The eWoms Module: A Primer

Andreas Lauser

October 19, 2017

Goals of this Talk

Intended subject of this talk:

- High-level overview of the simulator part of the OPM code base for C++ developers
- Focus on the core of the numerical framework, i.e., the eWoms module

This talk is **not**:

- An introduction to programming, C++, the DUNE framework, etc.
  - Some familiarity assumed
- A guide for implementing $YOUR_FAVOURITE_FEATURE
  - Commercial support available
- A tutorial
- A detailed discussion of the technicalities

Part I

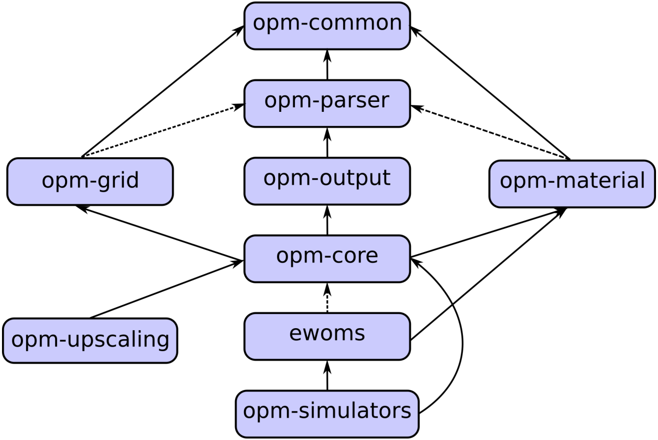
The Zoom-in

Overview

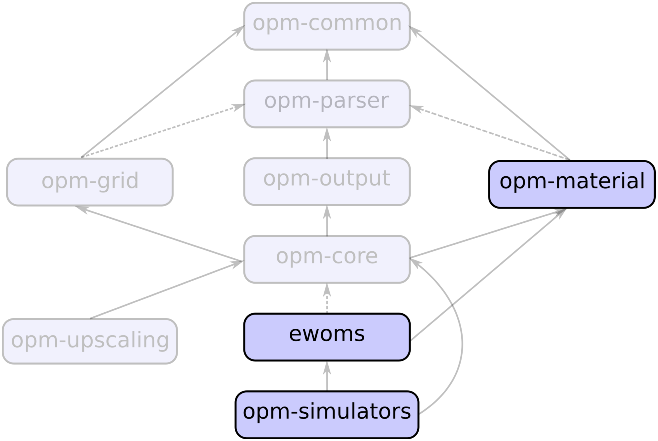
1. OPM: The View From Space
2. eWoms: A Birdseye View
3. Close Up: The Immiscible Model
4. Microscope: The Energy Extension

Overview

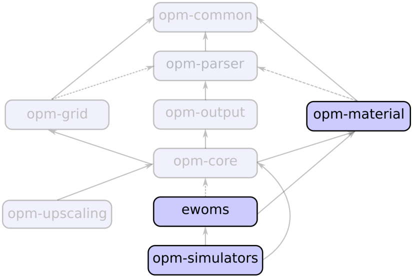
OPM: The View From Space



OPM: Relevant Modules



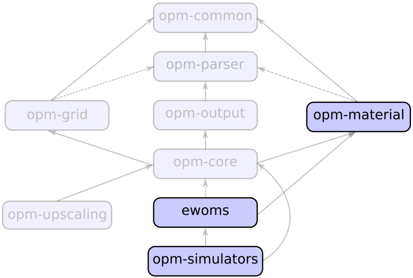
The opm-material Module



opm-material implements thermodynamic multi-phase relations as well as constitutive relations, e.g.:

- Thermodynamic representations (`FluidState`s)
- Thermodynamic properties (`FluidSystem`s)
- Capillary-pressure & relative permability relations ("material laws")
- Solvers for non-linear thermodynamic systems of equations (constraint solvers, e.g. flash)

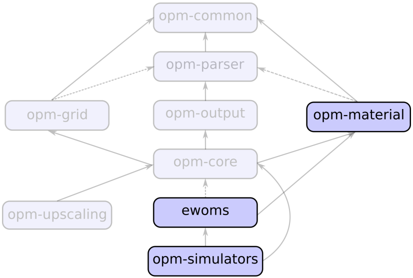
The eWoms Module



eWoms provides a versatile, extensible and performant numerical framework:

- Models for conservation equations
- Spatial and temporal discretization schemes
- Linear and non-linear solvers

The opm-simulators Module



opm-simulators features end-user ready simulator programs:

- In particular, the *flow* simulator for ECL decks
- (This module is currently undergoing a transition, many things do actually belong someplace else)

View form Space: Take-Home Message

Most of the numerics of flow is implemented by the framework layer, i.e., the eWoms module!

