State and Solution

State machines and manipulating the solution process with swak4Foam

Bernhard F.W. Gschaider

HFD Research GesmbH

Exeter, United Kingdom, Europe
24. July 2017

Outline I

Outline II

Outline

Outline

What is it about

- This is an advanced `swak4Foam` presentation
  - I won't explain the very basic things
- It shows how swak4Foam can be used to influence the solution
  - Either by changing "only" the numerics
  - or the physical solution
- One tool we will use are the rather new *State machines*
- We will modify three standard tutorials
  1. Changing the numerics during the run to improve the run-time
  2. Checking for convergence of the *phyiscal* solution instead of only the residuals and stopping the run depending on it
  3. Prototyping a simple physical model without writing a proper solver for it

How this presentation is to be used

- Intended audience
  - People who already worked with OpenFOAM
    - Know for instance how to modify the `system/controlDict`
  - Basic knowledge of swak4Foam would be nice
    - But if you've never used it and the presentation motivates you to check out: great
- This presentation tries to be as self-contained as possible
  - Theoretically you can work through it on your own
    - All the relevant changes are spelled out on the slides
  - I will present it as a 1.5h "lecture"
    - Too fast to redo the examples
    - You are encouraged to try the examples afterwards on your own
- The finished cases will be available in the `Examples/FromPresentations` folder of the swak4Foam sources
  - Names will start with `OFW12_`

Outline

Bernhard Gschaider

- Working with OpenFOAM™ since it was released
  - Still have to look up things in `Doxygen`
- I am not a core developer
  - But I don't consider myself to be an *Enthusiast*
- My involvement in the OpenFOAM™-community
  - Janitor of the `openfoamwiki.net`
  - Author of two additions for OpenFOAM™

    swak4foam Toolbox to avoid the need for C++-programming

    PyFoam Python-library to manipulate OpenFOAM™ cases and assist in executing them
  - In the admin-team of `foam-extend`
  - Organizing committee for the *OpenFOAM™ Workshop*
- The community-activies are not my main work but *collateral damage* from my real work at ...

Heinemann Fluid Dynamics Research GmbH

The company

- Subsidary company of *Heinemann Oil*
  - Reservoir Engineering
  - Reservoir management

Description

- Located in Leoben, Austria
- Works on
  - Fluid simulations
    - OpenFOAM™ and Closed Source
  - Software development for CFD
    - mainly OpenFOAM™
- Industries we worked for
  - Automotive
  - Processing
  - ...

Outline

Contents

What is PyFoam

- `PyFoam` is a library for
  - Manipulating `OpenFOAM`-cases
  - Controlling `OpenFOAM`-runs
- It is written in Python
- Based upon that library there is a number of utilities
  - For case manipulation
  - Running simulations
  - Looking at the results
- All utilities start with `pyFoam` (so TAB-completion gives you an overview)
  - Each utility has an online help that is shown when using the `--help`-option
  - Additional information can be found
    - on `http://openfoamwiki.net`

What is swak4Foam

From `http://openfoamwiki.net/index.php/Contrib/swak4Foam`

> `swak4Foam` stands for SWiss Army Knife for Foam. Like that knife it rarely is the best tool for any given task, but sometimes it is more convenient to get it out of your pocket than going to the tool-shed to get the chain-saw.

- It is the result of the merge of
  - funkySetFields
  - groovyBC
  - simpleFunctionObjects

  and has grown since
- The goal of swak4Foam is to make the use of C++ unnecessary
  - Even for complex boundary conditions etc

The core of swak4Foam

- At its heart swak4Foam is a collection of parsers (subroutines that read a string and interpret it) for expressions on OpenFOAM-types
  - fields
  - boundary fields
  - other (faceSet, cellZone etc)
- ... and a bunch of utilities, function-objects and boundary conditions that are built on it
- swak4foam tries to reduce the need for throwaway C++ programs for case setup and postprocessing

Building from smaller blocks

- Most of swak4foam are small, dynamically loadable parts
  - function objects
  - boundary conditions
  - fvOptions
- Each of them is quite limited in what it can do
- But they can pass information to each other
  - Through fields
  - Global variables
  - other things
- By using that quite complex applications can be built
  - It is a bit like programming

Definitions

Typical building blocks we'll use are

function objects small programs that are executed at the end of each time-step

fvOptions small programs that are used to modify the matrix and/or the solution at times specified by the solver

boundary conditions setting values on the boundary. Usually before a field is solved

function plugins these extend the swak4Foam-parser with special functions

- either not of general use
- or won't work in most solvers (for instance: because they require a radiation model)
- There is a presentation `swak4Foam for programmers` that demonstrates how to write your own functions

Outline

Command line examples

- In the following presentation we will enter things on the command line. Short examples will be a single line (without output but a ">" to indicate *input*)

```
> ls $HOME
```

- Long examples will be a grey/white box
  - Input will be prefixed with a > and blue
  - Long lines will be broken up
    - A pair of <brk> and <cont> indicates that this is still the same line in the input/output
  - «snip» in the middle means: "There is more. But it is boring"

Long example

```
> this is an example for a very long command line that does not fit onto one line of the slide <brk>
      <cont> but we have to write it anyway
first line of output (short)
Second line of output which is too long for this slide but we got to read it in all its glory
```

Used Foam version

- The examples here were derived from the tutorials in `OpenFOAM+ v1612+`
  - And calculated with that
- Equivalent tutorials from `OpenFOAM 4.1` should work as well
- `foam-extend` would need some modification
  - Due to differences in the dictionaries
  - But the principles apply as well
  - The third example definitely won't work
    - Because there are no `fvOptions` in `foam-extend`

Outline

Outline

The problems

- Some machines need more than one boundary conditions
  - Valves open and close
  - Heaters switch on and off
- These boundaries switches may depend on the state of the simulation
  - Pressure/temperature/etc goes above/below a certain threshold
  - Time has passed since an event
  - ...
- Adding such states to a simulation requires programming
  - Special solver
  - elaborate boundary conditions
- Programming should be avoided
  - it only leads to errors and heartache
    - especially in C++

The problems

- Some machines need more than one boundary conditions
  - Valves open and close
  - Heaters switch on and off
- These boundaries switches may depend on the state of the simulation
  - Pressure/temperature/etc goes above/below a certain threshold
  - Time has passed since an event
  - ...
- Adding such states to a simulation requires programming
  - Special solver
  - elaborate boundary conditions
- Programming should be avoided
  - it only leads to errors and heartache
    - especially in C++

Solution in swak4Foam (until now)

Implementing states in `swak4Foam` involved

- Function objects to create *global variables*
  - Variables that could be read in other function objects and boundary conditions
- Function objects that manipulated these global variables
- Function objects that executed depending on some conditions
- Boundary conditions that read these global variables
- and/or *stored variables*
  - Variables that "remembered" their states

It was a bit of a hack

- Hard to maintain
- Hard to understand

But at least it didn't require C++

Solution in swak4Foam (until now)

Implementing states in `swak4Foam` involved

- Function objects to create *global variables*
  - Variables that could be read in other function objects and boundary conditions
- Function objects that manipulated these global variables
- Function objects that executed depending on some conditions
- Boundary conditions that read these global variables
- and/or *stored variables*
  - Variables that "remembered" their states

It was a bit of a hack

- Hard to maintain
- Hard to understand

But at least it didn't require C++

Solution in swak4Foam (until now)

Implementing states in swak4Foam involved

- Function objects to create *global variables*
  - Variables that could be read in other function objects and boundary conditions
- Function objects that manipulated these global variables
- Function objects that executed depending on some conditions
- Boundary conditions that read these global variables
- and/or *stored variables*
  - Variables that "remembered" their states

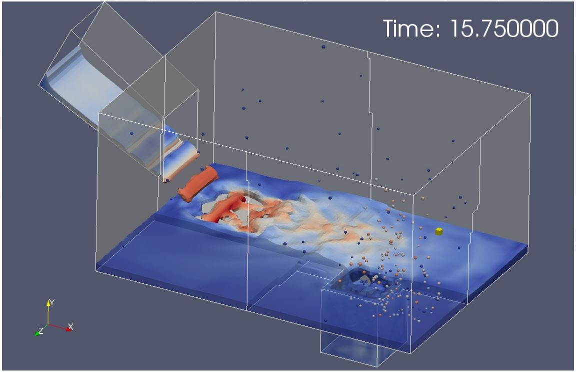
It was a bit of a hack

- Hard to maintain
- Hard to understand

But at least it didn't require C++

Example from OSCIC 2012 in London

- This example switched a number of things on and off with global variables
- In the swak-distribution: `Examples/FromPresentations/OSCFD_cleaningTank3D` (and 2D)



Outline

Definition of State machines

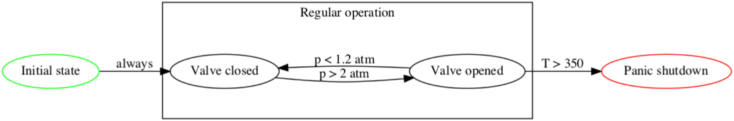
Stolen from Wikipedia:

- A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), or simply a state machine, is a mathematical model of computation used to design both computer programs and sequential logic circuits.
- It is conceived as an abstract machine that can be in one of a finite number of states.
- The machine is in only one state at a time
  - the state it is in at any given time is called the current state.
- It can change from one state to another when initiated by a triggering event or condition
  - this is called a transition.
- A particular FSM is defined by
  1. a list of its states,
  2. its initial state
  3. the triggering condition for each transition.

Example

State machine model for a valve

- 4 States: `Initial state`, `Valve opened` `Valve closed` and `Panic shutdown`
  - Represented by the circles
- Initial state is `Initial State`
- Transitions represented by the arrows
  - Condition written next to the arrow (in our case pressure thresholds trigger switches)
- `Panic dump` is a *Final State* (no transitions out of it)
  - Not necessary for a state machine



Outline

Add state machines to swak4Foam

- All things necessary are in one library
  - Names start with `stateMachine`
- Function object to create and update a State machine
- Function plugins to access them in expressions
- Other function objects to manipulate and write the state of the the State machine

controlDict

```
libs (
    "libswakStateMachine.so"
);
```

Specification of a state machine

The `stateMachineCreateAndUpdate` function object specifies a state machine

machineName name of the machine

states list of possible states

initialState state to start in

transitions list of dictionaries that specify transitions

- **from** source state (state the machine is currently in)
- **condition** expression with the condition that has to be true
- **logicalAccumulation** does `condition` have to be true only once (`or`) or everywhere (`and`)
- **to** state to move to if `condition` is true
- **description** Text to print if transition "fires"

Other typical swak-parameters like `valueType` and `variables` can also be specified

"Driving" the state machine

- `stateMachineCreateAndUpdate` is "executed" once every timestep
  - `transitions` where `from` is the current state are checked
  - They are evaluated in the order they are in the list
    - The first one that evaluates to `true` is used
    - Transition to state `to`
    - Record time of transition
  - If no `transition` "fires" machine stays in current state
- Function object `stateMachineSetState` unconditionally moves machine to a state
  - To be used in conditional function objects (`executeIf`)
- `stateMachineMachineState` writes the current state of the machine to a file
- State of the machine is written at every output time and will be used for a restart of the simulation

