AMDiS

Adaptive Multi-Dimensional Simulations

An overview about the Finite-Element Toolbox

Simon Praetorius

The Publisher $\boxed{\mathcal{PL}}$

Published by

Technische Universität Dresden
School of Sciences
D-01062 Dresden
Germany

phone: +49 123 45 67 89
e-mail: simon.praetorius@tu-dresden.de
http://www.tu-dresden.de

Cover illustration: The Tower of Babel, by Pieter Bruegel

ISBN: 000-11-22222-33-4
NUR: 000

Summary

English summary

Zusammenfassung

Deutsche Kurzzusammenfassung

Contents

Preface vii

Introduction 1

1 AMDiS tutorial **3**
- 1.1 Scalar linear second order PDEs 3
- 1.2 Handling data on unstructured grids 4
- 1.3 Adaptivity and systems of equations 4
- 1.4 Time-dependent and nonlinear problems 4

2 Parallelization **5**

3 Multi-Grid and Multi-Mesh **7**

4 Working with Meshes **9**

5 Expression templates **11**
- 5.1 Motivation 11
- 5.2 Syntaxbeschreibung 12
  - 5.2.1 Header 12
  - 5.2.2 Elementare Ausdrcke 12
  - 5.2.3 Operationen 13
  - 5.2.4 Vektor-/Matrix-Ausdrcke 13
  - 5.2.5 Differenzieren von Ausdrcken 13
  - 5.2.6 Anwendung 14
- 5.3 Beispiele 15
  - 5.3.1 Bilinearform 15
  - 5.3.2 Integration und Interpolation 16
  - 5.3.3 Funktoren 16
  - 5.3.4 Differentiation 18

6 Applications **19**

7 Conclusion and outlook **21**

Preface

Introduction

CHAPTER 1

AMDiS tutorial

Abstract

1.1 Scalar linear second order PDEs

$$[\partial_t u] + cu + \underline{b} \cdot \nabla u - \nabla \cdot (\mathbb{A} \nabla u) = f, \qquad \text{in } \Omega \times (0, T]$$

with $u = u(x)$ the unknown function and coefficients $c, \underline{b}, \mathbb{A}$ that can depend on space, time, and other quantities v living in Ω:

$$\begin{aligned} c &= c(t, x, v, \nabla v), \\ \underline{b} &= \underline{b}(t, x, v, \nabla v) \in \mathbb{R}^d, \\ \mathbb{A} &= \mathbb{A}(t, x, v, \nabla v) \in \mathbb{R}^{d \times d}, \\ f &= f(t, x, v, \nabla v), \end{aligned}$$

equipped with (initial- and) boundary conditions on $\partial\Omega$.

```
#include "AMDiS.h"
using namespace AMDiS;

int main(int argc, char** argv)
{
  AMDiS::init(argc, argv);

  // create and init the scalar problem
  ProblemStat prob("poisson");
  prob.initialize(INIT_ALL);
```

```
11
12    // define operators
13    Operator opLaplace(prob.getFeSpace(), prob.getFeSpace());
14      addSOT(opLaplace, 1.0); // <grad(u), grad(theta)>
15    Operator opF(prob.getFeSpace());
16      addZOT(opF, 1.0); // <f(x), theta>
17
18    // add operators to problem
19    prob.addMatrixOperator(opLaplace, 0, 0);
20    prob.addVectorOperator(opF, 0);
21
22    // add boundary conditions
23    BoundaryType nr = 1;
24    prob.addDirichletBC(nr, 0, 0, new Constant(0.0));
25
26    AdaptInfo adaptInfo("adapt");
27
28    // assemble and solve linear system
29    prob.assemble(&adaptInfo);
30    prob.solve(&adaptInfo);
31
32    prob.writeFiles(&adaptInfo, true);
33
34    AMDiS::finalize();
35  }
```

