The DuMu^X Material Law Framework

An excursion to thermodynamics
containing too much “multi-”

Andreas Lauser

Institute of Hydraulic Engineering, University of Stuttgart

October 7, 2010

Overview

Multiphase Thermodynamics

Implementation in DuMu^X

Overview

Multiphase Thermodynamics

Implementation in DuMu^X

Problem Description

In a porous medium context with M-fluid phases each comprising the same $N \geq M - 1$ components, calculate all quantities required by the physical model (i.e. the PDE) from a set of primary variables.

Fluid Properties Observed in the Wild

For each fluid phase α these quanties are

- Molar volume $V_{m\alpha}$ (Alternative: density ρ_α)
- Internal energy u_α (Alternative: enthalpy h_α)
- Fluid temperature T_α
- Fluid pressure p_α
- Fluid saturation S_α (Percentage of pore volume filled by α)
- Component mole fraction $x_{\alpha\kappa}$ for each component κ
  (Alternative: mass fractions or comp. concentrations)

Counting Unknowns

In our M-phase, N-component setting we need $\{V_{m\alpha}, p_\alpha, u_\alpha, T_\alpha, S_\alpha, x_{\alpha\kappa}\}$ where α has M posibilities and κ has N, so

- …we have $M \cdot (N+5)$ unknowns.
- …we also need $M \cdot (N+5)$ orthogonal conditions.

Common Sense Constraint

The following constraint can be accounted for as “common sense”:

- All pore space is used by some fluid, i.e. $\sum_\alpha S_\alpha = 1$

Thermodynamic Constraints

For each phase α the following constraints can be used:

- The pressure as an explicit function (“thermal equation of state”), i.e. $p_\alpha = p_\alpha(V_{m\alpha}, T_\alpha, x_{\alpha 1}, \ldots, x_{\alpha N})$
- The internal energy as an explicit function (“caloric equation of state”), i.e. $u_\alpha = u_\alpha(V_{m\alpha}, T_\alpha, x_{\alpha 1}, \ldots, x_{\alpha N})$

Also …

- …assume local thermodynamic equilibrium

Local Thermodynamic Equilibrium

"Local" means:

- Isolating any material point of the system at any given time yields a system at thermodynamic equilibrium

Thermodynamic equilibrium is composed of three sub-equilibria:

- Thermal Equilibrium: $\partial_t T_\alpha = 0 \; \forall \alpha$
- Mechanic Equilibrium: $\partial_t p_\alpha = 0 \; \forall \alpha$
- Chemical Equilibrium: $\partial_t x_{\alpha\kappa} = 0 \; \forall \alpha, \kappa$

Thermal Equilibrium

- Defined as $\partial_t T_\alpha = 0 \ \forall \alpha$
- Implies that there is only one well-defined temperature, i.e. $T_{\alpha_1} = T_{\alpha_2} = T$ for all α_1, α_2
- Suggests to use temperature T as primary variable

Mechanic Equilibrium

- Defined as $\partial_t p_\alpha = 0 \; \forall \alpha$
- Suggests to use a closed function $p_{c\alpha}$ which describes the pressure difference between phase α and phase 1 ("capillary pressure functions")
- $p_{c\alpha}$ depends on all saturations $S_{\langle \cdot \rangle}$, temperature and absolute pressure p_1 of an phase, as well as composition of all phases $x_{\langle \cdot, \cdot \rangle}$
- Usually $p_{c\alpha}$ is assumed to be just a function of the saturations

Chemical Equilibrium

- Defined as $\partial_t x_\alpha = 0 \; \forall \alpha$
- Implies that the fugacity (Alternative: chemical potential) of any component κ is the same in every phase, i.e. $\forall \kappa, \alpha_1, \alpha_2 : f_{\alpha_1 \kappa} = f_{\alpha_2 \kappa} = f_\kappa$
- $f_{\alpha\kappa}$ are explicit (albeit complicated) functions depending on $p_\alpha, x_{\alpha\kappa}, V_{m\alpha}$ and T
- Suggests using f_κ as a primary variable

Example: Fugacity Function

For a pure component using the Peng-Robinson thermal equation of state:

$$f = \frac{p}{Z}\,\frac{V_m}{V_m - b}\exp\{Z-1\}\left(\frac{V_m + b(1-\sqrt{2})}{V_m + b(1+\sqrt{2})}\right)^{\frac{1}{RT}\,\frac{a}{b\,2\sqrt{2}}}$$

where $Z = pV_m/RT$, a is “attractive factor” and b is “repulsive factor” for the component (are calculated using crititcal temperature and pressure and acentric factor of the component)