Functions for state machines

These functions can be used everywhere a logical expression is acceptable

`stateMachine_isState(machine,state)` true if the machine named `machine` is currently in the state `state`

`stateMachine_timeSinceChange(machine)` time since the `machine` changed into the current state (to implement conditions like "How long has the valve been open")

`stateMachine_stepsSinceChange(machine)` number of time steps since the last state change of `machine`

`stateMachine_changedTo(machine,state)` How many times has the `machine` changed to `state` (for conditions like "How often did the valve open")

Outline

Outline

The sonicFoam case nacaAirfoil

- We will use a standard tutorial

```
> pyFoamCloneCase.py $FOAM_TUTORIALS/compressible/sonicFoam/RAS/nacaAirfoil <brk>
    <cont>nacaAirfoilControlled
> cd nacaAirfoilControlled
```

- This case simulates an airfoil in a high Mach-number flow field
- Mesh was generated with a third-party tool
- Cell sizes differ significantly
- Next slides show the mesh
    - Yes: it is oriented that way

Overview of the geometry

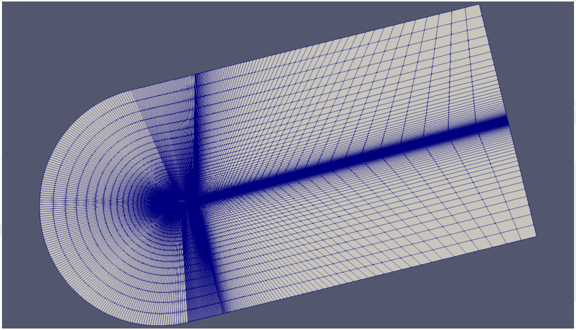


Figure: The whole geometry

Close-up on the foil

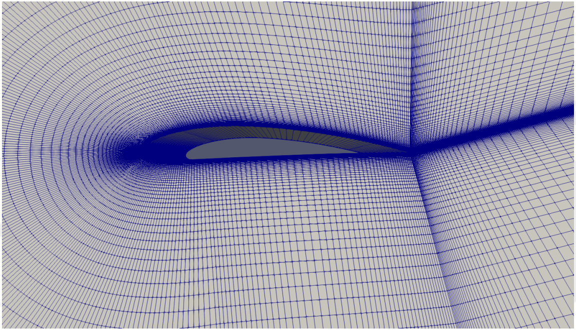


Figure: The actual foil

Extent of the foil

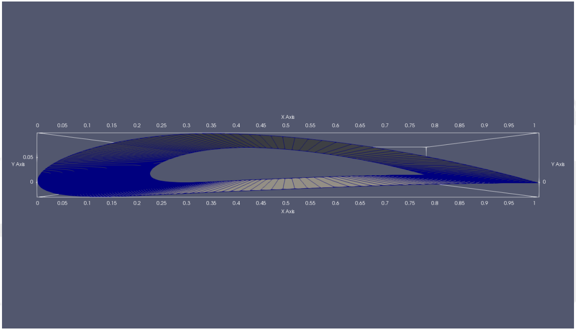


Figure: We will need this later

The Running_Notes file

- In the case directory there is a file with instructions
  - To only let the case run till the first write
  - Change the time-step size
  - Continue the run
- This is because the un-physical initial conditions make the solution diverge for large time-steps

Running_Notes

```
nacaAirfoil
~~~~~~~~~~~

 * large domain with airfoil section near centre
 * extremely non-orthogonal, highly skew mesh refined around the airfoil
 * running at Mach 1.78
 * limited corrected 0.5 on all laplacianSchemes because the mesh is so poor
 * run to t = 0.02 with nextWrite; change to stopAt endTime to continue running
 * deltaT can be increased later in the run to 2e-07
```

Remark: it should probably say `run to t=2e-4`

The plan

- Stopping and starting by hand is boring
  - Also: we have a suspicion that running that long with small time-steps is not necessary
- We don't want to modify the case by hand
  - Increase the time-step during the run
  - Increase should start once the residuals are small enough
  - And only go to a maximum
- Add some more evaluations
  - Mesh quality
  - Location of the shock-front before the foil
  - See the regions where the solution is not converged
    - Residuals are high

Outline

Preparing the mesh

- We remove the `Allrun-mesh` and prepare the case for use with `pyFoamPrepareCase.py`
  - Move the 0-directory to `0.org`
  - Create this script from `Allrun`

meshCreate.sh

```
#!/bin/sh

star4ToFoam -scale 1 \
    $FOAM_TUTORIALS/resources/geometry/nacaAirfoil/nacaAirfoil

# Symmetry plane -> empy
sed -i -e 's/symmetry\([)]*;\)/empty\1/' constant/polyMesh/boundary

# Don't need these extra files (from star4ToFoam conversion)
rm -f \
   constant/cellTable \
   constant/polyMesh/cellTableId \
   constant/polyMesh/interfaces  \
   constant/polyMesh/origCellId
```

Adding function objects and other stuff

These libraries are needed for the additional functionality

system/controlDict

```
libs (
    "libsimpleSwakFunctionObjects.so"
    "libswakStateMachine.so"
    "libswakMeshQualityFunctionPlugin.so"
    "libswakVelocityFunctionPlugin.so"
    "libswakLocalCalculationsFunctionPlugin.so"
    "libswakFunctionObjects.so"
    "libswakFvOptions.so"
);
```

Setting up the case

- This script is also called by pyFoamPrepareCase.py
- funkySetFields sets fields with the local non-orthogonalities of the mesh
  - mqFaceNonOrtho is from the libswakMeshQualityFunctionPlugin.so
    - Calculates the non-orthogonality of the faces
    - Paraview can't visualize face values
    - lcFaceMaximum from libswakLocalCalculationsFunctionPlugin.so sets the cell value to the maximum of its face-values
- The stuff below is a *template* that lets pyFoamPrepareCase.py decompose the case
  - There is a special presentation on that tool

caseSetup.sh.template

```
#!/bin/sh

rm -rf processor*

funkySetFields -time 0 -field cellNonOrth -create -expression "lcFaceMaximum(mqFaceNonOrtho())"
funkySetFields -time 0 -field faceNonOrth -create -expression "mqFaceNonOrtho()"

<!--(if numberOfProcessors>1)-->
pyFoamDecompose.py . |-numberOfProcessors-|
<!--(end)-->
```

Non-orthogonality field

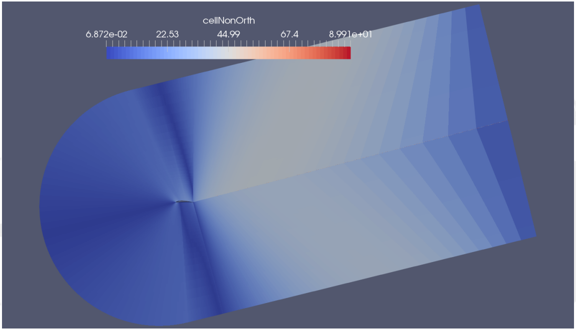


Figure: The non-orthogonality of the cells

Non-orthogonality field close-up

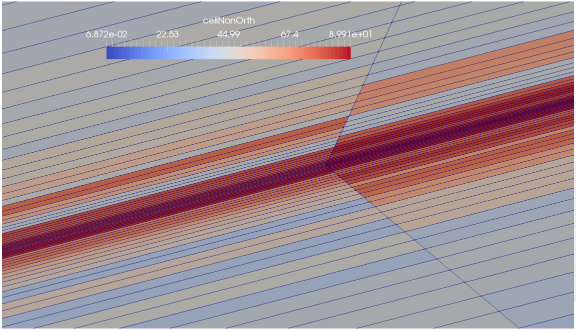


Figure: The worst cells

Running the case

- Lets run the case on 3 CPUs

Prepare and run

```
> pyFoamPrepareCase.py . --number=3
<snip>
> pyFoamPlotRunner.py --clear --auto --progress --with-all auto
<snip>
```

- Now we should have results like the following pictures
  - Don't ask me to interpret them. Supersonic flow is not my field

Solution: Velocity

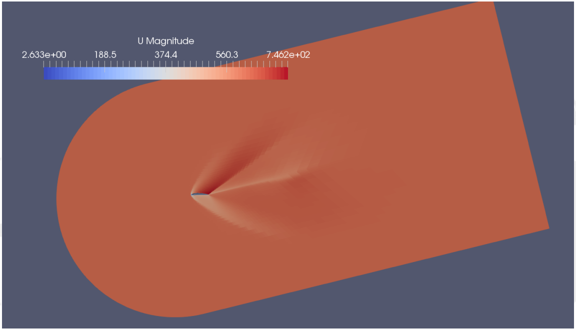


Figure: Flow solution

Solution: Pressure

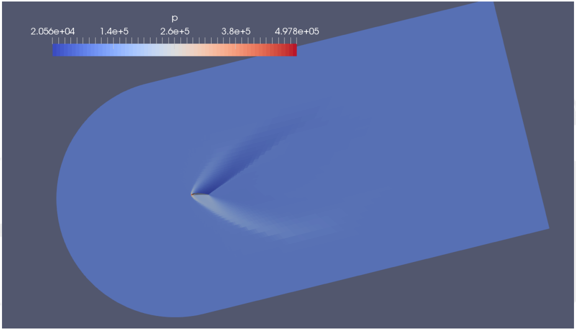


Figure: Overview of the pressure

Solution: Pressure Close-up

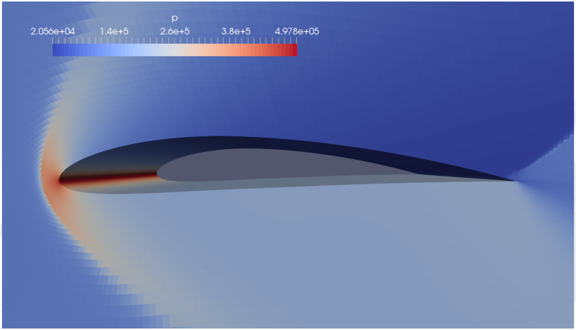


Figure: Shockwave in front of the foil

Solution: Temperature

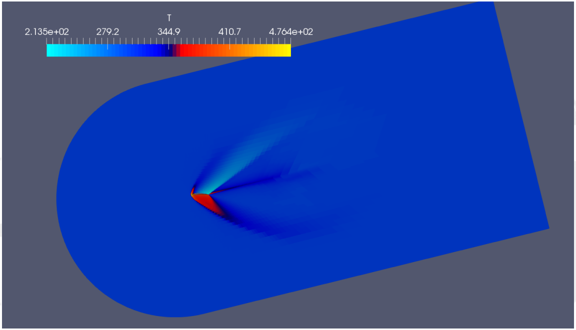


Figure: Heating

Solution: Turbulence

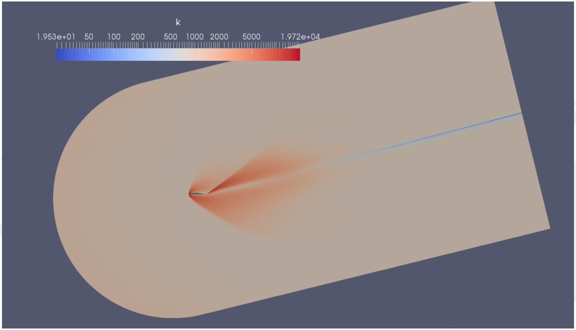


Figure: Turbulent kinetic energy

Writing the current time-step

- sonicFoam doesn't expect the time-step to change
  - Therefor it is not automatically written

functions in system/controlDict