1.2 Handling data on unstructured grids

1.3 Adaptivity and systems of equations

1.4 Time-dependent and nonlinear problems

CHAPTER 2

Parallelization

Abstract

CHAPTER 3

Multi-Grid and Multi-Mesh

Abstract

CHAPTER 4

Working with Meshes

Abstract

CHAPTER 5

Expression templates

Abstract

5.1 Motivation

Beim Umsetzen einer Differentialgleichung ist der Standardweg in AMDiS diese zu zerlegen in einzelne Terme, zu klassifizieren als 0.-Orderung, 1.-Ordnung oder 2.-Ordnungs Term und dann alle in Operatoren zu packen, die wiederum dem zu lsenden Problem zugewiesen werden. Beispielsweise findet man folgenden Ausdruck im Navier-Stokes Beispiel:

```
1 DOFVector<double>* U = prob->getSolution(j);
2 Operator opUGradU2(getFeSpace(i), getFeSpace(i));
3   opUGradU2.addTerm(new Vec2AndPartialDerivative_FOT(densityPhase, U, j),
  ↪  GRD_PHI);
4 prob->addMatrixOperator(opUGradU2, i, i);
```

Was genau dieser Operator nun implementiert lsst sich am Namen der Terme nur noch erahnen. Als Ausweg daraus und aus der Notwendigkeit fr immer neue Termkonstallationen habe ich folgende Variante implementiert, die die Sytax zumindest einer Zeile etwas aufrumt:

```
1 DOFVector<double>* U = prob->getSolution(j);
2 Operator opUGradU2(getFeSpace(i), getFeSpace(i));
3   addFOT(opUGradU2, valueOf(densityPhase) * valueOf(U), j, GRD_PHI);
4 prob->addMatrixOperator(opUGradU2, i, i);
```

Dies ist zwar nur unwesentlich krzer, gibt dafr direkte Auskunft darber, wie sich der Koeffizient des Terms zusammensetzt, nmlich als Produkt von 2 DOFVectoren (densityPhase und `solution[j]`). Die Syntax lsst sich aber nicht nur anwenden beim Fllen von Operatoren, sondern auch fr die Integration ber Ausdrcke von DOFVectoren und bei der Transformation von DOFVectoren. Dazu aber spter mehr. Nachfolgend ist die neue Syntax aufgelistet.

5.2 Syntaxbeschreibung

5.2.1 Header

Um die neue Sytax nutzen zu knnen muss folgende Header-datei eingebunden werden:

```
#include "Expressions.h"
```

5.2.2 Elementare Ausdrcke

In der Folgenden Tabelle ist DV ein Pointer/Reference auf `DOFVector<T>` mit beliebigem `T`:

Expression	Semantics
`valueOf(DV)`	Auswertung des DOFVectors an Quadraturpunkten
`gradientOf(DV)`	Auswertung des Gradienten eines DOFVectors an Quadraturpunkten
`derivativeOf(DV, i)`	Auswertung der partiellen Ableitung des DOFVectors nach der i-ten Koordinate an Quadraturpunkten
`hessianOf(DV)`	Auswertung der Hesse-Matrix des DOFVectors: $\nabla^2 DV = H_{ij} = \partial_i(\partial_j(DV))$ (kann nur in `transformDOF`, bzw. << verwendet werden)
`laplacianOf(DV)`	Auswertung des Laplace des DOFVectors: $\Delta DV = \sum_i \partial_i(\partial_i(DV)))$ (kann nur in `transformDOF`, bzw. << verwendet werden)
`componentOf(DV, i)`	Auswertung der i-ten Komponente eines DOFVectors an Quadraturpunkten. Dies macht nur Sinn fr `DOFVector<Vector<T> >` Objekten.
`X(), X<i>()`	Der Koordinatenvektor, bzw. die i-te Komponente des Koordinatenvektors an den QP.
`N(b), N(b,i)`	Der uere Normalenvektor zurgrig zur Flche mit dem Index b, bzw. deren i-te Komponente.
`M(), M(i)`	Der Elementnormalenvektor (bei Oberflchengittern), bzw. deren i-te Komponente
`constant(c), c`	Konstanten (numeric values) sind z.B. double, float, int. Entweder eingeschlossen in `constant()`, oder direkt
`constant<C>()`	(Compiletime) Konstanten sind Integer, die beim Differenzierten von Ausdrcken verwendet werden knnen (s.u.).
`var(c)`	Eine Referenz oder ein Pointer auf einen Wert, der sich dynamisch verndern kann.

