swak4Foam for programmers

Programming when it can't be avoided

Bernhard F.W. Gschaider

Zagreb, Croatia
24. June 2014

Strömungsforschung GmbH
ICE
Innovative Computational Engineering

Outline I

1. Introduction
   - About this presentation
   - What we're working with
   - Before we start
2. Programming-like structures
   - Stored variables
   - Global variables
   - More obscure variable types
   - "Programming" function objects
3. Python Integration
   - General

Outline II

Approximate the pressure drop

4 Plugin-functions
- Why plugin-functions
- The Game of Life plugin
- Testing the plugin
- Additional functions
- Developing your own functions

5 Conclusion

Outline

1. Introduction
   - About this presentation
   - What we're working with
   - Before we start
2. Programming-like structures
   - Stored variables
   - Global variables
   - More obscure variable types
   - "Programming" function objects
3. Python Integration
   - General
   - Approximate the pressure drop
4. Plugin-functions
   - Why plugin-functions
   - The Game of Life plugin
   - Testing the plugin
   - Additional functions
   - Developing your own functions
5. Conclusion

Outline

1. Introduction
   - About this presentation
   - What we're working with
   - Before we start
2. Programming-like structures
   - Stored variables
   - Global variables
   - More obscure variable types
   - "Programming" function objects
3. Python Integration
   - General
   - Approximate the pressure drop
4. Plugin-functions
   - Why plugin-functions
   - The Game of Life plugin
   - Testing the plugin
   - Additional functions
   - Developing your own functions
5. Conclusion

What is going to be thrown at you

- Programming in/with swak4Foam
  - Using the "programming-like" features in swak
    - And a bit of stuff that is useful when you develop your own solver
  - Python integration
  - Writing plugin-functions

Intended audience and aim

- Intended audience for this presentation:
  - people who already worked with swak4Foam
  - know some programming
    - either Python or C++ would be good
- Aim of the presentation
  - Explain some concepts necessary for the advanced usage of swak4foam
  - Show possibilities and limitations of the Python-integration
  - Demonstrate how to write functions that integrate into swak-expressions
- The presentation walks the user through two examples that demonstrate the concepts discussed
  - Sources to the examples are available separately

Outline

1. Introduction
   - About this presentation
   - What we're working with
   - Before we start
2. Programming-like structures
   - Stored variables
   - Global variables
   - More obscure variable types
   - "Programming" function objects
3. Python Integration
   - General
   - Approximate the pressure drop
4. Plugin-functions
   - Why plugin-functions
   - The Game of Life plugin
   - Testing the plugin
   - Additional functions
   - Developing your own functions
5. Conclusion

What is swak4Foam

- If you have to ask you're probably in the wrong presentation
- But lets explain it anyway

What is `swak4Foam`. Take 2

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

Outline

1. Introduction
   - About this presentation
   - What we're working with
   - Before we start
2. Programming-like structures
   - Stored variables
   - Global variables
   - More obscure variable types
   - "Programming" function objects
3. Python Integration
   - General
   - Approximate the pressure drop
4. Plugin-functions
   - Why plugin-functions
   - The Game of Life plugin
   - Testing the plugin
   - Additional functions
   - Developing your own functions
5. Conclusion

Command line examples

- In the following presentation we will enter things on the command line. Short examples will be a single line (without output)

```
ls $HOME
```

- Long examples will be a 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"

```
> this is an example for a very long command line that does not fit onto one line of <brk>
      <cont>the slide 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 <brk>
      <cont>its glory
```

Work environment

- You will use two programs
  - A terminal
  - A text-editor
- For the text-editor you have the choice (these should be installed):
  - Emacs (king of text-editors)
  - VI (my brother uses it. So. OK)
  - Kate with KDE
  - Gedit with Gnome
- It is assumed that you use the Workshop-USE-stick with pre-installed
  - foam-extend
  - swak4foam

Preparing the shell

- During the remaining presentation we assume that
  - the `zsh` is used (optional. `bash` works too)
  - we use `foam-extend 3.1` (required)
- Switch to `zsh`

```
zsh
```

- You should see a more colorful prompt with `(OF:-)` on the left
  - Only with correct environment set (probably only on the stick)
- Switch on Foam-Extend-3.1

```
f31
```

- Now the prompt should show `(OF:3.1-Opt)`
- Create a working directory and go there

```
mkdir swakProgramming; cd swakProgramming
```

Getting the examples

- Download the examples from the openfoamwiki.net

```
wget http://openfoamwiki.net/images/5/50/PyFoamProgramming_O
```

- or (same, but shorter)

```
wget http://bit.ly/1ioH6ix -O PyFoamProgramming_OFW9.tar.gz
```

- Extract examples

```
> tar xzf PyFoamProgramming_OFW9.tar.gz
> ls
PyFoamProgramming_OFW9.tar.gz
calcDrop/
gameOfLife/
```

Make sure `swak4Foam` is installed

- Call the most popular utility of `swak4Foam`
  - `swakVersion` reported below the usual header

```
> funkySetFields
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     3.1                                |
|   \\  /    A nd           | Web:         http://www.extend-project.de       |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
Build    : 3.1
Exec     : funkySetFields
Date     : Jun 07 2014
Time     : 18:35:01
Host     : bgs-cool-greybook
PID      : 11491
CtrlDict : /Users/bgschaid/OpenFOAM/foam-extend-3.1/etc/controlDict
Case     : /Volumes/Foam/Cases/Zagreb2014
nProcs   : 1
SigFpe   : Enabling floating point exception trapping (FOAM_SIGFPE).

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
swakVersion: 0.3.1 (Release date: Next release)
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //


--> FOAM FATAL ERROR:
funkySetFields: time/latestTime option is required


    From function main()
    in file funkySetFields.C at line 643.

FOAM exiting
```

Installing a development version of `swak4Foam`

- New features of `swak4Foam` are published to the development repository in advance
  - Beware: *Development* means "can be broken/unstable" (but doesn't have to be)
- Installing/Compiling should be simple three-step:
  1. Pull the repository
  2. Change to the correct branch
     - Usually `port_2.0.x` is a good guess
  3. Compile

Don't do that now. Do it at home

```
> hg clone http://hg.code.sf.net/p/openfoam-extend/swak4Foam
> cd swak4Foam
> hg update port_2.0.x
> ./Allwmake
```

Python-integration

- Usually the `python`-integration is not configured out of the box
  - Needs the location of the library and the headers
    - Different on different systems
  - In toy-distributions the headers are not part of the regular python-pagckage
    - For instance Ubuntu: you need `python-dev`
- If the compile-script finds a file `swakConfiguration` it sources it to set special variables
  - This is the place to set up Python-support
    - And other things (see `README`)
  - `Allwmake` will automatically compile the Python-support

`swakConfiguration` for Ubuntu

```
export SWAK_PYTHON_INCLUDE="-I/usr/include/python2.7"
export SWAK_PYTHON_LINK="-lpython2.7"

export SWAK_COMPILE_GRAMMAR_OPTION="-O1"
```

Upgrading `swak4Foam`

- Unfortunately the version on the USB-stick comes without
  - Python-support
  - the sources (needed for plugin-development)
- We're going to install packages to override this
  - They'll be gone when you reboot

```
> wget http://bit.ly/1lSv2RJ -O swak.deb
> wget http://bit.ly/1uLsbzn -O swak_dev.deb
> sudo dpkg -i --force-all *.deb
```

Strömungsforschung GmbH
ICE
Innovative Computational Engineering

Outline

1. Introduction
   - About this presentation
   - What we're working with
   - Before we start
2. Programming-like structures
   - Stored variables
   - Global variables
   - More obscure variable types
   - "Programming" function objects
3. Python Integration
   - General
   - Approximate the pressure drop
4. Plugin-functions
   - Why plugin-functions
   - The Game of Life plugin
   - Testing the plugin
   - Additional functions
   - Developing your own functions
5. Conclusion

Limitation of "regular" swak-expressions

- In the usual use swak-expressions are limited
  - The are always executed
    - No way to avoid execution (for instance if a field is not present)
  - They are executed "in the moment"
    - No sense of the past (aka *storage*)
    - The `variables` are just a way to make the expressions simpler
  - They live alone
    - No way to communicate with other parts of `swak`
- This section introduces facilities in `swak` to work around this limitations
  - They make `swak4Foam` an almost complete programming language
    - But it is not *Turing-complete*

What is *Turing-complete* ?

- Google it!

- You don't know "Pearls before swine"?
  - Google it

Outline

1 Introduction
About this presentation
What we're working with
Before we start
2 Programming-like structures
Stored variables
Global variables
More obscure variable types
"Programming" function objects
3 Python Integration
General
Approximate the pressure drop
4 Plugin-functions
Why plugin-functions
The Game of Life plugin
Testing the plugin
Additional functions
Developing your own functions
5 Conclusion

Purpose of stored variables

- Stored variables are like regular variables, but keep their value
- Possible applications:
  - "Did `alpha1` reach this part of the patch yet?"
  - "What is the highest pressure in this cell ... ever?"
  - "What is the accumulated mass-flow through this `faceSet`?"

Using stored variables

- Used in any other expression like regular variables
  - Say the name and use the value
    - Value from the previous time-step
  - Assign values to it
    - Which can be used at the next time-step
- To use a variable as *stored* it has to be declared
  - Optional dictionary `storedVariables` with list of dictionaries does this. Entries are

name the name of the variable

intialValue Value to be used at the first time-step

Example of a stored variable declaration

boundaryField of 0/wet

```
wall {
      type groovyBC;
      storedVariables (
 {
      name isWet;
      initialValue "0";
 }
      );
      variables (
 "isWet=(alpha1>0.5)␣?␣1␣:␣isWet;"
      );
      valueExpression "isWet";
}
```

Strömungsforschung GmbH
ICE
Innovative Computational Engineering

Schizophrenia of stored variables

- Stored variables have two different values
  - The values you set
  - The value you get
- At the end of the time-step the value you set becomes the variable you get
  - The last set value is used
- This distinction is important for expressions that may be called multiple times per time-step
  - For instance if the solver does more than one iteration per time-step
- This behavior is what you want most of the time
  - Example: You don't want the cumulative mass-flow to depend on the number of corrector-iterations
- But it may lead to unexpected behavior if you want to do something like (s is stored):

```
"s=s+val1; s=s+val2;"
```

You'll probably want

```
"s=s+val1+val2;"
```

or

```
"tmp=val1; tmp=tmp+val2; s=s+tmp;"
```

Restarting and limitiation

- Stored variables survive case restarts
  - In `groovyBC` restarting information is written into the field-file
  - Function objects etc write that information into a special sub-directory of the time-directory
- Stored variables work in parallel
  - Every processor stores its part of the information
  - Reconstructing and repartitioning not supported
- Dynamic meshes are not supported
  - Values would have to be remapped
    - Hard to do in a general way

Outline

1. Introduction
   - About this presentation
   - What we're working with
   - Before we start
2. Programming-like structures
   - Stored variables
   - Global variables
   - More obscure variable types
   - "Programming" function objects
3. Python Integration
   - General
   - Approximate the pressure drop
4. Plugin-functions
   - Why plugin-functions
   - The Game of Life plugin
   - Testing the plugin
   - Additional functions
   - Developing your own functions
5. Conclusion

Use cases

- Sometimes a swak-entity would like to access results from another entity
  - For example: multiple `groovyBC`-boundary conditions want to know if the valve is opened or shut
    - To have a consistent implementation *opened* or *closed* should be calculated in one place
  - One function object needs information from another
    - We'll see an example for that later
- Global variables allow implementing such things

Using global variables

- That is the easy part:
  - Specify a list `globalScopes`
  - If a variable is not found in the local scope then these scopes are searched one after another
  - Scopes allow organizing global variables by source/purpose
  - If a scope does not exist: Failure

