# Atlas

## A Flexible Parallel Framework for Earth System Modelling

## Version 0.8.0

## User-Guide

**November 22, 2016**

ECMWF, Shinfield Park, Reading, UK

# Contents

**Introduction** **v**

# I Getting Started 1

**1 Download and installation** **2**
1.1 General requirements . . . . . 2
1.2 Installation . . . . . 3
1.2.1 External third-party dependencies . . . . . 3
1.2.2 ECMWF third-party dependencies . . . . . 4
1.2.3 *Atlas* installation . . . . . 8
1.3 Inspecting your *Atlas* installation . . . . . 9
1.4 Using *Atlas* in your project . . . . . 10
1.4.1 C++ version . . . . . 11
1.4.2 Fortran version . . . . . 14

**2 Design** **19**
2.1 Grid . . . . . 19
2.2 Mesh . . . . . 22
2.2.1 Nodes . . . . . 22
2.2.2 Elements and Connectivity . . . . . 23
2.2.3 Mesh generation . . . . . 24
2.3 Field and FieldSet . . . . . 25
2.4 FunctionSpace . . . . . 25
2.5 Parallelisation . . . . . 26
2.6 Numerics . . . . . 27
2.7 Utilities . . . . . 27
2.7.1 Configuration . . . . . 28

2.7.2 Logging . . . . . . . . . . 28

**3 Theory** **29**
3.1 fvm: Median-dual Finite Volume Method . . . . . . . . 29

**II Core functionalities** **30**

**4 Create a Global Grid** **31**
4.1 Structured Grids . . . . . . . . . . 31
4.1.1 C++ version . . . . . . . . . . 32
4.1.2 Fortran version . . . . . . . . . . 34
4.2 Unstructured Grids . . . . . . . . . . 36

**5 Create a Mesh from a Grid** **37**
5.1 C++ version . . . . . . . . . . 37
5.2 Fortran version . . . . . . . . . . 39

**6 Create Fields and Field Sets** **42**
6.1 Standalone Fields and Field Sets . . . . . . . . . . 42
6.1.1 C++ version . . . . . . . . . . 42
6.2 Fortran version . . . . . . . . . . 46
6.3 Fields on a given Grid . . . . . . . . . . 50
6.3.1 C++ version . . . . . . . . . . 50
6.3.2 Fortran version . . . . . . . . . . 52

**7 Using the function space objects** **55**
7.1 NodeColumns . . . . . . . . . . 55
7.1.1 C++ version . . . . . . . . . . 56
7.1.2 Fortran version . . . . . . . . . . 63
7.2 StructuredColumns . . . . . . . . . . 71
7.3 Spectral . . . . . . . . . . 71

# Introduction

*Atlas* is an ECMWF software framework for parallel flexible data-structures supporting structured/unstructured grids, structured/unstructured meshes, various function spaces and utilities. The main aim of *Atlas* is to investigate and develop more scalable dynamical core options for numerical weather prediction (NWP). *Atlas* is also intended to create modern interpolation and product generation software.

*Atlas* is predominantly written in C++, with the main features available to Fortran through an F2003 interface. To be used effectively, it requires some knowledge of Unix (such as Linux). It is known to run on a number of systems, some of which are directly supported by ECMWF.

*Atlas* includes the following macro data objects.

- Grid: a list of coordinates (i.e. points) without connectivity rules;
- Mesh: a collection of elements linked by precise connectivity rules;
- Field: a physical quantity such as wind velocity or pressure;
- FieldSet: a collection of Fields;
- FunctionSpace: a given spatial discretization space (e.g. spectral, finite element, etc.).

From these objects it is possible to construct new algorithms to be tested within the context of numerical weather prediction (NWP), to generate and manipulate grids for production cases, etc. The overall structure of the library is depicted in figure 1.

From this figure, we note that there is the additional object called Metadata and related to the Field object. Metadata contains a description of a given Field (e.g. units, etc.). We also note that the Mesh object is formed by the Nodes and HybridElements objects, with the last being composed by Elements. These additional items represents the bricks to ultimately build the mesh object.

The structure in Fig. 1 will be further explained in chapter 2.

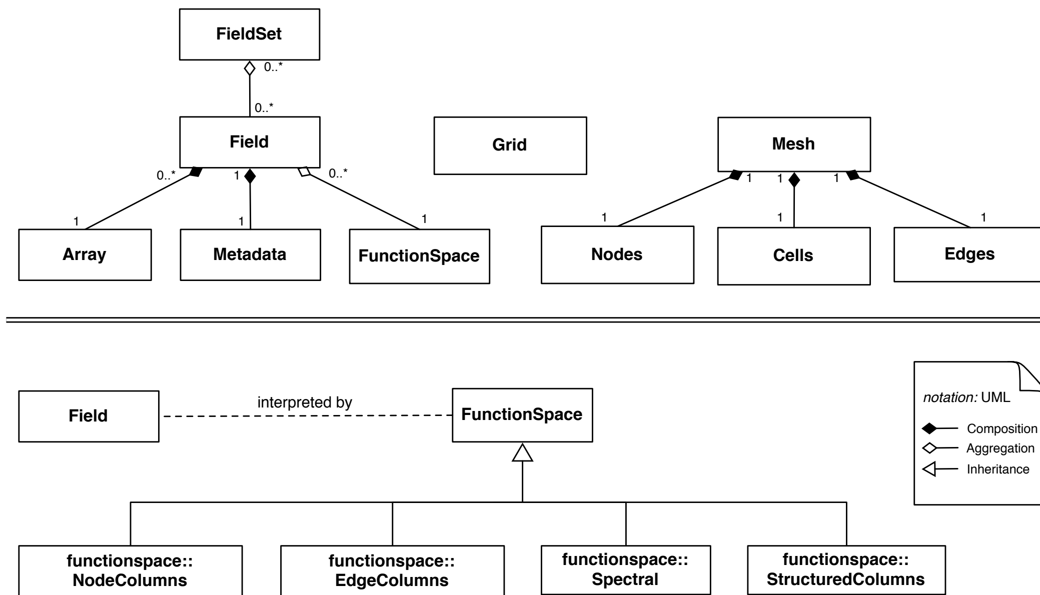



**Figure 1** Schematics of the *Atlas* library.

# Part I

# Getting Started

Chapter **1**

# Download and installation

This chapter is intended to be a general introduction on how to download, install and use *Atlas*. In particular, in section 1.1 we will present the general requirements of the library. In section 1.2 we will first describe how to install the third-party packages required by *Atlas* (if supported by ECMWF) and successively we will outline how to install *Atlas*. Finally, in section 1.4 we show how to use *Atlas* by creating a simple example that initializes and finalizes the library.

## 1.1 General requirements

*Atlas* is distributed as Git repository and is available at the ECMWF Stash git hosting service: `https://software.ecmwf.int/stash/projects`. This can only be accessed by ECMWF staff within the internal intranet. Occasionally, access can be granted to external partners working on specific projects.

*Atlas* is currently available under Stash as *Atlas* project itself or under the European project ESCAPE. These two repositories are separated - specifically the *Atlas* project under ESCAPE is a so-called Git fork of the main *Atlas* project. Given this structure of the *Atlas* project, one can find the library either in the main repository:
`https://software.ecmwf.int/wiki/display/ATLAS/Atlas`
or in the ESCAPE repository:
`https://software.ecmwf.int/stash/projects/ESCAPE/repos/atlas/browse`.
If you encounter any problem accessing these pages, please contact Willem Deconinck (`willem.deconinck@ecmwf.int`).

Note that the main *Atlas* project is intended for ECMWF internal developments, while the *Atlas* project under ESCAPE is intended for experimental developments within the ESCAPE project. Note also that *Atlas* requires third-party libraries as

described in the section 1.2 below.

Finally, *Atlas* has been tested and works correctly with the following compilers: GCC 4.8.1, Intel 13.0.1, 14.0.1 and CCE 8.2.7, 8.3.1.

## 1.2 Installation

*Atlas* requires a number of third-party libraries. Some of them are external third-party libraries not maintained by ECMWF - these external libraries are briefly described in section 1.2.1, where we also provide some useful links on how to download and install them.

Some other third-party libraries are developed and maintained by ECMWF. For this set of libraries we provide a download and installation instructions in section 1.2.2.

### 1.2.1 External third-party dependencies

*Atlas* requires the following external third-party libraries (some of the links provided may have changed, so we suggest the reader to lookup on the web for these packages):

- **Git**: Required for project management and to download the repository. For use and installation see `https://git-scm.com/`
- **CMake**: Required for configuration and cross-compilation purposes. For use and installation see `http://www.cmake.org/`
- **MPI**: Required for distributed memory parallelisation. For use and installation see for instance `https://www.open-mpi.org/`
- **Python**: Required for certain components of the build system. For use and installation see `https://www.python.org/`.
- **OpenMP** (optional): Required for shared memory parallelisation. For use and installation see `http://openmp.org/wp/`
- **boost_unit_test** (optional): Required for unit testing for C++. For use and installation see `http://www.boost.org/`
- **FFTW** (optional): Required for the Fast Fourier Transform. For use and installation see `http://www.fftw.org/`. This is only a dependency for the ECMWF *transi* project, described in section 1.2.2

Note that if you are an ECMWF staff member, you have some of the above libraries already available through the module system. In particular you can load the following packages as follows:

```
module load  git  cmake  python
```

If you are not an ECMWF staff member you need to either install them manually following the links above or ask your system administrator to verify whether these packages are already available within your working environment.

### 1.2.2 ECMWF third-party dependencies

*Atlas* additionally requires the following projects developed at ECMWF:

- **ecbuild**: It implements some CMake macros that are useful for configuring and cross-compiling *Atlas* and the other ECMWF third-party libraries required by *Atlas*. For further information, please visit: https://software.ecmwf.int/wiki/display/ECBUILD/ecBuild.
- **eckit**: It implements some useful C++ functionalities widely used in ECMWF C++ projects. For further information, please visit: https://software.ecmwf.int/wiki/display/ECKIT/ecKit
- **fckit** (optional): It implements some useful Fortran functionalities. For further information, please visit: https://software.ecmwf.int/stash/projects/ECSDK/repos/fckit/browse
- **transi** (optional): It implements the spectral transform. For further information, please visit: https://software.ecmwf.int/stash/projects/ATLAS/repos/transi/browse

In the following we will outline how to install each of the projects above.

The first step is to create a folder where to download, build and install all the third-party projects required as well as where to build and install *Atlas*. Let us call this folder `myproject`, create it and enter into the folder:

```
mkdir -p $(pwd)/myproject
cd myproject
```

We then need to create the following folder tree:

```
SRC=$(pwd)/sources
BUILDS=$(pwd)/builds
INSTALL=$(pwd)/install
mkdir -p $SRC
mkdir -p $BUILDS
mkdir -p $INSTALL
```

where the sources directory will contain the source files of each project, the builds directory will contain the built projects and the install directory will contain the installation of each project.
It is guaranteed at any point in time that all ECMWF projects have a git branches called “master” and “develop”. These branches respectively in each project are guaranteed to be compatible. The “master” branch contains the latest fixed release version of each project, whereas the “develop” branch contains the latest daily contributions to each project in preparation for future release versions. It is not guaranteed that the “develop” branch of e.g. *Atlas* would be compatible with the “master” branch of one if its dependencies (e.g. eckit).
When updating the “develop” branch of *Atlas*, it might therefore be advisable to also update the “develop” branches of all of its dependencies. With the following, we can specify which branch in every project will be built.

```
BRANCH=master
```

All ECMWF projects can be built with different optimisation options. There are the following three recommended options:

- `DEBUG`: No optimisation - used for debugging or development purposes only. This option may enable additional boundschecking.
- `BIT`: Maximum optimisation while remaining bit-reproducible.
- `RELEASE`: Maximum optimisation.

With the following, we can specify which optimisation to use for the installation of all projects.

```
BUILD_TYPE=RELEASE
```

We can now proceed to the download and install each of the ECMWF projects required by *Atlas*.

#### 1.2.2.1 ecbuild

To download the project and switch to the correct branch, we can type on the terminal the commands reported below:

```
git clone ssh://git@software.ecmwf.int:7999/ecsdk/ecbuild.git $SRC/ecbuild
cd $SRC/ecbuild
git checkout $BRANCH
```

This project is constituted by just a few CMake macros and it does not need to be compiled nor installed. We do not need to to any additional step for ecbuild!
In the ecbuild project resides an executable script called `ecbuild` to aid installation of all following projects. To make this script easily accessible, prepend it to the `PATH`.

```
export PATH=$SRC/ecbuild/bin:$PATH
```

This executable script `ecbuild` acts as a wrapper around the `cmake` executable. More information on this script can be obtained:

```
ecbuild --help
```

Particular options of the `ecbuild` script noteworthy are `--build` and `--prefix`.

- `--build=$BUILD_TYPE` sets the build type to specified optimisation
- `--install=$INSTALL` sets the install prefix to the specified path

#### 1.2.2.2 eckit

To download the project and switch to the correct branch, we can type on the terminal the commands reported below:

```
git clone ssh://git@software.ecmwf.int:7999/ecsdk/eckit.git $SRC/eckit
cd $SRC/eckit
git checkout $BRANCH
```

Now that we have downloaded the project and switched to the correct branch, we can proceed to build the project and install it. We first need to create the following folder where the files will be built:

```
mkdir $BUILDS/eckit
cd $BUILDS/eckit
```

Then, we need to run ecbuild in order to configure the library - i.e. to find the various dependencies required, etc. - and finally we need to run `make install` to compile and install the library. These two steps are reported below:

```
ecbuild --build=$BUILD_TYPE --prefix=$INSTALL/eckit -- $SRC/eckit
make -j4 install
```

Note that if the folder `$INSTALL/eckit` is not already present it will be automatically created.

#### 1.2.2.3 fckit (optional)

To download the library and switch to the correct branch called `develop`, we can type on the terminal the commands reported below:

```
git clone ssh://git@software.ecmwf.int:7999/ecsdk/fckit.git $SRC/fckit
cd $SRC/fckit
git checkout $BRANCH
```

Now that we have downloaded the library and switched to the develop branch, we can proceed to build the library and install it. We first need to create the following folder where the files will be built:

```
mkdir $BUILDS/fckit
cd $BUILDS/fckit
```

Then, we need to run ecbuild in order to configure the library - i.e. to find the various dependencies required, etc. - and finally we need to run `make install` to compile and install the library. These two steps are reported below:

```
ecbuild --build=$BUILD_TYPE --prefix=$INSTALL/fckit -- $SRC/fckit
make -j4 install
```

Note that if the folder `$INSTALL/fckit` is not already present it will be automatically created.

#### 1.2.2.4 transi (optional)

To download the library and switch to the correct branch called `develop`, we can type on the terminal the commands reported below:

```
git clone ssh://git@software.ecmwf.int:7999/atlas/transi.git $SRC/transi
cd $SRC/transi
```

```
git checkout $BRANCH
```

Now that we have downloaded the library and switched to the develop branch, we can proceed to build the library and install it. We first need to create the following folder where the files will be built:

```
mkdir $BUILDS/transi
cd $BUILDS/transi
```

Then, we need to run ecbuild in order to configure the library - i.e. to find the various dependencies required, etc. - and finally we need to run `make install` to compile and install the library. These two steps are reported below:

```
ecbuild --build=$BUILD_TYPE --prefix=$INSTALL/transi -- $SRC/transi
make -j4 install
```

Note that if the folder `$INSTALL/transi` is not already present it will be automatically created.

### 1.2.3 *Atlas* installation

Once we have downloaded, compiled and installed the third-party dependencies described above, we can now download and install *Atlas*. In particular, to download the library and switch to the correct branch called `develop`, we can type on the terminal the commands reported below:

```
git clone ssh://git@software.ecmwf.int:7999/atlas/atlas.git $SRC/atlas
cd $SRC/atlas
git checkout $BRANCH
```

Now that we have downloaded the library and switched to the develop branch, we can proceed to build the library and install it. We first need to create the following folder where the files will be built:

```
mkdir $BUILDS/atlas
cd $BUILDS/atlas
```

Then, we need to run `ecbuild` in order to configure the library - i.e. to find the various dependencies required, etc. - and finally we need to run `make install` to compile and install the library. These two steps are reported below:

```
$SRC/ecbuild/bin/ecbuild --build=$BUILD_TYPE --prefix=$INSTALL/atlas -- \
```

```
  -DECKIT_PATH=$INSTALL/eckit \
  -DFCKIT_PATH=$INSTALL/fckit \
  -DTRANSI_PATH=$INSTALL/transi \
  $SRC/atlas
make -j4 install
```

Note that if the folder `$INSTALL/atlas` is not already present it will be automatically created.

