling variable. Remove or rename one of the variables.
6170	Failed to evaluate variable	An error occurred when evaluating the variable. The domains in which COMSOL Multiphysics tried to evaluate the variable are indicated. Also, the error message shows the expression that COMSOL Multiphysics was unable to evaluate. Make sure that you have defined the variables correctly in the indicated domains.

TABLE 2-4: ASSEMBLY AND EXTENDED MESH ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
6176	Attempt to evaluate real logarithm of negative number	An expression contains `log(a)`, where `a` becomes negative or zero. To make the logarithm well-defined, make sure that `a>0`. Often, `a` becomes only slightly negative (due to approximations in the solution process). Then, a possible solution is to use `log(a+e)`, where `e` is a small constant. Another remedy is to use `log(abs(a))`. If you do want to have a complex logarithm, go to the **Advanced** tab of **Solver Parameters** and select the **Use complex functions with real input** check box.
6177	Matrix has zero on diagonal	When the equations have a structure such that the stiffness matrix (Jacobian matrix) has zeros on the diagonal, it is not possible to use the following linear system solvers/ preconditioners/smoothers: all versions of SOR and Diagonal scaling (Jacobi). Try the Vanka preconditioner/smoother instead.
6188	Out of memory during assembly	The software ran out of memory during assembly of the finite element model. See error 7144 regarding general memory-saving tips.
6194	Attempt to evaluate non-integral power of negative number	An expression contains `a^b`, where `a` becomes negative and `b` is not an integer. To make the power well-defined, make sure that `a>0`. Often, `a` becomes only slightly negative (due to approximations in the solution process). Then, a possible solution is to use `(a+e)^b`, where `e` is a small constant. Another remedy is to use `abs(a)^b`. If you do want to have a complex number `a^b`, go to the **Advanced** tab of **Solver Parameters** and select **Use complex functions with real input**.

TABLE 2-4: ASSEMBLY AND EXTENDED MESH ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
6199	Attempt to evaluate real square root of negative number	The model contains a `sqrt` (square root) function that takes the square root of a negative number. Either make sure that the square-root argument is nonnegative or select the **Use complex functions with real input** check box on the **Advanced** tab in the **Solver Parameters** dialog box.
6204	Undefined function call	An expression contains an undefined function name. Check that the function name is correct and that the function is in COMSOL Multiphysics' or MATLAB's path.
6206	Internal evaluation error: unexpected NaN encountered	Not-A-Number (NaN) appears unexpectedly. A possible cause is improperly defined coupling variables. As a first step, check that the definitions of the source and destination domains of any coupling variables or periodic boundary conditions are correct.
6245	Unsupported integration order	Integration order is too high. For triangular elements the order can be up to 10, and for tetrahedral elements the order can be up to 8. Find more information in the section "Numerical Quadrature" on page 505.
6259	Failed to evaluate variable Jacobian	An error occurred when evaluating the Jacobian of the indicated variable. The domains in which COMSOL Multiphysics tried to evaluate the variable are indicated. Make sure that you have defined the variable correctly in the indicated domains.

7000—7999 Solvers and Preconditioners

TABLE 2-5: SOLVER ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
7001	Adaption only implemented for tetrahedral meshes	It is only possible to use adaptive mesh refinement in 3D for models using tetrahedral mesh elements. Either turn off adaptive mesh refinement or switch from brick or prism elements to tetrahedral elements.
7002	Adaption only implemented for triangular meshes	It is only possible to use adaptive mesh refinement in 2D for models using triangular mesh elements. Either turn off adaptive mesh refinement or switch from quadrilateral elements to triangular elements.
7022	Segregated solver steps do not involve all of solcomp	The groups for the segregated solver do not include all dependent variables. One reason for this error could be that some boundary conditions (for example, for laminar inflow in fluid-flow models) add dependent variables that are not initially in the model.
7043	Initial guess leads to undefined function value	This error message usually appears when you have set up an expression that returns "not a value," that is, it is undefined, for the initial condition you have set. For instance, this happens if an expression contains a divisor that becomes zero or a logarithm of a negative value. To solve the problem, change the expression or the initial value so that the expression is well-defined when substituting the initial value of the variables. Also, watch out for warnings in the Log window.
7067	System matrix is zero	This error message appears if there are no volume elements in the mesh. In the case that you have a mapped surface mesh, try sweeping or extruding the surface mesh to get a volume mesh.

TABLE 2-5: SOLVER ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
7069	Maximum number of linear iterations reached	The iterative linear system solver did not converge due to a bad initial guess or a bad preconditioner. Increase the limit on the number of linear iterations or use a better preconditioner. If possible, use a direct linear system solver.
7081	No parameter name given	The parametric solver does not find a name for the parameter. Check the **Name of parameter** edit field on the **General** page of the S**olver Manager**.
7092	Out of memory in Algebraic multigrid	The Algebraic multigrid solver/ preconditioner ran out of memory. See error 7144 regarding general memory-saving tips.
7093	Out of memory during back substitution	The solver ran out of memory during back substitution. See error 7144 regarding general memory-saving tips.
7094	Out of memory during LU factorization	The solver ran out of memory during LU factorization. See error 7144 regarding general memory-saving tips.
7111	Singular matrix	The system matrix (Jacobian matrix or stiffness matrix) is singular, so the solver cannot invert it. Usually this means that the system is underdetermined. Check that all equations are fully specified and that the boundary conditions are appropriate. For instance, in a stationary model you usually need to have a Dirichlet condition on some boundary. A singular matrix could also occur if mesh elements are of too low quality. If the minimum element quality is less than 0.005 you might be in trouble. Another reason for this error message is that you have different element orders for two variables that are coupled by, for example, a weak constraint. Use the same element order for all variables that are coupled.

TABLE 2-5: SOLVER ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
7136	Very ill-conditioned preconditioner. The relative residual is more than 1000 times larger than the relative tolerance	You need to improve the quality of the preconditioner to get an accurate solution. For the Incomplete LU preconditioner, lower the drop tolerance.
7144	Out of memory in adaptive solver	The adaptive solver ran out of memory. The adaptive mesh refinement has generated a too fine mesh. In general, when you run out of memory, try to use memory-efficient modeling techniques such as utilizing symmetries, solving models sequentially, and selecting memory-efficient solvers. See the chapter “Solving the Model” on page 377 in the *COMSOL Multiphysics User’s Guide* for more information. See also the *COMSOL Installation and Operations Guide* for information about system memory management.
7145	Out of memory in eigenvalue solver	The eigenvalue solver ran out of memory. See error 7144 regarding general memory-saving tips.
7146	Out of memory in stationary solver	The stationary solver ran out of memory. See error 7144 regarding general memory-saving tips.
7147	Out of memory in time-dependent solver	The time-dependent solver ran out of memory. See error 7144 regarding general memory-saving tips.
7192	Invalid degree of freedom name in manual scaling	The name of a dependent variable in the **Manual scaling** edit field on the **Advanced** page in the **Solver Parameters** dialog box does not match any of the dependent variables in the model.
7199	Reordering failed	One of the PARDISO reordering algorithms failed. Try a different reordering algorithm or try turning off row preordering.

TABLE 2-5: SOLVER ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
7248	Undefined value found	See the explanation of error 7043 for some possible reasons as to why this error number appears. In most situations you get a more detailed description of the error by pressing the **Details** button.
7297	Undefined value found	This error number appears if one of the linear system solvers encounters an undefined value (such values appear, for instance, if a division by zero has been performed or if some arithmetic operation results in a larger number than can be represented by the computer). For direct solvers this error might appear if the stiffness matrix (Jacobian matrix) is singular or almost singular. For iterative solvers this error might appear, for instance, if the iterative process diverges. Press the **Details** button to see which linear solver caused the error.

9000–9999 General Errors

TABLE 2-6: GENERAL ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
9037	Failed to initialize 3D graphics. OpenGL not fully supported	OpenGL is not available on the computer. This can happen if your graphics card does not support OpenGL or if you have a Unix/ Linux computer where OpenGL has not been configured.
9040	Fatal error	If you receive this error, click the **Detail** button. Copy and paste the entire error message and send it to support@comsol.com along with your license file and details of how to reproduce the error.

TABLE 2-6: GENERAL ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE	EXPLANATION
9052	Invalid address/port	You did not enter the correct server name or server port when trying to connect a client to a server.
9084	Server connection error	The client somehow lost the connection to the server. For example, the server crashed unexpectedly, or the power saving mechanism on a laptop shut down the TCP/IP connection.
9143	License error	The most common reasons for this message: The license file license.dat has been removed from the right directory in the COMSOL software installation. The license.dat file must be located in the $COMSOL35a/license directory, where $COMSOL35a is the COMSOL 3.5a installation directory. The license manager has not started properly. Please find the FLEXlm log file (named by the person who started the license manager). Inspect this file to see the server status. Send it to support@comsol.com if you are in doubt about how to interpret this file. It is crucial that you use the correct license.dat file on both the server and the clients
9178	Error in callback	An error occurred when calling a MATLAB function from COMSOL Multiphysics. Make sure that the M-file that defines the function is correct and exists in the current path. Check that the function is written so that all inputs are vectors of the same size and the output is a vector of the same size.

Solver Error Messages

These error messages can appear during solution and appear on the **Log** tab in the Progress window.

TABLE 2-7: SOLVER ERROR MESSAGES IN LOG WINDOW

SOLVER ERROR MESSAGE	EXPLANATION
Cannot meet error tolerances. Increase absolute or relative tolerance.	The time-dependent solver cannot solve the model to the specified accuracy.
Error in residual computation Error in Jacobian computation	The evaluation of the residual or the Jacobian generated an error during a time-dependent solution. An additional message states the direct error. Some possible reasons are division by zero, range and overflow errors in mathematical functions, and interpolation failure in coupling variables with time-dependent mesh transformation.
Failed to find a solution	The nonlinear solver failed to converge. An additional error message gives some more details. See the description for that message.
Failed to find a solution for all parameters, even when using the minimum parameter step	During a parametric solution, the nonlinear iteration did not converge despite reducing the parameter step length to the minimum allowed value. The solution may have reached a turning point or bifurcation point.
Failed to find a solution for initial parameter	The nonlinear solver failed to converge for the initial value of the parameter during a parametric solution. An additional error message gives some more details. See the description for that message.
Failed to find consistent initial values	The time-dependent solver could not modify the initial conditions given to a DAE system to satisfy the stationary equations at the initial time. Make sure the initial values satisfy the equations and boundary conditions. In many cases, this can be achieved by solving for only the algebraic variables using a stationary solver before starting the time-dependent solver.
Ill-conditioned preconditioner. Increase factor in error estimate to X	The preconditioner is ill-conditioned. The error in the solution might not be within tolerances. To be sure to have a correct solution, open the **Linear System Solver Settings** dialog box from the **General** tab of **Solver Parameters**. Select **Linear system solver** in the tree, and increase **Factor in error estimate** to the suggested number X. Alternatively, use a better preconditioner or tune the settings for the preconditioner.

TABLE 2-7: SOLVER ERROR MESSAGES IN LOG WINDOW

SOLVER ERROR MESSAGE	EXPLANATION
Inf or NaN found, even when using the minimum damping factor	Despite reducing the step size to the minimum value allowed, the solver cannot evaluate the residual or modified Newton direction at the new solution iterate. This essentially means that the current approximation to the solution is close to the boundary of the domain where the equations are well-defined. Check the equations for divisions by zero, powers, and other functions that can become undefined for certain inputs.
Inverted mesh element near coordinates (x, y, z)	In some mesh element near the given coordinates, the (curved) mesh element is (partially) warped inside-out. More precisely, the Jacobian matrix for the mapping from local to global coordinates has a negative determinant at some point. A possible reason is that the linear mesh contains a tetrahedron whose vertices all lie on a boundary. When improving the approximation of the boundary using curved mesh elements, the curved mesh element becomes inverted. To see whether this is the case, you can change **Geometry shape order** to 1 in the **Model Settings** dialog box, which means that curved mesh elements will not be used. You can usually avoid such bad tetrahedra by using a finer mesh around the relevant boundary. Another reason for this error message can be that the mesh becomes inverted when using a deformed mesh.
Last time step is not converged.	The last time step returned from the time-dependent solver is not to be trusted. Earlier time steps are within the specified tolerances.
Matrix is singular	When encountered during time-dependent solution: the linear system matrix (which is a linear combination of the mass-, stiffness-, and possibly, damping-matrices) is singular. Usually the problem originates from the algebraic part of a DAE. In particular, the cause can often be found in weak constraints or constraint-like equations like the continuity equation in incompressible flow.
Maximum number of linear iterations reached	The iterative linear system solver failed to compute a Newton direction in the specified maximum number of iterations.

TABLE 2-7: SOLVER ERROR MESSAGES IN LOG WINDOW

SOLVER ERROR MESSAGE	EXPLANATION
Maximum number of Newton iterations reached	The nonlinear solver could not reduce the error below the desired tolerance in the specified maximum number of iterations. This is sometimes a sign that the Jacobian is not complete or badly scaled. It may even be almost singular, if the system is underdetermined. If the returned solution seems reasonable, it might be enough to restart the solver with this solution as the initial guess.
No convergence, even when using the minimum damping factor	The nonlinear solver reduced the damping factor below the minimum value allowed. The solver reduces the damping factor each time a computed step did not lead to a decrease in the error estimate. Make sure the model is well-posed, in particular that there are enough equations and boundary conditions to determine all degrees of freedom. If the model is well-posed, it should have one or more isolated solutions. In that case, the error is probably due to the initial guess being too far from any solution.
Nonlinear solver did not converge	During a time-dependent solution, the nonlinear iteration failed to converge despite reducing the time step to the minimum value allowed. Usually, the error is related to the algebraic part of a DAE. For example, the algebraic equations can have reached a turning point or bifurcation point. The error can also appear when the algebraic equations do not have a unique solution consistent with the given initial conditions. Make sure algebraic equations have consistent initial values and that they have a unique solution for all times and values reached by the other variables.
Not all eigenvalues returned	When the eigenvalue solver terminated (stopped by the user or due to an error), it had not found the requested number of eigenvalues. The eigenvalues returned can be trusted.
Not all parameter steps returned	After premature termination of the parametric solver, only some of the requested solutions have been computed.
Predicted solution guess leads to undefined function value	The solver computes the initial guess for the new parameter value based on the solution for the previous parameter value. This initial guess led to an undefined mathematical operation. Try using another **Predictor** on the **Parametric** tab of **Solver Parameters**.

TABLE 2-7: SOLVER ERROR MESSAGES IN LOG WINDOW

SOLVER ERROR MESSAGE	EXPLANATION
Repeated error test failures. May have reached a singularity.	During a time-dependent solution, the error tolerances could not be met despite reducing the time step to the minimum value allowed.
Returned solution has not converged.	The solution returned by the stationary solver is not to be trusted. It might, however, be useful as initial guess after modifying equations or solver settings.
The elasto-plastic solver failed to find a solution	The Newton iteration loop for the computation of the plastic state at some point in the geometry did not converge.

3

The Finite Element Method

This chapter contains a theoretical background to the finite element method and an overview of the finite element types in COMSOL Multiphysics. Sections in this chapter also explain how COMSOL Multiphysics forms the system of equations and constraints that it solves and the implications of Dirichlet conditions involving several solution components in a multiphysics model.

Understanding the Finite Element Method

This section describes how the Finite Element Method (FEM) approximates a PDE problem with a problem that has a finite number of unknown parameters, that is, a *discretization* of the original problem. This concept introduces *finite elements*, or *shape functions*, that describe the possible forms of the approximate solution.

Mesh

The starting point for the finite element method is a mesh, a partition of the geometry into small units of a simple shape, *mesh elements*. For more information about the types of elements that are available in 1D, 2D, and 3D, see “Mesh Elements” on page 300 in the *COMSOL Multiphysics Users Guide*.

Sometimes the term “mesh element” means any of the mesh elements—mesh faces, mesh edges, or mesh vertices. When considering a particular d-dimensional domain in the geometry (that is, a subdomain, boundary, edge, or vertex), then by its mesh elements you mean the d-dimensional mesh elements contained in the domain.

Finite Elements

Once you have a mesh, you can introduce approximations to the dependent variables. For this discussion, concentrate on the case of a single variable, u. The idea is to approximate u with a function that you can describe with a finite number of parameters, the so-called *degrees of freedom* (DOF). Inserting this approximation into the weak form of the equation generates a system of equations for the degrees of freedom.

Start with a simple example: linear elements in 1D. Assume that a mesh consists of just two mesh intervals: $0 < x < 1$ and $1 < x < 2$. Linear elements means that on each mesh interval the continuous function u is linear (affine). Thus, the only thing you need to know in order to characterize u uniquely is its values at the *node points* $x_1 = 0$, $x_2 = 1$, and $x_3 = 2$. Denote these as $U_1 = u(0)$, $U_2 = u(1)$, $U_3 = u(2)$. These are the *degrees of freedom.*

Now you can write

$$u(x) = U_1\varphi_1(x) + U_2\varphi_2(x) + U_3\varphi_3(x)$$

where $\varphi_i(x)$ are certain piecewise linear functions. Namely, $\varphi_i(x)$ is the function that is linear on each mesh interval, equals 1 at the i^{th} node point, and equals 0 at the other node points. For example,

$$\varphi_1(x) = \begin{cases} 1 - x & \text{if } 0 \le x \le 1 \\ 0 & \text{if } 1 \le x \le 2 \end{cases}$$

The $\varphi_i(x)$ are called the *basis functions.* The set of functions $u(x)$ is a linear function space called the *finite element space.*

For better accuracy, consider another finite element space corresponding to quadratic elements. Functions u in this space are second-order polynomials on each mesh interval. To characterize such a function, introduce new node points at the midpoint of each mesh interval: $x_4 = 0.5$ and $x_5 = 1.5$. You must also introduce the corresponding degrees of freedom $U_i = u(x_i)$. Then, on each mesh interval, the second-degree polynomial $u(x)$ is determined by the degrees of freedom at the endpoints and the midpoint. In fact, you get

$$u(x) = U_1\varphi_1(x) + U_2\varphi_2(x) + U_3\varphi_3(x) + U_4\varphi_4(x) + U_5\varphi_5(x)$$

where the basis functions $\varphi_i(x)$ now have a different meaning. Specifically, $\varphi_i(x)$ is the function that is quadratic on each mesh interval, equals 1 at the ith node point, and equals 0 at the other node points. For example,

$$\varphi_1(x) = \begin{cases} (1 - x)(1 - 2x) & \text{if } 0 \le x \le 1 \\ 0 & \text{if } 1 \le x \le 2 \end{cases}$$

In general, you specify a finite element space by giving a set of basis functions. The description of the basis functions is simplified by the introduction of *local coordinates* (or *element coordinates*). Consider a mesh element of dimension d in an n-dimensional geometry (whose space coordinates are denoted $x_1, \ldots, x_n$). Consider also the *standard d-dimensional simplex*

$$\xi_1 \ge 0, \xi_2 \ge 0, \ldots, \xi_d \ge 0, \xi_1 + \ldots + \xi_d \le 1$$

which resides in the local coordinate space parameterized by the local coordinates ξ_1, $\ldots$, ξ_d. If $d = 1$, then this simplex is the unit interval. If $d = 2$, it is a triangle with two 45 degree angles, and if $d = 3$ it is a tetrahedron. Now you can consider the mesh element as a linear transformation of the standard simplex. Namely, by letting the global space coordinates x_i be suitable linear (affine) functions of the local coordinates, you get the mesh element as the image of the standard simplex.

When described in terms of local coordinates, the basis functions assume one of a few basic shapes. These are the *shape functions*. In the example with linear elements in 1D, any basis function on any mesh element is one of the following:

$$\phi = \xi_1, \qquad \phi = 1 - \xi_1, \qquad \phi = 0$$

Thus the first two are the shape functions in this example (0 is not counted as a shape function). In the example with quadratic elements in 1D, the shape functions are

$$\phi = (1-\xi_1)(1-2\xi_1), \qquad \phi = 4\xi_1(1-\xi_1), \qquad \phi = \xi_1(2\xi_1 - 1)$$

CURVED MESH ELEMENTS

When using higher-order elements (that is, elements of an order > 1), the solution has a smaller error. The error also depends on how well the mesh approximates the true boundary. To keep errors in the finite element approximation and the boundary approximation at the same level, it is wise to use *curved mesh elements*. They are distorted mesh elements that can approximate a boundary better than ordinary straight elements (if the problem's boundary is curved). You can get curved mesh elements by writing the global coordinates x_i as polynomials of order k (the *geometry shape order*) in the local coordinates ξ_j. (The earlier example took $k = 1$). Then the mesh element is the image of the standard simplex. For mesh elements that do not touch the boundary, there is no reason to make them curved, so they are straight. It is customary to use the same order k here as for the order of the (Lagrange) element. This is referred to as using *isoparametric elements*.

The order k is determined by the geometry shape order for the frame (coordinate system) associated with the finite element. You can control the geometry shape order using the **Model Settings** dialog box. The frame is determined by the property `frame` to the finite element (the default is the reference frame); see "Shape Function Variables" on page 176. For certain finite elements, the geometry shape order given by the frame can be overridden by the property `sorder`. In the COMSOL Multiphysics user interface, the default setting is to use an automatic geometry shape order, which means that the geometry shape order is equal to the highest order of any shape function used in the model.

If a curved mesh element becomes too distorted, it can become inverted and cause problems in the solution. The software can then reduce the geometry shape order automatically to avoid inverted elements (see “Avoiding Inverted Mesh Elements” on page 374 in the *COMSOL Multiphysics User’s Guide*).

THE LAGRANGE ELEMENT

The preceding examples are special cases of the *Lagrange element*. Consider a positive integer k, the *order* of the Lagrange element. The functions u in this finite element space are piecewise polynomials of degree k, that is, on each mesh element u is a polynomial of degree k. To describe such a function it suffices to give its values in the *Lagrange points* of order k. These are the points whose local (element) coordinates are integer multiples of $1/k$. For example, for a triangular mesh in 2D with $k = 2$, this means that you have node points at the corners and side midpoints of all mesh triangles. For each of these node points p_i, there exists a degree of freedom $U_i = u(p_i)$ and a basis function φ_i . The restriction of the basis function φ_i to a mesh element is a polynomial of degree (at most) k in the local coordinates such that $\varphi_i = 1$ at node i, and $\varphi_i = 0$ at all other nodes. Thus the basis functions are continuous and you have

$$u = \sum_i U_i \varphi_i$$

The Lagrange element of order 1 is called the linear element. The Lagrange element of order 2 is called the quadratic element.

The Lagrange elements are available with all types of mesh elements. The order k can be arbitrary, but the available numerical integration formulas usually limits its usefulness to $k \le 5$ ($k \le 4$ for tetrahedral meshes).

Syntax for the Lagrange Element (shlag)

To specify a Lagrange shape function in the **shape** edit field on the **Element** page of **Subdomain** settings, enter a string of the form `shlag(k,`*`basename`*`)`, where `k` is the order (a positive integer) and *`basename`* is the name of the variable (a string enclosed in single quotes), for example, `shlag(2,'u')`. There is also an alternative syntax `shlag(…)` based on property names and values. The following properties are allowed:

TABLE 3-1: VALID PROPERTY NAME/VALUE PAIRS FOR THE SHLAG SHAPE FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
basename	variable name		Base variable name
order	positive integer		Basis function order
frame	string	reference frame	Frame

TABLE 3-1: VALID PROPERTY NAME/VALUE PAIRS FOR THE SHLAG SHAPE FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
border	positive integer		Alias for order
sorder	positive integer	determined by frame	Geometry shape order

It is not possible to abbreviate the property names, and you must write them in lowercase letters enclosed in single quotation marks. For example:

```
shlag('order',2,'basename','u')
```

Note: When using the property name/value syntax for `shlag` in MATLAB, you must enter the command as a string with each string argument enclosed in two single quotes because they become strings within a string: `'shlag(''order'',2,''basename'',''u'')'`.

The Lagrange element defines the following variables. Denote `basename` with u, and let x and y denote (not necessarily distinct) space coordinates. The variables are (sdim = space dimension and edim = mesh element dimension):

- u
- ux, meaning the derivative of u with respect to x, defined on edim = sdim
- uxy, meaning a second derivative, defined on edim = sdim
- uTx, the tangential derivative variable, meaning the x-component of the tangential projection of the gradient, defined on edim < sdim
- $uTxy$, meaning xy-component of the tangential projection of the second derivative, defined when edim < sdim

When computing the derivatives, the global space coordinates are expressed as polynomials of degree (at most) `sorder` in the local coordinates.

Note: The use of isoparametric elements means that u is not a polynomial in the global coordinates (if $k > 1$), only in the local coordinates.

THE ARGYRIS ELEMENT

For a function represented with Lagrange elements, the first derivatives between mesh elements can be discontinuous. In certain equations (for example, the biharmonic equation) this can be a problem. The *Argyris element* has basis functions with continuous derivatives between mesh triangles (it is defined in 2D). The second order derivative is continuous in the triangle corners. On each triangle, a function u in the Argyris finite element space is a polynomial of degree 5 in the local coordinates.

The Argyris element is available with triangular meshes only.

When setting Dirichlet boundary conditions on a variable that has Argyris shape functions, a locking effect can occur if the boundary is curved and constraint order (`cporder`) `5` is used. Use `cporder=4` if the boundary is curved and `cporder=5` for straight boundaries.

Syntax for the Argyris Element (sharg_2_5)

To specify Argyris shape functions in the **shape** edit field on the **Element** tab in the **Subdomain Settings** dialog box, enter a string of the form `sharg_2_5(`*`basename`*`)`, where *`basename`* is the name of the variable (a string enclosed in single quotes), for example, `sharg_2_5('u')`. There is also an alternative syntax `sharg_2_5(...)` based on property names and values. The following properties are allowed:

TABLE 3-2: VALID PROPERTY NAME/VALUE PAIRS FOR THE SHARG SHAPE FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
basename	variable name		Base variable name
frame	string	reference frame	Frame

The property names cannot be abbreviated and must be written in lowercase letters enclosed in single quotation marks.

Example: `sharg_2_5('basename','u')`.

Note: When using the property name/value syntax for `sharg` in MATLAB, you must enter the command as a string with each string argument enclosed in two single quotes because they become strings within a string:
`'sharg_2_5(''basename'',''u'')'`.

The Argyris element defines the following degrees of freedom (where u is the base name and x and y are the space coordinate names):

- u at corners
- ux and uy at corners, meaning derivatives of u
- uxx, uxy, and uyy at corners, meaning second derivatives
- un at side midpoints, meaning a normal derivative. The direction of the normal is to the right if moving along an edge from a corner with lower mesh vertex number to a corner with higher number

The Argyris element defines the following field variables (where sdim = space dimension = 2 and edim = mesh element dimension):

- u
- ux, meaning the derivative of u with respect to x
- uxy, meaning a second derivative, defined for edim = sdim and edim = 0
- $uxTy$, the tangential derivative variable, meaning the y-component of the tangential projection of the gradient of ux, defined for 0 < edim < sdim

When computing the derivatives, the global space coordinates are always expressed with shape order 1 in the Argyris element.

THE HERMITE ELEMENT

On each mesh element, the functions in the Hermite finite element space are the same as for the Lagrange element, namely, all polynomials of degree (at most) k in the local coordinates. The difference lies in which DOFs are used. For the Hermite element, a DOF u exists at each Lagrange point of order k, except at those points adjacent to a corner of the mesh element. These DOFs are the values of the function. In addition, other DOFs exist for the first derivatives of the function (with respect to the global coordinates) at the corners (ux and uy in 2D). Together, these DOFs determine the polynomials completely. Note that the functions in the Hermite finite element space have continuous derivatives between mesh elements at the mesh vertices. However, at other common points for two mesh elements, these derivatives are not continuous. Thus, you can think of the Hermite element as lying between the Lagrange and Argyris elements.

The Hermite element is available with all types of mesh elements. The order $k \geq 3$ can be arbitrary, but the available numerical integration formulas usually limits its usefulness to $k \leq 5$ ($k \leq 4$ for tetrahedral meshes).

When setting Dirichlet boundary conditions on a variable that has Hermite shape functions, a locking effect can occur if the boundary is curved and the constraint order `cporder` is the same as the order of the Hermite element. This means that the derivative becomes over constrained at mesh vertices at the boundary, due to the implementation of the boundary conditions. To prevent this locking, you can specify `cporder` to be the element order minus 1.

Syntax for the Hermite Element (shherm)

To specify Hermite shape functions in the **shape** edit field on the **Element** tab in the **Subdomain Settings** dialog box, enter a string of the form `shherm(`*`k, basename`*`)`, where *`k`* is the order (an integer > 2), and *`basename`* is the name of the variable (a string enclosed in single quotes), for example `shherm(3,'u')`. There is also an alternative syntax `shherm(...)` based on property names and values. The following properties are allowed:

TABLE 3-3: VALID PROPERTY NAME/VALUE PAIRS FOR THE SHHERM SHAPE FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
basename	variable name		Base variable name
order	integer >= 3		Basis function order
frame	string	reference frame	Frame
border	integer		Alias for `order`
sorder	positive integer	determined by frame	Geometry shape order

The property names cannot be abbreviated and must be written in lowercase letters enclosed in single quotation marks.

Example: `shherm('order',3,'basename','u')`.

Note: When using the property name/value syntax for `shherm` in MATLAB, you must enter the command as a string with each string argument enclosed in two single quotes because they become strings within a string:
`'shherm(''order'',3,''basename'',''u'')'`.

The Hermite element defines the following degrees of freedom:

- The value of the variable `basename` at each Lagrange node point that is not adjacent to a corner of the mesh element.
- The values of the first derivatives of `basename` with respect to the global space coordinates at each corner of the mesh element. The names of these derivatives are formed by appending the space coordinate names to `basename`.

The Hermite element defines the following field variables. Denote `basename` with u, and let x and y denote (not necessarily distinct) space coordinates. The variables are (sdim = space dimension and edim = mesh element dimension):

- u
- ux, meaning the derivative of u with respect to x, defined when edim = sdim or edim=0
- uxy, meaning a second derivative, defined when edim = sdim
- uTx, the tangential derivative variable, meaning the x-component of the tangential projection of the gradient, defined when 0 < edim < sdim
- $uTxy$, meaning xy-component of the tangential projection of the second derivative, defined when edim < sdim

When computing the derivatives, the global space coordinates are expressed as polynomials of degree (at most) `sorder` in the local coordinates.