```
deltaTValue {
    type swakExpression;
    valueType patch;
    patchName inlet_1;
    outputControlMode timeStep;
    outputInterval 1;
    accumulations (
        average
    );
    expression "deltaT()";
    verbose true;
}
```

customRegexp

```
timeStepValue {
    theTitle "Timestep␣[s]";
    expr "Expression␣deltaTValue␣:␣␣average=(.+)";
    titles (
        value
    );
    logscale true;
}
```

Write additional fields

- We write two additional fields

rho	which is already there. But not written
CoNumber	we calculate this with the plugin-function `courantCompressible` from the `libswakVelocityFunctionPlugin.so`

functions in system/controlDict

```
writeRho {
    type writeAdditionalFields;
    fieldNames (
        rho
    );
    outputControlMode outputTime;
}
courantField {
    type expressionField;
    autowrite true;
    fieldName CoNumber;
    expression "courantCompressible(phi,rho)";
    aliases {
        rhoField thermo:rho;
    }
}
```

Solution: Density

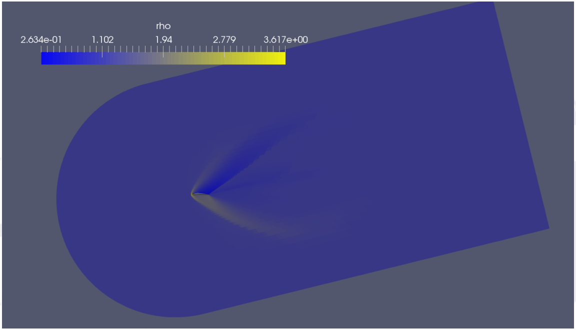


Figure: Usually this is not written

Solution: Local Courant number

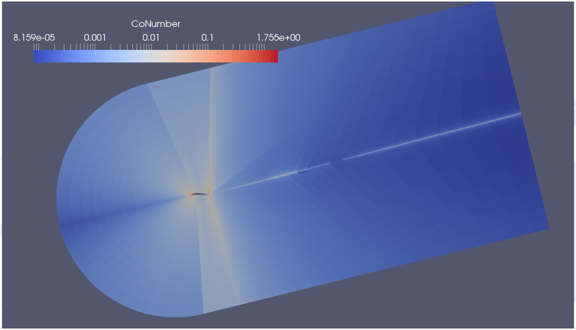


Figure: Courant number in all cells

Statistics of the local Courant-number

functions in system/controlDict

```
courantStatistics {
    type swakExpression;
    valueType internalField;
    expression "CoNumber";
    outputControlMode timeStep;
    outputInterval 1;
    verbose true;
    accumulations (
        weightedQuantile0.1
        weightedAverage
        weightedQuantile0.9
        weightedQuantile0.99
        weightedQuantile0.999
        max
    );
}
```

customRegexp

```
courantValues {
    theTitle "Courant␣number";
    expr "Expression␣courantStatistics␣:␣␣weightedQuantile0.1=(.+)␣weightedAverage=(.+)␣<brk>
      <cont>weightedQuantile0.9=(.+)␣weightedQuantile0.99=(.+)␣weightedQuantile0.999=(.+)␣max=(.+)"<brk>
      <cont>;
    logscale true;
    titles (
        "10%"
        average
        "90%"
        "99%"
        "99.9%"
        max
    );
}
```

Outline

Getting the performance of the solver

- First we've got to know how the solver is performing
- OpenFOAM stores this information in a data structure called `solverPerformance`
- swak4Foam can get it with `solverPerformanceToGlobalVariables`
    - `fieldNames` which fields we're interested in

functions in system/controlDict

```
solverValues {
    type solverPerformanceToGlobalVariables;
    fieldNames (
        p
    );
    toGlobalNamespace solver;
    outputControlMode timeStep;
    outputInterval 1;
}
```

Global variables

- To move data from one function object to another `swak4Foam` has something called *Global variables*
- To have some kind of separation they are organized in namespaces
  - Organize the variables into namespaces by "topic"
    - In our case `solver` for solver data
- Function objects that can write global variables have an entry `toGlobalNamespace`
- Everywhere where you can specify `variables` you can add an optional `globalScopes`
  - This is a list with names of global namespaces
  - All the variables in these namespaces are "injected" before the regular variables
  - Attention: the size of the global variables must match the size of the entity (for instance: number of faces)
    - If the variable is "uniform" it matches anywhere

Variables from `solverPerformance`

- All the variables are prefixed with the field name and a _
- Then there are the three informations usually printed to the console

  initialResidual the residual in the beginning
  finalResidual the residual in the end
  nIterations the number of iterations
- Then another _
- Then the information which solution attempt

  first First attempt
  last Last one. If there was only one attempt it is the same one as `first`
  intermediate attempts are not available. Sorry
  - Couldn't find an application for that

Calculation with the solver performance

Here we calculate the improvement per iteration f for the first solution assuming $\frac{r_{init}}{r_{final}} = f^{n_{iter}}$

functions in system/controlDict

```
printPImprovement {
    type swakExpression;
    valueType patch;
    patchName inlet_1;
    accumulations (
        average
    );
    expression "exp(log(p_initialResidual_first/p_finalResidual_first)/p_nIterations_first)<brk>
          <cont>";
    globalScopes (
        solver
    );
    outputControlMode timeStep;
    outputInterval 1;
    verbose true;
}
```

Pressure values

This is a "bread and butter" calculation I add almost everywhere

functions in system/controlDict

```
pressureValues {
    type swakExpression;
    valueType internalField;
    expression "p";
    outputControlMode timeStep;
    outputInterval 1;
    verbose true;
    accumulations (
        min
        weightedQuantile0.001
        weightedQuantile0.999
        max
    );
}
```

customRegexp

```
pressureValues {
    theTitle "Pressure";
    expr "Expression␣pressureValues␣:␣␣min=(.+)␣weightedQuantile0.001=(.+)␣weightedQuantile0.999=(.+)␣max=(.+)";
    titles (
        min
        "0.1%"
        "99.9%"
        max
    );
}
```

Where are the pressure extremes

- Sometimes it is sufficient to know the maximum `max(p)` of a field
- But sometimes we want to know where the maximum is located
  - `maxPosition(p)` gives us that
- Finding the shock front is a bit harder
  - "Find me the smalles x for which the pressure is bigger than 1.1 bar"

functions in system/controlDict

```
highPLocation {
    $pressureValues;
    expression "maxPosition(p)";
    accumulations (
        average
    );
}
lowPLocation {
    $highPLocation;
    expression "minPosition(p)";
}
shockPLocation {
    $highPLocation;
    expression "minPosition(p>1.1e5 ? pos().x : 0)";
}
```

Outline

Setting the possible time-steps

- To make things more readable we add two new entries to the controlDict:
  - smallest possible timestep
    - this is also the initial value of deltaT
  - biggest (target) timestep
- set adjustableRunTime to avoid "odd" time directories like 0.9973e-3

functions in system/controlDict

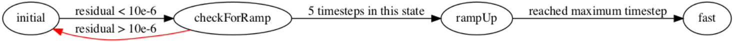
```
minDeltaT        4e-08;
maxDeltaT        20e-08;

deltaT           $minDeltaT;

writeControl     adjustableRunTime;
```

Our strategy for the timestep

- Stay in the `inital` state until the initial residual of p drops below 10^{-6}
- Then stay in `checkForRamp` for 5 timesteps
  - If the residual rises above 10^{-6} go back to `intial`
- Move to `rampUp`
  - Now we can scale the time-step up
- Once the target time-step is reached move to `fast`
  - Time-step stays constant



Specifying the state machine

functions in system/controlDict

```
    theStateMachine {
        type stateMachineCreateAndUpdate;
        valueType patch;
        patchName inlet_1;
        states (
            initial
            checkForRamp
            rampUp
            fast
        );
        machineName stepping;
        initialState initial;
        globalScopes (
            solver
        );
<<cont>>
```

Macro expansion in swak-expressions

- swak-expressions and the OpenFOAM-dictionaries are two completely different worlds
  - But sometimes it would be nice to access dictionary values in expressions
- The $-symbol allows this
  - After that inside of `[]` we specify two things
    - What type is the value (in cast-notation from C++)
    - Where to find it: in OpenFOAM-macro notation without the initial $
- In the following example `$[(scalar):maxDeltaT]` means "get `maxDeltaT` from the top-level of the dictionary and insert it as a scalar
- A detailed description (including the possible casts) is given in the *Incomplete Reference Guide*

Specifying the transitions

functions in system/controlDict

```
    transitions (
        {
            description "We're␣ready␣to␣speed␣up";
            condition "p_initialResidual_first<1e-6";
            logicalAccumulation and;
            from initial;
            to checkForRamp;
        }
        {
            description "Go␣back␣to␣intial";
            condition "p_initialResidual_first>1e-6";
            logicalAccumulation and;
            from checkForRamp;
            to initial;
        }
        {
            description "Only␣if␣5␣times␣good";
            condition "stateMachine_stepsSinceChange(stepping)>5";
            logicalAccumulation and;
            from checkForRamp;
            to rampUp;
        }
        {
            description "The␣ramp␣has␣succeeded";
            condition "deltaT()>=0.999*$[(scalar):maxDeltaT]";
            logicalAccumulation and;
            from rampUp;
            to fast;
        }
    );
}
```

Writing the transitions

- For plotting we write out the machine state
  - and tell PyFoam how to pick it up

functions in system/controlDict

```
writeState {
    type stateMachineState;
    outputControlMode timeStep;
    outputInterval 1;
    verbose true;
    machineName stepping;
}
```

customRegexp

```
steppingState {
    theTitle "State␣machine␣stepping";
    expr "Machine␣stepping␣in␣state␣.+␣\(code:␣([0-9]+)\)␣.+";
    titles (
        state
    );
    with points;
}
```

Setting the timestep

- Finally what we wanted
  - Scale the time-step up if we're in rampUp
  - Otherwise leave it alone

functions in system/controlDict

