Eine Sammlung von DROPS aller Geschmacksrichtungen

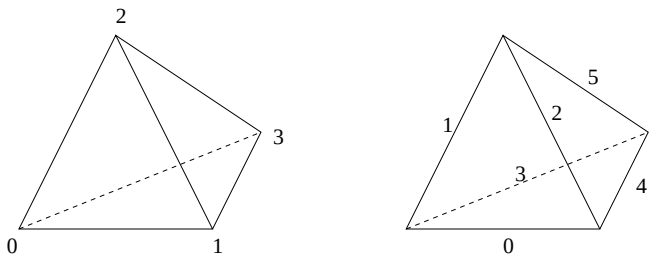
Inhaltsverzeichnis

1 geom: Geometrie

1.1 Tetraeder und Subsimplices

Datenstrukturen für die Subsimplices (Knoten, Kanten, Seitenflächen) und die Simplices (Tetraeder): `VertexCL`, `EdgeCL`, `FaceCL`, `TetraCL` (siehe `multigrid.h/cpp`).

Jeder Tetraeder hat Zugriff auf seine 4 Knoten, 6 Kanten und 4 Seitenflächen. Die interne Numerierung dieser Subsimplices auf einem Tetraeder kann der folgenden Skizze entnommen werden:



Die Seitenfläche i liegt dem Knoten i gegenüber. All diese topologischen Informationen sind in `topo.h/cpp` festgehalten.

1.2 Das MultiGrid

Das MultiGrid $\mathcal{M}$ ist durch die `MultiGridCL` repräsentiert. Darin sind die Tetraeder und Subsimplices nach Leveln geordnet abgespeichert. Das Level eines Objektes gibt gerade die Anzahl seiner Vorfahren an, entsprechend besitzt das gröbste Level die Nummer 0. Alle Tetraeder eines Levels k formen das sogenannte *Gitter der Stufe* k:

$$\mathcal{G}_k = \{T \in \mathcal{M} : \ell(T) = k\} \ .$$

Die *Triangulierung der Stufe* k wird vom Gitter $\mathcal{G}_k$ und allen gröberen Tetraedern $T \in \mathcal{G}_j, j < k$, die nicht verfeinert sind, gebildet:

$$\mathcal{T}_k := \mathcal{G}_k \cup \bigcup_{j<k} \{T \in \mathcal{G}_j : T \text{ ist unverfeinert}\} \ .$$

```
class MultiGridCL
{
    MultiGridCL (const MGBuilderCL& Builder);

    void Refine ();
```

```
    void Scale( double);
    void MakeConsistentNumbering();
    void SizeInfo(std::ostream&);
    void ElemInfo(std::ostream&, int Level= -1);

    bool IsSane (std::ostream&, int Level=-1) const;

    VertexIterator         GetVerticesBegin     (int Level);
    VertexIterator         GetVerticesEnd       (int Level);
    AllVertexIteratorCL    GetAllVertexBegin    (int Level=-1);
    AllVertexIteratorCL    GetAllVertexEnd      (int Level=-1);
    TriangVertexIteratorCL GetTriangVertexBegin (int Level=-1);
    TriangVertexIteratorCL GetTriangVertexEnd   (int Level=-1);
    // ... und dasselbe mit const_Iteratoren
    // ... und fuer Edge, Face, Tetra
};
```

Der Zugriff auf die Tetraeder und Subsimplices erfolgt über entsprechende Iteratoren, die sich drei Kategorien zuordnen. Am Beispiel der Tetraeder sind dies:

- `TetraIterator` iteriert über alle Tetraeder $T \in \mathcal{G}_k$ eines Gitters $\mathcal{G}_k$.
- `AllTetraIteratorCL` iteriert über alle Tetraeder $T \in \bigcup_{j \leq k} \mathcal{G}_j$ bis zum Gitter $\mathcal{G}_k$.
- `TriangTetraIteratorCL` iteriert über alle Tetraeder $T \in \mathcal{T}_k$ der Triangulierung $\mathcal{T}_k$.

Der Verfeinerungsalgorithmus wird durch die Elementfunktion `Refine` aufgerufen und verfeinert alle Tetraeder des MultiGrids $\mathcal{M}$, die zur Verfeinerung markiert sind (Ein Tetraeder wird durch Aufruf seiner Elementfunktion `TetraCL::SetRegRefMark` zur Verfeinerung markiert).

Ein MultiGrid wird mit Hilfe eines `MGBuilderCL`-Objektes erzeugt (siehe Konstruktor von `MultiGridCL`), welche im nächsten Abschnitt behandelt werden.

1.3 MultiGrid-Builder

Die Anfangstriangulierung $\mathcal{T}_0 = \mathcal{G}_0$ wird mit Hilfe einer Builder-Klasse konstruiert (siehe `geom/builder.h/cpp`). Derzeit gibt es in DROPS folgende Builder:

- `BrickBuilder` zur Erzeugung eines Quaders.
- `LBuilder` zur Erzeugung eines L-Gebietes.
- `BBuilder` zur Erzeugung eines B-Gebietes (Quader mit einspringender Ecke).
- `ReadMeshBuilderCL` liest GAMBIT (FLUENT) Mesh-Files.

```
class BrickBuilderCL : public MGBuilderCL
{
    BrickBuilderCL(const Point3DCL& orig, const Point3DCL& e1, const Point3DCL& e2, const Point3DCL& e3,
                   Uint n1, Uint n2, Uint n3);
};

class LBuilderCL : public MGBuilderCL
{
    LBuilderCL(const Point3DCL& origin, const Point3DCL& e1, const Point3DCL& e2, const Point3DCL& e3,
               Uint n1, Uint n2, Uint n3, Uint b1, Uint b2);
};

class BBuilderCL : public MGBuilderCL
{
    BBuilderCL(const Point3DCL& origin, const Point3DCL& e1, const Point3DCL& e2, const Point3DCL& e3,
               Uint n1, Uint n2, Uint n3, Uint b1, Uint b2, Uint b3);
};
```

```
class ReadMeshBuilderCL : public MGBuilderCL
{
    // Input stream, from which the mesh is read. Pass a pointer to an output stream,
    // e. g. msg= &std::cerr, if you want to know, what happens during multigrid-construction.
    ReadMeshBuilderCL(std::istream& f, std::ostream* msg= 0);

    BndCondT GetBC( Uint i)                    const;
    void     GetBC( std::vector<BndCondT>& BC) const;
};
```

1.3.1 `BrickBuilderCL`

Der Quader wird festgelegt durch den Ursprung `orig` und die aufspannenden Vektoren `e1, e2, e3`. Der Quader wird dann in $\mathtt{n1} \times \mathtt{n2} \times \mathtt{n3}$ Unterquader zerlegt, von denen jeder in 6 Tetraeder unterteilt wird.