BUBBLE ELEMENTS

Bubble elements have shape functions that are zero on the boundaries of the mesh element and have a maximum in the middle of the mesh element. The shape function (there is only one for each mesh element) is defined by a lowest-order polynomial that is zero on the boundary of the element.

The bubble element are available with all types of mesh elements.

Syntax for Bubble Elements (shbub)

To specify discontinuous shape functions in the **shape** edit field on the **Element** page in the **Subdomain Settings** dialog box, enter a string of the form `shbub(`*`mdim`*`,`*`basename`*`)`, where *`mdim`* is the dimension of the mesh elements for which the shape functions exist, and *`basename`* is the name of the variable (a string enclosed in single quotes), for example `shbub(3,'u')`. There is also an alternate syntax `shbub(...)` based on property names and values. The following properties are allowed:

TABLE 3-4: VALID PROPERTY NAME/VALUE PAIRS FOR THE SHBUB SHAPE FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
basename	variable name		Base variable name
mdim	nonnegative integer	sdim	Dimension of the mesh elements on which the bubble exist
frame	string	reference frame	Frame
sorder	positive integer	determined by frame	Geometry shape order

The property names cannot be abbreviated and must be written in lowercase letters enclosed in single quotation marks.

Example: `shbub('mdim',3,'basename','u')`.

Note: When using the property name/value syntax for `shbub` in MATLAB, you must enter the command as a string with each string argument enclosed in two single quotes because they become strings within a string:
`'shbub(''mdim'',3,''basename'',''u'')'`.

The bubble element has a single degree of freedom, `basename`, at the midpoint of the mesh element.

The bubble element defines the following field variables. Denote `basename` with u, and let x and y denote (not necessarily distinct) space coordinates. The variables are (sdim = space dimension and edim = mesh element dimension):

- u, defined when $\mathrm{edim} \le \mathrm{mdim}$, $u = 0$ if edim < mdim.
- ux, meaning the derivative of u with respect to x, defined when edim = mdim = sdim.

- uTx, the tangential derivative variable, meaning the x-component of the tangential projection of the gradient, defined when $\text{mdim} < \text{sdim}$ and $\text{edim} \le \text{mdim}$. uTx = 0 if $\text{edim} < \text{mdim}$.
- uTxy, meaning the xy-component of the tangential projection of the second derivative, defined when $\text{mdim} < \text{sdim}$ and $\text{edim} \le \text{mdim}$. uTxy = 0 if $\text{edim} < \text{mdim}$.

THE CURL ELEMENT

In electromagnetics, *curl elements* (also called *vector elements* or *Nédélec's edge elements*) are popular. Each mesh element has DOFs corresponding only to tangential components of the field. For example, in a tetrahedral mesh in 3D each of the three edges in a triangle face element has degrees of freedom that are tangential components of the vector field in the direction of the corresponding edges, and in the interior there are degrees of freedom that correspond to vectors tangential to the triangle itself (if the element order is high enough). Finally, in the interior of the mesh tetrahedron there a degrees of freedom in all coordinate directions (if the element order is high enough). This implies that tangential components of the vector field are continuous across element boundaries, but the normal component is not necessarily continuous. This also implies that the curl of the vector field is an integrable function, so these elements are suitable for equations using the curl of the vector field.

The curl elements are available for all types of mesh elements. The polynomial order of the curl element can be at most 3 in 3D, and at most 4 in 2D and 1D.

Syntax for the Curl Element (shcurl)

To specify curl shape functions in the **shape** edit field on the **Element** page in the **Subdomain Settings** dialog box, enter a string of the form `shcurl(`*`k`*`,`*`fieldname`*`)` where *`fieldname`* is the name of the vector field (a string enclosed in single quotes), and *k* is the order (a positive integer), for example `shcurl(3,'E')`. Alternatively, use the syntax `shcurl(`*`k`*`,`*`compnames`*`)`, where *`compnames`* is a cell array of strings with the vector components, for example `shcurl(3,{'Ex' 'Ey'})`. There is also a syntax `shcurl(...)` based on property names and values. The following properties are allowed:

TABLE 3-5: VALID PROPERTY NAME/VALUE PAIRS FOR THE SHCURL SHAPE FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
fieldname	string		Field name
compnames	cell array of strings	derived from fieldname	Names of vector field components

TABLE 3-5: VALID PROPERTY NAME/VALUE PAIRS FOR THE SHCURL SHAPE FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
dofbasename	string	See below	Base name of degrees of freedom
dcompnames	string	See below	Names of the anti-symmetrized components of the gradient of the vector field
order	integer		Basis function order
frame	string	reference frame	Frame
border	positive integer	`order`	Alias for `order`
sorder	positive integer	given by `frame`	Geometry shape order

The property names cannot be abbreviated and must be written in lowercase letters enclosed in single quotation marks.

Example: `shcurl('compnames’,{'Ex' 'Ey'},'dofbasename','tE')`.

Note: When using the property name/value syntax for `shcurl` in MATLAB, you must enter the command as a string with each string argument (including arguments within arguments) enclosed in two single quotes because they become strings within a string:
`'shcurl(''compnames'',''{''Ex'',''Ey''}'',''dofbasename'',''tE'')'`.

The default for `compnames` is `fieldname` concatenated with the space coordinate names. The default for `dofbasename` is `t`*`allcomponents`*, where *`allcomponents`* is the concatenation of the names in `compnames`.

The property `dcompnames` lists the names of the component of the antisymmetric matrix

$$dA_{ij} = \frac{\partial A_j}{\partial x_i} - \frac{\partial A_i}{\partial x_j},$$

where A_i are the vector field components and x_i are the space coordinates. The components are listed in row order. If a name is the empty string, the field variable corresponding to that component is not defined. If you have provided `compnames`, the default for the entries in `dcompnames` is compnames(j) sdimnames(i) compnames(i)

sdimnames(*j*) for off-diagonal elements. If only `fieldname` has been given, the default for the entries are dfieldname sdimnames(*i*)sdimnames(*j*). Diagonal elements are not defined per defaults. For example,
`shcurl('order',3,'fieldname','A','dcompnames',`
`{'','','curlAy','curlAz','','','','curlAx',''})`.

The curl element defines the following degrees of freedom: `dofbasename` $d\ c$, where $d = 1$ for DOFs in the interior of an edge, $d = 2$ for DOFs in the interior of a surface, etc., and c is a number between 0 and $d - 1$.

The curl element defines the following field variables (where comp is a component name from `compnames`, and dcomp is a component from `dcompnames`, sdim = space dimension and edim = mesh element dimension):

- comp, meaning a component of the vector, defined when edim = sdim.
- tcomp, meaning one component of the tangential projection of the vector onto the mesh element, defined when edim < sdim.
- comp*x*, meaning the derivative of a component of the vector with respect to global space coordinate *x*, defined when edim = sdim.
- tcompT*x*, the tangential derivative variable, meaning the *x* component of the projection of the gradient of tcomp onto the mesh element, defined when edim < sdim. Here, *x* is the name of a space coordinate.
- dcomp, meaning a component of the anti-symmetrized gradient, defined when edim = sdim.
- tdcomp, meaning one component of the tangential projection of the anti-symmetrized gradient onto the mesh element, defined when edim < sdim.

For performance reasons, prefer using dcomp in expressions involving the curl rather than writing it as the difference of two gradient components.

For the computation of components, the global space coordinates are expressed as polynomials of degree (at most) `sorder` in the local coordinates.

DISCONTINUOUS ELEMENTS

The functions in the discontinuous elements space are the same as for the Lagrange element, with the difference that the basis functions are discontinuous between the mesh elements. All degrees of freedom are located in the element interior.

The discontinuous elements are available with all types of mesh elements. The polynomial order k can be arbitrary, but the available numerical integration formulas usually limits its usefulness to $k \le 5$ ($k \le 4$ for tetrahedral meshes).

Syntax for the Discontinuous Element (shdisc)

To specify discontinuous shape functions in the **shape** edit field on the **Element** tab in the **Subdomain Settings** dialog box, enter a string of the form `shdisc(`*`mdim`*`,`*`order`*`,`*`basename`*`)`, where *`mdim`* is the dimension of the mesh elements for which the shape functions exist; *`order`* is the order (a positive integer); and *`basename`* is the name of the variable (a string enclosed in single quotes), for example `shdisc(3,2,'u')`. There is also an alternative syntax `shdisc(…)` based on property names and values. The following properties are allowed:

TABLE 3-6: VALID PROPERTY NAME/VALUE PAIRS FOR THE SHDISC SHAPE FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
basename	variable name		Base variable name
order	integer		Basis function order
mdim	nonnegative integer	`sdim`	Dimension of the mesh elements where the discontinuous element exists
frame	string	reference frame	Frame
border	nonnegative integer	`order`	Alias for `order`
sorder	positive integer	given by `frame`	Geometry shape order

The property names cannot be abbreviated and must be written in lowercase letters enclosed in single quotation marks.

Example: `shdisc('mdim',3,'order',2,'basename','u')`.

Note: When using the property name/value syntax for `shdisc` in MATLAB, you must enter the command as a string with each string argument enclosed in two single quotes because they become strings within a string: `'shdisc(''mdim'',3,''order'',2,''basename'',''u'')'`.

The discontinuous element defines the following field variables. Denote `basename` with u, and let x denote the space coordinates. The variables are (edim is the mesh element dimension):

- u, defined when edim = mdim.
- ux, meaning the derivative of u with respect to x, defined when edim = mdim = sdim.
- uTx, the tangential derivative variable, meaning the derivative of u with respect to x, defined when edim = mdim < sdim.

DENSITY ELEMENTS

The functions in the density elements space are the same as for the discontinuous element if the mesh element is not curved. If the element is curved, the functions define a density of the given order in *local* coordinates, and the value in global coordinates depends on the transformation between local and global coordinates.

The discontinuous elements are available with all types of mesh elements. The order k can be arbitrary, but the available numerical integration formulas usually limits its usefulness to $k \le 5$ ($k \le 4$ for tetrahedral meshes).

Syntax for the Density Element (shdens)

To specify discontinuous shape functions in the **shape** edit field on the **Element** tab in the **Subdomain Settings** dialog box, enter a string of the form `shdens(`*`order`*`,`*`basename`*`)`, where *`order`* is the order (a positive integer) and *`basename`* is the name of the variable (a string enclosed in single quotes), for example `shdens(2,'u')`. There is also an alternative syntax `shdens(…)` based on property names and values. The following properties are allowed:

TABLE 3-7: VALID PROPERTY NAME/VALUE PAIRS FOR THE SHDENS SHAPE FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
basename	variable name		Base variable name
order	integer		Basis function order
frame	string	reference frame	Frame
border	nonnegative integer	`order`	Alias for `order`
sorder	positive integer	given by `frame`	Geometry shape order

The property names cannot be abbreviated and must be written in lowercase letters enclosed in single quotation marks.

Example: `shdens('order',2,'basename','u')`.

Note: When using the property name/value syntax for `shdens` in MATLAB, you must enter the command as a string with each string argument enclosed in two single quotes because they become strings within a string:
`'shdens(''order'',2,''basename'',''u'')'`.

The density element defines the following field variables. Denote `basename` with u, and let x denote the space coordinates. The variables are (edim is the mesh element dimension):

- u, defined when edim = sdim.
- ux, meaning the derivative of u with respect to x, defined when edim = sdim.

DIVERGENCE ELEMENTS

For modeling the **B** (magnetic flux density) and **D** (electric displacement) fields in electromagnetics, the divergence elements are useful. The DOFs on the boundary of a mesh element correspond to normal components of the field. In addition, there are DOFs corresponding to all vector field components in the interior of the mesh element of dimension sdim (if the order is high enough). This implies that the normal component of the vector field is continuous across element boundaries, but the tangential components are not necessarily continuous. This also implies that the divergence of the vector field is an integrable function, so these elements are suitable for equations using the divergence of the vector field.

The divergence element are available with all types of mesh elements. The polynomial order of the divergence element can be at most 3 in 3D, and at most 4 in 2D and 1D.

Syntax for Divergence Elements (shdiv)

To specify divergence shape functions in the **shape** edit field on the **Element** tab in the **Subdomain Settings** dialog box, enter a string of the form `shdiv(`*`fieldname`*`)` where *`fieldname`* is the name of the vector field (a string enclosed in single quotes), for example `shdiv('B')`. Alternatively, use the syntax `shdiv(`*`compnames`*`)`, where *`compnames`* is a cell array of strings with the vector components, for example, `shdiv({'Bx' 'By'})`. There is also a syntax `shdiv(...)` based on property names and values. The following properties are allowed:

TABLE 3-8: VALID PROPERTY NAME/VALUE PAIRS FOR THE SHDIV SHAPE FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
fieldname	variable name		Name of vector field
compnames	cell array of strings	derived from fieldname	Names of vector field components
dofbasename	string	see below	Base name of degrees of freedom
divname	string	see below	Name of divergence field
order	integer	1	Basis function order
frame	string	reference frame	Frame
border	positive integer	`order`	Alias for `order`
sorder	positive integer	given by `frame`	Geometry shape order

The property names cannot be abbreviated and must be written in lowercase letters enclosed in single quotation marks.

Example: `shdiv('compnames',{'Bx' 'By'},'dofbasename','nB')`.

Note: When using the property name/value syntax for `shdiv` in MATLAB, you must enter the command as a string with each string argument (including arguments within arguments) enclosed in two single quotes because they become strings within a string:
`'shdiv(''compnames'',''{''Bx'',''By''}'',''dofbasename'',''nB'')'`.

The default for `compnames` is `fieldname` concatenated with the space coordinate names. The default for `dofbasename` is `n`*`allcomponents`*, where *`allcomponents`* is the concatenation of the names in `compnames`.

The vector element defines the following degrees of freedom: `dofbasename` on element boundaries, and `dofbasename` sdim *c*, *c* = 0, …, sdim – 1 for DOFs in the interior.

The divergence element defines the following field variables (where comp is a component name from `compnames`, divname is the `divname`, sdim = space dimension and edim = mesh element dimension):

- comp, meaning a component of the vector, defined when edim = sdim.
- ncomp, meaning one component of the projection of the vector onto the normal of mesh element, defined when edim = sdim–1.
- comp*x*, meaning the derivative of a component of the vector with respect to global space coordinate *x*, defined when edim = sdim.
- ncompT*x*, the tangential derivative variable, meaning the *x* component of the projection of the gradient of ncomp onto the mesh element, defined when edim < sdim. Here, *x* is the name of a space coordinate. ncompT*x* = 0.
- divname, means the divergence of the vector field.

For performance reasons, prefer using divname in expressions involving the divergence rather than writing it as the sum of sdim gradient components.

For the computation of components, the global space coordinates are expressed as polynomials of degree (at most) `sorder` in the local coordinates.

SCALAR PLANE WAVE BASIS FUNCTION

The scalar plane wave basis function, `shuwhelm`, is used to implement scalar plane wave basis functions for solving scalar wave equations of Helmholtz type using an *ultraweak variational formulation* (UWVF). These basis functions are discontinuous in between mesh elements.

Syntax for the Scalar Plane Wave Basis Function (shuwhelm)

To specify scalar plane wave basis functions in the **shape** edit field on the **Element** page in the **Subdomain Settings** dialog box, enter a string of the form `shuwhelm(`*`ndir`*`,`*`basename`*`,`*`kvar`*`)`, where *`ndir`* is the number of directions for the waves (a positive integer); *`basename`* is the name of the variable (a string enclosed in single quotes); and *`kvar`* is the name of a variable for the wave number (a string enclosed in single quotes), for example `shuwhelm(1,'p','k')`. In addition, you can use the syntax `shuwhelm(`*`ndir`*`,`*`basename`*`,`*`kvar`*`,{`*`xvar`*`,`*`yvar`*`})` to specify the expressions for the spatial coordinate transformation as strings in a cell array, such as `{'x','y'}` (in 2D). In domains that represent perfectly matched layers, the spatial coordinates are mapped to a complex domain, and special spatial coordinate variables provide the transformation of the spatial coordinates. There is also an alternative syntax `shuwhelm(...)` based on property names and values. The following properties are allowed:

TABLE 3-9: VALID PROPERTY NAME/VALUE PAIRS FOR THE SHUWHELM WAVE BASIS FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
basename	variable name		Base variable name
ndir	integer		Number of wave directions
kexpr	string		Variable for the wave number
xexpr	cell array of strings	`{'x','y','z'}`	Expressions for the x, y, and z coordinate transformations

The property names cannot be abbreviated and must be written in lowercase letters enclosed in single quotation marks.

Example: `shuwhelm('ndir',2,'basename','u')`.

Note: When using the property name/value syntax for `shuwhelm` in MATLAB, you must enter the command as a string with each string argument enclosed in two single quotes because they become strings within a string: `'shuwhelm(''ndir'',2,''basename'',''u'')'`.

The scalar plane wave basis function defines the following field variables. Denote `basename` with u, and let x denote the space coordinates. The variables are (sdim = space dimension):

- u
- ux, meaning the derivative of u with respect to x, defined on edim = sdim

Discretization of the Equations

This section describes how COMSOL Multiphysics forms the discretization of the PDE. Consider a 2D problem for simplicity. The starting point is the weak formulation of the problem. First comes the discretization of the constraints

$$0 = R^{(2)} \text{on } \Omega$$
$$0 = R^{(1)} \text{on B}$$
$$0 = R^{(0)} \text{on P}$$

starting with the constraints on the boundaries, B. For each mesh element in B (that is, each mesh edge in B), consider the Lagrange points of some order k (see "The Lagrange Element" on page 487). Denote them by $x_{mj}^{(1)}$, where m is the index of the mesh element. Then the discretization of the constraint is

$$0 = R^{(1)}(x_{mj}^{(1)}),$$

that is, the constraints must hold pointwise at the Lagrange points. The Lagrange point order k can be chosen differently for various components of the constraint vector $R^{(1)}$, and it can also vary in space. COMSOL Multiphysics' data structures denote the k as `cporder`. The constraints on subdomains Ω and points P are discretized in the same way. (Nothing needs to be done with the points P.) You can collect all these pointwise constraints in one equation $0 = M$, where M is the vector consisting of all the right-hand sides.

COMSOL Multiphysics approximates the dependent variables with functions in the chosen finite element space(s). This means that the dependent variables are expressed in terms of the degrees of freedom as

$$u_l = \sum_i U_i \varphi_i^{(l)}$$

where $\varphi_i^{(l)}$ are the basis functions for variable u_l. Let U be the vector with the degrees of freedoms U_i as its components. This vector is called the *solution vector* because it is what you want to compute. M depends only on U, so the constraints can be written $0 = M(U)$.

Now consider the weak equation:

$$0 = \int_\Omega W^{(2)} dA + \int_B W^{(1)} ds + \sum_P W^{(0)}$$
$$- \int_\Omega v \cdot h^{(2)^T} \mu^{(2)} dA - \int_B v \cdot h^{(1)^T} \mu^{(1)} ds - \sum_P v \cdot h^{(0)^T} \mu^{(0)}$$

where $\mu^{(i)}$ are the Lagrange multipliers. To discretize it, express the dependent variables in terms of the DOFs as described earlier. Similarly, approximate the test functions with the same finite elements (this is the Galerkin method):

$$v_l = \sum_i V_i \varphi_i^{(l)}$$

Because the test functions occur linearly in the integrands of the weak equation, it is enough to require that the weak equation holds when you choose the test functions as basis functions:

$$v_l = \varphi_i^{(l)}$$

When substituted into the weak equation, this gives one equation for each i. Now the Lagrange multipliers must be discretized. Let

$$\Lambda_{mj}^{(d)} = \mu^{(d)}(x_{mj}^{(d)})w_{mj}^{(d)}$$

where $x_{mj}^{(d)}$ are the Lagrange points defined earlier, and $w_{mj}^{(d)}$ are certain weights (see the following discussion). The term

$$\int_B \varphi_i \cdot h^{(1)^T}\mu^{(1)}ds$$

is approximated as a sum over all mesh elements in B. The contribution from mesh element number m to this sum is approximated with the Riemann sum

$$\sum_j \varphi_i(x_{mj}^{(1)}) \cdot h^{(1)^T}(x_{mj}^{(1)})\mu^{(1)}(x_{mj}^{(1)})w_{mj}^{(1)} = \sum_j \varphi_i(x_{mj}^{(1)}) \cdot h^{(1)^T}(x_{mj}^{(1)})\Lambda_{mj}^{(1)}$$

where $w_{mj}^{(1)}$ is the length (or integral of ds) over the appropriate part of the mesh element. The integral over Ω and the sum over P is approximated similarly.

All this means that you can write the discretization of the weak equation as

$$0 = L - N_F\Lambda$$

where L is a vector whose ith component is

$$\int_\Omega W^{(2)}dA + \int_B W^{(1)}ds + \sum_P W^{(0)}$$

evaluated for $v_l = \varphi_i^{(l)}$. Λ is the vector containing all the discretized Lagrange multipliers $\Lambda_{mj}^{(d)}$. N_F is a matrix whose ith row is a concatenation of the vectors

$$\varphi_i(x_{mj}^{(d)})h^{(d)}(x_{mj}^{(d)})^T$$

For problems using *ideal constraints*, N_F is equal to the *constraint Jacobian matrix* N, which is defined as

$$N = -\frac{\partial M}{\partial U}$$

To sum up, the discretization of the stationary problem is

$$0 = L(U) - N_F(U)\Lambda$$
$$0 = M(U)$$

The objective is to solve this system for the solution vector U and the Lagrange multiplier vector Λ. L is called the *residual vector*, M is the *constraint residual*, and N_F is the *constraint force Jacobian matrix*. Note that M is redundant in the sense that some pointwise constraints occur several times. Similarly, Λ is redundant. Solvers remove this redundancy.

NUMERICAL QUADRATURE

The integrals occurring in the components of the residual vector L (as well as K, as noted later in this discussion) are computed approximately using a *quadrature formula*. Such a formula computes the integral over a mesh element by taking a weighted sum of the integrand evaluated in a finite number of points in the mesh element. The *order* of a quadrature formula on a 1D, triangular, or tetrahedral element is the maximum number k such that it exactly integrates all polynomials of degree k. For a quadrilateral element, a formula of order k integrates exactly all products $p(\xi_1)q(\xi_2)$, where p and q are polynomials of degree k in the first and second local coordinates, respectively. A similar definition holds for hexahedral and prism elements. Thus the accuracy of the quadrature increases with the order. On the other hand, the number of evaluation points also increases with the order. As a rule of thumb, you can take the order to be twice the order of the finite element being used. COMSOL Multiphysics' data structures refer to the order of the quadrature formula as `gporder` (`gp` stands for Gauss points). The maximum available order of the quadrature formula (the `gporder` value) is:

- 41 for 1D, quadrilateral, and hexahedral meshes
- 30 for triangular and prism meshes
- 8 for tetrahedral meshes

TIME-DEPENDENT PROBLEMS

The discretization of a time-dependent problem is similar to the stationary problem

$$0 = L(U, \dot{U}, \ddot{U}, t) - N_F(U, t)\Lambda$$
$$0 = M(U, t)$$

where U and Λ now depend on time t.

LINEARIZED PROBLEMS

Consider a linearized stationary problem (see “The Linear or Linearized Model” on page 386). The linearization “point” u_0 corresponds to a solution vector U_0. The discretization of the linearized problem is

$$K(U_0)(U - U_0) + N_F(U_0)\Lambda = L(U_0)$$
$$N(U_0)(U - U_0) = M(U_0)$$

where K is called the *stiffness matrix*, and $L(U_0)$ is the *load vector*. For problems given in general or weak form, K is the Jacobian of $-L$:

$$K = -\frac{\partial L}{\partial U}$$

The entries in the stiffness matrix are computed in a similar way to the load vector, namely by integrating certain expressions numerically. This computation is called the *assembling* the stiffness matrix.

If the original problem is linear, then its discretization can be written

$$KU + N_F\Lambda = L(0)$$
$$NU = M(0)$$

Similarly, for a time-dependent model the linearization involves the *damping matrix*

$$D = -\frac{\partial L}{\partial \dot{U}}$$

and the *mass matrix*

$$E = -\frac{\partial L}{\partial \ddot{U}}$$

When $E = 0$, the matrix D is often called the mass matrix instead of the damping matrix.

EIGENVALUE PROBLEMS

The discretization of the eigenvalue problem is

$$\lambda^2 E(U_0)U - \lambda D(U_0)U + K(U_0)U + N_F(U_0)\Lambda = 0$$
$$N(U_0)U = 0$$

where U_0 is the solution vector corresponding to the linearization "point." If the underlying problem is linear, then D, K, and N do not depend on U_0, and you can write

$$KU + N_F\Lambda = \lambda DU - \lambda^2 EU$$
$$NU = 0$$

WEAK CONSTRAINTS

Weak constraints present an alternative way to discretize the Dirichlet conditions, as opposed to the pointwise constraints described earlier. The idea is to regard the Lagrange multipliers $\mu^{(d)}$ as field variables and thus approximate them with finite elements. This concept also introduces corresponding test functions $\nu^{(d)}$. Multiply the Dirichlet conditions with these test functions and integrate to end up with the following system in the case of a stationary problem in 2D:

$$0 = \int_\Omega W^{(2)} dA + \int_B W^{(1)} ds + \sum_P W^{(0)}$$
$$-\int_\Omega v \cdot h^{(2)T} \mu^{(2)} dA - \int_B v \cdot h^{(1)T} \mu^{(1)} ds - \sum_P v \cdot h^{(0)T} \mu^{(0)}$$
$$0 = \int_\Omega \nu^{(2)} \cdot R^{(2)} dA$$
$$0 = \int_B \nu^{(1)} \cdot R^{(1)} ds$$
$$0 = \sum_P \nu^{(0)} \cdot R^{(0)}$$

You could add these weak equations to form a single equation. This treatment of the Lagrange multipliers as ordinary variables has thus produced a weak equation without constraints. This can be useful if the Lagrange multipliers are of interest in their own right.

Take care when combining pointwise and weak constraints. For instance, if you have both types of constraints for some variable and the constraints are in adjacent domains, the resulting discretization does not work. Note that you can obtain pointwise constraints from the weak constraints formulation by using the basis functions

$$\delta(x - x_{mj}^{(d)})$$

for the Lagrange multipliers and their test functions, that is, let

$$\mu^{(d)} = \sum_{m,j} \Lambda_{mj}^{(d)} \delta(x - x_{mj}^{(d)})$$

where δ is Dirac's delta function.

What Equations Does COMSOL Multiphysics Solve?

This section explains how COMSOL Multiphysics forms the system of equations and constraints that it solves. It also discusses the implications of Dirichlet conditions involving several solution components in a multiphysics model.

You specify material parameters and boundary conditions in a number of *application modes*. Enter these settings in the **Subdomain Settings**, **Boundary Settings**, **Edge Settings**, and **Point Settings** dialog boxes, which you open from the **Physics** menu. Each application mode forms one or several PDEs and boundary conditions from these settings. If you use one of the PDE modes, you specify the equation coefficients and terms directly.

The software collects all the equations and boundary conditions formulated by the application modes into one large system of PDEs and boundary conditions. This process also includes converting equations and boundary conditions to the selected *solution form*, which can be *coefficient form*, *general form*, or *weak form*. You can select the solution form on the **Advanced** page in the **Solver Parameters** dialog box. This dialog box also provides the option *automatic* (the default setting), which means that you let the software select the solution form. COMSOL Multiphysics uses the weak solution form unless you have chosen to use the adaptive solver. The software then selects the general solution form because the adaptive solver does not work with the weak form. If any of the equation system forms is weak, the solution form is also the weak form even if you use the adaptive solver because it is not possible to convert the equations from weak form to general form.

In the **Model Settings** dialog box you can specify the *equation system form*—the form of the system of equations and boundary conditions that you can see in the **Equation System** dialog boxes. This form can differ from the solution form. If it does, COMSOL Multiphysics first converts the equations to the solution form before solving. If you use a PDE mode, notice the difference between the form of the PDE in the application mode, the equation system form, and the solution form.

Occasionally you might want to change the PDEs generated by the application modes. You can do this by editing the settings in **Equation System>Subdomain Settings** dialog box (see “Viewing and Modifying the Full Equation System” on page 217 in the *COMSOL Multiphysics User’s Guide*). Similarly, you can change the boundary conditions generated by the application modes in the **Equation System>Boundary Settings** dialog box (see “Modifying Boundary Settings for the Equation System” on page 249 in the *COMSOL Multiphysics User’s Guide*). If you have PDEs or constraints on edges or points, you can also modify the equations that the application modes generate in the **Equation System>Edge Settings** and **Equation System>Point Settings** dialog boxes.

The Equation System/Solution Forms

COEFFICIENT FORM

In the coefficient equation system form, the PDEs and boundary conditions are written in following form (for a time-dependent model):

$$\begin{cases} e_a\dfrac{\partial^2 u}{\partial t^2}+d_a\dfrac{\partial u}{\partial t}+\nabla\cdot(-c\nabla u-\alpha u+\gamma)+\beta\cdot\nabla u+au=f & \text{in } \Omega \\ \mathbf{n}\cdot(c\nabla u+\alpha u-\gamma)+qu=g-h^T\mu & \text{on } \partial\Omega \\ hu=r & \text{on } \partial\Omega \end{cases}$$

In addition to these PDEs, there can be weak-form contributions; see the **weak** and **dweak** edit fields on the **Weak** page of the **Equation System>Subdomain Settings** dialog box. If these edit fields are nonzero, COMSOL Multiphysics modifies the above PDE by:

- Converting the PDE to the weak form by multiplying it by a test function, integrating, and integrating the flux term by parts.
- Adding the **dweak** term to the left side of the resulting weak equation and adding the **weak** term to the right side.
- Adding weak-form contributions from the **Equation System>Boundary Settings**, **Equation System>Edge Settings**, and **Equation System>Point Settings** dialog boxes to the resulting weak equation.

