

amatos

*A*daptive *M*esh Generator for *A*tmosphere
and *O*cean *S*imulation

API Documentation Version 2.0.0
Jörn Behrens, 4/2003

Abstract

This document describes programming interface to **amatos**, the **A**daptive **M**esh Generator for **A**tmospheric and **O**ceanographic **S**imulations. This document does not describe the techniques behind **amatos**, but is simply meant to document the features and methods **amatos** provides to the user. This is a preliminary documentation, and will probably always remain so.

Title: **amatos** Documentation
Description: Documentation of the
application programming interface (API)
for the adaptive mesh generator **amatos**
Version: 2.0.0 (pdfL^AT_EX)
Date: July 13, 2007
Author: Jörn Behrens
E-Mail: behrens@ma.tum.de
Address: Technische Universität München
Center for Mathematical Sciences (M3)
BoltzmannstraSSe 3
85747 Garching, Germany

Partly supported by the German
Climate Research Program



Contents

1	Introduction	4
2	How to use amatos	5
3	FEM support	6
3.1	Examples for the signatures	6
3.2	Registering variables	7
4	Conventions	7
5	Variables and Constants in the GRID Application Programming Interface	8
5.1	Mesh-Related Constants	8
5.2	Physics-Related Constants	11
6	Routines in the GRID Application Programming Interface	13
6.1	Routines for mesh creation and termination	13
6.2	Routines for saving and restoring the mesh	14
6.3	Routines for data retrieval	15
6.4	Routines supporting FEM variables	19
6.5	Routines for numerical calculation	20
6.6	Routines for controlling the mesh	24
6.7	Routines for the dual mesh	26
6.8	Auxiliary Routines	26
7	Initial Triangulation	27
8	Installation and Testing	29
8.1	Directory Structure	29
8.2	Building Library and Test Driver	29
8.3	Running the Test Driver	31
8.4	Input Parameters for the Test Driver	31
A	Copyright	32
B	License	32
C	Warranty	32

1 Introduction

Mesh generation is a complicated and often very problem specific task. Therefore, we do not pretend to have found the ultimate grid generation tool so far. We developed an adaptive grid generator with specific applications in mind. Atmosphere and ocean circulation with semi-Lagrangian advection schemes is the generic field of application for **amatos**.

The philosophy of **amatos** is to hide away all nontrivial tasks concerning mesh generation and adaptation from the application programmer. The complete mesh generation process can be controlled by approx. 20 Fortran 90 subroutines (in object oriented wording: methods). The application programming interface (GRID API) provides additional variables, data structures and constants to be used by the programmer.

amatos is implemented in Fortran 90 and the GRID API is a Fortran 90 Module. This choice has been motivated by the field of application, **amatos** is meant for. Most theoretical oceanographers and meteorologists know Fortran, whereas C or C++ cannot be found very often. However, this choice is not a restriction for the programmer, as most programming environments allow the call of Fortran functions from C or C++.

amatos has been implemented in three different flavors. There is a spherical version (sometimes referred to as **samatos**) and a parallelized version (not yet supported). There are slight differences in the interface definitions of these three packages. Features corresponding to a certain package are indicated in the text.

Acknowledgements

I would like to thank Natalja Rakowski for extensive code optimizations, the introduction of a short triangulation file format, as well as reformulating the space-filling curve ordering which is first documented and widely used in Version 2.0 of the package, and many more valuable improvements. Matthias Lauter provided several bug fixes and the routine **grid_coordgradient**. Thomas Heinze provided some methods for the spherical version and Armin Iske gave the thin plate spline radial basis interpolation routines. Klaus Dethloff and Annette Rinke gave the inspiration to write such a code. Wolfgang Hiller supported the early stages of development.

Part of this code was developed while being supported by the Bundesministerium fur Bildung, Wissenschaft, Forschung und Technologie (BMBF) under grant no. 07/VKV01/1. Other parts have been developed within the framework of the Alfred-Wegener-Institute (AWI) “Programm zur Forderung besonderer Forschungsthemen” under the title *Anwendungen der Multiskalenmodellierung mit adaptiven Finite-Elemente-Methoden*. **amatos** is now maintained with support of BMBF under grant no. 01 LD 0037 within the DEKLIM research program.

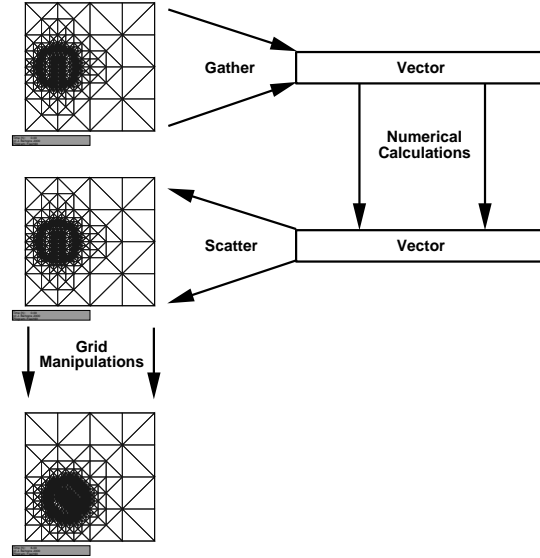


Figure 1: Two phases of calculation with **amatos**. Numerical calculations in vectors (after gathering data from mesh), and mesh manipulations (after scattering new values to mesh).

2 How to use amatos

amatos is a mesh generator for adaptive algorithms. There are two philosophies which seem to be the key to the understanding of **amatos**:

1. Think of the adaptive algorithm as a two phase procedure: In the first phase, the mesh is generated/adapted. Each mesh item keeps associated data. In the second phase, numerical calculations are performed. To achieve this, first gather all required data from mesh items into vectors, perform the calculations on vectors (utilizing consecutive storage positions for efficient pipelined or vectorized execution), and finally scatter the results back to mesh item storage positions. This is illustrated in Figure 1.
2. Think of the program as a data-flow, with methods acting and manipulating the data. A data structure (called **grid_handle**) represents a specific instance of the mesh. Methods (routines in the GRID API) act on the instance, manipulating it. Different methods can be applied to the mesh more or less independently.

The GRID API provides routines that implement methods in both of the above circumstances. Gathering (**grid_getinfo**) and scattering (**grid_putinfo**) accept the mesh handle without manipulating the mesh topology. Other methods, like **grid_adapt** alter the topology.

When performing numerical calculations on data gathered from the mesh into vectors, there **MUST** NOT be any change to the mesh topology, before scattering the results back to the mesh.

A new feature in Version 2.0 of **amatos** is the support of diverse kinds of finite elements. In order to use user defined finite elements, the element's characteristics have to be defined and implemented. This task cannot be performed at runtime but has to be concluded before compiling the package. There are two finite elements predefined in **amatos**, namely a linear and a quadratic Lagrange element.

Once the library has been compiled and an application uses **amatos**, each variable in the application has to be registered to a specific finite element type. A detailed documentation on finite element support can be found in section 3.

3 FEM support

From version 2.0 **amatos** comes with a flexible support for finite elements. Two types are predefined, a linear (unknowns defined in vertices) and a quadratic (unknowns defined in vertices and edge centers) Lagrange element.

In order to provide a flexible interface to all kinds of finite elements, **amatos** uses a signature data structure. The signature of a specific finite element contains the essential information of that type of element:

- name, order, and total number of unknowns,
- number of data items on the element's vertices,
- number and position of data on element's edges,
- number and position of data within the element.