Die 6 Randsegmente des Quaders werden in folgender Reihenfolge abgespeichert (als Beispiel der Einheitswürfel $\Omega = [0,1]^3$:)

- $x = 0, \quad x = 1,$
- $y = 0, \quad y = 1,$
- $z = 0, \quad z = 1.$

1.3.2 `LBuilderCL`

Das L-Gebiet wird ähnlich wie der Quader festgelegt. Die zusätzlichen Parameter $\mathtt{b1} < \mathtt{n1}$, $\mathtt{b2} < \mathtt{n2}$ legen die Aussparung des L-Gebietes fest, welche dem Ursprung gegenüber liegt und aus $(\mathtt{n1} - b1) \times (\mathtt{n2} - b2) \times \mathtt{n3}$ Unterquadern besteht.

Die 14 Randsegmente des L-Gebietes werden in folgender Reihenfolge abgespeichert (als Beispiel das L-Gebiet $\Omega = [0,1]^3 \setminus ([0.5,1]^2 \times [0,1])$):

- $x = 0, y \in [0,0.5], \quad x = 0, y \in [0.5,1], \quad x = 1, y \in [0,0.5], \quad x = 0.5, y \in [0.5,1],$
- $y = 0, x \in [0,0.5], \quad y = 0, x \in [0.5,1], \quad y = 1, x \in [0,0.5], \quad y = 0.5, x \in [0.5,1],$
- $z = 0, (x,y) \in [0,0.5] \times [0,0.5], \quad z = 0, (x,y) \in [0,0.5] \times [0.5,1], \quad z = 0, (x,y) \in [0.5,1] \times [0,0.5],$
  $z = 1, (x,y) \in [0,0.5] \times [0,0.5], \quad z = 1, (x,y) \in [0,0.5] \times [0.5,1], \quad z = 1, (x,y) \in [0.5,1] \times [0,0.5].$

1.3.3 `BBuilderCL`

Das B-Gebiet wird ähnlich wie der Quader festgelegt. Die zusätzlichen Parameter $\mathtt{b1} < \mathtt{n1}$, $\mathtt{b2} < \mathtt{n2}$, $\mathtt{b3} < \mathtt{n3}$ legen die Aussparung des B-Gebietes fest, welche dem Ursprung gegenüber liegt und aus $(\mathtt{n1} - b1) \times (\mathtt{n2} - b2) \times (\mathtt{n3} - b3)$ Unterquadern besteht.

Die 24 Randsegmente des L-Gebietes werden in folgender Reihenfolge abgespeichert (als Beispiel das L-Gebiet $\Omega = [0,1]^3 \setminus [0.5,1]^3$):

- $x = 0, (y,z) \in [0,0.5] \times [0,0.5], \quad x = 0, (y,z) \in [0,0.5] \times [0.5,1], \quad x = 0, (y,z) \in [0.5,1] \times [0,0.5],$
  $x = 0, (y,z) \in [0.5,1] \times [0.5,1],$
  $x = 1, (y,z) \in [0,0.5] \times [0,0.5], \quad x = 1, (y,z) \in [0,0.5] \times [0.5,1], \quad x = 1, (y,z) \in [0.5,1] \times [0,0.5],$
  $x = 0.5, (y,z) \in [0.5,1] \times [0.5,1].$
- analog für die x-z-Ebene.
- analog für die x-y-Ebene.

1.3.4 `ReadMeshBuilderCL`

Der Builder versteht das Format des „User's Guide“ zu TGrid 3, Anhang C, welches unter anderem von RAMPANT, FLUENT/UNS, TGrid verwendet wird und von dem Gittergenerator GAMBIT erzeugt wird. Die ID's der erzeugten Simplices entsprechen denen in der Datei.

Um Randsegmente zu repräsentieren, wird die Klasse `MeshBoundaryCL` verwendet. Pro Zone von Randdreiecken in der .msh-Datei wird ein `MeshBoundaryCL`-Objekt erzeugt, welches die zone-id und die boundary-condition enthält (siehe Ahnhang C des User's Guide).

2 `poisson`: Poisson-Gleichung

2.1 stationäre Poisson-Gleichung

Auf dem Gebiet Ω sei die (verallgemeinerte) Poissongleichung

$$\begin{aligned} -\Delta u(x) + q(x) \cdot u(x) &= f(x) && \text{in } \Omega \\ u(x) &= g_D(x) && \text{auf } \Gamma_D \\ \frac{\partial u(x)}{\partial n} &= g_N(x) && \text{auf } \Gamma_N \end{aligned}$$

gegeben, wobei die disjunkten Randstücke $\Gamma_D \cup \Gamma_N = \partial\Omega$ den Dirichlet- bzw. Neumannrand darstellen.

Für die Finite-Elemente-Diskretisierung mit P_1-Elementen steht die Problemklasse `PoissonP1CL` zur Verfügung:

```
template <class MGB, class Coeff>
class PoissonP1CL: public ProblemCL<MGB, Coeff, PoissonBndDataCL>
{
    IdxDescCL idx;
    VecDescCL x;
    VecDescCL b;
    MatDescCL A;

    PoissonP1CL(const MultiGridBuilderCL& mgb, const CoeffCL& coeff, const BndDataCL& bdata);
    // numbering of unknowns
    void CreateNumbering(Uint, IdxDescCL*);
    void DeleteNumbering(IdxDescCL*);
    // set up matrices and rhs
    void SetupSystem         (MatDescCL&, VecDescCL&) const;
    void SetupStiffnessMatrix(MatDescCL&) const;
    void SetupProlongation   (MatDescCL& P, IdxDescCL* cIdx, IdxDescCL* fIdx) const;
    // check computed solution etc.
    double CheckSolution(const VecDescCL&, scalar_fun_ptr) const;
    double CheckSolution(scalar_fun_ptr Lsg) const { return CheckSolution(x, Lsg); }
    void GetDiscError (const MatDescCL&, scalar_fun_ptr) const;
    void GetDiscError (scalar_fun_ptr Lsg) const { GetDiscError(A, Lsg); }

    bool          EstimateError         (const VecDescCL&, const double, double&, est_fun);
    static double ResidualErrEstimator  (const TetraCL&, const VecDescCL&, const BndDataCL&);
    static double ResidualErrEstimatorL2(const TetraCL&, const VecDescCL&, const BndDataCL&);

    DiscSolCL GetSolution() const;
};
```

2.2 instationäre Poisson-Gleichung

Auf dem Gebiet Ω sei die instationäre Poissongleichung bzw. Wärmeleitungsgleichung

$$\begin{aligned} \frac{\partial u(x,t)}{\partial t} - \Delta u(x,t) + q(x) \cdot u(x,t) &= f(x,t) && \text{in } \Omega \times [0, t_f] \\ u(x,0) &= u_0(x) && \text{in } \Omega \\ u(x,t) &= g_D(x) && \text{auf } \Gamma_D \times [0, t_f] \\ \frac{\partial u(x,t)}{\partial n} &= g_N(x) && \text{auf } \Gamma_N \times [0, t_f] \end{aligned}$$

gegeben.

