Setting up a test problem / application and using the build system (CMake)

Test Problems / Applications

- A test problem / application consists of:
  - a **main** file (*test_name.cc*)
  - a **problem** file (*test_name_problem.hh*)
  - a **spatialparameters** file (*test_name_spatialparams.hh*)
  - an **input** file (*test_name.input*)
  - a **build system** file (*CMakeLists.txt*)

The problem file and spatial parameters

- A DuMu^X problem (implemented in *problem.hh*)
  - implements a specific model scenario

```
template<class TypeTag>
class InjectionProblem2P : public PorousMediumFlowProblem<TypeTag>
{
    // Details of the model scenario (BCs, ICs, etc.)
}
```

The problem file and spatial parameters

- A DuMu^X problem (implemented in *problem.hh*)
  - implements a specific model scenario
  - configures a model for the scenario using properties and parameters

```
namespace Properties
{
    // define the TypeTag for this problem with a cell-centered two-point flux approximation
    // spatial discretization.
    NEW_TYPE_TAG(Injection2pTypeTag, INHERITS_FROM(TwoP));

    // Set the grid type
    SET_TYPE_PROP(Injection2pTypeTag, Grid, Dune::YaspGrid<2>);
    // Set the problem property
    SET_TYPE_PROP(Injection2pTypeTag, Problem, InjectionProblem2P<TypeTag>);
    // Set the spatial parameters
    SET_TYPE_PROP(Injection2pTypeTag, SpatialParams, InjectionSpatialParams<typename
    GET_PROP_TYPE(TypeTag, FVGridGeometry), typename GET_PROP_TYPE(TypeTag, Scalar)>);
    // Set fluid configuration
    SET_TYPE_PROP(Injection2pTypeTag, FluidSystem, FluidSystems::H2ON2<typename
    GET_PROP_TYPE(TypeTag, Scalar), FluidSystems::H2ON2DefaultPolicy<true>>);
}
```

- details tomorrow

```
namespace Properties
{
    // define the TypeTag for this problem with a cell-centered two-point flux approximation
    // spatial discretization.
    NEW_TYPE_TAG(Injection2pTypeTag, INHERITS_FROM(TwoP));
    NEW_TYPE_TAG(Injection2pCCTypeTag, INHERITS_FROM(CCTpfaModel, Injection2pTypeTag));
    // Set the grid type
    SET_TYPE_PROP(Injection2pTypeTag, Grid, Dune::YaspGrid<2>);
    // Set the problem property
    SET_TYPE_PROP(Injection2pTypeTag, Problem, InjectionProblem2P<TypeTag>);
    // Set the spatial parameters
    SET_TYPE_PROP(Injection2pTypeTag, SpatialParams, InjectionSpatialParams<typename
    GET_PROP_TYPE(TypeTag, FVGridGeometry), typename GET_PROP_TYPE(TypeTag, Scalar)>);
    // Set fluid configuration
    SET_TYPE_PROP(Injection2pTypeTag, FluidSystem, FluidSystems::H2ON2<typename
    GET_PROP_TYPE(TypeTag, Scalar), FluidSystems::H2ON2DefaultPolicy<true>>);
}
```

- details tomorrow

The problem file and spatial parameters

- A DuMu^X problem (implemented in *problem.hh*)
  - implements a specific model scenario
  - configures a model for the scenario using properties and parameters
  - defines boundary conditions

```
 /*!
  * \brief Specifies which kind of boundary condition should be
  *        used for which equation on a given boundary segment.
  *
  * \param globalPos The position for which the bc type should be evaluated
  */
BoundaryTypes boundaryTypesAtPos(const GlobalPosition &globalPos) const
{
    BoundaryTypes bcTypes;
    // set the left of the domain (with the global position in "0 = x" direction as a
    // Dirichlet boundary
    if (globalPos[0] < eps_)
        bcTypes.setAllDirichlet();
    // set all other as Neumann boundaries
    else
        bcTypes.setAllNeumann();

    return bcTypes;
}
```

```
 /*!
  * \brief Evaluates the boundary conditions for a Dirichlet
  *        boundary segment
  *
  * \param globalPos The global position
  */
PrimaryVariables dirichletAtPos(const GlobalPosition &globalPos) const
{
   return initialAtPos(globalPos);
}
```

```
PrimaryVariables neumannAtPos(const GlobalPosition &globalPos) const
{
    // initialize values to zero, i.e. no-flow Neumann boundary conditions
    PrimaryVariables values(0.0);

    // if we are inside the injection zone set inflow Neumann boundary conditions
    // using < boundary + eps_ or > boundary - eps_ is safer for floating point comparisons
    // than using <= or >= as it is robust with regard to imprecision introduced by
    // rounding errors.
    if (time_ < injectionDuration_ && globalPos[1] < 15 + eps_ && globalPos[1] > 7 - eps_
        && globalPos[0] > 0.9*this->fvGridGeometry().bBoxMax()[0])
    {
        // inject nitrogen. negative values mean injection
        // units kg/(s*m^2)
        values[Indices::conti0EqIdx + FluidSystem::N2Idx] = -1e-4;
        values[Indices::conti0EqIdx + FluidSystem::H2OIdx] = 0.0;
    }
    return values;
}
```