The exact definition of this data structure is provided in the next section. For defining a new FEM type, the following steps have to be taken:

1. Edit module **FEM_signature** and define new FEM type in subroutine **grid_signatureinit** using the definitions of the default types as templates;
2. Edit module **FEM_signature** and add a corresponding basis function calculation to subroutine **grid_fembasis**;
3. recompile **amatos** and install it in your favorite directory;
4. relink the application that uses the new FEM type with **amatos**.

This procedure involves alterations of **amatos**' code and needs to become a little involved with the structure of the sources. However, it is the author's believe that code alterations of this kind are not necessary very often. In any case, if users implement new element types, please send an e-mail with the corresponding code fragments to Jörn Behrens (behrens@ma.tum.de), I will try to include them in future versions on **amatos**.

3.1 Examples for the signatures

1. For a linear element with values in each vertex, the signature is zero everywhere, except for
 - **i_order**= 1.
 - **i_unknowns**= 3.
 - **i_npoints**= 1.
2. For a quadratic element with values in each vertex and in each edge center, the signature has following values:
 - **i_order**= 2.
 - **i_unknowns**= 6.
 - **i_npoints**= 1.

- `i_gpoints= 1`.
- `r_gweights= (1/2, 1/2)`.

This means, each nodal value is situated at position of the node (coordinates are known). Each edge value can be calculated by the following formula:

$$(x, y) = \text{p_edge\%i_node}(1) \cdot \text{r_gweights}(1) + \text{p_edge\%i_node}(2) \cdot \text{r_gweights}(2)$$

3. For a cubic element with one value in each vertex, two values in each edge, and one value in the element center, the signature has following values:

- `i_order= 3`.
- `i_unknowns= 10`.
- `i_npoints= 1`.
- `i_gpoints= 2`.
- `i_epoints= 1`.
- `r_gweights= [(1/3, 2/3), (2/3, 1/3)]`.
- `r_ewights= (1/3, 1/3, 1/3)`.

3.2 Registering variables

In order to use different types of finite elements in an application, each variable that is used, has to be registered with a specific FEM type. This has to be done before any mesh items are created. The best point for registering variables is right after the call to `grid_initialize`.

Once a variable has been registered, the appropriate memory will be allocated at each unknown position corresponding to that element type. The type and order of a finite element can be obtained by a call to `grid_femtypequery`. The FEM type identifier corresponding to a registered variable is returned by `grid_femvarquery`. For a detailed description of these functions, see section 6.

4 Conventions

There are not many conventions related to **amatos**. The few conventions are listed here. Variables are named with a preceded character indicating the data type:

- `c_` is a variable of type **CHARACTER**.
- `i_` is a variable of type **INTEGER**.
- `l_` is a variable of type **LOGICAL**.
- `p_` is a variable of user declared type.
- `r_` is a variable of type **REAL**.

In the following sections, when giving the syntax, we denote by the attribute (*opt*) an optional argument which can be omitted.

5 Variables and Constants in the GRID Application Programming Interface

5.1 Mesh-Related Constants

GRID_timesteps (integer):

Number of timesteps the grid can manage (allowed: 1,2,3). This value has to be predefined at compile time (see file `FEM_param.f`, variable `DEF_timesteps`).

GRID_dimension (integer):

Number of spatial dimensions the grid elements can handle (allowed 2,3; predefined at compile time).

GRID_dimspherical (integer):

Number of spatial dimensions in spherical geometry (allowed 2; predefined at compile time, **only available in samatos**).

GRID_elementnodes (integer):

Number of nodes per element (at current 3).

GRID_elementedges (integer):

Number of edges per element (at current 3).

GRID_elementchildren (integer):

Number of children of a refined element (2).

GRID_edgenodes (integer):

Number of nodes per edge (2).

GRID_edgeelements (integer):

Number of adjacent elements per edge (2).

GRID_edgechildren (integer):

Number of children of a refined edge (2).

GRID_patchelements (integer):

Number of elements in a node's patch (at current 16, but this number depends on the initial triangulation. Can be predefined at compile time: Variable `DEF_ndpatch` in `FEM_param.f`).

GRID_elementvalues (integer):

Length of array storing the element stiffness matrix (at current 9, depending on element order. Can be predefined at compile time: Variable `DEF_evalsize` in `FEM_param.f`).

GRID_nodevalues (integer):

Length of array storing nodal values (at current 5, see below. Can be predefined at compile time: Variable `DEF_nvalsize` in `FEM_param.f`).

GRID_pleaserefine (integer):

Predefined value for marking elements for refinement.

GRID_pleasecoarse (integer):

Predefined value for marking elements for coarsening.

GRID_loworder (integer):
Predefined value for routine grid_coordvalue. This is for low order shape conserving interpolation.
GRID_highorder (integer):
Predefined value for routine grid_coordvalue. This is for high order interpolation.
GRID_thinplate (integer):
Predefined value for routine grid_coordvalue. This is for thin plate spline (radial basis) interpolation.
GRID_ucomp (integer):
Predefined value for retrieving nodal values. This is for the u-component of wind. <i>N.B. from version 2.0 this is for backward compatibility.</i>
GRID_vcomp (integer):
Predefined value for retrieving nodal values. This is for the v-component of wind. <i>N.B. from version 2.0 this is for backward compatibility.</i>
GRID_wcomp (integer):
Predefined value for retrieving nodal values. This is for the w-component of wind (only available in samatos). <i>N.B. from version 2.0 this is for backward compatibility.</i>
GRID_phi (integer):
Predefined value for retrieving nodal values. This is for the geopotential height Φ . <i>N.B. from version 2.0 this is for backward compatibility.</i>
GRID_zeta (integer):
Predefined value for retrieving nodal values. This is for the vorticity ζ . <i>N.B. from version 2.0 this is for backward compatibility.</i>
GRID_tracer (integer):
Predefined value for retrieving nodal values. This is for the tracer. <i>N.B. from version 2.0 this is for backward compatibility.</i>
GRID_boundary (integer):
Predefined value for retrieving the domain boundary polygonal line by grid_getpolyline.
GRID_partition (integer):
Predefined value for retrieving partition boundary polygonal line by grid_getpolyline (for the parallel version only).
GRID_boundedges (integer):
Predefined value for retrieving the domain boundary polygonal line by grid_getpolyline. The real boundary in terms of the boundary edges of the grid are returned.
i_time (integer):
Tag for the current timestep.
i_timeplus (integer):
Tag for the future timestep.
i_timeminus (integer):
Tag for the past timestep (equals i_time in the 2-time case).

grid_handle (Type definition):

This is **amatos**'s main data structure, containing all relevant information on the mesh. It contains the following components:

- **i_timestep**: value for current time tag **integer**
- **i_total**: total number of elements **integer**
- **i_number**: number of elements for this time **integer**
- **i_enumfine**: number of elements on finest level **integer**
- **i_gtotal**: total number of edges **integer**
- **i_gnumber**: number of edges for this time **integer**
- **i_gnumfine**: number of edges on finest level **integer**
- **i_gnumboun**: number of boundary edges **integer**
- **i_ntotal**: total number of nodes **integer**
- **i_nnumber**: number of nodes for this time **integer**
- **i_maxlvl**: maximum (finest) level of refinement **integer**
- **i_minlvl**: minimum (coarsest) level of refinement **integer**
- **i_reflvlboun**: finest predefined refinement level **integer**
- **i_crslvlboun**: coarsest refinement level (macro triangulation) **integer**
- **i_unknows**: total number of unknowns **integer**, **DIMENSION(i_femtypes)**,
where **i_femtypes** = **grid_femtypes%i_numtypes**.

p_grid type(**grid_handle**), **DIMENSION(GRID_timesteps)** :

This is an instance of the grid data structure. **p_grid** will hold the above given data for the grid.

grid_param (Type definition):

This is **amatos**'s fundamental parameter data structure, containing different types of information. It contains the following components:

- **i_stringlength**: length of character strings **integer**
- **program_name**: name of program **character**
- **author_name**: name of the author **character**
- **author_affil1**: affiliation string for author **character**
- **author_affil2**: affiliation string for author **character**
- **author_affil3**: affiliation string for author **character**
- **author_email**: email address of author **character**
- **version**: version of **amatos** **integer**
- **subversion**: subversion **integer**
- **patchversion**: patch level **integer**
- **datemonth**: release date month **integer**
- **dateyear**: release date year **integer**
- **ioin**: input unit number (for redirection) **integer**
- **ioout**: output unit number (for redirection) **integer**
- **iolog**: log file unit number **integer**

GRID_parameters type(grid_param) :

This is an instance of the parameter data structure. GRID_parameters will hold the above given data.