In some `0/U`

```
globalScopes (
      outletState
      inletState
);
```

Declaring global variables/scopes

- Not everyone can do it
  - Only special function objects
    - Usually in `swakFunctionObjects`
- Usually: `addGlobalVariable`

  globalScope Name of the scope to specify
  globalVariables Dictionary with the variables in that scope.
  Entries include:
  valueType whether the variable is a `scalar`, `vector` etc
  value single value of type `valueType`.
- In the beginning global variables are uniform

Detour: uniform (single) values variables

- swak-variables have a `valueType`
- they can be single values (uniform) or non-uniform

  single values just one value
  - these variables can be used everywhere
  - can be overwritten with non-single values too

  non-uniform fields field with *N* different `valueType`
  - can only be used in expressions where *N* matches the other operands
  - for instance: in an `internalField` *N* must match the number of cells
  - this constraint is checked on processors for parallel runs
- swak tries to use single values wherever possible
  - `min`, `max` etc try to return single values

Specifying some global variables

functions in controlDict

```
defineState {
    type addGlobalVariable;
    outputControl timeStep;
    outputInterval 1;

    globalScope outletState;
    globalVariables {
closed {
    valueType scalar;
    value 0;
}
airReachedOutletTime {
    valueType scalar;
    value -1;
}
shutdownTime {
    valueType scalar;
    value 1;
}
    }
}
```

Setting global variables

- Usually with `calculateGlobalVariables` function object
  - Works like `swakExpression` without `expression`
  - Sets variables for one scope
    - Specified with `toGlobalNamespace`
- The entries in `variables` are evaluated as usual
- Variables in the list `toGlobalVariables` are copied to the global scope
  - Under that name

Example: open valve if sensor point is reached

```
openIfSensorReached {
    type calculateGlobalVariables;
    valueType set;
    setName sensor;
    toGlobalNamespace outletState;
    globalScopes (
outletState
    );
    set {
type swakRegistryProxy;
axis y;
setName sensor;
    }
    toGlobalVariables (
closed
airReachedOutletTime
    );
    variables (
"state=average(alpha1);"
"thresA=0.9;"
"opening=(closed>0.5 && state>thresA)?1:0;"
"closed=(opening>0.5) ? 0 : closed;"
"airReachedOutletTime=(opening>0.5) ? -1 : airReachedOutletTime;"
    );
}
```

Writing global variables

- Function object `dumpSwakGlobalVariable` dumps the value of a global variable to file
  - Rarely needed

Outline

1 Introduction
- About this presentation
- What we're working with
- Before we start

2 Programming-like structures
- Stored variables
- Global variables
- More obscure variable types
- "Programming" function objects

3 Python Integration
- General
- Approximate the pressure drop

4 Plugin-functions
- Why plugin-functions
- The Game of Life plugin
- Testing the plugin
- Additional functions
- Developing your own functions

5 Conclusion

Delayed variables

- Acts like a stored variable but the written values are only used after a delay t_{delay}
  - Usage example: sensor with a reaction time
- Delayed variables are declared like stored variables in a list `delayedVariables`:

  name guess

  delay the delay time t_{delay} (fixed value)

  startupValue value used during the first t_{delay} seconds of the simulation when no value is (yet) available

  storeInterval interval Δt in which values are going to be stored. Times between that are interpolated (linear).
- For an example see the `cleaningTank`-examples in `Example/FromPresentations`

Stacked variables

- These are very obscure
- Purpose of these is to collect multiple values into one array
  - Probably for consumption in another function object
- Setting the variable value appends a value to the variable
- Comes in two flavors

  StackExpressionResult Variable is reset at the start of each timestep
    - Application: collecting values from different sources/locations

  StoredStackExpressionResult Variable keeps its value between steps
    - Application: Collects values from different time-steps to check convergence

Words of caution

- All these variable types (stored, global, delayed . . .) are very useful but
- . . . also very dangerous:
  - Storing them costs memory. If used carelessly they quickly fill up your machine even for moderately-sized cases
    - Example: a delayed variable on an `internalField` with a small `storeInterval`
- `swak4Foam` currently has no memory holes (that I know of) but with these variables you can make it look like it
- Variables "with memory" have not been tested with dynamic meshes
  - Probably won't work

Outline

1. Introduction
   - About this presentation
   - What we're working with
   - Before we start
2. Programming-like structures
   - Stored variables
   - Global variables
   - More obscure variable types
   - "Programming" function objects
3. Python Integration
   - General
   - Approximate the pressure drop
4. Plugin-functions
   - Why plugin-functions
   - The Game of Life plugin
   - Testing the plugin
   - Additional functions
   - Developing your own functions
5. Conclusion

What means "programming"?

- This means function objects that
  - Store data
  - Control the execution of other function objects
- Also stuff that doesn't fit anywhere else
  - Especially if it helps programmers
- This section will only say "it is there"
  - Search the examples for usage examples

readAndUpdateFields

- Reads a field from disc at the start of the simulation
  - Keeps it in memory
    - Others can access it by name
  - Updates the boundary conditions at the end of each timestep
  - For updating the internal field use a `manipulateField` function object
- Typical applications:

  stand-in you have a boundary condition/function object that needs a special field that the current solver/utility does not provide

  recording for post-processing the loaded field has a `groovyBC` that calculates a new value. This value can later be used
    - Example: store wall-shear-stress for output
    - Example: use a stored variable to record the distribution of the maximum temperature on the patch

panicDump

- Sometimes the simulation diverges. You see it coming but have no idea why because nothing is written to disc
- This is an oldie:
  - Check limits for certain fields
  - If they're outside the "comfort zone" specified by the user:
    - Write all fields and stop the solver
  - "Comfort zone" depends on the solver:
    - For instance: maximum velocity higher than the speed of sound for an incompressible solver
    - Or: simulation of your tea-cup reaches temperatures that would melt steel
- New feature:
  - If optional parameter `storeAndWritePreviousState` is set to `true` then the N previous time-steps are also written
    - Application: "why did that cell freak out?"

`writeAndEnd`-function objects

- There is a number of function objects that start with `writeAndEnd`
- They generalize the idea of `panicDump`:
  - Check for a condition and end the run if the condition is fulfilled
- The concrete implementations offer as conditions:
  - Field ranges
  - swak-expressions
  - Python-programs
- With these it is easy to program conditions like: "If the time is bigger than 42 and the amount of H_2O is bigger than 0.1 stop. Or if ..... But not if ..."

Caution with stored time-steps

- In the next `swak4Foam`-version a number of function-objects offer the possibility to store and write old time-steps
- It can't be stressed enough: this needs memory. And lets be clear: lots of it
- Basically they work by
  - Going to the object registry
  - Saying: "What have you got"
  - Copying all into a separate registry
  - In case of an event writing that stuff
  - If no longer needed discarding it
- So if you want to write the 10 last timesteps expect memory usage to rise by a factor of 10
- Don't use in production runs

writeOldTimesOnSignal

- Sometimes panicDump doesn't work anymore because a signal was raised
  - Floating point exception
  - Segmentation fault
  - etc
- But it would be nice to see the latest state
  - Of course you could simply write all time-steps
    - But that takes time
    - State at the time of failure is unknown
- In these cases writeOldTimesOnSignal can be used
  - It replaces the standard signal-handler
  - When the signal is raised it writes stored old time-steps and the current state
  - Afterwards calls the original handler
- If you're hardcore you can even intercept Ctrl-C
  - the SIGINT-signal
- Don't use in production runs

Conditional writing with `writeIf`

- Function objects starting with `writeIf` offer the possibility to very flexibly write additional timesteps
- Acts in phases
  1. Does nothing and waits for a trigger
  2. Trigger "fired" starts writing until another condition is fulfilled
     - number of steps, general condition ...
     - can also write N stored timesteps
  3. After writing offers a "cooldown" period
     - So it doesn't start writing immediately
  4. Go back to step 1
- Possible application "If lagrangian particles impinge the fluid surface write the last 3 time-steps and the next 10 timesteps. Then wait $0.5s$ before recording the next incident"

Multiple function objects that store stuff

- You've seen that there are multiple function objects that can save the last N time-steps
- If you use more than one of these `swak4Foam` notices these
  - Fails
  - User has to set a special parameter to acknowledge "I know that I'm wasting memory"

Function object list proxy

- There is a number of function objects in `swak` that contain other function objects
- If these function objects are executed then they execute that list
- Usually these have two entries:

  functions a list/dictionary that specifies the function objects "inside". Works like the regular entry in `controlDict`

  readDuringConstruction Should `functions` be read at the start or when the function object is executed the first time
    - If `false` it is possible that `functions` are never constructed. This is sometimes the desired effect
- Does not work for all function objects because some function objects do not want to be contained

Conditional function objects

- These usually start with `executeIf`
  - Have a condition that depends on the concrete function-object
  - If that condition is `true` the `functions` are executed
  - If `false` an optional list `else` is executed
- Applications:
  - Function objects should only be executed under certain conditions ("Only execute this `sample` after the 200th iteration")
  - Function objects should be only executed under certain conditions (for instance: only with a certain solver, but not when initializing with `potentialFoam`) but you don't want to comment out
- Example: `executeIfExecutableFits` only executes if the name of the program fits a pattern (the `potentialFoam`-example)
- Guesses from the audience: what might `executeIfEnvironmentVariable`, `executeIfParallelSerial`, `executeIfSwakExpression` do?
  - There are more

dynamicFunctionObjectList

- If you like *Meta-programming* then you'll love this
  - If you don't know what *Meta-programming* is: see the "Pearls before swine"-cartoon above and act accordingly
- What it does:
  - Gets a text from a "provider"
  - Constructs the `functions` from that text
- Providers can be:
  - a regular dictionary file
  - a command (for instance a shell-script)
  - a Python-program
- Applications:
  - Construct function objects from information that is only available when the run starts
  - Construct multiple similar function objects (For instance: construct 20 sample-planes with evaluations on it. You don't want to do that by hand)

Time-manipulation

- There are function objects that allow manipulating the time
- They start with

  setDeltaT sets the time-step to a different value

  setEndTime set the `endTime` to a different value than in the `controlDict`
- Application: implement dynamic time-stepping for solvers that don't support it
  - Or be more flexible about it: "Valve opens in $10^{-3}s$. Lets prepare by decreasing the time-step"
- Does some things that might be considered illegal
  - Possible conflicts with regular adaptive time-steppin
- Currently only Python-variants implemented
  - And one where Δt can be set using a time-line file

Outline

1 Introduction
- About this presentation
- What we're working with
- Before we start

2 Programming-like structures
- Stored variables
- Global variables
- More obscure variable types
- "Programming" function objects

3 Python Integration
- General
- Approximate the pressure drop

4 Plugin-functions
- Why plugin-functions
- The Game of Life plugin
- Testing the plugin
- Additional functions
- Developing your own functions

5 Conclusion

Outline

1 Introduction
About this presentation
What we're working with
Before we start
2 Programming-like structures
Stored variables
Global variables
More obscure variable types
"Programming" function objects
3 Python Integration
General
Approximate the pressure drop
4 Plugin-functions
Why plugin-functions
The Game of Life plugin
Testing the plugin
Additional functions
Developing your own functions
5 Conclusion

Python-integration in swak4Foam

- Until now we heard multiple time "there is also a Python-implementation" for it
  - That is part of the Python-integration
- So what is the Python-integration?
  - It is the library `libwakPythonIntegration.so`
- And what does that do?
  - Provides a number of function objects that integrate a Python-interpreter
  - Injects data into the Python-namespace
  - Executes python-programs
    - Can use (almost) any Python-library (GUI-stuff for instance is problematic)
  - Puts python variables into the swak-namespace
- Python-integration is one of the few things completely documented in the *swak4Foam Incomplete Reference Guide* that comes with the sources

Why Python?

