ATHAM-FLUIDITY MICROPHYSICS API

1. Introduction

This document should be read in conjection with the main Fluidity manual. The Fluidity microphysics API exposes the internal represesentation of dynamic and tracer fields in the Fluidity state and field types to external routines. Three approaches are available, direct calls from Fortran using the Fluidity derived types, rewrapping of fortran/modules through the python interpreter, and writing python code directly to modify the variables. This document is arranged as follows: This section contains a brief explanation of the ATHAM-Fluidity core algorithm, section 2 introduces and describes the fluidity data structure and section 3 describes the three different approaches to building a microphysics model, with examples.

1.1. The ATHAM-Fluidity equations and algorithm.

1.1.1. *Fluidity Navier-Stokes solver.* The compressible branch of Fluidity solves the compressible Navier-Stokes equations

$$\rho\frac{\partial \boldsymbol{u}}{\partial t}+\rho\boldsymbol{u}\cdot\nabla\boldsymbol{u}+\boldsymbol{\Omega}\times\rho\boldsymbol{u}=\nabla p-\rho g\hat{\boldsymbol{z}},$$

$$\frac{\partial \rho}{\partial t}+\nabla\cdot\rho\boldsymbol{u}=-\sigma\rho+S_\rho,$$

with tracer equation(s)

$$\frac{\partial \rho\tau}{\partial t}+\nabla\cdot\rho F\left(\tau\right)\boldsymbol{u}=-\sigma\tau+S_\tau.$$

$$F\left(\tau\right)=\begin{cases}\tau & \text{(advection-diffusion)}\\ \tau+\frac{p}{\rho} & \text{(energy equation)}\end{cases}$$

Closure through equation of state

$$\rho=\rho\left(p,\tau\right)$$

Let the result of the i-th interation of the nonlinear loop for variable a evaluated at an intermediate time level be given by

$$a_i^{n+\theta}:=\left(1-\theta\right)a^n+a_{i+1}^{n+1}\approx a\left(t_n+\theta\Delta t\right)$$

$$\rho_i^n\left(\frac{\boldsymbol{u}_\dagger^{n+1}-\boldsymbol{u}^n}{\Delta}\right)+\rho\boldsymbol{u}_i^{n+\theta_u}\cdot\nabla\boldsymbol{u}_\dagger^{n+\theta_u}+\boldsymbol{\Omega}\times\rho\boldsymbol{u}_{i+1}^{n+\theta_u}=\nabla p_i^{n+\theta_p}-\rho_i^{n+1}g\hat{\boldsymbol{z}},$$

$$\frac{\tau_{i+1}^{n+1}-\tau^n}{\Delta t}+\boldsymbol{u}_i^{n+\theta_\tau}\cdot\nabla\tau_{i+1}^{n+\theta_\tau}=-\sigma_\tau\tau_{i+1}^{n+1}+S_\tau,$$

$$\rho_i^{n+1}=\rho\left(p_\dagger^{n+1},\tau_{i+1}^{n+1}\right),$$

$$\rho_\dagger=\rho_i+\left(p_{i+1}^{n+1}-p_\dagger^{n+1}\right)\left.\frac{\partial\rho}{\partial p}\right|_{p_\dagger,\tau_{i+1}^{n+1}},$$

$$\rho_\dagger^{n+\theta}\boldsymbol{u}_{i+1}^{n+1}=\rho_i^{n+\theta}\boldsymbol{u}_\dagger^{n+1}+\theta_p\Delta t\nabla p_i^{n+1},$$

$$\frac{\rho_\dagger^{n+1}-\rho^n}{\Delta t}+\nabla\cdot\left(\rho_\dagger^{n+\theta_\rho}\boldsymbol{u}_{i+1}^{n+\theta_\rho}\right)=-\sigma_\rho\rho_\dagger^{n+\theta}+S_\rho$$

$$\left(\left(\frac{1}{\Delta t}+\sigma_\rho\theta_\rho\right)\frac{\partial\rho}{\partial p}-\Delta t\theta_\rho\theta_p\nabla^2\right)\left(p_{i+1}^{n+1}-p_i^{n+1}\right)=-\frac{\rho_i^{n+1}+\frac{\partial\rho}{\partial p}-\rho^n}{\Delta t}-\nabla\cdot\left(\rho_\dagger^{n+\theta_\rho}\boldsymbol{u}_{i+1}^{n+\theta_\rho}\right)+\sigma_\rho\left(\rho_i^{n+\theta}+\theta_\rho\frac{\partial\rho}{\partial p}\left(p_i^{n+1}-p_\dagger^{n+1}\right)\right)$$

$$\rho_{i+1}^{n+1}=\rho\left(p_{i+1}^{n+1},\tau_{i+1}^{n+1}\right),$$

1.1.2. *Atham entropy variable and equation of state:* The ATHAM framework assumes that entropy is represented by an advected bulk potential temperature, Θ, satisfying an advection equation,

$$\frac{\partial \Theta}{\partial t} + \boldsymbol{u} \cdot \nabla \Theta = S_\Theta.$$

The equation of state is

$$\begin{aligned} p = \rho_g R_g T &= \left(\frac{\rho \sum_{n=g} R_n q_n}{1 - \sum_{n \neq g} \frac{\rho q_n}{\rho_n}} \right) T \\ &= \left(\frac{\rho \sum_{n=g} R_n q_n}{1 - \rho \sum_{n \neq g} \frac{q_n}{\rho_n}} \right) \frac{\sum_n c_{p,n} q_n \Theta}{\left(\frac{p_0}{p}\right)^{\frac{R_g}{c_{p,g}}} c_{p,g} + \sum_{n \neq g} c_{p,n} q_n} \end{aligned}$$

or in Fluidity's forward form,

$$\rho = \frac{p \left(\left(\frac{p_0}{p}\right)^{\frac{R_g}{c_{p,g}}} c_{p,g} + \sum_{n \neq g} c_{p,n} q_n \right)}{\sum_{n=g} R_n q_n \sum_n c_{p,n} q_n \Theta - p \left(\left(\frac{p_0}{p}\right)^{\frac{R_g}{c_{p,g}}} c_{p,g} + \sum_{n \neq g} c_{p,n} q_n \right) \sum_{n \neq g} \frac{q_n}{\rho_n}}$$