The following extra flags may be added to the `ecbuild` step to fine-tune configuration:

- `-DENABLE_OMP=OFF` — Disable OpenMP
- `-DENABLE_MPI=OFF` — Disable MPI
- `-DENABLE_FORTRAN=OFF` — Disable Compilation of Fortran bindings
- `-DENABLE_TRANS=OFF` — Disable compilation of the spectral transforms functionality. This is automatically disabled if the optional *transi* dependency is not compiled or found. In this case it is also unnecessary to provide `-DTRANSI_PATH=$INSTALL/transi`.

**Note**

By default compilation is done using shared libraries. Some systems have linking problems with static libraries that have not been compiled with the flag `-fPIC`. In this case, also compile atlas using static linking, by adding to the ecbuild step the flag: `--static`

The building and installation of *Atlas* should now be complete and you can start using it. With this purpose, in the next section we show a simple example on how to create a simple program to initialize and finalize the library.

## 1.3 Inspecting your *Atlas* installation

Once installation of atlas is complete, an executable called "atlas" can be found in `$INSTALL/bin/atlas`. Executing

```
>>> $INSTALL/bin/atlas --version
0.7.0

>>> $INSTALL/bin/atlas --git
```

```
2d683ab4aa0c

>>> $INSTALL/bin/atlas --info
atlas version (0.7.0), git-sha1 2d683ab

  Build:
    build type      : Release
    timestamp       : 20160215122606
    op. system      : Darwin-14.5.0 (macosx.64)
    processor       : x86_64
    c compiler      : Clang 7.0.2.7000181
      flags         :  -O3 -DNDEBUG
    c++ compiler    : Clang 7.0.2.7000181
      flags         :  -O3 -DNDEBUG
    fortran compiler: GNU 5.2.0
      flags         :  -fno-openmp -O3 -funroll-all-loops -finline-functions

  Features:
    Fortran        : ON
    MPI            : ON
    OpenMP         : OFF
    BoundsChecking : OFF
    Trans          : ON
    Tesselation    : ON
    gidx_t         : 64 bit integer

  Dependencies:
    eckit version  (0.12.3), git-sha1 7b76818
    transi version (0.3.2), git-sha1 bf33f60
```

gives you information respectively on the macro version, a more detailed git-version-controlled identifier, and finally a more complete view on all the features that Atlas has been compiled with, as well as compiler and compile flag information. Also printed is the versions of used dependencies such as eckit and transi.

## 1.4 Using *Atlas* in your project

In this section, we provide a simple example on how to use *Atlas*. The objective here is not to get familiar with the main functionalities of *Atlas*, but rather to show how to get started! Specifically, we will show a simple "Hello world" program that initialises and finalises the library, and uses the internal *Atlas* logging facilities to print "Hello world!". The steps necessary to compile and run the program will be detailed in this section.

Note that the *Atlas* supports both C++ and Fortran, therefore, in the following, we will show both an example using C++ and an example using Fortran. Before

starting, we create a folder called `project1` in the `sources` directory:

```
mkdir -p $SRC/project1
```

Here, we will add both the C++ and Fortran files of this simple example. Note that there are (at least) two ways to compile the code we are going to write. The first involves just using a C compiler for the C++ version and a Fortran compiler for the Fortran version, without using any cmake files. The second involves using cmake files. In the following, we will detail both possibilities.

### 1.4.1 C++ version

#### Program description

The C++ version of the *Atlas* initialization and finalization calls is depicted in listing 1.1.

```
1  #include "atlas/atlas.h"
2  #include "atlas/runtime/Log.h"
3
4  int main(int argc, char** argv)
5  {
6      atlas::atlas_init(argc, argv);
7      atlas::Log::info() << "Hello world!" << std::endl;
8      atlas::atlas_finalize();
9
10     return 0;
11 }
```

**Listing 1.1** Initialization and finalization of *Atlas* using C++

We can create a new file in the folder `project1` just generated:

```
touch $SRC/project1/hello-world.cc
```

and copy the content of the code in listing 1.1 into it. We can now have a closer look at the code. On line 1, we include the *Atlas* header file, we successively specify a simple main function, in which we call the initialization of the *Atlas* library on line 6. Note that we passed the two arguments of the main function `argc` and `argv` to the `atlas_init` function. We finally call the *Atlas* `atlas_finalize()` function at line 8 without passing any argument to it.

The function `atlas_init()` is responsible for setting up the logging facility and for the initialization of MPI (Message Passage Interface), while the function `atlas_finalize()` is responsible for finalizing MPI and closing the program. On

line 7, we log "Hello world!" to the `info` log channel.
*Atlas* provides 4 different log channels which can be configured separately: `debug`, `info`, `warning`, and `error`. By default, the `debug` channel does not get printed; the `info` and `warning` channel get printed to the std::cout stream, and the `error` channel gets printed to std::cerr. For more information on the logging facility, the reader is referred to section 2.7.2.

### Code compilation

We now need to compile the code. We first create a new directory into the `$BUILDS` folder, where we will compile the code

```
mkdir -p $BUILDS/project1
```

As mentioned above, there are (at least) two ways for compiling the source code above. These are detailed below.

Directly with C++ compiler

The first possibility is to avoid using cmake and ecbuild and directly run a C++ compiler, such as g++. For doing so, especially when *Atlas* is linked statically, we need to know all *Atlas* dependent libraries. This step can be easily achieved by inspecting the file

```
vi $INSTALL/atlas/lib/pkgconfig/atlas.pc
```

Here, all the flags necessary for the correct compilation of the C++ code in listing 1.1 are reported. For compiling the code, we first go into the builds directory just created and we generate a new folder where the executables will be stored:

```
cd $BUILDS/project1
mkdir -p bin
```

Note that, when using the cmake compilation route, it is not necessary to generate the bin directory since it will automatically created during compilation. After having generated the bin folder, we can run the following command:

```
g++ $SRC/project1/hello-world.cc -o bin/atlas_c-hello-world \
  $(pkg-config $INSTALL/atlas/lib/pkgconfig/atlas.pc --libs --cflags)
```

This will compile our hello-world.cc file and it will automatically link all the static and dynamic libraries required by the program. The executable, as mentioned, is generated into the folder bin.

CMake/ecbuild

The second possibility is to create an ecbuild project, which is especially beneficial for projects with multiple files, libraries, and executables. In particular, we need to create the following cmake file

```
1 # File: CMakeLists.txt
2 cmake_minimum_required(VERSION 2.8.4 FATAL_ERROR)
3 project(usage_example)
4
5 include(ecbuild_system NO_POLICY_SCOPE)
6 ecbuild_requires_macro_version(1.9)
7 ecbuild_declare_project()
8 ecbuild_use_package(PROJECT atlas REQUIRED)
9 ecbuild_add_executable(TARGET   atlas_c-usage_example
10                       SOURCES  hello-world.cc
11                       INCLUDES ${ATLAS_INCLUDE_DIRS}
12                       LIBS     atlas)
13 ecbuild_print_summary()
```

in the sources folder of our project `$SRC/project1`. We can create the `CMakeLists.txt` file in the correct directory following the two steps below:

```
cd $SRC/project1
touch CMakeLists.txt
```

and copy the CMake code above into it. In the second line of the CMake file above, we declare the minimum cmake version required to compile the code, while in the second line we declare the name of our ecbuild project. From line 5 to line 7 we include some required ecbuild macros necessary for using ecbuild. On line 8 we specify that the *Atlas* library is required for this project. Finally, on line 9 we add the instruction to compile the executable. Line 13 prints just a compilation summary. We can build this simple ecbuild project by going into our builds directory

```
cd $BUILDS/project1
```

and by typing the following command:

```
ecbuild -DATLAS_PATH=$INSTALL/atlas $SRC/project1/
make
```

Note that in the above command we needed to provide the path to the *Atlas* library installation. Alternatively, `ATLAS_PATH` may be defined as an environment variable. This completes the compilation of our first example

that uses *Atlas* and generates an executable into the bin folder (automatically generated by cmake) inside our builds directory for project1.

### Run the code

After the compilation of the source code is completed, we have an executable file into the folder `$BUILDS/project1/bin/`. If we simply run the executable file as follows:

```
./atlas_c-hello-world
```

we obtain the output

```
[0] (2016-03-09 T 15:07:15) (I) -- Hello world!
```

However, by adding `--debug` to the command line, also debug information is printed. In particular, if we type:

```
./atlas_c-hello-world --debug
```

we should obtain something similar to the following output:

```
[0] (2016-03-09 T 15:09:42) (D) -- Atlas program [atlas_c-hello-world]
[0] (2016-03-09 T 15:09:42) (D) --   atlas version [0.6.0]
[0] (2016-03-09 T 15:09:42) (D) --   atlas git
  [dabb76e9b696c57fbe7e595b16f292f45547d628]
[0] (2016-03-09 T 15:09:42) (D) --   eckit version [0.11.0]
[0] (2016-03-09 T 15:09:42) (D) --   eckit git
  [ac7f6a0b3cb4f60d9dc01c8d33ed8a44a4c6de27]
[0] (2016-03-09 T 15:09:42) (D) --   Configuration read from scripts:
[0] (2016-03-09 T 15:09:42) (D) --   rundir  :
  /home/na/nagm/myproject/builds/project1
[0] (2016-03-09 T 15:09:42) (I) -- Hello world!
[0] (2016-03-09 T 15:09:42) (D) -- Atlas finalized
```

which gives us some information such as the version of *Atlas* we are running, the identifier of the commit and the path of the executable.

## 1.4.2 Fortran version

### Program description

The Fortran version of the *Atlas* initialization and finalization calls is depicted in listing 1.2.

```fortran
program hello_world

use atlas_module, only : &
  & atlas_init, &
  & atlas_finalize, &
  & atlas_log

call atlas_init()
call atlas_log%info("Hello world!")
call atlas_finalize()

end program hello_world
```

**Listing 1.2** Initialization and finalization of *Atlas* using Fortran

We can create a new file in the folder `project1` just generated:

```
touch $SRC/project1/hello-world.F90
```

and copy the content of the code in listing 1.2 into it. We can now have a closer look at the code. On line 1, we define the program, called `usage_example`. On line 3, we include the required *Atlas* libraries (note that we include only the three functions required for this example - i.e. `atlas_init`, `atlas_finalize`), and `atlas_log`. The function `atlas_init()` on line 8 is responsible for setting up the logging and for the initialization of MPI (Message Passage Interface), while the function `atlas_finalize()` on line 10 is responsible for finalizing MPI and closing the program. On line 9, we log "Hello world!" to the `info` log channel.
*Atlas* provides 4 different log channels which can be configured separately: `debug`, `info`, `warning`, and `error`. By default, the `debug` channel does not get printed; the `info` and `warning` channel get printed to the std::cout stream, and the `error` channel gets printed to std::cerr. For more information on the logging facility, the reader is referred to section 2.7.2.

### Code compilation

We now need to compile the code. We first create a new directory into the `$BUILDS` folder, where we will compile the code

```
mkdir -p $BUILDS/project1
```

As mentioned above, there are (at least) two ways for compiling the source code above. These are detailed below.

### Directly with Fortran compiler

The first possibility is to avoid using cmake and ecbuild and directly run a Fortran compiler, such as gfortran. For doing so, especially when *Atlas* is linked statically, we need to know all *Atlas* dependent libraries. This step can be easily achieved by inspecting the file. This step can be easily achieved by inspecting the file

```
vi $INSTALL/atlas/lib/pkgconfig/atlas.pc
```

Here, all the flags necessary for the correct compilation of the Fortran code in listing 1.2 are reported. For compiling the code, we first go into the builds directory just created and we generate a new folder where the executables will be stored:

```
cd $BUILDS/project1
mkdir -p bin
```

Note that, when using the cmake compilation route, it is not necessary to generate the bin directory since it will automatically created during compilation. After having generated the bin folder, we can run the following command:

```
gfortran $SRC/project1/hello-world.F90 -o bin/atlas_f-hello-world \
$(pkg-config $INSTALL/atlas/lib/pkgconfig/atlas.pc --libs --cflags)
```

This will compile our hello-world.F90 file and it will automatically link all the static and dynamic libraries required by the program. The executable, as mentioned, is generated into the folder bin.

### CMake/ecbuild

The second possibility is to use a cmake file that uses some ecbuild macros. In particular, we need to create the following cmake file:

```
1  # File: CMakeLists.txt
2  cmake_minimum_required(VERSION 2.8.4 FATAL_ERROR)
3  project(usage_example)
4
5  include(ecbuild_system NO_POLICY_SCOPE)
6  ecbuild_requires_macro_version(1.9)
7  ecbuild_declare_project()
8  ecbuild_enable_fortran(MODULE_DIRECTORY ${CMAKE_BINARY_DIR}/module
9                         REQUIRED)
10 ecbuild_use_package(PROJECT atlas REQUIRED)
11 ecbuild_add_executable(TARGET  atlas_f-usage_example
12                        SOURCES hello-world.F90
13                        INCLUDES ${ATLAS_INCLUDE_DIRS}
14                                 ${CMAKE_CURRENT_BINARY_DIR}
15                        LIBS  atlas_f)
16 ecbuild_print_summary()
```

in the sources folder of our project `$SRC/project1`. We can create the `CMakeLists.txt` file in the correct directory following the two steps below:

```
cd $SRC/project1
touch CMakeLists.txt
```

and copy the cmake code above into it. In the second line of the cmake file, we declare the minimum cmake version required to compile the code, while in the second line we declare the name of our cmake project. From line 5 to line 7 we include some required ecbuild macros necessary for using ecbuild. On line 8 we enable Fortran compilation, while on line 10 we specify that the *Atlas* library is required for this project. Finally, on line 11 we add the instruction to compile the executable. Line 15 prints just a compilation summary. We can now run this simple cmake file by going into our builds directory

```
cd $BUILDS/project1
```

and by typing the following command:

```
$SRC/ecbuild/bin/ecbuild -DATLAS_PATH=$INSTALL/atlas $SRC/project1/
make
```

Note that in the above command we needed to provide the path to the *Atlas* library installation. Alternatively, `ATLAS_PATH` may be defined as an environment variable. This completes the compilation of our first example that uses *Atlas* and generates an executable file into the bin folder (automatically generated by CMake) inside our builds directory for project1.

### Run the code

After the compilation of the source code is completed, we have an executable file into the folder `$BUILDS/project1/bin/`. If we simply run the executable file as follows:

```
./atlas_c-hello-world
```

we obtain the output

```
[0] (2016-03-09 T 15:27:00) (I) -- Hello world!
```

However, by setting the environment variable `DEBUG=1`, also debug information is printed. In particular, if we type:

```
export DEBUG=1
./atlas_c-hello-world
```

we should obtain something similar to the following output:

```
[0] (2016-03-09 T 15:27:04) (D) -- Atlas program [atlas_f-hello-world]
[0] (2016-03-09 T 15:27:04) (D) --   atlas version [0.6.0]
[0] (2016-03-09 T 15:27:04) (D) --   atlas git
  [dabb76e9b696c57fbe7e595b16f292f45547d628]
[0] (2016-03-09 T 15:27:04) (D) --   eckit version [0.11.0]
[0] (2016-03-09 T 15:27:04) (D) --   eckit git
   [ac7f6a0b3cb4f60d9dc01c8d33ed8a44a4c6de27]
[0] (2016-03-09 T 15:27:04) (D) --   Configuration read from scripts:
[0] (2016-03-09 T 15:27:04) (D) --   rundir  :
  /home/na/nagm/myproject/builds/project1
[0] (2016-03-09 T 15:27:04) (I) -- Hello world!
[0] (2016-03-09 T 15:27:04) (D) -- Atlas finalized
```

which gives us some information such as the version of *Atlas* we are running, the identifier of the commit and the path of the executable.