- Just some reasons against it
  - It is generally slower than C++
  - It can't access all the stuff C++ can
    - But `pythonFlu` can. What is `pythonFlu`? The rat says "Google it!"
- Some reasons for it
  - It is easier than C++
  - There are a lot of libraries for it
    - Numerical
    - Databases
    - Web-stuff
    - if the library is installed on your system then you don't have to worry about compiling, linking etc
  - Run-time diagnostics is fun
  - Our friend `ParaView` uses it too
- And the best reason:
  - It is named after Monty Python

Other language integrations

- Currently none available
- Other languages that allow being embedded into a C-program are possible
  - Lua
  - Perl
  - Ruby
  - ...
- But unless there is a need it probably won't happen
  - Exchange with swak would have to be generalized
    - So that integrations don't have to do everything from scratch
    - To have some consistency in the interface

What is python

- Is a programming language
  - interpreted (not compiled)
  - object-oriented
    - but pragmatic about it
- It is widely used
  - And integrated as the scripting language in a number of applications
- Comes with a large standard-library
- Many third-party libraries available
  - Most interestingly for numerical purposes
    - Mostly based on `numpy`
  - But also plotting
  - Many C++/C-packages have Python-bindings
    - Allows using them as if they were Python-packages

3 things you need to know about Python

... to understand the examples (if you only know C++)

1. Indentation does the same thing { and } do for C++
2. [] is a list and also the access operator
3. {} creates dictionaries
4. `self` is the same as `this` for C++ (the object itself)

OK. That's 4.
I didn't expect the Spanish Inquisition. "Nobody expects ...".
If you don't understand it: Google `monty python spanish inquisition`. First YouTube-link

What is numpy

- It is the de-facto standard for handling numerical data in Python
  - Advanced libraries like `scipy` based on it
- The main feature is that it makes C-arrays look like Python-lists
  - "Vectorizes" operations on them (executes the loop in C)
    - That way they are almost as fast as C

This is slow

Assuming that `a`, `b` and `c` are `numpy`-arrays of equal length

```
for i in range(len(a)):
    a[i]=b[i]+c[i]
```

This is fast

```
a=b+c
```

And also easy to read

Implementation of the Python-integration

- The first function-object that uses Python initializes the interpreter
  - Every other function object uses that but gets a separate work-space
    - Technical reason: there can only be one interpreter per process
    - Separate workspaces means: no interference (what you see is yours)
    - Library imports are shared
- The namespace "lives" through the whole lifetime of the function-object:
  - A variable set at one time-step still has the same value at the next
- Control is handed over to the Python-interpreter at the "usual" times
  - `start`, `execute`, `write`, `end`

Before the Python-code - variables

- swak injects variables into the Python-namespace that might be of interest (for complete list see Reference Guide)

  runTime the current simulation time as a float
  timeName name of the current time as a string
  caseDir path to the case directory
  parRun boolean that says "is this a parallel run?"

- There are also two functions that create directories for time-dependent data (with slightly different use-cases):

  dataFile(fname) creates a directory `<case>/<name>_data/<time>`.
  timeDataFile(fname) creates a directory `<case>/<time>/<name>_data`.

Before the Python-code - getting data from `swak`

- if a list `swakToPythonNamespaces` is specified then all variables in these namespaces are injected into the Python-namespace
- These rules apply:
  - Single values are copied into Python (a single scalar becomes a single value)
  - Fields are projected as `numpy`-array
    - by-reference: no data is copied. The `numpy`-array just "points" to the OpenFOAM-data
    - `scalarField` becomes a 1D-array
    - `vectorField` of length N becomes a $N \times 3$-array
    - `tensorField` a N-array
    - what about `symmTensorField`?
    - Fields have convenience-attibutes like `.x` for the first column

Consequence of *by-reference*

- Changing single values affects the global variable
  - Sometimes that is what is wanted
- Overwriting a global variable `a` is a bit counter intuitive
  - `a=1` does not reset the whole field to 1. It sets `a` to a new variable with the single value 1
    - Global variable `a` is still alive and keeps its value
  - Slice operation `a[:]=1` resets the global variable
    - This makes perfect sense if you understand how `numpy` handles things
  - Same for `b.x`: to clear it "slice": `b.x[:]=0`

Predefined imports

There is a number of options on the swak-side that import certain libraries (the user code then doesn't have to do it)

useNumpy automatically use `numpy`. Without it only single values from swak are handled

useIPython for the interactive interpreter use `ipython` (highly recommended)

importLibs Optional dictionary with libraries to import at start.

- A single value `scipy` means `import scipy`
- A key/value-pair `stat scipy.stat` means `import scipy.stat as stat`

Importing libraries that way may help working around problems with libraries behaving strangely when imported from user-code

Options for interactivity

- One of the nice things about Python is the interactive shell
  - Try out things. If they work copy them to your program
  - With `ipython` even easier
- The relevant options are

interactiveAfterExecute After the user-code has executed the user is dropped to an interactive shell (`ipython` if specified).
  - This is useful during the development of user-code to try things out
  - Shell is ended with `Ctrl-D`. OpenFOAM continues
  - Careful: `Ctrl-C` will end the whole run

interactiveAfterException If an exception is raised by the Python-code the user gets an interactive shell
  - Very useful for debugging user-code

Parallel support

- Parallel support of the Python-integration is minimal
  - Advance stuff would have to be handled by the user code with the library `mpi4py`
- In a parallel run it checks a number of options to see what is supported:

  isParallelized run fails if this is unset
    - tells swak: "User thought about it"

  parallelMasterOnly Execute the Python-code only on the Master-processor
- Things like this work without problem in parallel

```
if runTime>3:
    val=1
else:
    val=0
```

- This probably won't (or won't give the expected result)

```
maxP=max(pressure)
```

Specifying user-code

- Depending on the function-object different Python-snipplets have to be specified
- A snipplet `start` can be specified in two ways:
  - **startCode** string with the actual Python-code
    - use for trivial code (or empty string)
  - **startFile** string with a file that has the Python-code
- These options are mutual exclusive
- The snipplets are executed
  - Some function-objects (`setDeltaTWithPython` for instance) expect a value to be returned
    - Snipplet must do so with `return`

Pushing values to global variables

- Two options specify which data is transferred to OpenFOAM:

  pythonToSwakNamespace one global namespace to which the values are transferred

  pythonToSwakVariables a list of variables that are going to be transferred

- Values are transferred in this way:

  single number transferred as a single `scalae`

  3 element list becomes a `vector`

  1D `numpy` array becomes a `scalarField` of size N (values are copied)

  $N \times 3$ **array** becomes a `vectorField` of size N

  - $N \times 9$ and $N \times 6$ become `tensorField` or `symmTensorField`

Outline

1 Introduction
- About this presentation
- What we're working with
- Before we start

2 Programming-like structures
- Stored variables
- Global variables
- More obscure variable types
- "Programming" function objects

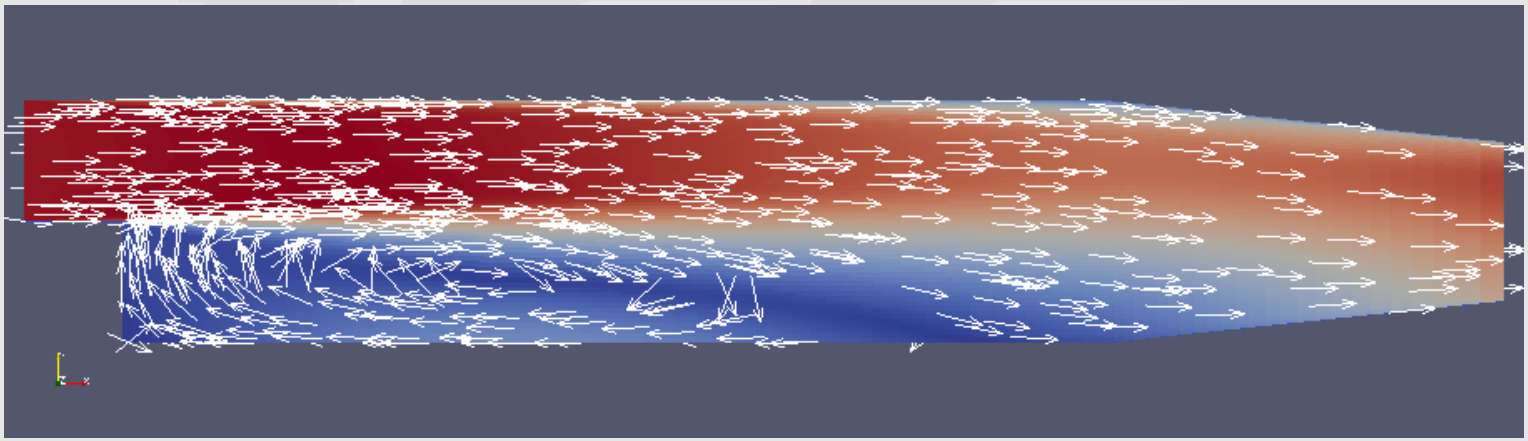
3 Python Integration
- General
- Approximate the pressure drop

4 Plugin-functions
- Why plugin-functions
- The Game of Life plugin
- Testing the plugin
- Additional functions
- Developing your own functions

5 Conclusion

The case

- Take the `simpleFoam` tutorial `pitzDaily`
  - Enhance it with some Python



What we're trying to do

- The assumption is that the pressure between inlet and outlet drops linearly
  - Don't laugh. Please!
- Using simulated pressure values and positions python should calculate k and d for the approximation

$$p(x) = kx + d$$

- Using these values swak calculates a field `pressureLinear` with the theoretical values for post-processing
- As a bonus Python should generate pictures
  - Comparing the fit to the date
  - Evolution of k and d as a function of the iterations

Preparing the case

- In real life we change into the ready-made case

```
cd $HOME/swakProgramming/calcDrop/pitzDailyWithPython
```

- And set it up

```
blockMesh
```

- Lets pretend we don't already have the case

Using pyFoam

```
> cd $HOME/swakProgramming/calcDrop
> pyFoamCloneCase.py $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily <brk>
    <cont>pitzDailyWithPyFoam
> cd pitzDailyWithPyFoam
> pyFoamPrepareCase.py .
```

Getting the function-objects

- 2 libraries have all the function-objects we need
  - Not really. But other libraries are loaded as dependencies

controlDict

```
libs (
    "libswakFunctionObjects.so"
    "libswakPythonIntegration.so"
);

functions {
}
```

Setting up the field pressureLinear

- Load the field into memory in the beginning

controlDict

```
functions {
    linearPressure {
type readAndUpdateFields;
outputControl timeStep;
outputInterval 1;
fields (
    pressureLinear
);
autowrite true;
    }
}
```

The field-file

- For the `read`-part `readAndUpdateFields` needs a field-file in `0`
  - Create by copying `0/p` and then adapting it

0/pressureLinear

```
dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
type            zeroGradient;
    }

    outlet
    {
type            zeroGradient;
    }

    upperWall
    {
```

Don't execute every time

- Executing the Python-program at every iteration would slow the program up
  - And fill the disk with unused pictures
- So we ask to execute it only every 10 timesteps

controlDict after linearPressure

```
every10Steps {
    type executeIfPython;
    outputControlMode timeStep;
    outputInterval 1;
    readDuringConstruction true;
    useNumpy false;
    conditionCode "return (int(runTime) % 10)==0";
    initCode "";

    functions {
    }
}
```

Exercise: change the condition

- During the first iterations the pressure oscillates
  - Results are useless
- Adapt the condition to say "Every 10 timesteps after iteration 50"

Get data from swak

- Get $\vec{u}$, $\vec{x}$ and p into global variables

controlDict in every10Steps

```
functions {
    toGlobalNamespace {
type calculateGlobalVariables;
outputControl timeStep;
outputInterval 1;

valueType internalField;
toGlobalNamespace fieldData;
toGlobalVariables (
    positions
    velocity
    pressure
);
variables (
    "velocity=U;"
    "positions=pos();"
    "pressure=p;"
);
    }
}
```