In addition to the above Dirichlet boundary condition, $hu = r$, there can be additional constraints in the **constr** edit field on the **Weak** page of the **Equation System>Boundary Settings** dialog box. The expressions in the **constr** edit field are constrained to be equal to zero. Similarly, the constraints in the **constr** edit fields in **Equation System>Subdomain Settings**, **Equation System>Edge Settings**, and **Equation System>Point Settings** dialog boxes are enforced.

GENERAL FORM

In the general equation system form, the PDEs and boundary conditions are written in following form (for a time-dependent model):

$$\begin{cases} e_a\ddot{u} + d_a\dot{u} + \nabla \cdot \Gamma = F & \text{in } \Omega \\ -\mathbf{n} \cdot \Gamma = G - h^T\mu & \text{on } \partial\Omega \\ 0 = R & \text{on } \partial\Omega \end{cases}$$

Just like for the coefficient form, you can modify this PDE by adding weak-form contributions in the **weak** and **dweak** edit field (see earlier discussion). Similarly, there can be additional constraints in the **constr** edit fields.

WEAK FORM

In the weak equation system form, the PDEs are written solely in the weak formulation (see the **weak** and **dweak** edit fields on the **Weak** tab of **Equation System>Subdomain Settings**, **Equation System>Boundary Settings**, **Equation System>Edge Settings**, and **Equation System>Point Settings** dialog boxes). The **dweak** fields contribute to the left side of the equations, and the **weak** fields contribute to the right side of the equations.

You specify constraints in the **constr** edit fields in the **Equation System>Subdomain Settings**, **Equation System>Boundary Settings**, **Equation System>Edge Settings**, and **Equation System>Point Settings** dialog boxes. These expressions are constrained to be equal to zero.

The Full Equation System

In addition to the PDEs and boundary conditions that you can view in the dialog boxes from the **Equation System** submenu on the **Physics** menu, there can sometimes be extra contributions, which are generated by the application modes or by periodic conditions and identity conditions. You cannot view these contributions directly in the user interface but only in the Model M-file or by exporting the FEM structure to the command window. The extra contributions show up in the fields `elemmph` and `elemcpl` of the FEM structure. They occur in the following cases:

- Periodic conditions generate extra constraints.
- Identity conditions generate extra constraints.

- Dirichlet boundary conditions for the tangential component of a vector field discretized using vector elements generate extra constraints.
- The Shell application mode in the Structural Mechanics Module generates extra contributions to the equations.

The full system of equations and constraints is approximated using the finite element method; see "Discretization of the Equations" on page 502.

Notes on Constraints in Multiphysics Models

In a multiphysics model, if a Dirichlet boundary condition involves two different dependent variables and there is also a Neumann boundary condition, that Neumann boundary condition is not the one displayed in the application mode. The displayed Neumann boundary condition is modified by adding an extra Lagrange multiplier term on the right-hand side.

To explain this, assume that you want to solve the system of PDEs

$$\begin{cases} -u_{xx} = 1 \\ -v_{xx} = 1 \end{cases}$$

on the interval $0 < x < 1$ with the Dirichlet boundary conditions $u = 0$ and $v = 0$ at $x = 0$, and $u = v$ at $x = 1$, and the Neumann boundary condition $v_x = 0$ at $x = 1$. Use two PDE, Coefficient Form application modes (one for u and one for v) and the general equation system form.

The general form boundary conditions at $x = 1$ (which you can inspect in the **Equation System>Boundary Settings** dialog box) read

$$\begin{cases} -\mathbf{n} \cdot \Gamma = G - h^T\mu \\ 0 = R \end{cases}$$

where

$$\mathbf{n} = 1, \quad \Gamma = \begin{bmatrix} -u_x \\ -v_x \end{bmatrix}, \quad G = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} v-u \\ 0 \end{bmatrix}$$

The matrix h is

$$h = -\begin{bmatrix} \partial R_1/\partial u & \partial R_1/\partial v \\ \partial R_2/\partial u & \partial R_2/\partial v \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix}$$

Thus, the resulting boundary conditions are

$$\begin{cases} \begin{bmatrix} u_x \\ v_x \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix} \\ \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} v-u \\ 0 \end{bmatrix} \end{cases}$$

The boundary condition $u_x = -\mu_1$ is expected. It just says that $-u_x$ is equal to the Lagrange multiplier μ_1, but because μ_1 is an unknown this condition can be eliminated. However, the condition $v_x = \mu_1$ is not expected and it invalidates the argument. The resulting boundary condition is $v_x = -u_x$. Note that the boundary-condition description in the PDE application mode for v incorrectly states that the Neumann condition is $v_x = 0$.

The reason for this unexpected result is the Lagrange multiplier term μ_1 in the right-hand side of the Neumann boundary condition for v. Such Lagrange multiplier terms are often called *constraint forces* in a structural mechanics model. The moral is that if you have a Dirichlet boundary condition involving both u and v, you get constraint forces in the Neumann boundary conditions for *both* variables u and v. This means that the Neumann boundary-condition description in the application modes must be modified if you use such Dirichlet boundary conditions.

The term $h^T\mu$ in the right-hand side of the Neumann boundary condition is what characterizes *ideal constraints*. In the above example, you would like to have a *non-ideal constraint* where the term $h^T\mu$ is changed to

$$\begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}$$

To accomplish this, you can remove the constraint specification on the **Coefficients** tab of the **Boundary Settings** dialog box. Instead select **User defined** from the **Constraint type** list on the **Weak** page. Then enter the constraint and the constraint force independently in the **constr** and **constrf** edit fields, respectively.

4

Advanced Geometry Topics

This chapter describes some advanced geometry topics that are part of the solid modeling tools in COMSOL Multiphysics.

Advanced Geometry Topics

Rational Bézier Curves

A rational Bézier curve is a parameterized curve of the form

$$\mathbf{b}(t) = \frac{\sum_{i=0}^{p} \mathbf{b}_i w_i B_i^p(t)}{\sum_{i=0}^{p} w_i B_i^p(t)}, 0 \le t \le 1$$

where the functions

$$B_i^p(t) = \binom{p}{i} t^i (1-t)^{p-i}$$

are the *Bernstein basis* functions of *degree* p; $\mathbf{b}_i = (x_1, \ldots, x_n)$ are the control vertices of the n-dimensional space; and w_i are the weights, which should always be positive real numbers. A rational Bézier curve has a direction defined by the parameter t. This direction is used to uniquely determine subdomain numbers to the left and to the right of a curve in 2D.

Note: The parameter t used in this section is named `s` or `s1` when used as a variable. It does not represent the arc length of a curve but is equivalent to the Bézier parameter as described above.

The end-point interpolation property corresponds to $\mathbf{b}(0) = \mathbf{b}_0$ and $\mathbf{b}(1) = \mathbf{b}_p$. Another useful property of the rational Bézier curves is that the direction of the tangent vector at $t = 0$ and $t = 1$ is determined by the vectors $\mathbf{b}_1 - \mathbf{b}_0$ and $\mathbf{b}_p - \mathbf{b}_{p-1}$, respectively. That is, the curve will always be tangent to the line connecting the control vertices $\mathbf{b}_0$ and $\mathbf{b}_1$ and also to the line connecting $\mathbf{b}_{p-1}$ and $\mathbf{b}_p$. When joining curves at end points, aligning the (nonzero) tangent vectors assures tangential continuity. This technique produces visually smooth transitions between adjacent curves.

There is also an interaction between the control polygon and the curve. For instance, the curve is always contained in the convex hull of its control polygon,

$\{\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_p\}$. A useful property is that of invariance under translation, rotation, and scaling. Translating, rotating, or scaling the control polygon by a certain amount, translates, rotates, or scales the curve that the polygon defines by exactly the same amount. In formal terms, this property of rational Bézier curves is called affine invariance.

A rational Bézier curve is equivalent to a polynomial Bézier curve (or simply a Bézier curve) if the control weights $w_0, \ldots, w_p$ are all equal. In this case the denominator equals the binomial expansion of $(t + (1 - t))^p$, in which each term is one of the Bernstein basis functions. This implies that the polynomial Bézier curves are a subset of the rational Bézier curves.

Note that a line could be viewed as a rational Bézier curve of degree 1.

Conic Sections

Rational Bézier curves of degree 2 can represent all conic sections: circles, ellipses, parabolas, and hyperbolas. Elliptical or circular curve segments are often called arcs. The conic sections are also called quadric curves or quadrics. Because the parameter t is constrained to be in the interval $[0, 1]$, only a segment of the conic section is represented. A 2nd degree curve consists of three control vertices and three weights. There is a simple rule for classifying a 2nd degree curve if the end point weights are set to 1, only allowing the central weight w_1 to vary: if $w_0 = w_2 = 1$, then $0 < w_1 < 1$ gives ellipses, $w_1 = 1$ gives parabolas, and $w_1 > 1$ gives hyperbolas. For a fixed control polygon, at most one value of w_1 (among the ellipses generated by letting $0 < w_1 < 1$) gives a circle segment. For example, a quarter of a full circle is generated by a control polygon with a right angle and with a central weight of $1/\sqrt{2}$.

RELATION TO CURVES ON IMPLICIT FORM.

A rational Bézier curve of degree 2 is a *rational parameterization* of an algebraic curve of degree 2, that is, a curve on the familiar implicit form for quadrics

$$ax^2 + by^2 + cxy + dx + ey + f = 0$$

The unit circle, for example, has $a=b=1, f=-1$ and $c=d=e=0$. The set of rational Bézier curves of degree 2 is essentially equivalent to the set of algebraic curves of degree 2.

Cubic Curves

Rational Bézier curves of degree 3 (cubic curves) have more dynamic properties than conic section curves. A cubic curve has four control vertices and four weights, making it possible, for example, to create a self-intersecting control polygon or a zigzag control polygon. A self-intersecting polygon may give rise to a self-intersecting curve, a loop.

A zigzag control polygon generates an S-shaped curve containing a point of inflection where the tangent line lies on both sides of the curve.

A curve with a cusp is a limiting case of a curve with a loop. A cusp occurs when a loop shrinks so that the area enclosed in the loop approaches zero. At the cusp the tangent vector of the curve vanishes. That is, the curve has no well-defined tangent line at the cusp.

RELATION TO CURVES ON IMPLICIT FORM

The set of rational Bézier curves of degree 3 is a strict subset of the set of algebraic curves of degree 3, that is, curves that contain terms of the type x^3, x^2y, xy^2, y^3, x^2, and so on in their implicit form. This is because some algebraic curves of degree 3 do not have a rational parameterization.

Rational Bézier Surfaces

When you create a 3D geometry object with a curved boundary, COMSOL Multiphysics represents it by rational Bézier surfaces. The software supports two types of Bézier surfaces: rectangular and triangular. A rectangular Bézier surface has a mixed degree (m, n), which represents the degree of the surface in terms of two parameters, often named s and t. A triangular Bézier surface has a single degree, m, just as a Bézier curve.

A rectangular rational Bézier surface of degree p-by-q is described by

$$\mathbf{S}(s,t) = \frac{\sum_{i=0}^{p}\sum_{j=0}^{q}\mathbf{b}_{i,j}w_{i,j}B_i^p(s)B_j^q(t)}{\sum_{i=0}^{p}\sum_{j=0}^{q}w_{i,j}B_i^p(s)B_j^q(t)}, 0 \le s, t \le 1 ,$$

where B_i^p and B_j^q are the Bernstein basis functions of degree p and q, respectively, as described in the previous section. This surface description is called rectangular because the parameter domain is rectangular, that is, the two parameters s and t can vary freely in given intervals.

Another form of surface description is the triangular surface, also called a *Bézier triangle*. A triangular rational Bézier surface is defined as

$$\mathbf{S}(s,t) = \frac{\sum\limits_{i+j \le p} \mathbf{b}_{i,j} w_{i,j} B^p_{i,j}(s,t)}{\sum\limits_{i+j \le p} w_{i,j} B^p_{i,j}(s,t)} \quad , 0 \le s, t \le 1$$

which differs from the Bézier curve description only by the use of *bivariate* Bernstein polynomials instead of *univariate*, for the curve case. The bivariate Bernstein polynomials of degree p are defined as

$$B^p_{i,j}(s,t) = \frac{p!}{i!j!(p-i-j)!} s^i t^j (1-s-t)^{p-i-j}, \quad i+j \le p$$

where the parameters s and t must fulfill the conditions

$$\begin{cases} 0 \le s, t \\ s + t \le 1 \end{cases}$$

which form a triangular domain in the parameter space, therefore the name of this surface description.

The normal vector, $\mathbf{n}(s, t)$, for a point, $\mathbf{S}(s, t)$, at a surface that is defined as

$$\mathbf{n}(s,t) = \frac{\partial \mathbf{S}}{\partial s}(s,t) \times \frac{\partial \mathbf{S}}{\partial t}(s,t)$$

determines the direction of the surface. This direction is used to define the up- and down subdomains of a surface.

The Bézier surfaces are contained in the convex hull of their control points. Bézier surfaces also have the affine invariance property: invariance of surface under translation, rotation, and scaling. Boundary curves of a Bézier surface are Bézier curves, and the corners in the parameter grid that define the control points all lie on the surface.

The simplest form of surface is a *plane*. A Bézier triangle of degree 1 can define a plane spanned by three distinct control points. A rectangular Bézier surface of degree (1,1), on the other hand, forms a bilinear surface where the boundary curves are lines.

COMSOL Multiphysics supports rectangular surfaces of mixed degree at most (3,3) and triangular surfaces of degree 1 to represent planar surfaces. Rectangular rational Bézier surfaces of mixed degree up to (2,2) can represent all common CAD surfaces, including bilinear surfaces, cylinders, cones, spheres, ellipsoids, and tori. The (3,3)-degree rational Bézier curves assist in the creation of additional free-form surfaces. To model a cone or a cylinder you need a rectangular surface of degree (2,1). Modeling a sphere or a torus requires rectangular surfaces of degree (2,2).

CONTROL VERTICES AND WEIGHTS

A rectangular rational Bézier surface of degree (m,n) is defined by a control net consisting of $(m+1)$-by-$(n+1)$ control vertices assigned a positive weight. The surface always interpolates the four corner points of the control net. A change in the net's shape produces a change in the surface's appearance. Its shape mimics that of the control net. The higher the surface degree the more complicated the shapes you can create. Increasing the weight pulls the curve toward the corresponding control vertex. This interaction between the control net and the surface makes the rational Bézier surface representation useful.

TRIMMED SURFACES

The 3D geometry objects in COMSOL Multiphysics are formed by a set of trimmed rational Bézier surfaces. A cylinder consists of four trimmed rectangular degree (2,1) surfaces and two trimmed triangular planar surfaces. The planar surfaces are trimmed by boundary curves in the parameter space so only a circular portion of each planar surface is used. For the curved surfaces, the boundary curves in the parameter space are lying on the rectangular boundary of the surface.

When using geometry modeling operations, the Bézier surfaces are trimmed by the intersection curves between surfaces. By trimming surfaces, surface boundaries can take virtually any shape. The connected surfaces of a 3D geometry object are called faces. A surface can be divided into any number of faces, which are curved areas bounded by trimming (intersection) curves.

Note: The parameters s and t used in this section are equivalent to the variables `s1` and `s2`.

RATIONAL BÉZIER REPRESENTATIONS AND NURBS

The *NURBS* representation (*nonuniform rational B-spline*) is another popular curve and surface representation scheme. It is usually possible to split a curve having a NURBS representation into a sequence of rational Bézier curves.

Parameterization of Curves and Surfaces

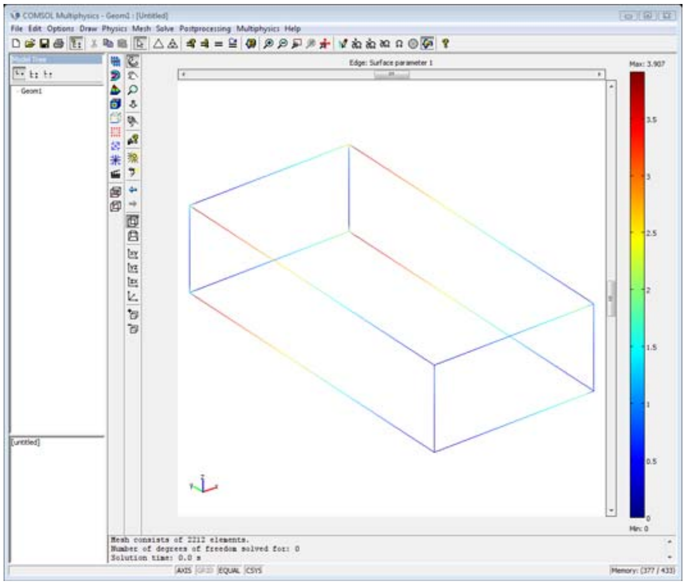
The curves and surfaces of a geometry object can have several mathematical representations. Thus, a local parameter s_1 is defined for curves, and two local parameters s_1 and s_2 are defined for faces. These parameters prove helpful when setting up a model or postprocessing the solution. More precisely, for each value of the curve parameter s_1 within its domain of definition, there is a unique point on the curve, while each pair of values (s_1, s_2) corresponds to a unique point on a face.

The faces and edges in the COMSOL geometry representation consist of trimmed surfaces and curves, respectively. Thus there is a well-defined boundary in the parameter domain that determines the valid values of s_1 and s_2. In 2D, the possible values of the curve parameter s_1 often lie in the interval [0, 1], but in 3D the parameter domain is more complicated for surfaces as well as for curves.

The best way to determine the parameterization is to plot the parameter values. For a block, you can do this in the way described here:

1. Open the **Model Navigator**, select **3D** in the **Space dimension** list and click **OK**.
2. Draw a **Block** object with side lengths 2, 4, and 1.
3. To enter Postprocessing mode, where the software displays the parameterization of the curves and faces, click the **Solve Problem** button on the Main toolbar.
4. Open the **Plot Parameters** dialog box, for example by clicking the **Plot Parameters** button on the Main toolbar. Clear the **Slice** plot type check box and select **Edge** plot instead.
5. Find the **Edge** page in the **Plot Parameters** dialog box and set the **Expression** to `s1`.

6 Click **OK**.

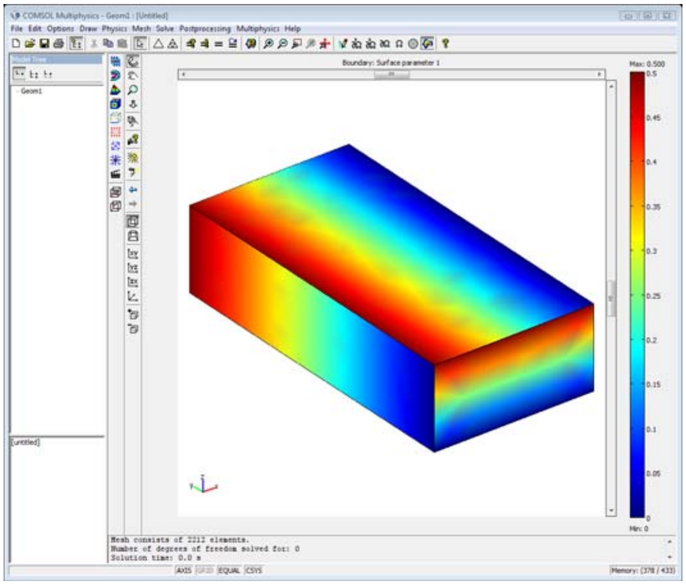

COMSOL Multiphysics - Geom1 : [Untitled]
File Edit Options Draw Physics Mesh Solve Postprocessing Multiphysics Help
Model Tree
Geom1
[untitled]
Edge: Surface parameter 1
Max: 3.907
3.5
3
2.5
2
1.5
1
0.5
0
Min: 0
AXIS
GRID
EQUAL
CSYS
Memory: (377 / 433)

```
Mesh consists of 2212 elements.
Number of degrees of freedom solved for: 0
Solution time: 0.0 s
```

The different geometric variables available for plotting appear in the tables in "Geometric Variables" on page 170.

7 To visualize the surface parameters on the faces, select **Boundary** plot in the **Plot Parameters** dialog box, and enter either `s1` or `s2` as the **Expression** on the **Boundary**

page. Make sure to clear the **Smooth** check box before plotting to avoid incorrect smoothing over the edges.


COMSOL Multiphysics - Geom1 : [Untitled]
File Edit Options Draw Physics Mesh Solve Postprocessing Multiphysics Help
Model Tree
Geom1
[untitled]
Boundary: Surface parameter 1
Max: 0.500
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Min: 0
AXIS
GRID
EQUAL
CSYS
Memory: (378 / 433)

```
Mesh consists of 2212 elements.
Number of degrees of freedom solved for: 0
Solution time: 0.0 s
```

Geometric Variables

Note: In the following table of geometric variables, replace all letters in italic font with the actual names for the independent variables (spatial coordinates) in the model. Replace *x*, *y*, and *z* with the first, second, and third spatial coordinate variable, respectively. If the model contains a deformed mesh, you can replace the symbols *x*, *y*, *z* with either the spatial coordinates (`x`, `y`, `z` by default) or the reference coordinates (`X`, `Y`, `Z` by default).

The geometric variables in the following table characterize geometric properties.

DOMAIN \ SPACE DIM	1D	2D	3D	ND
POINT				
EDGE			`s1`,`t1`*x*,`t1`*y*, `t1`*z*	

DOMAIN \ SPACE DIM	**1D**	**2D**	**3D**	**ND**
BOUNDARY	`dn`*x*`,n`*x*`,un`*x*	`dn`*x*`,dn`*y*`,n`*x*`,n`*y*`,s,t`*x*`,t`*y*`,un`*x*`,un`*y*	`dn`*x*`,dn`*y*`,dn`*z*`,n`*x*`,n`*y*`,n`*z*`,s1,s2,t1`*x*`,t1`*y*`,t1`*z*`,t2`*x*`,t2`*y*`,t2`*z*`,un`*x*`,un`*y*`,un`*z*	
SUBDOMAIN				`h,sd,reldetjac,reldetjacmin`
ALL	*x*`,`*x*`g,dvol`	*x*`,`*y*`,`*x*`g,`*y*`g,dvol`	*x*`,`*y*`,`*z*`,`*x*`g,`*y*`g,`*z*`g,dvol`	`dvol,dom,meshtype,meshelement`

In this table the space dimension refers to the number of independent variables. Most geometric variables of interest are defined on boundaries.

The variables *x*`g`, *y*`g`, and *z*`g` contain the spatial coordinate values of the original geometry as opposed to the standard spatial coordinate variables *x*, *y*, and *z*, which are based on polynomial shape functions. It is the standard spatial coordinate variables' values that COMSOL Multiphysics uses to compute the solution. The difference between these two sets of spatial coordinate variables is normally very small. If a deformed mesh is used, the variables *x*`g`, *y*`g`, *z*`g` are only available when *x*, *y*, *z* are the reference coordinates (`X`, `Y`, `Z` by default).

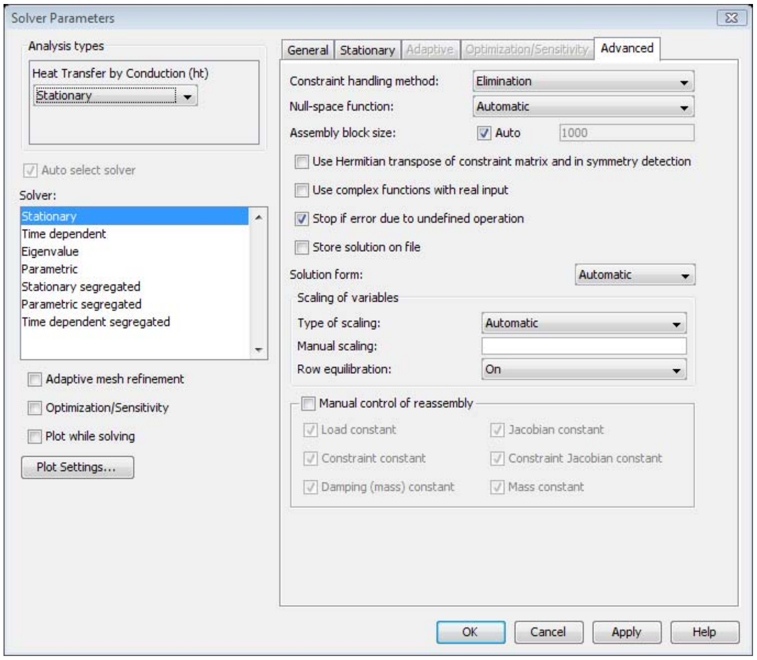
5

Advanced Solver Topics

This chapter describes some advanced solver settings in COMSOL Multiphysics—settings that you for most simulations need not worry about. It also examines the various solvers in COMSOL Multiphysics in some detail.

Advanced Solver Settings

The **Advanced** page in the **Solver Parameters** dialog box controls solver settings you normally do not need to change.



Solver Parameters

Analysis types: Heat Transfer by Conduction (ht) — Stationary

✓ Auto select solver

Solver: Stationary (selected), Time dependent, Eigenvalue, Parametric, Stationary segregated, Parametric segregated, Time dependent segregated

☐ Adaptive mesh refinement
☐ Optimization/Sensitivity
☐ Plot while solving

Constraint handling method: Elimination
Null-space function: Automatic
Assembly block size: ✓ Auto 1000
☐ Use Hermitian transpose of constraint matrix and in symmetry detection
☐ Use complex functions with real input
✓ Stop if error due to undefined operation
☐ Store solution on file
Solution form: Automatic

Scaling of variables
Type of scaling: Automatic
Manual scaling:
Row equilibration: On

☐ Manual control of reassembly
✓ Load constant ✓ Jacobian constant
✓ Constraint constant ✓ Constraint Jacobian constant
✓ Damping (mass) constant ✓ Mass constant

The Advanced page of the Solver Parameters dialog box.

The following sections describe the settings on this page.

Constraint Handling, Null-Space Functions, and Assembly Block Size

The **Constraint handling method** list controls how COMSOL Multiphysics handles constraints. The default elimination method is always preferable, but see “Constraint Handling” on page 533 for details.

The default selection in the **Null-space function** list is **Automatic**, which means that COMSOL Multiphysics chooses the most appropriate of the orthonormal and sparse null-space functions. To override this choice, select **Orthonormal** or **Sparse** from the **Null-space function** list. The orthonormal null-space function computes an orthonormal basis for the null space of the constraint Jacobian N. For models that involve constraints (Dirichlet boundary conditions) that couple values at several nodes, this method typically runs out of memory. For example, this can happen if the model contains:

- Periodic boundary conditions
- Identity conditions
- Constraints involving coupling variables, for instance integral constraints
- Constraints involving derivatives (but it is always good practice to rewrite constraints on normal derivatives as Neumann boundary conditions).

In these cases, the sparse null-space function performs better than the orthonormal null-space function. On the other hand, the sparse null-space function has the following drawbacks:

- It does not always work well together with the Geometric multigrid (GMG) solver/ preconditioner.
- The computation of initial values is less efficient.
- When two boundaries with different Dirichlet boundary conditions meet, the value at the joining node is less predictable. In the case of the orthonormal null-space function, the average value is obtained.

The **Assembly block size** edit field determines the number of mesh elements that the solver processes together during the assembly process. A lower value results in lower memory usage, but it can also make the assembly slower. If the automatically chosen block size gives an unsatisfactory performance for the given problem, then it is recommended to test with block size equal to `1000`.

Settings Related to Complex-Valued Data and Undefined Operations

TAKING THE COMPLEX CONJUGATE FOR COMPLEX-VALUED MODELS

For a complex-valued model, the **Use Hermitian transpose of constraint matrix and in symmetry detection** check box affects the meaning of the Lagrange multiplier term $h^T\mu$ in the Neumann boundary condition of the general or coefficient form, and in general the term $N_F\Lambda$ in the discretized model. If you select this check box, the complex conjugate is taken for the matrix N_F (that is $N_F \to N_F^*$). This check box also affects the automatic symmetry detection. By default, the complex conjugate is not taken. This check box is only active when **Nonsymmetric** or **Automatic** is selected in the **Matrix symmetry** list on the **General** tab. Otherwise, the setting of the check box is determined by the choice **Symmetric/Hermitian**. That is, if **Symmetric** is selected, the conjugate of the constraint force matrix is not taken, and if **Hermitian** is selected, the conjugate -is taken.

USING COMPLEX FUNCTIONS WITH REAL INPUT

If a function takes real inputs, you can assume the output is real by default. For instance, `sqrt(-1)` generates an error message. To change this behavior, select the **Use complex functions with real input** check box.

STOPPING IF ERROR DUE TO UNDEFINED OPERATIONS OCCURS

By default, the solver stops with an error message when it encounters an undefined mathematical operation in an expression that appears in the model settings, for instance, division by zero or square root of a negative number. To change this behavior, clear the **Stop if error due to undefined operation** check box. Then the solver treats the result of the operation as `Inf` (infinity) or `NaN` (not a number). This feature can be useful in a nonlinear problem where the steps in the iterative solution process lead to variable values for which an expression is undefined. The solver then reduces the step size in the Newton iteration when it encounters `Inf` or `NaN` so that it can find a solution.

Storing Solutions on File

By default, COMSOL Multiphysics stores the solution in memory. If you select the **Store solution on file** check box, the solution is primarily stored in a file. This option is useful if you do a time-dependent or parametric simulation with a large number of time steps or parameter steps. The large amount of solution data would otherwise fill up the computer's memory. The software deletes the file that it creates when the solution is no longer needed (for instance, when you exit COMSOL Multiphysics). This file is located in the default directory for temporary files provided by the operating system. You can override this location with an option when starting COMSOL Multiphysics (see the chapter "Running COMSOL" in the *COMSOL Installation and Operations Guide*).

Solution Form

The *solution form* determines the form into which COMSOL Multiphysics converts a PDE and its boundary conditions before solving it. For a description of how the software forms the PDE system, see "What Equations Does COMSOL Multiphysics Solve?" on page 509.

