ForeFire user's guide

Part I

General Usage

ForeFire code is an API, a library and an interpreter with a "pythonesque" syntax. Its purpose is to simulate the evolution of reactive fronts that can be represented with front velocity models. It is designed such as it can easily be run from the command line, coupled with other codes and extended with user defined propagation and emission (fluxes) models.

Chapter 1

Install

ForeFire has been designed and run on Unix systems, three modules can be built with the source code.

1.1 Main binaries

- An interpreter (executable)
- A dynamic library (shared, with C/C++/Java and Fortran bindings)
- Pre-Post processing helping scripts.

NetCDF Library V3 or later must be installed on the system to build Forefire Get it from `http://www.unidata.ucar.edu/software/netcdf/`

Compilation requires a c++ compiler, but it has only been tested on gcc/g++ compiler. the SCons python tool is used to make the library and executable, get it from `http://www.scons.org/` A sample SConstruct file is included with the distribution, edit it and insert the path to the Netcdf (and Java headers for JNI bindings).

```
env = Environment(CPPPATH=['/usr/include/','/usr/local/include/'
,'/usr/lib64/jvm/java-1.6.0-openjdk-1.6.0/include/']
,CCFLAGS=['-Wall','-g0','-O3','-m64','-D_JNI_IMPLEMENTATION'])
Program('forefire', Glob('*.cpp'), LIBS=['netcdf_c++', 'netcdf'], LIBPATH=['/usr/local/lib/'])
env.SharedLibrary('ForeFire', Glob('*.cpp'), LIBS=['netcdf_c++', 'netcdf'], LIBPATH=['/usr/local/lib/'])
```

When the file is complete, simply run "scons" in the SConstruct file directory.

examples can be found in the examples directory to run it, type "forefire -i examplescript" from the commandline

1.2 Tools and scripts

Pre and post processing tools cans be found in the tools/ directory. These scripts require numpy, scipy, evtk, netcdf4python libraries.

Chapter 2

The interpreter

ForeFire interpreter can be found in the build path after compilation. It is named "forefire" and can be linked or copied in you bin/ directory.

The general operation is that interactions with the simulation is performed with commands, that may be run interactively, or scripted (with forefire -i "scriptfile"). All commands are quickly documented in the interpreter (by entering "help[]" command). outputs can be redirected to a file with forefire -i "scriptfile" -o "outputfile"

```
FireDomain[date, location, extension] : creates a fire domain
FireFront[date] : creates a fire front with the following fire nodes
FireNode[location, velocity] : adds a fire node to the given location
step[duration] : advances the simulation for a user-defined amount of time
goTo[date] : advances the simulation till a user-defined time
setParameter[name=value] : sets a given parameter
setParameters[name=value;name=value] : sets a given list of parameters
getParameter[name] : return the parameter from a key
loadData[Netcdf File] : load a landscape file, define a domain
startFire[location, date] : start a triangular fire front
include[command file] : include commands from a file
print[output file(optional)] : prints the state of the simulation
save[NetCDF file] : saves the simulation state in NetCDF format
clear[] : resets the simulation
quit : quit and terminate
```

2.1 Parametrisation

The first step in the definition of a simulation is to set a few parameters (otherwise setted as default), by typing, in the interpreter or script file:

```
setParameters[perimeterResolution=0.3;spatialIncrement=0.05]
setParameters[propagationModel=Iso;Iso.speed=1]
```

This sets the propagation model as the "Iso" isotropic model, with a velocity of $0.1m.s^{-}1$. Parameters can be setted indifferently in one line or by calling several time the setParameters command, possibly re-setting the value of an already defined parameter.

2.2 Initialisation

A second step is required to define the simulation domain (extents) :

```
FireDomain[sw=(-1000.,-1000.,0.);ne=(1000.,1000.,0.);t=0.]
```

This sets a domain with a south west corner at (x=-1000,y=-1000,z=0) a north east corner at (x=1000,y=1000,z=0), and at time= 0.

Finally a front, with any given number of markers is created with :

```
FireFront[t=0.]
    FireNode[loc=(6.801727,99.768414,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(42.251148,90.635757,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(71.403174,70.011333,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(95.697571,29.016805,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(96.262344,-27.084334,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(70.861606,-70.559427,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(28.648564,-95.808454,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(-20.024551,-97.974575,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(-68.66058,-72.70299,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(-96.548591,-26.045528,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(-94.783809,31.875218,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(-79.29024,60.934865,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(-56.08024,82.794967,0.);vel=(0.,0.,0.);t=0.]
```

Note the tabulation before the FireFront command, meaning is that the front is contained in the domain. The nature of the front (expanding or contracting) is given by the orientation between the markers. In the clockwise direction case the front is expanding, contracting in the counter-clockwise direction.