The central function object

- This is the central function object
  - Because it is so important we split it into three parts
  - First the administrative stuff

controlDict after toGlobalNamespace

```
calculatePressureFit {
    type pythonIntegration;
    useIPython true;
    useNumpy true;
    importLibs {
matplotlib;
stats scipy.stats;
    }
    parallelMasterOnly false;
    isParallelized false;
    interactiveAfterException true;
    interactiveAfterExecute false;
```

Part 2: Specifying the Python-sniplets

- This specifies which code to execute during startup, each timestep, write etc
- `$FOAM_CASE` is set to the current case
  - By specifying the file names like this you make sure the Python-files are always found

`calculatePressureFit` continued

```
startFile "$FOAM_CASE/system/calcPressureStart.py";
writeFile "$FOAM_CASE/system/calcPressureWrite.py";
executeFile "$FOAM_CASE/system/calcPressureExecute.py";
endFile "$FOAM_CASE/system/calcPressureEnd.py";
```

Specifying data exchange

- The global variables go in
- Python writes to global scope `fieldData`
  - A complete field `pLinear`
  - The fitted parameters k (`slope`) and d (`offset`)

`calculatePressureFit` finished

```
    swakToPythonNamespaces (
fieldData
    );
    pythonToSwakNamespace fromPython;
    pythonToSwakVariables (
pLinear
slope
offset
    );
}
```

Setting the calculated field

- We now copy the theoretical pressure values `pLinear` Python calculated into `pressureLinear`

`controlDict` after `calculatePressureFit`

```
writeLinear {
    type manipulateField;
    outputControl timeStep;
    outputInterval 1;
    fieldName pressureLinear;
    mask "true";

    expression "pLinear";

    globalScopes (
fromPython
    );
}
```

Alternate pressure field

- This is redundant
  - Only demonstrates that we could also use the two parameters to calculate the field in swak4Foam

controlDict after writeLinear

```
writeLinear2 {
    $writeLinear;
    type expressionField;
    autowrite true;
    expression "pos().x*slope+offset";
    fieldName pressureLinear2;
}
```

Python code executed in the beginning

- Imports the plotting library
- Sets up variables that collect the evolution of parameters
- Initializes the variables to be exported
  - Necessary because they are always exported to swak
  - `pLinear` becomes field of size of `pressure`

system/calcPressureStart.py

```
print "Python: start"

from matplotlib import pyplot

times=[]
slopes=[]
offsets=[]

slope=0
offset=0

pLinear=0*pressure
```

Executed every time

- The actual work is done by `stats.linregress`
- `pLinear` is calculated with a simple one-liner
- `slope` and `offset` are set from `r` and appended to the time-lists

system/calcPressureExecute.py

```
print "Python: Execute"

times.append(runTime)
r=stats.linregress(positions.x,pressure)
slopes.append(r[0])
offsets.append(r[1])

pLinear=positions.x*r[0]+r[1]

print pLinear
slope=r[0]
offset=r[1]
```

Exercise

- The result of `scipy.stats.linregress` has more than two values
  - Find out what the other values are and which one might describe the accuracy of the linear fit
    - How to find that information: What would the rat do?
- Adapt Python-codes to print this accuracy
  - Print it as a function of time

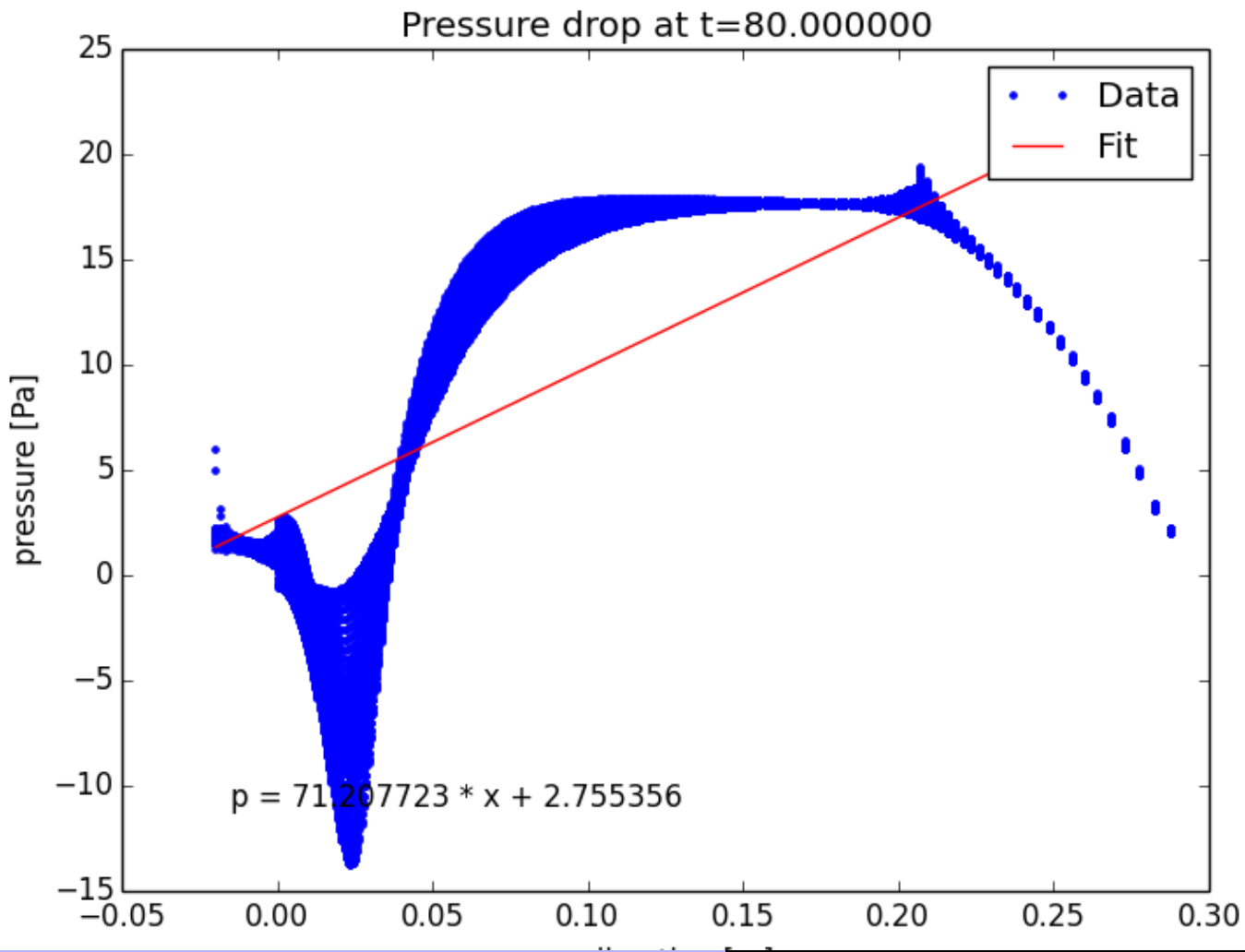
Write pictures of the fit

- This code is executed whenever OpenFOAM writes a time-step to disk
- The actual code is `matplotlib`-specific
  - To get that information even the *goat* would Google

system/calcPressureWrite.py

```
print "Python: write"

f=pyplot.figure()
pyplot.title("Pressure drop at t=%f" % runTime)
pyplot.xlabel('x-direction [m]')
pyplot.ylabel('pressure [Pa]')
data=pyplot.plot(positions.x,pressure,'b.',label="Data")
xLine=numpy.array([min(positions.x),max(positions.x)])
line=pyplot.plot(xLine,r[1]+r[0]*xLine,'r',label="Fit")
a=line[0].get_axes()
pyplot.text(0.1,0.1,"p = %f * x + %f" % r[:2],transform=a.transAxes)
pyplot.legend()
pyplot.savefig("pressuredrop_t=%f.png" % runTime)
```