```
setDeltaT {
    type setDeltaTBySwakExpression;
    outputControlMode timeStep;
    outputInterval 1;
    deltaTExpression {
        expression "targetDeltaT";
        independentVariableName t;
        valueType patch;
        patchName inlet_1;
        storedVariables (
            {
                name targetDeltaT;
                initialValue "$[(scalar):deltaT]";
            }
        );
        variables (
            "targetDeltaT=stateMachine_isState(stepping,rampUp)␣?␣min(targetDeltaT*1.01,$[:<brk>
                  <cont>maxDeltaT])␣:␣targetDeltaT;"
        );
    };
}
```

Stored variables

- To be able to scale `targetDeltaT` up we've got to know which value it had before
- Stored variables allow us to do that
  - Keep their values between time-steps
  - If they were never set an `intialValue` is used
- These variables are declared in a list `storedVariables`

The states of the machine

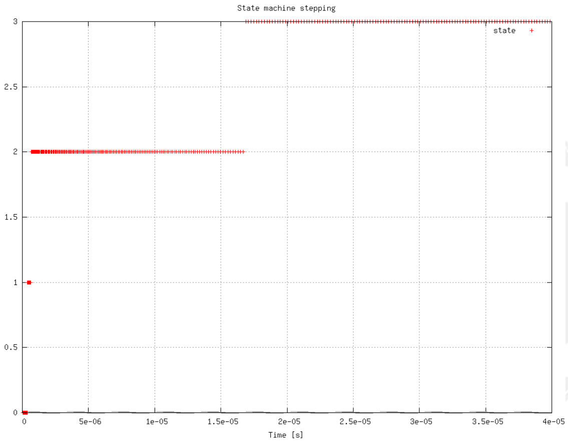


Figure: after the startup nothing changes

Size of the timesteps

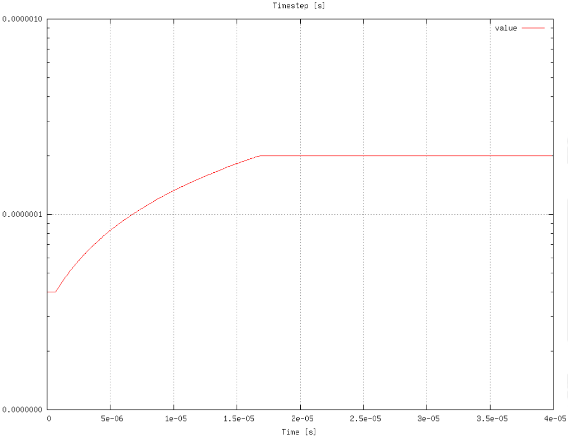


Figure: Going to a maximum

Size of the timesteps during the whole simulation

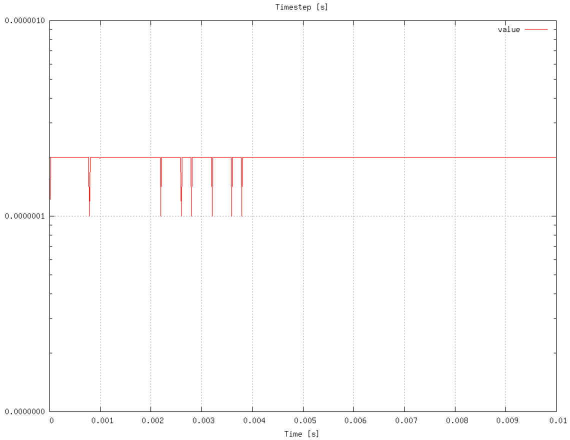


Figure: Scaled down for writing

Courant number distribution

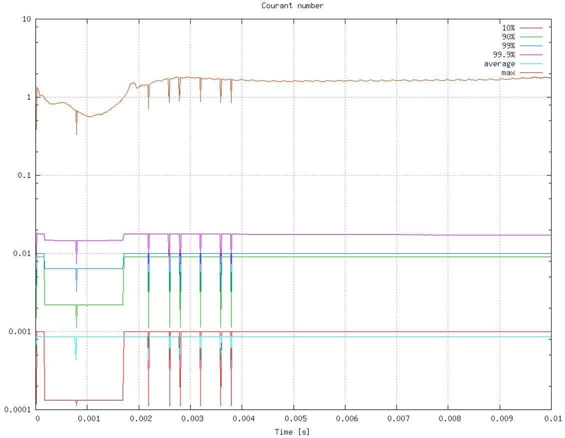


Figure: Over the whole simulation

Residuals of the linear solver

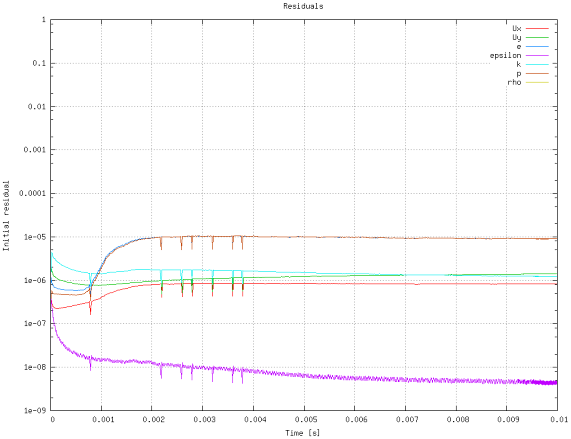


Figure: This is a standard-plot

How much does the solver improve the pressure equation

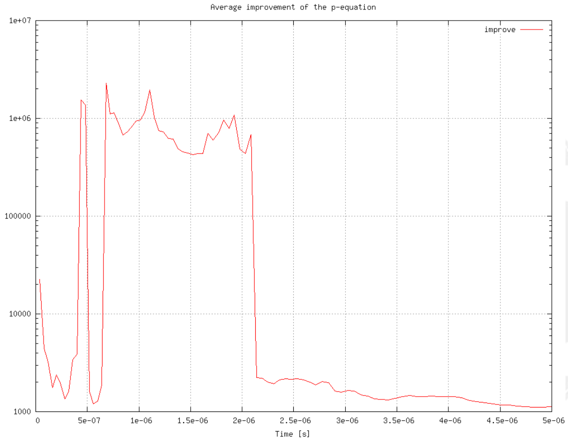


Figure: Residual gets smaller by this factor

Development of pressure at startup

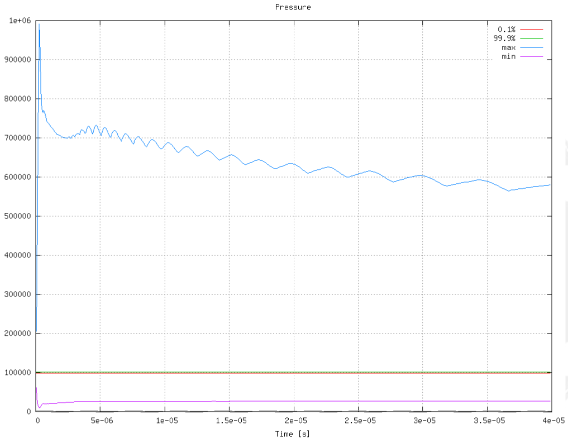


Figure: How does the pressure distribution evolve

Development of pressure during the simulation

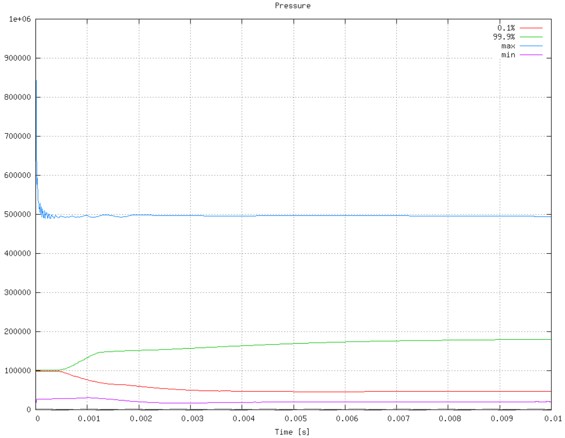


Figure: Pressure goes to fixed values

Where are the pressure extremes

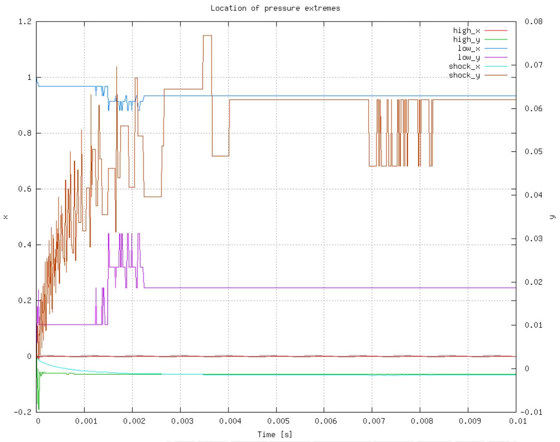


Figure: Minimum, Maximum and Shock-front

Residuum of the momentum equation

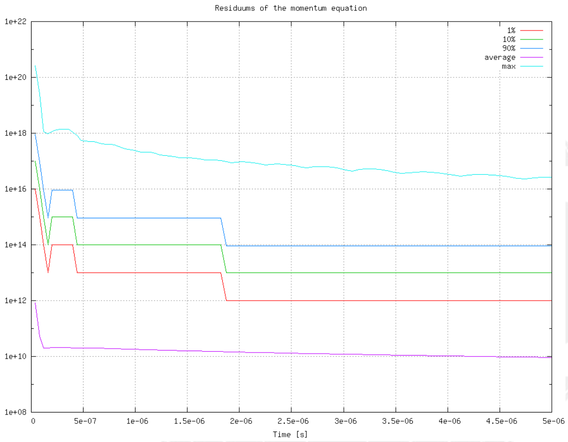


Figure: Evolution of the error

Outline

Which fvOption-entry points are available

- Not all possible entry-points for fvOptions are implemented
  - Sometimes with very good reasons
- Finding out which are actually can be quite a pain
  - One has to go to the source
- This fvOption prints this information every time a fvOption could be used
  - the name of the field
  - the available fvOption hook
- Does nothing else

constant/fvOptions

```
showFvOptions {
    type reportAvailableFvOptions;
    active true;
    selectionMode all;
    reportAvailableFvOptionsCoeffs {}
}
```

Calculating the residual

- This fvOption calculates the residual $\vec{r} = \vec{A}\vec{x} - \vec{b}$ for the current matrix and solution for fieldName
  - Stores the result in a field whose name is composed of namePrefix and fieldName
- doAtAddSup specifies whether this should be done when the source terms are done
- Caution: the order inside the fvOptions-file is important here
  - "Whicvh fvOption already manipulated the matrix
  - Especially when used together with its After-sibling
    - See below

constant/fvOptions

```
momentumResidual {
    type matrixChangeBefore;
    active true;
    selectionMode all;
    matrixChangeBeforeCoeffs {
        doAtAddSup no;
        fieldName U;
        namePrefix residual;
    }
}
```

Calculating the relative residual

- The way the residual is calculated it depends on the cell size
  - By scaling it with the cell size we get something more meaningful

functions in system/controlDict