A front with inner front will be created by shifting the front with another tabulation (so it is contained by the predefined front) :

```
FireFront[t=0.]
    FireNode[loc=(6.801727,99.768414,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(42.251148,90.635757,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(71.403174,70.011333,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(95.697571,29.016805,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(96.262344,-27.084334,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(70.861606,-70.559427,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(28.648564,-95.808454,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(-20.024551,-97.974575,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(-68.66058,-72.70299,0.);vel=(0.,0.,0.);t=0.]
```

```
FireNode[loc=(-96.548591,-26.045528,0.);vel=(0.,0.,0.);t=0.]
FireNode[loc=(-94.783809,31.875218,0.);vel=(0.,0.,0.);t=0.]
FireNode[loc=(-79.29024,60.934865,0.);vel=(0.,0.,0.);t=0.]
FireNode[loc=(-56.08024,82.794967,0.);vel=(0.,0.,0.);t=0.]
FireFront[t=0.]
    FireNode[loc=(5,6,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(6,6,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(6,5,0.);vel=(0.,0.,0.);t=0.]
```

The direction of the front is provided by its winding, a clockwise front will expand with positive propagation speed, a counter-clockwise front will shrink with a positive propagation speed.

If two fronts were to be defined, starting at two different times, they must be aligned

```
FireFront[t=0.]
    FireNode[loc=(0,10,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(10,0,0.);vel=(0.,0.,0.);t=0.]
    FireNode[loc=(10,10,0.);vel=(0.,0.,0.);t=0.]
FireFront[t=0.]
    Fire Node[loc=(5,6,0.);vel=(0.,0.,0.);t=10.]
    FireNode[loc=(6,5,0.);vel=(0.,0.,0.);t=10.]
    FireNode[loc=(6,6,0.);vel=(0.,0.,0.);t=10.]
```

2.3 Simulation

Time is handled with events in ForeFire, so there is no such exact things as iteration number. In order to move the simulation forward in time, the step[] command will advance the simulation up to the absolute time given in parameter with:

```
goTo[t=500]
```

or advance by a relative time amount with.

```
step[dt=100s]
```

See examples in /examples directory for 1

2.4 Data input

In addition to the simple data that can be input form the interpreter, field data can be in putted as a netCDF file. Depending on the model used, this file may include elevation, fuel, modelmaps or any other field required for the model to be solved in the simulation.

Sample input files can be found in /examples directory. Generation of input file may be performed by the script in tools/genForeFireCase.py.

Typical fuel data is table/indices based, the table must be provided as an ascii comma separated value file (see examples/03Canyon/01forefire/fuels.ff)

2.5 Data output

ForeFire generate output (fronts) that is directly readable, and at the same format as the outputs. Fore scalar/Vector field data

The sample helper script "tools/FFtoVTK.py" can be used to transform this data into python readable format and export to legacy Vtk files.

Chapter 3

Library: coupled simulations

Coupled simulation has only been designed for the MESO-NH code, although it is likely that any Eulerian code that is parallelized in a domain decomposition manned should be able to be coupled with minimum effort.

3.1 Build

The legacy ForeFire shared library can be used with ISO C-Fortran bindings, it must be accessible to the MESONH executable (in the same directory or library path). The MESO-NH FOREFIRE user contains code that belongs to MesoNH and cannot be distributed freely. A version may be obtained upon request to filippi-at-univ-corse.fr.

A USER version of MNH is to compiled that enables to call the ForeFire library, impose the surface wind to the fire model and inject the fluxes stemming from the fire in the MNH simulation. Sources are availlable upon request to filippi at univ-corse.fr. In order to compile MesoNH using ISO C Fortran bindings one has to change the Makefile, *i.e.* adding the line iso c binding.mod at the end of the Makefile.

3.2 Run

In order to activate the coupling with ForeFire, user have to :

1. Put the name list NAM_FOREFIRE in the file EXSEG1.nam with the following variables :
   - LFOREFIRE : when .TRUE., activates the coupling with ForeFire.
   - COUPLINGRES : maximum resolution (in meters) for the coupling.

- NFFSCALARS : number of scalars that will be coupled (calculated from ForeFire)
- FFSVNAMES(NFFSCALARS) : name of scalars
- FFOUTUPS : backup frequency outputs in seconds.
- PHYSOUT : output physical variables ($moist, P, T, TKE$)
- FLOWOUT : output flow variables (U, V, W)
- CHEMOUT : output chemical variables

2. Create a set of directory :

   - one to save the evolution of the front ; this directory is usualy named ForeFire/ and also contains the 4 files case.NC, fuel.ff, init.ff, Param.ff (see chapter 3 for complete description),
   - n diectories named MODELn/, where n is the number of nested domains (the name can be changed in the file Param.ff) in order to save the outputs.

Part II

Controlling parameters

Chapter 4

Controlling parameters of a ForeFire simulation

ForeFire comes with a set of pre-defined parameters listed below. The user should be aware that ForeFire parameters are CASE SENSITIVE !

Table 4.1: Existing parameters.