The solution form does not have to be the same as the *equation system form*. The equation system form is the form in which the **Equation System** dialog boxes display the equations. You select the equation system form in the **Model Settings** dialog box. If the solution form is different from the equation system form, COMSOL Multiphysics transforms the equations to the solution form before solving.

When using a PDE mode, be sure not to confuse the PDE form with the solution form; you can, for instance, solve a PDE, Coefficient Form model using the general solution form.

When selecting the solution form, you have (at most) four options: *automatic* (the default), *coefficient*, *general*, and *weak*. With the automatic option, COMSOL Multiphysics selects the solution form according to the following rules:

- If you use the adaptive solver with **Residual method** set to **Coefficient** and the equation system form is *not* the weak form, COMSOL Multiphysics selects the general solution form.
- In all other cases, COMSOL Multiphysics selects the weak solution form

The reason the software selects the general solution form when using the adaptive solver with **Residual method** set to **Coefficient** is that it does not work with the weak solution form (see "The Adaptive Solver Algorithm" on page 540).

The only situation when you need to manually select the solution form is when you want to use *equation variables* (for instance, `cux` and `ncu`), because these are not available for the weak solution form (see "Special Variables" on page 180 of the *COMSOL Multiphysics User's Guide*). Consider the following aspects when selecting the solution form:

- The weak solution form is usually the best choice because you normally get a correct Jacobian (see "The Importance of a Correct Jacobian Matrix" on page 382 of the *COMSOL Multiphysics User's Guide*) and the assembly is somewhat faster than for the coefficient and general forms.
- The general solution form generates an incorrect Jacobian if the model has derivatives in the boundary conditions or some terms in the PDE depend on a second-order spatial derivative. It also generates an incorrect Jacobian if some coefficient or term in the PDE or boundary conditions depends on a coupling variable.

- The general solution form generates an incorrect Jacobian if the model contains time derivatives in other places than multiplying the d_a or e_a coefficients, or in the **weak** or **dweak** edit fields.
- The coefficient solution form is more restricted than the general form. In addition to the disadvantages of the general form, this solution form results in an incorrect Jacobian if some of the coefficients depend on the solution. Therefore use the coefficient form only for linear single-physics or uncoupled models.

Note that not all solution forms might be available, depending on the formulation of the equations in the application modes in use. For instance, it is not possible to solve the nonlinear Navier-Stokes equations in coefficient form.

Manual Control of Reassembly

It is important for the efficiency of the time-stepping algorithm to assemble the time-independent matrices only once. The solver automatically detects the coefficients in your equations that are time dependent and reassembles only those quantities. The nonlinear and parametric solvers also follow this logic (with the parameter playing the role of time).

If the Jacobian is incorrect (see "The Importance of a Correct Jacobian Matrix" on page 382 of the *COMSOL Multiphysics User's Guide*), the automatic detection can fail, which means that you might get an incorrect solution. In this case you must either manually control the reassembly (see below) or, better, use the weak solution form, which you specify on the **General** tab. If you use periodic boundary conditions, identity conditions, or coupling variables, the automatic detection is too sensitive, which means that the solution you get is correct but the reassembly process might take an unnecessarily long time. For such models, you can speed up time-stepping, by manually specifying which matrices are constant. To do so, first select the **Manual control of reassembly** check box and then select the check boxes in this area according to the following guidelines:

- Select the **Load constant** check box if the PDE and Neumann boundary conditions are linear with time-independent coefficients and right-hand sides. For the discretized model, this means that the residual vector L depends linearly on U ($L = L_0 - KU$) and that L_0, K, and the mass matrix D are constant. It is assumed that the Jacobian matrix K is correct.
- Select the **Jacobian constant** check box if the Jacobian matrix K is time-independent. You can also choose this option if you want to use the same Jacobian throughout the time-dependent or nonlinear solver. This choice cuts down linear-system

factorization/preconditioning time but causes more iterations because the Newton iteration is degraded into a fixed-point iteration.

- Select the **Damping (mass) constant** check box if the coefficients of the first-order time-derivative terms are time independent (often the case). In the discretized model, this means that the damping (sometimes called mass) matrix D is constant.
- Select the **Mass constant** check box if the coefficients of the second-order time-derivative terms are time independent (often the case). In the discretized model, this means that the mass matrix E is constant.
- Select the **Constraint constant** check box if the Dirichlet boundary conditions (constraints) are linear and time-independent. For the discretized model, this means that the constraint residual M depends linearly on U ($M = M_0 - NU$) and that M_0 and N are constant. It is also assumed that the constraint Jacobian N is correct.
- Select the **Constraint Jacobian constant** check box if the Dirichlet boundary conditions are linear with time-independent coefficients (not right-hand side). For the discretized model this means that N is constant.

Scaling of Variables and Equations

If the dependent variables in your model have widely different magnitudes, the solver might have problems with the resulting ill-conditioned matrix. For instance, in a structural-mechanics problem the displacements can be of the order of 0.0001 m while the stresses are 1,000,000 Pa (1 MPa). To remedy this situation, COMSOL Multiphysics internally rescales the variables so that a well-scaled system results.

The default is *automatic scaling*, which works well for most models. It is based on the magnitudes of the elements in the Jacobian and mass matrices. If you know the order of magnitudes of the variables in advance, you can opt for *manual scaling*. For instance, suppose that a problem involves two degrees of freedom, `u` and `sigma`, and that the values of `u` are on the order of 10^{-4} and the values of `sigma` are approximately 10^6. To take this knowledge into account, in the **Scaling of variables** area, select **Manual** from the **Type of scaling** list and type `u 1e-4 sigma 1e6` in the **Manual scaling** edit field. When you start the solvers, they internally use the rescaled degrees of freedom `U = u/1e-4` and `Sigma = sigma/1e6`, which both are of the order 1. The units for the values of the degrees of freedom are the units for the corresponding quantities in the model's base unit system. For example, if `u` in the manual scaling above represents a displacement in a model that uses SI units, its value is 10^{-4} m. If you provide an initial value that gives a good estimate of the scales of your variables, another choice is to use *initial-value based scaling* by selecting **Initial value based** in the **Type of scaling** list. To turn off scaling, select **None** in the **Type of scaling** list.

Note: The automatic scaling in COMSOL Multiphysics does not work when using the nonlinear stationary solver and one solution component is identically zero in the solution (the solver does not converge). In this case use **Manual** or **None**.

Even if variables are well scaled, equations can have very different scales. To overcome this problem you can equilibrate the equations by selecting **On** in the **Row equilibration** list (the default). To turn off equation scaling, choose **Off** in the **Row equilibration** list. To preserve the possible symmetry of the matrix, COMSOL Multiphysics does not use row equilibration in the following cases:

- Automatic matrix symmetry detection is used and the system matrices are symmetric
- **Symmetric** or **Hermitian** is selected in the **Matrix symmetry** list
- The Conjugate gradients or Geometric multigrid solver is used
- The eigenvalue solver is used.

THE RESCALED LINEAR SYSTEM

The rescaling of the discretized linear system (see "The Discretized Linearized Model" on page 388 of the *COMSOL Multiphysics User's Guide*) occurs before constraint handling. Assume that the degrees of freedom U_i are expressed terms of rescaled degrees of freedom $\tilde{U}_i$ according to the formula

$$U_i = s_i\tilde{U}_i$$

where s_i are positive scale factors. Using a diagonal matrix S, the relation between U and $\tilde{U}$ is

$$U = S\tilde{U},$$

and you can write the rescaled linear system as

$$\begin{bmatrix} \tilde{K} & \tilde{N}_F \\ \tilde{N} & 0 \end{bmatrix}\begin{bmatrix} \tilde{U} \\ \tilde{\Lambda} \end{bmatrix} = \begin{bmatrix} \tilde{L} \\ \tilde{M} \end{bmatrix}$$

where

$$\Lambda = R\tilde{\Lambda} \qquad \tilde{N}_F = SN_FR \qquad \tilde{K} = SKS \qquad \tilde{N} = RNS$$

and

$$\tilde{L} = SL, \quad \tilde{M} = RM .$$

Here, R is a diagonal matrix of positive scale factors chosen such that the rows in the matrix $\tilde{N}$ are of magnitude 1.

Constraint Handling

Consider the linear (scaled) system

$$\begin{bmatrix} K & N_F \\ N & 0 \end{bmatrix} \begin{bmatrix} U \\ \Lambda \end{bmatrix} = \begin{bmatrix} L \\ M \end{bmatrix} .$$

The Lagrange multiplier vector Λ is typically underdetermined, and COMSOL Multiphysics does not solve for it. Similarly, the constraint $NU = M$ often contains the same equation several times. To handle this problem, COMSOL Multiphysics turns to a *constraint-handling method* using elimination, Lagrange multipliers, or stiff springs. Select the desired constraint-handling method on the **Advanced** page of the **Solver Parameters** dialog box.

ELIMINATION CONSTRAINT HANDLING

This is the default constraint-handling method. The solver computes a solution U_d to the constraint $NU = M$ as well as a matrix Null, whose columns form a basis for the null space of N. For non-ideal constraints ($N_F \neq N^T$) then a matrix Nullf is also computed, whose column forms a basis for the null space of N_F^T. Then it obtains the solution as $U = \mathrm{Null}\, U_n + U_d$, where U_n is the solution of $K_e\, U_n = L_e$, where

$$\begin{cases} K_e = \mathrm{Nullf}^T K\, \mathrm{Null} \\ L_e = \mathrm{Nullf}^T (L - KU_d) \end{cases} .$$

For the ideal constraint case $\mathrm{Nullf} = \mathrm{Null}$, the corresponding eliminated D and E matrices are

$$D_e = \mathrm{Nullf}^T D\, \mathrm{Null}, \qquad E_e = \mathrm{Nullf}^T E\, \mathrm{Null}$$

LAGRANGE MULTIPLIERS CONSTRAINT HANDLING

The linear solver computes a matrix Range, whose columns form a basis for the range of N, and a matrix Rangef, whose columns form a basis for the range of $N_\mathrm{F}{}^T$. Then it constrains Λ to be of the form $\Lambda = s\mathrm{Rangef}\Lambda$ where s is a scaling factor. This transforms the system to

$$K_l \begin{bmatrix} U \\ \hat{\Lambda} \end{bmatrix} = L_l$$

where

$$K_l = \begin{bmatrix} K & sN_F\mathrm{Rangef} \\ s\mathrm{Range}^T N & 0 \end{bmatrix} \qquad L_l = \begin{bmatrix} L \\ s\mathrm{Range}^T M \end{bmatrix}$$

COMSOL Multiphysics solves this system for both U and $\hat{\Lambda}$, but it returns only U. The corresponding D matrix is

$$D_l = \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}$$

For the ideal constraint case $\mathrm{Rangef} = \mathrm{Range}$.

STIFF SPRINGS CONSTRAINT HANDLING

This method approximates the constraint $NU = M$ by

$$NU = M + \frac{1}{\kappa}\Lambda$$

where κ is a suitably large constant. Eliminating Λ gives the system $K_sU = L_s$, where

$$\begin{cases} K_s = K + \kappa N_F N \\ L_s = L + \kappa N_F M \end{cases}$$

The corresponding D matrix is $D_s = D$.

Solver Algorithms

The Nonlinear Solver Algorithm

The nonlinear solver uses an affine invariant form of the damped Newton method as described in Ref. 1. You can write the discrete form of the equations as $f(U) = 0$, where $f(U)$ is the residual vector and U is the solution vector. Starting with the initial guess U_0, the software forms the linearized model using U_0 as the linearization point. It solves the discretized form of the linearized model $f'(U_0)\delta U = -f(U_0)$ for the Newton step δU using the selected linear system solver ($f'(U_0)$ is the Jacobian matrix). It then computes the new iteration $U_1 = U_0 + \lambda\delta U$, where λ ($0 \le \lambda \le 1$) is the damping factor. Next the modified Newton correction estimates the error E for the new iteration U_1 by solving $f'(U_0)E = -f(U_1)$. If the relative error corresponding to E is larger than the relative error in the previous iteration, the code reduces the damping factor λ and recomputes U_1. This algorithm repeats the damping-factor reduction until the relative error is less than in the previous iteration or until the damping factor underflows the minimum damping factor. When it has taken a successful step U_1, the algorithm proceeds with the next Newton iteration.

A value of $\lambda = 1$ results in Newton's method, which converges quadratically if the initial guess U_0 is sufficiently close to a solution. In order to enlarge the domain of attraction, the solver chooses the damping factors judiciously. Nevertheless, the success of a nonlinear solver depends heavily on a carefully selected initial guess. Thus you should spend considerable time providing the best value for U_0, giving at least an order of magnitude guess for different solution components.

CONVERGENCE CRITERION

The nonlinear iterations terminate when the following convergence criterion is satisfied: Let U be the current approximation to the true solution vector, and let E be the estimated error in this vector. The software stops the iterations when the relative tolerance exceeds the relative error computed as the weighted Euclidean norm

$$\mathrm{err} = \left(\frac{1}{N}\sum_{i=1}^{N}(|E_i|/W_i)^2\right)^{1/2}$$

Here N is the number of degrees of freedom and $W_i = \max(|U_i|, S_i)$, where S_i is a scale factor that the solver determines on the basis of the **Type of scaling** option selected in the **Scaling of variables** area on the **Advanced** page according to the following rules:

- For **Automatic**, S_i is the average of $|U_j|$ for all DOFs j having the same name as DOF i times a factor equal to 10^{-5} if the **Highly nonlinear problem** check box is selected or 0.1 otherwise.
- For **Manual**, S_i is the value for DOF i given in the **Manual scaling** edit field.
- For **Initial value based**, S_i is the factor s (see below) times the average of $|U_{0j}|$ for all DOFs j having the same name as DOF i, where U_0 is the solution vector corresponding to the initial value.
- For **None**, $W_i = 1$. In this case, err is an estimate for the absolute error.

Note that nonlinear solver only checks the convergence criterion if the damping factor for the current iteration is equal to 1. Thus, the solver continues as long as the damping factor is not equal to 1 even if the estimated error is smaller than the requested relative tolerance.

The Augmented Lagrangian Solver Algorithm

Denoting the main components in step i of the algorithm U_i and the corresponding augmented-Lagrangian components V_i, the following steps describe the algorithm:

1. Initialize U_0 and V_0, and set $i = 0$.
2. Solve the nonlinear problem for $U = U_{i+1}$ with U_i as initial data and with $V = V_i$ held fixed.
3. Solve the linear problem for $V = V_{i+1}$ with $U = U_{i+1}$ held fix.
4. If $|V_{i+1} - V_i|/V_{i+1} \le \delta_V$ and $|U_{i+1} - U_i|/U_{i+1} \le \delta_U$, or $i > i_{\max}$, then terminate, else set $i = i+1$ and go to Step 2.

This procedure is called Uzawa iterations (or segregated iterations). The value in the **Tolerance** edit field controls the tolerance δ_V in the convergence criterion, and you control the other tolerance δ_U using the **Relative tolerance** edit field in the **Nonlinear settings** area. The value in the **Maximum number of iterations** edit field controls the $i_{\max}$ parameter in Step 4. You can choose the linear system solver used for Step 3 in the **Solver** list. The Structural Mechanics Module uses this process to solve contact problems with the augmented-Lagrangian technique. See the *Structural Mechanics Module User's Guide* for more information.

The Time-Dependent Solver Algorithm

The finite element discretization of the time-dependent PDE problem is

$$\begin{aligned} 0 &= L(U, \dot{U}, \ddot{U}, t) - N_F(U, t)\Lambda \\ 0 &= M(U, t) \end{aligned},$$

which is often referred to as the *method of lines*. Before solving this system, the algorithm eliminates the Lagrange multipliers Λ. If the constraints $0 = M$ are linear and time independent and if the constraint force Jacobian N_F is constant then the algorithm also eliminates the constraints from the system. Otherwise it keeps the constraints, leading to a differential-algebraic system.

In COMSOL Multiphysics, the two solvers IDA and generalized-α are available to solve the above ODE or DAE system.

IDA was created at the Lawrence Livermore National Laboratory (see Ref. 2) and is, in turn, a modernized implementation of the DAE solver DASPK (see Ref. 3), which uses variable-order variable-step-size backward differentiation formulas (BDF).

Generalized-α is an implicit, second-order accurate method with a parameter, $0 \leq \rho_\infty \leq 1$, to control how much high frequencies are damped. With $\rho_\infty = 1$, the method has no numerical damping. For linear problems, this corresponds to the midpoint rule. The choice $\rho_\infty = 0$ gives the maximal numerical damping; for linear problems, the highest frequency is then annihilated in one step. The method was first developed for the second-order equations in structural mechanics (see Ref. 4) and later extended to first-order systems (see Ref. 5).

Because the time-stepping schemes used are implicit, a possibly nonlinear system of equations must be solved each time step. Included in IDA is a Newton solver to solve this nonlinear system of equations. The Newton solver, in turn, uses an arbitrary COMSOL Multiphysics linear solver for the resulting linear systems. Alternatively, you can use the COMSOL Multiphysics Newton solver (see "The Nonlinear Solver Algorithm" on page 535) to solve the nonlinear system (by selecting **Manual tuning of nonlinear solver**). This gives you more control of the nonlinear solution process; it is possible to choose the nonlinear tolerance, damping factor, how often the Jacobian is updated, and other settings such that the algorithm solves the nonlinear system more efficiently. When you select the generalized-α time-stepping algorithm, COMSOL Multiphysics uses the Newton solver.

The linearization of the above system used in the Newton iteration is

$$E\ddot{V} + D\dot{V} + KV = L - N_F\Lambda$$
$$NV = M$$

where $K = -\partial L/\partial U$ is the stiffness matrix, $D = -\partial L/\partial \dot{U}$ is the damping matrix, and $E = -\partial L/\partial \ddot{U}$ is the mass matrix. When $E = 0$, D is often called the mass matrix.

When using IDA for problems with second-order time derivatives ($E \neq 0$), extra variables are internally introduced so that it is possible to form a first-order time-derivative system (this does not happen when using generalized-α because it can integrate second-order equations). The vector of extra variables, here U_v, comes with the extra equation

$$\dot{U} = U_v$$

where U denotes the vector of original variables. This procedure expands the original ODE or DAE system to double its original size, but the linearized system is reduced to the original size with the matrix $E + \sigma D + \sigma^2 K$, where σ is a scalar inversely proportional to the time step. By the added equation the original variable U is therefore always a differential variable (index-0). The error test excludes the variable U_v unless **Consistent initialization of DAE systems** is set to **On**, in which case the differential U_v-variables are included in the error test and the **Error estimation strategy** applies to the algebraic U_v-variables.

The Eigenvalue Solver Algorithm

Finite element discretization leads to the generalized eigenvalue system

$$(\lambda - \lambda_0)^2 EU - (\lambda - \lambda_0)DU + KU + N_F\Lambda = 0$$
$$NU = 0$$

where the solver evaluates E, D, K, N and N_F for the solution vector U_0, λ denotes the eigenvalue, and λ_0 is the linearization point. If $E = 0$, it is a linear eigenvalue problem; if E is nonzero, it is a quadratic eigenvalue problem. To solve the quadratic eigenvalue problem, COMSOL Multiphysics reformulates it as a linear eigenvalue problem. After constraint handling, it is possible to write the system in the form $Ax = \lambda Bx$.

More general eigenvalue problems sometimes arise when boundary conditions or material properties are nonlinear functions of the eigenvalue. These cases can be handled as a series of quadratic eigenvalue problems. COMSOL Multiphysics treats general dependences on the eigenvalue by assembling a quadratic approximation around the eigenvalue linearization point λ_0. Normally, iteratively updating the linearization point leads to rapid convergence.

Finding the eigenvalues closest to the shift σ is equivalent to computing the largest eigenvalues of the matrix $C = (A - \sigma B)^{-1}B$. To do this, the solver uses the ARPACK FORTRAN routines for large-scale eigenvalue problems (Ref. 6). This code is based on a variant of the Arnoldi algorithm called the *implicitly restarted Arnoldi method* (IRAM). The ARPACK routines must perform several matrix-vector multiplications Cv, which they accomplish by solving the linear system $(A - \sigma B)x = Bv$ using one of the linear system solvers.

The Parametric Solver Algorithm

The parametric solver performs a loop around the usual stationary solver in which it estimates the initial guess using the solution for the previous parameter value. If the nonlinear solver does not converge and you are solving for a single parameter, it tries a smaller parameter step; COMSOL Multiphysics determines the size of this step on the basis of the convergence speed of the Newton iteration using step-size selection criteria based on work in Ref. 7.

The Stationary Segregated Solver Algorithm

CONVERGENCE CRITERION

When termination of the segregated solver is based on the estimated error, it terminates if, for all the groups j, the error estimate is smaller than the corresponding tolerance,

$$\mathrm{err}_{j,k} < \mathrm{tol}_j ,$$

where the error estimate in segregated iteration k is

$$\mathrm{err}_{j,k} = \max(\mathrm{e}^N_{j,k}, \mathrm{e}^S_{j,k}) .$$

The number tol_j is taken from the **Relative tolerance** edit field for the corresponding group settings for the **Stationary segregated** solver on the **General** page of the **Solver Parameters** dialog box. Furthermore,

$$e_{j,k}^{N} = \max_{l}(1-\alpha_{l})\left[\frac{1}{N_j}\sum_{i=1}^{N_j}\left(\frac{|(\Delta U^{l,j,k})_i|}{W^j_i}\right)^2\right]^{1/2}$$

is an estimate of the largest damped Newton error. Here l is taken for all iterations in all substeps solving for the group j, α_l is the damping factor, $\Delta U^{l,j,k}$ is the Newton increment vector, and N_j is the number of DOFs. The weight factor W^j_i is described below. Moreover,

$$e_{j,k}^{S} = \left[\frac{1}{N_j}\sum_{i=1}^{N_j}\left(\frac{|(U^{j,k}-U^{j,k-1})_i|}{W^j_i}\right)^2\right]^{1/2},$$

is the relative increment over one complete iteration k. In this expression, $U^{j,k}$ is the segregated solution vector for the group j, and $W^j_i = \max(|U^j_i|, S_i)$, where S_i is a scale factor that the solver determines from the settings in the **Scaling of variables** area on the **Advanced** page.

The following choices are available in the **Type of scaling** list:

- For **Automatic**, S_i is the factor 0.1 times the average of $|U_m|$ for all DOFs m having the same name as DOF i.
- For **Manual**, S_i is the value for DOF i given in the **Manual scaling** edit field.
- For **Initial value based**, S_i is the factor 0.1 times the average of $|U_{0m}|$ for all DOFs m having the same name as DOF i, where U_0 is the solution vector corresponding to the initial value.
- For **None**, $W_i = 1$.

The Adaptive Solver Algorithm

THE L2 NORM ESTIMATE

The L2 norm error estimate relies on an assumption of a strong stability estimate for the PDE problem (satisfied, for example, for Poisson's equation over a domain with a smooth boundary). From such an assumption, it is possible to show that there is a constant C, such that the L2 norm of the error, e_l, in the lth equation satisfies

$$\|e_l\| \le C\left\|h^{q_l}\rho_l\right\|$$

where ρ_l is the residual in the lth equation and q_l is the stability estimate derivative order. The adaptive solver algorithm uses the following L2-norm error indicator:

$$\left(\int_{\Omega}\sum_{l} s_l^{-2} h^{2q_l} |\rho_l|^2 dA\right)^{\frac{1}{2}}$$

with the default value $q_l = 2$. This formula also introduces the scaling factors s_l for the residual with the default value $s_l = 1$.The local error indicator for a mesh element is

$$\sum_{l} s_l^{-2} h^{2q_l} \tau_l^2 A$$

where A is the area (volume, length) of the mesh element, and τ_l is the absolute value of the lth equation residual (one number per mesh element).

THE FUNCTIONAL ESTIMATE

The functional-based estimate relies on adjoint solution error estimation. Instead of approximating the error of the solution, the adaptive solver uses the approximation of the error of a certain error functional (Ref. 8). Under rather general assumptions, it is possible to show that the error e (of a functional) can be estimated as

$$|e| \le \sum_{l} \|e_l^*\| \|\rho_l\|$$

where e_l^* and ρ_l are the error in the dual or adjoint solution to, and the residual for, the lth equation, respectively. The adaptive solver algorithm uses the following error indicator for a mesh element:

$$\sum_{l} w_l \tau_l A$$

where A is the area (volume, length) of the mesh element, and τ_l is the absolute value of the lth equation residual (one number per mesh element). Here w_l is an estimate of the adjoint solution error for the lth equation. This error is estimated in one of two ways. For both methods the sensitivity solver finds the discrete adjoint solution. If only Lagrange element basis functions are used, the solver uses the *ppr* technique to compute w_l as an element average of $|\mathrm{pprint}(u_l^*) - u_l^*|$. Here u_l^* is the current computed adjoint solution for the lth equation. If not only Lagrange-element basis functions are used, then $w_l = hD_l$ where D_l is an element average of $|\nabla u_l^*|$. The global error printed in the solver log is the sum of the error indicator for all the mesh elements.

THE RESIDUAL METHODS

The residual methods *Weak* and *Coefficient* compute τ_l in rather different ways.

The Coefficient residual method uses an explicit *strong form* of the PDE to compute the equation residual τ_l on each mesh element. This method evaluates the PDE residual at the center of each element takes normal flux jumps to neighboring elements into account. However, it does not take residual contributions from equations formulated with the Weak solution form into account. Neither does it add terms in the Weak edit fields, and constraint forces do not contribute to the residual. Because there is no compelling reason to use the Coefficient residual method, you should therefore avoid selecting it. It is provided for backward compatibility only.

The Weak residual method (the default) uses an auxiliary mesh case to estimate the residualτ_l. This method automatically generates the mesh case by increasing the order of the shape functions used (by one) for the problem considered, while using the same mesh. The solution is mapped to this auxiliary mesh case and the discrete residual vector L is assembled. The equation residualτ_l for a mesh element is computed by:

- Finding how the lth field variable (dependent variable) is coupled to the degrees of freedom
- Taking an average per element for the corresponding components of L.

Due to the extra residual assembly work, the Weak residual method is somewhat slower than the Coefficient residual method. On the other hand, the Weak method is more general; for example, it supports vector elements. It also takes boundary fluxes into account. Degrees of freedom that are constrained do not contribute to the residual.

The adaptive solver performs the following iterative algorithm (Ref. 9):

1. Solve the problem on the existing mesh using the stationary or eigenvalue solver.
2. Evaluate the residual of the PDE on all mesh elements.
3. Estimate the error in the solution on all mesh elements. The computed error estimate is really just an *error indicator* because the estimate involves an unknown constant (C above).
4. Terminate execution if it has made the requested number of refinements or if it has exceeded the maximum number of elements.
5. Refine a subset of the elements based on the sizes of the local error indicators.
6. Repeat from Step 1.

The Sensitivity Solver Algorithm

When you enable sensitivity analysis, the stationary solvers compute—in addition to the basic forward solution—the sensitivity of a functional

$$Q = Q(u_p, p) \tag{5-1}$$

with respect to the sensitivity variables p. The forward solution u_p is a solution to the parameterized discrete forward problem

$$L(u_p, p) = N_F\Lambda_p \qquad M(u_p, p) = 0 \tag{5-2}$$

where Λ_p are the constraint Lagrange multipliers, or (generalized) reaction forces, corresponding to the constraints M. Note that it is assumed that Q does not explicitly depend on Λ_p.

To compute the sensitivity of Q with respect to p, first apply the chain rule:

$$\frac{dQ}{dp} = \frac{\partial Q}{\partial p} + \frac{\partial Q}{\partial u}\frac{\partial u}{\partial p} \tag{5-3}$$

In this expression, the sensitivity of the solution with respect to the sensitivity variables, $\partial u/\partial p$, is still an unknown quantity. Therefore, differentiate the forward problem, Equation 5-2, formally with respect to p:

$$K\frac{\partial u_p}{\partial p} + N_F\frac{\partial \Lambda_p}{\partial p} = \frac{\partial L}{\partial p} + \frac{\partial N_F}{\partial p}\Lambda_p \qquad N\frac{\partial u_p}{\partial p} = \frac{\partial M}{\partial p}$$

Here, $K = -\partial L/\partial u$ and $N = -\partial M/\partial u$ as usual. Assuming that the constraint force Jacobian N_F is independent of p, that is, $\partial N_F/\partial p = 0$, you can write the above relations in matrix form

$$J\begin{pmatrix} \frac{\partial u_p}{\partial p} \\ \frac{\partial \Lambda_p}{\partial p} \end{pmatrix} = \begin{pmatrix} \frac{\partial L}{\partial p} \\ \frac{\partial M}{\partial p} \end{pmatrix} \qquad J = \begin{bmatrix} K & N_F \\ N & 0 \end{bmatrix} \tag{5-4}$$

solve for the sensitivities $\partial u_p/\partial p$ and $\partial \Lambda_p/\partial p$, and plug them back into Equation 5-3:

$$\frac{dQ}{dp} = \frac{\partial Q}{\partial p} + \begin{pmatrix} \frac{\partial Q}{\partial u} \\ 0 \end{pmatrix}^T J^{-1}\begin{pmatrix} \frac{\partial L}{\partial p} \\ \frac{\partial M}{\partial p} \end{pmatrix} \tag{5-5}$$

This formula gives dQ/dp explicitly in terms of known quantities, but in practice, it is too expensive to actually invert the matrix J.

If the number of individual sensitivity variables, p_j, is small, Equation 5-4 can be solved for each right-hand side $[\partial L/\partial p_j\ \partial M/\partial p_j]^T$ and the solution inserted into Equation 5-3. This is the *forward method*, which in addition to the sensitivity dQ/dp returns the sensitivity of the solution, $\partial u_p/\partial p$. Note that the matrix J is in fact the same matrix as in the last linearization of the forward problem. The forward method therefore requires one additional back-substitution for each sensitivity variable.