```
notOnStart {
    type executeIfStartTime;
    readDuringConstruction false;
    runIfStartTime false;

    functions {
        relativeChange {
            type expressionField;
            autowrite true;
            fieldName relResidualU;
            expression "residualU/vol()";
        }
        momentumChange {
            $pressureValues;
            accumulations (
                weightedQuantile0.01
                weightedQuantile0.1
                weightedAverage
                weightedQuantile0.9
                max
            );
            expression "mag(relResidualU)";
        }
    }
}
```

Conditional function object execution

- At the first time-step no residual field is available
  - To avoid an error we guard it with `executeIfStartTime`
    - `runIfStartTime` to `false` negates the meaning
- This is an example for a function object whose main purpose is the calling of other function objects
  - The other function objects are listed in a `functions` dictionary
- A number of such function objects is available
  - All starting with `executeIf`
  - Even depending on swak-expressions
- Optionally they can have an `else`-entry
- The `readDuringConstruction`-entry controls when the `functions`-list is read
  - May be necessary to set to avoid problems with the "client" function objects

Solution: Residual of $\vec{u}$

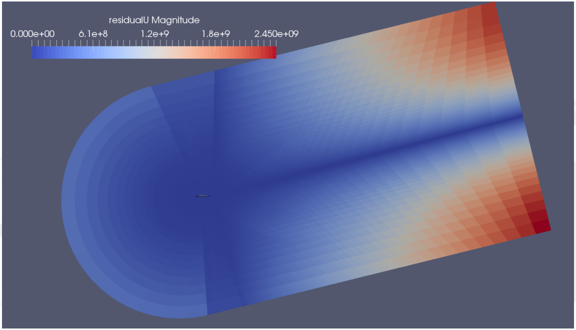


Figure: The absolute residual

Solution: Relative residual of $\vec{u}$

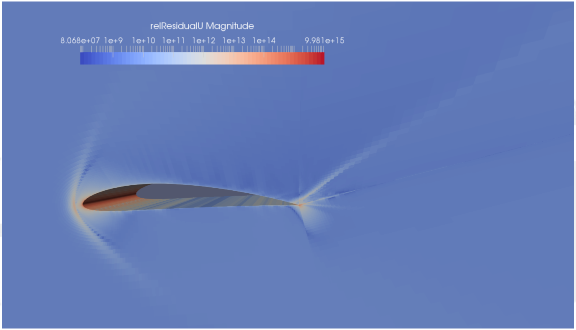


Figure: The relative residual

Outline

Checking for convergence of physical parameters

- When calculating a steady case we want the solution to converge
  - Meaning "It should not change anymore"
  - Numerical convergence is only an indication for this
- Checking for this is tricky
  - Comparing the current solution with the previous solution is not enough
    - The "peak" of an oscillation may look like a final state
  - Storing more solutions is prohibitive
  - Small cells may oscillate without influencing the overall solution
- In this section it is demonstrated how to check for convergence using a subset of the solution

Speeding up the simulation

- When starting from "unphysical" initial conditions the simulation is likely to crash
- Often you hear hints like
  - "In the beginning .."
    - use smaller timesteps
    - use small relaxation factors
    - use lower order schemes
  - "... and after some iterations ... "
    - increase the timestep
    - increase the relaxation
    - switch to higher order schemes
- In the last example you saw how to manipulate the time-step
  - Here we'll do the other two

Outline

The tutorial case

We use `$FOAM_TUTORIALS/incompressible/simpleFoam/simpleCar/`

- This is an incompressible steady simulation
- It simulates a simplified car
  - 2D
  - No wheels
  - A porous zone to simulate flow through the engine
- What we want to do with this case
  - Check for convergence by looking on the flow field 6m from the inlet
    - this was chosen because it is still in the recirculation
  - After some time increase the relaxation
    - Spoiler: good idea
  - Switch to higher order schemes
    - Spoiler: bad idea

Adaption for pyFoam

- Again we switch from `Allrun` to `pyFoamPrepareCase.py`

meshCreate.sh

```
#!/bin/sh

blockMesh
topoSet
```

caseSetup.sh.template

```
#! /bin/sh

rm -rf processor*
<!--(if numberOfProcessors>1)-->
pyFoamDecompose.py . |-numberOfProcessors-|
<!--(end)-->
```

Running it

```
> pyFoamPrepareCase . --number=2
> pyFoamRunner.py --clear --progress --auto auto
```

Solution: the velocity field

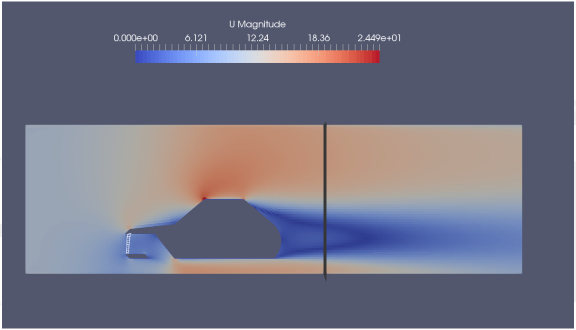


Figure: Mark at 6m

Solution: the turbulence

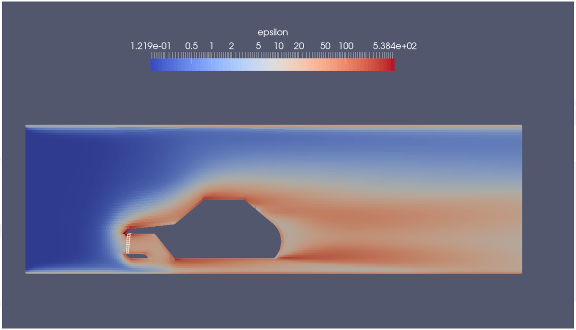


Figure: Turbulence converged

Outline

Don't use residuals for convergence

- Previously the run stopped when residuals fell below a limit
  - We comment that out
- Now the run would continue until endTime

system/fvSolution

```
SIMPLE
{
    nNonOrthogonalCorrectors 0;

    residualControl
    {
        // p              1e-2;
        // U              1e-4;
        // "(k|epsilon)" 1e-4;
    }
}
```

Adding our stuff

- This time we don't need much

system/controlDict

```
libs (
    "libsimpleSwakFunctionObjects.so"
    "libswakFunctionObjects.so"
    "libswakStateMachine.so"
);
```

Creating a line to calculate on

- We create the 6m line with `createSampleSet`
  - Syntax is similar to the `set` function object
    - But not written
    - Instead swak4Foam can access it under the `setName`

functions in system/controlDict

```
createSampleLine {
    type createSampledSet;
    outputControl timeStep;
    outputInterval 1;
    setName sixmLine;
    set {
        type uniform;
        axis distance;
        start (6 0 0.05);
        end (6 3 0.05);
        nPoints 100;
    }
    writeSetOnConstruction true;
    autoWriteSet true;
    setFormat vtk;
}
```

Calculating and storing the difference

- This is where the magic happens: current velocity on sixmLine is compared with the one 50 iterations ago

functions in system/controlDict

```
calcDifference {
    type calculateGlobalVariables;
    valueType set;
    setName sixmLine;
    verbose true;
    outputControl timeStep;
    outputInterval 1;
    variables (
        "oldU=U;"
        "diffU=U-oldU;"
    );
    toGlobalNamespace velDifference;
    toGlobalVariables (
        diffU
    );
    delayedVariables (
        {
            name oldU;
            startupValue "vector(0,0,0)";
            storeInterval 1;
            delay        50;
        }
    );
}
```

Calculating our own global variables

- Previously `solverPerformanceToGlobalVariables` calculated the global variables for us
- `calculateGlobalVariables` allows us to calculate them ourselves
  1. Calculates all expressions in `variables`
  2. Looks at the list `toGlobalVariables`
  3. Variables found in that list are stored in `toGlobalNamespace`
- Now the variable values are available for other function objects

Delayed variables

- Delayed variables are special variables with a schizophrenic behaviour
  - When written to they behave like regular variables
  - When read they don't use the current value but the value set some time ago (the *delay*)
- They are declared in a list `delayedVariables` of dictionaries
  - **name** the name under which the variable is known
  - **delay** how far back in time it should go
  - **startupValue** during the first `delay` seconds there is nothing to remember. This value is used instead
  - **storeInterval** this is the interval at which values should be remembered. When remembering values between that are interpolated
    - set it too high: you might run out of memory
    - set it too low: it might be inaccurate
    - in our steady simulation 1 means: we remember everything
- Values longer ago than `delay` are forgotten

Reporting the change

- We want to see how big the changes are

functions in system/controlDict

```
changedU {
    type swakExpression;
    expression "mag(diffU)";
    accumulations (
        average
        max
    );
    valueType set;
    setName sixmLine;
    verbose true;
    outputControl timeStep;
    outputInterval 1;
    globalScopes (
        velDifference
    );
    //        debugCommonDriver 1;
}
```

Same for the porosity

- We duplicate this for the porous block
  - We don't use it
  - But we don't mind: macro expansion serves us the typing

functions in system/controlDict

```
calcDifferencePoro {
    $calcDifference;
    valueType cellZone;
    zoneName porousZone;
    toGlobalNamespace velDifferencePoro;
}
changedUPoro {
    $changedU;
    valueType cellZone;
    zoneName porousZone;
    globalScopes (
        velDifferencePoro
    );
}
```

Plotting the changes

- Seeing the changes convinces us that they get smaller

customRegexp

```
changeU {
    theTitle "Change␣of␣velocity␣6m␣after␣inlet";
    expr "Expression␣changedU␣:␣␣average=(.+)␣max=(.+)";
    logscale true;
    titles (
        average
        max
    );
}
changeUPoro {
    type slave;
    master changeU;
    expr "Expression␣changedUPoro␣:␣␣average=(.+)␣max=(.+)";
    titles (
        "average␣poro"
        "max␣poro"
    );
}
```

Strategy to find convergence

- Start in `initial`
- Wait 50 timesteps before we go to `waiting` and consider the changes
  - This is to allow our delayed variable to "fill up"
- When all changes are smaller than $1\frac{cm}{s}$ we move to `lookingGood`
- If we stay in `lookingGood` for 100 timesteps we move to `converged`
  - If change goes above $1\frac{cm}{s}$ we move back to `waiting`
- We don't leave `converged` but hope that someone will stop the simulation now

Create state machine

- We create the state machine

functions in system/controlDict

```
convergedStateMachine {
    type stateMachineCreateAndUpdate;
    valueType set;
    setName sixmLine;
    states (
        initial
        waiting
        lookingGood
        converged
    );
    machineName converged;
    initialState initial;
    globalScopes (
        velDifference
    );
```

The transitions

- and implement the transitions
  - Note: use of or and and when checking for "bigness"

functions in system/controlDict

```
    transitions (
        {
            description "Startup␣is␣over";
            condition "stateMachine_stepsSinceChange(converged)>50";
            logicalAccumulation and;
            from initial;
            to waiting;
        }
        {
            description "Go␣back␣to␣intial";
            condition "max(mag(diffU))<0.01";
            logicalAccumulation and;
            from waiting;
            to lookingGood;
        }
        {
            description "Got␣a␣big␣difference";
            condition "mag(diffU)>=0.01";
            logicalAccumulation or;
            from lookingGood;
            to waiting;
        }
        {
            description "Been␣good␣long␣enough";
            condition "stateMachine_stepsSinceChange(converged)>100";
            logicalAccumulation and;
            from lookingGood;
            to converged;
        }
    );
}
```

End if converged

- Someone has to end the run when the state machine converged is in state converged
  - `writeAndEndSwakExpression` is the kind of function object that has no problem with this
    - And it also triggers the data to be written (couldn't tell from the name)

functions in system/controlDict