Picture of a fit



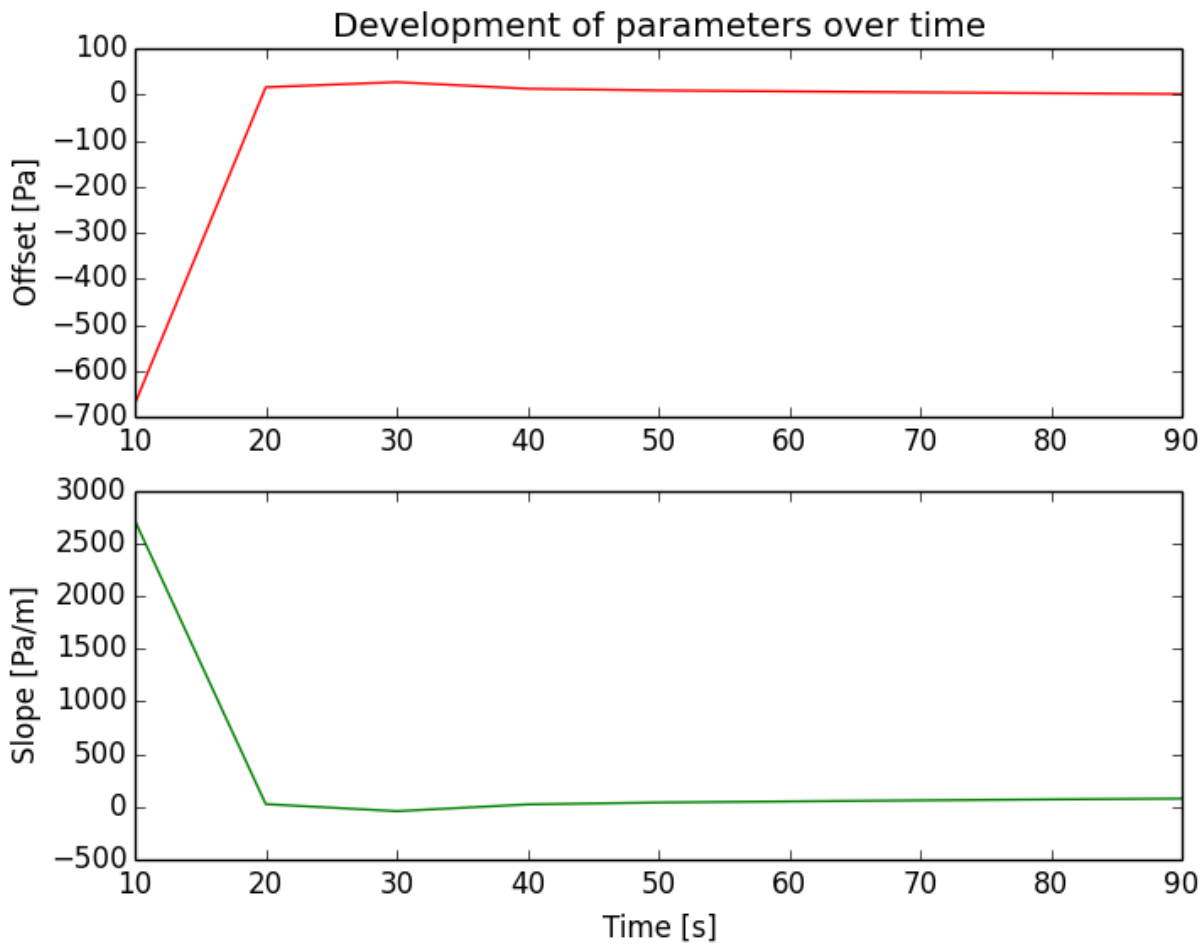
Plotting evolution in the end

system/calcPressureEnd.py

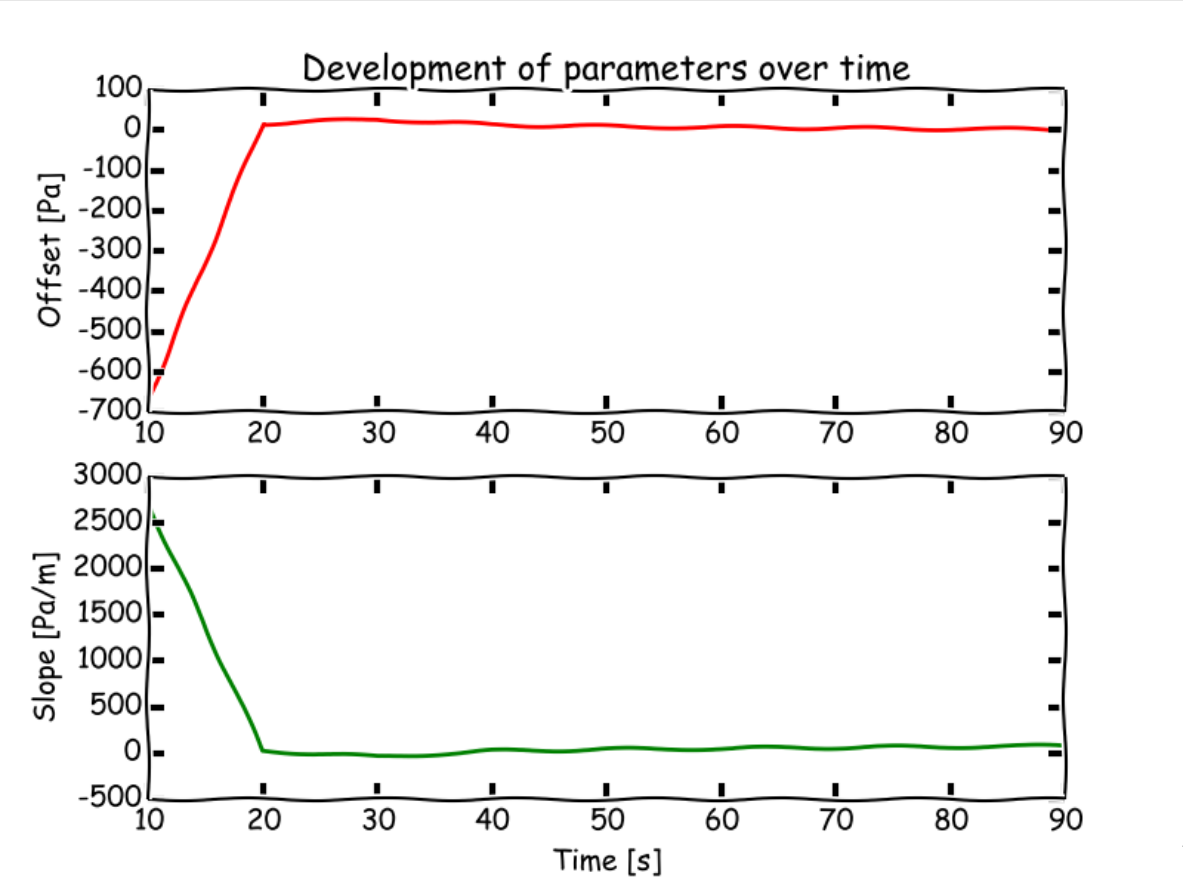
- This is only boring matplotlib-code

```
def makePlot():
    pyplot.figure()
    pyplot.xlabel("Time [s]")
    pyplot.subplot(2,1,1)
    pyplot.title("Development of parameters over time")
    pyplot.plot(times,offsets,"r")
    pyplot.ylabel("Offset [Pa]")
    pyplot.subplot(2,1,2)
    pyplot.plot(times,slopes,"g")
    pyplot.ylabel("Slope [Pa/m]")
    pyplot.xlabel("Time [s]")

makePlot()
pyplot.savefig("ParameterDevelopment.png")
try:
    pyplot.xkcd()
    makePlot()
    pyplot.savefig("ParameterDevelopmentXKCD.png")
except AttributeError:
    print "No XKCD"
```

Development of the parameters



XKCD-style



XKCD? G_____ it!

You don't know

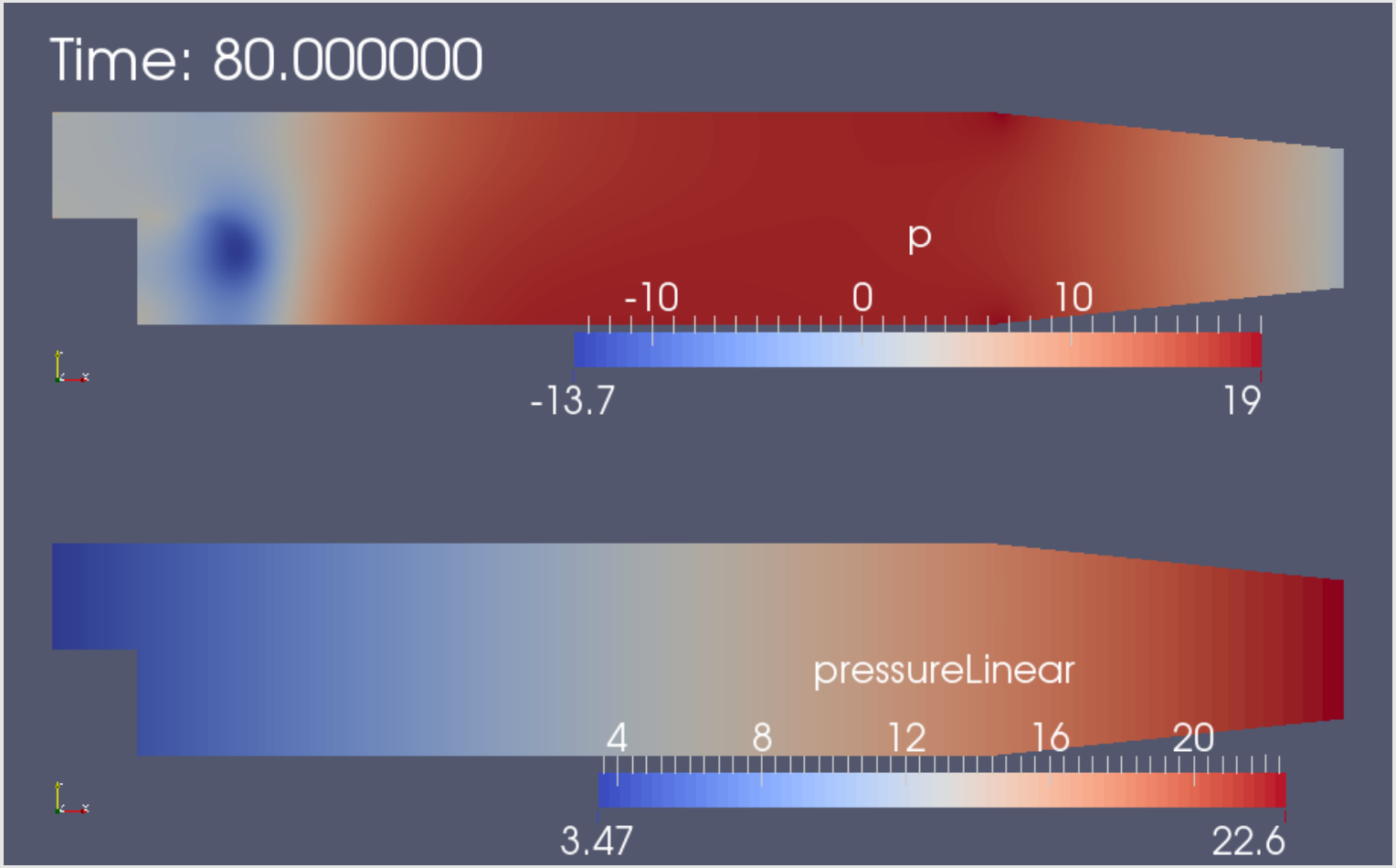
Running the solver

- Now it is time to execute the solver

Excerpts from the solver output

```
> simpleFoam
<<snip>>
DILUPBiCG:  Solving for epsilon, Initial residual = 0.0160891, Final <brk>
    <cont>residual = 0.00103827, No Iterations 1
DILUPBiCG:  Solving for k, Initial residual = 0.0154531, Final residual = <brk>
    <cont>0.000852423, No Iterations 1
ExecutionTime = 10.01 s  ClockTime = 87 s

Python: Execute
[  5.69550741   5.77655262   5.8543598  ...,  21.23283186  21.48525816
  21.74750274]
Python: write
Manipulated field pressureLinear in 12225 cells with the expression "<brk>
    <cont>pLinear"
Manipulated field pressureLinear not rewritten. Set 'writeManipulated'
Creating expression field pressureLinear2 ... type:volScalarField
Time = 61

DILUPBiCG:  Solving for Ux, Initial residual = 0.00452582, Final residual =<brk>
    <cont> 0.00019914, No Iterations 1
```

The "theory" is far off



Strömungsforschung GmbH
ICE
Innovative Computational Engineering

Exercise: different fit

- Doesn't look like a linear drop at all
- One solution: let the simulation converge
- But in our desperation we want to use a higher-order fit
- The function `numpy.polyfit` finds a polynomial fitting that data
- Try fitting the data with a cubic polynomial
  - Get the fit "outside" and visualize it

Interactivity

- Lets have a look at the interactive possibilities by provoking an error
- In `system/calcPressureExecute.py` replace `print pLinear` with `print boo`

Executing again

```
> simpleFoam
<<snip>>
ExecutionTime = 2.73 s  ClockTime = 13 s

Python: Execute
Traceback (most recent call last):
  File "<string>", line 12, in <module>
NameError: name 'foo' is not defined
Python Exception
Got an exception for "# Execute often\
<<snip>>
" now you can interact.
Python 2.7.7 (default, Jun 17 2014, 23:22:44)
Type "copyright", "credits" or "license" for more information.

IPython 2.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

Exception handling

In [1]:
```

Example of an interactive session

- Examining values and getting help

```
In [1]: print r
(2717.674697110564, -670.92006669301622, 0.93911994174592917, 0.0, <brk>
     <cont>8.9934716158273247)

In [2]: min(pressure)
Out[2]: -986.23933174489468

In [3]: velocity.shape
Out[3]: (12225, 3)

In [4]: stats.linregress?
Type:        function
String form: <function linregress at 0x114286ed8>
File:        /opt/local/Library/Frameworks/Python.framework/Versions/2.7/<brk>
     <cont>lib/python2.7/site-packages/scipy/stats/stats.py
Definition:  stats.linregress(x, y=None)
Docstring:
Calculate a regression line

This computes a least-squares regression for two sets of measurements.

Parameters
----------
x, y : array_like
    two sets of measurements.  Both arrays should have the same length.
```

Alternate way of using Python

- Not everybody wants to use Python in there production runs
  - Performance considerations
  - Not possible because of missing libraries
    - Ask your admin to install `matplotlib` with all dependencies on the cluster. If you do: please send me a copy of the mails exchanged
- For such cases there is the utility `funkyPythonPostproc`
  - Allows using Python on data stored on disc
  - Uses the usual function-objects to get data to and from python
- The utility is also nice to develop scripts
- Utility can even be used without python
  - Just for applying function objects

Outline

1 Introduction
About this presentation
What we're working with
Before we start
2 Programming-like structures
Stored variables
Global variables
More obscure variable types
"Programming" function objects
3 Python Integration
General
Approximate the pressure drop
4 Plugin-functions
Why plugin-functions
The Game of Life plugin
Testing the plugin
Additional functions
Developing your own functions
5 Conclusion

Outline

1 Introduction
About this presentation
What we're working with
Before we start
2 Programming-like structures
Stored variables
Global variables
More obscure variable types
"Programming" function objects
3 Python Integration
General
Approximate the pressure drop
4 Plugin-functions
Why plugin-functions
The Game of Life plugin
Testing the plugin
Additional functions
Developing your own functions
5 Conclusion