If there are many sensitivity variables and the sensitivity of the solution itself, $\partial u_p/\partial p$, is not required, the *adjoint method* is more efficient. It is based on using auxiliary variables u^* and L^*, known as the *adjoint solution*, to rewrite Equation 5-5:

$$\frac{dQ}{dp} = \frac{\partial Q}{\partial p} + \begin{pmatrix} u^* \\ \Lambda^* \end{pmatrix}^T \begin{pmatrix} \dfrac{\partial L}{\partial p} \\ \dfrac{\partial M}{\partial p} \end{pmatrix}$$

$$J^T \begin{pmatrix} u^* \\ \Lambda^* \end{pmatrix} = \begin{pmatrix} \dfrac{\partial Q}{\partial u} \\ 0 \end{pmatrix}$$

On this form, only one linear system of equations must be solved regardless of the number of sensitivity variables, followed by a simple scalar product for each variable. Obviously, this is much faster than the forward method if the number of variables is large. Note that the system matrix which must be factorized is the transpose of the last linearization of the forward problem. If J is symmetric or Hermitian, this makes no difference compared to the forward method and all direct solvers can reuse the factorization. In the general case, however, UMFPACK is more efficient than the others since it can reuse the factorization of J to perform back-substitution on J^{-T}.

References

1. P. Deuflhard, "A modified Newton method for the solution of ill-conditioned systems of nonlinear equations with application to multiple shooting," *Numer. Math.*, vol. 22, pp. 289–315, 1974.

2. A.C. Hindmarsh, P.N. Brown, K.E. Grant, S.L. Lee, R. Serban, D.E. Shumaker, and C.S. Woodward, "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers," *ACM T. Math. Software*, vol. 31, p. 363, 2005.

3. P.N. Brown, A.C. Hindmarsh, and L.R. Petzold, “Using Krylov methods in the solution of large-scale differential-algebraic systems,” *SIAM J. Sci. Comput.*, vol. 15, pp. 1467–1488, 1994.

4. J. Chung, G.M. Hulbert, “A time integration algorithm for structural dynamics with improved numerical dissipation: The generalized-α method,” *J. Appl. Mech.*, vol. 60, pp. 371–375, 1993.

5. K.E. Jansen, C.H. Whiting, G.M. Hulbert, “A generalized-α method for integrating the filtered Navier–Stokes equations with a stabilized finite element method,” *Comput. Methods Appl. Mech. Engrg.*, vol. 190, pp. 305–319, 2000.

6. The ARPACK Arnoldi package, http://www.caam.rice.edu/software/ARPACK.

7. P. Deuflhard, “A Stepsize Control for Continuation Methods and its Special Application to Multiple Shooting Techniques,” *Numer. Math.*, vol. 33, pp. 115–146, 1979.

8. R. Rannacher, “A feed-back approach to error control in finite element methods: Basic analysis and examples,” *East-West J. Numer. Math*, vol. 4, pp. 237–264, 1996.

9. R. Verfürth, *A Review of a Posteriori Error Estimation and Adaptive Mesh-Refinement Techniques*, Teubner Verlag and J. Wiley, Stuttgart, 1996.

10. http://www.netlib.org/ode.

Linear System Solvers

The UMFPACK Direct Solver

UMFPACK is the default linear system solver in most application modes. It solves general systems of the form $Ax = b$ using the nonsymmetric-pattern multifrontal method and direct LU factorization of the sparse matrix A. It employs the COLAMD and AMD approximate minimum degree preordering algorithms to permute the columns so that the fill-in is minimized. The code, written in C, uses level-3 BLAS (Basic Linear Algebra Subprograms) for optimal performance. COMSOL Multiphysics uses UMFPACK version 4.2 written by Timothy A. Davis (Ref. 1).

In the **Linear System Solver Settings** dialog box you can set the *memory-allocation factor*, a positive number (default = 0.7). The solver makes a rough estimate of the required memory before performing the calculations. The memory-allocation factor dictates how much memory COMSOL Multiphysics should allocate. A value of 0.7 results in using 70% of the estimate. A low allocation factor saves memory, but the simulation might then run much slower.

If you select the **Check tolerances** check box, COMSOL Multiphysics estimates and checks the error after the solution phase. For information about the error estimate see the section "Convergence Criteria" on page 552. By default the error estimate is turned off for UMFPACK.

You can also set the *pivot threshold*, a number between 0 and 1 (default = 0.1). The solver permutes rows for stability. In any given column the algorithm accepts an entry as a pivot element if its absolute value is greater than or equal to the pivot threshold times the largest absolute value in the column. A low pivot threshold means less fill-in and thus saves memory. If the default setting leads to poor accuracy in the linear solution, try to increase the pivot threshold from the default to, for example, 1 (which means that the linear solver always applies partial pivoting). This action can lead to a more stable solution process and a more accurate solution of the linear systems.

When using UMFPACK as a preconditioner, you can also provide a *drop tolerance* in the range 0 to 1. A value of 0.01 means that it drops matrix entries smaller than 1% of the maximum value in each column of the LU factors. Doing so reduces the size of the factors and reduces memory requirements. However, the dropping occurs only when writing the LU factors, and it does not affect the rest of the factorization process. In contrast, in the Incomplete LU preconditioner the element dropping affects the rest of the factorization process, which leads to a more memory-efficient preconditioner.

The SPOOLES Direct Solver

The SPOOLES solver works on general systems of the form $Ax = b$ using the multifrontal method and direct LU factorization of the sparse matrix A. When the matrix A is symmetric or Hermitian, the solver uses an LDLT version of the algorithm, which saves half the memory. SPOOLES uses several preordering algorithms to permute the columns and thereby minimize the fill-in. SPOOLES is multithreaded on platforms that support multithreading. The code is written in C. COMSOL Multiphysics uses SPOOLES version 2.2 developed by Cleve Ashcraft and collaborators (Ref. 2).

In the **Linear System Solver Settings** dialog box you choose among the following preordering algorithms:

- Minimum degree
- Nested dissection (the default algorithm)
- Multisection
- The best of nested dissection and multisection

If you select the **Check tolerances** check box, COMSOL Multiphysics estimates and checks the error after the solution phase. For information about the error estimate see the section "Convergence Criteria" on page 552. By default the error estimate is turned off for SPOOLES.

You can also specify a pivot threshold in the range of 0 to 1 (default = 0.1). When using SPOOLES as a preconditioner, you can provide a drop tolerance in the range of 0 to 1 (see "The UMFPACK Direct Solver" on page 546).

The PARDISO Direct Solver

The parallel sparse direct linear solver PARDISO works on general systems of the form $Ax = b$. In order to improve sequential and parallel sparse numerical factorization performance, the solver algorithms are based on a Level-3 BLAS update, and they exploit pipelining parallelism with a combination of left-looking and right-looking supernode techniques. The code is written in C and Fortran. COMSOL Multiphysics uses the PARDISO version developed by Olaf Schenk and collaborators (Ref. 3), which is included with Intel MKL (Intel Math Kernel Library).

In the **Linear System Solver Settings** dialog box you choose among the following preordering algorithms:

- Minimum degree
- Nested dissection (the default algorithm)

You can also specify if the solver should use a maximum weight matching strategy by choosing row preordering on (default) or off. For symmetric matrices there is a choice between using 2-by-2 Bunch-Kaufmann pivoting (default) or not. In the case of positive definite matrices (see "Which Models Are Positive Definite?" on page 433 of the *COMSOL Multiphysics User's Guide*) you do not need row preordering and 2-by-2 Bunch-Kaufmann pivoting. The solution time is usually reduced if you deselect these features.

To avoid pivoting, PARDISO uses a pivot perturbation strategy that tests the magnitude of the potential pivot against a constant threshold of $\varepsilon = \alpha|PP_{\mathrm{MPS}}D_rAD_cP|_\infty$, where P and P_{MPS} are permutation matrices, D_r and D_c are diagonal scaling matrices, and $|.|_\infty$ is the infinity norm (maximum norm). If the solver encounters a tiny pivot during elimination, it sets it to $\mathrm{sign}(l_{ii})\varepsilon|PP_{\mathrm{MPS}}D_rAD_cP|_\infty$. You can specify the pivot threshold ε. The perturbation strategy is not as robust as ordinary pivoting. In order to improve the solution PARDISO uses iterative refinements.

PARDISO also includes out-of-core capabilities. The PARDISO out-of-core solver stores the LU factors on the hard drive. This minimizes the internal memory usage. The price is longer solution times because it takes longer time to read and write to disk than using the internal memory. You can specify the temporary directory where PARDISO stores the LU factors using the `-tmpdir` switch; see page 53 of the *COMSOL Multiphysics Installation and Operations Guide* for further details. The LU factors are stored as blocks on the hard drive. The **In core memory** option controls the maximum amount of internal memory (in megabytes) allowed for the blocks. If a block is too large to fit into the maximum allowed internal memory you get an out-of-memory error. In that case you must increase the amount of internal memory that you enter in the **In core memory** edit field. The default value is 512 MB.

COMSOL Multiphysics can optionally estimate and check the error after the solution phase. You control this option through the **Check tolerances** list. For the **Automatic** selection, error checking is at least done for problems where pivot perturbation or iterative improvement has been used. For information about the error estimate, see the section "Convergence Criteria" on page 552. By default the error checking is enabled (**On**). You can disable the check by instead selecting **Off**.

For information about running COMSOL Multiphysics using parallelism, see "Shared-Memory Parallel COMSOL" on page 68 in the *COMSOL Installation and Operations Guide*.

Note: PARDISO is available on Linux, Windows, and Intel Mac. On Sun and PowerPC Mac, COMSOL Multiphysics uses SPOOLES instead.

The TAUCS Cholesky Direct Solver

The TAUCS Cholesky direct solver handles systems of the form $Ax = b$, where A is a positive definite symmetric sparse matrix (see "Which Models Are Positive Definite?" on page 433 of the *COMSOL Multiphysics User's Guide*) using a multifrontal supernodal Cholesky factorization. It employs the multiple minimum degree reordering algorithm to permute the rows and columns and thus minimize the fill-in. Written in C, it uses level-3 BLAS for maximum performance. COMSOL Multiphysics uses TAUCS version 2.2 written by Sivan Toledo and collaborators (Ref. 4).

Due to the algorithm's recursive nature, it can run out of stack space for large models. If this happens, you can increase the value of the `STACKSIZE` parameter in the appropriate COMSOL startup script. For PC/Windows set the `STACKSIZE` parameter in the `\bin\comsol.opts` file, located in the COMSOL installation directory; for UNIX/Linux/Mac OS X set the `STACKSIZE` parameter in the `/bin/comsol` command, located in the COMSOL installation directory. The default value is `2m` (2 MB). For example, if you edit the script and double the value of `STACKSIZE` to `4m` (4 MB), the maximum recursion depth for the algorithm also doubles. Continue doubling the value of `STACKSIZE` until the TAUCS algorithm is successful.

Note: To use the TAUCS Cholesky and LDLT solvers, you must select **Symmetric** or **Hermitian** in the **Matrix symmetry** list in the **Solver Parameters** dialog box.

The TAUCS LDLT Direct Solver

This linear system solver handles real symmetric or Hermitian matrices. It has no parameters to set. You can use the direct LDLT (TAUCS) solver as an alternative to the SPOOLES symmetric solver.

The GMRES Iterative Solver

This linear system solver uses the restarted GMRES (generalized minimum residual) method (see Ref. 5 and Ref. 6). This is an iterative method for general linear systems of the form $Ax = b$. For fast convergence it is important to use an appropriate *preconditioner* (see "Selecting a Preconditioner" on page 429 of the *COMSOL Multiphysics User's Guide*).

The value in the **Number of iterations before restart** edit field in the **Linear System Solver Settings** dialog box specifies the number of iterations the solver performs until it restarts (the default is 50). There is no guarantee that a restarted GMRES will converge for a small restart value. A larger restart value increases the robustness of the interactive procedure, but it also increases memory use and computational time. For large problems, the computational cost is often very large to produce a preconditioner of such a high quality that the termination criteria are fulfilled for a small number of iterations and for a small restart value. For those problems it is often advantageous to set up a preconditioner with a somewhat lesser quality and instead increase the restart value or iterate more steps. Doing so typically increases the condition number for the preconditioned system, so an increase in the error-estimate factor might be needed as well.

Two slightly different versions of GMRES are available in COMSOL Multiphysics. The difference between these two versions is whether left or right preconditioning is used (see "The Preconditioned Linear System" on page 435 of the *COMSOL Multiphysics User's Guide*). Select the preconditioning type from the **Preconditioning** list. The default choice is left preconditioning. Normally, the two versions of GMRES have similar convergence behavior (see Ref. 7). If the preconditioner is ill-conditioned there could, however, be differences in the behavior.

For information about the convergence criterion used by GMRES and the **Relative tolerance** and **Factor in error estimate** edit fields, see "Convergence Criteria" on page 552.

If the solver does not converge, it terminates with an error message when it reaches the value in the **Maximum number of iterations** edit field (default = 10,000).

The FGMRES Iterative Solver

This solver uses the restarted FGMRES (flexible generalized minimum residual) method (see Ref. 8). The solver is a variant of the GMRES solver that can handle a wider class of preconditioners in a robust way. You can, for example, use any iterative solver as preconditioner for FGMRES. The downside with the method is that it uses twice as much memory as GMRES for the same value in the **Number of iterations before restart** edit field. FGMRES uses right preconditioning and therefore has the same convergence criterion as right-preconditioned GMRES. If FGMRES is used together with a constant preconditioner such as, for example, the Incomplete LU preconditioner, then the FGMRES solver is identical to the right preconditioned GMRES solver.

For information about the convergence criterion used by FGMRES and the **Relative tolerance** and **Factor in error estimate** edit fields, see "Convergence Criteria" on page 552.

The Conjugate Gradients Iterative Solver

This solver uses the conjugate gradients iterative method (see Ref. 5, Ref. 9, and Ref. 10). It is an iterative method for linear systems of the form $Ax = b$ where the matrix A is positive definite and (Hermitian) symmetric (see "Which Models Are Positive Definite?" on page 433 of the *COMSOL Multiphysics User's Guide*). Sometimes the solver also works when the matrix is not positive definite, especially if it is close to positive definite. This solver uses less memory and is often faster than the GMRES solver, but it applies to a restricted set of models.

For fast convergence it is important to use an appropriate *preconditioner* (see "Selecting a Preconditioner" on page 429 of the *COMSOL Multiphysics User's Guide*), which should be positive definite and (Hermitian) symmetric.

Select the preconditioning type from the **Preconditioning** list. The default choice is left preconditioning. For the Conjugate gradient method this choice only affects the convergence criterion and not the algorithm itself. For information about the convergence criterion and the **Relative tolerance** and **Factor in error estimate** edit fields, see "Convergence Criteria" on page 552.

The BiCGStab Iterative Solver

This solver uses the biconjugate gradient stabilized iterative method (see Ref. 5 and Ref. 11) for solving general linear systems of the form $Ax = b$. The required memory and the computational time for one iteration with BiCGStab is constant; that is, the time and memory requirement does not increase with the number of iterations as it does for GMRES. BiCGStab uses approximately the same amount of memory as

GMRES uses for two iterations. Therefore, BiCGStab typically uses less memory than GMRES.

The convergence behavior of BiCGStab is often more irregular than that of GMRES. Intermediate residuals may even be orders of magnitude larger than the initial residual, which can affect the numerical accuracy as well as the rate of convergence. The iterations are restarted with the current solution as initial guess if the algorithm detects poor accuracy in the residual or the risk for stagnation.

In contrast to GMRES and conjugate gradients, BiCGStab uses two matrix-vector multiplications each iteration. This also requires two preconditioning steps in each iteration. Also, when using left preconditioned BiCGStab, an additional preconditioning step is required each iteration. That is, left preconditioned BiCGStab requires a total of three preconditioning steps in each iteration.

Select the preconditioning type from the **Preconditioning** list. The default choice is right preconditioning, since left preconditioning requires an additional preconditioning step in each iteration. For information about the convergence criterion and the **Relative tolerance** and **Factor in error estimate** edit fields, see "Convergence Criteria" on page 552.

Convergence Criteria

When you use an iterative solver COMSOL Multiphysics estimates the error of the solution while solving. Once the error estimate is small enough, as determined by the convergence criterion

$$\rho\left|M^{-1}(b-Ax)\right| < \mathrm{tol}\cdot\left|M^{-1}b\right| \tag{5-6}$$

the software terminates the computations and returns a solution. When you use a direct solver COMSOL Multiphysics can optionally make a check (**Error check**), to determine if the above convergence criterion is fulfilled after the solution step. If the error criterion is not met, the solution process is stopped an error message is given.

The definitions of M for the various solvers are:

- For UMFPACK, PARDISO, and SPOOLES, $M = LU$, where L and U are the LU factors computed by the solver.
- When using left-preconditioning with the iterative solvers GMRES, Conjugate Gradients, and BiCGStab, M is the preconditioner matrix.
- For the remaining iterative solvers, M is the identity matrix.

The convergence criterion 5-6 states that the iterations terminate when the relative (preconditioned) residual times the factor ρ is less than a tolerance tol. You can set the factor ρ in the **Factor in error estimate** edit field (default = 400). For solvers where M is equal to the identity matrix, the iterations can sometimes terminate too early with an incorrect solution if the system matrix A is ill-conditioned. For solvers where M is not equal to the identity matrix, the iterations can sometimes terminate too early if M is a poor preconditioner. If the iterations terminate too early due to an ill-conditioned system matrix or a poor preconditioner, increase the factor ρ to a number of the order of the condition number for the matrix $M^{-1}A$. Note that if ρ is greater than the condition number for the matrix $M^{-1}A$, the convergence criterion implies that the relative error is less than tol: $|x - A^{-1}b| < \mathrm{tol} \cdot |A^{-1}b|$.

LINEAR SYSTEM TOLERANCE

For the **Stationary** linear solver and the **Stationary segregated** solver, the tolerance tol in the convergence criterion 5-6 is the value specified in the **Relative tolerance** edit field in the **Linear System Solver Settings** dialog box.

For the **Stationary** nonlinear solver, tol is adaptive and based on the maximum of the number entered in the **Relative tolerance** edit field in the **Linear System Solver Settings** dialog box and the number entered in the **Relative tolerance** edit field on the **Stationary** page of the **Solver Parameters** dialog box. During the nonlinear iterations tol can, however, be larger or smaller than this number.

For the **Parametric** solvers, tol is used in the same way as for the corresponding **Stationary** solver.

When using the **Time dependent** solver and the **Time dependent segregated** solver, tol is the maximum of the number in the **Relative tolerance** edit field in the **Linear System Solver Settings** dialog box and the number in the **Relative tolerance** edit field on the **General** page of the **Solver Parameters** dialog box.

When using the **Eigenvalue** solver together with an iterative method, tol is the number in the **Relative tolerance** edit field in the **Linear System Solver Settings** dialog box. When using a direct method, tol is the number in the **Relative tolerance** edit field in the **Linear System Solver Settings** dialog box without any safety factor.

For the main components of the **Augmented Lagrangian** solver, tol is used in the same way as for the **Stationary** solver. For the **Augmentation components** the error check for the direct solvers is disabled.

References

1. http://www.cise.ufl.edu/research/sparse/umfpack.

2. http://www.netlib.org/linalg/spooles

3. http://www.pardiso-project.org/

4. http://www.tau.ac.il/~stoledo/taucs

5. Greenbaum, A., “Iterative Methods for Linear Systems,” *Frontiers in Applied Mathematics*, 17, SIAM, 1997.

6. Y. Saad and M.H. Schultz, “GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems,” *SIAM J. Sci. Statist. Comput.*, vol. 7, pp. 856–869, 1986.

7. Y. Saad, *Iterative Methods for Sparse Linear Systems*, Boston, 1996.

8. Y. Saad, “A flexible inner-outer preconditioned GMRES algorithm,” *SIAM J. Sci. Statist. Comput.*, vol. 14, pp. 461–469, 1993.

9. M.R. Hestenes and E. Stiefel, “Methods of conjugate gradients for solving linear systems,” *J. Res. Nat. Bur. Standards*, 49, pp. 409–435, 1952.

10. C. Lanczos, “Solutions of linear equations by minimized iterations,” *J. Res. Nat. Bur. Standards*, vol. 49, pp. 33–53, 1952.

11. H.A. Van Der Vorst, “A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems,” *SIAM J. Sci. Statist. Comput.*, vol. 13, pp. 631–644, 1992.

Preconditioners for the Iterative Solvers

The Incomplete LU Preconditioner

The Incomplete LU preconditioner performs an incomplete LU factorization of the system matrix A. That is, it drops small elements during the column-oriented Gaussian elimination (see Ref. 1 and Ref. 2). Thus it saves memory, and the resulting factors L and U are approximate. The resulting preconditioner is an approximation to A. The preconditioner supports threshold drop, fill-ratio drop, and threshold pivoting. It can optionally respect the nonzero pattern in the original matrix. The preconditioner accepts matrices in symmetric and Hermitian format but expands these to full storage before factorization.

In the **Linear System Solver Settings** dialog you can select a drop rule (see the following section) from the **Drop using** list. Depending on the selected drop rule, you can specify a **Drop tolerance** or a **Fill ratio**. You can also control the drop tolerance on the **General** tab of the **Solver Parameters** dialog box either numerically (by supplying a number between 0 and 1) or using the **Memory efficiency/Precond. quality** slider. A smaller drop tolerance means that the preconditioner drops fewer elements and so the preconditioner becomes more accurate. This leads to fewer iterations in the iterative solver, but on the other hand memory requirements and preconditioning time increases. A larger drop tolerance means that the preconditioner drops more elements and so memory use and preconditioning time decrease. In this case, however, the preconditioner becomes less accurate, which leads to more iterations in the iterative solver, or, if the drop tolerance is too high, to no convergence at all. Often it is most efficient to use as high a drop tolerance as possible, that is, choose it so that the iterative solver barely converges.

You can also set the **Pivot threshold**, a number between 0 and 1 (default = 1). The solver permutes rows for stability. In any given column, if the absolute value of the diagonal element is less than the pivot threshold times the largest absolute value in the column, it permutes rows such that the largest element is on the diagonal. Thus the default value 1 means that it uses partial pivoting.

Once the approximate factors L and U have been computed, you can use the incomplete LU factorization as an iterative preconditioner/smoother (see "The SSOR, SOR, SORU, and Diagonal Scaling (Jacobi) Algorithms" on page 566). Here,

$M = (LU)/\omega$, where ω is a relaxation factor, and L and U are the approximate factors. When using incomplete LU factorization as a preconditioner or smoother, it performs a fixed number of sweeps as dictated by the value in the **Number of iterations** edit field in the **Linear System Solver Settings** dialog box (default = 1).

Specify ω in the **Relaxation factor (omega)** edit field (default = 1).

SELECTING A DROP RULE

The Incomplete LU preconditioner uses either the *threshold drop rule* (the default) or the *fill-ratio drop rule*. The preconditioner drops (neglects) an element during the elimination phase if its absolute value is smaller than the Euclidean norm of the entire column times a *drop tolerance*. In contrast, the fill-ratio drop rule limits the number of nonzeros in the incomplete factors L and U, and it keeps the largest absolute values. The number of values it keeps depends on the number of nonzeros in the corresponding column of the original matrix times a fill-ratio factor. There are two exceptions to these drop rules:

- The preconditioner never drops diagonal elements.
- The preconditioner optionally drops nonzeros in positions where the original matrix is nonzero. The default is to keep these nonzeros. To make the preconditioner drop them, clear the **Respect pattern** check box in the settings for the Incomplete LU preconditioner.

The primary problem with setting up a preconditioner is the tradeoff between resources (computer time and memory) and the preconditioner's quality. The computational cost of setting up a preconditioner with the Incomplete LU preconditioner is at least proportional to the number of nonzeros in the produced factors L and U. An advantage of using the fill-ratio drop rule is that you can estimate and limit the cost beforehand; the main disadvantage is that the quality of the preconditioner is typically not as good as using the threshold drop rule with a drop tolerance resulting in the same number of nonzeros. However, with the threshold drop rule there is no good way of estimating resource requirements beforehand. Furthermore, there is no general formula for these drop rules that gives a drop tolerance or fill ratio that guarantees fast convergence for a certain iterative method. Therefore, it is often necessary to rely on experiments and experience for this difficult and, from a performance point of view, crucial choice.

The TAUCS Incomplete Cholesky Preconditioner

This TAUCS incomplete Cholesky preconditioner is applicable to models where the system matrix is (Hermitian) symmetric positive definite (see “Which Models Are Positive Definite?” on page 433 of the *COMSOL Multiphysics User’s Guide*). It performs an incomplete Cholesky factorization LL^T of the system matrix A. The resulting preconditioner $M = LL^T$ is an approximation to A. The code, written in C, uses a column-based left-looking approach with row lists. COMSOL Multiphysics uses TAUCS version 2.2 written by Sivan Toledo and collaborators (see http://www.tau.ac.il/~stoledo/taucs).

In the **Linear System Solver Settings** dialog box you can specify a value for the drop tolerance and select an option for a modified Cholesky factor. You can also control the drop tolerance on the **General** tab of **Solver Parameters** dialog box either numerically (giving a number between 0 and 1) or using the **Memory efficiency/Precond. quality** slider. The preconditioner drops elements from the L matrix if they are smaller than the drop tolerance times the norm of the corresponding column of L, provided that they are not on the diagonal and not in the nonzero pattern of A (see “The Incomplete LU Preconditioner” on page 555 for details on the implications of changing the drop tolerance). If you select the **Modified Cholesky** check box (which is not the default state) the preconditioner modifies the factor L so that the row sums of LL^T are equal to the row sums of the input matrix.

Note: To use the incomplete Cholesky preconditioner you must select **Symmetric** or **Hermitian** in the **Matrix symmetry** list in the **Solver Parameters** dialog box.

The Geometric Multigrid Solver/Preconditioner

The Geometric multigrid solver or preconditioner is a fast and memory-efficient iterative method for elliptic and parabolic models (see “Elliptic and Parabolic Models” on page 434 of the *COMSOL Multiphysics User’s Guide*). It performs one or several cycles of the geometric multigrid method. The classical multigrid algorithm uses one or several auxiliary meshes that are coarser than the original (fine) mesh. The idea is to perform just a fraction of the computations on the fine mesh. Instead, it performs computations on the coarser meshes when possible, which leads to fewer operations. The size of the extra memory used for the coarser meshes and associated matrices is comparable to the size of the original data. This leads to an iterative algorithm that is both fast and memory efficient. See Ref. 3 for more information.

COMSOL Multiphysics uses a hierarchy of *multigrid levels* where each corresponds to a mesh and a choice of shape functions. Thus, in addition to coarsening the mesh it is possible to construct a new "coarser" level by lowering the order of the shape functions. The number of degrees of freedom decreases when you go to a coarser multigrid level. The meshes for the different levels can be constructed either "manually" or automatically. The automatic version applies a coarsening algorithm to the fine mesh, which leads to meshes that are not aligned to each other. There is also an option to generate the finer meshes from the coarsest mesh by successive mesh refinements, which leads to aligned (nested) meshes. The manual option can be useful when you have a quadrilateral, hexahedral, or prism mesh, or when you for some other reason wish to control details in the meshes.

To describe the multigrid algorithm, assume that you have $N+1$ multigrid levels numbered from 0 to N, where 0 is the finest level (the level for which you seek the solution). To solve the linear system $A_0x = b$ (corresponding to level 0), the algorithm must reform the system matrices $A_1, \ldots, A_N$ for the coarse multigrid levels. It must also compute the *prolongation matrices* P_i that map a solution x vector on level i to the corresponding solution vector P_ix on the next finer level $i-1$.

The prolongation matrices are constructed using plain interpolation from one multigrid level to the other. The system matrices for the coarse levels can be constructed in two ways:

- By assembling A_i on the mesh of level i (the default method).
- By projection from the finer level: $A_i = P_i^T A_{i-1} P_i$. This is also called the Galerkin method. It typically leads to more nonzero elements in the system matrix A_i, but the convergence can be faster than in the default method.

The following algorithm describes one multigrid cycle:

1. The input to the algorithm is some initial guess x_0 to the solution of the system $A_0x = b$.
2. Starting with x_0, apply a few iterations of a *presmoother* to the linear system $A_0x = b$, yielding a more accurate iterate x_{0s}. Typically the presmoother is some simple iterative algorithm such as SOR, but you also chose an arbitrary iterative solver.
3. Compute the residual $r_0 = b - A_0x_{0s}$. The presmoother "smooths" the residual so the oscillations in r have such a long wavelength that they are well resolved on the

next coarser level (1). Therefore, project the residual onto level 1 by applying the transpose of the prolongation: $r_1 = P_1{}^T r_0$.

4 If $N = 1$ use the *coarse solver* to solve the system $A_1 x_1 = r_1$. The coarse solver is typically a direct solver such as UMFPACK. The number of degrees of freedom on level 1 is less than for level 0, which means that solving $A_1 x_1 = r_1$ is less expensive. If instead $N > 1$, solve the system $A_1 x_1 = r_1$ (approximately) by recursively applying one cycle of the multigrid algorithm for levels $1, 2, \ldots, N$. In both cases the obtained solution x_1 is called the *coarse grid correction*.

5 Map the coarse grid correction to level 0 using the prolongation matrix: $x_{0\mathrm{c}} = x_{0\mathrm{s}} + P_1 x_1$.

6 Starting with $x_{0\mathrm{c}}$, apply a few iterations of a *postsmoother* to the linear system $A_0 x = b$, yielding a more accurate iterate $x_{0\mathrm{mg}}$. The default postsmoother is SORU (the version of SOR using the upper triangle of the matrix). The iterate $x_{0\mathrm{mg}}$ is the output of the multigrid cycle.

The cycle just described is called the V-cycle. The recursive call in Step 4 (when $N > 1$) is a also a V-cycle. For the W-cycle and the F-cycle, the Steps 1–6 above are the same but with the twist that the recursive call in Step 4 is substituted with two multigrid calls for the coarser levels. For the W-cycle these two calls are recursive calls, they are W-cycle calls. For the F-cycle the first call is a W-cycle and the second a V-cycle.

For only two multigrid levels ($N = 1$) these cycles are the same because the algorithm uses the coarse solver in Step 4. Also note that the amount of work on the finest level is the same for the different cycles. Normally the V-cycle is sufficient, but the W-cycle and the F-cycle can be useful for more difficult problems.