Parameter	Default value	Description
caseDirectory	./	directory of the simulation (usually the directory of the atmospheric simulation).
ForeFireDataDirectory	./ForeFire/	directory containing all the data needed for a ForeFire simulation.
experiment	ForeFire	name of the simulation. Affects the outputs when saving the simulation.
NetCDFfile	data.nc	netCDF file containing the data needed by the simulation.
fuelsTableFile	fuels.ff	file containing the values of the parameters of the fuels.
paramsFile	Params.ff	file containing the values of the parameters controlling the simulation.
parallelInit	0	boolean for parallel initialization (restart from a previous atmospheric simulation).
InitFile	Init.ff	file containing the location of the fire front at initialization (non-parallel init).

Continued on next page

Table 4.1 – continued from previous page

Parameter	Default value	Description
InitFiles	output	pattern for the files containing the locations of the fronts during a parallel restart.
InitTime	0	time for the parallel restart. Files containing the information at that time should follow InitFiles.i.InitTime where i is the number of the processor that has saved its state in the previous simulation.
BMapsFiles	undefined	pattern of the files containing precomputed burning maps in case of restart or one-way couped simulations
SHIFT_ALL_ABSCISSA_BY	0	shift in all the abscissa of the data provided by the user, when needed in nested coupled atmospheric simulations.
SHIFT_ALL_ORDINATES_BY	0	shift in all the ordinates of the data provided by the user, when needed in nested coupled atmospheric simulations.
perimeterResolution	10	maximum distance allowed between two nodes discretizing the fire front.
spatialIncrement	0.2	distance the nodes discretizing the fronts are advanced each time they are updated.
spatialCFLmax	0.3	maximum allowed value in the spatial 'CFL like' number = spatialIncrement/perimeterResolution.
normalScheme	medians	scheme used during the computation of the medians (medians, weightedMedians and splines are available).
smoothing	5	smoothing parameter in the computation of the normal
relax	0.2	relaxation parameter in the computation of the normal and front depth
curvatureComputation	0	boolean for the computation of the front local curvature, eventually needed by some propagation models.
curvatureScheme	circumradius	scheme used during the computation of the front curvature (angles, circumradius ans splines are available).

Continued on next page

Table 4.1 – continued from previous page

Parameter	Default value	Description
frontDepthComputation	0	boolean for the computation of the front depth, eventually needed by some propagation models.
frontDepthScheme	normalDir	scheme used during the computation of the front depth (normalDir is up to now the only one available).
propagationModel	TroisPourcent	if the propagation models to be used are not provided in the 'NetCDFfile', solely one model is used: the one provided by the 'propagationModel' parameter.
burningTresholdFlux	500	threshold expressed in terms of radiated flux ($W.m^{-2}$) from the fire to determine if the location is still burning (needed when computing the front depth).
minimalPropagativeFrontDepth	1	limiting distance (in m) for the propagation of the fire. If the front depth is below this limit, the fire is considered inactive.
maxFrontDepth	20	maximum depth (in m) reachable by the front. When superior to this limit, effects of front depth on the propagation are considered constants.
initialFrontDepth	1	depth (in m) of the front at initialization.
initialBurningTime	30	duration of the burning process (in s) inside the initial front.
atmoNX	100	number of atmospheric cells in the longitudinal direction. Atmospheric cells determines the regions over which the fluxes stemming from the fire are computed. This parameter is used in a ForeFire when not coupled to an atmospheric simulation, when coupled to an atmospheric simulation it is imposed by it.
atmoNY	100	number of atmospheric cells in the latitude direction.

Continued on next page

Table 4.1 – continued from previous page

Parameter	Default value	Description
atmoNZ	20	number of atmospheric cells in the vertical direction.
refLongitude	0	reference longitude of the Fore simulation (imposed by th atmospheric simulation when coupling).
refLatitude	0	reference latitude of the Fore simulation (imposed by th atmospheric simulation when coupling).
year	2012	year of the Fore simulation (imposed by th atmospheric simulation when coupling).
month	1	month of the Fore simulation (imposed by th atmospheric simulation when coupling).
day	1	day (relative to the month) of the Fore simulation (imposed by th atmospheric simulation when coupling).
SWCornerX	-10	abscissa of the southwest corner (for uncoupled simulations).
SWCornerY	-10	ordinate of the southwest corner (for uncoupled simulations).
NECornerX	10	abscissa of the northeast corner (for uncoupled simulations).
NECornerY	10	ordinate of the northeast corner (for uncoupled simulations).
watchedProc	-2	number of the MPI processor to be watched when debugging. If watchedProc = -2 no output from any processor is done. If watchedProc=-1, outputs from all processors are done. If watchedProc=n, only outputs from processor with mpi rank n are done.
CommandOutputs	1	boolean for printing debug outputs (in the standard output) related to the commands.
FireDomainOutputs	1	boolean for printing debug outputs (in the standard output) related to the behaviour of the domain of the simulation (parallel processing, event handling, ...).