Functionality needed by some

- Functionality that might be essential to some is not needed by most
  - But it would be nice to have it on demand
- In swak this means:
  - User X says "For my work it would be nice to have this function in swak so that I can use it in `swakExpression` and elsewhere"
    - Problem is: not that many people need that function. So it would bloat the swak-core
  - Or: "I want to use swak to test our in-house combustion code"

The solution: function plugins

- A library is loaded
  - Registers new functions
- Now these functions can be used like "regular" functions in swak-expressions
- Technically these plugin-functions are C++-classes
  - Implement an interface that tells the parser
    - What parameters are required
    - What the function returns
    - Gets the parameter values from the parser
    - Calculates the result and sends it to the parser
- We will discuss an example plugin here
  - There will be a lot of hand-waving
    - It doesn't make sense to wade through pages of C++-code
    - The important parts will be pointed out

C++ if you've never used it before

- That would be a presentation of its own
- Just relax and trust me

Outline

1. Introduction
   - About this presentation
   - What we're working with
   - Before we start
2. Programming-like structures
   - Stored variables
   - Global variables
   - More obscure variable types
   - "Programming" function objects
3. Python Integration
   - General
   - Approximate the pressure drop
4. Plugin-functions
   - Why plugin-functions
   - The Game of Life plugin
   - Testing the plugin
   - Additional functions
   - Developing your own functions
5. Conclusion

The Game of life

- If you don't know it: have a look at `http://en.wikipedia.org/wiki/Conway's_Game_of_Life` for an explanation
- The *Game of Life* was developed/invented/discovered by John Conway
  - It is a cellular automaton on a 2-dimensional grid
  - Each cell has 8 neighbors
  - Each cell is either "alive" or "dead"
  - Cells "live" in the next time-step if currently it is
    alive and 2 or 3 of its neighbors live
    dead and exactly 3 neighbors live
- These simple rules evolve in surprisingly complex patterns

Our implementation

- We want to write a function that
  - Gets a scalar field
  - Values $> \frac{1}{2}$ are assumed to be "alive"
  - Neighbors of a cell are used as neighbors
    - Current implementation only knows neighbors that share a face
    - To correctly implement Conway's Game we'd also need neighbors that share an edge
  - Returns a field with `dead` (value 0) and `live` (value 1) cells
  - Boundary treatments (patch faces are treated as "neighbors"):

    Coupled patches get value from cell on the other side
    Symmetry get the value of the cell itself
    Empty not counted as a neighbor
    all other dead
- Why would we need such function
  - For nothing really. But having a GoL is cool
  - As a pseudo-random distribution generator for complex initial conditions

Declaring the class

- Subclass of `FieldValuePluginFunction` to be used on the parser for `internalField`
  - Different subclasses for `patch` and other parsers

gameOfLifeFunction.H

```
#ifndef gameOfLifeFunction_H
#define gameOfLifeFunction_H

#include "FieldValuePluginFunction.H"

namespace Foam
{

class gameOfLifeFunction
:
    public FieldValuePluginFunction
{
    //- Disallow default bitwise assignment
    void operator=(const gameOfLifeFunction &);

    gameOfLifeFunction(const gameOfLifeFunction &);
```

The constructor

- This is the constructor that needs to be implemented to be able to register the function

gameOfLifeFunction.H

```
// the actual function
gameOfLifeFunction(
    const FieldValueExpressionDriver &parentDriver,
    const word &name
);
```

Registering the function

- For the parser to be able to use it it must be registered with the runtime-selection table
  - This is a standard-technique in OpenFOAM
- Here we set the name of the function to be `gameOfLife_step`

gameOfLifeFunction.C

```
defineTypeNameAndDebug(gameOfLifeFunction,0);
addNamedToRunTimeSelectionTable(FieldValuePluginFunction, <brk>
    <cont>gameOfLifeFunction , name, gameOfLife_step);
```

Constructor of the parent class

- In this constructor the function declares its "signature"
  - How to be called, what it returns
- The parameters of the constructor are
  1. The actual driver (the class in charge of evaluation)
  2. The name of the function
  3. Type of the return value
     - Be sure to really return an object of that type
  4. A string with the specification of the parameters of the function

Parameter specification string

- The parameters in the specification string are separated by ,
- Each parameter specification consists of three parts
  1. Descriptive name of the parameter (for the user)
  2. The parser that swak should use to parse the parameter
     - Any swak-parser is possible (`internalField`, `patch`, etc)
     - Yes: parameters can be full swak-expressions
  3. The type that the parameter is supposed to be
     - `volScalarField` for instance

Restriction: no function overloading

- In C++ you can have functions with the same names but different parameters
  - Return type is determined by the parameters
- For instance: `max(a)` in OpenFOAM
  - If `a` is a `vectorField` then the return type is a `vector`
  - For a `scalarField` it is a `scalar`
- This would be too complicate to implement in swak
  - And there are not that many uses for it
- As a workaround in our example there would have to be two functions
  - `maxVector` and
  - `maxScalar`

Our constructor

- Our constructor looks like this
  - Gets a scalar. Returns a scalar

gameOfLifeFunction.C

```
gameOfLifeFunction::gameOfLifeFunction(
    const FieldValueExpressionDriver &parentDriver,
    const word &name
):
    FieldValuePluginFunction(
parentDriver,
name,
word("volScalarField"),
string("oldState internalField volScalarField")
    )
{
    setConwayNumbers();
}
```

After construction

- After construction of the function the driver continues parsing the input string
  - Tries to extract the parameters according to the specification we provided in the constructor
    - This involves calling another parser
    - This may lead to "stacked" error messages if the sub-parser fails
  - If the parser has a parameter of the correct type he hands it to the function
    - via a `setArgument`-method
    - the function is responsible for storing that value
- After all parameters are parsed the driver ask the function to do the actual evaluation
  - by calling the `doEvaluation`-method
  - that triggers the actual evaluation
  - the `ExpressionResult` is set with the `result`-method
    - That is the "function return"

The missing parts of the interface

- There are other parts, but they are application specific
  - `oldState_` and `doStep` are the only ones that interest us

gameOfLifeFunction.H

```
    autoPtr<volScalarField> oldState_;

protected:

    autoPtr<volScalarField> doStep(const volScalarField &old);

    virtual void setArgument(
label index,
const string &content,
const CommonValueExpressionDriver &driver
    );

    void doEvaluation();
```

Setting the argument

- The argument `index` helps distinguishing multiple arguments of the same type
- `content` is there for technical reasons

gameOfLifeFunction.C

```
void gameOfLifeFunction::setArgument(
    label index,
    const string &content,
    const CommonValueExpressionDriver &driver
) {
    if(index==0) {
oldState_.set(
    new volScalarField(
dynamic_cast<const FieldValueExpressionDriver &>(
    driver
).getResult<volScalarField>()
    )
);
    } else {
FatalErrorIn("gameOfLifeFunction::setArgument("
"label index,"
"const string &content,"
"const CommonValueExpressionDriver &driver)")
    << "This should not happen. Expecting index==0. Got " << index
<< endl
<< exit(FatalError);
    }
}
```

Evaluating the function and setting the result

- This is where the actual work happens
  - For the actual implementation of doStep see the sources
    - We're already behind schedule and it is "standard" OpenFOAM-programming
- `setObjectResult` sets the `ExpressionResult` with the `volScalarField` returned by `doStep`
  - This type must be the same promised in the "signature"

gameOfLifeFunction.C

```
void gameOfLifeFunction::doEvaluation()
{
    result().setObjectResult(doStep(oldState()));
}
```

Locating the swak-sources

- For compiling the function-plugin it must have the sources
- There is no standard-location for that
  - With our packages it is in `$FOAM_SRC/swak4Foam` but this is not the norm
  - Usually the sources are in the user directory
- By convention `Make/options` looks for them in `$SWAK4FOAM_SRC`
  - This is better than hardcoding a path and when you move the sources you've got to change that
- Make sure that `SWAK4FOAM_SRC` points to the correct location

Compile the plugin and go to the case

- Go to the sources, set the environment, compile
  - Pretty easy

On the shell

```
> cd $HOME/swakProgramming/gameOfLife/gameOfLifeFunctionPlugin
> export SWAK4FOAM_SRC=$FOAM_SRC/swak4Foam
> wmake libso
<<snip>>
> ls $FOAM_USER_LIBBIN
libswakGameOfLifeFunctionPlugin.so
> cd ../gameOfDamBreak
```

Strömungsforschung GmbH
ICE
Innovative Computational Engineering

Outline

1 Introduction
About this presentation
What we're working with
Before we start
2 Programming-like structures
Stored variables
Global variables
More obscure variable types
"Programming" function objects
3 Python Integration
General
Approximate the pressure drop
4 Plugin-functions
Why plugin-functions
The Game of Life plugin
Testing the plugin
Additional functions
Developing your own functions
5 Conclusion

The case gameOfDamBreak

- This is the standard damBreak-case
  - Slightly modified: symmetry on the left
- Set up to be used with pyFoamPrepareCase.py
  - First executes setFields with the "usual" initial condition
  - Then calls funkySetFields

caseSetup.sh

```
#! /usr/bin/env bash

setFields

funkySetFields -time -0 -expression "gameOfLife_step(alpha1)" -field alpha1<brk>
    <cont> -keepPatches
```

Setting up the case

- We set up th case with

```
pyFoamPrepareCase.py .
```

- The most interesting part about the output is where swak reports the available plugin functions
  - This is the "online help" for us
  - The signature is constructed from the information we provided in the constructor

```
"Loaded plugin functions for 'FieldValueExpressionDriver':"
  gameOfLife_step:
    "volScalarField gameOfLife_step(internalField/volScalarField oldState)"

swak4Foam: Setting default mesh
 Setting 2268 of 2268 cells
 Writing to "alpha1"
End
```