When using multigrid as a preconditioner, the action of this preconditioner is obtained by applying a fixed number of multigrid cycles. When using multigrid as a solver, the multigrid cycle repeats until it reaches convergence.

When using multigrid as a preconditioner for the conjugate gradients method for a symmetric matrix A, the preconditioning matrix M should also be symmetric. This requirement is fulfilled if the matrices M (see “The SSOR, SOR, SORU, and Diagonal Scaling (Jacobi) Algorithms” on page 566) associated with the presmoother and the postsmoother are transposes of each other. For instance, this is the case if the presmoother is SOR and the postsmoother is SORU, and if the same number of smoothing steps is used. This combination with two smoothing steps is the default.

NOTES ON THE EFFICIENCY OF SMOOTHERS

COMSOL Multiphysics performs smoothing on all but the coarsest multigrid level. A smoother should be computationally cheap and be effective at reducing the part of the error that has a high spatial frequency on the mesh to which it is applied. Therefore, the applying a smoother on several meshes with a hierarchy of mesh sizes results in a more efficient solver than if the smoother were applied only on the finest mesh.

The efficiency of the multigrid method with simple iterations as a smoother (such as the Jacobi and SOR iteration) hinges on the ellipticity of the underlying mathematical problem. For Helmholtz problems originating from an equation

$$-\nabla\cdot\left(\frac{1}{a}\nabla u\right)-\omega^2 u = f$$

or

$$\nabla\times\left(\frac{1}{a}\nabla\times\mathbf{E}\right)-\omega^2\mathbf{E} = \mathbf{F}$$

the obtained linear problem is indefinite for large frequencies ω. For these problems, a simple iteration amplifies smooth eigenmodes if the mesh is too coarse and makes these methods unsuitable as smoothers. To determine when to use a simple iteration, apply the *Nyquist criterion*

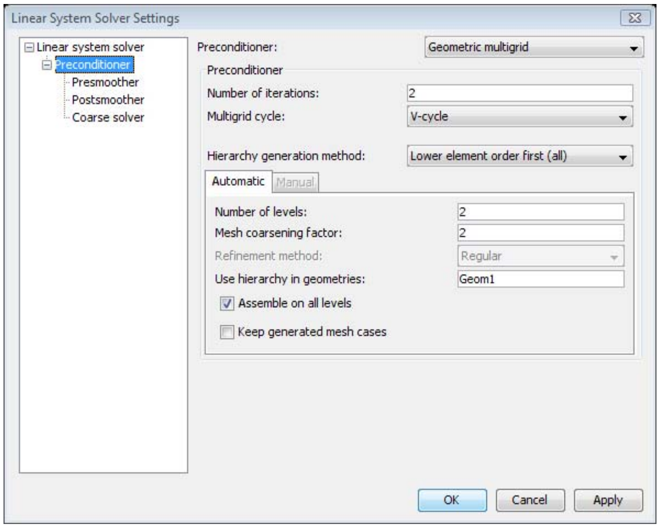
$$h_{\max}<\frac{\lambda}{2} = \frac{\pi}{\omega\sqrt{a}}$$

which says that the mesh must have at least two mesh elements per wavelength. Thus, when using the Geometric multigrid solver for these type of problems, you should ensure that this criterion is fulfilled on the coarsest mesh if simple iterations is used as a smoother. In situations where the criterion is not fulfilled on coarse meshes GMRES can be a suitable smoother (Ref. 6).

CONSTRUCTING A MULTIGRID HIERARCHY

The multigrid hierarchy can be constructed either automatically or manually. To select which method to use, go to the **General** page of the **Solver Parameters** dialog box and click the **Settings** button. This action opens the **Linear System Solver Settings** dialog box.

If you use multigrid as a preconditioner, select **Preconditioner** in the list to the left, otherwise select **Linear system solver**.


Linear System Solver Settings

Linear system solver
Preconditioner
Presmoother
Postsmoother
Coarse solver

Preconditioner: Geometric multigrid

Preconditioner
Number of iterations: 2
Multigrid cycle: V-cycle

Hierarchy generation method: Lower element order first (all)

Automatic | Manual

Number of levels: 2
Mesh coarsening factor: 2
Refinement method: Regular
Use hierarchy in geometries: Geom1
[✓] Assemble on all levels
[ ] Keep generated mesh cases

OK | Cancel | Apply

Preconditioner settings for the Geometric multigrid solver.

In the **Hierarchy generation method** list you can select among the following methods:

- **Lower element order first (all)** (default). In this method, the coarse levels are constructed automatically from the finest level by lowering the shape-function orders in steps of one. When a given shape function order cannot be decreased more, the mesh is instead coarsened by the factor given in the **Mesh coarsening factor** edit field (default = 2). The coarsened mesh is constructed by generating a new mesh of the geometry such that the element edge size at any location is approximately equal to the mesh coarsening factor times the element edge size in the old mesh. This procedure repeats until the number of multigrid levels (including the finest level) equals the **Number of levels** (default = 2).
- **Lower element order first (any)**. In this method, the coarse levels are constructed automatically from the finest level by lowering the shape-function orders in steps of one. When none of the given shape function orders can be decreased more, the mesh is instead coarsened by the factor given in the **Mesh coarsening factor** edit field (default = 2). The coarsened mesh is constructed by generating a new mesh of the geometry such that the element edge size at any location is approximately equal to the mesh coarsening factor times the element edge size in the old mesh. This

procedure repeats until the number of multigrid levels (including the finest level) equals the **Number of levels** (default = 2).

- **Coarse mesh and lower order**. The coarse levels are constructed automatically from the finest level by coarsening the mesh and lowering the order of the shape functions at the same time. More precisely, the first coarse level is constructed by coarsening the mesh by a factor given in the **Mesh coarsening factor** edit field (default = 2) and lowering the shape function orders by 1. If some of the shape function orders cannot be decreased, only the mesh coarsening is done. This procedure repeats until the number of multigrid levels (including the finest level) equals the **Number of levels** (default = 2).
- **Coarse mesh**. The coarse levels are constructed automatically from the finest level by coarsening the mesh by the factor given in the **Mesh coarsening factor** edit field (default = 2). This procedure repeats until the number of multigrid levels (including the finest level) equals the **Number of levels** (default = 2).
- **Refine mesh**. In this method, the current mesh is refined multiple times until the number of multigrid levels (including the finest level) equals the **Number of levels** (default = 2). Thus the given mesh becomes the coarsest multigrid level, and the solution is delivered for a refined mesh. When you use the **Refine mesh** method, the software automatically selects the **Keep generated mesh cases** check box because the refined mesh is needed in postprocessing. You can also make a selection from the **Refinement method** list. Selecting **Regular** (the default) refines the elements in a regular pattern, whereas **Longest** refines only the longest element edge. For 3D models, we recommend the **Longest** method because it produces fewer mesh elements. The **Longest** method is not available for 1D models.
- **Manual**. When you select this method, the **Manual** page shows a list of all available *mesh cases*. The term mesh case refers to a mesh together with the choice of shape functions (and corresponding integration orders and constraint-point orders). You can construct new ones by going to the **Mesh** menu and choosing **Mesh Cases**, which opens the **Mesh Case Settings** dialog box where you add and delete them. Each mesh case is identified by a nonnegative number. Existing mesh cases appear at the bottom of the **Mesh** menu, where the *current mesh case* is also indicated. The current mesh case also appears in the status bar. The mesh, shape functions, integration orders, and constraint-point orders are specific to the mesh case. To change those settings for a mesh case, first make that mesh case the current one by selecting it in the list on the **Mesh** menu. Then modify any desired settings, for instance changing mesh parameters and generating a new mesh, or changing the shape functions in the

Subdomain Settings dialog box. For more information, see “Mesh Cases” on page 373 of the *COMSOL Multiphysics User’s Guide*.

When you have defined some mesh cases, go to the settings for **Geometric multigrid** in the **Linear System Solver Settings** dialog box. On the **Manual** page you can choose for each mesh case whether it should be used in the multigrid hierarchy (if so, select the **Use** check box) and whether the system matrix should be assembled on this level (if so, select the **Assemble** check box). By default the hierarchy includes all mesh cases, and matrices are assembled on the coarse levels. The solver sorts the multigrid levels according to decreasing number of degrees of freedom. The solution is delivered for the finest of the selected mesh cases, and that mesh case is made current when the solver returns.

If you select the **Assemble on all levels** check box on the **Automatic** page, the solver assembles the system matrices on the coarse levels (the default). Otherwise the coarse level matrices are formed using the Galerkin projection method.

When using the automatic hierarchy generation methods, the default behavior is that the solver deletes the coarse levels when it finishes. If you want to inspect them, select the **Keep generated mesh cases** check box, which makes the generated levels appear as new mesh cases. When this happens, the hierarchy generation method changes to the manual method. This means that the solver reuses the generated mesh cases the next time you solve, which saves some work.

The automatic hierarchy generation methods operate only on the geometries specified in the **Use hierarchy in geometries** edit field, where you provide a space-separated list of positive numbers. The mesh coarsening and shape function changes are applied only to these geometries.

Note: The automatic hierarchy generation methods construct coarsened meshes consisting of isotropic triangles or tetrahedra. If the original mesh contains quadrilaterals, hexahedrons, or prisms, or if it is anisotropic, you get better results by constructing the coarse meshes manually.

CONTROLLING THE ASSEMBLY ON COARSE LEVELS

There is a variable defined with the name `gmg_level` that takes the values 0 on the level where the solution is sought and increases with one for each level that is used in the hierarchy (that is, `gmg_level=1`,2, ..., and so on). When assembling the matrices on the coarse levels, this variable can be used in equations, for example to add extra artificial stabilization only for the coarse levels.

SETTINGS FOR THE MULTIGRID SOLVER/PRECONDITIONER

Apart from settings controlling the multigrid hierarchy, you can specify the following settings in the **Linear System Solver Settings** dialog box. If multigrid is used as a preconditioner, you can specify the *number of iterations* (default = 2). This gives the number of times the multigrid cycle is performed each time the preconditioner is applied.

If you use multigrid as a linear system solver, you can instead specify a *relative tolerance*, a *factor in error estimate*, and a *maximum number of iterations*. For information about the convergence criterion used by multigrid and the **Relative tolerance** and **Factor in error estimate** edit fields, see "Convergence Criteria" on page 552. The tolerance in the convergence criterion is determined by the nonlinear stationary solver or the time-dependent solver. When using the linear stationary solver or the eigenvalue solver, you can adjust the tolerance in the **Relative tolerance** edit field (default = 10^{-6}).

If the solver does not converge, it terminates with an error message when it reaches the value in the **Maximum number of iterations** edit field (default = 10,000).

You can also select the type of multigrid cycle: V-cycle, W-cycle, or F-cycle.

SETTINGS FOR THE SMOOTHERS

To control the settings for the presmoother, select one in the list on the left side of the **Linear System Solver Settings** dialog box. In the **Presmoother** list you can select among the following smoothers: SOR (default), SORU, SSOR, SOR vector, SORU vector, SSOR vector, SOR gauge, SORU gauge, SSOR gauge, Jacobi, Vanka, Incomplete LU, GMRES, FGMRES, Conjugate gradients, BiCGStab, and Algebraic multigrid. Change settings for the selection in the **Presmoother** area (see the sections on the specific smoothers in the following sections). For instance, it is possible to control the number of smoothing iterations here.

You control settings for the postsmoother in a similar fashion. The default postsmoother is SORU (the version of SOR using the upper triangle of the matrix).

When solving an electromagnetics model using vector elements for a PDE involving the curl-curl operator, you should select the SOR vector presmoother and the SORU vector postsmoother.

When solving fluid-dynamics problems using the incompressible Navier-Stokes equations or when using weak constraints, an *algebraic saddle-point problem* results. Such problems often have zeros on the diagonal of the system matrix, which makes the standard smoothers fail. Use the Vanka smoother (or Incomplete LU) in that case.

SETTINGS FOR THE COARSE SOLVER

To control the settings for the coarse solver, select its name in the list to the left in the **Linear System Solver Settings** area. In the **Coarse solver** list you can choose from the following: UMFPACK, SPOOLES, PARDISO, TAUCS Cholesky (if **Symmetric** is selected), GMRES, FGMRES, Conjugate gradients, BiCGStab, Algebraic multigrid, SSOR, SOR, SORU, SSOR vector, SSOR gauge, and Jacobi. Normally choose a direct solver (UMFPACK, SPOOLES, PARDISO, or TAUCS Cholesky). Make any desired modifications to the settings in the **Coarse solver** area (refer to the sections on the specific linear system solvers).

When an iterative solver is used as coarse solver you can choose whether to solve using a tolerance (default) or to perform a fixed number of iterations. Choose either **Use tolerance** or **Fixed number of iterations** in the **Termination** list. Note that some default values for an iterative solver, when used as a coarse solver, are different from the default values when the solver is used as a linear system solver, preconditioner, or smoother. The edit fields that have different default values are: **Relative tolerance** (default = 0.1), **Factor in error estimate** (default = 1), **Maximum number of iterations** (default = 500), and **Number of iterations** (default = 10).

The Algebraic Multigrid Solver/Preconditioner

The algebraic multigrid solver or preconditioner performs one or several cycles of the algebraic multigrid method. This is similar to the geometric multigrid algorithm (see "The Geometric Multigrid Solver/Preconditioner" on page 557), the difference being that it constructs the multigrid levels directly from the finest-level system matrix A_0. That is, it constructs the prolongations P_i from A_0 without using auxiliary meshes. It constructs the coarse level matrices A_i from A_0 with the Galerkin projection method. The advantage is that you need not bother about the coarse multigrid levels. The disadvantages are twofold:

- Algebraic multigrid does not work well for vector-valued PDEs in COMSOL's implementation. That is, it handles only scalar PDEs.
- COMSOL's implementation does not support complex-valued system matrices.

In the **Linear System Solver Settings** dialog box you can control the automatic construction of the multigrid hierarchy with the **Maximum number of levels**, **Max DOFs at coarsest level**, and **Quality of multigrid hierarchy** edit fields. Coarse levels are added until the number of DOFs at the coarsest level is less than the max DOFs at coarsest level (default = 5000) or until it has reached the maximum number of levels, including the finest level (default = 6). In the **Quality of multigrid hierarchy** edit field specify an integer value between 1 and 10 (default = 3) to make a tradeoff between memory requirements and preconditioner quality. For instance, if the linear solver does not converge or if it uses too many iterations, try a higher value to increase the accuracy in each iteration, meaning fewer iterations. In contrast, if the algebraic multigrid algorithm runs into memory problems, try a lower value to use less memory. When using algebraic multigrid as a preconditioner, it is also possible to set the value for the quality of the multigrid hierarchy in the **Linear system solver** area on the **General** tab of **Solver Parameters** dialog box either numerically or by using the **Memory efficiency/ Preconditioner quality** slider.

The remaining settings for the algebraic multigrid solver/preconditioner and its smoothers and coarse solver are identical to those for the Geometric multigrid solver (see “The Geometric Multigrid Solver/Preconditioner” on page 557).

The SSOR, SOR, SORU, and Diagonal Scaling (Jacobi) Algorithms

These simple and memory-efficient solvers/preconditioners/smoothers are based on classical iteration methods for solving a linear system of the form $Ax = b$. Given a relaxation factor ω (usually between 0 and 2), a sweep of the Jacobi (diagonal scaling) method transforms an initial guess x_0 to an improved approximation $x_1 = x_0 + M^{-1}(b - Ax_0)$, where $M = D/\omega$, and D is the diagonal part of A.

The SOR (successive over-relaxation) method uses the same formula with $M = L + D/\omega$, where L is the strictly lower triangular part of A. When $\omega = 1$ (the default), the Gauss-Seidel method is obtained. In the SORU method, $M = U + D/\omega$, where U is the strictly upper triangular part of A. The SOR and SORU methods use a more accurate approximation of the matrix, which leads to fewer iterations but slightly more work per iteration than in the Jacobi method.

The SSOR (symmetric successive over-relaxation) method is one SOR sweep followed by a SORU sweep. The output x_1 for an input x_0 also comes from the above formula but with

$$M = \frac{\omega}{2-\omega}\left(L + \frac{D}{\omega}\right)D^{-1}\left(U + \frac{D}{\omega}\right).$$

When A is symmetric, the SSOR method has the advantage that M is symmetric. Symmetry of the preconditioner matrix is necessary when using the conjugate gradients iterative method. In such cases, the SSOR preconditioner is preferable to the SOR preconditioner.

By default a blocked version of the SOR algorithms is used. It is optimized for parallel computations. In this case M is constructed from a column permuted version of A.

When these algorithms run as linear system solvers, they perform sweeps until they have established convergence or they have reached the maximal number of iteration. You control this aspect with the parameters in the **Relative tolerance**, **Factor in error estimate**, and **Maximum number of iterations** edit fields in the same way as for the other iterative solvers (see for instance "Settings for the Multigrid Solver/Preconditioner" on page 564 of the *COMSOL Multiphysics User's Guide* and "Convergence Criteria" on page 552 of this *Reference Guide*).

When the algorithms run as preconditioners or smoothers, they perform a fixed number of sweeps as dictated by the value in the **Number of iterations** edit field in the **Linear System Solver Settings** dialog box (default = 2).

Specify ω in the **Relaxation factor (omega)** edit field (default = 1).

The SSOR Vector, SOR Vector, and SORU Vector Algorithms

These preconditioners/smoothers are intended for problems involving the $\nabla\times(a\nabla\times.)$ curl-curl operator and where you use *vector elements*. The vector elements are available primarily for electromagnetic-wave simulations in the RF Module. The algorithm is an implementation of the concepts in Ref. 9 and Ref. 4. The algorithm applies SOR iterations on the main linear equation $Ax = b$ but also makes SOR iterations on a projected linear equation $T^TATy = T^Tb$. Here the projection matrix, T, is the discrete gradient operator, which takes values of a scalar field in the mesh vertices and computes the vector-element representation of its gradient. Loosely speaking, the argument for using this projection is the following: For example, let the linear equation $Ax = b$ represent the discretization of a PDE problem originating from the vector Helmholtz equation

$$\nabla\times(a\nabla\times\mathbf{E}) + c\mathbf{E} = \mathbf{F}$$

for the unknown vector field $\mathbf{E}$, where a and c are scalars, and $\mathbf{F}$ is some right-hand side vector. Standard preconditioners/smoothers cannot smooth the error in the null space of the operator $\nabla\times(a\nabla\times.)$. This null space is the range of the gradient operator. This algorithm adds a correction $\mathbf{E}\to\mathbf{E}+\nabla\phi$ to the standard SOR smoothed solution (or residual), where it computes ϕ from SOR iterations on a projected problem. The projected problem is obtained by taking the divergence (or discretely $-T^T$) of the Helmholtz equation and plugging in the correction. You then obtain (for clarity, boundary constraints are disregarded)

$$-\nabla\cdot(c\nabla\phi) = -\nabla\cdot\mathbf{F},$$

which, if c is definite (strictly positive or strictly negative), is a standard elliptic type of equation for the scalar field ϕ.

When using this algorithm as a smoother for the multigrid solver/preconditioner, it is important—for the correct discrete properties of the projected problem—to generate nested meshes. Also note that it does assembly on all mesh levels (controlled by the multigrid **Assemble** check box). You can generate nested meshes through manual mesh refinements or do so automatically by going to the **Linear System Solver Settings** dialog box and selecting **Refine mesh** from the **Hierarchy generation method** list.

The projection matrix T is computed in such a way that non-vector shape functions are disregarded, and therefore you can use it in a multiphysics setting. It can also handle contributions from different geometries. Non-vector shape function variables are not affected by the correction from the projected system, and the effects on them are therefore the same as when you apply the standard SOR algorithm (see above).

The parameter in the **Number of iterations** edit field in the **Presmoother** (or **Postsmoother**) area controls the number of main iterations (default = 2). For each main iteration, the algorithm makes a number of SOR iterations for the projected equation system; set that number (default = 1) in the **Number of secondary iterations** edit field.

In more detail, to preserve symmetry as a preconditioner and also when used as symmetric pre- and postsmoother in a multigrid setting, the SOR iterations are done in the following order:

- In each main iteration, the SOR vector version of this algorithm makes one SOR iteration on the main system followed by a number of secondary SOR iterations on the projected system.

- In each main iteration, the SORU vector version first makes a number of secondary SORU iterations on the projected system followed by one SORU iteration on the main system.
- In each main iteration, the SSOR vector version makes one SOR iteration on the main system followed by a number of secondary SSOR iterations on the projected system and then one SORU iteration on the main system.

You specify the relaxation factor ω in the **Relaxation factor (omega)** edit field (default = 1). It applies to all the different types of SOR iterations in this algorithm.

The SSOR Gauge, SOR Gauge, and SORU Gauge Algorithms

These preconditioners/smoothers are primarily intended for 3D magnetostatic problems in the AC/DC Module discretized with vector elements. The smoothers are basically SOR smoothers with some added functionality.

Magnetostatic problems are often formulated in terms of a magnetic vector potential. The solution of problems formulated with such a potential is in general not unique. Infinitely many vector potentials result in the same magnetic field, which typically is the quantity of interest. A finite element discretization of such a problem results in a singular linear system of equations, $Ax = b$. Despite being singular, these systems can be solved using iterative solvers, provided that the right hand side of the discretized problem is the range of the matrix A. For discretized magnetostatic problems, the range of A consists of all divergence free vectors. Even if the right side of the mathematical problem is divergence free, the right side of the finite element discretization might not be numerically divergence free. To ensure that b is in the range of A, SOR gauge performs a divergence cleaning of the right side by using the matrices T and T^T; see "The SSOR Vector, SOR Vector, and SORU Vector Algorithms" on page 451. To this end, the system $T^T T\psi = -T^T b$ is first solved. Adding $T\psi$ to b will then make the numerical divergence of the right side small.

As in the case of SOR the blocked version is used by default. It performs better when running on a parallel machine.

In addition to the initial divergence cleaning, SOR gauge also performs a number of cleaning iterations in each linear solver iteration. You can control the number of such divergence cleaning iterations in the **Number of secondary iterations** edit field in the **Linear System Solver** area. The default number of secondary iterations is 1. In the **Variables** edit field you can specify which vector degrees of freedom to include in the divergence cleaning (this applies both to the initial and secondary cleaning iterations). By default, all vector degrees of freedom are included in the divergence cleaning.

The settings **Number of iterations** and **Relaxation factor (omega)** work in the same way as for the usual SOR smoothers; see "The SSOR, SOR, SORU, and Diagonal Scaling (Jacobi) Algorithms" on page 450.

The Vanka Algorithm

This preconditioner/smoother is intended for, but not restricted to, problems involving the Navier-Stokes equations. Formally it applies to saddle-point problems. A saddle-point problem is a problem where the (equilibrium) solution is neither a maximum nor a minimum. The corresponding linear system matrix is indefinite, and often it has zeros on the diagonal. This is the case for the Navier-Stokes equations but also for problems formulated with weak constraints.

The algorithm is a local smoother/preconditioner of Vanka type. It is based on the ideas in Ref. 5, Ref. 10, and Ref. 11. It is possible to describe it as a block SOR method, where the local coupling of the degrees of freedom (DOFs) determines the blocks. The important idea in this algorithm is to use the Lagrange multiplier variable (or set of variables) to form the blocks. For illustration purposes, consider the Navier-Stokes equations. For these equations the pressure variable plays the role of Lagrange multiplier. The linearized equations on discrete form has the following structure:

$$A\begin{bmatrix} U \\ P \end{bmatrix} = \begin{bmatrix} S & D^T \\ D & 0 \end{bmatrix}\begin{bmatrix} U \\ P \end{bmatrix} = \begin{bmatrix} F \\ G \end{bmatrix}$$

where U and P are the velocity and pressure degrees of freedom, respectively. The algorithm loops over the Lagrange multiplier variable DOFs, here the pressure DOFs P_j, and finds the direct connectivity to this DOF. To do so, the algorithm locates the nonzero entries in the matrix column corresponding to P_j. The row indices of the nonzero entries defines the DOFs U_k, and the software forms a local block matrix based on this connectivity:

$$A_j = \begin{bmatrix} S_j & D_j^T \\ D_j & 0 \end{bmatrix}$$

One *Vanka update* loops over all P_j and updates

$$\begin{bmatrix} U_j \\ P_j \end{bmatrix} \leftarrow \begin{bmatrix} U_j \\ P_j \end{bmatrix} + \omega A_j^{-1}\left(\begin{bmatrix} F \\ G \end{bmatrix} - A \begin{bmatrix} U \\ P \end{bmatrix} \right)_j$$

where the $(.)_j$ denotes the restriction of a vector to the rows corresponding to the block j. ω is a relaxation parameter. The algorithm does not form the inverses of the block matrices explicitly. Instead, it computes the Vanka update either with a LAPACK direct solver subroutine call or by a GMRES iterative method subroutine call. The GMRES method is the restarted GMRES without preconditioning. The algorithm relies on that it is possible to invert the submatrices A_j. If it is not possible, the algorithm gives an error message. Note that a zero on the diagonal of A or A_j is not necessarily a problem for this updating strategy.

Note: If you use the Vanka algorithm as preconditioner, or as smoother to a multigrid preconditioner when either of GMRES, Conjugate gradients, or BiCGStab is used as the linear system solver, you should use the **Direct** option in the **Solver** list in order to get a stationary preconditioner. The **GMRES** option can be useful if you use the FGMRES method as linear system solver since it can handle preconditioners that are not stationary. The **GMRES** option can also be useful if you use the Vanka algorithm as smoother to a multigrid solver because GMRES can be a bit faster than the direct solver.

In general, the Vanka update does not necessarily update all DOFs. This is, for example, the case for problems with weak constraints, where only a small subset of the problem's DOFs are directly coupled to the Lagrange multipliers for the constraints. Another example is the Navier-Stokes equations (or similar types of equations) coupled to other equations, but where the coupling is not directly through the pressure variable. This is, for example, the case with the k-ε turbulence model. The Vanka algorithm automatically detects DOFs that are not updated by the above Vanka updating procedure and performs, for each Vanka update, a number of SSOR sweeps for these DOFs. This part of the algorithm is denoted the *SSOR update***.** The SSOR update only works for a submatrix that has a nonzero diagonal. Just as the SOR and Jacobi preconditioner algorithms, this algorithm gives an error message if it finds zeros on the diagonal for the DOFs in the SSOR update.

A blocked version that works on a permuted version of the system matrix is used by default, as in the case of SOR. It is especially suited for parallel computations.

Control the number of Vanka updates and consecutive SSOR updates by the parameter in the **Number of iterations** edit field in the **Linear System Settings** dialog box. For the Vanka update, control the Lagrange variables used for the local block definitions by the **Variables** edit field, and control the type of solver used for the block inverse operation by the **Solver** list. If you choose GMRES, then you can control the convergence tolerance and the number of iterations before restart by the parameters in the **Tolerance** and **Number of iterations before restart** edit fields, respectively. Control the Vanka update relaxation parameter ω by the parameter in the **Relaxation factor** edit field. For the SSOR update, control the number of SSOR sweeps by the parameter in the **Number of secondary iterations** edit field and control the SSOR relaxation factor, used in these sweeps, with the parameter in the **Relaxation factor** edit field.

References

1. J.R. Gilbert and S. Toledo, "An Assessment of Incomplete-LU Preconditioners for Nonsymmetric Linear Systems," *Informatic*a, vol. 24, pp. 409–425, 2000.

2. Y. Saad, *ILUT: A dual threshold incomplete LU factorization*, Report umsi-92-38, Computer Science Department, University of Minnesota, available from http://www-users.cs.umn.edu/~saad.

3. W. Hackbusch, *Multi-grid Methods and Applications*, Springer-Verlag, Berlin, 1985.

4. R. Beck and R. Hiptmair, "Multilevel solution of the time-harmonic Maxwell's equations based on edge elements," *Int. J. Num. Meth. Engr.*, vol. 45, pp. 901–920, 1999.

5. S. Vanka, "Block-implicit multigrid calculation of two-dimensional recirculating flows," *Computer methods in Applied Mechanics and Engineering*, vol. 59, no. 1, pp. 29–48, 1986.

6. H.C. Elman and others, "A Multigrid method enhanced by Krylov subspace iteration for discrete Helmholtz equations," *SIAM J. Sci. Comp.*, vol. 23, pp. 1291–1315, 2001.

COMSOL's implementations of the algebraic multigrid solver and preconditioner are based on the following references:

7. K. Stüben, *Algebraic Multigrid (AMG): An introduction with Applications*, GMD Report 70, GMD, 1999.

8. C. Wagner, *Introduction to Algebraic Multigrid*, Course notes, University of Heidelberg, 1999.

9. R. Hiptmair, “Multigrid method for Maxwell’s equations,” *SIAM J. Numer. Anal.*, vol. 36, pp. 204–225, 1999.

10. V. John and G. Matthies, “Higher-order finite element discretization in a benchmark problem for incompressible flows,” *Int. J. Numer. Meth. Fluids*, vol. 37, pp. 885–903, 2001.

11. V. John, “Higher-order finite element methods and multigrid solvers in a benchmark problem for the 3D Navier-Stokes equations,” *Int. J. Numer. Meth. Fluids*, vol. 40, pp. 775–798, 2002.

Optimization Solver Properties

This section provides detailed explanations of the properties that control the behavior of SNOPT—the optimization solver that comes with the Optimization Lab, which is avaliable as an optional add-on to COMSOL Multiphysics.

When solving multiphysics optimization problems using the Optimization application mode, some of the properties listed in this section can be controlled through components in the COMSOL Multiphysics graphical user interface while others, the *advanced properties*, always take their default values. To find out which properties you can modify in the graphical user interface, see the section "Solver Settings" on page 321 in Chapter 11, "Optimization," of the *COMSOL Multiphysics Modeling Guide*.

Modifying the value of an advanced property requires calling the optimization solver from the MATLAB command line using the `femoptim` command. A list of all available optimization solver properties also appears in the "Command Reference" entry for `femoptim` on page 149 of this manual.

Note: In the following sections, ε represents the machine precision (available as `eps` in MATLAB) and is approximately equal to $2.2 \cdot 10^{-16}$.