**Tip**

The outputs obtained for the Fortran and C++ versions should be identical since they call exactly the same routines.

This completes your first project that uses the *Atlas* library.

Chapter 2

# Design

The flowchart presented in the introduction of this document represents the macro components making up the design of *Atlas* and reflects closely the capabilities of *Atlas*. We subdivide this chapter in various sections, each of them representing one of the components in figure 1. The order in which these components will be presented starts from the grid and closes with the numerics. Note that some simple examples are provided as part of this user-guide in part II.

## 2.1 Grid

The Grid object forms the base class of a hierarchical inheritance tree as shown in figure 2.1. A Grid implementation may be structured or unstructured. The Grid base class interface gives the capability to list the coordinates representing the points of a given grid, and how many points exist in a given grid. It has no knowledge of any domain decomposition or parallelisation in general. The grids that *Atlas* constructs can be *global* or *local*. The *GlobalGrid* type represents a complete spherical grid, whereas the *LocalGrid* type represents a limited area of the sphere. Both global and local grids can be either structured or unstructured. In figure 2.1 we show the various derived classes of the Grid class. Focussing on the Global grid classification, there exist several possible sub classifications, such as the following derived types:

- Unstructured — No structure is present in the grid
- Structured — The grid has a structured distribution of latitudes, and on each latitude a uniform distribution in zonal direction (constant $\Delta\lambda$ for one latitude). No assumption is made on the location of latitudes, or the first longitude value on one latitude.

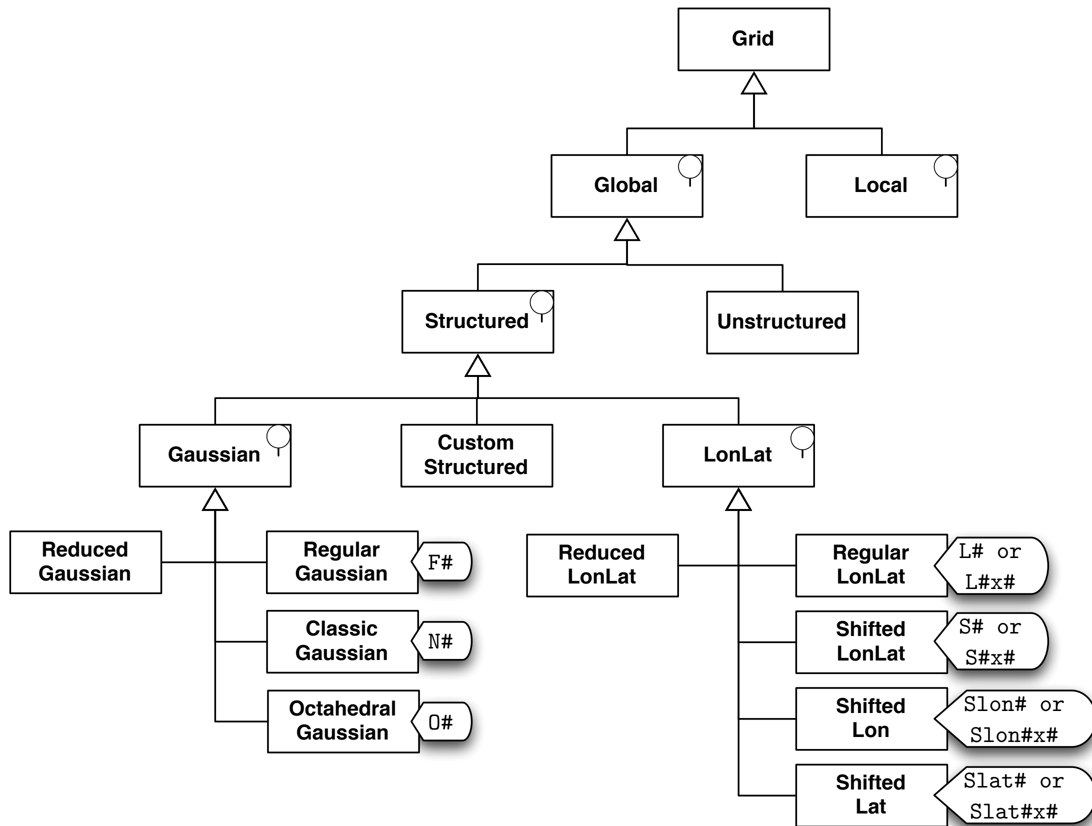



**Figure 2.1** Grid class hierarchy.

The Structured serves as the base class for three additional derived types:

- CustomStructured — To instantiate a customised Structured grid, you have to use the CustomStructured concrete class. Any of the following grids could be expressed as a CustomStructured grid as well.
- Gaussian — The grid's latitude distribution follows the roots of Legendre polynomials. The resolutions of these grids are usually expressed by a number, describing the number of latitudes between a pole and the equator.
- LonLat — The grid's latitude distribution is uniform.

The Gaussian grid, in turn, serves as the base class for

- RegularGaussian — also referred to as *full* Gaussian grid. This Gaussian

grid has a constant number of points on each latitude, equal to 4 $N$, with $N$ the Gaussian number or number of latitudes between pole and equator.

- ClassicGaussian — The number of points on each latitude is computed by optimisations involving the orthogonality of associated Legendre polynomials. This is a costly computations. Hence these tables have been pre-computed for resolutions that have been used in the past at ECMWF.
- OctahedralGaussian — The number of points on each latitude can be inferred from triangulating a regular octahedron, projected to sphere. Certain modifications are required such as modifying the latitude locations to the roots of the Legendre polynomials.
- ReducedGaussian — The number of points on each latitude can be configured by the user.

The LonLat grid class, in turn, serves as the base class for

- RegularLonLat — This grid has a constant number of points on each latitude, and includes typically the pole and the equatorial latitudes, and the Greenwich meridian.
- ShiftedLonLat — This grid has a constant number of points on each latitude, but is shifted half of a longitude increment and half of a latitude increment compared to the RegularLonLat grid. This grid can also be referred to as the dual grid of the RegularLonLat grid. It does not include pole and equatorial latitudes, and not the Greenwich meridian.
- ShiftedLat — This grid has a constant number of points on each latitude, but is shifted half of a latitude increment. It does not include pole and equatorial latitudes, but includes the Greenwich meridian.
- ShiftedLon — This grid has a constant number of points on each latitude, but is shifted half of a longitude increment compared to the RegularLonLat grid. It includes pole and equatorial latitudes, but not the Greenwich meridian.
- ReducedLonLat — The number of points on each latitude can be configured by the user. Typically these grids include the pole latitudes, and the Greenwich meridian.

We note that the object-oriented construction of the Grid object allows one to add any other grid that might be of interest without disrupting the existing grid workflow.

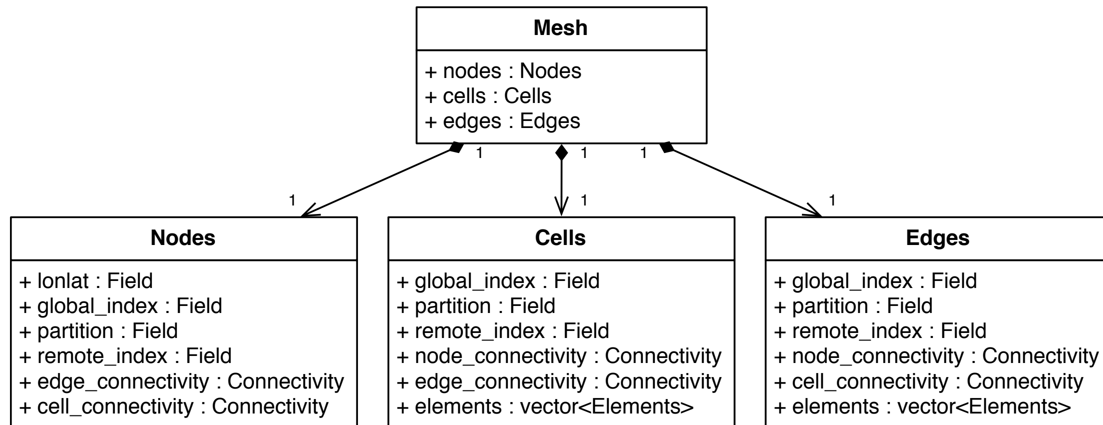
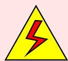



**Figure 2.2** Mesh object composition

## 2.2 Mesh

The *Mesh* object describes how grid points are connected via lines, essentially forming elements such as triangles, quadrilaterals, ... Furthermore, the *Mesh* can be distributed across MPI tasks, called partitions. A *Mesh* object has no inherent notion of structure. Therefore, the nodes and elements in a mesh partition, even if originating from a structured grid could be in any order, and should be treated as such. A *Mesh* is composed of *Nodes*, *Cells*, and *Edges*. These components each contain information stored in *Fields* and *Connectivity* tables. In figure 2.2 we show the composition of a *Mesh* object. Due to the distributed nature of the mesh, three specific fields are required in the *Nodes*, *Cells* and *Edges*, i.e. the `global_index`, `remote_index`, and `partition`. More on parallelisation is presented in section 2.5.



**Warning**

Note that a mesh is not equal to a grid. The grid describes globally the list of points without elements and connectivities, while a mesh is a list of nodes with specific connectivity rules, forming elements, and is distributed among MPI tasks. Note also that, in *Atlas*, the number of points of a grid is generally different from the number of nodes of a mesh.

### 2.2.1 Nodes

The *Nodes* contains a *Field* called `lonlat` which contains the coordinates of the nodes on the sphere in longitude and latitude degrees. Apart from the `lonlat`

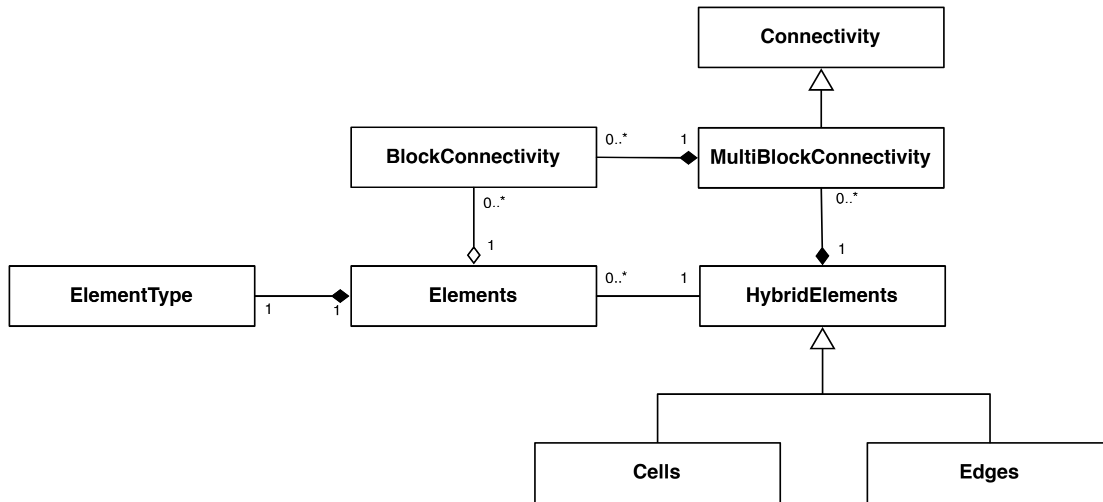



**Figure 2.3** Class diagram related to the mesh *Edges* and *Cells*

field, any *Field* which relates to the mesh nodes can be stored in the *Nodes* object. The *Nodes* therefore can also be seen as a specialised *FieldSet* (see section 2.3). Also stored in the *Nodes* object are connectivity tables relating the mesh nodes to edges and/or elements present in the mesh. These tables are only allocated upon request.

### 2.2.2 Elements and Connectivity

As depicted in figure 2.3, the *Cells* and *Edges* classes derive from a class *HybridElements*. The reason for the naming *hybrid* refers to the possibility of having groups of elements with different element types present under the same umbrella. One group of elements sharing the same *ElementType* is accessible through the *Elements* class. The *HybridElements* class on the other hand offers access to all elements irrespective of their *ElementType*. It could be advantageous for an algorithm to loop over *Elements* corresponding to one *ElementType*, and apply specialised instructions in a nested loop applicable to all elements of that specific *ElementType*.

Because of grouping of elements per *ElementType*, *Connectivity* tables that e.g. relate elements to nodes have the appearance of blocks. To clarify, imagine 2 triangular elements and 3 quadrilateral elements. The table of connectivities represented by the *MultiBlockConnectivity* class would have the form:

```
element 1:    x  x  x
element 2:    x  x  x
element 3:    x  x  x  x
element 4:    x  x  x  x
element 5:    x  x  x  x
```

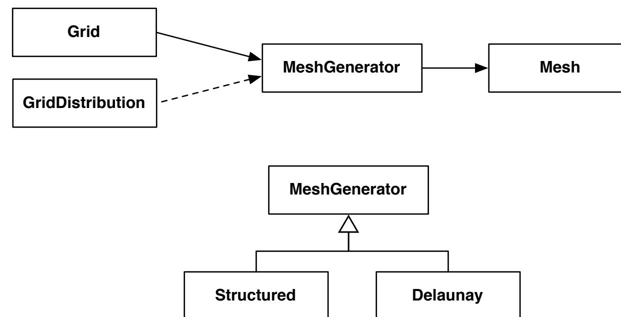



**Figure 2.4** Mesh generation workflow, and *MeshGenerator* class diagram

This *MultiBlockConnectivity* table, can also be interpreted by two *BlockConnectivity* tables:

```
block1, element 1:    x  x  x
block1, element 2:    x  x  x

block2, element 1:    x  x  x  x
block2, element 2:    x  x  x  x
block2, element 3:    x  x  x  x
```

Multiple *BlockConnectivity* tables share exactly the same memory as the one *MultiBlockConnectivity* which is stored in the *HybridElements*. The *Elements* objects, which are also stored in the *HybridElements* and each hold a *BlockConnectivity*, therefore don't occupy any significant memory.

### 2.2.3 Mesh generation

The mesh is constructed using the class *MeshGenerator* which can generate a mesh taking as input a *Grid*, and and optionally a *GridDistribution*. The *GridDistribution* describes for each grid point which MPI task or partition the point belongs to. If the *GridDistribution* is not given, the *MeshGenerator* will internally create a temporary *GridDistribution* internally. More on this can be found in section 2.5. The *MeshGenerator* base class can have several concrete implementations. Depending on the *Grid* type used, some concrete implementations may not be available. For *Structured* grid types, the *mesh::generators::Structured* is available. This mesh generator is very fast as it can take advantage of the structured nature of a *Structured* grid. For any *Grid*, a slower *Delaunay* triangulation is also available. Figure 2.4 shows the usage and class diagram of the *MeshGenerator*.

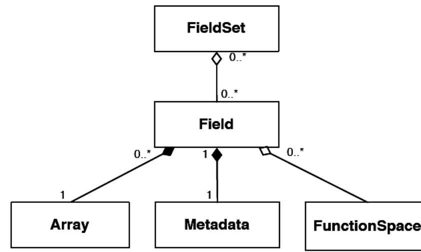



**Figure 2.5** Field and FieldSet object composition

## 2.3 Field and FieldSet

*Field* objects encapsulate given fields, such as the temperature or the pressure, and they can be grouped into FieldSets. The class diagram of the field object is depicted in figure 2.3. In particular the *Field* object is composed of the *Array* object which contains the actual field data, and the *Metadata* object which contains descriptions of the field. A *Field* also contains a reference to a *FunctionSpace* object, described section 2.4.

## 2.4 FunctionSpace

The *FunctionSpace* defines how a *Field* is represented on the domain. There exist a variety of possible function spaces. Examples include *functionspace::Spectral*, *functionspace::StructuredColumns*, *functionspace::NodeColumns*, and *functionspace::EdgeColumns*. These are illustrated in the *FunctionSpace* class inheritance diagram in figure 2.6.