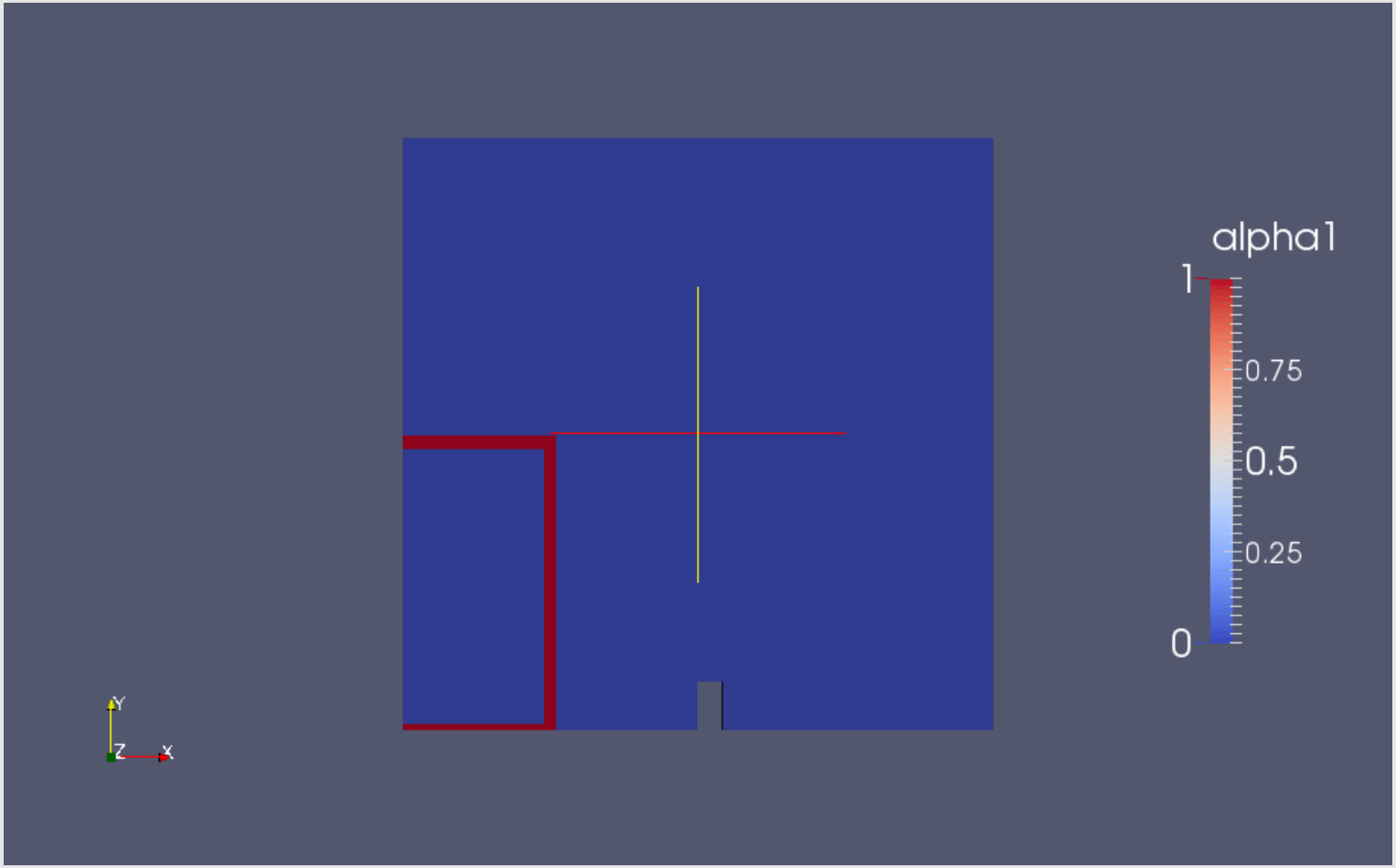
Playing around with the function

- We create some more fields by applying the function
  - And make a mistake

```
> funkySetFields -time -0 -expression "gameOfLife_step(gameOfLife_step(alpha1))" -<brk>
    <cont>field alpha2 -create
<< snip >>
> funkySetFields -time -0 -expression "gameOfLife_step(gameOfLife_step(random()-0.25)<brk>
    <cont>+pos().y)" -field alpha3 -create
<< snip >>
--> FOAM FATAL ERROR:
 Parser Error for driver FieldValueExpressionDriver at "1.1-6" :"field random not <brk>
     <cont>existing or of wrong type"
"random()-0.25)+pos().y)"
  ^^^^^^
--|

Context of the error:


- Driver constructed from scratch
  Evaluating expression "gameOfLife_step(gameOfLife_step(random()-0.25)+pos().y)"
  Plugin Function "gameOfLife_step" Substring "gameOfLife_step(random()-0.25)+pos().y<brk>
      <cont>)"
- Driver constructed from scratch
  Evaluating expression "gameOfLife_step(random()-0.25)+pos().y)"
  Plugin Function "gameOfLife_step" Substring "random()-0.25)+pos().y)"
- Driver constructed from scratch
  Evaluating expression "random()-0.25)+pos().y)"

> funkySetFields -time -0 -expression "gameOfLife_step(gameOfLife_step(rand()-0.25)+<brk>
    <cont>pos().y)" -field alpha3 -create
```

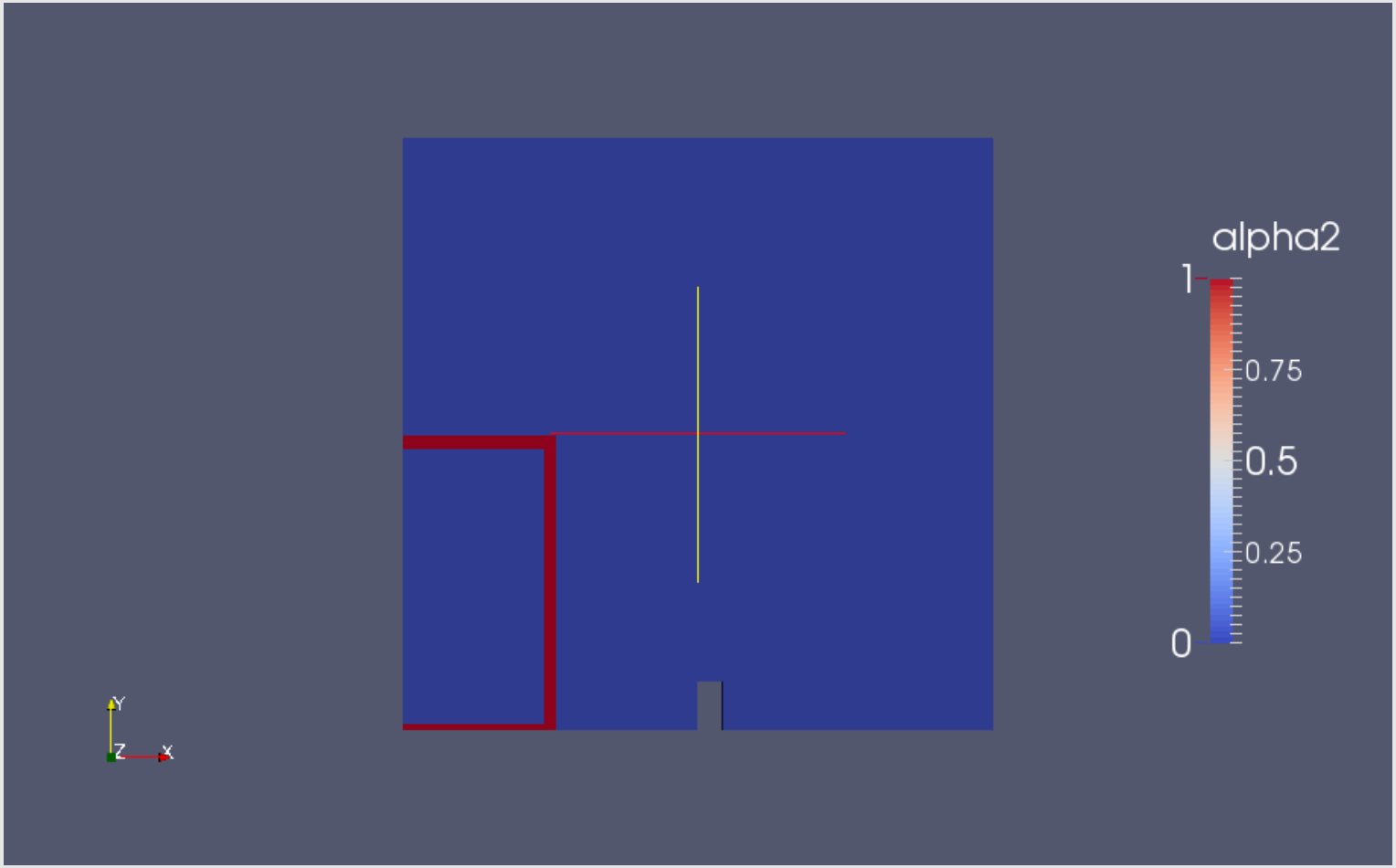
Strömungsforschung GmbH
ICE
Innovative Computational Engineering

Boundary of the original



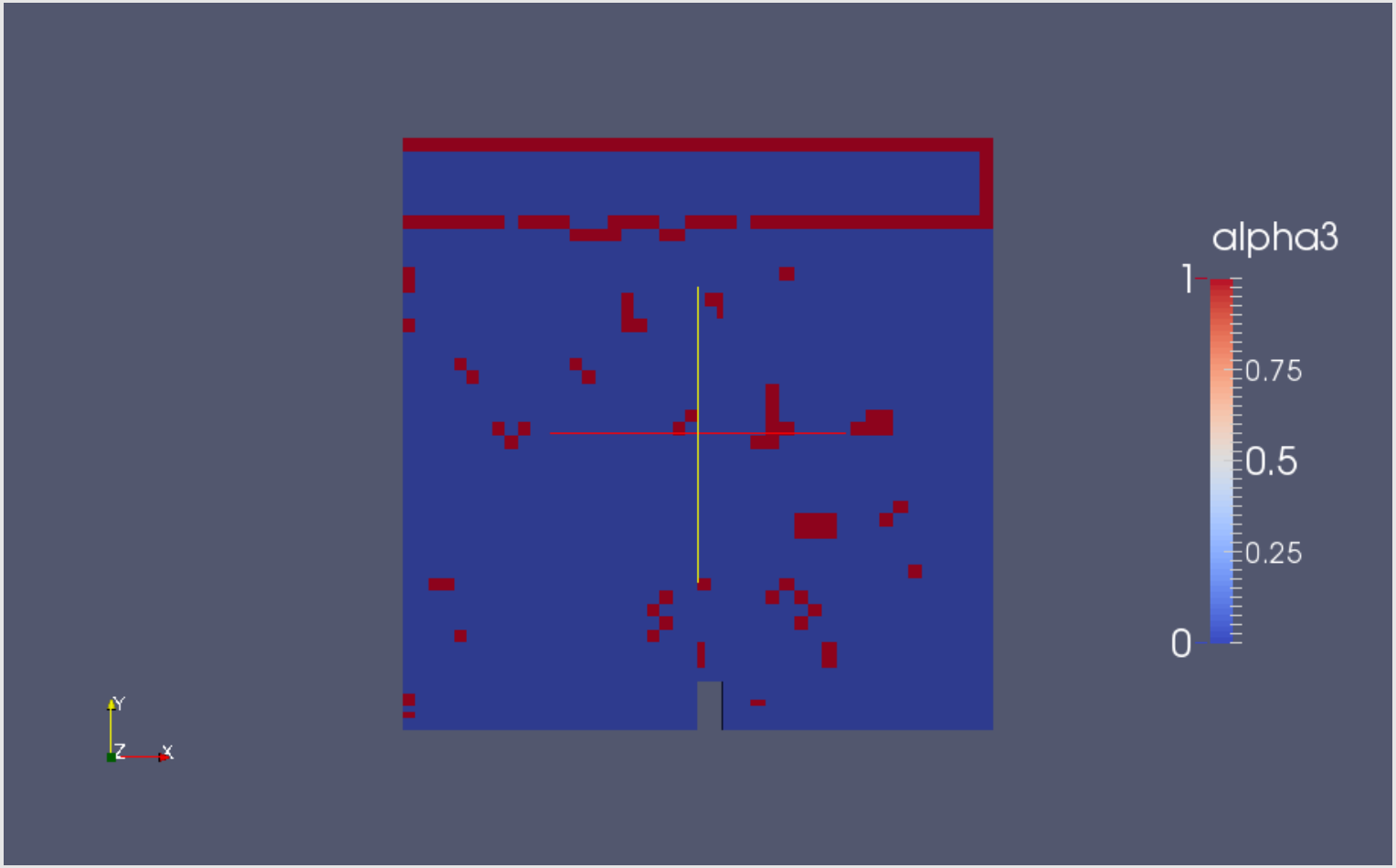
Strömungsforschung GmbH
ICE
Innovative Computational Engineering

Seems like this is a stable configuration



Strömungsforschung GmbH
ICE
Innovative Computational Engineering

Interesting but still boring



Strömungsforschung GmbH
ICE
Innovative Computational Engineering

Outline

1 Introduction
About this presentation
What we're working with
Before we start
2 Programming-like structures
Stored variables
Global variables
More obscure variable types
"Programming" function objects
3 Python Integration
General
Approximate the pressure drop
4 Plugin-functions
Why plugin-functions
The Game of Life plugin
Testing the plugin
Additional functions
Developing your own functions
5 Conclusion

Additional functions

- To test the evolution of large numbers of iterations can be tedious
  - And also memory-consuming if `gameOfLife_step` calls itself 10 times
- One project:
  - Write a function that calls that allows specifying "evolve the system N times"
- The other project:
  - Because we didn't implement the neighbors properly the result is boring. We want a function that allows us to specify a "rules-string"