Continued on next page

Table 4.1 – continued from previous page

Parameter	Default value	Description
FireFrontOutputs	1	boolean for printing debug outputs (in the standard output) related to the behaviour of the fire fronts (topology, ...).
FireNodeOutputs	1	boolean for printing debug outputs (in the standard output) related to the behaviour of the lagrangian nodes (speed and normal computation, ...).
FDCellsOutputs	1	boolean for printing debug outputs (in the standard output) related to the atmospheric cells (flux computation, ...).
HaloOutputs	1	boolean for printing debug outputs (in the standard output) related to the halos (packing of parallel data, ...).
fireOutputDirectory	./ForeFire/	directory where the fire outputs will be made.
atmoOutputDirectories	./ForeFire/	directories where the atmospheric outputs will be made. This parameter can be a multi-valued parameter if several atmospheric models are used (ex: './MODEL1/,./MODEL2/')
outputFiles	output	pattern the names of the files containing the ForeFire outputs. ForeFire automatically adds the number of the processor responsible for the output, the name of the variable and the time.
outputsUpdate	0	frequency (in s) of the saving outputs. If outputsUpdate=0, no saving outputs are made. If outputsUpdate=n, saving outputs are made each n seconds.
debugFronts	0	boolean to save fronts each atmospheric step (files are overwrited each time the process is done).
surfaceOutputs	0	boolean for saving the surface properties, *i.e.* fluxes stemming from the fire.

Continued on next page

Table 4.1 – continued from previous page

Parameter	Default value	Description
bmapOutputUpdate	0	frequency (in s) of the saving outputs for the burning map. If bmapOutputsUpdate=0, no saving outputs are made. If bmapOutputsUpdate=n, saving outputs are made each n seconds. This process is really heavy (storing data of a huge matrix...) and should be used only in dire cases.

Chapter 5

Associated parameters in a MNH simulation

Part III

Models in ForeFire

Chapter 6

What are models in ForeFire made for?

ForeFire is a scientific code suitable for propagating an interface on a surface, *i.e.* a fire front on the ground. The propagation is detemined by a propagation model that should, according to several local properties and assumptions, give the velocity in the normal direction to the current front. In return ForeFire can determine at each time the location of the interface as well as many physical/chemical fluxes that influence the behaviour of the surrounding atmosphere. Models in ForeFire are then of major importance in two ways:

- Propagation models give the behaviour of the propagation of the interface,
- Flux models enable to compute the considered physical/chemical flux on a given surface.

Chapter 7

Layers

All ForeFire models are related to the computation of a specific property, either the propagation velocity or some flux. Several models can then have the same intent (*i.e.* aim to compute the same property but in different manners). Suppose for example you have different areas in your domain where the released heat flux has different behaviours. This might happen if you have an interface between a wildland fire and urban areas. Indeed the heat release during the combustion process in vegetation and construction material do not present the same behaviour. Another example is given by a volcanic eruption where the SO_2 emissions are different above the crater or above the lava.

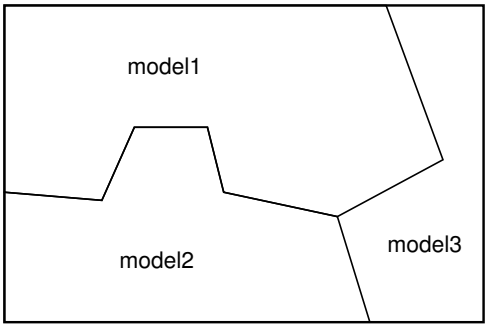

Figure 7.1: Different models can be used in different areas

The purpose of a layer is to gather all the information required for the computation of a specific property (propagation velocity or some flux) and enable to use the desired model at the desired location. One unique layer should then be defined for each specific property you want to compute. It is of particular importance for the layer responsible of the propagation models as it is the core of the interface tracking by ForeFire.

Important: for each specific property you want to compute with ForeFire one has to define an associated layer. In all cases it is mandatory to define the propagative layer (layer of the propagation models).

7.1 map of models

Layers are designed to meet the need to apply different models depending on the location where the propagation velocity and/or the flux is computed. The user can define beforehand the areas where each separate model should be used, as shown in figure 7.1. This process defines the map of models that should be used during the ForeFire simulation for a given property.

Each model is affected with an index and the layer stores a map of the indexes of the models to be used (see fig. 7.2). Two models of the same type (propagation or flux) should not have an identical index, but a propagation model can have the same index as a flux model. This limitation stems from the inside machinery of ForeFire (a table of correspondance are defined for propagation models and another one for all the flux models)

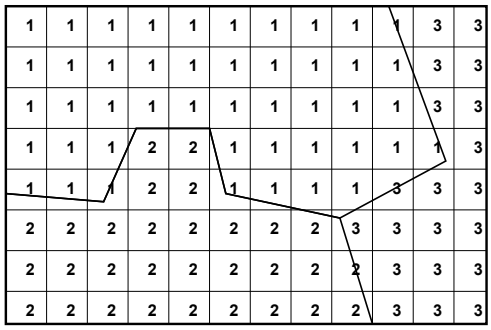

Figure 7.2: Discretization of the domain for the map of models. Each model is given an index.