```
endIfConverged {
    type writeAndEndSwakExpression;
    valueType set;
    setName sixmLine;
    logicalExpression "stateMachine_isState(converged,converged)";
    logicalAccumulation and;
}
```

Change of the velocity

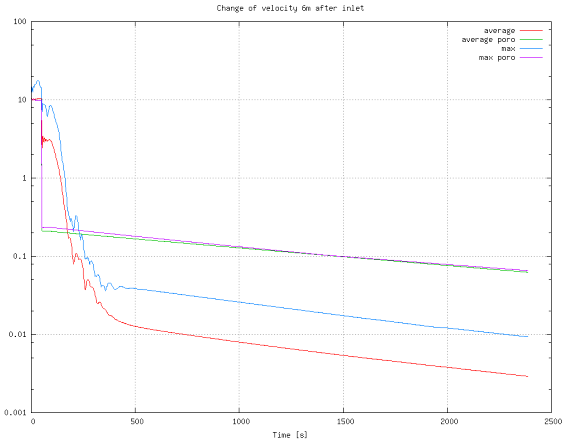


Figure: Change of the velocity on the 6m line

The residuals

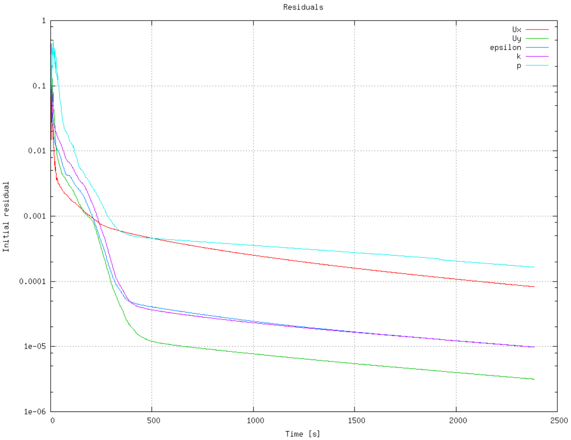


Figure: Going down steady

Outline

The original relaxation

- These are the "safe" relaxation parameters
  - They make sure that during the startup-phase the simulation does not diverge
  - But later they could be higher
    - Faster conergence

system/fvSolution

```
relaxationFactors
{
    fields
    {
        p               0.3;
    }
    equations
    {
        U               0.7;
        k               0.7;
        epsilon         0.7;
    }
}
```

Switching schemes and relaxation

- This function manipulates fvSchemes and fvSolution in memory at specified time
  - Second parameters are the names of the sub-directories to use
    - For instance "At time 200 use the contents of fastTransport to modify fvSolution
  - resetBeforeTrigger specifies whether old modifications should be removed
    - In our case fastFluid will be used in addition to fastTransport
- There is a similar function object stateMachineFvSolutionFvSchemes that does this based on the state of a state machine
  - But we would have needed to add a second state machine

functions in system/controlDict

```
switchFasterRelaxation {
    type timeDependentFvSolutionFvSchemes;
    solutionTriggers (
        (200 fastTransport)
        (400 fastFluid)
    );
    schemesTriggers (
        (500 highOrderTurb)
    );
    resetBeforeTrigger false;
}
```

Alternate relaxation factors

- First we speed up turbulence
  - Then the actual flow solution
- Maybe even higher relaxations are possible

system/fvSolution

```
fastTransport {
    relaxationFactors
    {
        equations
        {
            k               0.8;
            epsilon         0.8;
        }
    }
}
fastFluid {
    relaxationFactors
    {
        fields
        {
            p               0.4;
        }
        equations
        {
            U               0.8;
        }
    }
}
```

Change of schemes

Original convection schemes in system/fvSchemes

```
divSchemes
{
    default          none;
    div(phi,U)       bounded Gauss upwind;
    div(phi,k)       bounded Gauss upwind;
    div(phi,epsilon) bounded Gauss upwind;
    div((nuEff*dev2(T(grad(U))))) Gauss linear;
}
```

The higher-order overriding schemes in system/fvSchemes

```
highOrderTurb {
    divSchemes
    {
        div(phi,k)       bounded Gauss linearUpwind phi;
        div(phi,epsilon) bounded Gauss linearUpwind phi;
    }
}
```

Change of the velocity

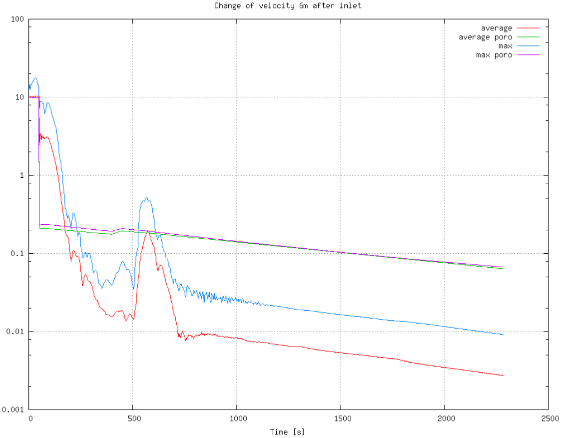


Figure: Change of the velocity on the 6m line

Change of the velocity - closer look

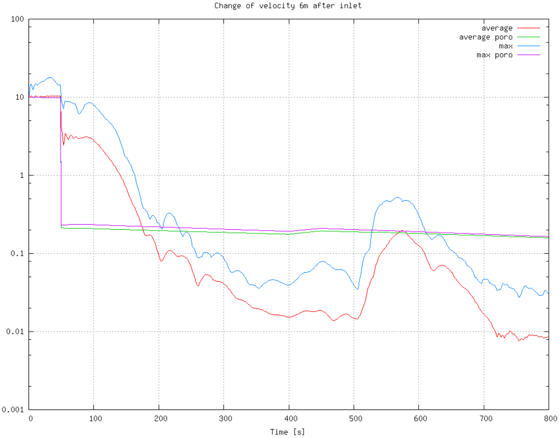


Figure: Change of the velocity on the 6m line

Residual

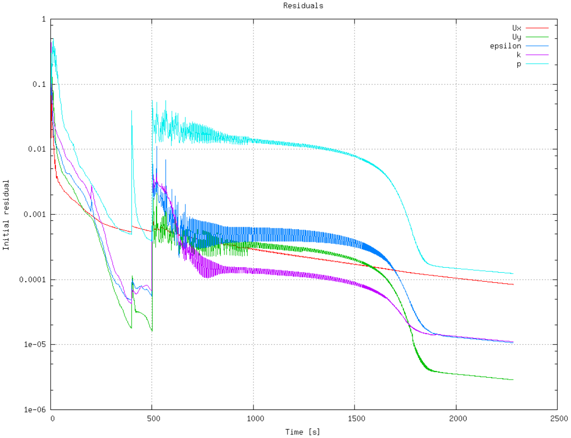


Figure: Higher order scheme "excite" the residuals

Residual in the beginning

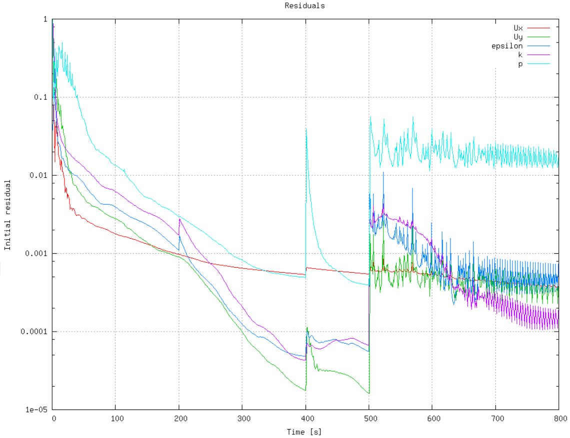


Figure: Changes in relaxation clearly visible

Outline

Outline

The tutorial case

The case we use is
`$FOAM_TUTORIALS/lagrangian/reactingParcelFoam/filter`

- Air flows through a filter
- Particles are injected
  - Can't pass through the filter
  - Water evaporates from the particles
    - Vapor is transported through the filter to the outlet

What we'll change about the case

- Particles disappear
  - Particles that lost 10% of their initial mass will be removed from the system
- Vapor condenses in the filter
  - In the filter a fraction of the vapor is removed from the air
  - It accumulates in the filter material
    - But distributes by diffusion
- The wet filter changes its permeability
  - Places with more condensed water resist the air-flow

All these changes are not completely improbable

- But the constants have been changed to make a quick simulation
- Does not resemble a real system

Adding our own weird preparation

Again: we make pyFoamPrepareCase.py happy

meshCreate.sh

```
#!/bin/sh

blockMesh

topoSet

createBaffles -overwrite
```

Add the swak-stuff

- Adding the necessary libraries

system/controlDict

```
libs (
    "libsimpleSwakFunctionObjects.so"
    "libswakLagrangianParser.so"
    "libswakFvOptions.so"
    "libswakSourceFields.so"
    "libswakFunctionObjects.so"
    "libswakLagrangianCloudSourcesFunctionPlugin.so"
    "libswakCloudFunctionObjects.so"
    "libsimpleCloudFunctionObjects.so"
);
```

Checking the pressure drop

Monitor the effect of the permeability change

functions in system/controlDict

```
pressureDrop {
    type patchExpression;
    patches (
        inlet
    );
    verbose true;
    accumulations (
        min
        weightedAverage
        max
    );
    variables (
        "pOut{outlet}=average(p);"
    );
    expression "p-pOut";
}
```

customRegexp

```
pressureDrop {
    theTitle "Pressure␣Drop␣[Pa]";
    expr "Expression␣pressureDrop␣on␣inlet:␣␣min=(.+)␣weightedAverage=(.+)␣max=(.+)";
    titles (
        min
        average
        max
    );
    progress "dP:␣$2";
}
```

How much water evaporates from the particles?

- lcsSpeciesSource is a plugin function
  - Asks a cloud for the amount of a species it transfers to the continuous phase
  - This is the source term that is usually used by the solvers

functions in system/controlDict

```
waterSource {
    type expressionField;
    autowrite true;
    fieldName H2Osource;
    expression "lcsSpeciesSource(reactingCloud1,H2O)";
}
waterSourceTotal {
    type swakExpression;
    valueType internalField;
    verbose true;
    expression "H2Osource";
    accumulations (
        integrate
    );
}
```

Outline

This is technical

- The parser for particle clouds has to do some ... strange ... things to give similar experience as the others
  - Sometimes it wants additional information because the cloud it uses does not exactely match the one the solver uses
- Here we had to add 4 values to make it work

constant/reactingCloudProperties

```
constantProperties
{
    rho0            1000;
    T0              300;
    Cp0             4100;

    constantVolume  false;

    // to keep the parser happy
    epsilon0 1;
    f0 0.5;
    LDevol 0;
    hRetentionCoeff 1;
}
```

How much work is moving the parcels

- `cloudFunctions` is the `functions` for lagrangian particles
  - swak4foam provides some function objects for this too
- This one collects statistics about how often particles hit patches etc
  - Quite useful if the solver starts to run slow and you suspect that it is because somewhere particles are caught in "infinite loop"

In `cloudFunctions` in `constant/reactingCloudProperties`