Thermodynamics Wrap-up

- $M \cdot (N+5)$ unknowns (For each phase $\{V_{m\alpha}, p_\alpha, u_\alpha, T_\alpha, S_\alpha, x_{\alpha\kappa}\}$)
- $M(N+4) - (N+1)$ thermodynamic constraints
  - $2M$ from the thermal and caloric equations of state
  - 1 for the saturation
  - $(M-1)$ from the thermal equilibrium
  - $(M-1)$ from the mechanic equilibrium
  - $N(M-1)$ for the chemical equilibrium
- The remaining $(M+N+1)$ constraints have to be defined by the physical model (PDEs plus auxiliary equations, i.e. primary variables)
- Result is a highly non-linear system of equations
- Multi-phase multi-component thermodynamics quickly becomes a messy affair

Overview

Multiphase Thermodynamics

Implementation in DuMu^X

Big **fat** disclaimer

This is still work in progress!

Design Patterns

For the API design, the it was tried to adhere to the following principles

Simple: No advanced C++ techniques like template meta programming are used

Stateless relations: Classes representing a collection of relations do not encompass any internal state (i.e. they only exhibit static methods)

Separation of parameters and independent variables: Function parameters are passed via an extra argument, the static member functions do not care how they are stored and calculated.

Pure Components

A component provides the **fluid parameters of a pure chemical substance** (like water, carbon dioxide, etc) **or pseudo substance**, i.e. a fixed mixture of substances like air, oil, etc.

The Class Head

```
template <class Scalar>
class MyComponent : public Component<Scalar, MyComponent<Scalar> >
{ public:

  // a human readable name of the component
  static const char *name();

  // The mass in [kg] of one mole of the component
  static Scalar molarMass();

  // ...
```

Critical and Triple Points

```
// ...

static Scalar criticalTemperature(); // [K]
static Scalar criticalPressure(); // [Pa]
static Scalar tripleTemperature(); // [K]
static Scalar triplePressure(); // [Pa]
static Scalar acentricFactor(); // []

// ...
```

Fluid Properties

```
  // ...

  static Scalar vaporPressure(Scalar T); // [Pa]

  static Scalar gasDensity(Scalar T, Scalar p); // [kg/m^3]
  static Scalar liquidDensity(Scalar T, Scalar p); // [kg/m^3]

  static Scalar gasViscosity(Scalar T, Scalar p); // [Pa s]
  static Scalar liquidViscosity(Scalar T, Scalar p); // [Pa s]

  static Scalar gasInternalEnergy(Scalar T, Scalar p); // [J/kg]
  static Scalar liquidInternalEnergy(Scalar T, Scalar p); // [J/kg]
};
```

Fluid States

Fluid states …

- …provide access to the thermodynamic quantities of all phases, i.e. $\{V_{m\alpha}, p_\alpha, u_\alpha, T_\alpha, S_\alpha, x_{\alpha\kappa}\}$

Fluid State: API

```
template <class Scalar>
class FluidState
{public:
  Scalar moleFrac(int phaseIdx, int compIdx);
  Scalar massFrac(int phaseIdx, int compIdx);
  Scalar concentration(int phaseIdx, int compIdx);
  Scalar density(int phaseIdx);
  Scalar temperature(int phaseIdx);
  Scalar pressure(int phaseIdx);
  // ...
};
```

Fluid Systems

Fluid systems …

- …provide an **interface to the fluid characteristics of mixtures**, e.g. viscosity, density, internal energies diffusion and fugacity coefficients
- …select the equations of state and mixing rules

```
template <class Scalar>
class MyFluidSystem {
 public:
  enum { numComponents = 2 };
  enum { numPhases = 2 } ;

  // indices for convenience
  enum { wPhaseIdx = 0 }; // wetting phase index
  enum { nPhaseIdx = 1 }; // non-wetting phase index

  static void init() {}

  static const char *componentName(int compIdx) {};
  static Scalar molarMass(int compIdx)

  // Given a phase's composition, temperature, pressure,
  // return its density [kg/m^3].
  template <class FluidState>
  static Scalar phaseDensity(int phaseIdx,
                             const FluidState &fluidState) {}

  // analogous for phaseViscosity and phaseInternalEnergy

  // ...
  // ...
  // Returns the fugacity coefficient for a component in a phase.
  template <class FluidState>
  static Scalar fugacityCoefficient(int phaseIdx,
                                    int compIdx,
                                    const FluidState &fluids);

  // the binary diffusion coefficient of two components
  // in a phase with a given composition
  template <class FluidState>
  static Scalar diffCoeff(int phaseIdx,
                          int compIIdx,
                          int compJIdx,
                          const FluidState &fluidState);
};
```

Fluid-Matrix Interactions

Fluid-Matrix interactions . . .