Important: each model should have a unique index. The user has to be very cautious when defining these indexes as two models of the same type (for examples flux models such as a model A for heat flux and model B for vapor flux) should never have the same index even if computed properties are different.

7.2 how can I define the layers needed for my simulation?

The only present-day way to define a tailored layer with a specified map is through the use of a netCDF file. An example of the structure of a layer defined in the netCDF file should be as following (as verbatim of the result of a ncdump):

Listing 7.1: example of layer definition in a netCDF file

```
netcdf case {
dimensions:
        nx = ... ;
        ny = ... ;
        nz = 1 ;
        nt = ... ;
        domdim = 1 ;
variables:
        char domain(domdim) ;
                domain:type = "domain" ;
                domain:SWx = ... ;
                domain:SWy = ... ;
                domain:SWz = 0 ;
                domain:Lx = ... ;
                domain:Ly = ... ;
                domain:Lz = 0 ;
                domain:t0 = 0 ;
                domain:Lt = ... ;
        int heatFlux(nx, ny, nz, nt) ;
                heatFlux:type = "flux" ;
                heatFlux:indices = 0, 1, 2 ;
                heatFlux:model0name = "heatFluxNominal" ;
                heatFlux:model1name = "HeatFluxBasic" ;
                heatFlux:model2name = "BurnupHeatFlux" ;
data:
...
```

First the domain defines the extension of the layer, with respect to space and time (SW stands for the SouthWest point coordinates, L for the lenghts of the domain, t0 for the initial time and Lt for the duration).

Then a layer is defined by its name (here heatFlux), its type (flux), the indices of its models $(0, 1, 2)$ and the associated names of these models. These names are defined along with the indices, *i.e.* one has to name each modelNname with the effectiv name of the model in ForeFire. Several layers can be defined on the same domain (usually the domain of your simulation), but be careful about the indexes of models of the same type.

Finally the map of indexes for each layer is stored in the data section (see netCDF user's guide).

Important: each model defined in the netCDF should have a counterpart in ForeFire, which means that a model of the right type has been implemented in ForeFire and its name is the same as defined in the netCDF file.

The easiest way to define the complete netCDF file (with all the different layers you want to use: propagative, heat flux, vapor flux, ...) is by far to use the Java program developed by Jean-Baptiste Filippi that will automatically generate the file according to your needs.

7.3 what happens if I didn't define my layer beforehand in a netCDF file?

When only one model is needed for the whole domain it is a bit tedious to go through the whoe process of defining the layer in a netCDF file. Depending if it concerns propagation or flux there exist two way to define layers more easily:

- if you want to use only one propagation model you can choose it directly in the parameters file of ForeFire (usually Params.ff). ForeFire will then construct automatically the propagative layer using the desired model (obviously is the desired model is implemented in ForeFire with the right name),
- if you want to define a flux layer with only one model (for example in case of homogeneous vegetation) you can call the CheckLayer function (see 12) with the desired name of the model. This will automaticlly create a layer with the desired name and using the desired model everywhere.

7.4 do I have to do anything else for the layers?

No. As long as you have well-defined counterparts in ForeFire for all the models defined in the netCDF file the rest is handled automatically by ForeFire.

Chapter 8

Common properties and behaviours of ForeFire models

All ForeFire models share common properties and behaviours, be it a propagation or a flux model. Let's take a look at the class ForeFireModel important attributes:

Listing 8.1: FluxModel.h

```
class ForeFireModel {
protected:
        DataBroker* dataBroker;
        SimulationParameters* params;
        size_t registerProperty(string);
public:
        int index;
        size_t numProperties;
        vector<string> wantedProperties;
        size_t numFuelProperties;
        vector<string> fuelPropertiesNames;
        FFArray<double>* fuelPropertiesTable;
        double* properties;
};
```

A ForeFire model has an index (stored in 'index') and the table of properties needed to compute the desired property (stored in 'properties'). Each of these properties have a name stored 'wantedProperties', but some of them comes from the fuel. The values of these desired fuel properties are stored in another table 'fuelPropertiesTable' for the sake of ForeFire internal machinery; this should never be changed by the user and is handled transparently and automatically by ForeFire.

A ForeFire model is also related to a data broker (which will responsible of getting the values of the properties at the desired locations and times,), to the simulation parameters (which means it can access any variable defined in ForeFire parameters file, which is usually Params.ff) and finally each model has a function to register the property it will need. This function is essential to the internal machinery of ForeFire and is the entry point to the user in case he wants to define a new model (see 11).

Chapter 9

Propagation handling in ForeFire

9.1 Propagative layer

Propagation of the interfaces in ForeFire are carried through the advection of lagrangian markers which velocity is given by the rightful propagation model at the location. The propagative layer thus contains the information on the map of the propagation models to be used. If no map is defined in a netCDF file, the model defined in the parameters file is used on the whole domain.