```
howMuchWork {
    type cloudMoveStatistics;
}
```

Typical output

```
howMuchWork:reactingCloud1:cloudMoveStatistics: Face hit Nr: 160 (716 particles) Min: 0 <brk>
      <cont>Mean: 0.2234636843 Max: 2
howMuchWork:reactingCloud1:cloudMoveStatistics: Moves Nr: 2936 (716 particles) Min: 4 Mean:<brk>
      <cont> 4.100558758 Max: 9
howMuchWork:reactingCloud1:cloudMoveStatistics Patch walls hit 1 times
howMuchWork:reactingCloud1:cloudMoveStatistics Patch cycLeft_half0 hit 6 times
```

Tracing a particle

Sometimes for debugging we want to follow on (or more) particles

In cloudFunctions in constant/reactingCloudProperties

```
whereGoes42 {
    type traceParticles;
    particleIds (
        {
            origProc 0;
            origId 42;
        }
    );
}
```

Typical output

```
whereGoes42:reactingCloud1:traceParticles: traced 1 particles
```

postProcessing/lagrangian/reactingCloud1/whereGoes42/0/trace0

```
# Time   descr (Px Py Pz) celli facei stepFraction tetFacei tetPti origProc origId active typeId <brk>
      <cont>nParticle d dTarget (Ux Uy Uz) rho age tTurb (UTurbx UTurby UTurbz) T Cp mass0 nPhases(Y1..<brk>
      <cont>YN) nGas(Y1..YN) nLiquid(Y1..YN) nSolid(Y1..YN)
0.544   postFace_face282        (0.1 0.44 0.05) 141 282 0.5 282 1 0 42 1 -1 19.09859317 0.001 0 (0.5 <brk>
      <cont>-0.1 0) 1000 0 0 (0 0 0) 300 4200 5.235987756e-07 3(0 1 0) 0() 1(1) 0()
0.544   postMove_cell141        (0.1000000625 0.4400000063 0.049999875) 142 282 0.5 282 1 0 42 1 -1 <brk>
      <cont>19.09859317 0.001 0 (0.5 -0.1 0) 1000 0 0 (0 0 0) 300 4200 5.235987756e-07 3(0 1 0) 0() <brk>
      <cont>1(1) 0()
```

"Lose 10%: you've got to go"

- This function object uses an expression:
  - "Check if the current mass is 90% of the initial mass"
  - This is checked after moving the particle
  - If it is true the particle is eliminated

In cloudFunctions in constant/reactingCloudProperties

```
eliminateLowMass {
    type eliminateBySwakExpression;
    eliminatePre false;
    eliminatePost true;
    eliminationExpression "mass/mass0<0.9"; // approx 90% of the mass
}
```

Output particle properties

- Here we get the distribution of the particle diameters and the temperature difference to the surrounding air
- Particle parsers work like every other parser
  - For clouds of parcels the weight is the total mass of the parcel (not the weight of one particle)
- `fluidPhase` allows interpolating the value of a fluid field to the particle position
  - It is necessary to specify an interpolation scheme

functions in system/controlDict

```
parcelDiameter {
    type swakExpression;
    verbose true;
    valueType cloud;
    expression "d";
    accumulations (
        min
        weightedAverage
        max
    );
    cloudName reactingCloud1;
}
parcelTDiff {
    $parcelDiameter;
    expression "T-fluidPhase(T)";
    interpolationSchemes {
        T cell;
    }
}
```

Output the first time the parser is used

- Each cloud type has a different set of values that can be accessed
  - The first time a parser is called it lists them all
    - That way you don't have to search for it in outdated documentation
  - constant means that the value can only be read

Different clouds have different properties

```
Driver for cloud reactingCloud1 of type Cloud<basicReactingMultiphaseParcel> (Proxy type: <brk>
    <cont>CloudProxy)
     List of functions:
                  Name |             Type | Description
-------------------------------------------------------------
                LDevol |           scalar | Latent heat of devolatilisation (constant)
                     T |           scalar | Temperature
                    T0 |           scalar | Initial temperature (constant)
                  TMin |           scalar | Minimum temperature (constant)
                     U |           vector | Velocity
                 UTurb |           vector | Turbulent velocity fluctuations
                active |             bool | Is this parcel active?
                   age |           scalar | Age of the prticle
                 areaP |           scalar | Particle projected area
                 areaS |           scalar | Particle surface area
                  cell |           scalar | number of the cell
                    cp |           scalar | Specific heat capacity
                   cp0 |           scalar | Specific heat capacity (constant)
           currentTime |           scalar | current time of the particle
                     d |           scalar | Diameter
               dTarget |           scalar | Target diameter
...
```

Outline

Model for condensed water in the filter

- We model the condensed water with a diffusion equation with a source term
  - solverLaplacianPDE solves such an equation at every timestep
- A field file condensed has to be added
  - With boundary conditions, dimensions and initial conditions
- For the relevant terms swak-expressions can be used
  - A proper dimension has to be provided
    - swak4Foam doesn't propagate dimensions on purpose when doing calculations
    - Dimension-checker of OpenFOAM would fail otherwise

functions in system/controlDict

```
condensedWater {
    type solveLaplacianPDE;
    solveAt timestep;
    fieldName condensed;
    steady false;
    rho "1" [0 0 0 0 0 0 0];
    lambda "zone(filter) ? 1e-3 : 0" [0 2 -1 0 0 0 0];
    source "rho*H2O*(zone(filter) ? 1 : 0)" [1 -3 -1 0 0 0 0];
}
```

How much is in the filter?

- We want statistics about the condensed water

functions in system/controlDict

```
condensedValue {
    type swakExpression;
    valueType cellZone;
    zoneName filter;
    accumulations (
        min
        weightedQuantile0.1
        weightedAverage
        weightedQuantile0.9
        max
    );
    expression "condensed";
    verbose true;
}
condensedTotalSource {
    $condensedValue;
    expression "H2O*rho";
    accumulations (
        integrate
    );
}
```

Condensed water must be removed from the fluid phase

- For mass conservation the water that condenses must be removed from the air
- swak4Foam has `fvOptions` that allow adding any source term to equations
  - If the equations support `fvOption`-source terms
- We use the implicit variant to avoid "undershooting"
  - Technically this is `-rho*H2O`
- Again: dimension has to be provided

constant/fvOptions

```
waterSwak {
    type scalarSwakImplicitSource;
    active true;
    scalarSwakImplicitSourceCoeffs {
        selectionMode   cellZone;
        cellZone        filter;
        switchExplicitImplicit true;
        expressions {
            H2O "-rho" [1 -3 -1 0 0 0 0];
        }
    }
}
```

The "regular" filter

- This is the original Darcy-term in the model
    - We disable it

constant/fvOptions

```
filter1
{
    type            explicitPorositySource;
    active          no;

    explicitPorositySourceCoeffs
    {
        selectionMode   cellZone;
        cellZone        filter;

        type            DarcyForchheimer;

        DarcyForchheimerCoeffs
        {
            d   (500000 -1000 -1000);
            f   (0 0 0);

            coordinateSystem
            {
                type    cartesian;
                origin  (0 0 0);
                coordinateRotation
                {
                    type    axesRotation;
                    e1  (1 0 0);
                    e2  (0 1 0);
                }
            }
        }
    }
}
```

Condensed Water adds to the resistance

- Now we add our own resistance
  - The same factor as in the original
  - Plus a term that depends on the condensed water
- Problem:
  - Implicit only allows us to specify a scalar (no anisotropy)
  - Explicit unstable (that's what the `resist` variable was for

constant/fvOptions

```
filterSwak {
    type vectorSwakImplicitSource;
    active true;
    vectorSwakImplicitSourceCoeffs {
        selectionMode   cellZone;
        cellZone        filter;
        switchExplicitImplicit true;
        aliases {
            mu thermo:mu;
        }
        variables (
            "coeff=500000*(1+condensed/0.005);"
            "baseResist=coeff*mu;"
            "resist=baseResist*vector(1,1000,1000);"
        );
        expressions {
            U "-baseResist" [1 -3 -1 0 0 0 0];
        }
    }
}
```

Finding out how big the source terms are

- For some `fvOptions` (heat exchanger, porosity) it would be nice to know how big the source term is
  - But they don't provide it
  - If they modify the matrix it is hard to tell
- The way swak4foam allows doing this is
  1. Calculate the residual before: $\vec{r}_1 = \vec{A}_1\vec{x} - \vec{b}_1$
  2. Let the other `fvOption` manipulate $\vec{A}$ and $\vec{b}$
  3. Calculate the residual after: $\vec{r}_2 = \vec{A}_2\vec{x} - \vec{b}_2$
  4. The added source term is $\vec{r}_2 - \vec{r}_1$
- There are two `fvOptions` that have to be used as a pair
  - Need the same `fieldName` and `namePrefix`

Before all source terms

constant/fvOptions

```
momentumSourceBefore {
    type matrixChangeBefore;
    active true;
    selectionMode all;
    matrixChangeBeforeCoeffs {
        doAtAddSup yes;
        fieldName U;
        namePrefix fvChange;
    }
    matrixChangeAfterCoeffs {
        $matrixChangeBeforeCoeffs;
    }
}
waterSourceBefore {
    type matrixChangeBefore;
    active true;
    selectionMode all;
    matrixChangeBeforeCoeffs {
        doAtAddSup yes;
        fieldName H2O;
        namePrefix fvChange;
    }
    matrixChangeAfterCoeffs {
        $matrixChangeBeforeCoeffs;
    }
}
```

After all source terms

- After this the fields fvChangeU and fvChangeH2O "know" hat has been "done" to the matrix

constant/fvOptions

```
momentumSourceAfter {
    $momentumSourceBefore;
    type matrixChangeAfter;
}
waterSourceAfter {
    $waterSourceBefore;
    type matrixChangeAfter;
}