GRID_EPS (real):

Machine precision.

grid_femsignatur (Type definition):

This data structure defines properties of different finite elements supported by **amatos**. It contains the following components:

- **c_name**: name of finite element **character**
- **i_order**: discretization order of finite element **integer**
- **i_unknowns**: total number of unknowns per element **integer**,
this value corresponds to

$$\text{GRID_elementnodes} \cdot \text{i_npoints} + \text{GRID_elementedges} \cdot \text{i_gpoints} + \text{i_lepoints}$$

- **i_npoints**: number of points per node **integer**
- **i_gpoints**: number of points per edge **integer**
- **i_lepoints**: number of points per element **integer**
- **r_gweights**: weights for point position calculation at edges (barycentric coordinates) **real**
- **r_ewights**: weights for point position calculation at elements (barycentric coordinates) **real**

grid_femtypearr (Type definition):

This data structure contains all signatures for finite elements supported by **amatos**. It contains the following components:

- **i_numtypes**: number of FEM types supported **integer**
- **p_signatures**: array of signatures for each type **fem_signatur**

GRID_femtypes (type(grid_femtypearr)):

Instance of **grid_femtypearr**.

GRID_SR (integer):

Single precision real KIND parameter.

GRID_DR (integer):

Double precision real KIND parameter.

GRID_SI (integer):

Single precision integer KIND parameter.

GRID_DI (integer):

Double precision integer KIND parameter.

5.2 Physics-Related Constants

GRID_RADIUS (real):

Radius of the sphere [m] (**only available in samatos**).

GRID_PI (real):

π .

GRID_SIDDAY (real):

Sideric day [s].

GRID_GRAV (real):

Gravitational constant g [ms^{-2}].

GRID_OMEGA (real):

Angular velocity of the earth Ω [s^{-1}] (**only available in samatos**).

6 Routines in the GRID Application Programming Interface

6.1 Routines for mesh creation and termination

grid_initialize:	
Syntax:	subroutine grid_initialize(INTEGER i_output, INTEGER i_logging)
Input:	i_output (opt): Redirect standard output to unit i_output. i_logging (opt): Invoke logging to file on unit i_logging.
Output:	none
Description:	This subroutine initializes <code>amatos</code> . It must be called before any other routines in <code>amatos</code> and also before any use of variables provided by <code>amatos</code> .
grid_definegeometry:	
Syntax:	subroutine grid_definegeometry(INTEGER i_vertices, INTEGER i_dimensions, INTEGER i_polylines, INTEGER i_polymask, REAL r_vertexarr)
Input:	i_vertices: Number of vertices in vertex array i_dimensions (opt): Number of dimensions (default 2). i_polylines (opt): Number of polygons for islands (default 1). i_polymask (opt): Number of vertices for each polygon (only valid, if <code>i_polylines</code> is also given). r_vertexarr (opt): array with boundary polygon vertices DIMENSION(i_dimension,i_vertices) (default unit square).
Output:	none
Description:	This subroutine defines the computational domain. It must be called before creating a mesh.
grid_createinitial:	
Syntax:	subroutine grid_createinitial(type(grid_handle) p_mesh, character c_filename)
Input:	p_mesh: grid handle data structure c_filename (opt): file name for the initial grid definitions (default "Triang.dat").
Output:	p_mesh: grid handle data structure
Description:	This subroutine reads the initial grid from a file. It creates all necessary data structures and refines globally and uniformly up to the coarsest user-given level. It is called at program start.
grid_readinitial:	
Syntax:	subroutine grid_readinitial(type(grid_handle) p_mesh, character c_filename)
Input:	p_mesh: grid handle data structure c_filename (opt): file name for the initial grid definitions (default "amatos_save.save").

Output: p_mesh: grid handle data structure

Description: This subroutine reads a complete grid from a save set.

grid_terminate:	
Syntax:	subroutine grid_terminate
Input:	none
Output:	none
Description:	This subroutine destroys all mesh related data structures. It should be called at program termination.

6.2 Routines for saving and restoring the mesh

grid_writesaveset:	
Syntax:	subroutine grid_writesaveset(CHARACTER (len=32) c_file, type(grid_handle) p_handle)
Input:	c_file: output file name. p_mesh: grid handle data structure.
Output:	none
Description:	This subroutine writes the whole data structure of the mesh including corresponding data to an unformatted file. This serves as a possibility to save (and restore with <code>grid_readsaveset</code>) break points in a numerical simulation. Note that some of the mesh information gets lost. For example, no information about new mesh items can be preserved. Global internal mesh object identifiers may also change after restoring the mesh.

grid_readsaveset:	
Syntax:	subroutine grid_readsaveset(CHARACTER (len=32) c_file, type(grid_handle) p_handle)
Input:	c_file: file name from which to read. p_mesh: grid handle data structure.
Output:	p_mesh: updated grid handle data structure
Description:	This subroutine reads a previously saved mesh data structure from file <code>c_file</code> . This serves as a possibility to save and restore break points in a numerical simulation.