9.2 Propagation model

A propagation model is used in ForeFire to determine the velocity of the lagrangian markers discretizing the interfaces. It is the code tanscription of some empirical/physical modeling of the behaviour of these interfaces. Let's take a simple example of a propagation velocity depending on the moiture content of the vegetation and the wind in normal direction (normal to the front). The mathematical transcription of this physical model could be:

$$v_i = A\frac{\vec{u}(\vec{x}_i).\vec{n}_i}{1+m(\vec{x}_i)}, \tag{9.1}$$

where $\vec{x}_i$, v_i and $\vec{n}_i$ are the location, velocity and normal of the i^{th} lagrangian marker, $\vec{u}$ the ground wind, m the moisture of the vegetation and A a fitted coefficient. $\vec{u}$ and m are assumed to vary in space and are thus taken at the marker's location.

A ForeFire propagation model can take into account this physical model in the ForeFire simulation as following.

9.3 pattern of a propagation model

The pattern of a propagation model is available within the files 'PropagationModel.h' and 'PropagationModel.cpp' and incorporates all the common properties and behaviours of a ForeFire model (see 8).

Listing 9.1: PropagationModel.h

```
class FluxModel {
        PropagationModel(const int& = 0, DataBroker* = 0);
        virtual string getName(){return "stub propagation model";}
        double getSpeedForNode(FireNode*);
        virtual double getSpeed(double*){return 0.;}
};
PropagationModel* getDefaultPropagationModel(const int& = 0
            , DataBroker* = 0);
```

Listing 9.2: PropagationModel.cpp

```
PropagationModel* getDefaultPropagationModel(const int & mindex
            , DataBroker* db) {
        return new PropagationModel(mindex, db);
}
PropagationModel::PropagationModel(const int & mindex
            , DataBroker* db)
: ForeFireModel(mindex, db) {
}
double PropagationModel::getSpeedForNode(FireNode* fn){
        dataBroker->getPropagationData(this, fn);
        return getSpeed(properties);
}
```

the getSpeedForNode function

The getSpeedForNode function is the one called for each lagrangian marker (called FireNodes in ForeFire) to determine its velocity. It takes the lagrangian marker as argument and returns its propagation velocity. As one can see in 'PropagationModel.cpp' this function calls in its turn:

1. the data broker in order to retrieve all the wanted properties at the location of the marker,
2. the getSpeed function that is the actual funcion implementing the model.

> **Important**: the getSpeedForNode function shouldn't be modified, moreover it should not appear in a new propagation model implementation.

the getSpeed function

The getSpeed function is one translating into code the mathematical formulation 9.1. It is then different for every propagation model but its structure is the same. It takes the table of values of the different properties needed by the model and returns the computed velocity. Let's see how this is done in our example:

Listing 9.3: getSpeed

```
double Model::getSpeed(double* valueOf){
        return A*valueOf[normalWind]/(1.+valueOf[moisture]);
}
```

This line of code looks simple and straightforward from the mathematical equation we want to use. This feature is enabled by the fact that the model ask beforehand the data broker what are the values at the marker's location. These values are automatically stored in the table 'valueOf' and the value of each specific property can be retrieved easily by the name chosen for that property (for definition and initialization of these properties refer to 11).

Chapter 10

Flux handling in ForeFire

10.1 Flux layer

Fluxes in ForeFire are handled by flux layers (see 'FluxLayer.h' for the general framework). These layers contains two main information:

1. the information about the atmospheric mesh (number, size and location of the atmospheric cells),
2. the map for the models that are to be used in each part of the domain.

These flux layers are used to compute the fluxes needed in the coupling of ForeFire with an atmospheric model. Most of the times these fluxes are reduced to the heat and vapor fluxes (plus an optional tracer flux for visualization) but anyone can define a new flux layer to handle other physical/chimical processes. The user should be aware that doing this job in ForeFire requires to do the counterpart in the atmospheric model to take into accont the desired new phenomenom.

> The user should only be taking care of the map of the different models as well as their implementation in ForeFire, the information about the atmospheric model being automatically handled by ForeFire.

A flux layer in ForeFire has all the relevant information (as far as flux is concerned) about the atmospheric mesh, *i.e.* number, size and location of the atmospheric cells. ForeFire also defines an underlying finely resolved map of the times of arrival of the fire front (see fig. 10.1). The resolution of this so-called burning map is at least 10 times better than the atmospheric resolution.

The flux for a given atmospheric cell is then computed by summing all the fluxes in the cells of the map of arrival times. The flux in each of these cells is determined by the following process:

1. determine which model is to be used at the location of the cell,

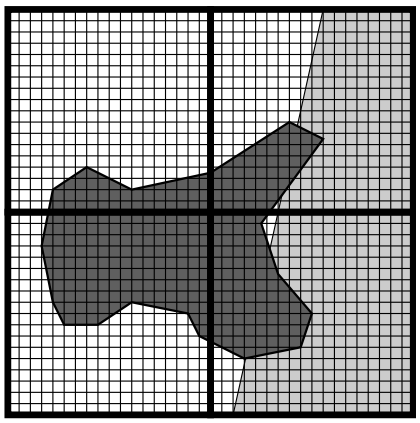

Figure 10.1: Four cells of the amospheric model (wide lines) with the contour of the fire front (dark grey). The underlying map of arrival times has a better resolution than the atmospheric cells (narrow lines), and different models can be applied in several regions (for example model A in the light gray region and model B everywhere else).