The problem file and spatial parameters

- A DuMu^X problem (implemented in *problem.hh*)
  - implements a specific model scenario
  - configures a model for the scenario using properties and parameters
  - defines boundary conditions
  - defines initial conditions

```
PrimaryVariables initialAtPos(const GlobalPosition &globalPos) const
{
    PrimaryVariables values(0.0);

    // get the water density at atmospheric conditions
    const Scalar densityW = FluidSystem::H2O::liquidDensity(temperature(), 1.0e5);

    // assume an initially hydrostatic liquid pressure profile
    // note: we subtract rho_w*g*h because g is defined negative
    const Scalar pw = 1.0e5 - densityW*this->gravity()[dimWorld-1]*(aquiferDepth_ -
                      globalPos[dimWorld-1]);

    values[Indices::pressureIdx] = pw;
    values[Indices::saturationIdx] = 0.0;

    return values;
}
```

The problem file and spatial parameters

- A DuMu^X problem (implemented in *problem.hh*)
  - implements a specific model scenario
  - configures a model for the scenario using properties and parameters
  - defines boundary conditions
  - defines initial conditions
  - defines source/sink terms

```
/*!
 * \brief Evaluate the source term for all phases within a given
 *        sub-control-volume.
 *
 * \param globalPos The position for which the source term should be evaluated
*/
NumEqVector sourceAtPos(const GlobalPosition &globalPos) const
{
    // The units must be according to either using mole or mass fractions.
    // (mole/(m^3*s) or kg/(m^3*s))
    return NumEqVector(0.0);
}
```

The problem file and spatial parameters

- A DuMu^X problem (implemented in *problem.hh*)
  - implements a specific model scenario
  - configures a model for the scenario using properties and parameters
  - defines boundary conditions
  - defines initial conditions
  - defines source/sink terms

- The spatial parameters (implemented in *spatialparams.hh*)
  - define spatial parameters of the porous material (permeability, porosity, material laws)

```
    PermeabilityType permeabilityAtPos(const GlobalPosition& globalPos) const
    {
        // here, either aquitard or aquifer permeability are returned, depending on the global
        // position
        if (isInAquitard_(globalPos))
                return aquitardK_;
        return aquiferK_;
    }

private:
    // provides a convenient way distinguishing whether a given location is inside the aquitard
    bool isInAquitard_(const GlobalPosition &globalPos) const
    {
        // globalPos[dimWorld-1] is the y direction for 2D grids or the z direction for 3D
        // grids
        return globalPos[dimWorld-1] > aquiferHeightFromBottom_ + eps_;
    }
```

The input file *(*.input)*

- DUNE INI syntax:

```
[Problem]
MyInteger = 2
```

- Input files are specified as arguments to the executable

```
./myexecutable inputfile.input
```

- If no input file is given it looks for my`executable.input`

- Parameters can be overwritten through the command line like this

```
./executable –Problem.MyInteger 2
```

The input file *(*.input)*

- A list of parameters which can be defined in the input file and the corresponding groups these parameters belong to can be found here:
  - http://www.dumux.org/doxygen-stable/html-2.12/a06485.php

- More details on parameters in next lesson

The main source file *(*.cc)*

- Each problem has a specific main file (*test_name.cc*) which sets up the program structure and calls assembler and solvers to assemble and solve the PDEs.
- Depending on the complexity of the problem the main file can be either set up to solve a linear problem, a non-linear problem and stationary as well as instationary problems
- The main file always includes the problem, the solvers, the assembler, the vtkoutputmodule and the gridmanager.

```
#include <config.h>
#include "1pproblem.hh
#include <dumux/nonlinear/newtonsolver.hh>
#include <dumux/linear/seqsolverbackend.hh>
#include <dumux/assembly/fvassembler.hh>
#include <dumux/io/vtkoutputmodule.hh>
#include <dumux/io/grid/gridmanager.hh>
```