- *functionspace::Spectral* describes the discretisation in terms of spherical harmonics. The parallelisation (gathering and scattering) in spectral space is delegated to a *Trans* object ( in turn delegating to the IFS trans library)

- *functionspace::StructuredColumns* describes the discretisation of distributed fields on a *Structured* grid. At the moment the *Structured* must be a *Gaussian* grid for parallel usage as a *Trans* object is responsible for the parallelisation. In a future release this will be generalised to use a *GatherScatter* object instead, which does not rely on having a *Gaussian* grid.
  *Field*s described using this function space may also be defined to have levels.

- *functionspace::NodeColumns* describes the discretisation of fields with values colocated in the nodes of a *Mesh*, and may have multiple levels defined in a vertical direction. These fields may have parallel halo's defined. A *HaloExchange* object and *GatherScatter* object are responsible for parallelisation.

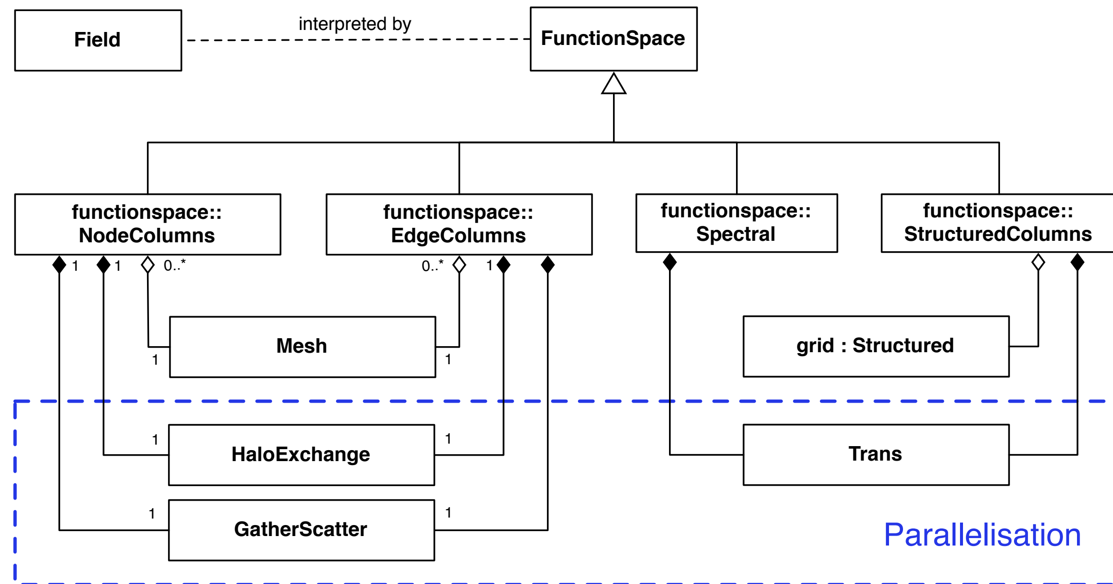



**Figure 2.6** FunctionSpace object diagram

- *functionspace::EdgeColumns* describes the discretisation of fields with values colocated in the edge-centres of a *Mesh*, and may have multiple levels defined in a vertical direction. These fields may have parallel halo's defined. A *HaloExchange* object and *GatherScatter* object are responsible for parallelisation.

## 2.5 Parallelisation

As described section 2.2 and shown in figure 2.2, the *Nodes*, *Cells*, and *Edges* contain three fields related to parallelisation:

- `global_index` — This *Field* contains for each node or element in the mesh partition a unique global index or ID as if the mesh was not distributed. This global index is independent of the number of partitions the mesh is distributed in.

- `partition` — This *Field* contains for each node or element the partition index that has ownership of the node or element. Nodes or elements whose partition does not match the partition index of the mesh partition are also called ghost nodes or ghost elements respectively. These ghost entities merely exist to facilitate stencil operations or to complete e.g. a mesh element.

- `remote_index` — This *Field* contains for each node or element the local index in the mesh partition with index given by the `partition` field.

With the knowledge of `partition` and `remote_index`, we now know for each element or node which partition owns it, and at which location. Usually the user has not to be aware of these three fields as they are more required for *Atlas*' internal parallelisation capabilities.
*Atlas* has two parallel communication pattern classes that can be setup to store how data can be sent and received between MPI tasks.

- The *GatherScatter* class stores the communication pattern of gathering all data to one MPI task, and the inverse, scattering or distributing all data from one MPI task to all MPI tasks.

- The *HaloExchange* class stores the communication pattern of sending and receiving data to nearest neighbour (in domain decomposition sense) MPI tasks, typically required in exchanging small halo's of ghost entities surrounding a domain partition.

## 2.6 Numerics

Apart from serving as a framework for constructing a flexible data structure, *Atlas* also provides some numerical algorithms such as gradient, divergence, curl, and laplacian operators. The gradient, divergence, curl and laplacian operators are bundled in a abstract *Nabla* class, of which a concrete implementation can be instantiated with a *Method* object. This is illustrated in figure 2.7. Here a concrete *fvm::Nabla* is constructed using a concrete *fvm::Method*. These concrete classes implement a median-dual finite volume method. The *fvm::Method* internally uses two *FunctionSpace*s, i.e. a *functionspace::NodeColumns*, and a *functionspace::EdgeColumns*, required to interprete *Field*s residing in nodes and edge-centres. For more information on the median-dual Finite Volume Method, see section 3.1.

## 2.7 Utilities

A number of utilities is also available *Atlas*. These include emphMesh writing, mpi communication, error and exception handling, runtime behaviour, etc.

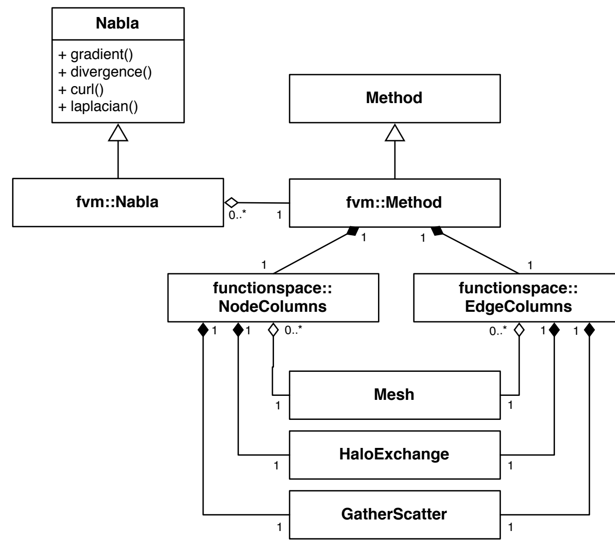



**Figure 2.7** Class diagram for the *fvm::Nabla* operator

### 2.7.1 Configuration

### 2.7.2 Logging

TODO

# Chapter 3

# Theory

## 3.1 fvm: Median-dual Finite Volume Method

TODO

# Part II

# Core functionalities

Chapter **4**

# Create a Global Grid

In this section, we show how to create a global grid with *Atlas*. We remind the reader that global grid refers to a grid covering the entire sphere (e.g. the entire earth for instance). In addition, with the term *grid* we intend a list of points without any specific connectivity rule.

As explained in chapter 2, *Atlas* supports various global grids. Within the library, these grids are identified by some convenient keys. In the rest of the chapter we will divide the various global grids available in the following macro areas:

- Structured grids,
- Unstructured grids.

and we will specify the keys needed to automatically generate a given grid in both C++ and Fortran. Note that it is also possible to create a grid manually, however this approach is not currently explained in this document. It is also important to remark that for the Structured grids, the memory footprint is negligible since the information regarding the points are not stored but are computed at runtime when requested. The last typology, unstructured, has instead a non-negligible memory footprint since in this case the points are effectively stored in memory.

## 4.1 Structured Grids

There exists various subsets of structured grids in *Atlas*. All of them can be constructed using the same method called `Structured::create()`. In particular, we need to pass a predefined key identifying the grid we want to create to the constructor in order to generate the object grid. This object, in turn, contains

various methods that allows the user to retrieve the number of points of the generated grid, the number of longitudes and latitudes, and the coordinates a given longitude and latitude.

The predefined keys existing in *Atlas* for Gaussian grids are reported in the following:

- Classic reduced Gaussian grid — `N#`,
- Regular Gaussian grid — `F#`,
- Octahedral reduced Gaussian grid — `O#`,

where `#` represents the number of latitudes in one hemisphere.

- RegularLonLat — `L#` or `L#×#`
- ShiftedLonLat — `S#` or `S#×#`

where in the alternative key, the first `#` represents the total number of longitudes, while the second `#` specifies the total number of latitudes from pole to pole.

In the next two subsections we will present two simple examples, the first in C++ and the second in Fortran, to generate global grids through the predefined keys just outlined. We will also show how to retrieve some useful information regarding the grid generated.

### 4.1.1 C++ version

The listing 4.1 shows how to construct a generic structured grid.

```
1  #include "atlas/atlas.h"
2  #include "atlas/grid/Structured.h"
3  #include "atlas/runtime/Log.h"
4
5  using atlas::atlas_init;
6  using atlas::atlas_finalize;
7  using atlas::grid::Structured;
8  using atlas::Log;
9
10 int main(int argc, char *argv[])
11 {
12     atlas_init(argc, argv);
13
14     Structured::Ptr grid(Structured::create( "O32" ));
15
16     Log::info() << "nlat   = " << grid->nlat()  << std::endl;
17     Log::info() << "nlon   = " << grid->nlon(0) << std::endl;
18     Log::info() << "npts   = " << grid->npts()  << std::endl;
```

```
19
20      double lonlat[2];
21      grid->lonlat(0, 1, lonlat);
22
23      Log::info() << "lat    = " << grid->lat(0)   << std::endl;
24      Log::info() << "lon    = " << grid->lon(0,1) << std::endl;
25      Log::info() << "lonlat = " << lonlat[0] << "  "
26                                 << lonlat[1] << std::endl;
27
28      atlas_finalize();
29      return 0;
30  }
```

**Listing 4.1** Generating a Gaussian grid with *Atlas* using C++

We create the global grid object (see line 14). In particular, we call the grid constructor passing the command-line string (that represents the key of the grid we want to create) to it. The default value is set to `O32`, that represents a octahedral reduced Gaussian grid with 32 latitudes in one hemisphere (i.e. 64 latitudes in total). The command-line string (i.e. the key) must be contained in the predefined keys for reduced grid generation as specified before. If the user uses a non-existent keyword, there will appear an error message. The grid object, `grid`, is a `Structured` type and it will allow us to access some useful information about the grid.

It is now possible to run this simple program by using one command-line argument representing the keyword that identifies an *Atlas* predefined grid. For instance, we can execute the following command line

```
./atlas_c-global-grids-Structured
```

This will generate an octahedral reduced Gaussian grid with 32 latitudes on one hemisphere (i.e. 64 latitudes in total).

The output to the screen of the code we just executed contains some useful information regarding the grid.

In particular, the first three numbers represent the number of latitudes, longitudes and points of the structured grid, respectively. Note that we can directly access them through the `grid` object (see lines 19, 20, 21).

**Tip**

For the number of latitudes and points we do not need to provide any argument to the function retrieving the information. On the other hand, for the number of longitudes we do need to additionally provide the index of a specific latitude, since the number of longitudes may depend on the latitude (see chapter 2 for more details on how a structured grid is constructed).

The last three numbers represent the coordinates of a given latitude, longitude and both longitude and latitude together, respectively. Note again how these information can be directly accessed through the object `grid`. Also, for retrieving a given latitude coordinate it is only necessary to specify one index that refers to that particular latitude (see line 23). On the other hand, for a given longitude we need to provide both the index of the latitude and the index of the longitude, since the longitude may depend on the latitude (see line 21 and 24).

### 4.1.2 Fortran version

The listing 4.2 shows how to construct a generic structured grid.

```
1  program main
2  use atlas_module
3  implicit none
4  character(len=1024)         :: string
5  type(atlas_grid_Structured) :: grid
6
7  call atlas_init()
8
9  grid = atlas_grid_Structured( "O32" )
10
11 write(string, "(A,I0)") "nlat = ", grid%nlat()
12 call atlas_log%info(string)
13
14 write(string, "(A,I0)") "nlon = ", grid%nlon(1)
15 call atlas_log%info(string)
16
17 write(string, "(A,I0)") "npts = ", grid%npts()
18 call atlas_log%info(string)
19
20 write(string, "(A,F8.4)") "lat(1)    = ", grid%lat(1)
21 call atlas_log%info(string)
22
23 write(string, "(A,F8.4)") "lon(1,1) = ", grid%lon(1, 1)
24 call atlas_log%info(string)
25
26 call atlas_finalize()
27
28 end program main
```

**Listing 4.2** Generating a Gaussian grid with *Atlas* using Fortran

We create the global grid object (see line 9). In particular, we call the grid constructor passing the command-line string (that represents the key of the grid we want to create) to it. The default value is set to `O32`, that represents a octahedral reduced Gaussian grid with 32 latitudes in one hemisphere (i.e. 64 latitudes in total). The grid object, `grid`, is an `atlas_grid_Structured` type and it will allow us to access some useful information about the grid.

It is now possible to run this simple program with the following command line

```
./atlas_f-global-grids-Structured
```

This will generate an octahedral reduced Gaussian grid with 32 latitudes on one emisphere (i.e. 64 latitudes in total).

The output to the screen of the code we just executed contains some useful information regarding the grid.

In particular, the first three numbers represent the number of latitudes, longitudes and points of the grid, respectively. Note that we can directly access them through the `grid` object (see lines 11, 14, 17).

**Tip**

For the number of latitudes and points we do not need to provide any argument to the function retrieving the information. On the other hand, for the number of longitudes we do need to additionally provide the index of a specific latitude, since the number of longitudes may depend on the latitude (see chapter 2 for more details on how a structured grid is constructed).

The last two numbers represent the coordinates of a given latitude and longitude, respectively. Note again how these information can be directly accessed through the object `grid`. Also, for retrieving a given latitude coordinate it is only necessary to specify one index that refers to that particular latitude (see line 20). On the other hand, for a given longitude we need to provide both the index of the latitude and the index of the longitude, since the longitude may depend on the latitude (see line 23).

You can now play with line 9 to generate different types of global structured grid using the keys introduced at the beginning of this section!

## 4.2 Unstructured Grids

Coming soon ...

Chapter 5

# Create a Mesh from a Grid

In this chapter, we show how to create a mesh with *Atlas* starting from a grid. We assume that the reader already knows how to create a grid (see chapter 4).

**Note**
With the term *grid* we intend a list of points without any specific connectivity rule, while with the term mesh we intend a list of points with well-specified connectivity rules.

As before, we show both the C++ and Fortran version for this example.

## 5.1 C++ version

The listing 5.1 shows how to construct a mesh starting from a grid.

```
1  #include "atlas/atlas.h"
2  #include "atlas/grid/Grid.h"
3  #include "atlas/mesh/Mesh.h"
4  #include "atlas/mesh/generators/Structured.h"
5  #include "atlas/output/Gmsh.h"
6  #include "atlas/util/Config.h"
7
8  using atlas::atlas_init;
9  using atlas::atlas_finalize;
10 using atlas::grid::Grid;
11 using atlas::mesh::Mesh;
12 using atlas::output::Gmsh;
13 using atlas::util::Config;
14
15 int main(int argc, char *argv[])
16 {
17     atlas_init(argc, argv);
18
```

```
19     atlas::mesh::generators::Structured meshgenerator;
20
21     Grid::Ptr grid( Grid::create( "O32" ) );
22     Mesh::Ptr mesh( meshgenerator.generate(*grid) );
23
24     Gmsh gmsh_2d("mesh2d.msh");
25     Gmsh gmsh_3d("mesh3d.msh", Config("coordinates", "xyz") );
26
27     gmsh_2d.write(*mesh);
28     gmsh_3d.write(*mesh);
29
30     atlas_finalize();
31
32     return 0;
33 }
```

**Listing 5.1** Generating a mesh starting from a grid in *Atlas* using C++

Once defined the command-line behaviour, we first create a global structured grid object (see line 24). For more details on how to create a grid see chapter 4.