2. gather all the information needed by the model (for example vegetation, wind, temperature, current time, arrival time of the fire ...),
3. apply the flux model with these parameters.

In the following each of these steps will be further explained.

10.2 Flux Model

A flux model is used in ForeFire to compute some specific properties such as heat flux, chemical flux, ... and is the transcription into code of these physical/-chemical models. Let's take a simple example. One would want to implement in ForeFire a heat flux model that has the following properties:

- the flux is null when the location is not burning,
- the flux is given by an empiric value A when the location is burning,
- as soon as the fire touches a location, that location will be burning for a time $\tau = \tau_0/s_d$, where τ_0 and s_d are properties of the vegetation.

The mathematical transcription of this physical model is:

$$\phi(t, t^i) = A\mathcal{H}_{[0,\tau_0/s_d]}(t - t^i), \tag{10.1}$$

where ϕ is the heat flux, t^i the ignition time (arrival time of the fire), $\mathcal{H}$ the heaviside function and t the current time.

A ForeFire flux model is here to implement this formula into the ForeFire simulation.

pattern of a flux model

The pattern of a flux model is available within the files 'FluxModel.h' and 'FluxModel.cpp' and incorporates the general pattern of a ForeFire model (see 8).

Listing 10.1: FluxModel.h

```
class FluxModel {
        FluxModel(const int& = 0, DataBroker* = 0);
        virtual string getName(){return "stub flux model";}
        double getValueAt(FFPoint&, const double&, const double&);
        virtual double getValue(double*, const double&
              , const double&){return 1.;}
};
FluxModel* getDefaultFluxModel(const int& = 0
              , DataBroker* = 0);
```

Listing 10.2: FluxModel.cpp

```
FluxModel* getDefaultFluxModel(const int & mindex
              , DataBroker* db) {
        return new FluxModel(mindex, db);
}
FluxModel::FluxModel(const int & mindex
              , DataBroker* db)
        : ForeFireModel(mindex, db) {
}
double FluxModel::getValueAt(FFPoint& loc
              , const double& t, const double& at){
        dataBroker->getFluxData(this, loc, t);
        return getValue(properties, t, at);
}
```

the getValueAt function

The getValueAt function is the one called for each cell of the map of arrival times. It takes as arguments the location, the present time and the arrival time of the fire at that location. As one can see in "FluxModel.cpp" this function calls in its turn the getValue function after the data broker has gathered the needed properties (dataBroker→getFluxData(this, loc, t)).

Important: the getValueAt function shouldn't be modified, moreover it should not appear in a new flux model implementation.

the getValue function

The getValue function is the one actually implementing the physical model. It takes the table of values of the different properties needed by the model and returns the desired flux. Let's see how this is done in our example:

Listing 10.3: getValue

```
double FluxModel::getValue(double* valueOf
                , const double& t, const double& at){
        if ( t-at < valueOf[tau0]/valueOf[sd] ) return A;
        return 0.;

}
```

This line of code looks simple and straightforward from the mathematical equation we want to use. This feature is enabled by the fact that the model ask beforehand the data broker what are the values at the marker's location. These values are automatically stored in the table 'valueOf' and the value of each specific property can be retrieved easily by the name chosen for that property (for definition and initialization of these properties refer to 11).

This model simply returns what the physical model specified:

- the wanted amplitude if the difference between the current time t and the arrival time of the fire at that location at is inferior to the computed burning duration at that location,
- zero if the location is not burning.

Chapter 11

Implementing a new model in ForeFire

We will now see how to implement a new model by constructing the two files needed to represent the basic heat flux defined by eq. (10.1).

11.1 conformity with the pattern

The first step is to copy both "FluxModel.h" and "FluxModel.cpp" into new files with explicit names, for example here "HeatFluxNominalModel.h" and "HeatFluxNominalModel.cpp".

> **Important**: all the occurences of "PropagationModel" or "FluxModel" in the files should be replaced by the new name given to the model, in our example "HeatFluxNominalModel".

11.2 give a name to the model

Then one has to give a name to the model in accordance to the names given in the netCDF file for the map of models (see § 7.2). This is done in two steps:

1. declare the name in the .h file as in:

```
static const string name;
```

2. give the name in the .cpp file as in:

```
const string HeatFluxNominalModel::name = "heatFluxNominal";
```

11.3 register the model to ForeFire

ForeFire simulator should have a notice that you created a new model that might be used in a ForeFire simulation. In order to send this information a new model should declare a static variable (see C++ documentation) and initialize it as following:

1. declare the initialization variable in the .h file as in:

```
static int isInitialized;
```

2. initialize it in the .cpp file as in:

```
int HeatFluxNominalModel::isInitialized =
FireDomain::registerFluxModelInstantiator(name,
getHeatFluxNominalModel );
```

N.B.: Registration for propagation models is done by registerPropagation-ModelInstantiator instead of registerFluxModelInstantiator.