Overview

eWoms: A Birdseye View

High level control flow:

Time loop	`Simulator::run()`
Non-linear solve	`NewtonMethod::apply()`
Linearize	`FvBaseLinearizer::linearize()`
Compute local t.d. state	`$MODELIntensiveQuantities::update()`
Calculate local residual	`$MODELLocalResidual::computeStorage()`
	`$MODELLocalResidual::computeFlux()`
	`$MODELLocalResidual::computeSource()`
Linear solve	`ParallelBiCGStabSolverBackend::solve()`
Update	`NewtonMethod::update()`

Important Concepts

Simulator	High-level control of program execution, central “Nexus” for all information
Model	Specifies the conservation equations, primary variables, etc. Also, spatial and temporal discretization
Problem	Specifies the physical set-up

Central User Facing Class: The Problem

Specifies all externally “impressed” parameters:

- Initial solution
- Boundary conditions
- Porosity
- Intrinsic permeabilities
- Material-law parameters
- . . .

Problems are concerned with specifying the **physical set-up** mostly independently of the selected model.

Central User Facing Class: The Problem

Time loop	`Simulator::run()`
Non-linear solve	`NewtonMethod::apply()`
Linearize	`FvBaseLinearizer::linearize()`
Compute local t.d. state	`$MODELIntensiveQuantities::update()`

Calculate local residual	`$MODELLocalResidual::computeStorage()`
	`$MODELLocalResidual::computeFlux()`
	`$MODELLocalResidual::computeSource()`

Linear solve	`ParallelBiCGStabSolverBackend::solve()`
Update	`NewtonMethod::update()`

Wait a second: There's no problem here!

Central User Facing Class: The Problem

Time loop	`Simulator::run()`
Non-linear solve	`NewtonMethod::apply()`
Linearize	`FvBaseLinearizer::linearize()`
Compute local t.d. state	`$MODELIntensiveQuantities::update()`

Calculate local residual	`$MODELLocalResidual::computeStorage()`
	`$MODELLocalResidual::computeFlux()`
	`$MODELLocalResidual::computeSource()`

Linear solve	`ParallelBiCGStabSolverBackend::solve()`
Update	`NewtonMethod::update()`

Here it enters the picture!

Overview

1. OPM: The View From Space
2. eWoms: A Birdseye View
3. Close Up: The Immiscible Model
4. Microscope: The Energy Extension

Close Up: The Immiscible Model

Models specify conservation equations. The “immiscible” model deals with M fluid phases and ...

- ...assumes that fluid phases are completely immiscible
- ...conserves the mass in *kg* of each fluid phase
- ...selects the pressure of the first phase plus the saturations of the first $M - 1$ phases as primary variables

Intensive Quantities

Using the primary variables, compute everything else:

- Saturations of *all* fluid phases:

$$S_M = 1 - \sum_{\alpha=1}^{M-1} S_\alpha$$

- Pressures of *all* fluid phases using the reference phase' pressure and the capillary pressures:

$$p_\alpha = p_1 + p_{c,1\to\alpha}$$

- Phase compositions
  - Already specified by assuming immiscibility
- Other quantities needed for the residual, e.g., ρ_α, μ_α, $\mathbf{K}$
- Thermodynamic relations computed opm-material or quantities directly provided by the problem

The Local Residual

Based on the thermodynamic state, compute the residual for a degree of freedom:

- Storage: Mass in kg/m^3 for each phase at a given time t:

$$\sigma_{\alpha,t} = \phi_t S_{\alpha,t} \rho_{\alpha,t}$$

- Fluxes: Mass in $kg/(m^2 s)$ for a phase at a given time t:

$$\zeta_{\alpha,t} = -\rho_{\alpha,t} \frac{k_{r,\alpha,t}}{\mu_{\alpha,t}} \mathbf{K} \nabla (p_{\alpha,t} - g \rho_{\alpha,t})$$

- Source: Mass change in $kg/(m^3 s)$; just forward the problem's $q_{\alpha,t}$

The Local Residual

Generic code calculates the local residual for phase α:

$$r_\alpha = \frac{\sigma_{\alpha,t_2} - \sigma_{\alpha,t_1}}{t_2 - t_1} + \frac{1}{|\mathcal{V}|} \sum_{\partial\mathcal{V}} |\partial\mathcal{V}| \, \zeta_{\alpha,t2} - q_{\alpha,t_2}$$

(for the implicit Euler time- and a finite volume space discretization)

Overview

Microscope: The Energy Extension

- Models can be extended generically
- Extension mechanism is cooperative
- Idea: Derive all classes from extension classes
  - Provide real and dummy implementations with same API
- Use callbacks in the base model
- Compiler optimizes dummy callbacks away