6.3 Routines for data retrieval

grid_getinfo:	
Syntax:	<pre> subroutine grid_getinfo(type(grid_handle) p_mesh, LOGICAL l_finelevel, LOGICAL l_relative, INTEGER i_arraypoint, INTE- GER i_femtype, INTEGER i_newsdepth, INTEGER i_nlength, INTE- GER i_glength, INTEGER i_elength, REAL r_dofcoordinates, REAL r_dofsphericals, REAL r_dofvalues, INTEGER i_dofboundary, INTEGER i_dofpatch, REAL r_nodecoordinates, REAL r_nodesphericals, REAL r_nodevalues, INTEGER i_nodeboundary, INTEGER i_nodepatch, INTEGER i_nodedofs, INTEGER i_edgedofs, INTEGER i_edgeinnerdofs, INTE- GER i_edgenodes, INTEGER i_edgeboundary, REAL r_elementwidth, INTEGER i_elementdofs, INTEGER i_elementinnerdofs, INTEGER i_elementnodes, INTEGER i_elementstatus, INTEGER i_elementlevel, INTEGER i_elementmark, INTEGER i_elementproc,) </pre>
Input:	<p>p_mesh: grid handle data structure</p> <p>l_finelevel (opt): toggles to finelevel info</p> <p>l_relative (opt): give index numbers relative to one (i.e. consecutively numbered).</p> <p>i_arraypoint (opt): Array of variable identifiers to be retrieved, variables have to be registered (preregistered GRID_ucomp, GRID_vcomp, GRID_phi, GRID_zeta, GRID_tracer).</p> <p>i_femtype (opt): type of FEM that is considered, this is required input for r_dofcoordinates, i_dofboundary, i_dofpatch, i_edgedofs, i_edgeinnerdofs, i_elementdofs, i_elementinnerdofs, r_dofsphericals.</p> <p>i_newsdepth (opt): if given, only those grid items are collected that have been created in the last i_newsdepth inner iterations. Length values for the arrays are returned in i_nlength, i_glength, and i_elength respectively.</p>
Output:	<p>r_dofcoordinates (opt): array for coordinates of degrees of freedom (DOF) DIMENSION(GRID_dimension,i_len), where i_len = p_mesh%i_unknowns(i_femtype).</p> <p>r_dofsphericals (opt): array for spherical coords. of DOFs DIMENSION(GRID_dimension,i_len), where i_len = p_mesh%i_unknowns(i_femtype) (samatos only).</p> <p>r_dofvalues (opt): array for values at DOFs DIMENSION(i_point, i_len), where i_len = p_mesh%i_unknowns(i_femtype); i_point = size(i_arraypoint), pointers to registered variables have to be given in i_arraypoint.</p> <p>i_dofboundary (opt): boundary flags for all DOFs DIMENSION(i_len), where i_len = p_mesh%i_unknowns(i_femtype).</p> <p>i_dofpatch (opt): patch element indices for all DOFs DIMENSION(GRID_patchelements,i_len), where i_len = p_mesh%i_unknowns(i_femtype).</p> <p>r_nodecoordinates (opt): array for the nodal coordinates DIMENSION(GRID_dimension,i_len), where i_len = p_mesh%i_nnumber.</p>

Output (contd.):	<code>r_nodesphericals (opt):</code>	array for the nodal spherical coordinates <code>DIMENSION(GRID_dimspherical,i_len)</code> , where <code>i_len = p_mesh%i_nnumber</code> (samatosonly).
	<code>i_nodeboundary (opt):</code>	boundary flags for all nodes <code>DIMENSION(i_len)</code> , where <code>i_len = p_mesh%i_nnumber</code> .
	<code>i_nodepatch (opt):</code>	patch element indices for all nodes <code>DIMENSION(GRID_patchelements,i_len)</code> , where <code>i_len = p_mesh%i_nnumber</code> .
	<code>i_nodedofs (opt):</code>	DOF indices corresponding to nodes <code>DIMENSION(i_npoints,i_len)</code> , where <code>i_npoints</code> DOFs per node, <code>i_len = p_mesh%i_nnumber</code> .
	<code>i_edgedofs (opt):</code>	DOF indices corresponding to edges <code>DIMENSION(i_gpoints,i_len)</code> , where <code>i_gpoints</code> DOFs per edge, <code>i_len = p_mesh%i_gnumber</code> , or <code>i_len = p_mesh%i_gnumfine</code> if <code>l_finelevel = .TRUE.</code>
	<code>i_edgeinnerdofs (opt):</code>	DOF indices corresponding exclusively to edges <code>DIMENSION(i_gpoints,i_len)</code> , where <code>i_gpoints</code> DOFs per edge, <code>i_len = p_mesh%i_gnumber</code> , or <code>i_len = p_mesh%i_gnumfine</code> if <code>l_finelevel = .TRUE.</code>
	<code>i_edgenodes (opt):</code>	edge's node indices <code>DIMENSION(GRID_edgenodes, i_len)</code> , <code>i_len = p_mesh%i_gnumber</code> , or <code>i_len = p_mesh%i_gnumfine</code> if <code>l_finelevel = .TRUE.</code>
	<code>i_edgeboundary (opt):</code>	boundary flags for all edges <code>DIMENSION(i_len)</code> , <code>i_len = p_mesh%i_gnumber</code> , or <code>i_len = p_mesh%i_gnumfine</code> if <code>l_finelevel = .TRUE.</code>
	<code>r_elementwidth (opt):</code>	mesh width element-wise <code>DIMENSION(i_len)</code> , <code>i_len = p_mesh%i_enumber</code> , or <code>i_len = p_mesh%i_enumfine</code> if <code>l_finelevel = .TRUE.</code>
	<code>i_elementdofs (opt):</code>	DOF indices corresponding to elements <code>DIMENSION(i_epoints,i_len)</code> , where <code>i_epoints</code> DOFs per element, <code>i_len = p_mesh%i_enumber</code> , or <code>i_len = p_mesh%i_enumfine</code> if <code>l_finelevel = .TRUE.</code>
	<code>i_elementinnerdofs (opt):</code>	DOF indices corresponding exclusively to elements <code>DIMENSION(i_epoints,i_len)</code> , where <code>i_epoints</code> DOFs per element, <code>i_len = p_mesh%i_enumber</code> , or <code>i_len = p_mesh%i_enumfine</code> if <code>l_finelevel = .TRUE.</code>
	<code>i_elementnodes (opt):</code>	element's node indices <code>DIMENSION(GRID_elementnodes, i_len)</code> , <code>i_len = p_mesh%i_enumber</code> , or <code>i_len = p_mesh%i_enumfine</code> if <code>l_finelevel = .TRUE.</code>
	<code>i_elementstatus (opt):</code>	status of each element <code>DIMENSION(i_len)</code> , <code>i_len = p_mesh%i_enumber</code> , or <code>i_len = p_mesh%i_enumfine</code> if <code>l_finelevel = .TRUE.</code>
	<code>i_elementlevel (opt):</code>	level of refinement of each element <code>DIMENSION(i_len)</code> , <code>i_len = p_mesh%i_enumber</code> , or <code>i_len = p_mesh%i_enumfine</code> if <code>l_finelevel = .TRUE.</code>

Output
(contd.):

`i_elementmark` (opt): marked edge in each element
`DIMENSION(i_len)`, `i_len = p_mesh%i_enumber`, or
`i_len = p_mesh%i_enumfine` if `l_finelevel = .TRUE.`

`i_elementproc` (opt): processor number for each element
`DIMENSION(i_len)`, `i_len = p_mesh%i_enumber`, or
`i_len = p_mesh%i_enumfine` if `l_finelevel = .TRUE.`
(parallel version only).

Description: This subroutine, contained in module `FEM_dataretrieve` (exported), allows retrieval of information from the mesh. It can also be seen as the “gather” operation of `amatos`. It gathers information from grid items like nodes, elements, etc. and returns consecutively filled arrays.

The philosophy behind this procedure is that the numerical parts of the simulation software work on consecutive arrays, while the mesh is built from (mesh) objects.

In `amatos` version 1.2 `grid_getinfo` has substantially been revised. Now, any number of information can be retrieved in one call to `grid_getinfo`. For example, the following call would be allowed and give proper results (namely an array of size $dim \times \#nodes$ with node coordinates and an array of length $\#elements$ with elements status):

```
CALL grid_getinfo(p_mesh, l_finelevel=.TRUE., &
                 r_nodecoordinates=realarr, i_elementstatus=intarr)
```

If one likes to retrieve values corresponding to DOFs of all elements, two arrays have to be retrieved by `grid_getinfo`: The `r_dofvalues`-array, and the `i_elementdofs` index array.

<code>grid_putinfo:</code>

Syntax: subroutine `grid_putinfo`(type(grid_handle) `p_mesh`, LOGICAL
`l_finelevel`, INTEGER `i_arraypoint`, INTEGER `i_newsdepth`, REAL
`r_dofvalues`, REAL `r_nodevalues`, INTEGER `i_elementstatus`, INTEGER
`i_elementproc`,)

Input:

`p_mesh`: grid handle data structure

`l_finelevel` (opt): toggles to finelevel info

`i_arraypoint` (opt): Array of variable identifiers to
be retrieved, variables have to be registered
(preregistered `GRID_ucomp`, `GRID_vcomp`, `GRID_phi`,
`GRID_zeta`, `GRID_tracer`).