3 stokes: Stokes-Gleichung

3.1 stationäre Stokes-Gleichung

3.2 instationäre Stokes-Gleichung

4 navstokes: Navier-Stokes-Gleichung

4.1 stationäre Navier-Stokes-Gleichung

4.2 instationäre Navier-Stokes-Gleichung

5 levelset: Levelset-Methode

Die `LevelsetP2CL` implementiert die Funktionalität der Levelset-Methode für P_2-Elemente.

```
template<class StokesProblemT>
class LevelsetP2CL
{
    LevelsetP2CL( MultiGridCL& mg, double theta= 0.5, double SD= 0.);

    void CreateNumbering( Uint level, IdxDescCL*);
    void DeleteNumbering();
    void Init( scalar_fun_ptr);
    void SetVel( const DiscVelSolCL& vel);
    // call SetupSystem *before* calling SetTimeStep!
    void SetupSystem();
    void SetTimeStep( double dt);
    void DoStep();
    void Reparam( Uint steps, double dt);

    bool   Intersects( const TetraCL&) const;
    double GetMass() const;
    static void AccumulateBndIntegral( const MultiGridCL&, VecDescCL& f, const VecDescCL& Phi, double sigma);

    DiscSolCL GetSolution() const;
};
```

6 num: Numerik

6.1 Vektoren und dünnbesetzte Matrizen

Datentypen für Vektoren und dünnbesetzte Matrizen sind in der Datei `num/spmat.h` definiert.

Als Vektordatentyp dient die Template-Klasse `VectorBaseCL<T>`, die direkt von `std::valarray<T>` abgeleitet ist und so die *expression templates* der Valarrays ausnutzt. Wenn `DebugNumericC` als Debug-Option gesetzt ist, findet beim Zugriff auf Vektoreinträge mit `[]` sowie bei Benutzung der Operatoren `+, -, =, +=, -=, *=, /=` eine Bereichsprüfung statt (im Gegensatz zum reinen valarray). Folgende Funktionen sind für Vektoren definiert:

- `dot(x,y)` berechnet das Skalarprodukt $x^T y$,
- `norm(x)` berechnet die euklidische Norm $\|x\|_2$, `norm_sq(x)` dessen Quadrat $\|x\|_2^2$,
- `supnorm(x)` berechnet die Maximumsnorm $\|x\|_\infty$,
- `axpy(a,x,y)` für `y+= a*x`,
- `z_xpay(z,x,a,y)` für `z= x + a*y`,
- `z_xpaypby2(z,x,a,y,b,y2)` für `z= x + a*y + b*y2`.

Die dünnbesetzten Matrizen werden in einem zeilenbasierten Format gespeichert und durch folgende Template-Klasse realisiert:

```
template <typename T>
class SparseMatBaseCL
{
    SparseMatBaseCL (size_t rows, size_t cols, size_t nz);

    size_t num_rows     () const;
    size_t num_cols     () const;
    size_t num_nonzeros () const;

    inline T  operator() (size_t i, size_t j) const;
    SparseMatBaseCL& operator*= (T c);
    SparseMatBaseCL& operator/= (T c);
    SparseMatBaseCL& LinComb (double, const SparseMatBaseCL<T>&,
                              double, const SparseMatBaseCL<T>&);
    void clear();
};
```

Mit Hilfe der Funktion `LinComb` wird die Matrix als Linearkombination zweier anderer Matrizen erzeugt (interessant bei der Zeitintegration: $M + \theta dt A$). Die Funktion `clear` dagegen löscht den Inhalt der Matrix.

Matrix-Vektor-Multiplikation erfolgt mit `A*x` für Ax und `transp_mul(A,x)` für $A^T x$.

Da das *sparsity pattern* einer Matrix vorher meist nicht bekannt ist, wird beim Aufstellen der Matrizen zunächst ein Zwischenformat, die `SparseMatBuilderCL` erzeugt, in der das *sparsity pattern* aufgebaut wird, bevor schließlich die endgültige Matrix mit dem Befehl `Build` erzeugt wird.

```
template <typename T>
class SparseMatBuilderCL
{
    SparseMatBuilderCL(spmatT* mat, size_t rows, size_t cols);

    T& operator() (size_t i, size_t j);
    void Build();
};
```

Besitzt die an den Builder übergebene Matrix bereits ein *sparsity pattern*, so wird dieses aus Effizienzgründen wiederverwendet. Deswegen sollte nach jeder Gitteränderung der Inhalt der Matrizen mit `clear` gelöscht werden, um das *sparsity pattern* neu aufbauen zu lassen.

Zur einfacheren Handhabung sind die folgenden typedefs definiert:

```
typedef VectorBaseCL<double>       VectorCL;
typedef SparseMatBaseCL<double>    MatrixCL;
typedef SparseMatBuilderCL<double> MatrixBuilderCL;
```

6.2 Vorkonditionierer

Die Vorkonditionierer für ein lineares Gleichungssystem $Ax = b$ sind in `num/solver.h` definiert. Sie haben im wesentlichen die folgende Struktur:

```
template <PreMethGS PM>
class PreGSCL<PM, false>
{
  private:
    double _omega;

  public:
    PreGSCL (double om= 1.0) : _omega(om) {}

    template <typename Vec>
    void Apply(const MatrixCL& A, Vec& x, const Vec& b) const
};
```

Die Funktion `Apply` wendet den Vorkonditionierer an. Zur Zeit gibt es folgende Verfahren:

- `SORsmoothCL`, Gauss-Seidel-Verfahren mit Überrelaxation.
- `SGSPcCL`, symmetrisches Gauss-Seidel-Verfahren für den Startvektor $x = 0$.
- `SSORPcCL`, SSOR-Verfahren für den Startvektor $x = 0$.
- `SSORDiagPcCL`, SSOR-Verfahren, für schnelleren Zugriff auf die Diagonale von A optimiert.
- `DummyPcCL`, die Identität als trivialer Vorkonditionierer.

Ferner existiert die Funktion `SolveGSstep(const PreDummyCL<PB_JAC>&, const MatrixCL& A, Vec& x, const Vec& b, double omega)`, die einen Jacobi-Schritt zum Startvektor x durchführt. Der Funktionsname ist durch die Auswahlregeln für überladene Funktionen in C++ bedingt.

Die eigentliche Information über den Typ des Vorkonditionierers steckt in `enum PreMethGS` bzw. der Template-Klasse `template <PreBaseGS> class PreDummyCL {}`, die die `enum`-Werte auf eigene C++-Typen abbildet.