The main source file *(*.cc)*

Common structure for all main files:

```
// define the type tag for this problem
using TypeTag = TTAG(TYPETAG);

//////////////////////////////////////////////////////////
// initialize MPI, finalize is done automatically on exit
const auto& mpiHelper = Dune::MPIHelper::instance(argc, argv);

// print dumux start message
if (mpiHelper.rank() == 0)
    DumuxMessage::print(/*firstCall=*/true);

// initialize parameter tree
Parameters::init(argc, argv);

////////////////////////////////////////////////////////////////////
// try to create a grid (from the given grid file or the input file)
////////////////////////////////////////////////////////////////////

GridManager<typename GET_PROP_TYPE(TypeTag, Grid)> gridManager;
gridManager.init();

// we compute on the leaf grid view
const auto& leafGridView = gridManager.grid().leafGridView();
```

The main source file *(*.cc)*

Common structure for all main files (stationary):

```
// create the finite volume grid geometry
using FVGridGeometry = typename GET_PROP_TYPE(TypeTag, FVGridGeometry);
auto fvGridGeometry = std::make_shared<FVGridGeometry>(leafGridView);
fvGridGeometry->update();

// the problem (initial and boundary conditions)
using Problem = typename GET_PROP_TYPE(TypeTag, Problem);
auto problem = std::make_shared<Problem>(fvGridGeometry);

// the solution vector
using SolutionVector = typename GET_PROP_TYPE(TypeTag, SolutionVector);
SolutionVector x(fvGridGeometry->numDofs());



// the grid variables
using GridVariables = typename GET_PROP_TYPE(TypeTag, GridVariables);
auto gridVariables = std::make_shared<GridVariables>(problem, fvGridGeometry);
gridVariables->init(x);

// intialize the vtk output module
using VtkOutputFields = typename GET_PROP_TYPE(TypeTag, VtkOutputFields);
VtkOutputModule<TypeTag> vtkWriter(*problem, *fvGridGeometry, *gridVariables, x,
                                   problem->name());
VtkOutputFields::init(vtkWriter); //!< Add model specific output fields
```

The main source file *(*.cc)*

Common structure for all main files (instationary):

```
// create the finite volume grid geometry
using FVGridGeometry = typename GET_PROP_TYPE(TypeTag, FVGridGeometry);
auto fvGridGeometry = std::make_shared<FVGridGeometry>(leafGridView);
fvGridGeometry->update();

// the problem (initial and boundary conditions)
using Problem = typename GET_PROP_TYPE(TypeTag, Problem);
auto problem = std::make_shared<Problem>(fvGridGeometry);

// the solution vector
using SolutionVector = typename GET_PROP_TYPE(TypeTag, SolutionVector);
SolutionVector x(fvGridGeometry->numDofs());
problem->applyInitialSolution(x);
auto xOld = x;

// the grid variables
using GridVariables = typename GET_PROP_TYPE(TypeTag, GridVariables);
auto gridVariables = std::make_shared<GridVariables>(problem, fvGridGeometry);
gridVariables->init(x,xOld);

// intialize the vtk output module
using VtkOutputFields = typename GET_PROP_TYPE(TypeTag, VtkOutputFields);
VtkOutputModule<TypeTag> vtkWriter(*problem, *fvGridGeometry, *gridVariables, x,
                                   problem->name());
VtkOutputFields::init(vtkWriter); //!< Add model specific output fields
```

The main source file *(*.cc)*

Specialization for different solving strategies

- a **stationary linear** problem:

```
using Assembler = FVAssembler<TypeTag, DiffMethod::numeric>;
auto assembler = std::make_shared<Assembler>(problem, fvGridGeometry, gridVariables);

// the linear solver
using LinearSolver = ILUOBiCGSTABBackend;
auto linearSolver = std::make_shared<LinearSolver>();

// the discretization matrices for stationary linear problems
using JacobianMatrix = typename GET_PROP_TYPE(TypeTag, JacobianMatrix);
auto A = std::make_shared<JacobianMatrix>();
auto r = std::make_shared<SolutionVector>();
assembler->setLinearSystem(A, r);
assembler->assembleJacobianAndResidual(x);

// we solve Ax = -r to save update and copy
(*r) *= -1.0;
linearSolver->solve(*A, x, *r);

// the grid variables need to be up to date for subsequent output
gridVariables->update(x);
```

The main source file *(*.cc)*

Specialization for different solving strategies

- a **stationary non-linear** problem:

```
using Assembler = FVAssembler<TypeTag, DiffMethod::numeric>;
auto assembler = std::make_shared<Assembler>(problem, fvGridGeometry, gridVariables);

// the linear solver
using LinearSolver = ILUOBiCGSTABBackend;
auto linearSolver = std::make_shared<LinearSolver>();

// the non-linear solver
using NewtonSolver = Dumux::NewtonSolver<Assembler, LinearSolver>;
NewtonSolver nonLinearSolver(assembler, linearSolver);

// linearize & solve
nonLinearSolver.solve(x);
```