We then create a `mesh::generators::Structured` object called `generate_mesh` that will allow us to generate the mesh starting from a structured grid. We successively create a mesh object, `mesh` of the `Mesh` type on line 27 using the mesh generator.

In this simple example we took the freedom to add a few lines to show how to visualize the mesh in Gmsh. In particular, on line 29 we define a Gmsh object called `gmsh` that will be used to generate a Gmsh output. On line 30 we specify that we want to have some information regarding the mesh - this is achieved by defining as `true` the tag `"info"`. Between line 31 and 35, we add two additional lines to visualize the mesh in 3D if required on the command-line. Finally, on line 36 we write the mesh and save it into `mesh.msh`. Note that the file containing the information on the mesh just created is called `mesh_info.msh`.

It is now possible to run this simple program by using two command-line arguments representing the keyword that identifies an *Atlas* predefined grid and the visualization type we want (either 2D or 3D), respectively. For instance, we can execute the following command line

```
./atlas_c-meshes-Structured
```

This will produce an octahedral reduced Gaussian mesh (stored in mesh.msh) with 32 latitudes on one hemisphere (i.e. 64 latitudes in total). It will also produce an additional file, called `mesh_info.msh`, containing some information regarding the

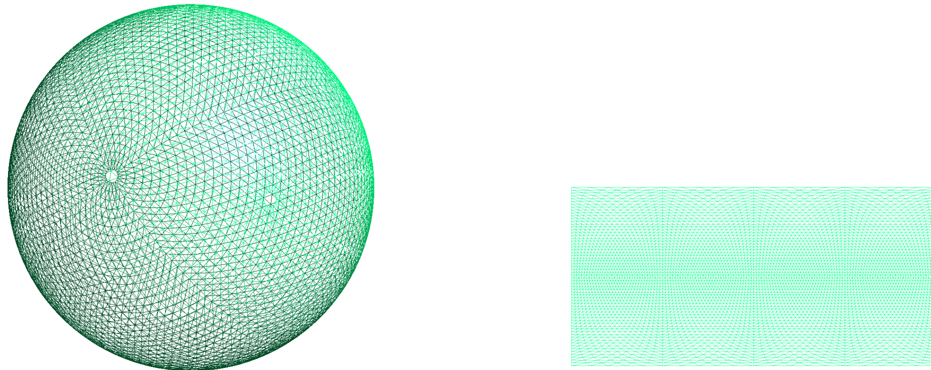


**Figure 5.1** Meshes visualised in Gmsh

mesh. Note that we used the additional command-line argument `-visualize 3D`. This will produce a 3D representation of the mesh, such as the one depicted in left side of figure 5.1.

We can re-run the executable file in order to obtain a 2D representation as follows:

```
./atlas_c-meshes-Structured --grid O32 --visualize 2D
```

This will produce a representation of the mesh like the one depicted on the right side of figure 5.1. You can now play with the command-line argument to generate different types of global structured meshes using the keys introduced in chapter 4!

## 5.2 Fortran version

The listing 5.1 shows how to construct a mesh starting from a grid.

```
1  program main
2  use atlas_module
3  implicit none
4  type(atlas_Grid)           :: grid
5  type(atlas_Mesh)           :: mesh
6  type(atlas_MeshGenerator) :: meshgenerator
7  type(atlas_Output)         :: gmsh_2d, gmsh_3d
8
9  call atlas_init()
10
11 ! Generate mesh
12 meshgenerator = atlas_meshgenerator_Structured()
13 grid = atlas_grid_Structured( "O32" )
14 mesh = meshgenerator%generate(grid)
15
16 gmsh_2d = atlas_output_Gmsh("mesh2d.msh")
17 gmsh_3d = atlas_output_Gmsh("mesh3d.msh",coordinates="xyz")
18
```

```
19 ! Write mesh
20 call gmsh_2d%write(mesh)
21 call gmsh_3d%write(mesh)
22
23 ! Cleanup
24 call grid%final()
25 call mesh%final()
26 call gmsh_2d%final()
27 call gmsh_3d%final()
28 call meshgenerator%final()
29
30 call atlas_finalize()
31
32 end program main
```

**Listing 5.2** Generating a mesh starting from a grid in *Atlas* using Fortran

We first create a global structured grid object (see line 13). For more details on how to create a grid see chapter 4.

We successively create the mesh object on line 15 and 16. In particular, we first define a `atlas_Meshgenerator` object that is then used to effectively generate the mesh object `mesh` that is an `atlas_Mesh` type.

In this simple example we took the freedom to add just one line line to visualize the mesh in Gmsh. In particular, on line 17 we call `atlas_write_gmsh` to write a Gmsh file called `mesh.msh`. Note that at the end of the program we also need to destroy the local object created in this program (see lines 19 to 21).

It is now possible to run this simple program by using a command-line arguments representing the keyword that identifies an *Atlas* predefined grid. For instance, we can execute the following command line

```
./atlas_c-meshes-Structured
```

This will produce an octahedral reduced Gaussian mesh (stored in mesh.msh) with 32 latitudes on one hemisphere (i.e. 64 latitudes in total). If we visualize it in Gmsh, we will obtain something similar to figure 5.2. You can now play with the command-line argument to generate different types of meshes for global grids using the keys introduced in chapter 4!

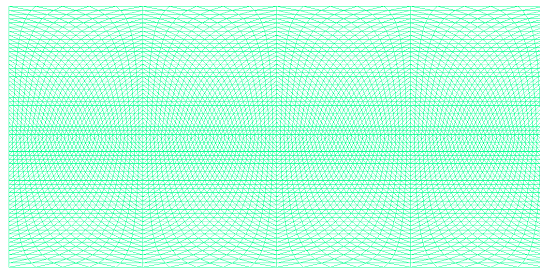


**Figure 5.2** Mesh visualised in Gmsh

Chapter 6

# Create Fields and Field Sets

In this chapter, we show how to create fields and field sets using *Atlas*. Specifically, we outline how to create two simple fields and how to include them into a field set. These two fields are standalone, thus not related to any grid - i.e. they are just defined as generic multidimensional arrays containing some values and a short description of what is stored inside them. Successively, we introduce how to create two fields on a given grid and again how to add them to a field set. As done for the other examples, we show both the C++ and Fortran versions.

## 6.1 Standalone Fields and Field Sets

### 6.1.1 C++ version

The listing 6.1 shows how to construct two standalone fields and encapsulate them into a field set.

```
1  #include "atlas/atlas.h"
2  #include "atlas/runtime/Log.h"
3  #include "atlas/field/Field.h"
4  #include "atlas/field/FieldSet.h"
5  #include "atlas/util/Metadata.h"
6
7  using atlas::atlas_init;
8  using atlas::atlas_finalize;
9  using atlas::Log;
10 using atlas::field::Field;
11 using atlas::field::FieldSet;
12 using atlas::array::ArrayView;
13 using atlas::array::make_shape;
14
15 int main(int argc, char *argv[])
16 {
17     atlas_init(argc, argv);
18
19     // Define fields
```

```
20     Field::Ptr field_pressure(
21       Field::create<double>("pressure", make_shape(100)) );
22     Field::Ptr field_wind(
23       Field::create<double>("wind", make_shape(100, 2)) );
24
25     // Access field data
26     ArrayView <double,1> pressure(*field_pressure);
27     ArrayView <double,2> wind     (*field_wind);
28
29     // Assign values to fields
30     for (size_t jnode = 0; jnode < 100; ++jnode)
31     {
32         pressure(jnode) = 101325.0;
33         wind(jnode,0)   = 0.01 + double(jnode);
34         wind(jnode,1)   = 0.02 + double(jnode);
35     }
36
37     // Add info to fields
38     std::string unitsP, unitsW;
39     field_pressure->metadata().set("units", "[Pa]");
40     field_pressure->metadata().get("units", unitsP);
41     field_wind    ->metadata().set("units", "[m/s]");
42     field_wind    ->metadata().get("units", unitsW);
43
44     // Define fieldSet
45     FieldSet fields;
46     fields.add(*field_pressure); // Add field_pressure to fieldSet
47     fields.add(*field_wind);     // Add field_wind to fieldSet
48
49     // Retrieve field from fieldSet
50     Field& field_pressure2 = fields.field("pressure");
51     Field& field_wind2     = fields.field("wind");
52
53     // Print some useful info
54     Log::info() << "name   = " << field_wind->name()     << std::endl;
55     Log::info() << "size   = " << field_wind->size()     << std::endl;
56     Log::info() << "units  = " << unitsW                 << std::endl;
57     Log::info() << "rank   = " << field_wind->rank()     << std::endl;
58     Log::info() << "shape  = " << field_wind->shape(0)   << "    "
59                                << field_wind->shape(1)   << std::endl;
60     Log::info() << "memory = " << field_wind->bytes()
61                                << " bytes"              << std::endl;
62     Log::info() << "type   = " << field_wind->datatype().str()  << std::
    endl;
63     Log::info() << "kind   = " << field_wind->datatype().kind() << std::
    endl;
64
65     // Print some values
66     Log::info() << "pressure(9) = " << pressure(9) << std::endl;
67     Log::info() << "wind(9, 0)  = " << wind(9,0)   << std::endl;
68     Log::info() << "wind(9, 1)  = " << wind(9,1)   << std::endl;
69
70     atlas_finalize();
71
72     return 0;
73 }
```

**Listing 6.1** Generating two fields and encapsulating them into a FieldSet using C++

On the first few lines of the code, we include the necessary *Atlas* header files needed for this example. Note in particular the inclusion of `Field.h`, `FieldSet.h` and `Metadata.h`. The first is necessary to define fields, the second to define field set and the third to add a description to the fields.

We then define two fields, one called `field_pressure` that, for instance, will contain the pressure, and the other one called `field_wind` that will for example contain the velocity of the wind in two orthogonal directions.

How does the creation of a field work?
On lines 18 and 20, we can see the construction of the two fields. We first need to declare a pointer of type `Field` and we successively call the constructor for this field. This is composed by three elements: the type of data contained within the field (in our case double), the name of the field as a string (in our case 'pressure') and its dimensions. Note that we allow multidimensional fields up to 6 dimensions (this number can be extended if required)!

**Tip**

In a field we can only store numbers - no strings or characters! In particular, we support 32 and 64 bit integer and real types.

Once the fields are defined we need to initialize them and give them some values. This task can be achieved by using the code on lines 24 and 25, where we acquire access to the two fields using `ArrayView` objects, `pressure` and `wind`. We successively prescribe some values to these two objects (see lines 28 to 33). This step automatically updates what is stored in the two field objects, `field_pressure` and `field_wind`.

The work for defining the two fields is almost completed. We can add just one more little feature - one or more descriptors. This task is performed on lines 37 to 40, where we use the `metadata` object to set the units of our fields and retrieve them through the functions, `set` and `get`, respectively.

These two fields are fully functional and can be used for our specific application. However, we may want to encapsulate several fields into one object. This task can be achieved by using the object `FieldSet`, as show on lines 43 to 45, where we define the field set and we add the two fields into it.

Any field can also be retrieved from a *FieldSet* by using the code on lines 48 and 49, where we ask for the field 'pressure' and the field 'wind' to be retrieved by two new empty fields.

**Note**

It is possible to retrieve a field from a *FieldSet* either by using the name of the field or by using the number identifying it. In our example, `field_pressure` assumes id=0 (since stored first), while `field_wind` assumes id=1 (since stored second).

After having defined the field set, we print some useful information regarding the fields (for the sake of brevity we print just some information regarding the `field_wind` - the information regarding the `field_pressure` can be obtained in an identical way). In particular, we print the name, the size and the `metadata` associated to the `field_wind` (see lines 52 to 54). We then extract its rank, shape and dimensions in bytes (see lines 55 to 58). We finally print type of the data stored in the field (see lines 60 and 61).

On lines 64 to 66 we also print the values of one element per each field. Note that the memory of the objects defined in this example is automatically released when the execution ends. So, there is no need to manually destroy the objects. This aspect is different in the Fortran example below, where we will need to explicitly finalise all the objects created!

It is now possible to run this simple program typing the following text on the terminal

```
./atlas_c-fields
```

This will produce the two fields described and a field set and will destroy them at the end of the routine, thus automatically releasing the memory. It will also print to the screen some useful information regarding the fields - specifically you should obtain a screen output similar to the one below:

```
[0] (2016-03-15 T 15:57:33) (I) -- name   = wind
[0] (2016-03-15 T 15:57:33) (I) -- size   = 200
[0] (2016-03-15 T 15:57:33) (I) -- units  = [m/s]
[0] (2016-03-15 T 15:57:33) (I) -- rank   = 2
[0] (2016-03-15 T 15:57:33) (I) -- shape  = 100    2
[0] (2016-03-15 T 15:57:33) (I) -- memory = 1600 bytes
[0] (2016-03-15 T 15:57:33) (I) -- type   = real64
[0] (2016-03-15 T 15:57:33) (I) -- kind   = 8
[0] (2016-03-15 T 15:57:33) (I) -- pressure(9) = 101325
[0] (2016-03-15 T 15:57:33) (I) -- wind(9, 0)  = 9.01
[0] (2016-03-15 T 15:57:33) (I) -- wind(9, 1)  = 9.02
```

You can now play with the code in listing 6.1 to generate as many fields/field sets as you want!

## 6.2 Fortran version

The listing 6.2 shows how to construct two standalone fields and encapsulate them into a field set.

```
1  program main
2
3  use, intrinsic :: iso_c_binding, only : c_double
4  use atlas_module
5  implicit none
6
7  integer, parameter                :: wp = c_double
8  integer                           :: jnode
9  character(len=1024)               :: string
10 character(len=:), allocatable     :: unitsW, unitsP
11 type(atlas_Field)                 :: field_pressure , field_wind
12 type(atlas_Field)                 :: field_pressure2, field_wind2
13 real(wp), pointer                 :: pressure(:), wind    (:,:)
14 type(atlas_FieldSet)              :: fields
15 type(atlas_Metadata)              :: metadata
16 call atlas_init()
17
18 ! Define fields
19 field_pressure = atlas_Field(name="pressure", kind=atlas_real(wp), shape
     =[100])
20 field_wind     = atlas_Field(name="wind"    , kind=atlas_real(wp), shape
     =[2, 100])
21
22 ! Access fields data
23 call field_pressure%data(pressure)
24 call field_wind    %data(wind)
25
26 ! Assign values to fields
27 do jnode=1,100
28   pressure(jnode) = 101325._wp
29   wind(1, jnode)  = 0.01_wp + real(jnode,kind=wp)
30   wind(2, jnode)  = 0.02_wp + real(jnode,kind=wp)
31 enddo
32
33 ! Add info to fields
34 metadata = field_pressure%metadata()
35 call metadata%set("units", "[Pa]")
36 call metadata%get("units", unitsP)
37 metadata = field_wind%metadata()
38 call metadata%set("units", "[m/s]")
39 call metadata%get("units", unitsW)
40
41 ! Define fieldSet
42 fields = atlas_FieldSet()
43 call fields%add(field_pressure) ! Add field_pressure to fieldSet
44 call fields%add(field_wind)     ! Add field_wind to fieldSet
45
46 ! Retrieve field from fieldSet
47 field_pressure2 = fields%field("pressure")
```

```
48 field_wind2      = fields%field("wind")
49
50 ! Print some useful info
51 write(string, *) "name    = ", field_wind%name()
52 call atlas_log%info(string)
53 write(string, *) "size    = ", field_wind%size()
54 call atlas_log%info(string)
55 write(string, *) "units   = ", unitsW
56 call atlas_log%info(string)
57 write(string, *) "rank    = ", field_wind%rank()
58 call atlas_log%info(string)
59 write(string, *) "shape(1)   = ", field_wind%shape(1)
60 call atlas_log%info(string)
61 write(string, *) "shape(2)   = ", field_wind%shape(2)
62 call atlas_log%info(string)
63 write(string, *) "shape   = ", field_wind%shape()
64 call atlas_log%info(string)
65 write(string, *) "memory = ", field_wind%bytes(), "bytes"
66 call atlas_log%info(string)
67 write(string, *) "type    = ", field_wind%datatype()
68 call atlas_log%info(string)
69 write(string, *) "kind    = ", field_wind%kind()
70 call atlas_log%info(string)
71
72 ! Print some values
73 write(string, *) "pressure(10) = ", pressure(10)
74 call atlas_log%info(string)
75 write(string, *) "wind(1, 10)   = ", wind(1,10)
76 call atlas_log%info(string)
77 write(string, *) "wind(2, 10)   = ", wind(2,10)
78 call atlas_log%info(string)
79
80 ! Finalize object to release memory
81 call field_pressure%final()
82 call field_wind    %final()
83 call fields        %final()
84
85 call atlas_finalize()
86 end program main
```

**Listing 6.2** Generating two fields and encapsulating them into a FieldSet using Fortran

On the first few lines of the code, we define the variables needed for this program. In particular, the *Atlas* specific variables needed for this example are the `atlas_Field`, `atlas_FieldSet` and `atlas_Metadata` objects.