6.2.1 Vorkonditionierer für die stationären Stokes-Gleichungen

In `stokes/sdropsP2.cpp` befindet sich die Klasse

```
class MGPreCL
{
    MGPreCL( const MGDataCL& mgd, const MatrixCL& ps);

    template <typename Mat, typename Vec>
    void
    Apply(const Mat& K, Vec& x, const Vec& b) const;
};
```

Sie implementiert einen Blockdiagonal-Vorkonditionierer für die stationären Stokes-Gleichungen. Die Argumente x, b von `Apply` müssen Zugriff auf die Gechwindigkeitskomponenten ermöglichen (via der Elementfunktion `u`) und ebenso auf die Druckkomponenten (via `p`). Dies leistet im Moment die Vektorklasse `StokesVectorCL`, welche in `stokes/sdropsP2.cpp` definiert ist.

Die Matrix K repräsentiert die Block-Matrix zu den Stokes-Gleichungen und muß Zugriff auf die Teilmatrizen A und B erlauben. Zur Zeit die in `stokes/sdropsP2.cpp` definierte Klasse `StokesMatrixCL` eingesetzt.

Der Vorkonditionierer verwendet für den Poissonanteil der Stokes-Gleichungen eine Iteration des übergebenen Mehrgitter-V-Zyklus mit den in diesem eingestellten Glättern.

Die Schur-Komplementmatrix wird durch einen SSOR-Schritt mit der P_2-Massenmatrix vorkonditioniert.

Als traditionelle Alternative kann ein ähnlicher aufgebauter Vorkonditionierer vom Typ

```
class FullPreCL
{
    FullPreCL( const MatrixCL& ps);

    template <typename Mat, typename Vec>
    void
    Apply(const Mat& K, Vec& x, const Vec& b) const;
};
```

verwendet werden. Im Unterschied zu `MGPreCL` wird hier der Poissonanteil mit Hilfe eines SSOR-vorkonditionierten PCG-Lösers näherungsweise gelöst.[1]

Beide Vorkonditionierer liefern bereits korrekte Ergebnisse. Die Implementierung hat sich jedoch in Details noch nicht stabilisiert. Deshalb befindet sich der gesamte Code in `stokes/sdropsP2.cpp`.

6.3 Poisson-Löser

Die iterativen Löser für lineare Gleichungssysteme sind in `num/solver.h` implementiert. Es gibt einen CG-Löser,

```
template <typename Mat, typename Vec>
bool
CG(const Mat& A, Vec& x, const Vec& b, int& max_iter, double& tol),
```

einen PCG-Löser, der die in 6.2 beschriebenen Vorkonditionierer verwenden kann:

```
template <typename Mat, typename Vec, typename PreCon>
bool
PCG(const Mat& A, Vec& x, const Vec& b, const PreCon& M,
    int& max_iter, double& tol),
```

Ein MINRES-Löser

```
template <typename Mat, typename Vec>
bool
MINRES(const Mat& A, Vec& x, const Vec& rhs, int& max_iter, double& tol);
```

und ein vorkonditionierter PMINRES-Löser

```
template <typename Mat, typename Vec, typename Lanczos>
bool
PMINRES(const Mat& A, Vec& x, const Vec& rhs, Lanczos& q, int& max_iter, double& tol);
```

stehen zur Verfügung. Sie können auch direkt zum Lösen der Stokes-Gleichungen verwendet werden. Beispiele dazu findet man in `stokes/sdropsP2`.

Ferner existiert eine GMRES-Implementierung:

```
template <typename Mat, typename Vec, typename PreCon>
bool
GMRES(const Mat& A, Vec& x, const Vec& b, const PreCon& M,
      int m, int& max_iter, double& tol).
```

Um die Handhabung der Löser zu erleichtern gibt es die Klasse

[1] Diese Klasse ist viel ineffizienter als `MGPreCL`.

```
class SolverBaseCL
{
  protected:
    int    _maxiter, _iter;
    double _tol, _res;

    SolverBaseCL (int maxiter, double tol)
        : _maxiter(maxiter), _iter(-1), _tol(tol), _res(-1.) {}

  public:
    void   SetTol    (double tol) { _tol= tol; }
    void   SetMaxIter(int iter)   { _maxiter= iter; }

    double GetTol    () const { return _tol; }
    int    GetMaxIter() const { return _maxiter; }
    double GetResid  () const { return _res; }
    int    GetIter   () const { return _iter; }
},
```

welche als Basisklasse für die iterativen Löser dient. Die abgeleiteten Klassen enthalten die Elementfunktion `Solve`. Diese ruft den eigentlichen Löser mit den vorher gewählten Parametern auf.

6.4 Stokes-Löser

Die Stokes-Löser ermitteln die Lösung eines linearen Gleichungssystems der Form

$$\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} b \\ c \end{pmatrix}.$$

Sie sind in `num/stokessolver.h` implementiert.

6.4.1 Schur-Verfahren

Die Schur-Komplementmatrix $S = BA^{-1}B^T$ wird in Drops durch die Klasse `SchurComplMatrixCL` in `stokes/stokes.h` realisiert. Der Matrix-Vektor-Multiplikationsoperator verwendet zu Lösung des Gleichungssystems mit A ein PCG-Verfahren mit dem in der `SchurComplMatrixCL` definierten Vorkonditionierer. Dies ist gegenwärtig die `SSORPcCL`.

Damit kann das Schur-Verfahren in `SchurSolverCL` und `PSchurSolverCL` so darstellen:

1. $r = BA^{-1}b - c$ berechnen: $r = -c$, $Av = b$ lösen, $r+ = Bv$.
2. $Sp = r$ lösen.
3. $Av = b - B^T p$ lösen.

Beide Klassen verwenden in Schritt 1 und 3 einen der Poissonlöser, dessen Typ als Template-Parameter in die Klasse gebracht wird (Hält eine Referenz auf den Löser.). Ein Objekt vom Lösertyp wird erst durch öffentliches Ableiten zugefügt. In Schritt 2 verwendet `SchurSolverCL` das CG-Verfahren und `PSchurSolverCL` ein PCG-Verfahren mit SSOR-Vorkonditionierung einer Matrix in den Druckunbekannten (z. B. Massenmatrix).