EnergyIntensiveQuantities

```
template <class TypeTag, bool enableEnergy>
class EnergyIntensiveQuantities;

template <class TypeTag>
class EnergyIntensiveQuantities<TypeTag, true>
{ // ...
  void updateEnergy()
  { /* ... */ }
  const Evaluation& heatCapacitySolid() const
  { return heatCapSolid_; }
};

template <class TypeTag>
class EnergyIntensiveQuantities<TypeTag, false>
{ // ...
  void updateEnergy()
  { }
  const Evaluation& heatCapacitySolid() const
  { OPM_THROW(std::logic_error, "Energy is not conserved"); }
};
```

ImmiscibleIntensiveQuantities

```
template <class TypeTag>
class ImmiscibleIntensiveQuantities
: public EnergyIntensiveQuantities<TypeTag,
                                   GET_PROP_VALUE(TypeTag,
                                       EnableEnergy)>
{
  typedef EnergyIntensiveQuantities<TypeTag, GET_PROP_VALUE(
      TypeTag, EnableEnergy)> EnergyIQ;
  // ...
  void update() { // ...
    EnergyIQ::updateEnergy();
  }
};
```

Same for the other classes which need to be aware of the extension. (Local residual, extensive quantities, output writing classes, . . .)

Finally

The problem decides if energy is conserved by setting the `EnableEnergy` property

- If yes, it needs to provide some additional methods

```
SET_BOOL_PROP(Co2InjectionNiProblem, EnableEnergy, true);
```

Part II

Important Concepts

Overview

Overview

C++ Traits

Idea: Use specialization to give generic template code the chance to take different code paths based on its template arguments

Example:

```
template <class T>
struct is_float { static const bool value = false; }

template <>
struct is_float<float> { static const bool value = true; }

template <class T>
void f(const T& x)
{ std::cout << is_float<T>::value?"float: ":"non-float: "
            << x <<std::endl; }

int main()
{ f(float(1.0)); f(std::string("foo")); return 0; }
```

C++ Traits

- Observation: Class bodies are arbitrary
- Great! This can be (mis-)used to pass any number of compile time parameters using a single template parameter `T`!
- Not so great: We might want to inherit these properties
  - eWoms simulators define about 150 parameters
- It might be nice to know which traits have been defined where and what their values are

The eWoms Property System

- "C++ traits on steroids": Same basic idea as C++ traits, but with inheritance and introspection
- Duct tape which holds the eWoms models together
- Slightly different terminology than C++ traits:

C++ Traits	eWoms property system
trait name	property tag
`T` (specialized-for type)	type tag
trait class body	property

- Macros to hide the template kung-fu

The eWoms Property System: Example

```
namespace Ewoms { namespace Properties {
NEW_PROP_TAG(Foo);
NEW_PROP_TAG(Bar);

NEW_TYPE_TAG(BaseTypeTag);
SET_INT_PROP(BaseTypeTag, Foo, 0);
SET_INT_PROP(BaseTypeTag, Bar, 1);

NEW_TYPE_TAG(DerivedTypeTag, INHERITS_FROM(BaseTypeTag));
SET_INT_PROP(DerivedTypeTag, Foo, 2);
}}

int main() {
    Ewoms::Properties::printValues<TTAG(BaseTypeTag)>();
    Ewoms::Properties::printValues<TTAG(DerivedTypeTag)>();
    std::cout << GET_PROP_VALUE(TTAG(DerivedTypeTag), Foo)
              << std::endl;
    return 0;
}
```

Overview

The eWoms Parameter System

- eWoms properties (and C++ traits) must be set at compile time
- The eWoms parameter system deals which values which ought to be specified at runtime:
  - The type of parameters are specified at compile time
  - For each parameter, an eWoms property with exactly the same name must exist
    - The value of the property is used as default for the parameter
  - Parameters must be registered before their value can be retrieved
    - Guarantees the help message to be comprehensive
  - Same parameter can be registered multiple times
    - Description and type specification needs to be identical

The eWoms Parameter System: Example

In `lensproblem.hh`:

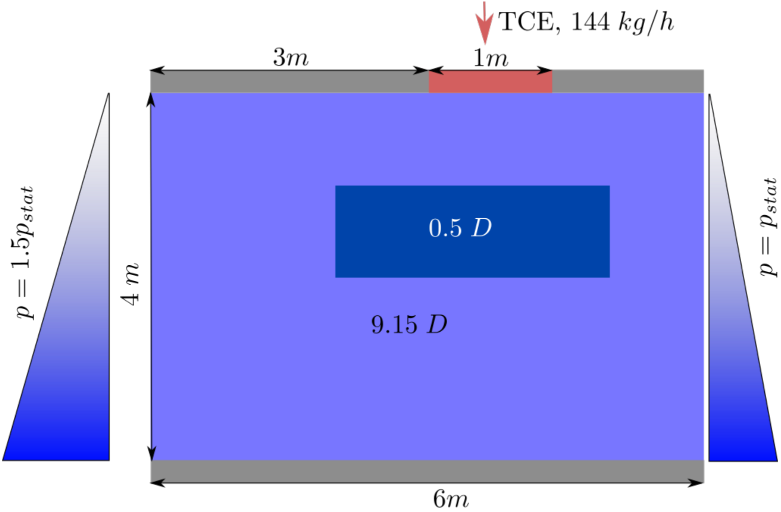
```
static void registerParameters()
{
  EWOMS_REGISTER_PARAM(TypeTag, Scalar, LensLowerLeftX,
                       "Lower-left x of the lens [m].");
};

void finishInit()
{ // ...
  lensLowerLeft_[0] =
    EWOMS_GET_PARAM(TypeTag, Scalar, LensLowerLeftX);
};
```

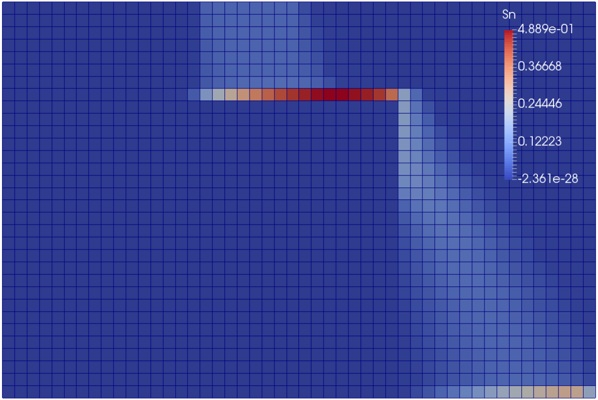
Overview

The Lens Problem

Simple yet relevant setup from ground remediation:

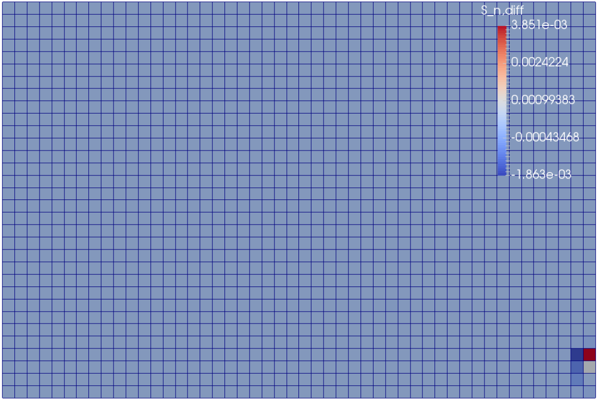


Results



Non-wetting phase saturation after 8 hrs, 20 mins

Comparison with DuMu^X



Difference of final non-wetting phase saturation between eWoms and DuMu^X

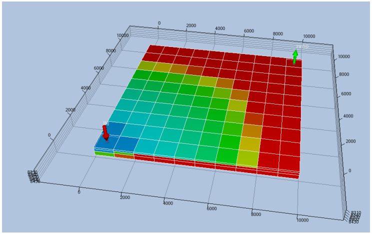
(~ 25 time steps)

Results of DuMu^X and eWoms basically identical!

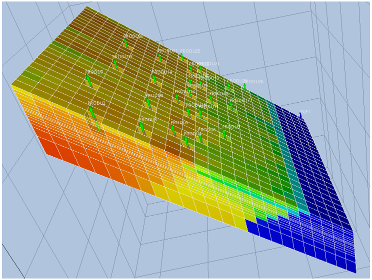
Overview

ebos

- The **E**cl **B**lack-**O**il **S**imulator
- Implemented as a standard eWoms problem
- The core of `flow` is a (relatively) thin wrapper around `ebos`
  - Initially a proof of concept for localized linearization with dense automatic differentiation
  - Well model, high-level control code and disk output code derived from flow_legacy



SPE-1



SPE-9

Overview

Summary

- eWoms/ebos constitute the core of the flow reservoir simulator
- eWoms is extremely flexible and can be quite performant
- Unfortunately, eWoms thus is also somewhat complex
- Many things are done differently than in other frameworks

Outlook

Some things are set to be improved or added in the medium term future:

- Documentation, in particular introductory guides
- Unification of ebos and flow
- Performance is quite good, but has not been a prime focus yet
- Python scripting

Outlook

Some things are set to be improved or added in the medium term future:

- Documentation, in particular introductory guides
- Unification of ebos and flow
- Performance is quite good, but has not been a prime focus yet
- Python scripting

Thank you for your attention.