`i_newsdepth` (opt): if given, only those grid items are collected that
have been created int the last `i_newsdepth` inner iterations.
Length values for the arrays are returned in `i_nlength`,
`i_glength`, and `i_elenlength` respectively.

`r_dofvalues` (opt): array for values at DOFs `DIMENSION(i_point, i_len)`,
where `i_len = p_mesh%i_unknowns(i_femtype)`;
`i_point = size(i_arraypoint)`, pointers to registered
variables have to be given in `i_arraypoint`.

`i_elementstatus` (opt): status of each element
`DIMENSION(i_len)`, `i_len = p_mesh%i_enumber`, or
`i_len = p_mesh%i_enumfine` if `l_finelevel = .TRUE.`

`i_elementproc` (opt): processor number for each element
`DIMENSION(i_len)`, `i_len = p_mesh%i_enumber`, or
`i_len = p_mesh%i_enumfine` if `l_finelevel = .TRUE.`
(parallel version only).

Output: none

Description: This subroutine, contained in module `FEM_dataretrieve` (exported), allows to update information on grid items. In contrast to `grid_getinfo` it can be seen as the “scatter” operation of `amatos`. It scatters information from consecutive arrays to all grid items like nodes, elements, etc.

CAUTION: The grid should never be changed between a call to `grid_getinfo` and a corresponding call to `grid_putinfo`.

Like `grid_getinfo`, `grid_putinfo` has been changed substantially to reflect the changes in the former routine. One can give any number of update arrays in one single call.

By specifying `i_scatterlength` and `i_scatterindex` the user is able to only update a specific set of grid items (see description of `grid_getinfo`).

grid_getiteminfo:

Syntax: subroutine `grid_getiteminfo`(`INTEGER i_itemindex`, `CHARACTER c_itemtype`, `INTEGER i_arrlen`, `REAL r_values`, `REAL r_coordinates`, `REAL r_sphericals` `INTEGER i_nodes`, `INTEGER i_edges`, `INTEGER i_elements`, `INTEGER i_status`, `INTEGER i_time`, `INTEGER i_level`, `INTEGER i_patch`, `INTEGER i_boundary`, `INTEGER i_femtype`, `REAL r_dofcoordinates`, `REAL r_dofvalarray`, `INTEGER i_dofvalindex`)

Input:

<code>i_itemindex:</code>	index of mesh item
<code>c_itemtype:</code>	4-character string for item type, allowed are 'elmt', 'edge', 'node'.
<code>i_arrlen (opt):</code>	array length.
<code>i_time (opt):</code>	time tag for the item status (default <code>i_futuretime</code>).
<code>i_femtype (opt) :</code>	FEM type for DOF retrieval.
<code>i_dofvalindex (opt) :</code>	Array with indices (handles) for registered variables at DOF positions.

Output:

<code>r_values:</code>	values array.
<code>r_coordinates:</code>	nodal coordinates ('node' only).
<code>r_sphericals:</code>	nodal spherical coordinates ('node' only, spherical version only).
<code>i_nodes:</code>	node indices ('elmt', 'edge' only).
<code>i_edges:</code>	edge indices ('elmt' only).
<code>i_elements:</code>	element indices ('edge' only).
<code>i_status:</code>	status ('elmt', 'edge' only).
<code>i_level:</code>	level ('elmt', 'edge' only).
<code>i_boundary:</code>	boundary condition ('edge' only).
<code>i_patch:</code>	patch element indices ('node' only), <code>i_arrlen</code> returns no. of patch elements.
<code>r_dofcoordinates :</code>	coordinates of corresponding DOFs.
<code>r_dofvalues :</code>	values at DOFs for registered variables.

Description: This subroutine, contained in module `FEM_dataretrieve` (exported), allows to retrieve information from individual grid items. Like `grid_getinfo` and `grid_putinfo` it operates only on the information corresponding to the given array.

grid_getpolyline:

Syntax: subroutine `grid_getpolyline`(`type(grid_handle) p_mesh`, `INTEGER i_linetype`, `INTEGER i_arrlen`, `INTEGER i_efflen`, `REAL r_vertices`)

Input: **p_mesh:** grid handle data structure.
 i_linetype: type of polygonal line.
 i_arrlen: array length of **r_vertices**.

Output: **i_efflen:** fill length of array.
 r_vertices: vertices of the polygon
 DIMENSION(**GRID_dimension**, **i_arrlen**)

Description: This subroutine retrieves the polygonal lines defining the boundary. Either the boundary defining line can be retrieved (**i_linetype= GRID_boundary**) or the line made from the boundary edges (**i_linetype= GRID_boundedges**).

grid_findelmt:	
-----------------------	--

Syntax: **function grid_findelmt**(**REAL r_coord**, **type(grid_handle) p_ghand**)
 result(**INTEGER i_found**)

Input: **r_coord:** coordinate array.
 DIMENSION(**GRID_dimension**)
 p_ghand: grid handle data structure.

Output: **i_found:** index of found element.

Description: This function returns the index of an element containing coordinate **r_coord**.

6.4 Routines supporting FEM variables

grid_registerfemvar:	
-----------------------------	--

Syntax: **function grid_registerfemvar**(**INTEGER i_femtype**)
 result(**INTEGER i_varindex**)

Input: **i_femtype:** Finite element type the variable is to
 be associated with.

Output: **i_varindex:** index (handle) for newly registered variable.

Description: This function, contained in module **FEM_signature**, returns a handle for variables stored at the DOFs corresponding to the given FEM type (**i_femtype**). Each variable that is used by an application has to be registered by this function to a supported FEM type.
 This function has to be called before any grid item is created, so it is best to call it right after **grid_initialize**.
 There are five (in **samatos** six) variables pre-registered by default. Handles to these variables, registered to the default linear element type (see section 3), are **GRID_ucomp**, **GRID_vcomp**, (**GRID_wcomp** only **samatos**), **GRID_phi**, **GRID_zeta**, and **GRID_tracer**.

grid_femtypequery:	
---------------------------	--

Syntax: **subroutine grid_femtypequery**(**INTEGER i_femtype**, **INTEGER i_order**,
 CHARACTER c_description, **INTEGER i_unknowns**)

Input: **i_femtype:** Finite element type.

Output: **i_order (opt):** order of FEM.
 c_description (opt) : description string of FEM.
 i_unknowns (opt) : number of unknowns.

Description: Given a supported FEM type, this subroutine (contained in module `FEM_signature`) returns information for that particular FEM type. Information is returned for each argument specified.

grid_femvarquery:	
Syntax:	<code>function grid_femvarquery(INTEGER i_varindex) result(INTEGER i_femtype)</code>
Input:	<code>i_varindex:</code> index (handle) for registered variable.
Output:	<code>i_femtype:</code> Finite element type the variable is registered.
Description:	This function, contained in module <code>FEM_signature</code> , returns the FEM type for a given variable index (handle).