Cendiff

Central difference interval
Type: numeric
Default: $\varepsilon^{1/3} \approx 6.0 \cdot 10^{-6}$

When some problem derivatives are unknown, the solver uses the central difference interval near an optimal solution to obtain more accurate (but more expensive) estimates of gradients. Twice as many function evaluations are required compared to forward differencing. If r is the central difference interval, the interval used for the jth variable is $h_j = r(1 + |x_j|)$. The resulting derivative estimates should be accurate to $O(r^2)$, unless the functions are badly scaled.

Checkfreq

Check frequency
Type: integer
Default = 60

Every ith iteration after the most recent basis factorization, the solver makes a numerical test to see if the current solution x satisfies the general linear constraints (including linearized nonlinear constraints, if any). The constraints are of the form $Ax - s = b$ where s is the set of slack variables. To perform the numerical test, the solver computes the residual vector $r = b - Ax + s$. If the largest component of r is judged to be too large, the current basis is refactorized and the basic variables are recomputed to satisfy the general constraints more accurately.

`checkfreq = 1` is useful for debugging purposes, but otherwise this option should not be needed.

Diffint

Difference interval
Type: numeric
Default: $\varepsilon^{1/2} \approx 1.5 \cdot 10^{-8}$

This property alters the interval that the solver uses to estimate gradients by forward differences in the following circumstances:

- In the initial ("cheap") phase of verifying the problem derivatives
- For verifying the problem derivatives
- For estimating missing derivatives

In all cases, the solver estimates a derivative with respect to x_j by perturbing that component of x to the value $x_j + h_1(1 + |x_j|)$, where h_1 is the difference interval, and then evaluating the goal function at the perturbed point.

The resulting gradient estimates should be accurate to $O(h_1)$ unless the functions are badly scaled. Judicious alteration of the difference interval can sometimes lead to greater accuracy.

Elasticw

Elastic weight
Type: numeric
Default: 1.0 *for linear and quadratic problems,* $1.0 \cdot 10^4$ *for nonlinear problems*

This property determines the initial weight associated with the elastic QP problem. For more details, see the SNOPT User's Guide. In general, in Elastic mode, if the original problem has a feasible solution and the elastic weight is sufficiently large, a feasible point is eventually obtained for the perturbed constraints and optimization can continue.

Expfreq

Expand frequency
Type: integer
Default: 10,000

This option is part of the internal procedure designed to make progress even on highly degenerate problems.

For linear models, the strategy is to force a positive step at every (minor) iteration at the expense of violating the bounds on the variables by a small amount. Suppose that the expand frequency is i and the feasibility tolerance (property `feastol`) is δ. Over a period of i iterations, the tolerance the solver actually uses increases from 0.5δ to δ (in steps of $0.5\delta/i$).

For nonlinear models, the same procedure is used for iterations in which there is only one superbasic variable. (Cycling can occur only when the current solution is at a vertex of the feasible region.) Thus, zero steps are allowed if there is more than one superbasic variable, but otherwise positive steps are enforced.

Increasing i helps reduce the number of slightly infeasible nonbasic variables (most of which are eliminated during a resetting procedure). However, it also diminishes the freedom to choose a large pivot element (see `pivtol` on page 586).

Facfreq

Factorization frequency
Type: integer
Default: 100 *for linear problems,* 50 *for quadratic or nonlinear problems*

At most k basis changes occur between factorizations of the basis matrix, where k is the factorization frequency.

With linear programs, the basis factors are usually updated every iteration. The default k is reasonable for typical problems. Higher values to $k = 100$ might be more efficient on problems that are extremely sparse and well scaled.

When the objective function is nonlinear or quadratic, fewer basis updates occur as an optimum is approached. The number of iterations between basis factorizations therefore increases. During these iterations a test is made regularly (according to the check frequency, `Checkfreq`) to ensure that the general constraints are satisfied. If necessary the basis is refactorized before the limit of k updates is reached.

Feastol

Feasibility tolerance
Type: numeric
Default: $1.0 \cdot 10^{-6}$

The solver tries to ensure that all bound and linear constraints are eventually satisfied to within the feasibility tolerance t. (Feasibility with respect to nonlinear constraints is instead judged by the major feasibility tolerance, `majfeastol`.)

If the bounds and linear constraints cannot be satisfied to within t, the problem is declared infeasible. Let sInf be the corresponding sum of infeasibilities. If sInf is quite small, it might be appropriate to raise t by a factor of 10 or 100. Otherwise you should suspect some error in the data.

Nonlinear functions are evaluated only at points that satisfy the bound and linear constraints. If there are regions where a function is undefined, every attempt should be made to eliminate these regions from the problem. For example, if $f(x) = \sqrt{x_1} + \log x_2$, it is essential to place lower bounds on both variables. If $t = 10^{-6}$, the bounds $x_1 \geq 10^{-5}$ and $x_2 \geq 10^{-4}$ might be appropriate. (The log singularity is more serious. In general, keep x as far away from singularities as possible.)

If `scaleopt` (see page 589) is `1`, feasibility is defined in terms of the scaled problem (because it is then more likely to be meaningful).

In practice, the solver uses t as a feasibility tolerance for satisfying the bound and linear constraints in each QP subproblem. If the sum of infeasibilities cannot be reduced to zero, the QP subproblem is declared infeasible. The solver is then in the Elastic mode thereafter (with only the linearized nonlinear constraints defined to be elastic).

Funcprec

Function precision
Type: numeric
Default: $\varepsilon^{0.8} \approx 3.8 \cdot 10^{-11}$

The relative function precision is intended to be a measure of the relative accuracy with which the nonlinear functions can be computed. For example, if $f(x)$ is computed as 1000.56789 for some relevant x and if the first 6 significant digits are known to be correct, the appropriate value for the function precision would be 10^{-6}. (Ideally the functions should have a magnitude of order 1. If all functions are substantially less than 1 in magnitude, the function precision should be the absolute precision. For example, if $f(x) = 1.23456789 \cdot 10^{-4}$ at some point and if the first 6 significant digits are known to be correct, the appropriate precision would be 10^{-10}.)

The default value is appropriate for simple analytic functions.

In some cases the function values are the result of extensive computations, possibly involving an iterative procedure that can provide rather few digits of precision at reasonable cost. Specifying an appropriate function precision might lead to savings by allowing the line search procedure to terminate when the difference between function values along the search direction becomes as small as the absolute error in the values.

Hessdim

Hessian dimension
Type: numeric
Default: $\min\{1000, n_1 + 1\}$, *where* n_1 *is the number of nonlinear variables.*

Let r be the value given by the `hessdim` property. This specifies that an r-by-r triangular matrix R is to be available for use by the Cholesky QP solver (to define the reduced Hessian according to $RTR = Z^T HZ$). See the SNOPT User's Guide for further details.

Hessfreq

Hessian frequency
Type: numeric
Default: 999,999

If the `hessmem` property is set to `'full'` and `hessfreq` BFGS updates have already been carried out, the Hessian approximation is reset to the identity matrix. (For certain problems occasional resets might improve convergence, but in general they should not be necessary.) `hessmem` set to `'full'` and `hessfreq` set to `20` have a similar effect to `hessmem` set to `'limited'` and `hessupd` set to `20` (except that the latter retains the current diagonal during resets).

Hessmem

Hessian memory
Type: string 'full' or 'limited'
Default: 'full' if the number of nonlinear variables is ≤ `75` *. When the QP problem solver is set to conjugate-gradient, the default is always 'limited'.*

This option selects the method for storing and updating the approximate Hessian. (The solver uses a quasi-Newton approximation to the Hessian of the Lagrangian. A BFGS update is applied after each major iteration.)

If Hessian `full` memory is specified, the approximate Hessian is treated as a dense matrix and the BFGS updates are applied explicitly. This option is most efficient when the number of nonlinear variables is not too large (say, less than 75). In this case, the storage requirement is fixed and you can expect 1-step Q-superlinear convergence to the solution.

Hessian `limited` memory should be used on problems where the number of nonlinear variables is very large. In this case a limited-memory procedure stores a fixed number of BFGS update vectors and a diagonal Hessian approximation.

Hessupd

Hessian updates
Type: integer
Default: `10`

If `hessmem` is set to `limited` memory and `hessupd` BFGS updates have already been carried out, all but the diagonal elements of the accumulated updates are discarded and the updating process starts again. Broadly speaking, the more updates stored, the better the quality of the approximate Hessian. However, the more vectors stored, the greater the cost of each QP iteration. The default value is likely to give a robust algorithm without significant expense, but faster convergence can sometimes be obtained with significantly fewer updates (for example, `hessupd = 5`).

Infbound

Infinite bound size
Type: positive numeric
Default: $1.0{\cdot}10^{20}$

Defines the “infinite” bound in the definition of the problem constraints. Any upper bound greater than or equal to this bound is regarded as plus infinity (and similarly for a lower bound less than or equal to `-infbound`).

Itlim

Iterations limit
Type: nonnegative integer
Default: 500

This is the number of minor iterations for the optimality phase of the QP subproblem. If `itlim` is exceeded, then all nonbasic QP variables that have not yet moved are frozen at their current values and the reduced QP is solved to optimality.

Note that more than `itlim` minor iterations might be necessary to solve the reduced QP to optimality. These extra iterations are necessary to ensure that the terminated point gives a suitable direction for the line search.

Note that `totitlim` (total iterations limit; see page 590) defines an independent absolute limit on the total number of minor iterations (summed over all QP subproblems).

Linesearch

Linesearch method
Type: string 'derivative' or 'nonderivative'
Default: 'derivative

At each major iteration a line search is used to improve the merit function. A `derivative` linesearch uses safeguarded cubic interpolation and requires both function and gradient values to compute estimates of the step. If some analytic derivatives are not provided or a nonderivative linesearch is specified, the solver employs a line search based upon safeguarded quadratic interpolation, which does not require gradient evaluations.

A `nonderivative` line search can be slightly less robust on difficult problems, and we recommend use of the default if the functions and derivatives can be computed at approximately the same cost. If the gradients are very expensive relative to the functions, a nonderivative line search might give a significant decrease in computation time.

Linestol

Linesearch tolerance
Type: numeric
Default: 0.9

This parameter controls the accuracy with which a step length is located along the direction of search during each iteration. At the start of each line search, the solver identifies a target directional derivative for the merit function. This parameter determines the accuracy to which this target value is approximated.

`linestol` must be a real value in the range 0 to 1. The default value of 0.9 requests just moderate accuracy in the line search. If the nonlinear functions are cheap to evaluate, a more accurate search might be appropriate; try a `linestol` value of 0.1, 0.01 or 0.001. The number of major iterations might decrease.

If the nonlinear functions are expensive to evaluate, a less accurate search might be appropriate. If all gradients are known, try `linestol` = 0.99. (The number of major iterations might increase, but the total number of function evaluations could decrease enough to compensate.)

If not all gradients are known, a moderately accurate search remains appropriate. Each search requires only one to five function values (typically), but many function calls are then needed to estimate missing gradients for the next iteration.

Majfeastol

Major feasibility tolerance
Type: numeric
Default: $1.0 \cdot 10^{-6}$

This parameter specifies how accurately the nonlinear constraints should be satisfied. The default value of $1.0 \cdot 10^{-6}$ is appropriate when the linear and nonlinear constraints contain data to roughly that accuracy.

Let rowerr be the maximum nonlinear constraint violation, normalized by the size of the solution. It is required to satisfy

$$\text{rowerr} = \max_i \text{viol}_i / (\|x\| + 1) \leq \text{majfeastol}$$

where viol_i is the violation of the ith nonlinear constraint. If some of the problem functions are known to be of low accuracy, a larger major feasibility tolerance might be appropriate.

Majitlim

Major iterations limit
Type: numeric
Default: max$(1000, m)$, *where* m *is the number of general constraints.*

This is the maximum number of major iterations allowed. It is intended to guard against an excessive number of linearizations of the constraints.

Majprintfreq

Print frequency
Type: integer
Default: 100

When printing to file, print frequency k indicates that one line of the iteration log will be printed every kth (minor) iteration.

Majprintlevel

Major print level
Type: integer
Default: 00001

Controls the amount of output to the print file for the major iterations.

In general, the value being specified may be thought of as a binary number of the form `JFDXbs`, where each letter stands for a digit that is either 0 or 1 as follows:

TABLE 5-1: MAJOR PRINT LEVEL OPTIONS, JFDXBS

LETTER	DESCRIPTION
s	A single line that gives a summary of each major iteration. (This entry in `JFDXbs` is not strictly binary because the summary line is printed whenever `JFDXbs` ≥ 1).
b	BASIS statistics, i.e., information relating to the basis matrix whenever it is refactorized. (This output is always provided if `JFDXbs` ≥ 10).
X	x_k, the nonlinear variables involved in the objective function or the constraints.
D	π_k, the dual variables for the nonlinear constraints.
F	$F(x_k)$, the values of the nonlinear constraint functions.
J	$J(x_k)$, the Jacobian matrix.

To obtain output of any of the items `JFDXbs`, set the corresponding digit to 1, otherwise to 0. `Majprintlevel` 1 gives normal output for linear and nonlinear problems, and `Majprintlevel` 11 gives addition details of the Jacobian factorization that commences each major iteration.

If `J` = 1, the Jacobian matrix will be output column-wise at the start of each major iteration. Column j will be preceded by the value of the corresponding variable x_j and a key to indicate whether the variable is basic, superbasic or nonbasic. (Hence if `J` = 1, there is no reason to specify `X` = 1 unless the objective contains more nonlinear variables than the Jacobian.) A typical line of output is:

```
3 1.250000D+01 BS 1 1.00000E+00 4 2.00000E+00
```

which would mean that x_3 is basic at value 12.5, and the third column of the Jacobian has elements of 1.0 and 2.0 in rows 1 and 4.

Major print level 0 suppresses most output, except for error messages.

Majsteplim

Major step limit
Type: numeric
Default: 2.0

This parameter limits the change in x during a line search. It applies to all nonlinear problems once the solver has found a "feasible solution" or "feasible subproblem."

- A line search determines a step α over the range $0 < \alpha \leq \beta$, where β is 1 if there are nonlinear constraints, or the step to the nearest upper or lower bound on x if all the constraints are linear. Normally, the first step length tried is $\alpha_1 = \min(1, \beta)$.
- In some cases, such as $f(x) = ae^{bx}$ or $f(x) = ax^b$, even a moderate change in the components of x can lead to floating-point overflow. The parameter `majsteplim` is therefore used to define a limit $\bar{\beta} = r(1 + \|x\|)/\|p\|$ (where p is the search direction and r the value of `majsteplim`), and the first evaluation of $f(x)$ is at the potentially smaller step length $\alpha_1 = \min(1, \bar{\beta}, \beta)$.
- Wherever possible, use upper and lower bounds on x to prevent evaluation of nonlinear functions at meaningless points. The major step limit provides an additional safeguard. The default value `majsteplim` = 2.0 should not affect progress on well-behaved problems, but setting it to 0.1 or 0.01 might be helpful when rapidly varying functions are present. A "good" starting point might be required. An important application is to the class of nonlinear least-squares problems.
- In cases where several local optima exist, specifying a small value for `majsteplim` could help locate an optimum near the starting point.

Maximize

Maximize objective
Type: string 'on' or 'off'
Default: 'off'

Setting this property to `on` causes the objective to be maximized instead of minimized.

Opttol

Optimality tolerance
Type: numeric
Default: $1.0 \cdot 10^{-6}$

This is the major optimality tolerance and specifies the final accuracy of the dual variables. On successful termination, the solver computes a solution (x, s, π) such that

$$\text{maxComp} = \max_j \ \text{Comp}_j / \|\pi\| \leq \text{opttol}$$

where Comp_j is an estimate of the complementarity slackness for variable j. The values Comp_j are computed from the final QP solution using the reduced gradients $d_j = g_j - \pi^T a_j$, as above. Hence you have

$$\mathrm{Comp}_j = \begin{cases} d_j \min\{x_j - l_j, 1\} & \text{if } d_j \geq 0 \\ -d_j \min\{u_j - x_j, 1\} & \text{if } d_j < 0 \end{cases}$$

See the SNOPT User's Guide for further details.

Newsuplim

New superbasics limit
Type: integer
Default: 99

This option causes early termination of the QP subproblems if the number of free variables has increased significantly since the first feasible point. If the number of new superbasics is greater than `newsuplim`, the nonbasic variables that have not yet moved are frozen and the resulting smaller QP is solved to optimality.

Parprice

Partial price
Type: integer
Default: 10 *for linear problems,* 1 *for quadratic or linear problems*

This parameter is recommended for large problems that have significantly more variables than constraints. It reduces the work required for each "pricing" operation (when a nonbasic variable is selected to become superbasic).

When the partial price is 1, all columns of the constraint matrix $(A - I)$ are searched. Otherwise, A and I are partitioned to give the partial price i roughly equal segments A_j, I_j (j = 1 to i). If the previous pricing search was successful on A_j, I_j, the next search begins on the segments A_{j+1}, I_{j+1}. (All subscripts here are modulo i.)

If a reduced gradient is found that is larger than some dynamic tolerance, the variable with the largest such reduced gradient (of appropriate sign) is selected to become superbasic. If nothing is found, the search continues on the next segments A_{j+2}, I_{j+2}, and so on.

Partial price t (or $t/2$ or $t/3$) might be appropriate for time-stage models having t time periods.

Pivtol

Pivot tolerance
Type: numeric
Default: $3.7 \cdot 10^{-11}$

During solution of QP subproblems, the solver uses the pivot tolerance to prevent columns entering the basis if they would cause the basis to become almost singular.

When x changes to $x + \alpha p$ for some search direction p, a "ratio test" is used to determine which component of x first reaches an upper or lower bound. The corresponding element of p is called the pivot element.

Elements of p are ignored (and therefore cannot be pivot elements) if they are smaller than the pivot tolerance.

It is common for two or more variables to reach a bound at essentially the same time. In such cases, the (minor) feasibility tolerance (say t) provides some freedom to maximize the pivot element and thereby improve numerical stability. Excessively small values of t should therefore not be specified.

To a lesser extent, the expand frequency (property `expfreq`) also provides some freedom to maximize the pivot element. Excessively large values of `expfreq` should therefore not be specified.

Print

Print information about the solver progress and solution to file
Type: string
Default: empty

When the print option is activated, the following information is output to the file during the solution process. All printed lines are less than 131 characters.

- An estimate of the working storage needed and the amount available.
- Some statistics about the problem being solved.
- The storage available for the LU factors of the basis matrix.
- A summary of the scaling procedure, if `scaleopt` > 0.

- Notes about the initial basis.
- The iteration log.
- Basis factorization statistics.
- The exit condition and some statistics about the solution obtained.
- The printed solution, if requested.

For a more detailed overview of the various sections of the print files, see the SNOPT User's Guide.

Printfreq

Print frequency
Type: integer
Default: 100

When printing to file, print frequency k indicates that one line of the iteration log will be printed every kth (minor) iteration.

Printlevel

Print level
Type: integer
Default: 1

Controls the amount of output to the print file for the (minor) iterations.

TABLE 5-2: PRINT LEVEL OPTIONS

VALUE	DESCRIPTION
0	No output except for error messages.
≥1	A single line of output each minor iteration (controlled by `Printfreq`)
≥10	Basis factorization statistics generated during the periodic refactorization of the basis (see `Facfreq`). For nonlinear problems, the statistics for the first factorization each major iteration are controlled by the `Majprintlevel`.

Proxmeth

Proximal point method
Type: 1 *or* 2
Default: 1

proxmeth set to 1 or 2 specifies minimization of $\|x - x_0\|_1$ or $\frac{1}{2}\|x - x_0\|_2^2$, respectively, when the starting point x_0 is changed to satisfy the linear constraints (where x_0 refers to nonlinear variables).

Qpsolver

QP problem solver
Type: string 'cholesky', 'cg', or 'qn'
Default: 'cholesky'

Specifies the active-set algorithm used to solve the QP problem, or in the nonlinear case, the QP subproblem.

'cholesky' holds the full Cholesky factor R of the reduced Hessian Z^THZ. As the QP iterations proceed, the dimension of R changes with the number of superbasic variables. If the number of superbasic variables increases beyond the value of reduced Hessian dimension (property Hessdim), the reduced Hessian cannot be stored and the solver switches to qpsolver = 'cg'.

The Cholesky solver is reactivated if the number of superbasics stabilizes at a value less than the reduced Hessian dimension.

'qn' solves the QP subproblem using a quasi-Newton method. In this case, R is the factor of a quasi-Newton approximate Hessian.

'cg' uses an active-set method similar to 'qn' but uses the conjugate-gradient method to solve all systems involving the reduced Hessian.

The Cholesky QP solver is the most robust but might require a significant amount of computation if the number of superbasics is large.

The quasi-Newton QP solver does not require the computation of the exact R at the start of each QP and might be appropriate when the number of superbasics is large but each QP subproblem requires relatively few minor iterations.

The conjugate-gradient QP solver is appropriate for problems with large numbers of degrees of freedom (many superbasic variables). The Hessian memory option 'hessmem' is defaulted to 'limited' when this solver is used.

Scaleopt

Scale option
Type: integers 0, 1, *or* 2
Default: 1

Three scale options are available:

TABLE 5-3: SCALE OPTIONS

SCALEOPT	DESCRIPTION
0	No scaling. This is recommended if it is known that x and the constraint matrix (and Jacobian) never have very large elements (say, larger than 1000).
1	Linear constraints and variables are scaled by an iterative procedure that attempts to make the matrix coefficients as close as possible to 1.0. This sometimes improves the performance of the solution procedures.
2	All constraints and variables are scaled by the iterative procedure. Also, an additional scaling is performed that takes into account columns of $(A - I)$ that are fixed or have positive lower bounds or negative upper bounds. If nonlinear constraints are present, the scales depend on the Jacobian at the first point that satisfies the linear constraints. Scale option 2 should therefore be used only if a good starting point is provided and the problem is not highly nonlinear.

Scaletol

Scale tolerance
Type: numeric
Default: 0.9

Scale tolerance affects how many passes might be needed through the constraint matrix. On each pass, the scaling procedure computes the ratio of the largest and smallest nonzero coefficients in each column:

$$\rho_j = (\max_i \, |a_{ij}| / \min_i \, |a_{ij}|) \qquad (a_{ij} \neq 0)$$

If $\max_j \rho_j$ is less than `scaletol` times its previous value, another scaling pass is performed to adjust the row and column scales. Raising the scale tolerance from 0.9 to 0.99 (for instance) usually increases the number of scaling passes through A. At most 10 passes are made.

Suplim

Superbasics limit
Type: integer
Default: $n_1 + 1$*, where* n_1 *is the number of nonlinear variables*

This parameter places a limit on the storage allocated for superbasic variables. Ideally, `suplim` should be set slightly larger than the "number of degrees of freedom" expected at an optimal solution.

For linear programs, an optimum is normally a basic solution with no degrees of freedom. (The number of variables lying strictly between their bounds is no more than m, the number of general constraints.) The default value of `suplim` is therefore 1.

The number of degrees of freedom is often called the "number of independent variables."

For quadratic problems, `suplim` normally need not be greater than the number of leading nonzero columns of H. For many problems, `suplim` might be considerably smaller than that, which saves storage if the number of leading nonzero columns is very large.

For nonlinear problems, `suplim` normally need not be greater than $n_1 + 1$, where n_1 is the number of nonlinear variables. For many problems it might be considerably smaller than n_1. This saves storage if n_1 is very large.

Totitlim

Total iterations limit
Type: numerical
Default: $\max\{10000, 20m\}$*, where* m *is the number of general constraints*

This is the maximum number of minor iterations allowed (that is, iterations of the simplex method or the QP algorithm), summed over all major iterations.

Verify

Verification level
Type: integers –1, 0, 1, 2, *or* 3
Default: 0

This option refers to finite-difference checks on the derivatives computed by user-provided routines. Derivatives are checked at the first point that satisfies all bounds and linear constraints.

TABLE 5-4: THE VERIFY OPTION

VERIFY	DESCRIPTION
-1	Derivative checking is disabled.
0	Only a "cheap" test is performed, requiring 2 calls to user functions.
1	Individual objective gradients are checked (with a more reliable test).
2	Individual columns of the problem Jacobian are checked.
3	Options 2 and 1 both occur (in that order).

Verify level 3 is intended mainly for use when developing a new function routine. Missing derivatives are not checked, so they result in no overhead.

Viollim

Violation limit
Type: numeric
Default: 10

This keyword defines an absolute limit on the magnitude of the maximum constraint violation after the line search. On completion of the line search, the new iterate x_{k+1} satisfies the condition

$$v_i x_{k+1} \le \tau \max\{1, v_i(x_0)\}$$

where x_0 is the point at which the nonlinear constraints are first evaluated, and $v_i(x)$ is the ith nonlinear constraint violation $v_i(x) = \max(0, l_i - f_i(x), f_i(x) - u_i)$, where l_i and u_i are the lower and upper bounds, respectively.

The effect of this violation limit is to restrict the iterates to lie in an expanded feasible region whose size depends on the magnitude of τ. This makes it possible to keep the iterates within a region where the objective is expected to be well defined and bounded below. If the objective is bounded below for all values of the variables, then τ can be any large positive value.

6

The COMSOL Multiphysics Files

This chapter describes the COMSOL Multiphysics files in binary format and text format.

Overview

A COMSOL Multiphysics file is used to store COMSOL Multiphysics data. The format is suitable for exchange of mesh or CAD data between COMSOL Multiphysics and other software systems. It is possible to save a COMSOL Multiphysics file in a text file format, using the extension `.mphtxt`, or a binary file format, using the extension `.mphbin`. The file formats contain the same data in the same order.

File Structure

The COMSOL Multiphysics file format has a global version number, so that it is possible to revise the whole structure. The first entry in each file is the file format, indicated by two integers. The first integer is the major file version and the second is referred to as the minor file version. For the current version, the first two entries in a file is `0 1`.

The following sections describe the file structure of the supported version.

FILE VERSION 0.1

After the file version, the file contains three groups of data:

- A number of *tags* stored as strings, which gives an identification for each *record* stored in the file.
- A number of *types*, which are strings that can be used in serializing the object. The tag should be used as an indication of the contents of the file, but can also be an empty string.
- The *records* containing the serialized data in the file.

Example When using `flsave` to save a COMSOL Multiphysics mesh, the tag equals the variable name (`m1`), the type is set to `obj` (but this is not used), and the record contains the serialization of the mesh object, including point coordinates and element data of the mesh. See "Examples" on page 629 for more examples of COMSOL Multiphysics text files.

```
# Created by COMSOL Multiphysics Fri Aug 26 14:19:54 2005

# Major & minor version
0 1
######### Tags
1 # number of tags
```

```
2 m1

######## Types
1 # number of types
3 obj

######## Records

# A planar face object

0 0 1
4 Mesh # class
...
```

Records

The record contains the serialization data in the file and additional information on how to process the serialized data. It also has a version number.

The record is a wrapper for *serializable types* stored in the file. The reason for having this wrapper is to be able to use a version number, so that the serialization can be revised in the future while maintaining backward compatibility.

The following sections describe the format of the supported version:

RECORD VERSION 0

This record is a wrapper for *serializable types* stored in the file. The following table contains the attributes of the records:

ENTITY/OBJECT	VARIABLE	DESCRIPTION
Integer		Version
Integer		Not used
Integer	type	Serialization type, 1 for Serializable
Serializable	obj	If type equals 1, this field follows

Serialization type 1 indicates that the following field is a subtype to `Serializable`. COMSOL Multiphysics uses type equal to 0 internally, but such files are only used for temporary purposes.

Terminology

The following data types are used in the serialization:

- *Boolean* refers to an 8-bit signed character which must be 0 or 1.
- *Character* refers to an 8-bit signed character.
- *Integer* refers to a 32-bit signed integer.
- *Double* refers to a 64-bit double.

Matrices are stored in row-major order. In this documentation we use brackets to indicate a matrix. Hence, `integer[3][4]` means that 12 integers representing a matrix are store in the file. The first three entries corresponds to the integers in the first row of the matrix, and so on.

Text File Format

COMSOL Multiphysics text file, using the file extension `.mphtxt`, are text files where values are stored as text separated by whitespace characters.

Lexical conventions:

- Strings are serialized as the length of the string followed by a space and then the characters of the string, for example, "`6 COMSOL`". This is the only place where whitespace matters.
- The software ignores everything following a `#` on a line except when reading a string. This makes it possible to store comments in the file.

Binary File Format

COMSOL Multiphysics binary file, using the extension `.mphbin`, are binary files with the following data representation:

- Integers and doubles are stored in little-endian byte order.
- Strings are stored as the length of the string (integer) followed by the characters of the string (integers).

SAVING AND LOADING

You can import COMSOL Multiphysics files into the COMSOL Multiphysics GUI or load them as variables into MATLAB.

To load and save COMSOL Multiphysics files in MATLAB, use the functions `flsave` and `flload`, respectively.

To export to a COMSOL Multiphysics file from the GUI, select **File>Export>Geometry to File**. There is also a corresponding import menu. Note that the multiphysics files do not describe a complete model, so it is not possible to open them from the standard **Open File** dialog box.

Serializable Types

Supported Versions 0

Subtype of `Serializable`

Fields The class is defined by the following fields:

ENTITY/OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer	n	Number of attribute fields
Serializable[n]		Each entity field is stored as a serializable

Description An `Attribute` is a general purpose class from which different subtypes can be derived. Each such subtype should then be serialized using the serialization of the `Attribute` class, which means that all that it should add to the serialization is the version number.

Attributes are used in COMSOL Multiphysics for internal purposes, and these attributes are not documented. However, because `Attribute` has a strict serialization structure, the serialization of these attributes is well documented.

