# Setting up a test problem / application

# Test Problems / Applications

- A test problem / application consists of:
  - a **main** file (*test_name.cc* or *main.cc*)
  - a **problem** file (*test_name_problem.hh* or *problem.hh*)
  - a **spatialparameters** file (*test_name_spatialparams.hh* or *spatiailparams.hh*)
  - an **input** file (*params.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.
// Create new type tags
namespace TTag {
struct Injection2p { using InheritsFrom = std::tuple<TwoP>; };

} // end namespace Ttag




}
```

```
namespace Properties
{
// define the TypeTag for this problem with a cell-centered two-point flux approximation
spatial discretization.
// Create new type tags
namespace TTag {
struct Injection2p { using InheritsFrom = std::tuple<TwoP>; };
struct Injection2pCC { using InheritsFrom = std::tuple<Injection2p, CCTpfaModel>; };
} // end namespace Ttag




}
```

```
namespace Properties
{
// define the TypeTag for this problem with a cell-centered two-point flux approximation
spatial discretization.
// Create new type tags
namespace TTag {
struct Injection2p { using InheritsFrom = std::tuple<TwoP>; };
struct Injection2pCC { using InheritsFrom = std::tuple<Injection2p, CCTpfaModel>; };
} // end namespace Ttag

// Set the grid type
template<class TypeTag>
struct Grid<TypeTag, TTag::Injection2p> { using type = Dune::YaspGrid<2>; };

// Set the problem property
template<class TypeTag>
struct Problem<TypeTag, TTag::Injection2p> { using type = InjectionProblem2P<TypeTag>; };
}
```

```
namespace Properties
{

// Set the spatial parameters
template<class TypeTag>
struct SpatialParams<TypeTag, TTag::Injection2p> {
private:
    using FVGridGeometry = GetPropType<TypeTag, Properties::FVGridGeometry>;
    using Scalar = GetPropType<TypeTag, Properties::Scalar>;
public:
    using type = InjectionSpatialParams<FVGridGeometry, Scalar>; };

// Set fluid configuration
template<class TypeTag>
struct FluidSystem<TypeTag, TTag::Injection2p> { using type = FluidSystems::H2ON2<GetPropType
<TypeTag, Properties::Scalar>, FluidSystems::H2ON2DefaultPolicy</*fastButSimplifiedRelations=*/
true>>; };
}
```

# 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` or `params.input`

- Parameters can be overwritten through the command line like this

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

# The main source file *(*.cc)*

- Each problem has a specific main file (*test_name.cc* or *main.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>
```

# 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

## The main source file *(*.cc)*

Common structure for all main files:

```
// define the type tag for this problem
using TypeTag = Properties::TTag::OnePIncompressible;

//////////////////////////////////////////////////////////
// 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<GetPropType<TypeTag, Properties::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 = GetPropType<TypeTag, Properties::FVGridGeometry>;
auto fvGridGeometry = std::make_shared<FVGridGeometry>(leafGridView);
fvGridGeometry->update();

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

// the solution vector
using SolutionVector = GetPropType<TypeTag, Properties::SolutionVector>;
SolutionVector x(fvGridGeometry->numDofs());



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

// intialize the vtk output module
VtkOutputModule<GridVariables, SolutionVector> vtkWriter(*gridVariables, x, problem->name());
using VelocityOutput = GetPropType<TypeTag, Properties::VelocityOutput>;
vtkWriter.addVelocityOutput(std::make_shared<VelocityOutput>(*gridVariables));
using IOFields = GetPropType<TypeTag, Properties::IOFields>;
IOFields::initOutputModule(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 = GetPropType<TypeTag, Properties::FVGridGeometry>;
auto fvGridGeometry = std::make_shared<FVGridGeometry>(leafGridView);
fvGridGeometry->update();

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

// the solution vector
using SolutionVector = GetPropType<TypeTag, Properties::SolutionVector>;
SolutionVector x(fvGridGeometry->numDofs());
problem->applyInitialSolution(x);
auto xOld = x;

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

// intialize the vtk output module
VtkOutputModule<GridVariables, SolutionVector> vtkWriter(*gridVariables, x, problem->name());
using VelocityOutput = GetPropType<TypeTag, Properties::VelocityOutput>;
vtkWriter.addVelocityOutput(std::make_shared<VelocityOutput>(*gridVariables));
using IOFields = GetPropType<TypeTag, Properties::IOFields>;
IOFields::initOutputModule(vtkWriter); //!< Add model specific output fields
```

# The main source file *(*.cc)*

Specialization for different solving strategies

- a **stationary linear** problem:

```
// the assembler for stationary problems
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 = GetPropType<TypeTag, Properties::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());
```

# Exercises:

- 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