After having defined all the data needed for this example, we initialize the *Atlas* library as usual and we define two fields, one called `field_pressure` that, for instance, will contain the pressure and the other one called `field_wind` that will for example contain the velocity of the wind in two orthogonal directions.

How does the creation of a field work?
On lines 17 and 18 we can see the construction of the two fields. We first need to specify the name of the field (in our case 'pressure' and 'wind'), then we need to specify the type of the data stored into the fields (in our case double precision numbers) and finally we need to provide the dimension of the field. Note that we allow multidimensional fields up to 6 dimensions (this number can be extended if required)!

**Tip**

In a field we can only store numbers - no strings or characters! In particular, we support integers, float types and double types.

Once the fields are defined we need to access the data and give them some values. This task can be achieved by using the code on lines 23 and 24, where we access the data of the two fields by two pointers `pressure` and `wind`, respectively. We successively prescribe some values to these two pointers (see lines 27 to 31). This step automatically updates what is stored in the two field objects, `field_pressure` and `field_wind`.

The work for defining the two fields is almost completed. We can add just one more little feature - one or more descriptors. This task is performed on lines 34 to 39, where we use the `metadata` object to set the units of our fields and retrieve them through the functions, `set` and `get`, respectively.

These two fields are fully functional and can be used for our specific application. However, we may want to encapsulate several fields into one object. This task can be achieved by using the object `atlas_FieldSet`, as show on lines 40 to 42, where we define the field set and we add the two fields into it.

Any field can also be retrieved from a field set by using the code on lines 45 and 46, where we ask for the field 'pressure' and the field 'wind' to be retrieved by two new `atlas_Field` objects.

**Note**

It is possible to retrieve a *Field* from a *FieldSet* either by using the name of the field or by using the number identifying it. In our example, `field_pressure` assumes id=1 (since stored first), while `field_wind` assumes id=2 (since stored second).

After having defined the field set, we print some useful information regarding the fields (for the sake of brevity we print just some information regarding the `field_wind` - the information regarding the `field_pressure` can be obtained in an identical way). In particular, we print the name, the size and the `metadata` associated to the `field_wind` (see lines 51 to 55). We then extract its rank, shape and dimensions in bytes (see lines 57 to 66). We finally print type of the data stored in the field (see lines 67 and 70).

On lines 73 to 78 we also print the values of one element per each field and we finalise the field objects on lines 81 and 82 (thus releasing the memory).

Note that finalising the `atlas_Field` objects is enough to also finalise `atlas_FieldSet` object; we need to explicitly finalise it as well to completely free the memory associated to all the objects defined in this example (see line 83).

It is now possible to run this simple program typing the following text on the terminal

```
./atlas_f-fields
```

This will produce the two fields described and a field set and will destroy them at the end of the routine (thus releasing the memory). It will also print to the screen some useful information regarding the fields. In particular, you should obtain an output similar to the one below:

```
[0] (2016-03-15 T 17:16:01) (I) -- name   = wind
[0] (2016-03-15 T 17:16:01) (I) -- size   = 200
[0] (2016-03-15 T 17:16:01) (I) -- units  = [m/s]
[0] (2016-03-15 T 17:16:01) (I) -- rank   = 2
[0] (2016-03-15 T 17:16:01) (I) -- shape(1)  = 2
[0] (2016-03-15 T 17:16:01) (I) -- shape(2)  = 100
[0] (2016-03-15 T 17:16:01) (I) -- shape  = 2         100
[0] (2016-03-15 T 17:16:01) (I) -- memory = 1600.0000 bytes
[0] (2016-03-15 T 17:16:01) (I) -- type   = real64
[0] (2016-03-15 T 17:16:01) (I) -- kind   = 8
[0] (2016-03-15 T 17:16:01) (I) -- pressure(10) = 101325.0000
[0] (2016-03-15 T 17:16:01) (I) -- wind(1, 10)  = 10.01000000
[0] (2016-03-15 T 17:16:01) (I) -- wind(2, 10)  = 10.02000000
```

You can now play with the code in listing 6.2 to generate as many fields/field sets as you want!

## 6.3 Fields on a given Grid

### 6.3.1 C++ version

The listing 6.3 shows how to construct one field on a given grid. To see how to create a generic field and a field set and how to use some additional functionalities related to fields, please refer to section 6.1 above.

```
1  #include "atlas/atlas.h"
2  #include "atlas/runtime/Log.h"
3  #include "atlas/grid/grids.h"
4  #include "atlas/field/Field.h"
5
6  using atlas::atlas_init;
7  using atlas::atlas_finalize;
8  using atlas::Log;
9  using atlas::array::ArrayView;
10 using atlas::array::make_shape;
11 using atlas::field::Field;
12 using atlas::grid::Structured;
13
14 int main(int argc, char *argv[])
15 {
16     atlas_init(argc, argv);
17
18     int jnode = 0;
19     const double rpi = 2.0 * asin(1.0);
20     const double deg2rad = rpi / 180.;
21     const double zlatc = 0.0 * rpi;
22     const double zlonc = 1.0 * rpi;
23     const double zrad  = 2.0 * rpi / 9.0;
24     double  zdist, zlon, zlat;
25
26     Structured::Ptr grid( Structured::create( "N32" ) );
27     const size_t nb_nodes = grid->npts();
28
29     Field::Ptr field_pressure(
30       Field::create<double>("pressure", make_shape(nb_nodes)));
31
32     ArrayView <double,1> pressure(*field_pressure);
33     for (size_t jlat =0; jlat < grid->nlat(); ++jlat)
34     {
35         zlat = grid->lat(jlat);
36         zlat = zlat * deg2rad;
37         for (size_t jlon =0; jlon < grid->nlon(jlat); ++jlon)
38         {
39             zlon = grid->lon(jlat, jlon);
40             zlon = zlon * deg2rad;
41             zdist = 2.0 * sqrt((cos(zlat) * sin((zlon-zlonc)/2)) *
42                               (cos(zlat) * sin((zlon-zlonc)/2)) +
43                               sin((zlat-zlatc)/2) * sin((zlat-zlatc)/2));
44
45             pressure(jnode) = 0.0;
46             if (zdist < zrad)
47             {
48                 pressure(jnode) = 0.5 * (1. + cos(rpi*zdist/zrad));
49             }
50             jnode = jnode+1;
51         }
```

```
52     }
53
54     Log::info() << "=========================================" << std::
       endl;
55     Log::info() << "memory field_pressure = "
56                 << field_pressure->bytes() * 1.e-9 << " GB"      << std::
       endl;
57     Log::info() << "=========================================" << std::
       endl;
58
59     atlas_finalize();
60
61     return 0;
62 }
```

**Listing 6.3** Generating a field on a given grid using C++

On the first few lines of the code, we include the necessary header files for this example. In particular, we include `grids.h`, `Field.h`. We then initialize the *Atlas* library and define some constants needed to define the function we are going to implement later in the code.

We then create a `grid` object and a `field` object. Note that we used a command-line argument to decide what grid to use (see chapter 4 for more details on how to create global grids).

On line 24, we define the grid using a command-line key that can be specified by the user (see chapter 4 for more details on how to create global grids). On lines 27 and 28, we define the pressure field, while, on line, 30 we initialize the associated `ArrayView` object, needed to manipulate and access the data inside the `Field` object. From line 31 to line 50, we specify the a Gaussian-type (e.g. a hill) function on our grid (specifically, the field is defined between line 43 and 47). We finally close the program outputting on the screen the memory footprint of the field just created. Note that we do not need to free the memory of the grid and field objects, since it is automatically released at the end of the execution (in contrast to Fortran, where we explicitly need to destroy the objects created).

It is now possible to run this simple program typing the following text on the terminal

```
./atlas_c-fields-on-grid
```

This will produce a field (called pressure) defined on an octahedral grid that has the shape of a hill or Gaussian-type function. The output on the screen should be the memory footprint of the field created on the grid and it should be similar to

the one below:

```
========================================
memory field_pressure = 0.0338493 GB
========================================
```

Not a big deal for this grid!

You can now play with the command-line argument and generate different grids and see the impact on the memory footprint of the pressure field.

### 6.3.2 Fortran version

The listing 6.4 shows how to construct one field on a given grid. To see how to create a generic field and a field set and how to use some additional functionalities related to fields, please refer to section 6.1 above.

```
1  program main
2
3  use, intrinsic :: iso_c_binding, only : c_double
4  use atlas_module
5
6  integer, parameter                :: wp = c_double
7  real(wp), parameter               :: rpi = 2.0_wp * asin(1.0_wp)
8  real(wp), parameter               :: deg2rad = rpi / 180.
9  real(wp)                          :: zlatc = 0._wp * rpi
10 real(wp)                          :: zlonc = 1._wp * rpi
11 real(wp)                          :: zrad  = 2._wp * rpi / 9._wp
12 real(wp)                          :: zdist, zlon, zlat
13
14 integer                           :: jnode
15 character(len=1024)               :: string
16 character(len=32)                 :: gridID
17 type(atlas_grid_Structured)       :: grid
18 type(atlas_Field)                 :: field_pressure
19 real(wp), pointer                 :: pressure(:)
20
21 call atlas_init()
22
23 gridID = "N32"
24 grid = atlas_grid_Structured(gridID)
25
26 field_pressure = atlas_Field("pressure", atlas_real(wp), [grid%npts()])
27 call field_pressure%data(pressure)
28
29 jnode = 1
30 do jlat=1,grid%nlat()
31   zlat = grid%lat(jlat)
32   zlat = zlat * deg2rad
33   do jlon=1,grid%nlon(jlat)
34     zlon = grid%lon(jlat,jlon)
35     zlon = zlon * deg2rad
36
37     zdist = 2._wp * sqrt((cos(zlat) * sin((zlon - zlonc) / 2))**2 + &
38                        & sin((zlat - zlatc) / 2)**2)
```

```fortran
39
40      pressure(jnode) = 0._wp
41      if (zdist < zrad) then
42        pressure(jnode) = 0.5_wp * (1._wp + cos(rpi * zdist / zrad))
43      endif
44      jnode = jnode + 1
45    enddo
46  enddo
47
48  write(string, *) "==================================================="
49  call atlas_log%info(string)
50  write(string, *) "memory field_pressure = ", &
51              & field_pressure%bytes()/1000000000., "GB"
52  call atlas_log%info(string)
53  write(string, *) "==================================================="
54  call atlas_log%info(string)
55
56  call grid%final()
57  call field_pressure%final()
58
59  call atlas_finalize()
60
61  end program main
```

**Listing 6.4** Generating a field on a given grid using Fortran

On the first few lines of the code, we define the variables needed for this program. In particular, we define some constants needed for the function we are going to implement later in the code and we declare an `atlas_grid_Structured` grid object and an `atlas_Field` object.

On lines 21 and 22 we define the grid using a command-line key that can be specified by the user (see chapter 4), while on lines 24 and 25, we initialize the pressure field. From line 28 to line 45, we specify the a Gaussian-type (e.g. a hill) function on our grid (specifically, the field is defined on lines 39 to 42). We finally close the program outputting on the screen the memory footprint of the field just created. As usual, we also explicitly need to free the memory calling the function `final` on the grid and field objects.

It is now possible to run this simple program typing the following text on the terminal

```
./atlas_f-fields-on-grid
```

This will produce a field (called pressure) defined on an octahedral grid that has the shape of a hill or Gaussian-type function. The output on the screen should be the memory footprint of the field created on the grid and it should be similar to

the one below:

```
=================================================
memory field_pressure = 3.3849344000000003E-002 GB
=================================================
```

Not a big deal for this grid!

You can now play with the command-line argument and generate different grids and see the impact on the memory footprint of the pressure field.

Chapter 7

# Using the function space objects

In this chapter, we show how to use the function space objects. These objects are intended to interpret a given field. In particular, by using them on a given field, we equip the field with the communication pattern (thus the Field knows about parallelism), and it allows some simple operations on the field. The function space objects that will be presented in this chapter are the following three:

- `NodeColumns`: It relates a given field to the underlying mesh, thus enabling the field to parallel communication. It also allows some simple operations on the Field, such as calculating minimum and maximum values, going from a local (to a parallel partition) to a global indexing and viceversa, etc.
- `StructuredColumns`: It relates a given field to the underlying structured grid, thus enabling the field to parallel communication. It also allows similar operations as the `NodeColumns` function space, such as going from a local to a global indexing and viceversa.
- `Spectral`: It allows the spectral representation of a Field (no relation to a grid or a mesh here!). The parallelisation in this case is achieved through the `Trans` library.

For each of the function spaces presented in the following, we show both the C++ and Fortran version.

## 7.1 NodeColumns

For this example, given the length of the code, we divide the code listings into four different pieces, each of which will highlight different functionalities of the function

space.

### 7.1.1 C++ version

Construction of Fields

The listing 7.1 shows how to construct the function space `nodes` of type `NodeColumns` starting from a mesh. In particular, we first initialize the *Atlas* library and we define a global structured grid (see lines 21, 24 and 25). Using this grid, we then construct the mesh (see lines 28 and 29) and we get the number of nodes of the mesh (see line 33). On line 36, we define another integer that is used to construct three-dimensional fields if necessary. Specifically, this integer is intended to constitute the number of vertical levels required. Finally, on lines 39 and 40, we define the function space `nodes`. Note that we pass two arguments here: the first is the mesh, while the second is the halo. A halo is defined as an extra layer of ghost elements that is required, for instance, to calculated derivatives when a larger stencil is needed. In this case, we just asked for one extra layer of ghost elements (i.e. `Halo(1)`). Using the function space `nodes` just generated, we create various fields to highlight the different existing possibilities.

From line 43 to 46, we define two scalar fields (e.g. pressure, wind velocity magnitude, etc.). The first field is two-dimensional since it does not specify any vertical level. In addition, its dimensions automatically correspond to the number of nodes present in the mesh (i.e. we do not have to specify its dimensions!), because the field is constructed using the function space. Also, by using the function space, we automatically enable the field to parallel computation.

From line 47 to 50, we define two vector fields (e.g. wind velocity, etc.). Again, the first field is purely two-dimensional, while the second contains the vertical direction through the parameter `nb_levels`, that represents the number of vertical levels.