Der Ausdruck `valueOf(DV)` kann auerdem auf Vektoren und Matrizen von DOFVectoren angewendet werden:

Expression	Semantics
`valueOf(DV)`	Auswertung des DOFVectors an Quadraturpunkten
`valueOf(vector<DV<T>>)`	Ergibt an den Quadraturpunkten einen `vector<T>` mit den Werten der einzelnen DOFVektoren and dieser Stelle.
`valueOf(matrix<DV<T>>)`	Ergibt an den Quadraturpunkten eine `matrix<T>` mit den Werten der einzelnen DOFVektoren and dieser Stelle.
`valueOf<Name>(X)`	Eine Expression, der ein Name zugeordnet ist. `Name` ist dabei ein C++-Typ. Ist kein Name explizit angegeben, wird `_unknown` verwendet. (siehe auch Differenzieren von Ausdrcken)

5.2.3 Operationen

In der folgenden Tabelle sind Operationen aufgelistet, ber die die elementaren Term (s.o.) kombiniert werden knnen. Dabei steht `T`, bzw. `Tn` fr einen Term und `F` fr ein Funktor-Objekt (siehe unten).

Expression	Semantics
`+, -, *, /`	Elementare arithmetische Ausdrcke
`pow<p>(T)`	Die p-te Potenz eines Term, wenn Term skalar ist.
`sqrt(T)`	Die Wurzel des Ausdrucks `T`
`exp(T)`	Die Exponentialfunktion angewendet auf das Ergebnis der Auswertung des Terms `T`.
`log(T)`	Der natrlich Logarithmus der Auswertung des Terms `T`.
`cos(T), sin(T), tan(T)`	Der Cosinus, Sinus, bzw. Tangens der Auswertung des Terms `T`.
`acos(T), asin(T), atan(T)`	Der Arkus-Cosinus, Arkus-Sinus, bzw. Arkus Tangens der Auswertung des Terms `T`.
`atan2(T1, T2)`	Der Arkus-Tangens 2 = `atan(T1 / T2)`.
`cosh(T), sinh(T), tanh(T)`	Der Cosinus-Hyperbolicus, Sinus-Hyperbolicus, bzw. Tangens-Hyperbolicus der Auswertung des Terms `T`.
`acosh(T), asinh(T), atanh(T)`	Der Arkus Cosinus-Hyperbolicus, Arkus Sinus-Hyperbolicus, Arkus Tangens-Hyperbolicus der Auswertung des Terms `T`.
`max(T1,T2), min(T1,T2)`	Das Maximum/Minimum der Auswertungen von `T1`, bzw. `T2`.
`absolute(T), signum(T)`	Der Absolutbetrag von T, bzw. das Vorzeichen (`if(T < 0) signum(T) = -1 else if(T > 0) signum(T) = 1 else signum(T) = 0`)
`ceil(T), floor(T)`	die kleines ganze Zahl grer, bzw. die grte ganze Zahl kleiner als Wert von `T`
`func(F, T1, T2,...)`	Die Anwendung eines Funktors `F` auf die Auswertung der Terme `T1,T2,...`

Ein Funktor ist dabei eine Klasse mit folgender Struktur:

```
struct Functor : public FunctorBase
{
  typedef (...) value_type;
  int getDegree(int d1, int d2, ...) const { return (...); }

  value_type operator()(const T1::value_type&, const T2::value_type&, ...)
  ↪  const { return (...); }
};
```

Die Argumente, die an die Funktion `getDegree` bergeben werden sind die Polynomgrade der Terme:

```
getDegree(T1.getDegree(), T2.getDegree(), ...)
```

5.2.4 Vektor-/Matrix-Ausdrcke

Fr vektor- bzw. matrixwertige Term, wie z.B. `gradientOf(DV)` gibt es eine Reihe von Ausdrcken / Operationen, die im Folgenden aufgelistet sind. Dabei bezeichnet `V` eine vektorwerte Expression und `M` eine matrixwertige Expressions.