11.4 declaring and registering the properties needed by the model

A flux model should need several properties such as the vegetation properties, the wind or the temperature atmosphere. Such properties that do not depend on the modeling itself are called external properties. ForeFire is able to handle external properties on its own and the user that wants to define a new flux model should only focus on the declaration and definition of these properties. For each wanted property it is a two step process. Let's take the example of τ_0 in our model. It is a property of the fuel so it should be defined the prefix 'fuel.'.:

1. first I declare the property in the .h file as in:

```
static const size_ t tau0;
```

   where I can choose the name of the property. This name will be the reference when you want to use the value of this property. For instance when I will need this property I can its value at the desired location by calling 'valueOf[tau0]'. If I have called it only 'tautau', I would have to call 'valueOf[tautau]'.

2. then I register this property in the .cpp file. The property is regitered in the constructor of the model (see other models) as in:

tau0 = registerProperty("fuel.Tau0");

where the name on the right hand side ("fuel.Tau0") is the name that is valued in the ForeFire fuel file (usually 'fuels.ff').

> **Important**: The fuel properties are particular in the sense that the values are read in the fuel properties' file of ForeFire, usually 'fuels.ff'. The user should be well aware that if he wants to use a fuel property in its model the values for all the fuel in the domain should be informed in the fuels.ff file.

You can define and register as many properties as needed by the model as long as their names do not overlap.

Currently available properties for propagation models

There are actually several properties already implemented in ForeFire which can be used directly in any propagation and flux model:

fuel.X	gets the required fuel X property
altitude	gets the altitude
slope	gets the slope in the outward normal direction to the front
windU	gets the longitudinal wind from an atmospheric model
windV	gets the transversal wind from an atmospheric model
normalWind	gets the wind in the outward normal direction to the front
frontDepth	gets the depth of the front in the normal direction
curvature	gets the curvature of the front

Table 11.1: Currently available properties for propagation models.

Currently available properties for flux models

fuel.X	gets the required fuel X property
altitude	gets the altitude
windU	gets the longitudinal wind from an atmospheric model
windV	gets the transversal wind from an atmospheric model

Table 11.2: Currently available properties for flux models.

These properties are those usually used in our models. Once again if it does not fit your needs feel free to contact the ForeFire team for hand-tailored solutions.

11.5 defining coefficients for the model

The vast majority of physical/chemical models rely on one or more coefficients. To ease the use of such coefficients you can declare this coefficients in ForeFire parameters' file (usually Params.ff) and use it in your model. The best way to deal with this is to declare and define this coefficients in a two-step process:

1. declare the coefficient in the .h file as in:

   double nominalHeatFlux;

   where I can choose the name of the coefficient. This name will be the reference when you want to use the value of this coeffiecient, simply by calling it.

2. then I define this coeffiecient in the constructor of the model (see other models). First I give a default value, then I check if this coefficient is valued in the parameters' file. If so, I give this new value to the coefficient. In our example this gives:

   nominalHeatFlux = 150000.;
   if (params-¿isValued("nominalHeatFlux"))
   nominalHeatFlux = params-¿getDouble("nominalHeatFlux");

   where 'params' is the reference to the simulation parameters.

This simple process allows you to change the value of the desired coefficient simply by changing its value in the parameters' file. For example if I want to change the nominal heat flux of my model I would simply need to had the line 'setParameter[nominalHeatFlux=...]' in the parameters' file.

11.6 defining the 'geSpeed' or 'getValue' function

After defining all these variables and properties it is now time to use them and effectively implement our model. This is done in the 'getSpeed' (for propagation models) or the 'getValue' (for flux models). Coefficients defined earlier are directly reachable by their names, and variable values at the desired location are reachable through the table of double in argument of the function (usually 'valueOf'). In our example this gives the simple code:

Listing 11.1: the getValue for the 'HeatFluxNominal' model

```
double HeatFluxNominalModel::getValue(double* valueOf
                , const double& t, const double& at){
        if ( t-at < valueOf[tau0]/valueOf[sd] )
                return nominalHeatFlux;
        return 0.;

}
```

Chapter 12

API: which functions can be called to get the computed values?

Now that you have all these layers and models ready to compute the desired properties, how can you retrieve it. Most of the time this will have to be done when coupling ForeFire with a meso-scale atmospheric model.

Part IV

Example Cases

Chapter 13

Isotropic circle

Chapter 14

Two circles

Chapter 15

Fire simulation of a canyon case

Chapter 16

Coupled simulation of a canyon case

Chapter 17

nested coupled fire/atmosphere simulations