Finally, from line 51 to 54, we show an example on how to construct two tensor fields, the first two-dimensional and the second three-dimensional. The same observations done before for scalar and vector fields hold also in this case.

```
1 #include "atlas/atlas.h"
2 #include "atlas/runtime/Log.h"
3 #include "atlas/grid/grids.h"
4 #include "atlas/field/Field.h"
5 #include "atlas/array/ArrayView.h"
6 #include "atlas/mesh/Mesh.h"
7 #include "atlas/mesh/Nodes.h"
8 #include "atlas/mesh/generators/Structured.h"
```

```
9  #include "atlas/output/Gmsh.h"
10 #include "atlas/functionspace/NodeColumns.h"
11
12 using atlas::array::ArrayView;
13 using atlas::array::make_shape;
14 using atlas::atlas_finalize;
15 using atlas::atlas_init;
16 using atlas::field::Field;
17 using atlas::field::FieldSet;
18 using atlas::field::global;
19 using atlas::functionspace::NodeColumns;
20 using atlas::gidx_t;
21 using atlas::grid::Structured;
22 using atlas::Log;
23 using atlas::mesh::Halo;
24 using atlas::mesh::Mesh;
25 using atlas::output::Gmsh;
26
27 int main(int argc, char *argv[])
28 {
29     atlas_init(argc, argv);
30
31     // Generate global classic reduced Gaussian grid
32     Structured::Ptr grid( Structured::create( "N32" ) );
33
34     // Generate mesh associated to structured grid
35     atlas::mesh::generators::Structured meshgenerator;
36     Mesh::Ptr mesh ( meshgenerator.generate(*grid) );
37
38     // Number of nodes in the mesh
39     // (different from number of points on a grid!)
40     size_t nb_nodes = mesh->nodes().size();
41
42     // Number of vertical levels required
43     size_t nb_levels = 10;
44
45     // Generate functionspace associated to mesh
46     NodeColumns::Ptr fs_nodes( new NodeColumns(*mesh, Halo(1)) );
47
48     // Note on field generation
49     Field::Ptr field_scalar1(
50       fs_nodes->createField<double>("scalar1") );
51     Field::Ptr field_scalar2(
52       fs_nodes->createField<double>("scalar2", nb_levels) );
53     Field::Ptr field_vector1(
54       fs_nodes->createField<double>("vector1", make_shape(2)) );
55     Field::Ptr field_vector2(
```

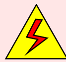
**Listing 7.1** Functionspace NodeColumns usage (1) using C++

## Definition/visualization of a scalar Field

In listing 7.2, we show the effective construction of a scalar field. We use the same function adopted in section 6.3, however, in this case, the function is not defined on a grid but the mesh through the function space `nodes`. This

also allows us to visualize the function in gmsh.

From line 3 to 8, we define some variables needed for the function that will be generated. On line 11, we define the `ArrayView` object on the two-dimensional scalar field defined in listing 7.1. On line 12, we extract the implicitly defined `lonlat` object - this step is particularly important since allows us to have access to all the coordinates of the nodes in our mesh, order in a 'lonlat fashion' (where the first dimension, 0, represents the longitudes, while the second, 1, represents the latitudes). From line 14 to 28, we define the function, a Gaussian-type (hill) function - note that the function is now defined on the number of nodes of the mesh (not on the number of points of the grid as in section 6.3!)



**Warning**

Note that the number of points of a grid is different from the number of nodes of a mesh!

From line 31 to 34, we finally write the mesh and the field in a gmsh format, so that we can visualize it!

```
1     Field::Ptr field_vector2(
2       fs_nodes->createField<double>("vector2", nb_levels,
3                                     make_shape(2)) );
4     Field::Ptr field_tensor1(
5       fs_nodes->createField<double>("tensor1", make_shape(2,2)) );
6     Field::Ptr field_tensor2(
7       fs_nodes->createField<double>("tensor2", nb_levels,
8                                     make_shape(2,2)) );
9     /* .... */
10    // Variables for scalar1 field definition
11    const double rpi = 2.0 * asin(1.0);
12    const double deg2rad = rpi / 180.;
13    const double zlatc = 0.0 * rpi;
14    const double zlonc = 1.0 * rpi;
15    const double zrad  = 2.0 * rpi / 9.0;
16    double  zdist, zlon, zlat;
17
18    // Retrieve lonlat field to calculate scalar1 function
19    ArrayView <double,1> scalar1(*field_scalar1);
20    ArrayView <double,2> lonlat ( mesh->nodes().lonlat() );
21    for (int jnode = 0; jnode < nb_nodes; ++jnode)
22    {
23        zlon = lonlat(jnode,0) * deg2rad;
24        zlat = lonlat(jnode,1) * deg2rad;
25
26        zdist = 2.0 * sqrt((cos(zlat) * sin((zlon-zlonc)/2)) *
27                        (cos(zlat) * sin((zlon-zlonc)/2)) +
28                     sin((zlat-zlatc)/2) * sin((zlat-zlatc)/2));
29
30        scalar1(jnode) = 0.0;
31        if (zdist < zrad)
32        {
33            scalar1(jnode) = 0.5 * (1. + cos(rpi*zdist/zrad));
```

```
34          }
35      }
```

**Listing 7.2** Functionspace NodeColumns usage (2) using C++

Parallel management

In listing 7.3, we show some operations related to the parallel behaviour of the function space. One of the most important operations from this perspective is `HaloExchange` (see line 3). This operation allows the correct exchange of information across different parallel partitions, when, for instance, calculating derivatives. A useful command to verify that this operation (or other operations) has not corrupted the data is the one reported on line 4, `checksum`. This prints a unique identifier for the object that is being passed as an argument. If anything in the object changes, this identifier will also change, permitting the identification of a possible unwanted data corruption. Note also that when printing to the screen this identifier, we access the MPI rank through `eckit`.

On line 8, we define a global field, `field_global`. This, even if the job we are running is parallel, is defined on one task only. A global field can be particularly useful for input/output purposes! Note also that the construction of the global field is based on the existing scalar field - i.e. the global field assumes the same characteristics of the scalar field, with the only exception that it is defined on one task only.

On line 11, we apply the `gather` operation to the global field. Note that the first argument is the input (in our case `field_scalar1`), while the second argument is the output (in our case `field_global`).

We successively print to the screen the number of local mesh nodes per parallel partition, the number of points of the grid and the number of nodes of the global field.

On line 24, we perform the opposite of the gather operation, `scatter`. As for the `gather` operation, the first argument is the input and the second the output.

From line 27 to 29, we perform an additional `HaloExchange`, and check the integrity of our `field_scalar1` after all the operations performed using it! Finally, from line 32 to 36, we show how `checksum` can be applied also to the `FieldSet` object.

```
1       }
2
3       // Write mesh and field in gmsh format for visualization
```

```
4     Gmsh gmsh("scalar1.msh");
5     gmsh.write(*mesh);
6     gmsh.write(*field_scalar1);
7
8     /* .... */
9     // Halo exchange
10    fs_nodes->haloExchange(*field_scalar1);
11    std::string checksum = fs_nodes->checksum(*field_scalar1);
12    Log::info() << checksum << std::endl;
13
14    // Create a global field
15    Field::Ptr field_global(
16       fs_nodes->createField("global", *field_scalar1, global() ) );
17    // Gather operation
18    fs_nodes->gather(*field_scalar1, *field_global);
19
20    Log::info() << "local nodes          = "
21                << fs_nodes->nb_nodes()  << std::endl;
22    Log::info() << "grid points           = "
23                << grid->npts()   << std::endl;
24    Log::info() << "field_global.shape(0) = "
25                << field_global->shape(0) << std::endl;
26
27    // Scatter operation
28    fs_nodes->scatter(*field_global, *field_scalar1);
29
30    // Halo exchange and checksum
31    fs_nodes->haloExchange(*field_scalar1);
32    checksum = fs_nodes->checksum(*field_scalar1);
33    Log::info() << field_scalar1->name() << " checksum : "
34                << checksum << std::endl;
35
36    // FieldSet checksum
37    FieldSet fields;
```

**Listing 7.3** Functionspace nodes usage (3) using C++

## Simple operations

In listing 7.4, we show some simple operations that can be performed using the `nodes` function space. Specifically, on lines 8 and 9 we compute the minimum and maximum values of our `field_scalar1` - note that the function space knows about the parallelisation, therefore there is no need to do anything else to obtain the correct minimum and maximum values. This will also be true for the other operations described below.

On lines 14 and 15, we again calculate the minimum and the maximum values of our field but, in this case, we also retrieve the position of these values through the global index of the mesh nodes.

On line 22 and 27, we calculate the sum of all the values of our field present in the mesh. The two approaches should return the same number except for possible round-off errors related to the mesh partitioning.

Finally, on line 32 and 36, we compute the mean value of our field and its standard deviation. Note that the mean and the standard deviation are normalised with respect to the total number of nodes present in the mesh.

It is also important to observe that we have used a different approach than `cout` to print the values of the quantities just calculated to the screen. In particular, we used the `Log::Info()` utility of *Atlas*, that will be better explained in section part 2.7.2, of this user-guide. On line 42, we finalize the library as usual.

```
1      FieldSet fields;
2      fields.add(*field_scalar1);
3      fields.add(*field_vector1);
4      checksum = fs_nodes->checksum(fields);
5      Log::info() << "FieldSet checksum : " << checksum << std::endl;
6
7      /* .... */
8
9      // Operations
10     size_t N;
11     gidx_t gidx_min, gidx_max;
12     double min, max, sum, mean, stddev;
13
14     // Minimum and maximum
15     fs_nodes->minimum(*field_scalar1, min);
16     fs_nodes->maximum(*field_scalar1, max);
17     Log::info() << "min: " << min << std::endl;
18     Log::info() << "max: " << max << std::endl;
19
20     // Minimum and maximum + location
21     fs_nodes->minimumAndLocation(*field_scalar1, min, gidx_min);
22     fs_nodes->maximumAndLocation(*field_scalar1, max, gidx_max);
23     Log::info() << "min: "         << min << ",  "
24                 << "global_id = " << gidx_min << std::endl;
25     Log::info() << "max: "         << max << ",  "
26                 << "global_id = " << gidx_max << std::endl;
27
28     // Summation
29     fs_nodes->sum(*field_scalar1, sum, N);
30     Log::info() << "sum: "          << sum
31                 << ", nb_nodes = " << N << std::endl;
32
33     // Order independent (from partitioning) summation
34     fs_nodes->orderIndependentSum(*field_scalar1, sum, N);
35     Log::info() << "oi_sum: "       << sum
36                 << ", nb_nodes = " << N << std::endl;
37
38     // Average over number of nodes
39     fs_nodes->mean(*field_scalar1, mean, N);
40     Log::info() << "mean: " << mean << ", nb_nodes = " << N << std::
       endl;
41
42     // Average and standard deviation over number of nodes
```

**Listing 7.4** Functionspace nodes usage (4) using C++

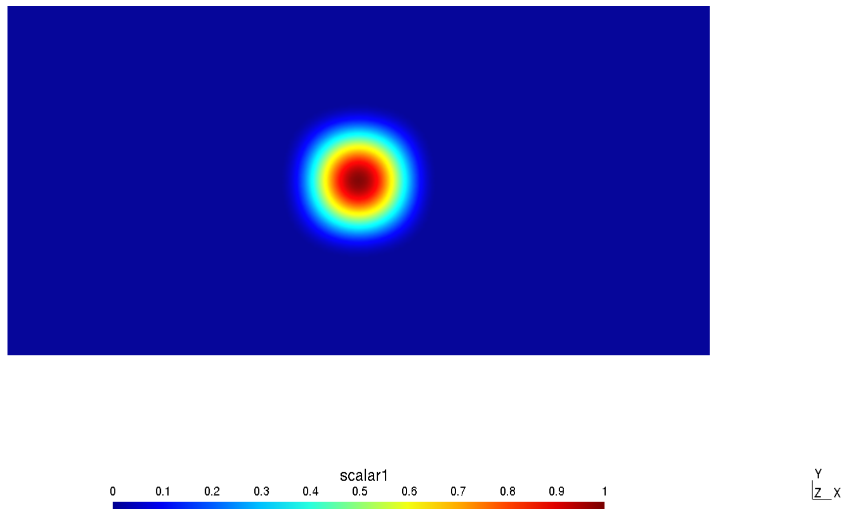



**Figure 7.1** Gaussian-type field visualised in Gmsh

It is now possible to run this simple program by using a command-line argument representing the keyword that identifies an *Atlas* predefined grid. For instance, we can execute the following command line

```
./atlas_c-NodeColumns --grid O128
```

This will produce an octahedral reduced Gaussian grid with 128 latitudes on one hemisphere (i.e. 256 latitudes in total), that is then used to generate the mesh and the `nodes` function space. These two will then be used to construct our Gaussian-type (hill) scalar field. The mesh and the field are then written into two `.msh` files. These can be easily opened using Gmsh and they should look like figure 7.1. It will also print to the screen the following output:

```
[0] (2016-02-11 T 19:24:05) (I) -- writing mesh to gmsh file mesh.msh
[0] (2016-02-11 T 19:24:05) (I) -- writing field partition \
to gmsh file ./mesh_info.msh
[0] (2016-02-11 T 19:24:05) (I) -- writing field partition...
[0] (2016-02-11 T 19:24:05) (I) -- writing field scalar1 \
to gmsh file scalar1.msh
[0] (2016-02-11 T 19:24:05) (I) -- writing field scalar1...
de6c4b5f15bde3b75e3fe927a0e4904c
local nodes          = 70912
grid points           = 70144
field_global.shape(0) = 70144
de6c4b5f15bde3b75e3fe927a0e4904c
9d3b18735c8f114cf4f033204db73e78
[0] (2016-02-11 T 19:24:05) (I) -- min: 0
[0] (2016-02-11 T 19:24:05) (I) -- max: 0.99981
```

```
[0] (2016-02-11 T 19:24:05) (I) -- min: 0,  global_id = 1
[0] (2016-02-11 T 19:24:05) (I) -- max: 0.99981,  global_id = 34809
[0] (2016-02-11 T 19:24:05) (I) -- sum: 2872.24, nb_nodes = 70144
[0] (2016-02-11 T 19:24:05) (I) -- oi_sum: 2872.24, nb_nodes = 70144
[0] (2016-02-11 T 19:24:05) (I) -- mean: 0.0409478, nb_nodes = 70144
[0] (2016-02-11 T 19:24:05) (I) -- mean = 0.0409478,  \
std_deviation: 0.1496,  nb_nodes: 70144
```

In the first few lines, we note that the code informs us that it is writing the mesh into gmsh format as well as the field in gmsh format. Successively, we print the first `checksum`, that is a string, the number of local (to partition) nodes, the total number of grid points and the number of entries present in `field_global`. Note how, for this non-parallel example the number of mesh points differs from the number of grid points as mentioned earlier. We then print the other two `checksum`, the first of them is identical to the one printed previously - i.e. `field_scalar1` has not been corrupted! - while the second is different since it refers to a different object - i.e. it refer to `FieldSet`. Finally, on the last few lines we plot the various minimum, maximum, summation, average and standard deviation calculated using the `nodes` function space operations. Note the additional verbosity of the `Log::Info()` *Atlas* utility!

You can now try to generate different meshes and run it in parallel!

### 7.1.2 Fortran version

Construction of Fields

The listing 7.5 shows how to construct the function space `nodes` starting from a mesh. In the first few lines, we define the variables needed for this example - note in particular the definition of `atlas_functionspace_NodeColumns` and `atlas_mesh_Nodes`. We then create a structured grid (see lines 48 and 49) and the associated mesh (see lines 52 and 53). On line 45 we initialize the library as usual, while, on line 56, we define the `nodes` function space - note that we pass two arguments here: the first is the mesh, while the second is the halo. A halo is defined as an extra layer of ghost elements that is required, for instance, to calculate derivatives when a larger stencil is needed. In this case, we just asked for one extra layer of ghost elements (i.e. `halo_size = 1`).

Using the function space `nodes` just generated, we create various fields to highlight the different existing possibilities.

From line 59 to 62, we define two scalar fields (e.g. pressure, wind velocity magnitude, etc.). The first field is two-dimensional since it does not specify any vertical level. In addition, its dimensions automatically correspond to

the number of nodes present in the mesh (i.e. we do not have to specify its dimensions!), because the field is constructed using the function space. Also, by using the function space, we automatically enable the field to parallel computation.

From line 63 to 66, we define two vector fields (e.g. wind velocity, etc.). Again, the first field is purely two-dimensional, while the second contains the vertical direction through the parameter `nb_levels`, that represents the number of vertical levels.