Zur einfacheren Handhabung werden folgende abgeleiteten Klassen bereitgestellt:

```
class Schur_PCG_CL: public SchurSolverCL<PCG_SsorCL>
{
    Schur_PCG_CL(int outer_iter, double outer_tol, int inner_iter, double inner_tol);
};


class PSchur_PCG_CL: public PSchurSolverCL<PCG_SsorCL>
{
```

```
    PSchur_PCG_CL( MatrixCL& M, int outer_iter, double outer_tol, int inner_iter, double inner_tol);
};


class PSchur_IPCG_CL: public PSchurSolverCL<PCG_SsorDiagCL>
{
    PSchur_IPCG_CL( MatrixCL& M, int outer_iter, double outer_tol, int inner_iter, double inner_tol);
};


class PSchur_GSPCG_CL: public PSchurSolverCL<PCG_SgsCL>
{
    PSchur_GSPCG_CL( MatrixCL& M, int outer_iter, double outer_tol, int inner_iter, double inner_tol);
};


class PSchur_MG_CL: public PSchurSolverCL<MGSolverCL>
{
    PSchur_MG_CL( MatrixCL& M,      int outer_iter, double outer_tol,
                  MGDataCL& MGData, int inner_iter, double inner_tol );
};
```

6.4.2 Uzawa-Verfahren

Die folgenden Uzawa-Verfahren stehen zur Verfügung:

```
template <typename PoissonSolverT>
class UzawaSolverCL : public SolverBaseCL
{
    UzawaSolverCL (PoissonSolverT& solver, MatrixCL& M, int maxiter, double tol, double tau= 1.);

    void Solve( const MatrixCL& A, const MatrixCL& B, VectorCL& v, VectorCL& p,
                const VectorCL& b, const VectorCL& c);
// ...
};

class Uzawa_IPCG_CL : public SolverBaseCL
{
    Uzawa_IPCG_CL(MatrixCL& M, int outer_iter, double outer_tol, int inner_iter, double inner_tol, double tau= 1.);

    // Always call this when A has changed, before Solve()!
    void Init_A_Pc(MatrixCL& A) { _A_IPCGsolver.GetPc().Init(A); }

    inline void Solve( const MatrixCL& A, const MatrixCL& B, VectorCL& v, VectorCL& p,
                       const VectorCL& b, const VectorCL& c);
// ...
};
```

In der ersten Klasse ist der Typ des Poissonlösers als Template-Parameter gegeben. Für die in DROPS vorhandenen Poisson-Verfahren existieren in `num/stokessolver.h` von `UzawaSolverCL` abgeleitete Klassen, die den entsprechenden Löser in den im folgenden Algorithmus auftretenden Poisson-Problemen einsetzen:

1. $Mdp = (Bv - c)$ lösen.
2. $p+ = dp$.
3. $Adv = Av + B^T p - b$ lösen.
4. $v- = dv$.

In der Klasse `Uzawa_IPCG_CL` können in Schritt 1 und 3 unterschiedliche Poisson-Verfahren benutzt werden. Zur Zeit sind beide Löser SSOR-vorkonditionierte PCG-Verfahren.

Zur einfacheren Handhabung werden folgende abgeleiteten Klassen bereitgestellt:

```
class Uzawa_CG_CL : public UzawaSolverCL<CGSolverCL>
{
    Uzawa_CG_CL( MatrixCL& M, int outer_iter, double outer_tol, int inner_iter, double inner_tol, double tau= 1.);
};

class Uzawa_PCG_CL : public UzawaSolverCL<PCG_SsorCL>
{
    Uzawa_PCG_CL( MatrixCL& M, int outer_iter, double outer_tol, int inner_iter, double inner_tol, double tau= 1.);
};

class Uzawa_SGSPCG_CL : public UzawaSolverCL<PCG_SgsCL>
{
    Uzawa_SGSPCG_CL( MatrixCL& M, int outer_iter, double outer_tol, int inner_iter, double inner_tol, double tau= 1.);
};

class Uzawa_MG_CL : public UzawaSolver2CL<PCG_SsorCL, MGSolverCL>
{
    Uzawa_MG_CL(MatrixCL& M,      int outer_iter, double outer_tol,
                MGDataCL& MGData, int inner_iter, double inner_tol, double tau= 1.);
};
```

6.5 Navier-Stokes-Löser

Im`num/nssolver.h` sind die Lösungsverfahren für die stationären Navier-Stokes-Gleichungen definiert. Sie lösen das nicht-lineare Gleichungssystem

$$\begin{pmatrix} A + \alpha N(v) & B^T p \\ B & 0 \end{pmatrix} \begin{pmatrix} v \\ p \end{pmatrix} = \begin{pmatrix} b \\ c \end{pmatrix}.$$

Der Parameter α ist hier stets 1, wird jedoch für die Zeitintegration benötigt. In DROPS wird ein Fixpunktverfahren eingesetzt, in dem folgende Schritte bis zur Konvergenz iteriert werden:

1. Stelle $N(v)$ auf.
2. Löse
$$\begin{pmatrix} A + \alpha N(v) & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} w \\ q \end{pmatrix} = \begin{pmatrix} (A + \alpha N(v))v + B^T p - b \\ Bv - c \end{pmatrix}.$$
3. $v- = w$, $p- = q$.

Die Implementierung findet man in der Klasse `FixedPtDefectCorrCL`, deren abgeleitete Klassen konkrete Stokes-Löser aus `num/stokessolver.h` verwenden.

Nach Turek (Seite 187 ff.) kann man obige Defektkorrektur dämpfen. Diese Variante des Fixpunktverfahrens wird durch die Klasse `AdaptFixedPtDefectCorrCL` implementiert.

7 Zeitintegration

Die Verfahren zur Lösung zeitabhängiger Gleichungen befinden sich in der Erprobung und haben noch keinen festen Platz in der DROPS-Verzeichnishierarchie. In `poisson`, `stokes` und `navstokes` gibt es je eine Datei `integrtime.h`, die stets ein Theta-Schema enthält.

7.1 θ-Schema

Diese Verfahren ergeben sich dadurch, daß man

$$\partial_t u \approx \frac{(1-\theta)u^{\text{alt}} + \theta u^{\text{neu}}}{dt}$$

in die entsprechende Differentialgleichung einsetzt. Man erhält für jeden Zeitschritt ein (nicht-) lineares Gleichungssystem. Bei den Stokes- bzw. Navier-Stokes-Gleichungen wird der Druck stets vollimplizit diskretisiert. Auf diese Weise wird die Berechnung eines Startwertes für den Druck vermieden.

Die Klassen `Instat***ThetaSchemeCL` enthalten eine Funktion `SetTimeStep`, die vor jedem Zeitschritt aufgerufen werden muß. Sie stellt das Gleichungssystem mit der Massenmatrix auf und setzt die Zeitschritt-Größe. Der eigentliche Zeitschritt wird durch einen Aufruf von `DoStep` durchgeführt.

7.2 Fractional-Step-Verfahren

Für die Stokes-Gleichungen ist ein weiteres Verfahren verfügbar: (XXX beschreiben. Ist im Moment defekt...)