- . . . provide the capillary-pressure and relative permeability functions
- . . . only take saturations into account
- . . . are currently pretty stable for twophase flow
- . . . are build using the a plugable architecture

Brooks-Corey Capillary Pressure

```
template <class ScalarT, class ParamsT = BrooksCoreyParams<ScalarT> >
class BrooksCorey
{
public:
    typedef ParamsT Params;
    typedef typename Params::Scalar Scalar;

    static Scalar pC(const Params &params, Scalar Swe)
    { return params.pe()*pow(Swe, -1.0/params.alpha()); }

    static Scalar Sw(const Params &params, Scalar pC)
    { Scalar tmp = pow(pC/params.pe(), -params.alpha());
      return std::min(std::max(tmp, Scalar(0.0)), Scalar(1.0)); }

    static Scalar dpC_dSw(const Params &params, Scalar Swe)
    { return - params.pe()/params.alpha() * pow(Swe, -1/params.alpha()

  // ...
};
```

Regularized Brooks-Corey

```
template <class ScalarT, class ParamsT =
          RegularizedBrooksCoreyParams<ScalarT> >
class RegularizedBrooksCorey
{
    typedef Dumux::BrooksCorey<ScalarT, ParamsT> BrooksCorey;

public:
    typedef ParamsT Params;
    typedef typename Params::Scalar Scalar;

    static Scalar pC(const Params &params, Scalar Swe)
    {
        const Scalar Sthres = params.thresholdSw();
        if (Swe <= Sthres) {
            Scalar m = BrooksCorey::dpC_dSw(params, Sthres);
            Scalar pC_SweLow = BrooksCorey::pC(params, Sthres);
            return pC_SweLow + m*(Swe - Sthres);
        }
        else if (Swe > 1) {
            Scalar m = BrooksCorey::dpC_dSw(params, 1.0);
            Scalar pC_SweHigh = BrooksCorey::pC(params, 1.0);
            return pC_SweHigh + m*(Swe - 1.0);
        }
```

Residual Saturation Handling

```
template <class EffLawT, class AbsParamsT =
          EffToAbsLawParams<typename EffLawT::Params> >
class EffToAbsLaw
{
    typedef EffLawT EffLaw;

public:
    typedef AbsParamsT Params;
    typedef typename EffLaw::Scalar Scalar;

    static Scalar pC(const Params &params, Scalar Sw)
    { return EffLaw::pC(params, SwToSwe(params, Sw)); }

    static Scalar SwToSwe(const Params &params, Scalar Sw)
    { return (Sw - params.Swr())/(1 - params.Swr() - params.Snr()); }

    // ...
};
```

Bringing it Together

```
typedef RegularizedBrooksCorey<Scalar> EffectiveLaw;
typedef EffToAbsLaw<EffectiveLaw> MaterialLaw;
typedef MaterialLaw::Params MaterialLawParams;

// create parameters object
MaterialLawParams params;
params.setPe(1000); // B-C entry pressure
params.setAlpha(2); // B-C shape parameter
params.setSwr(0.04); // wetting phase residual saturation
params.setSnr(0.14); // non-wetting phase residual saturation

std::cout << MaterialLaw::pC(params, 0.123);
```

Conclusion

- Thermodynamics is very challenging in a multi-phase multi-component context
- Initial work has been done within the DuMu^X material framework
- The current solutions are far from perfect
- We are happy to share code and ideas

Thank you for your attention

References

R.C. Reid, J.M. Prausnitz, B.E. Poling:
The properties of liquids & gases
4th edition, McGraw-Hill, 1987.

A. Lauser, C. Hager, R. Helmig and B. Wohlmuth:
A new approach for phase transitions in miscible multi-phase flow in porous media.
Submitted to *Advances in Water Resources*