6.5 Routines for numerical calculation

grid_coordvalue:	
Syntax:	<code>function grid_coordvalue(type(grid_handle) p_mesh, REAL r_coordinate, INTEGER i_interpolorder, INTEGER i_valpoint, INTEGER i_index, INTEGER i_domaincheck) result(REAL r_value)</code>
Input:	<p><code>p_mesh:</code> grid handle data structure.</p> <p><code>r_coordinate:</code> coordinate at which to evaluate <code>DIMENSION(GRID_dimension)</code>.</p> <p><code>i_interpolorder:</code> order of interpolation (allowed <code>GRID_loworder</code> <code>GRID_highorder</code>, <code>GRID_thinplate</code>).</p> <p><code>i_valpoint (opt):</code> nodal value to be retrieved (possible choices <code>GRID_ucomp</code>, <code>GRID_vcomp</code>, <code>GRID_phi</code>, <code>GRID_zeta</code>, <code>GRID_tracer</code>).</p> <p><code>i_domaincheck (opt):</code> value of a previous call to <code>grid_domaincheck</code>, this prevents a second call within <code>grid_coordvalue</code>.</p>
Output:	<p><code>i_index (opt):</code> if given, the index of the fine mesh element containing <code>r_coordinate</code> is returned.</p> <p><code>r_value:</code> interpolated value.</p>
Description:	<p>This function evaluates the finite element function corresponding to the tracer, vorticity, geopot. height, wind components resp. at the position given by <code>r_coordinate</code>.</p> <p>Low order (bi-linear), high order (bi-cubic spline), and thin plate spline radial basis function interpolations are provided.</p> <p>In default operation, before calculating an interpolation, <code>grid_coordvalue</code> checks if the requested coordinate lies within the computational domain, i.e. no extrapolation is performed. If the user is sure that the coordinate value lies within the domain, <code>i_domaincheck</code> can be specified in the same way as would be by a call to <code>grid_domaincheck</code>.</p>

grid_coordgradient:	
Syntax:	<code>function grid_coordgradient(type(grid_handle) p_mesh, REAL r_coordinate, INTEGER i_interpolorder, INTEGER i_valpoint, INTEGER i_index, INTEGER i_domaincheck) result(REAL r_value)</code>

Input:

p_mesh:	grid handle data structure.
r_coordinate:	coordinate at which to evaluate DIMENSION(GRID_dimension).
i_interpolorder:	order of interpolation (allowed GRID_loworder GRID_highorder, GRID_thinplate).
i_valpoint (opt):	nodal value to be retrieved (possible choices GRID_ucomp, GRID_vcomp, GRID_phi, GRID_zeta, GRID_tracer).
i_domaincheck (opt):	value of a previous call to grid_domaincheck, this prevents a second call within grid_coordgradient.

Output:

i_index (opt):	if given, the index of the fine mesh element containing r_coordinate is returned.
r_value:	interpolated value.

Description: This function evaluates the finite element function corresponding to the tracer, vorticity, geopot. height, wind components resp. at the position given by r_coordinate and calculates the gradient.
Only low order (bi-linear) gradient estimation is provided in release 1.2.
In default operation, before calculating an interpolation, grid_coordgradient checks if the requested coordinate lies within the computational domain, i.e. no extrapolation is performed. If the user is sure that the coordinate value lies within the domain, i_domaincheck can be specified in the same way as would be by a call to grid_domaincheck.

grid_integral:

Syntax: function grid_integral(type(grid_handle) p_mesh, INTEGER
i_valpoint, REAL r_watermark, LOGICAL l_lowbound)
result(REAL r_value)

Input:

p_mesh:	grid handle data structure.
i_valpoint:	nodal value to be retrieved. (possible choices GRID_ucomp, GRID_vcomp, GRID_phi, GRID_zeta, GRID_tracer).
r_watermark:	watermark for integration (see below).
l_lowbound:	lower/upper bound toggle (see below).

Output: r_value: integral over the domain.

Description: This calculates the (discrete) integral over the domain. It is possible to give a certain watermark in order to integrate only over those regions, where the integrated function is above (below) a certain level. r_watermark is used as a lower bound, if l_lowbound is true (default), otherwise it is used as an upper bound.

grid_nodearea:

Syntax: subroutine grid_nodearea(type(grid_handle) p_mesh, INTEGER i_siz,
REAL r_area, INTEGER i_selectlength, INTEGER i_selectindex)

Input:

p_mesh:	grid handle data structure.
i_siz:	array size.
i_selectlength (opt.):	array size for selection array.
i_selectindex (opt.):	index array for selected nodes.

Output: **i_siz:** effective array size.
 r_area: array with nodal areas of influence
 DIMENSION(i_siz).

Description: This subroutine calculates an area of influence for each node of the grid or – if **i_selectlength** and **i_selectindex** is given – a selection of nodes (this is calculated by the following formula:

$$a = \frac{1}{3} \sum_{\tau \in S} |\tau|,$$

where a is the node area, S is the set of surrounding elements, and $|\tau|$ denotes the area of element τ).

grid_polygridintersect:	
Syntax:	subroutine grid_polygridintersect(type(grid_handle) p_handle, INTEGER i_vert, REAL r_vertcoo, INTEGER i_len, INTEGER i_triang, REAL r_area, LOGICAL l_relative)
Input:	p_handle: grid handle data structure. i_vert: number of vertices of polygon. r_vertcoo: array of vertex coordinates, DIMENSION(GRID_dimension,i_vert). l_relative (opt): toggle for absolute and relative indices.
Output:	i_len: output array size. i_triang: array with intersecting mesh elements, DIMENSION(i_len), POINTER. r_area (opt): array with intersection areas, DIMENSION(i_len), POINTER.
Description:	This subroutine, contained in module FEM_gridmanag (exported by Klaschka 09/2006), calculates the intersection of a given polygon with the mesh. The polygon intersection algorithm is taken from Alan Murtas gpc library (http://www.cs.man.ac.uk/aig/staff/alan/software/). The (pointer) array i_triang will contain the indices of all elements that have an intersection with the polygon. r_area (if given) will contain the intersection area corresponding to each element in i_triang . Both output arrays, i_triang and r_area , are allocated in grid_polygridintersect , so they must be deallocated after use! If l_relative is .TRUE. then a relative indexing scheme is returned in i_triang , corresponding to the list of nodes retrieved by grid_getinfo .

grid_nodegradient:	
Syntax:	function grid_nodegradient(INTEGER i_node, INTEGER i_valpoint, INTEGER i_time) result(REAL r_deriv)
Input:	i_node: node index. i_valpoint (opt): nodal value to be retrieved (possible choices GRID_ucomp, GRID_vcomp, GRID_phi, GRID_zeta, GRID_tracer). i_time (opt): time tag for the grid.
Output:	r_deriv: first derivatives. DIMENSION(GRID_dimension).

Description: This subroutine, contained in module `FEM_interpolation`, calculates the numerical gradient of a function defined at the nodes. The calculation is performed by means of the normal vectors on the triangles (not available in `pamatos`).

grid_kartgeo:	
Syntax:	function grid_kartgeo(REAL r_xyz) result(REAL r_lamphi)
Input:	r_xyz: coordinate in Karthesian geometry. DIMENSION(GRID_dimension).
Output:	r_lamphi: coordinate in geographical (λ, ϕ) geometry. DIMENSION(GRID_dimspherical).
Description:	This function calculates the geographical coordinates from Karthesian coordinates (only available in samatos).

grid_geokart:	
Syntax:	function grid_geokart(REAL r_lamphi) result(REAL r_xyz)
Input:	r_lamphi: coordinate in geographical (λ, ϕ) geometry. DIMENSION(GRID_dimspherical).
Output:	r_xyz: coordinate in Karthesian geometry. DIMENSION(GRID_dimension).
Description:	This function calculates the Karthesian coordinates from geographical coordinates (only available in samatos).

grid_edgelen :	
Syntax:	subroutine <code>grid_edgelen</code> (type(grid_handle) p_mesh, REAL r_max, REAL r_min, REAL r_edgelen)
Input:	p_mesh: grid handle data structure.
Output:	r_max (opt): maximum edge length. r_min (opt): minimum edge length. r_edgelen (opt): array of edge lengths. DIMENSION(p_mesh%i_gnumfine).
Description:	This subroutine returns edge lengths of all edges in the mesh.