Example This is a serialization of an attribute used internally in COMSOL Multiphysics. It is serialized as a vector of integers.

```
11 AssocAttrib # class
0 0 # version
1 # nof attribute fields
9 VectorInt # class
18 3 2 2 2 1 0 1 1 1 1 1 1 0 1 1 1 1 0
```

Supported Versions 0

Subtype of `BezierMfd`

Fields The class is defined by the following fields:

ENTITY/OBJECT	DESCRIPTION
integer	Version
BezierMfd	Parent class containing common data

Description A rational Bézier curve is a parameterized curve of the form

$$\mathbf{b}(t) = \frac{\sum_{i=0}^{p} \mathbf{b}_i w_i B_i^p(t)}{\sum_{i=0}^{p} w_i B_i^p(t)} \quad , 0 \le t \le 1$$

where the functions

$$B_i^p(t) = \binom{p}{i} t^i (1-t)^{p-i}$$

are the *Bernstein basis* functions of *degree* p, $\mathbf{b}_i = (x_1, \ldots, x_n)$ are the control vertices of the n-dimensional space, and w_i are the weights, which should always be positive real numbers to get a properly defined rational Bézier curve. A rational Bézier curve has a direction defined by the parameter t.

Example The following illustrates a linear Bézier curve.

```
11 BezierCurve # class
0 0 # version
2 # sdim
0 2 1 # transformation
1 0 # degrees
2 # number of control points
# control point coords and weights
0 0 1
1 1 1
```

See also `BSplineCurve`

Supported Versions 0

Subtype of `Manifold`

Fields The class is defined by the following fields:

ENTITY/OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer	d	Space dimension
Transform		Transformation class
integer	m	Degree in first parameter
integer	n	Degree in second parameter
integer	k	Number of control points
double[k][d+1]	P	Matrix of control points with the weights in the last column

Description The `BezierMfd` type is the abstract base class for the different type of Bézier surfaces and curves that are supported. These can all be represented in using the generalized equation

$$\mathbf{S}(s,t) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n}\mathbf{b}_{i,j}w_{i,j}B(s,t)}{\sum_{i=0}^{m}\sum_{j=0}^{n}w_{i,j}B(s,t)}$$

where B are functions as described in the respective entry, and $\mathbf{b}$ are the control point coordinates in `P` and w are the weights stored in the last column of `P`.

See also `BSplineMfd`

BezierSurf

Supported Versions 0

Subtype of `BezierMfd`

Fields The class is defined by the following fields:

ENTITY/ OBJECT	DESCRIPTION
integer	Version
BezierMfd	Parent class containing common data

Description A rectangular rational Bézier patch of degree p-by-q is described by

$$\mathbf{S}(s,t) = \frac{\sum_{i=0}^{p}\sum_{j=0}^{q}\mathbf{b}_{i,j}w_{i,j}B_i^p(s)B_j^q(t)}{\sum_{i=0}^{p}\sum_{j=0}^{q}w_{i,j}B_i^p(s)B_j^q(t)},\ 0 \le s,t \le 1\ ,$$

where B_i^p and B_j^q are the Bernstein basis functions of degree p and q, respectively, as described in the entry of `BezierCurve`. This surface description is called rectangular because the parameter domain is rectangular, that is, the two parameters s and t can vary freely in given intervals.

See also `BSplineSurf`, `BezierTri`

Supported Versions 0

Subtype of `BezierMfd`

Fields The class is defined by the following fields:

ENTITY/ OBJECT	DESCRIPTION
integer	Version
BezierMfd	Parent class containing common data

Description Another form of surface description is the triangular patch, also called a *Bézier triangle*. A triangular rational Bézier patch is defined as

$$\mathbf{S}(s,t) = \frac{\sum\limits_{i+j\le p} \mathbf{b}_{i,j} w_{i,j} B^p_{i,j}(s,t)}{\sum\limits_{i+j\le p} w_{i,j} B^p_{i,j}(s,t)}, 0 \le s, t \le 1$$

which differs from the Bézier curve description only by the use of *bivariate* Bernstein polynomials instead of *univariate*, for the curve case. The bivariate Bernstein polynomials of degree p are defined as

$$B^p_{i,j}(s,t) = \frac{p!}{i!j!(p-i-j)!} s^i t^j (1-s-t)^{p-i-j}, \quad i+j \le p$$

where the parameters s and t must fulfill the conditions

$$\begin{cases} 0 \le s, t \\ s+t \le 1 \end{cases}$$

which form a triangular domain in the parameter space, therefore the name of this surface description.

See also `BezierSurf`

Supported Versions 0

Subtype of `BSplineMfd`

Fields The class is defined by the following fields:

ENTITY/ OBJECT	DESCRIPTION
integer	Version
BSplineMfd	Parent class containing common data

Description The `BSplineCurve`, describes a general spline curve, using B-spline basis functions, as defined in `BSplineMfd`. Splines on this form are often referred to as B-splines.

A *p*th-degree spline curve is defined by

$$\mathbf{C}(u) = \frac{\sum_{i=0}^{n} N_i^p(u) w_i \mathbf{P}_i}{\sum_{i=0}^{p} N_i^p(u) w_i} \quad , a \le u \le b$$

where $\mathbf{P}_i$ are the control points., the wi are the weights and the N_i^p are the *p*th degree B-spline basis functions defined in the nonperiodic and nonuniform knot vector

$$U = \{a, \dots, a, u_{p+1}, \dots, u_{m-p-1}, b, \dots, b\}$$

For non-rational B-splines, all weights are equal to 1 and the curve can be expressed as

$$\mathbf{C}(u) = \sum_{i=0}^{n} N_i^p(u) w_i \mathbf{P}_i, \ a \le u \le b$$

See also `BezierCurve`

Supported Versions 0

Subtype of `Manifold`

Fields The class is defined by the following fields:

ENTITY/ OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer	d	Space dimension
Transform		Transformation class
integer		Dimension (1 if curve, 2 if surface)
integer	p, q	Degree in each dimension (1 or 2 integers)
boolean		If rational equal to1
integer		Number of knot vectors (1 for curves, 2 for surfaces)
integer	m1	Length of first knot vector
double[m1]	U	First knot vector
integer	m2	Length of second knot vector (not for curves)
double[m2]	V	Second knot vector (not for curves)
integer	n1	Number of control points in first parameter direction
integer	n2	Number of control points in second parameter dimension
integer	n3	Number of coordinates per control point
double [n1][n2][n3]	P	Matrix of coordinates where the last dimension is increased by 1 to store the weights if the manifold is rational

Description The `BSplineMfd` type is the abstract base type for `BSplineCurve` and `BSplineSurf`, which represent general spline curves and surfaces, respectively.

They are represented using *B-spline basis functions*. Let $U = \{u_0, \dots, u_m\}$ be a nondecreasing sequence of real numbers; U is called the *knot vector* and the elements u_i of U are called *knots*. The ith B-spline basis function of p-degree, $N_i^p(u)$, is defined as

$$N_i^0(u) = \begin{cases} 1 & u_i \le u < u_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$N_i^p(u) = \frac{u - u_i}{u_{i+p} - u_i} N_i^{p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1}^{p-1}(u)$$

A general B-spline can be described by

$$\mathbf{S}(u, v) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{i,j} \mathbf{b}_{i,j}}{\sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{i,j}}, a \le u \le b, c \le v \le d$$

where

$$U = \{a, \ldots, a, u_{p+1}, \ldots, u_{m-p-1}, b, \ldots, b\}$$

and

$$V = \{c, \ldots, c, v_{p+1}, \ldots, v_{m-p-1}, d, \ldots, d\}$$

are the two knot vectors stored in the entry, and $\mathbf{b}$ and w are the control points coordinates and weights stored in P.

For periodic splines, the first and last parameter values in the knot vectors are not duplicated.

Supported Versions 0

Subtype of `BSplineMfd`

Fields The class is defined by the following fields:

ENTITY/ OBJECT	DESCRIPTION
integer	Version
BSplineMfd	Parent class containing common data

Description The generalization of B-spline curves to surfaces is a tensor product surfaces given by

$$\mathbf{S}(u, v) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{i,j} \mathbf{P}_{i,j}}{\sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{i,j}}, a \le u, v \le b$$

See also `BezierSurf`

Ellipse

Supported Versions 0

Subtype of `Manifold`

Fields The class is defined by the following fields:

ENTITY/ OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer	d	Space dimension
Transform		Transformation class
double[d]	center	Center coordinate
boolean		Equal to 1 if clockwise rotation (only if d==2)
double[d]	normal	Normal vector coordinates,
double[d]	M	Major axis
double	rat	Ratio of minor axis length to major axis length
double	offset	Parameter at the end of major axis
boolean		Equal to 1 if ellipse is degenerated

Description This manifold defines an ellipse in the two or three dimensional space.

In 2D, an ellipse is defined by a center point `center`, a vector defining the major axis `M` of the ellipse (including the magnitude of the major axis), the radius ratio of the minor axis length to the major axis length `rat`, the direction of the ellipse, and the parameter offset at the major axis `offset`.

In 3D, an ellipse is defined by a center point `center`, a unit vector normal to the plane of the ellipse `normal`, a vector defining the major axis of the ellipse `M` (including the magnitude of the major axis), the radius ratio, and the parameter offset at the major axis `offset`. The direction of the ellipse is defined by the right hand rule using the normal vector.

An ellipse is a closed curve that has a period of 2π. It is parameterized as:

```
point = center + M cos(t - offset) + N sin(t - offset)
```

where `M` and `N` are the major and minor axes respectively.

Example

```
7 Ellipse # class
0 0 # version
2 # sdim
0 2 1 # transformation
0 0 # center
```

```
0 # reverse?
2 0 # major axis
0.5 # minor axis length / major axis length
0 # parameter value at end of major axis
0 # degenerated?
# Attributes
0 # nof attributes
```

Geom0

Supported Versions 1

Subtype of `Serializable`

Fields The class is defined by the following fields:

ENTITY/OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer	type	Geometry type
double		Relative geometry tolerance
integer	p	Number of points (0 or 1)
integer	na	Number of attributes
Attribute[na]		Vector of attributes

Description The type represent a 0D geometry class, as described in the entry `geom0, geom1, geom2, geom3` on page 226.

The type can be either 0 for solid or –1 for general object.

Example A solid 0D geometry object.

```
5 Geom0 # class
1 0 1e-010 1
0 # nof attributes
```

Supported Versions 1

Subtype of `Serializable`

Fields The class is defined by the following fields:

ENTITY/OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer	type	Geometry type
double		Geometry tolerance
integer	nv	Number of vertices
double[nv]	vtx	Vector of vertex coordinates
integer[nv][2]	ud	Matrix of integers giving subdomains on up and down side of each vertex

Description The type represent a 1D geometry class, as described in the entry `geom0, geom1, geom2, geom3` on page 226.

The type can be either 0, 1, or –1 for point, solid, or general objects.

Example A solid 1D object.

```
5 Geom1 # class
1 # version
1 # type
1e-010 # gtol
3 # number of vertices
# Vertex coordinates
0
1
3
# Vertex up/down
1 0
2 1
0 2
# Attributes
0 # nof attributes
```

Geom2

Supported Versions 1

Subtype of `Serializable`

Fields The class is defined by the following fields:

ENTITY/OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer		Type
double		Relative geometry tolerance
integer	nv	Number of vertices
integers/doubles [nv][4]	vertex	Matrix of vertex data
integer	ne	Number of edges
integers/doubles [ne][8]	edge	Matrix of edge data
integer	nc	Number of curves
Manifold[nc]	curve	An array of Manifold objects

Description The type represent a 2D geometry class, as described in the entry `geom0, geom1, geom2, geom3` on page 226.

The type can be either 0, 1, 2, or –1 for point, curve, solid or general objects.

Example

```
5 Geom2 # class
1 # version
1 # type
1e-010 # gtol
0.0001 # resTol
2 # number of vertices
# Vertices
# X Y sub tol
0 0 -1 NAN
1 2.2999999999999998 -1 NAN

1 # number of edges
# Edges
# vtx1 vtx2 s1 s2 up down mfd tol
1 2 0 1 0 0 1 NAN
1 # number of manifolds
11 BezierCurve # class
0 0 # version
2 # sdim
0 2 1 # transformation
```

```
1 0 # degrees
2 # number of control points
0 0 1
1 2.2999999999999998 1
0 # nof attributes
```

Supported Versions 1

Subtype of `Serializable`

Fields The class is defined by the following fields:

ENTITY/OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer	type	Type
double		Relative geometry tolerance
double		Relative resolution tolerance
integer	nv	Number of vertices
integers/doubles [nv][5]	vertex	Matrix of vertex data
integer	npv	Number of parameter vertices
integers/doubles [npv][6]	pvertex	Matrix of parameter vertex data
integer	ne	Number of edges
integers/doubles [ne][7]	edge	Matrix of edge data
integer	npe	Number of parameter edges
integers/doubles [nep][10]	pedge	Matrix of parameter edge data
integer	nf	Number of faces
integers/doubles [nf][4]	face	Matrix of face data
integer	nm	Number of 3D manifolds, curves and surfaces
Manifold[nmfd]	mfd	Vector of manifolds
integer	npc	Number of parameter curves
Manifold[npc]	pcurve	Vector of parameter curves

Description The type represent a 3D geometry class, as described in the entry `geom0, geom1, geom2, geom3` on page 226.

The type can be either 0, 1, 2, 3, or −1 for point, curve, shell, solid, or general objects.

Supported Versions 1

Subtype of `Manifold`

Fields The class is defined by the following fields:

ENTITY/ OBJECT	VARIABLE	DESCRIPTION
integer		Version
string		M-file name
integer		Boundary number in M-file
double		Start parameter value
double		End parameter value

Description This curve represent a trimmed part of a boundary described by a geometry M-file. The boundary index, and start and end parameters of the trimming parts are store din the curve entry. For details on Geometry M-files, see the entry `geomfile` on page 248.

Example A curve representation using the `cardg.m` Geometry M-file.

```
8 GeomFile # class
0 0 # version
2 # sdim
0 2 1 # transformation
5 cardg # filename
1 # boundary number
1.5707963267948966 3.1415926535897931 # parameter range
```

Manifold

Supported Versions	0
Subtype of	`Serializable`
Fields	This is an abstract class with no fields.
Description	A manifold is the common supertype for curve and surface types. It is used by the geometry types.
See also	`Geom0`, `Geom1`, `Geom2`, `Geom3`

Supported Versions 1

Subtype of `Serializable`

Fields The class is defined by the following fields:

ENTITY/OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer	d	Space dimension (if equal to 0 no more fields)
integer	np	Number of mesh points
integer		Lowest mesh point index
double[d][np]	p	Mesh points
integer	nt	Number of element types (fives the number of repeats of the following fields)
string		Element type
integer	nep	Number of nodes per element
integer	ne	Number of elements
integer[ne][nep]	elem	Matrix of point indices for each element.
integer	ner	Number of parameter values per element
integer	nr	Number of parameter sets
double[nr][ner]	par	Matrix of parameter values
integer	ndom	Number of domain values
integer[ndom]	dom	Vector of domain labels for each element.
integer	nud	Number of up/down boundary relations
integer[nud]	ud	Matrix of integers stating subdomain number on up and down side of the boundary

Description This type represent a mesh that can be used by COMSOL Multiphysics. The entries `p`, `elem`, `par`, `dom`, and `ud` are all described in the reference entry `femmesh` on page 136. However, the domain numbering for points, edges, and boundaries must start from 0 when defining a mesh through a COMSOL Multiphysics mesh file.

Example The following displays a mesh with triangular elements on a unit square. Neither point or edge elements are present.

```
4 Mesh # class
2 # sdim
5 # number of mesh points
0 # lowest mesh point index
# Mesh point coordinates on unit square
```

```
0 0
1 0
1 1
0 1
0.5 0.5
1 # number of element types
3 tri # type name
3 # number of nodes per element
4 # number of elements
# Elements, 4 triangular elements
0 1 4
3 0 4
2 3 4
1 2 4
6 # number of parameter values per element
0 # number of parameters
4 # number of domains
# Domains
1
1
2
2
0 # number of up/down pairs
```

Supported Versions 1

Subtype of `Manifold`

Fields The class is defined by the following fields:

ENTITY/OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer		Space dimension
Transform		Transformation
integer	np	Number of points
double[np][d]	p	Matrix of point coordinates
double[np]	par	Vector of point parameters
Manifold	intcurve	Interpolating curve

Description Mesh structures can also be used to define manifolds. Because meshes contain a number of nodes and, in the case of COMSOL Multiphysics, corresponding parameter values, a good geometric representation can be obtained using a suitable interpolation method for evaluating the values of the manifolds and its derivatives on parameter values that are inside the intervals between the given nodes. Mesh curves are handled by cubic spline interpolation.

The matrix `p` and the vector `par` corresponds to the structures corresponding structures in an edge mesh representation. For the MeshCurve, they serve as the interpolation data to obtain `intcurve`.

See also `BSplineCurve`

MeshSurf

Supported Versions 1

Subtype of `Manifold`

Fields The class is defined by the following fields:

ENTITY/OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer		Space dimension
Transform		Transformation
integer	nv	Number of vertices
double[nv][3]	p	Matrix of mesh vertex coordinates
double[nv][2]		Matrix of mesh vertex parameters
integer	nt	Number of triangles
integers[nv][3]	elem	Matrix of vertex indices for each element

Description Mesh structures for surface meshes can be used to make a geometric definition of unstructured data. Since a mesh type in COMSOL Multiphysics have coordinates as well as parameter values for each element, interpolation can be employed to create smooth surfaces.

A quadratic interpolation is used to define a parametric surface from a surface mesh.

The matrix `p` of coordinates and the triangles `elem` with indices into p are used as the interpolation data.

See also `Mesh`

Supported Versions 1

Subtype of `Manifold`

Fields The class is defined by the following fields:

ENTITY/ OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer	d	Space dimension
Transform		Transformation
double[d]	p	The point in the plane with parameter value (0,0)
double[d]	n	Normal vector
double[d]	b	Direction of first parameter axis

Description This manifold defines a plane in the three dimensional space. It is represented by a point, a unit vector normal to the plane, and the vector of the u derivative.

A plane is open in both parameter directions and neither periodic nor singular at any point. It is parameterized as:

```
pos = p + u*b + v*(n x b)
```

Example

```
5 Plane # class
0 0 # version
3 # sdim
0 3 1 # transformation
1.3 0.80000000000000004 1.6000000000000001 # root point
-6.1257422745431001e-017 0 1 # normal
-1 0 -6.1257422745431001e-017 # derivatives
0 # degenerated?
```

PolChain

Supported Versions 1

Subtype of `Manifold`

Fields The class is defined by the following fields:

ENTITY/ OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer	d	Space dimension
Transform		Transformation
integer	n1	First dimension of matrix of point coordinates (equal to d)
integer	n2	Second dimension of matrix point coordinates, number of points
doubles	pol	n1-by-n2 matrix of point coordinates

Description Polygon chains are piece wise linear curves, that are used as approximations of curves in the decomposition algorithm. They have an implicit parameter representation, that is, $[(i-1)(p-1), i(p-1)]$ on the ith interval in a polygon chain with p points. This is not a suitable representation because the derivatives may vary substantially along the curve.

See also `MeshCurve`

Supported Versions	0
Subtype of	
Fields	This is the abstract base type of all other types. It has no fields.
Description	Serializable is the abstract base type. It is used to indicate that a field can contain all supported types, as is the case for the Attribute type.
See also	`Attribute`

Straight

Supported Versions 1

Subtype of `Manifold`

Fields The class is defined by the following fields:

ENTITY/ OBJECT	VARIABLE	DESCRIPTION
integer		Version
integer	d	Space dimension
Transform		Transformation
double[d]	root	The point from which the ray starts
double[d]	dir	The direction in which the ray points
double	pscale	Parameter scale

Description This manifold defines an infinite straight line in the two or three dimensional space. It is represented by a point and a unit vector specifying the direction. A straight also has a scale factor for the parameterization, so the parameter values can be made invariant under transformation. If not specified the value of this parameter is set to 1.0.

A straight line is an open curve that is not periodic. It is parameterized as:

```
pos = root + u*pscale*dir
```

where `u` is the parameter.

Example

```
8 Straight # class
0 0 # version
3 # sdim
0 3 1 # transformation
1.3 0.8 0.0 # root point
-1 0 0 # direction
1 # parameter scale
```

See also `Plane`

Supported Versions 1

Subtype of `Serializable`

Fields The class is

ENTITY/ OBJECT	VARIABLE	DESCRIPTION
integer	d	Space dimension
boolean		1 if transformation is a unit transformation, 0 otherwise. If the value is 1, no more fields are present
double [d+1][d+1]	M	Values in transformation matrix
boolean		1 if determinant is positive, 0 otherwise
boolean		1 if matrix is isotropic, 0 otherwise

Description The transformation class is defined by the transformation matrix, that operates as a pre-multiplier on column vectors containing homogeneous coordinates thus

$$\begin{bmatrix} x' & y' & z' & s' \end{bmatrix} = M \cdot \begin{bmatrix} x & y & z & s \end{bmatrix}'$$

where the conventional 3D coordinates are

$$\begin{bmatrix} \frac{x}{s} & \frac{y}{s} & \frac{z}{s} \end{bmatrix}$$

The matrix thus consists of

$$\begin{bmatrix} & T_x \\ \mathbf{R} & T_y \\ & T_z \\ 0\ 0\ 0 & S \end{bmatrix}$$

where $\mathbf{R}$ is a nonsingular transformation matrix, containing the rotation, reflection, non-uniform scaling, and shearing components; $\mathbf{T}$ is a translation vector; and S is a global scaling factor greater than zero.

VectorDoble

Supported Versions

Subtype of `Serializable`

Fields The class is defined by the following fields:

ENTITY/ OBJECT	VARIABLE	DESCRIPTION
integer	n	Number of elements
double[n]	d	Elements

Description This is just a wrapper for a vector of doubles, that can be used to store fields in the `Attribute` class.

See also `Attribute`

Supported Versions

Subtype of `Serializable`

Fields The class is defined by the following fields:

ENTITY/ OBJECT	VARIABLE	DESCRIPTION
integer	n	Number of elements
integer[n]	d	Elements

Description This is just a wrapper for a vector of integers, that can be used to store fields in the `Attribute` class.

See also `Attribute`

VectorString

Supported Versions

Subtype of `Serializable`

Fields The class is defined by the following fields:

ENTITY/ OBJECT	VARIABLE	DESCRIPTION
integer	n	Number of elements
string[n]	d	Elements

Description This is just a wrapper for a vector of strings, that can be used to store fields in the `Attribute` class.

See also `Attribute`

Examples

To illustrate the use of the serialization format, some text files are listed in this session.

A Mesh with Mixed Element Types

A file containing a 3D mesh with 2nd order tetrahedral, prism and block elements. Some rows in the file are removed and replaced by an ellipsis(`...`).

```
# Created by COMSOL Multiphysics Fri Aug 26 12:43:42 2005


# Major & minor version
0 1
1 # number of tags
# Tags
7 fem35.0
1 # number of types
# Types
3 obj

# A mesh object

0 0 1
4 Mesh # class
3 # sdim
1503 # number of mesh points
0 # lowest mesh point index

# Mesh point coordinates
0 0 0
2.5 0 0
5 0 0

...
12.5 28.333330000000004 15
12.5 30 13.125
12.5 28.333330000000004 13.125

7 # number of element types

# Type #0
4 tet2 # type name

10 # number of nodes per element
318 # number of elements
# Elements
926 18 13 17 971 958 61 967 66 60
11 345 918 342 950 951 1137 949 373 1129
924 164 345 5 1026 1138 384 938 385 378
```

```
20 339 15 16 352 68 356 69 960 64
...
287 919 930 285 1100 1102 1152 317 1096 1098
3 227 4 8 936 28 243 35 945 36

30 # number of parameter values per element
0 # number of parameters
# Parameters
318 # number of domains
# Domains
1
...
1
1

0 # number of up/down pairs
# Up/down

# Type #1
6 prism2 # type name

18 # number of nodes per element
96 # number of elements
# Elements
85 174 90 476 474 557 221 118 237 499 1171 494 1170 1172 566 496 563 597
174 225 90 474 588 557 238 237 244 494 1173 579 1172 1174 566 598 597 604
174 175 225 474 472 588 222 238 239 494 1175 489 1173 1176 579 491 598 599
...
654 528 530 404 409 408 693 692 541 703 1333 538 1332 1334 543 460 459 465
654 504 528 404 405 409 687 693 694 703 1324 520 1333 1335 538 453 460 461
504 506 528 405 410 409 523 694 536 520 1336 525 1335 1337 538 462 461 466

54 # number of parameter values per element
0 # number of parameters
# Parameters

96 # number of domains
# Domains
2
2
2
...
2
2

0 # number of up/down pairs
# Up/down

# Type #2
4 hex2 # type name

27 # number of nodes per element
36 # number of elements
# Elements
```

```
410 506 409 528 762 760 831 859 525 466 1337 536 538 780 1339 775 1338 1343
1340 840 1341 853 777 837 1342 865 867
506 507 528 529 760 758 859 860 526 536 1281 537 539 775 1344 770 1340 1348
1345 853 1346 854 772 865 1347 866 868
...
893 785 809 787 718 719 722 723 908 910 1446 804 815 916 1487 801 1495 1502
1499 817 1500 806 747 750 1501 751 754

81 # number of parameter values per element
0 # number of parameters
# Parameters

36 # number of domains
# Domains
3
3
...
3
3

0 # number of up/down pairs
# Up/down

# Type #3
3 vtx # type name

1 # number of nodes per element
16 # number of elements
# Elements
0
4
20
...
723

0 # number of parameter values per element
0 # number of parameters
# Parameters

16 # number of domains
# Domains
0
1
...
14
15

0 # number of up/down pairs
# Up/down

# Type #4

4 edg2 # type name
```

```
3 # number of nodes per element
102 # number of elements
# Elements
4 9 37
9 14 50
14 19 63
...
175 174 222
174 85 221

3 # number of parameter values per element
102 # number of parameters
# Parameters
0 5 2.5
5 10 7.5
10 15 12.5
...
7.5 11.25 9.375
11.25 15 13.125

102 # number of domains
# Domains
0
0
...
27
27
27

0 # number of up/down pairs
# Up/down

# Type #5

4 tri2 # type name


6 # number of nodes per element
224 # number of elements
# Elements
18 17 13 66 61 60
164 5 345 385 384 378
20 339 15 352 68 356
...
404 409 405 460 453 461
405 409 410 461 462 466

12 # number of parameter values per element
0 # number of parameters
# Parameters

224 # number of domains
# Domains
12
7
```

```
7
...
5
5
5

224 # number of up/down pairs
# Up/down
1 0
1 0
1 0
1 0
...
2 0
2 0

# Type #6

5 quad2 # type name


9 # number of nodes per element
102 # number of elements
# Elements
85 90 476 557 118 499 1170 566 563
85 476 174 474 499 221 1171 496 494
...
809 787 722 723 815 817 1500 806 754
718 722 719 723 750 747 1501 754 751

18 # number of parameter values per element
0 # number of parameters
# Parameters

102 # number of domains
# Domains
1
4
4
...
8
2

102 # number of up/down pairs
# Up/down
2 0
2 0
...
3 0
3 0
```

A Planar Face

The following listing is a complete representation of a planar 3D face.

```
# Created by COMSOL Multiphysics Fri Aug 26 14:19:54 2005

# Major & minor version
0 1
######### Tags
1 # number of tags
# Tags
2 b1

######## Types
1 # number of types
# Types
3 obj

######## Records

# A planar face object

0 0 1
5 Geom3 # class
1 # version
2 # type
1e-010 # gtol
0.0001 # resTol
4 # number of vertices
# Vertices
# X Y Z sub tol
0 0 0 -1 1e-010
1 0 6.1257422745431001e-017 -1 1e-010
0 1 0 -1 1e-010
1 1 6.1257422745431001e-017 -1 1e-010
4 # number of parameter vertices
# Parameter vertices
# vtx s t fac mfd tol
1 0 0 -1 1 NAN
2 1 0 -1 1 NAN
3 0 1 -1 1 NAN
4 1 1 -1 1 NAN

4 # number of edges
# Edges
# vtx1 vtx2 s1 s2 sub mfd tol
1 2 0 1 -1 2 NAN
2 4 0 1 -1 3 NAN
4 3 0 1 -1 4 NAN
3 1 0 1 -1 5 NAN
4 # number of parameter edges
# Parameter edges
# edg v1 v2 s1 s2 up down mfdfac mfd tol
1 1 2 0 1 1 0 1 1 NAN
2 2 4 0 1 1 0 2 1 NAN
3 4 3 0 1 1 0 3 1 NAN
4 3 1 0 1 1 0 4 1 NAN

1 # number of faces
```

```
# Faces
# up down mfd tol
0 0 1 NAN

5 # number of 3D manifolds
# Manifold #0
5 Plane # class
0 0 # version
3 # sdim
0 3 1 # transformation
0 0 0 # root point
-6.1257422745431001e-017 0 1 # normal
1 0 6.1257422745431001e-017 # derivatives
0 # degenerated?

# Manifold #1
8 Straight # class
0 0 # version
3 # sdim
0 3 1 # transformation
0 0 0 # root point
1 0 6.1257422745431001e-017 # direction
1 # parameter scale

# Manifold #2
8 Straight # class
0 0 # version
3 # sdim
0 3 1 # transformation
1 0 6.1257422745431001e-017 # root point
0 1 0 # direction
1 # parameter scale

# Manifold #3
8 Straight # class
0 0 # version
3 # sdim
0 3 1 # transformation
1 1 6.1257422745431001e-017 # root point
-1 0 -6.1257422745431001e-017 # direction
1 # parameter scale

# Manifold #4
8 Straight # class
0 0 # version
3 # sdim
0 3 1 # transformation
0 1 0 # root point
0 -1 0 # direction
1 # parameter scale

4 # number of parameter curves
# Paramerer curve #0
8 PolChain # class
0 0 # version
```

```
2 # sdim
0 2 1 # transformation
2 2 0 0 1 0 # chain


# Paramerer curve #1
8 PolChain # class
0 0 # version
2 # sdim
0 2 1 # transformation
2 2 1 0 1 1 # chain


# Paramerer curve #2
8 PolChain # class
0 0 # version
2 # sdim
0 2 1 # transformation
2 2 1 1 0 1 # chain


# Paramerer curve #3
8 PolChain # class
0 0 # version
2 # sdim
0 2 1 # transformation
2 2 0 1 0 0 # chain

# Attributes
0 # nof attributes
```

I N D E X