Finally, from line 67 to 70, we show an example on how to construct two tensor fields, the first two-dimensional and the second three-dimensional. The same observations done before for scalar and vector fields hold also in this case.

```
1  program main
2  use, intrinsic :: iso_c_binding, only: c_double
3  use atlas_module
4  implicit none
5  integer, parameter                          :: wp = c_double
6  character(len=1024)                         :: string
7  character(len=1024)                         :: gridID
8  character(len=32)                           :: checksum
9  type(atlas_grid_Structured)                   :: grid
10 type(atlas_mesh)                            :: mesh
11 type(atlas_meshgenerator)                   :: meshgenerator
12 type(atlas_Output)                          :: gmsh
13 type(atlas_functionspace_NodeColumns)       :: fs_nodes
14 type(atlas_mesh_Nodes)                      :: meshnodes
15 type(atlas_Field)                           :: field_scalar1
16 type(atlas_Field)                           :: field_scalar2
17 type(atlas_Field)                           :: field_vector1
18 type(atlas_Field)                           :: field_vector2
19 type(atlas_Field)                           :: field_tensor1
20 type(atlas_Field)                           :: field_tensor2
21 type(atlas_Field)                           :: lonlatField
22 type(atlas_Field)                           :: field_global
23 type(atlas_FieldSet)                        :: fields
24 integer                                     :: nb_nodes, jnode
25 integer                                     :: nb_levels = 10
26 integer                                     :: halo_size = 1
27 type(atlas_Field)                           :: global, scal
28 real(wp), pointer                           :: scalar1(:)
29 real(wp), pointer                           :: lonlat(:,:)
30 real(wp)                                    :: minimum, maximum
31 real(wp)                                    :: sum, oisum
32 real(wp)                                    :: mean, stddev
33 real(wp), allocatable                       :: minimumv(:), maximumv(:)
34 real(wp), allocatable                       :: sumv(:), oisumv(:)
35 real(wp), allocatable                       :: meanv(:), stddevv(:)
36 integer(ATLAS_KIND_GIDX)                    :: glb_idx
37 integer(ATLAS_KIND_GIDX), allocatable       :: glb_idxv (:)
38
39 ! Variables for scalar1 field definition
40 real(wp), parameter :: rpi = 2._wp * asin(1._wp)
41 real(wp), parameter :: deg2rad = rpi / 180._wp
42 real(wp), parameter :: zlatc = 0._wp * rpi
```

```
43 real(wp), parameter :: zlonc = 1._wp * rpi
44 real(wp), parameter :: zrad  = 2._wp * rpi / 9._wp
45 real(wp)            :: zdist, zlon, zlat;
46
47 call atlas_init()
48
49 ! Generate global classic reduced Gaussian grid
50 gridID = "N32"
51 grid = atlas_grid_Structured(gridID)
52
53 ! Generate mesh associated to structured grid
54 meshgenerator = atlas_meshgenerator_Structured()
55 mesh          = meshgenerator%generate(grid)
56
57 ! Generate functionspace associated to mesh
58 fs_nodes      = atlas_functionspace_NodeColumns(mesh, halo_size)
59
60 ! Note on field generation
61 field_scalar1 = fs_nodes%create_field("scalar1", &
62               & atlas_real(wp))
63 field_scalar2 = fs_nodes%create_field("scalar2", &
64               atlas_real(wp), nb_levels)
65 field_vector1 = fs_nodes%create_field("vector1", &
66               & atlas_real(wp), [2])
67 field_vector2 = fs_nodes%create_field("vector2", &
68               atlas_real(wp), nb_levels, [2])
69 field_tensor1 = fs_nodes%create_field("tensor1", &
70               & atlas_real(wp), [2,2])
71 field_tensor2 = fs_nodes%create_field("tensor2", &
```

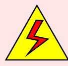
**Listing 7.5** Functionspace atlas_functionspace_NodeColumns usage (1) using Fortran

## Definition/visualization of a scalar Field

In listing 7.6, we show the effective construction of a scalar field. We use the same function adopted in section 6.3, however, in this case, the function is not defined on a grid but the mesh through the function space `nodes`. This also allows us to visualize the function in gmsh.

On lines 4 and 5, we define the number of nodes of the mesh using the `nodes` function space object.

On line 8, we initialize the pointer `scalar1` associated to the two-dimensional `field_scalar1` object defined in listing 7.1. On lines 9 and 10, we extract the implicitly defined `lonlat` object - this step is particularly important since allows us to have access to all the coordinates of the nodes in our mesh, ordered in a 'lonlat fashion' (where the first dimension, 1, represents the longitudes, while the second, 2, represents the latitudes). From line 12 to 24, we define the function, a Gaussian-type (hill) function - note that the function is now defined on the number of nodes of the mesh (not on the number of points of the grid as in section 6.3!)



**Warning**

Note that the number of points of a grid is different from the number of nodes of a mesh!

On lines 27 and 28, we finally write the mesh and the field in a gmsh format, so that we can visualize it!

```
field_tensor2 = fs_nodes%create_field("tensor2", &
                atlas_real(wp), nb_levels, [2,2])
!........!
! Number of nodes in the mesh
! (different from number of points on a grid!)
meshnodes     = fs_nodes%nodes()
nb_nodes      = fs_nodes%nb_nodes()

! Retrieve lonlat field to calculate scalar1 function
call field_scalar1%data(scalar1)
lonlatField = meshnodes%lonlat()
call lonlatField%data(lonlat)

do jnode=1,nb_nodes
  zlon = lonlat(1,jnode) * deg2rad
  zlat = lonlat(2,jnode) * deg2rad

  zdist = 2._wp * sqrt((cos(zlat) * sin((zlon-zlonc)/2._wp)) *  &
          & (cos(zlat) * sin((zlon-zlonc)/2._wp)) + &
          &  sin((zlat-zlatc)/2._wp) * sin((zlat-zlatc)/2._wp))

  scalar1(jnode) = 0._wp;
  if (zdist < zrad) then
    scalar1(jnode) = 0.5_wp * (1._wp + cos(rpi*zdist/zrad));
  endif
enddo

! Write mesh and field in gmsh format for visualization
gmsh = atlas_output_Gmsh("mesh.msh")
```

**Listing 7.6** Functionspace atlas_functionspace_NodeColumns usage (2) using Fortran

## Parallel management

In listing 7.7, we show some operations related to the parallel behaviour of the function space. One of the most important operations from this perspective is `halo_exchange` (see line 3). This operation allows the correct exchange of information across different parallel partitions, when, for instance, calculating derivatives. A useful command to verify that this operation (or other operations) has not corrupted the data is the one reported on line 5, `checksum`. This prints a unique identifier for the object that is being passed as an argument. If anything in the object changes, this identifier will also change, permitting the identification of a possible unwanted data corruption.

Note also that when printing to the screen this identifier, we access the MPI rank through the `atlas_mpi_rank()` function.

On line 11, we define a global field, `field_global`. This, even if the job we are running is parallel, is defined on one task only. A global field can be particularly useful for input/output purposes! Note also that the construction of the global field is based on the existing scalar field - i.e. the global field assumes the same characteristics of the scalar field, with the only exception that it is defined on one task only.

On line 14, we apply the `gather` operation to the global field. Note that the first argument is the input (in our case `field_scalar1`), while the second argument is the output (in our case `field_global`).

We successively print to the screen the number of local mesh nodes per parallel partition, the number of points of the grid and the number of nodes of the global field.

On line 22, we perform the opposite of the gather operation, `scatter`. As for the `gather` operation, the first argument is the input and the second is the output.

On lines 25 and 26, we perform an additional `halo_exchange`, and check the integrity of our `field_scalar1` after all the operations performed using it! Finally, from line 33 to 35, we show how `checksum` can be applied also to an `atlas_FieldSet` object.

```
1  gmsh = atlas_output_Gmsh("mesh.msh")
2  call gmsh%write(mesh)
3  gmsh = atlas_output_Gmsh("scalar1.msh")
4  call gmsh%write(field_scalar1)
5  !........!
6  ! Halo exchange
7  call fs_nodes%halo_exchange(field_scalar1)
8
9  checksum = fs_nodes%checksum(field_scalar1)
10 write(string, *) checksum
11 call atlas_log%info(string)
12
13 ! Create a global field
14 field_global = fs_nodes%create_field("global", field_scalar1, global
       =.true.)
15
16 ! Gather operation
17 call fs_nodes%gather(field_scalar1, field_global);
18
19 write(string, *) "local nodes          = ", fs_nodes%nb_nodes()
20 call atlas_log%info(string)
21
22 write(string, *) "grid points          = ", grid%npts()
23 call atlas_log%info(string)
24
25 write(string, *) "field_global.shape(1) = ", field_global%shape(1)
26 call atlas_log%info(string)
```

```
27
28 ! Scatter operation
29 call fs_nodes%scatter(field_global, field_scalar1)
30
31 ! Halo exchange and checksum
32 call fs_nodes%halo_exchange(field_scalar1);
33 checksum = fs_nodes%checksum(field_scalar1);
34 write(string, *) checksum
35 call atlas_log%info(string)
36
37 ! FieldSet checksum
38 fields = atlas_FieldSet("")
39 call fields%add(field_scalar1);
```

**Listing 7.7** Functionspace atlas_functionspace_NodeColumns usage (3) using Fortran

Simple operations

In listing 7.8, we show some simple operations that can be performed using the `nodes` function space. Specifically, on lines 5 and 6 we compute the minimum and maximum values of our `field_scalar1` - note that the function space knows about the parallelisation, therefore there is no need to do anything else to obtain the correct minimum and maximum values. This will also be true for the other operations described below.

On lines 12 and 15, we again calculate the minimum and the maximum values of our field but, in this case, we also retrieve the position of these values through the global index of the mesh nodes.

On lines 20 and 21, we calculate the sum of all the values of our field present in the mesh. The two approaches should return the same number except for possible round-off errors related to the mesh partitioning.

Finally, on lines 26 and 31, we compute the mean value of our field and its standard deviation. Note that the mean and the standard deviation are normalised with respect to the total number of nodes present in the mesh.

It is also important to observe that we have used a different approach than the standard `write` to print the values of the quantities just calculated to the screen. In particular, we used the `atlas_log` utility of *Atlas*, that will be better explained in section part 2.7.2, of this user-guide.

Note also that from line 38 to line 48, we destroy all the objects of created in the program in order to release the memory and on line 50 we finalize the *Atlas* library.

```
1 call fields%add(field_scalar1);
2 call fields%add(field_global);
3 checksum = fs_nodes%checksum(fields);
4 write(string, *) checksum
```

```
5  call atlas_log%info(string)
6  !........!
7  ! Operations
8
9  ! Minimum and maximum
10 call fs_nodes%minimum(field_scalar1, minimum)
11 call fs_nodes%maximum(field_scalar1, maximum)
12 write(string, *) "min = ",minimum, " max = ", maximum;
13 call atlas_log%info(string)
14
15
16 ! Minimum and maximum + location
17 call fs_nodes%minimum_and_location(field_scalar1, minimum, glb_idx)
18 write(string,*) "min = ",minimum, " gidx = ", glb_idx
19 call atlas_log%info(string)
20 call fs_nodes%maximum_and_location(field_scalar1, maximum, glb_idx)
21 write(string,*) "max = ",maximum, " gidx = ", glb_idx
22 call atlas_log%info(string)
23
24 ! Summation and order indipedent summation
25 call fs_nodes%sum(field_scalar1, sum)
26 call fs_nodes%order_independent_sum(field_scalar1, oisum)
27 write(string,*) "sum = ", sum, " oisum = ", oisum
28 call atlas_log%info(string)
29
30 ! Average over number of nodes
31 call fs_nodes%mean(field_scalar1, mean)
32 write(string,*) "mean = ", mean
33 call atlas_log%info(string)
34
35 ! Average and standard deviation over number of nodes
36 call fs_nodes%mean_and_standard_deviation(&
37                 & field_scalar1, mean, stddev)
38 write(string,*) "mean = ", mean
39 call atlas_log%info(string)
40 write(string,*) "stddev = ", stddev
41 call atlas_log%info(string)
42
43 call grid         %final()
44 call mesh         %final()
45 call fs_nodes     %final()
46 call field_scalar1%final()
47 call field_scalar2%final()
48 call field_vector1%final()
49 call field_vector2%final()
50 call field_tensor1%final()
51 call field_tensor2%final()
52 call field_global %final()
```

**Listing 7.8** Functionspace atlas_functionspace_NodeColumns usage (4) using Fortran

It is now possible to run this simple program by using a command-line argument representing the keyword that identifies an *Atlas* predefined grid. For instance, we can execute the following command line

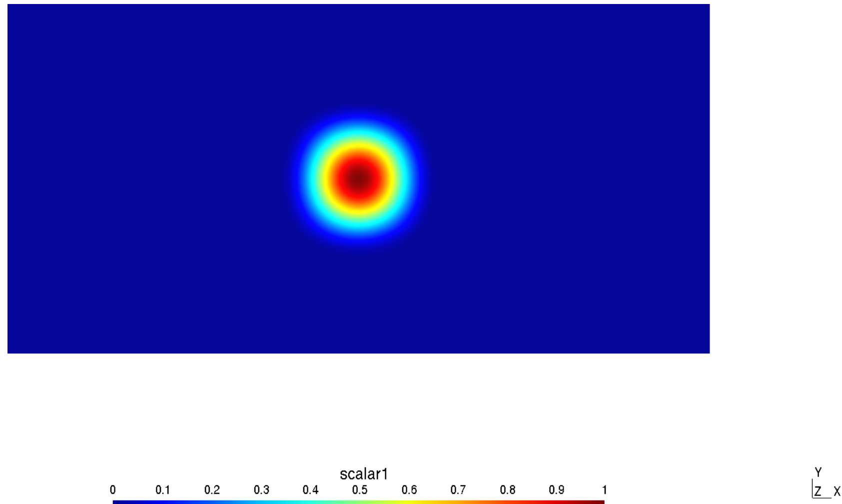



**Figure 7.2** Gaussian-type field visualised in Gmsh

```
./atlas_f-NodeColumns --grid O128
```

This will produce an octahedral reduced Gaussian grid with 128 latitudes on one hemisphere (i.e. 256 latitudes in total), that is then used to generate the mesh and the `nodes` function space. These two will then be used to construct our Gaussian-type (hill) scalar field. The mesh and the field are then written into two `.msh` files. These can be easily opened using Gmsh and they should look like figure 7.2. It will also print to the screen the following output:

```
[0] (2016-02-12 T 15:41:33) (I) -- Looking for \
MeshGeneratorFactory [Structured]
[0] (2016-02-12 T 15:41:34) (I) -- writing mesh \
to gmsh file mesh.msh
[0] (2016-02-12 T 15:41:34) (I) -- writing field \
field_scalar1 to gmsh file scalar1.msh
[0] (2016-02-12 T 15:41:34) (I) -- writing field \
field_scalar1...
de6c4b5f15bde3b75e3fe927a0e4904c
local nodes          =        70912
grid points          =        70144
field_global.shape(1) =        70144
de6c4b5f15bde3b75e3fe927a0e4904c
9d3b18735c8f114cf4f033204db73e78
[0] (2016-02-12 T 15:41:34) (I) --  min =    0.0000000000000000 \
max =   0.99981015482709013
[0] (2016-02-12 T 15:41:34) (I) --  min =    0.0000000000000000 \
gidx =                     1
```

```
[0] (2016-02-12 T 15:41:34) (I) --  max =   0.99981015482709013 \
gidx =                 34809
[0] (2016-02-12 T 15:41:34) (I) --  sum =    2872.2433544940809 \
oisum =    2872.2433544940809
[0] (2016-02-12 T 15:41:34) (I) --  mean =    4.0947812421505490E-002
[0] (2016-02-12 T 15:41:34) (I) --  mean =    4.0947812421505490E-002
[0] (2016-02-12 T 15:41:34) (I) --  stddev =   0.14959959397019881
```

In the first few lines, we note that the code informs us that it is writing the mesh into gmsh format as well as the field in gmsh format. Successively, we print the first `checksum`, that is a string, the number of local (to partition) nodes, the total number of grid points and the number of entries present in `field_global`. Note how, for this non-parallel example the number of mesh points differs from the number of grid points as mentioned earlier. We then print the other two `checksum`, the first of them is identical to the one printed previously - i.e. `field_scalar1` has not been corrupted! - while the second is different since it refers to a different object - i.e. it refer to `atlas_FieldSet`. Finally, on the last few lines we plot the various minimum, maximum, summation, average and standard deviation calculated using the `nodes` function space operations. Note the additional verbosity of the `atlas_log` *Atlas* utility!

You can now try to generate different meshes and run it in parallel!

## 7.2 StructuredColumns

## 7.3 Spectral