The main source file *(*.cc)*

Specialization for different solving strategies

- a **instationary non-linear** problem:

```
// instantiate time loop
auto timeLoop = std::make_shared<CheckPointTimeLoop<Scalar>>(0.0, dt, tEnd);
timeLoop->setMaxTimeStepSize(maxDt);

using Assembler = FVAssembler<TypeTag, DiffMethod::numeric>;
auto assembler = std::make_shared<Assembler>(problem, fvGridGeometry, gridVariables, timeLoop);

// the linear solver
using LinearSolver = ILUOBiCGSTABBackend;
auto linearSolver = std::make_shared<LinearSolver>();

// the non-linear solver
using NewtonSolver = Dumux::NewtonSolver<Assembler, LinearSolver>;
NewtonSolver nonLinearSolver(assembler, linearSolver);

// time loop
timeLoop->start(); do
{
    // Calculate solution within each time step
} while (!timeLoop->finished());
```

The main source file *(*.cc)*

Specialization for different solving strategies

```
    // time loop
    timeLoop->start(); do
    {
        // set previous solution for storage evaluations
        assembler->setPreviousSolution(xOld);

        // linearize & solve
        nonLinearSolver.solve(x, *timeLoop);

        // make the new solution the old solution
        xOld = x;
        gridVariables->advanceTimeStep();

        // advance to the time loop to the next step
        timeLoop->advanceTimeStep();

        // report statistics of this time step
        timeLoop->reportTimeStep();

        // set new dt as suggested by the newton solver
        timeLoop->setTimeStepSize(nonLinearSolver.suggestTimeStepSize(timeLoop->timeStepSize()));

    } while (!timeLoop->finished());

    timeLoop->finalize(leafGridView.comm());
```

Build system (CMake)

What is CMake?

- open source build system tool developed by KITware.
- offers a one-tool-solution to all building tasks, like configuring, building, linking, testing and packaging.
- is a build system generator: It supports a set of backends called *generators*
- is portable and supports cross-platform compilation
- is controlled by ONE rather simple language
- Every directory in a project contains a file called **CMakeLists.txt**, which is written in the CMake language. You can think of these as a distributed configure script. Upon configure, the top-level CMakeLists.txt is executed

Build system (CMake)

Configuring

- Configure build time compiler parameters / linking information
- Create „targets“ that can be build to create executables

Build system (CMake)

Configuring

- with the script “dune-common/bin/dunecontrol <options>”
- takes care of all dependencies and modular dune structure
- option **all:** build all libraries and executables
- option **--opts=<optionfile.opts>** specify e.g. compiler flags
- option **--build-dir=<build directory>** specify path for out-of-source build
  - default: every module contains its own build directory **build-cmake/**
- You have to reconfigure (possibly deleting all build directories first) whenever a dependency changes or a Dune library is updated

```
./dune-common/bin/dunecontrol --opts=install.opts all
```

Build system (CMakeLists.txt)

Important basic commands:

- `add_subdirectory` for recursively adding subdirectories
  - the subdirectory has to contain a CMakeLists.txt file (can be empty)
- `add_executable(<name> source1 [source2 …])`
- `dune_add_test(…)` add a test executable to the test suite
- `dune_symlink_to_source_files(FILES file1 [file2 …])`

```
dune_add_test(NAME test_2p_incompressible_box
              SOURCES test_2p_fv.cc
              CMD_ARGS test_2p.input)
```

```
dune_add_test(NAME test_2p_incompressible_box
              SOURCES test_2p_fv.cc
              CMD_ARGS test_2p.input)
```

```
dune_add_test(NAME test_2p_incompressible_box
              SOURCES test_2p_fv.cc
              COMPILE_DEFINITIONS TYPETAG=TwoPIncompressibleBox
              COMMAND ${CMAKE_SOURCE_DIR}/bin/testing/runtest.py
              CMD_ARGS --script fuzzy
                       --files ${CMAKE_SOURCE_DIR}/test/references/lensbox-reference.vtu
                               ${CMAKE_CURRENT_BINARY_DIR}/2p_box-00007.vtu
                       --command "${CMAKE_CURRENT_BINARY_DIR}/test_2p_incompressible_box
                                  test_2p.input -Problem.Name 2p_box")
```

Build system (CMakeLists.txt)

Important basic commands:

- See also Dune build system documentation on
  - https://www.dune-project.org/sphinx/core/
- Comprehensive CMake online documentation

Exercises:

- Exercise about setting boundary conditions, the problem file etc:

Go to https://git.iws.uni-stuttgart.de/dumux-repositories/dumux-course/tree/master/exercises/exercise-basic and check out the README

- Exercise for the main-files:

Go to https://git.iws.uni-stuttgart.de/dumux-repositories/dumux-course/tree/master/exercises/exercise-mainfile and check out the README