6.6 Routines for controlling the mesh

grid_adapt :	
Syntax:	subroutine <code>grid_adapt</code> (type(grid_handle) p_mesh, LOGICAL l_changed)
Input:	p_mesh: grid handle data structure.
Output:	p_mesh: grid handle data structure (changed). l_changed: indicator for performed changes in mesh.
Description:	This subroutine adapts the mesh according to given “flags”. The user has to mark elements for refinement or coarsening, then <code>grid_adapt</code> changes the mesh and cares for the admissibility, etc. If <code>l_changed</code> is true, then the grid has really changed. Otherwise no refinement or coarsening has been performed.

grid_timeduplicate :	
Syntax:	subroutine <code>grid_timeduplicate</code> (type(grid_handle) p_grid1, type(grid_handle) p_grid2)
Input:	p_grid1: grid handle data structure (old grid). p_grid2: grid handle data structure (empty new grid).
Output:	p_grid2: grid handle data structure (new duplicate of old grid).
Description:	This subroutine, contained in module <code>FEM_gridgen</code> (exported), duplicates a grid. It takes a grid handle from an existing grid and duplicates it. All necessary grid items for the new (duplicated) grid are created. A call to this routine resets the new items counter (see <code>grid_newitems</code>).

grid_timetoggle :	
Syntax:	subroutine <code>grid_timetoggle</code>
Input:	none
Output:	none
Description:	This subroutine toggles the time tags, <code>i_time</code> → <code>i_futuretime</code> , <code>i_pasttime</code> → <code>i_time</code> , etc.

grid_setparameter:	
Syntax:	subroutine grid_setparameter(type(grid_handle) p_mesh, INTEGER i_coarselevel, INTEGER i_finelevel)
Input:	p_mesh: grid handle data structure. i_coarselevel (opt): coarsest level of refinement (upper bound). i_finelevel (opt): finest level of refinement (lower bound).
Output:	p_mesh: grid handle data structure (changed).
Description:	This subroutine is an interface to set the coarse and fine level parameters.
grid_sweep:	
Syntax:	subroutine grid_sweep
Input:	none
Output:	none
Description:	This subroutine removes obsolete data items from the mesh. It should be called from time to time during the execution (e.g. at the end of each time step).
grid_domaincheck:	
Syntax:	function grid_domaincheck(type(grid_handle) p_mesh, REAL r_coordinate) result(i_inout)
Input:	p_mesh: grid handle data structure. r_coordinate: coordinate to be checked.
Output:	i_inout: indicator for inside/outside.
Description:	This checks if a given coordinate is inside or outside of the computational domain. It returns 0 (zero), if the coordinate is inside, -1 otherwise.
grid_boundintersect:	
Syntax:	subroutine grid_boundintersect(type(grid_handle) p_mesh, REAL r_start, REAL r_end, INTEGER i_info) result(REAL r_intersect)
Input:	p_mesh: grid handle data structure. r_start: coordinate of starting point. r_end: coordinate of end point. i_info (opt): information on intersection.
Output:	r_intersect: intersection coordinate.
Description:	This function calculates the intersection point of a line given by r_start and r_end with the boundary. The calculated result can only be trusted, if i_info =0.

6.7 Routines for the dual mesh

grid_createdual:	
Syntax:	subroutine grid_createdual(type(grid_handle) p_mesh, INTEGER i_duallen, INTEGER i_dualedge, REAL r_dualcoor, INTEGER i_selectlength, INTEGER i_selectindex)
Input:	p_mesh: grid handle data structure. i_selectlength (opt): length of selection array. i_selectindex (opt): selection array for special nodes.
Output:	i_duallen: effective length of arrays. i_dualedge: element-edge-array for the dual grid. DIMENSION(2, GRID_nodepatch, i_origlen). r_dualcoor: node coordinates for the dual grid. DIMENSION(GRID_dimension, i_duallen).
Description:	This subroutine calculates information for the dual mesh. <code>i_dualedge</code> stores two indices (start and endpoint of an edge) for each element in the dual mesh. The dual mesh has as many elements as the original mesh's nodes. <code>r_dualcoor</code> stores the coordinates of the dual nodes, pointed at by <code>i_dualedge</code> . Both arrays, <code>r_dualcoor</code> and <code>i_dualedge</code> are declared as Fortran 90 pointer arrays.

grid_destroydual:	
Syntax:	subroutine grid_destroydual(INTEGER i_duallen, INTEGER i_dualedge, REAL r_dualcoor)
Input:	i_duallen: array length. i_dualedge: element-edge array. r_dualcoor: coordinate array.
Output:	none
Description:	This subroutine destroys the dual mesh, created by <code>grid_createdual</code> .

6.8 Auxiliary Routines

grid_error:	
Syntax:	subroutine grid_error(integer i_error, CHARACTER c_error)
Input:	i_error (opt): numerical error code. c_error (opt): character string error code.
Output:	none
Description:	This subroutine provides a simple error handling interface. It is provided for convenience and ease of use and may be replaced by the user's own (and probably more sophisticated) error handling mechanism. If <code>i_error</code> is given and less than 20, a warning message is printed and execution will be continued. Otherwise, an error message is printed and the program is stopped immediately.

7 Initial Triangulation

The initial triangulation – usually consisting of only a few elements – has to be given by the user. This definition is handed over to **amatos** by a special input file. This section describes the input file (by default **Triang.dat**).

The input file is constructed in a certain way: *Keywords* precede the values for specified input parameters, *Comments* are marked by a '!' or '#' in the first column of a line. Each line contains either a Keyword or a value, never both!

In **amatos** release 1.2 a new (alternative) file format has been introduced. It is shorter and much more suitable for large initial triangulations. Instead of giving **NODE_***, **EDGE_***, and **ELEMENT_*** keywords (and data), new keywords **NODES_DESCRIPTION** and **ELEMENTS_DESCRIPTION** are provided. Edges are calculated from the information given in these two descriptions.

The following table lists the allowed Keywords:

Keyword	Description	Range of Values [Default]
GRID_DIMENSION	Global parameter defining the grid's space dimensions.	1 to 3 [2]
ELEMENT_VERTICES	Global parameter defining the shape of elements. In 2D for example, 3 is triangular mesh, 4 quadrilateral, etc.	3 to 4... [?]
NUMBER_OF_NODES	This defines the total number of nodes in the initial mesh.	1 to integer range [required]
NUMBER_OF_EDGES	This defines the total number of edges in the initial mesh.	1 to integer range [required]
NUMBER_OF_ELEMENTS	This defines the total number of elements in the initial mesh.	1 to integer range [required]
DEF_INNERITEM	This global value defines the attribute value for a grid item within the domain (no boundary).	- integer range to 0 [required]
DEF_DIRICHLETBOUNDARY	This global value defines the attribute value for a grid item on a Dirichlet boundary section of the domain.	- integer range to 0 [required]
DEF_NEUMANNBOUNDARY	This global value defines the attribute value for a grid item on a Neumann boundary section of the domain. [Note: periodic boundary conditions are defined by the (positive) index number of the corresponding periodic partner item in the grid.]	- integer range to 0 [required]
NODE_INDEXNUMBER	Defines the (unique) index of a node. This keyword occurs as often as the value of NUMBER_OF_NODES indicates.	1 to integer range [required]
NODE_COORDINATES	Defines a block of GRID_DIMENSION coordinates of the node.	- real range to + real range [required]
EDGE_INDEXNUMBER	Defines the (unique) index of an edge. This keyword occurs as often as the value of NUMBER_OF_EDGES indicates.	1 to integer range [required]
EDGE_NODEINDICES	Two indices of nodes (defined in the previous section) which span the edge.	1 to integer range [required]