8 `out`: Ausgabe

8.1 Ausgabe in Geomview

- `GeomMGOutCL`: nur Triangulierung ausgeben.
- `GeomSolOutCL`: Triangulierung ausgeben, die nach den Werten einer skalaren Variable eingefärbt ist.

```
class GeomMGOutCL : public MGOutCL
{
    GeomMGOutCL (const MultiGridCL& MG, int TriLevel=-1, bool onlyBnd=false, double explode=0.5);
};

template <class DiscSol>
class GeomSolOutCL : public MGOutCL
{
    GeomSolOutCL (const MultiGridCL& MG, const DiscSol& discsol, const ColorMapperCL* colmap, int TriLevel=-1,
                  bool onlyBnd=false, double explode=0.5, double min=0., double max=1.);
};
```

Mit `onlyBnd=true` werden nur die am Rand liegenden Tetraeder ausgegeben (schnellerer Aufbau von Postscript-Bildern!). Über den Parameter `explode` kann der Versatz der Tetraeder untereinander gesteuert werden, bei `explode=0` ist dieser Effekt ausgeschaltet. Die Parameter `min, max` geben an, ab welchen Grenzen die Werte der dargestellten Lösung geclippt werden sollen.

Die Geomview-Ausgabe ist vor allem für die Ausgabe der Geometrie und Triangulierung geeignet. Um Lösungen zu betrachten, sollten die folgende TecPlot- (für 2D-Daten) oder Ensight-Ausgabe (für 3D-Daten) genutzt werden.

8.2 Ausgabe in TecPlot

- `TecPlot2DSolOutCL`: Ausgabe von Druck und Geschwindigkeit in einer Schnittebene senkrecht zu einer der Koordinatenachsen.
- `TecPlotSolOutCL`: Ausgabe von Druck und Geschwindigkeit im gesamten Gebiet. Da Tecplot zur 3D-Visualisierung nicht so geeignet ist, empfiehlt sich anstattdessen eine Ausgabe für Ensight!

```
template <class DiscVelT, class DiscPrT>
class TecPlot2DSolOutCL: public MGOutCL
{
    TecPlot2DSolOutCL( const MultiGridCL& mg, const DiscVelT& v, const DiscPrT& p,
                       const IdxDescCL& freeVertIdx, int lvl, Uint cutvar, double cutplane);
};
```

```
template <class DiscVelT, class DiscPrT>
class TecPlotSolOutCL: public MGOutCL
{
    TecPlotSolOutCL( const MultiGridCL& mg, const DiscVelT& v, const DiscPrT& p, int lvl= -1);
};
```

Die Klasse `TecPlot2DSolOutCL` benötigt zur Nummerierung der Knoten einen Index, in dem alle P_1-Freiheitsgrade durchnummeriert sind. Dieser wird nicht verändert, es kann also z.B. auch der Index der Druckvariable hierzu benutzt werden. Hier ein Beispiel zur Benutzung:

```
DROPS::IdxDescCL tecIdx;
tecIdx.Set( 1);
Stokes.CreateNumberingPr( mg.GetLastLevel(), &tecIdx);

std::ofstream v2d("data2D.dat");
DROPS::TecPlot2DSolOutCL< MyStokesCL::DiscVelSolCL, MyStokesCL::DiscPrSolCL>
    tecplot2d( mg, prob.GetVelSolution(), prob.GetPrSolution(), tecIdx, -1, 1, 0.5); // cutplane is y=0.5
v2d << tecplot2d;
v2d.close();
```

8.3 Ausgabe in Ensight

`EnsightP2SolOutCL`: Ausgabe im Ensight-Case-Format, auch transiente Daten möglich.

```
class EnsightP2SolOutCL
{
    EnsightP2SolOutCL( const MultiGridCL& mg, const IdxDescCL* idx);
    // call CaseBegin() before any other member function
    void CaseBegin ( const char casefileName[], Uint numsteps= 0);
    void DescribeGeom  ( const char geoName[], std::string fileName, bool timedep= false);
    void DescribeScalar( const char varName[], std::string fileName, bool timedep= false);
    void DescribeVector( const char varName[], std::string fileName, bool timedep= false);
    void putGeom      ( std::string, double t= -1);
    template<class DiscScalT>
    void putScalar    ( std::string, const DiscScalT&, double t= -1);
    template<class DiscVecT>
    void putVector    ( std::string, const DiscVecT&, double t= -1);
    void Commit    ();    // rewrites case file
    // call CaseEnd() after finishing all other output
    void CaseEnd   ();
};
```

Die Klasse `EnsightP2SolOutCL` benötigt einen eigenen Index, in dem alle P_2-Freiheitsgrade durchnummeriert sind. Die Aufrufe von Elementfunktionen werden durch `CaseBegin` am Anfang und `CaseEnd` am Ende eingeschlossen. Beachte, dass das Case-File in der Regel fehlerhaft sein wird, wenn `CaseEnd` nicht aufgerufen wird. Die Funktionen `DescribeXXX` dienen zur Beschreibung der Geometrie und skalarer oder vektorwertiger Variablen und werden einmal zu Anfang aufgerufen. Der Parameter `timedep` gibt dabei an, ob es sich um transiente Daten handelt. Mit den Funktionen `putXXX` können die Daten dann tatsächlich abgespeichert werden. Bei transienten Daten muss dabei noch der entsprechende Zeitpunkt angegeben werden. Sind alle Daten für einen Zeitschritt geschrieben, so sollte `Commit` aufgerufen werden, um den Zeitschritt im case-file eintragen zu lassen. Somit können die bereits berechneten Zeitschritte visualisiert werden, während die übrige Rechnung noch läuft.

Hier ein Beispiel zur Ausgabe von transienten skalaren Daten bei festem Gitter und festem Geschwindigkeitsfeld:

```
IdxDescCL lidx;
lidx.Set( 1, 1);
lset.CreateNumbering( mg.GetLastLevel(), &lidx);
EnsightP2SolOutCL ensight( mg, &lidx);

const char datgeo[]= "ensight/drop.geo",
           datvec[]= "ensight/drop.vec",
```

```
            datscl[]= "ensight/drop.scl";

ensight.CaseBegin     ( "drop.case", num_steps+1);
ensight.DescribeGeom  ( "rotating vel field", datgeo);
ensight.DescribeScalar( "Levelset", datscl, true);
ensight.DescribeVector( "Velocity", datvec);

ensight.putGeom( datgeo);
ensight.putVector( datvec, prob.GetVelSolution());
ensight.putScalar( datscl, lset.GetSolution(), 0);

for (int i=1; i<=num_steps; ++i)
{
    // berechne Loesung zum neuen Zeitschritt
    // ...
    ensight.putScalar( datscl, lset.GetSolution(), i/10.);
    ensight.Commit();
}

ensight.CaseEnd();
```

9 misc: Verschiedenes

9.1 Einlesen von Parameterdaten

Zum Einlesen von Parameterdaten aus Dateien wird die Klasse `ReadParamsCL` aus `misc/params.h` zur Verfügung gestellt. Das Einlesen von Paramatern der Typen `int`, `double`, `Point3DCL` und `string` wird damit stark vereinfacht.

```
class ReadParamsCL
{
  public:
    ReadParamsCL() {}

    // register parameters
    void RegInt   ( int&,       string);
    void RegDouble( double&,    string);
    void RegCoord ( Point3DCL&, string);
    void RegString( string&,    string);

    // build groups (may be used recursively)
    void BeginGroup( const string& group);
    void EndGroup();

    // IO
    void ReadParams( std::istream&);
    void WriteParams( std::ostream&) const;

    //cleanup
    void Clear(); // deallocate memory
};
```