2. Fluidity Data Structures

See also section 3.2 of the fluidity manual. Fluidity is a finite element solver. Functions are represented piecewise on elements by members of a function space, V. By default these are the continuous/discountinuous Lagrange piecewise polynomials P_n on triangular elements. A scalar field is thus represented by an array containing the weights of the basis functions on each node point. The continuous representation has only one node point at each physical location, while the discontinuous representation has multiple node points coinciding, but exactly one node point per element. Vectors are represented by storing the component tuples on the node points in a 2D array. Similiarly, tensors form a 3D array. See the manual or Figure 1 for schematics of the representation.

Fluidity's prefered element choice is $P_{1DG}P_2$. That is piecewise linear discontinuous velocity fields and piecewise continuous quadratic pressure. The $P_{(N-1)DG}P_N$ element pairing has a number of numerically advantagous properties, including LLB stability. Tracer fields are usually either P_{1DG} or in a control representation. This is a mixed representation. For fields which live on a particular mesh the ordering of the array is constant but this cannot be assumed here. It cannot be assumed (an in fact is practically guaranteed not) that the ordering on different meshes meshes will involve an obvious mapping.

3. Microphysics Interfaces

3.1. **Direct Fortran Interface through Fluidity Objects.** This API gets called when the .flml file fohr the problem has the option /cloud_microphysics/fortran_library/ selected and the code has been compiled and linked with the relevant options. In this case the external routine microphysics_main is called from the routine in the file parameterisation/Microphysics.F90. This has an explicit interface given in the same file.

```
interface
    subroutine microphysics_main(phases,extra_fields,current_time,dt)
         use state_module
         use fields
         type(state_type), intent(inout), dimension(:) :: phases
         type(scalar_field), intent(in),  dimension(:) :: pressure
         real, intent(in)                              :: time, dt
    end subroutine
end interface
```

The derived types (state_type and scalar_field) are defined in the modules femtools/State_module.F90 and femtools/field.F90 respectively. The kind of the real variables is defined at compile time, but is double length (real*8) by default.

Since this interface explicitly exposes the underlying Fluidity state it is both the most powerful and versatile interface. However it requires an understanding of the object oriented interface and details of the underlying data structure to be able to access and modify the necessary fields.

3.1.1. *Configuring and compiling with external Fortran Microphhysics.* To configure with the above call enabled, Fluidity must be configured with the preprocessor identifier FORTRAN_MICROPHYSICS defined when the file parameterisation/Microphysics.F90 is compiled, and with the file containing the correct symbols available at linking time. Providing the microphysics routines have been wrapped into a static library file, called e.g. libmicrophysics.a, then this can be performed using the following procedure

```
cd /path/to/fluidity/branch
./configure FCFLAGS=''-DFORTRAN_MICROPHYSICS \
LIBS=''-L/path/to/microphysics/directory -lmicrophysics'' --enable-2d-adaptivity
make clean
make
```

3.1.2. *An example microphysics file.*

3.2. **Combined Python/fortran wrapper.** This provides a more lightweight interface to the underlying data structure. The f2py automated wrapping tool, provided with the numpy python module, is used to compile a python shared object module which handles the underlying linking of the microphysics.

The Fortran type for scalar fields exposed externally here has the form

```
type basic_scalar
    sequence
    real, dimension(:), pointer :: new=>null()
    real, dimension(:), pointer :: old=>null()
    real, dimension(:), pointer :: source=>null()
end type basic_scalar
```

where for a scalar field, τ

$$\texttt{tau\%old(n)} = \tau(X_n, t^n),$$

the value of τ on the basis function of the nth node and at the old timelevel,

$$\texttt{tau\%new(n)} = \tau^*(X_n, t^{n+1} := t^n + \Delta t),$$

the value of τ after the latest nonlinear iteration, on the basis function of the nth node and at the new timelevel, while`tau%source(n)`is the value of the soure term in the advection-diffusion equation (see Fluidity manual) for τ, evaluated for the nth basis function. These are "lightweight" versions of the data arrays of thie equivalent scalar fields.For convenience the default wrapper also allows for calls at a point, using basic Fortran arrays.

An array wrapped microphysics module requires:

(1) A python file of the form

```
use fluidity.microphysics.FortranMicrophysics as FM
use fluidity.microphysics
Fields={'FieldType1':[('PhaseName1','FieldName1'),
                      ('PhaseName2','FieldName2'),]
        'FieldType2':[('PhaseName3','FieldName3'),],
        ...
       }
cmd_name='run_microphysics'
if __name__=='__main__':
    FM.MakeWrapper(Fields,cmd_name)
```

A library or object file containing a public Fortran routine matching the pattern

```
interface
    subroutine run_microphysics(FTs)
         type basic_scalar
             sequence
             real, dimension(:), pointer :: old=>null()
             real, dimension(:), pointer :: new=>null()
             real, dimension(:), pointer :: source=>null()
         end type basic_scalar
         type(basic_scalar), allocatable, dimension(:) :: FTS
    end subroutine
end interface
```

3.3. **Pure python interface.**