momentumSourceResidual {
    $momentumSourceBefore;
    matrixChangeBeforeCoeffs {
        doAtAddSup no;
        fieldName U;
        namePrefix residual;
    }
}
```

Outline

It follows: a gallery

- The follwing slides show the results of our changes
  1. lines that were plotted with `pyFoamPlotRunner.py`
  2. Several fields in the middle of the simulation
     - Illustrate the model features we added
  3. Series of pictures that show how the condensed water diffuses in the filter

Number of particles

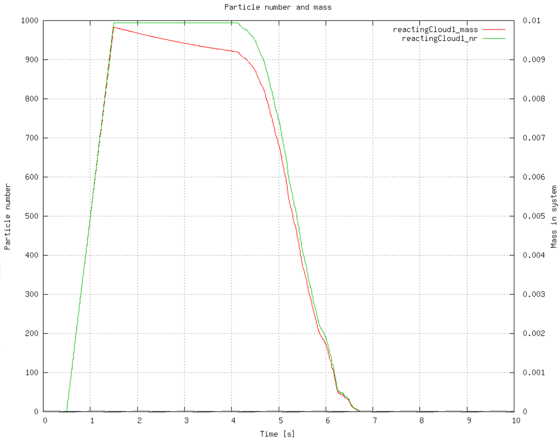


Figure: This plot is generated automatically by PyFoam

Different temperatures

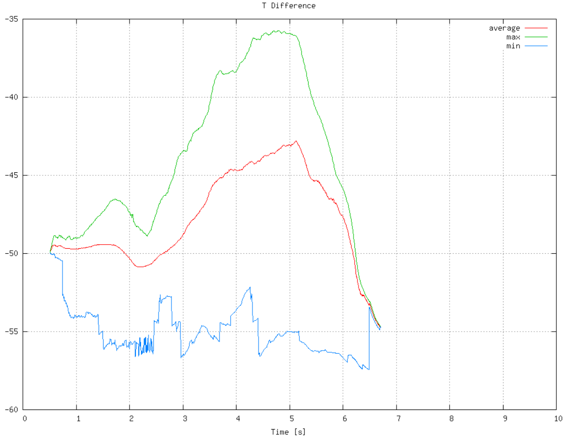


Figure: Difference between particle and surrounding gas

Evaporated water

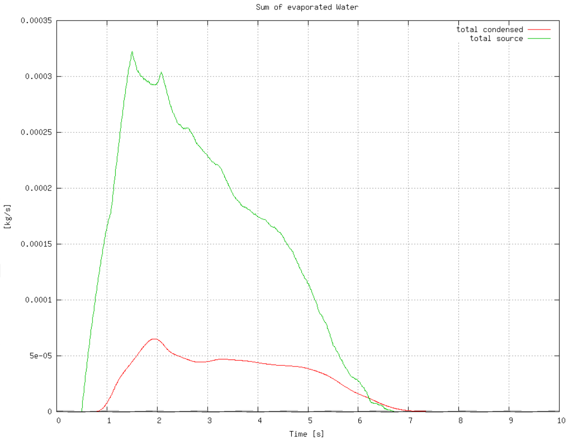


Figure: Water in the gas

Condensed water

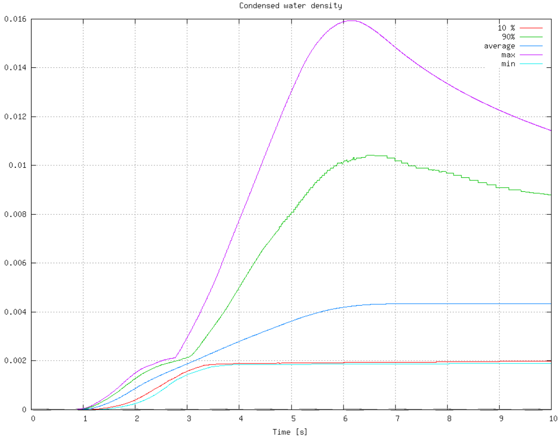


Figure: Average shows preservation after particles are gone

Velocity

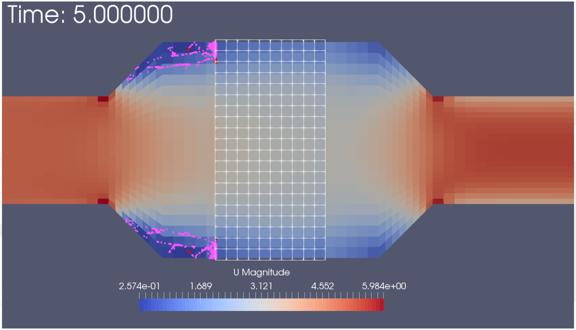


Figure: Gas velocity

Pressure

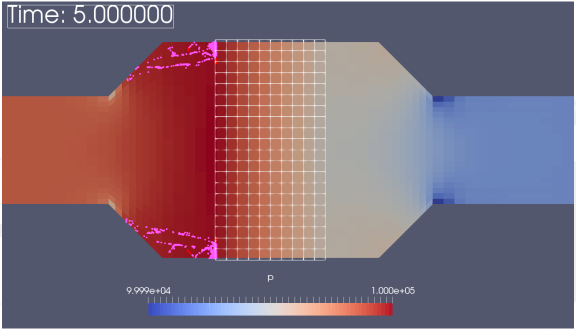


Figure: The filter makes a difference

Temperature

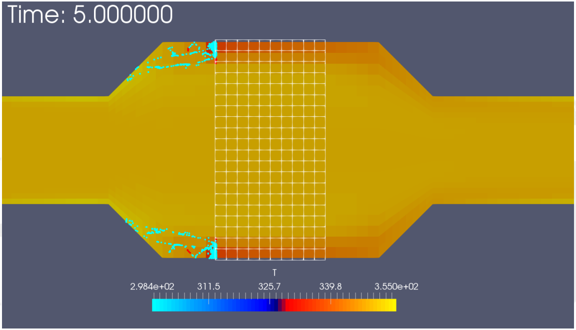


Figure: Particles cool the fluidPhase

Velocity source

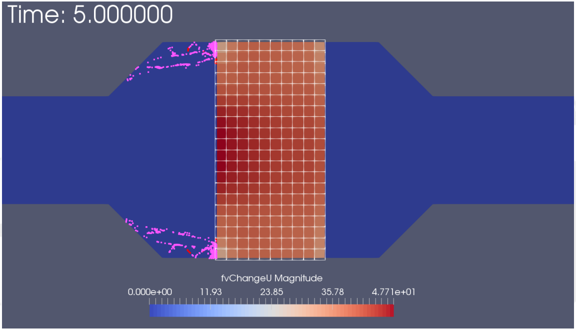


Figure: Resistance of the filter

Velocity residual

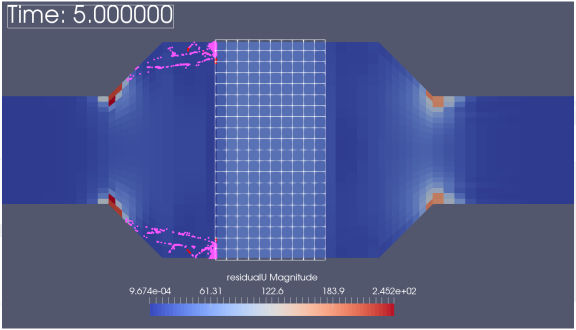


Figure: Problematic regions for the calculation

Water vapor source

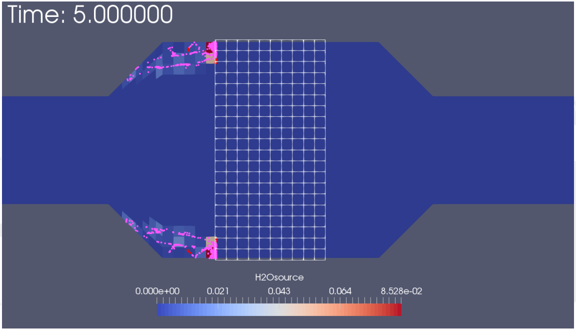


Figure: Water evaporating from the particles

Water vapor

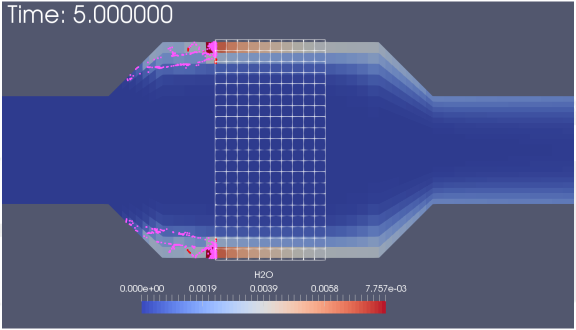


Figure: Water in the air

Water vapor condensing

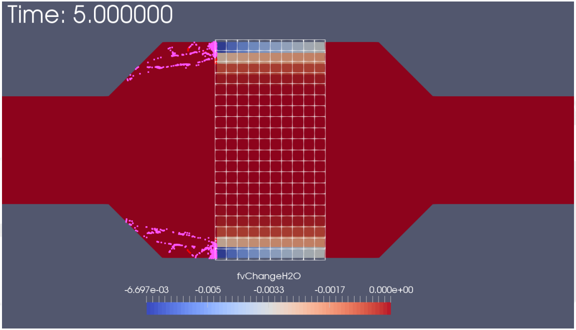


Figure: Water condensing on the filter

Water condensed

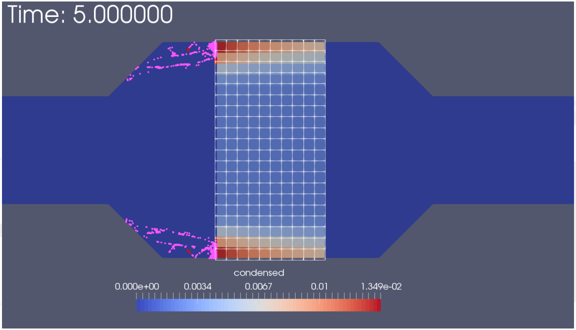


Figure: Water condensed in the filter

Water condensed when last particle "dies"

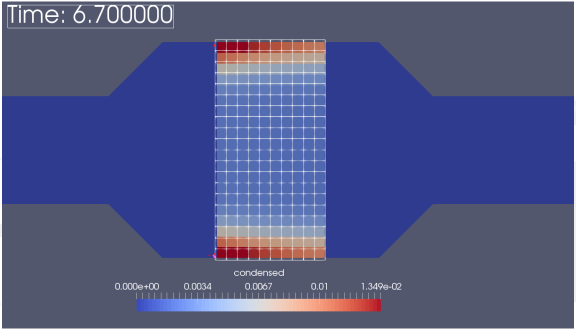


Figure: Maxiumum of condensed water

Water condensed in the end

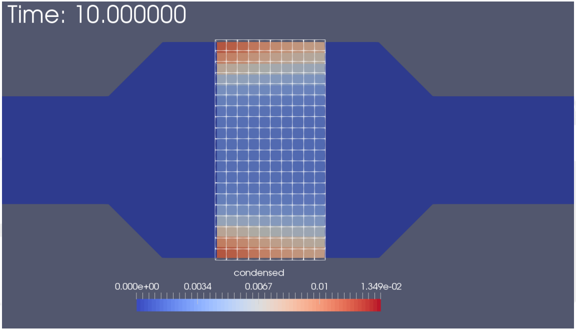


Figure: Water distributed in the filter

Outline

Words of warning

- The techniques outlined here can be very useful
- BUT when used improperly
  - they can make your run unstable
  - they can make your simulation unphysical

swak4Foam allows you to shoot yourself in the foot

Further reading

- This presentation only covered parts of `PyFoam` and `swak4Foam`, but there is further information available:
  - On the OpenFOAM-wiki:
    - `http://openfoamwiki.net/index.php/Contrib/swak4Foam` in the section *Further Information* are links to previous presentations
    - `http://openfoamwiki.net/index.php/Contrib/PyFoam` in section *Other material*
  - The `Examples` directory of the `swak`-sources
  - Did I mention the *Incomplete reference guide* for `swak`?
  - The `--help`-option of the `PyFoam`-utilities

Further presentations

- `pyFoamPrepareCase.py` can handle lots of things
  - With something called *templates*
  - See "Automatic case setup with `pyFoamPrepareCase`" from the Ann Arbor Workshop 2015
- We skipped the parts about writing data
  - These are explained in another presentation
    - "PyFoam for the lazy" from 2016
- The training about advanced swak-usage in the same session

Goodbye to you

Thanks for listening

Questions?

License of this presentation

Authors of this document are:

Bernhard F.W. Gschaider original author and responsible for the strange English grammar. Contact him for a copy of the sources if you want to extend/improve/use this presentation