Vor Einlesen der Datei mit `ReadParams` müssen die einzelnen Parameter mit Hilfe der Elementfunktionen `RegXXX` registriert werden. Die Parameter können dabei mit Hilfe von Gruppen geordnet werden, die auch geschachtelt sein dürfen. Beispiel:

```
ReadParamsCL rp;
rp.BeginGroup("Solver");
    rp.BeginGroup("Stokes");
        rp.RegDouble( outer_tol, "Tol");
        rp.RegInt( outer_iter,   "Iter");
    rp.EndGroup();
    rp.BeginGroup("Levelset");
        rp.RegDouble( lset_tol, "Tol");
        rp.RegInt( lset_iter,   "Iter");
    rp.EndGroup();
rp.EndGroup();
```

Z.B. ist nun `lset_tol` unter dem Namen `Solver:Levelset:Tol` registriert. Eine entsprechende Parameterdatei könnte so aussehen:

```
# Dies ist ein Kommentar.
Solver
{
    Levelset {
        Tol = 1e-8 # Genauigkeit
        Iter = 1000 # max. Anzahl Iterationen
    }
    Stokes {
        Tol = 1e-6
        Iter = 100
    }
}
```

Alle Zeichen hinter einem `#` werden bis zum Zeilenende ignoriert. Statt Gruppen zu bilden, hätte man auch die Parameter direkt ansprechen können, z.B. `Solver:Levelset:Tol = 1e-8`.

Üblicherweise werden alle Parameter für ein Hauptprogramm in einer Klasse zusammengefasst, die von der `ParamBaseCL` (`misc/params.h`) abgeleitet ist. Diese enthält ein `ReadParamsCL`-Objekt; Lesen und Schreiben der Parameter erfolgt (nach deren Registrierung) über den Eingabe-/Ausgabeoperator `>>` bzw. `<<`.

```
class ParamBaseCL
{
  protected:
    ReadParamsCL rp_;

  public:
    void Clear() { rp_.Clear(); } // cleanup

    friend std::istream& operator >> ( std::istream&, ParamBaseCL&);
    friend std::ostream& operator << ( std::ostream&, const ParamBaseCL&);
};
```

Ein Beispiel für konkrete Parameterklassen findet sich z.B. in `levelset/params.h`. Die Bedeutung der Parameter ist in Anhang A erläutert.

A Parameter Files

A.1 Stokes Flow, zweiphasig

```
#============================================================
#    DROPS parameter file
#    simulation of two-phase flow:
#    droplet in measuring cell used in NMR measurements
#============================================================

# time stepping
Time {
  NumSteps = 500
  StepSize = 1e-4
  Theta = 0.5 # Crank-Nicholson
}

# flow solver
Stokes {
  InnerIter = 1000
  OuterIter = 200
  InnerTol = 1e-14
  OuterTol = 1e-10
}

# levelset solver
Levelset {
  Tol = 1e-14
  Iter = 10000
  SD = 0.1
  CurvDiff = 5e-9
  VolCorrection = 1
}

# re-initialization of levelset function
Reparam {
  Freq = 0 # 0 = no reparametrization
  NumSteps = 5
  StepSize = 0.001
  Diffusion = 1e-4
}

# material data, all units are SI
Mat {
  DensDrop = 955
  ViscDrop = 2.6e-3
  DensFluid = 1107
  ViscFluid = 1.2e-3
  SmoothZone = 1e-4

  SurfTension = 0
}

# experimental conditions
Exp {
  RadDrop = 1.75e-3
  PosDrop = 0 -8e-3 0

  Gravity = 0 -9.81 0

  FlowDir = 1 # flow in y-direction

  InflowVel = -0.1
  RadInlet = 3.5e-3
}

# miscellaneous

NumDropRef = 2
CouplingSteps = -1 # -1 = till convergence

MeshFile = gambit/NMR_klein_grob.msh
EnsightCase = NMRmzi
```

```
EnsightDir = ensight
```

Bedeutung der Parameter:

`Time:NumSteps` > 0 **Typ:** `int`

Anzahl der Zeitschritte.

`Time:StepSize` > 0 **Typ:** `double`

Größe des Zeitschrittes.

`Time:Theta` $\in [0, 1]$ **Typ:** `double`

steuert die Implizitheit des θ-Schemas. Für `Theta=0` erhält man das explizite Eulerverfahren, für `Theta=1` erhält man das implizite Eulerverfahren und für `Theta=0.5` das Crank-Nicholson-Verfahren.

`Stokes:InnerIter` ≥ 0 **Typ:** `int`

Maximale Iterationszahl des inneren Lösers.

`Stokes:OuterIter` ≥ 0 **Typ:** `int`

Maximale Iterationszahl des äußeren Lösers.

`Stokes:InnerTol` > 0 **Typ:** `double`

Abbruchkriterium für den inneren Löser: Ist das erreichte Residuum kleiner als die vorgegebene Toleranz `InnerTol`, so wird die Iteration abgebrochen. *Hinweis:* `InnerTol` sollte einige Größenordnungen kleiner als `OuterTol` gewählt werden, da sonst der äußere Löser divergiert.

`Stokes:OuterTol` > 0 **Typ:** `double`

Abbruchkriterium für den äußeren Löser: Ist das erreichte Residuum kleiner als die vorgegebene Toleranz `OuterTol`, so wird die Iteration abgebrochen. Die Impuls- und Massenerhaltungsgleichung werden also bis zu diesem Residuum gelöst.

`Levelset:Iter` ≥ 0 **Typ:** `int`

Maximale Iterationszahl des Levelset-Lösers. Dies ist ein GMRES-Löser, der für die Lösung der Gleichungen eingesetzt wird, die die Levelset-Funktion als Unbekannte enthalten: Dies sind die Advektionsgleichung, die Reparametrisierungsgleichung und die Glättung zur Krümmungsberechnung.

`Levelset:Tol` > 0 **Typ:** `double`

Abbruchkriterium für den Levelset-Löser. Ist das erreichte Residuum kleiner als die vorgegebene Toleranz `Tol`, so wird die Iteration abgebrochen.

`Levelset:SD` > 0 **Typ:** `double`

steuert die Stabilisierung der Advektionsgleichung mit *streamline diffusion*. `SD=0` bedeutet keine Stabilisierung. Ein typischer Wert zur Stabilisierung ist `SD=0.1`.