Arguments not parsed by swak

- The additional arguments for the new functions have one thing in common:
  - They are not swak-expression
    - Integer for one of them
    - A string for the other
- In such cases in the variable specification we specify a `primitive` parser
  - Type `label` for the integer
  - Type `string` for the string
- Additional primitive types available are
  - `word`:string without quotes)
  - `scalar`: floating point value
- We have to provide special `setArgument` methods for these

Inheriting from the original function

- Both functions should inherit from the original `gameOfLifeFunction`
  - So that functionality only has to be specified once
- A new constructor has to be specified in `gameOfLifeFunction` so that the sub-classes my get their argument specifications to the `FieldValuePluginFunction`

gameOfLifeFunction.C

```
gameOfLifeFunction::gameOfLifeFunction(
    const FieldValueExpressionDriver &parentDriver,
    const word &name,
    const string &arguments
):
    FieldValuePluginFunction(
parentDriver,
name,
word("volScalarField"),
arguments
    )
{
    setConwayNumbers();
}
```

Constructor for the N-times function

- This class calls the new constructor
  - With an additional argument specified
- Needs an additional variable `nIterations_` to store N

gameOfLifeNFunction.C

```
gameOfLifeNFunction::gameOfLifeNFunction(
    const FieldValueExpressionDriver &parentDriver,
    const word &name
):
    gameOfLifeFunction(
parentDriver,
name,
string(
    "oldState internalField volScalarField,"
    "nr_of_iterations primitive label"
)
    ),
    nIterations_(-1)
{
}
```

Getting the scalar argument

- We need to provide a `setArgument` for integers
  - Sanity-checking is optional but recommended

gameOfLifeNFunction.C

```
void gameOfLifeNFunction::setArgument(
    label index,
    const label &value
) {
    if(index==1) {
nIterations_=value;
if(value<0) {
    FatalErrorIn("gameOfLifeNFunction::setArgument("
    "label␣index,"
    "const␣label␣&value)")
<< "Got␣iteration␣number␣" << value
    << ".␣Should␣be␣>=0"
    << endl
    << exit(FatalError);
}
    } else {
FatalErrorIn("gameOfLifeNFunction::setArgument("
"label␣index,"
"const␣label␣&value)")
    << "This␣should␣not␣happen.␣Expecting␣index==1.␣Got␣" << index
```

Calculating N steps

- Just loop

gameOfLifeNFunction.C

```
void gameOfLifeNFunction::doEvaluation()
{
    autoPtr<volScalarField> r(
new volScalarField(oldState())
    );

    for(label i=0;i<nIterations_;i++) {
r=doStep(r());
    }

    result().setObjectResult(r);
}
```

The rule string

- Convention for the class of finite automate that are similar to the original game of life is a string of the form `survive/birth` that specifies which cells live in the next step
  - `survive` and `birth` are sequences of digits
    - If the cell is alive and the number of neighbors is one of the digits in `survive` the cell survives
    - If the cell is dead and the neighbor-number is in `birth` then the cell lives
- The original Conway-game is specified by the string 23/3
- Our function should use such a string

Specification of the *Rules*-function

gameOfLifeRulesFunction.C

```
gameOfLifeRulesFunction::gameOfLifeRulesFunction(
    const FieldValueExpressionDriver &parentDriver,
    const word &name
):
    gameOfLifeFunction(
parentDriver,
name,
string(
    "oldState internalField volScalarField,"
    "surviveSlashBirth primitive string"
)
    )
{
}
```

Strömungsforschung GmbH
ICE
Innovative Computational Engineering

Notes on the further implementation of the rules function

- `setArguments` parses the rules-string
  - Sets the rule in the parent-class
  - Therefor no need to store the rule-string
- No special `doEvaluation` needed
  - The parent implementation works as well

After compiling

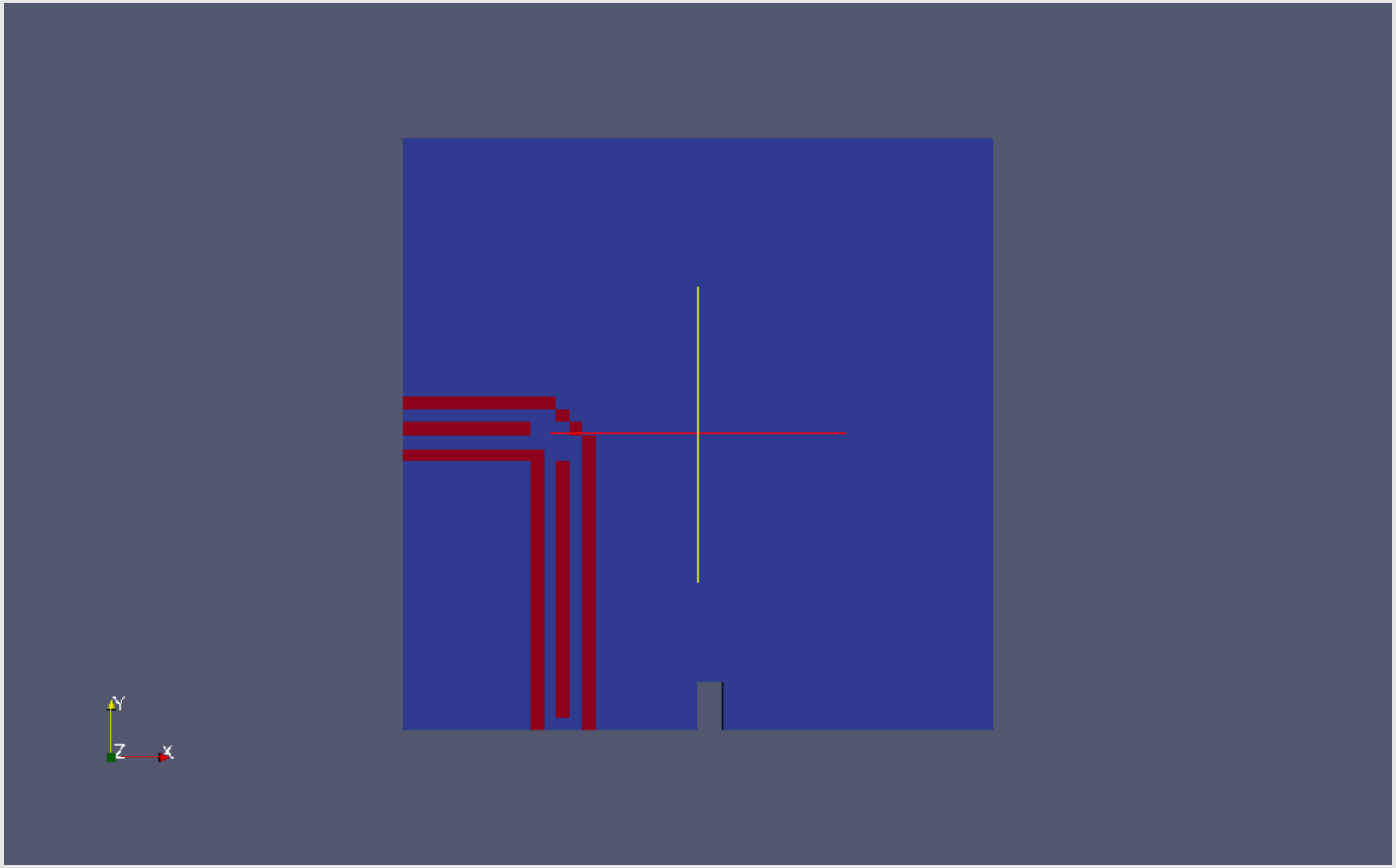
- Don't forget registering the functions with the run-time selection table in the source
  - Otherwise you won't see them
- If they are registered you see them in the "online help"

```
"Loaded␣plugin␣functions␣for␣'FieldValueExpressionDriver':"
  gameOfLife_Nstep:
    "volScalarField␣gameOfLife_Nstep(internalField/volScalarField␣oldState,<brk>
        <cont>primitive/label␣nr_of_iterations)"
  gameOfLife_rules_step:
    "volScalarField␣gameOfLife_rules_step(internalField/volScalarField␣<brk>
        <cont>oldState,primitive/string␣surviveSlashBirth)"
  gameOfLife_step:
    "volScalarField␣gameOfLife_step(internalField/volScalarField␣oldState)"
```

Exercises

- Play around with the rules sets and try to find more interesting combinations
  - The quotes for the rules-string have to be escaped with \
    - The old "string in string"-problem
- Try the functions on grids with different connectivies (tetrahedral for instance)
- Extend the implementation to include "edge neighbors"
- Write a function that executes a specified rule N times
  - Double inheritance is not a good idea

Can you find the rule-set? More than one step



Strömungsforschung GmbH
ICE
Innovative Computational Engineering

Outline

1 Introduction
- About this presentation
- What we're working with
- Before we start

2 Programming-like structures
- Stored variables
- Global variables
- More obscure variable types
- "Programming" function objects

3 Python Integration
- General
- Approximate the pressure drop

4 Plugin-functions
- Why plugin-functions
- The Game of Life plugin
- Testing the plugin
- Additional functions
- Developing your own functions

5 Conclusion

Do it

- It is not that hard:
  - 1 sub-class
  - 1 constructor
  - 2 methods
  - the actual functionality is up to you
- It allows you to use your function from all the other swak4Foam-stuff

Naming convention for functions

- Of course you may name the function however you like
- But:
  - Try to make sure that the function name does not clash with built-in functions
    - And others
  - One useful convention is to split the function name into 2 parts
    1. A short prefix that identifies the library. Ended with a _
    2. The actual function name
  - Don't make the function name too short: 3 weeks later it is the only documentation you have

The problem with outdated function-plugins

- One problem is that the swak-sources are updated and recompiled
  - But your functions are not
- This may lead to all kinds of weird behaviours and crashes
- Solution: the `SWAK_USER_PLUGINS`-variable
  - If the variable is set then it is used by the `Allmake`-script of swak
  - Content of the variable is the list of locations of sources for functions
    - Separated by ;
  - The `Allmake`-script compiles all these directories with `wmake libso`
- So when you recompile swak your functions are recompiled automagically
  - You have to set the variable somewhere

Outline

1. Introduction
   - About this presentation
   - What we're working with
   - Before we start
2. Programming-like structures
   - Stored variables
   - Global variables
   - More obscure variable types
   - "Programming" function objects
3. Python Integration
   - General
   - Approximate the pressure drop
4. Plugin-functions
   - Why plugin-functions
   - The Game of Life plugin
   - Testing the plugin
   - Additional functions
   - Developing your own functions
5. Conclusion

The most important thing we learned today

http://www.gocomics.com/pearlsbeforeswine

Strömungsforschung GmbH
ICE
Innovative Computational Engineering

What else we learned

- There are a lot of function objects in swak4foam
  - Some are quite useful
- Python-integration allows us to use Python-libraries in our OpenFOAM-runs
  - Python is fun
- Function plugins make it possible to integrate user-specific functions
  - Sub-class one class and add three methods and you're in business
    - The actual functionality depends on you

Acknowledgment

- All the things here build on the technical marvel that is `OpenFOAM / FOAM`
  - Honorable mentions (without these swak4Foam wouldn't be possible):
    - run-time selection
    - object registries
- So: three cheers to Henry and Hrv (and all others who worked on it)

The exercises

- By the time we've reached this slide the next trainer is probably knocking on my shoulder and asking me "politely" to leave
- Nevertheless you're encouraged to try the examples yourself
  - and do the exercises
- I'm willing to help you with the exercises in the next few weeks
  - To do so I created a Reddit
    `http://www.reddit.com/r/swakPyFoam/`
  - Will start a thread there with the name of this presentation
  - Post your questions there and brag about your solutions
    - Don't spam the message board. Others will be annoyed
    - Don't EMail. Others can't read it

Strömungsforschung GmbH
ICE
Innovative Computational Engineering

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