Table continued

Keyword	Description	Range of Values [Default]
EDGE_ELEMENTINDICES	Two indices of elements adjacent to the edge. 0 for one side if the edge is at a (non-periodic) boundary.	1 to integer range [required]
EDGE_BOUNDARYCONDITION	Defines the boundary condition according to the previously defined values for Dirichlet, Neumann, or periodic boundary conditions.	- integer range to + integer range [required]
ELEMENT_INDEXNUMBER	Defines the (unique) index of an element. This keyword occurs as often as the value of NUMBER_OF_ELEMENTS indicates.	1 to integer range [required]
ELEMENT_NODEINDICES	Three indices of nodes which span the element.	1 to integer range [required]
ELEMENT_EDGEINDICES	Three indices of edges which span the element.	1 to integer range [required]
ELEMENT_MARKEDEDGE	The local edge number that is marked for refinement (by bisection).	1 to 3 [required]
NODES_DESCRIPTION	A list of rows with node descriptions for the short file format. The first entry is the index, the other two the (floating point) coordinates	1 to integer range, - real range to + real range [alternative]
ELEMENTS_DESCRIPTION	A list of rows with the following integer values: index, node 1, node 2, node 3, marked edge, boundary condition for edge 1, edge 2 , edge 3.	1 to integer range (first 4 entries), 1 to 3, - integer range to + integer range (last 3 entries) [alternative]

A sample triangulation file can be found in the **data** subdirectory of the **amatos** distribution.

8 Installation and Testing

8.1 Directory Structure

This section describes installation of the **amatos** software package. **amatos** is distributed as a source code **tar**-file for Unix systems. The installation and execution has been tested on IRIX, Solaris, and Linux Systems. A current list of tested operating systems can be found in the **doc** directory.

When unpacking the **tar**-archive with the following commands

```
unix> gunzip amatos2d.tar.gz
unix> tar xvf amatos2d.tar
```

a new directory is installed named **amatos2d** with a directory structure as given in Figure 2. In **data** some example input files can be found, while **doc** contains this documentation (as the reader might have realized by now...). Subdirectories **compile** contains OS-specific files for building the executables and libraries for **aix**, **irix**, **linux**, and **solaris**. **src** contains the Fortran 90 sources. **include** and **lib** contain the libraries and module files necessary for linking the executable.

The sources for **amatos** are all in subdirectory **src/gridgen**. In **src/test** we put the sources for a test driver. **src/system** contains some system dependent routines, mainly for command line input (we provide NAGWare, Posix compliant and Standard variants).

8.2 Building Library and Test Driver

As a preliminary step, build supportive libraries in the **3rdparty** directory:

1. If required (i.e. no optimized LAPACK interface is provided on your machine), build the “poor man’s LAPACK”^t, provided for convenience here:

```
unix> cd amatos2d/3rdparty/poor_mans_lapack/irix_mips
unix> make all
```

The files **libPMLAPACK.a** and **libPMLAPACK.so** should be available in **lib/irix_mips** after performing this step.

Now, the user has to decide whether the spherical version of **amatos** is required or the planar version. There are two files called **Makefile*_amatos** and **Makefile*_samatos** for building the planar and spherical versions resp. in the **compile/Makefiles** directory (a file **Makefile*_PML_amatos** is provided for illustrative reasons, when linking with **libPMLAPACK.so** is required). In Order to build the software library containing the programming interface to **amatos** the following easy procedure has to be taken.

1. Change into the directory corresponding to your target machine’s OS, e.g.

```
unix> cd amatos2d/compile/irix_mips
```

2. Copy either **Makefile_irix_amatos** or **Makefile_irix_samatos** from the **Makefiles** directory to **irix_mips**, but as default there is already available **Makefile** for **samatos** in there.
3. Change location specific settings in the **Makefile**.

```

amatos2d
  3rdparty
    poor_mans_lapack
  compile
    Makefiles
    aix_power
    irix_mips
    linux_ia32
    linux_ia64
    solaris_sparc
  data
  doc
  eval
  include
    Makefiles
    aix_power
    irix_mips
    linux_ia32
    linux_ia64
    solaris_sparc
  lib
    Makefiles
    aix_power
    irix_mips
    linux_ia32
    linux_ia64
    solaris_sparc
  src
    gridgen
    system
      nag-f90
      posix-f90
      std-f90
    test
    timing

```

Figure 2: Directory structure.

4. Type

```
unix> make lib
```

If everything went right, there should be a files `libamatos.a` and `libamatos.so` in the `lib/irix_mips` directory and some kind of `GRID_api.mod` in the `include/irix_mips` directory.

Now, in order to build the test driver (an executable program, called – guess – `AMATOS`) some similarly easy steps have to be taken, while still residing in the `compile/irix_mips` directory. Just type

```
unix> make AMATOS
```

(for the spherical case type `make SAMATOS`).

All the above steps (except for building the support libraries) can also be accomplished by just one command:

```
unix> make all
```

8.3 Running the Test Driver

In order to test `amatos` the executable `AMATOS`, built in the previous subsection, has to be executed. `AMATOS` can be called with several command line options. A complete list of these options can be obtained by calling

```
unix> AMATOS -h
```

Before any useful output from `AMATOS` can be expected, however, some data files have to be copied into the right position by typing

```
unix> make datacopy
```

After that, a useful output should be obtained by calling

```
unix> AMATOS -b -f Parameters.dat
```

8.4 Input Parameters for the Test Driver

Input for the executable `AMATOS` can be provided interactively or by means of a Parameter file that can be read by the program in batch mode. The program requires five different pieces of information:

1. The finest level of refinement (Keyword `FINE_GRID_LEVEL`).
2. The coarsest level of refinement (Keyword `COARSE_GRID_LEVEL`).
3. MATLAB output option (Keyword `MATLAB_PLOTTING`).
4. GMV output option (Keyword `GMV_FILE_PLOTTING`).
5. Filename for initial triangulation (Keyword `TRIANG_FILE_NAME`).

The file is constructed in the same way as the file defining the initial triangulation (see section 7).

A Copyright

COPYRIGHT NOTICE

This software is provided for non-commercial use only. See the license conditions in file LICENCE and the warranty conditions in file WARRANTY.

Copyright (c) 1997-2003 Jörn Behrens

B License

LICENSE

The use of amatos is hereby granted free of charge for an unlimited time, provided the following rules are accepted and applied:

1. You may use or modify this code for your own non commercial and non violent purposes.
2. The code may not be re-distributed without the consent of the authors.
3. The copyright notice and statement of authorship must appear in all copies.
4. You accept the warranty conditions (see file WARRANTY).
5. In case you intend to use the code commercially, we oblige you to sign an according licence agreement with the authors.

C Warranty

WARRANTY

This code has been tested up to a certain level. Defects and weaknesses, which may be included in the code, do not establish any warranties by the authors.

The authors do not make any warranty, express or implied, or assume any liability or responsibility for the use, acquisition or application of this software.