`Levelset:CurvDiff` $\ll 1$ **Typ:** `double`

Bei starken Oberflächenspannungen ist es ratsam, zur Berechnung des Krümmungsterms eine geglättete Levelset-Funktion zu verwenden. Ein typischer Wert ist `CurvDiff=1e-8`. Ist `CurvDiff` ≤ 0, so erfolgt keine Glättung. *Hinweis:* Die Glättung erfolgt nur für eine temporäre Variable, die zur Krümmungsberechnung verwendet wird; die eigentliche Phasengrenze wird nicht verändert.

`Levelset:VolCorrection` $\in \{0, 1\}$ **Typ:** `bool`

schaltet die Volumenkorrektur an (`VolCorrection=1`) oder aus (`VolCorrection=0`). Die Korrektur ist global, d.h. das Gesamtvolumen bleibt über der Zeit konstant. Die Korrektur erfolgt nach jedem Zeitschritt sowie nach jeder Reparametrisierung der Levelset-Funktion.

`Reparam:Freq` ≥ 0 **Typ:** `int`

gibt an, nach wievielen Zeitschritten jeweils die Levelset-Funktion reparametrisiert werden soll. `Freq=0` schaltet die Reparametrisierung ab.

`Reparam:NumSteps` > 0 **Typ:** `int`

Bei der Reparametrisierung wird eine Transportgleichung in der künstlichen Zeit τ gelöst. `NumSteps` legt die Anzahl der Zeitschritte fest.

`Reparam:StepSize` > 0 **Typ:** `double`

`StepSize` legt die Größe des Zeitschrittes $\Delta\tau$ fest. Wird bis zum Zeithorizont $\tau_f =$ `NumSteps`$\cdot$`StepSize` gelöst, so ist die Levelset-Funktion in einem Streifen der Breite τ_f um die Phasengrenze herum wieder eine Abstandsfunktion (jedenfalls theoretisch ;-).

`Reparam:Diffusion` ≥ 0 **Typ:** `double`

Zusätzliche Diffusion bei der Reparametrisierung bewirkt eine Glättung der Phasengrenze. `Diffusion` steuert den Anteil des diffusiven Terms gegenüber dem konvektiven Term.

`Mat:DensDrop` > 0 **Typ:** `double`

Dichte des Tropfens in $kg \cdot m^{-3}$.

`Mat:ViscDrop` > 0 **Typ:** `double`

Dynamische Viskosität des Tropfens in $kg \cdot s^{-1}\, m^{-1}$ or $Pa \cdot s$.

`Mat:DensFluid` > 0 **Typ:** `double`

Dichte des umgebenden Fluids in $kg \cdot m^{-3}$.

`Mat:ViscFluid` > 0 **Typ:** `double`

Dynamische Viskosität des umgebenden Fluids in $kg \cdot s^{-1}\, m^{-1}$ or $Pa \cdot s$.

`Mat:SurfTension` ≥ 0 **Typ:** `double`

Oberflächenspannung in $kg \cdot s^{-2}$ oder N/m.

`Mat:SmoothZone` ≥ 0 **Typ:** `double`

Der Sprung von Viskosität und Dichte wird an der Phasengrenze numerisch geglättet, so dass ein Übergangsbereich der Breite `SmoothZone` rund um die Phasengrenze entsteht.

`Exp:RadDrop` **Typ:** `double`

Radius des kugelförmigen Tropfens in m zum Anfangszeitpunkt. *Hinweis:* Ist `RadDrop` negativ, so wird die Strömung des Fluids ohne Tropfen berechnet.

`Exp:PosDrop` **Typ:** `Point3DCL`

Mittelpunktposition des kugelförmigen Tropfens in m zum Anfangszeitpunkt.

`Exp:Gravity` **Typ:** `Point3DCL`

Wirkungsrichtung und Stärke der Schwerkraft in $kg \cdot m \cdot s^{-2}$.

`Exp:FlowDir` $\in \{0, 1, 2\}$ **Typ:** `int`

Richtung der Strömung am Einfluss. `InflowDir=0/1/2` bezeichnen jeweils die x-/y-/z-Richtung. Fließt die Strömung in *negative* Koordinatenrichtung, so muss beim Parameter `Exp:InflowVel` ein negatives Vorzeichen gewählt werden.

`Exp:InflowVel` **Typ:** `double`

Einströmgeschwindigkeit in $m \cdot s^{-2}$.

`Exp:RadInlet` > 0 **Typ:** `double`

Radius der kreisförmigen Einlassöffnung in m.

`NumDropRef` ≥ 0 **Typ:** `int`

Anzahl der zusätzlichen Verfeinerungen des eingelesenen Gitters im Bereich des Tropfens.

`CouplingSteps` ≥ -1 **Typ:** `int`

Maximale Anzahl der Fixpunktschritte zur Kopplung der Levelset- und (Navier-)Stokes-Gleichung. `CouplingSteps=-1` bewirkt, dass jeweils bis zur Konvergenz der Fixpunktiteration iteriert wird.

`MeshFile` **Typ:** `string`

Name der Mesh-Datei, aus der das Gitter eingelesen wird. Dieses muss im Format GAMBIT/FLUENT UNS abgespeichert sein.

`EnsightCase` **Typ:** `string`

Name, unter dem die Daten zur Visualisierung mit Ensight in Dateien ausgegeben werden sollen. Das case file liegt im aktuellen Verzeichnis und erhält die Endung `.case`, die übrigen Dateien werden im Verzeichnis `EnsightDir` abgespeichert.

`EnsightDir` **Typ:** `string`

Name des Verzeichnisses, in dem die Geometriedaten (Endung `.geom*`) und die Variablen in Dateien

abgelegt werden (Endung `.pr*` für Druck, `.vec*` für Geschwindigkeit, `.scl*` für Levelset).

A.2 Navier-Stokes Flow, zweiphasig

Folgende Parameter kommen gegenüber dem Stokes Flow hinzu:

```
NavStokes {
  Nonlinear = 0.5
  Scheme = 0 # Baensch
  NonlinearStat = 0
  ThetaStat = 0.5
}
```

Bedeutung der Parameter:

`NavStokes:Nonlinear` $\in [0,1]$ **Typ:** `double`

Koeffizient vor dem nichtlinearen Term, um die Stärke der Nichtlinearität beeinflussen zu können. Bei `Nonlinear=0` erhält man die Stokes-Gleichung, bei `Nonlinear=1` wird die volle Navier-Stokes-Gleichung gelöst.

`NavStokes:Scheme` $\in \{0,1\}$ **Typ:** `int`

bestimmt das Zeitintegrationsschema für die Lösung der instationären Navier-Stokes-Gleichung. `Scheme=1` ist das θ-Schema, was auch zur Lösung der Stokes-Gleichung eingesetzt wird, allerdings mit einem anderen Löser (GMRES, aufgrund des zusätzlichen Terms sind die Matrizen i.A. nicht mehr symmetrisch). Robuster ist das modifizierte fractional step scheme (siehe Bänsch), das bei `Scheme=0` gewählt wird.

...