Expression	Semantics
`+, -`	Elementare arithmetische Ausdrcke (Elementweise)
`unary_dot(V)`	Skalarprodukt eines vektorwertigen Terms mit sich selbst: `result = V^H * V`.
`dot(V1, V2)`	Skalarprodukt zweier vektorwertigen Terme: `result = V1^H * V2`.
`one_norm(V)`	Die 1-Norm eines Vektors `result = sum_i(abs(V_i))`.
`one_norm(M)`	Die 1-Norm einer Matrix `result = max_j(sum_i(abs(M_ij)))`.
`two_norm(V)`	Die 2-Norm eines Vektors `result = sqrt(V^H * V)`.
`p_norm<p>(V)`	Die p-Norm eines Vektors `result = [sum_i(abs(v_i)^p)]^(1/p)`.
`cross(V1, V2)`	Kreuzprodukt zweier vektorwertigen Terme: `result = V1 x V2`.
`diagonal(V)`	Diagonalmatrix aus eintrgen eines vektorwertigen Terms: `matrix = diagonal(V)`.
`outer(V1, V2)`	ueres Produkt (dyadisches Produkt / Tensorprodukt) zweier vektorwertigen Terme: `matrix = V1 * V2^T`.
`trans(M)`	Das Transponierte eines matrixwertigen Terms: `result = M^T`.
`at(V, i)`	Zugriff auf eine Komponente eines Vektors: `result = V_i`.
`at(M, i, j)`	Zugriff auf eine Komponente einer Matrix: `result = M_ij`.

5.2.5 Differenzieren von Ausdrcken

Terme / Expressions sind Ausdrcke mit Variablen und Operationssymbolen, die Formal nach einzelnen Variablen differenziert werden knnen. Variablen sind hierbei Ausdrcke `valueOf<Name>(DV)` denen ein Variablenname zugeordnet ist. Jedem `valueOf` Ausdruck ist standardmig der Name `_unknown` zugeordnet. Fr skalare Terme (also Terme ohne Vektorausdrcke und vektorwertige Variablen) lsst sich die Differentiation automatisch durchfhren.

Um dies nutzen zu knnen, muss die Datei `diff_expr.hpp` eingebunden werden:

```
#include "expressions/diff_expr.hpp"
```

Expression	Semantics
`diff<Name>(T)`	Differentiation nach Variable "Name": `d_Name (Term)`
`diff<Name>(T, U)`	Richtungsableitung nach Variable "Name", in Richtung `U`: `d_Name (Term)[U]`
`simplify(T)`	Vereinfachen eines Terms. Insbesondere werden Multiplikationen mit 0 und mit 1 entfernt und verschachtelte Operationen zwischen Konstanten zusammengefasst.

5.2.6 Anwendung

Diese verallgemeinerten Terme knnen nun beim Assemblieren, integrieren und transformieren von DOFVectoren verwendet werden: In der Folgenden Tabelle wird `Operator` fr einen Pointer/Reference auf ein Operator Objekt verwendet und Term steht fr einen Ausdruck, der sich aus den obigen Zutaten zusammensetzt. `Flag` ist entweder `GRD_PHI`, oder `GRD_PSI`, je nachdem, ob die Ableitung bei einem 1.-Ordnungs Term auf der Ansatzfunktion `u`, oder Testfunktion `v` steht.

Expression	Semantics
`addZOT(Operator, Term)`	Ein 0.-Ordnungs Term $(\text{Term} \cdot u,\ v)$
`addFOT(Operator, Term, Flag)`	Ein 1.-Ordnungs Term $(\text{Term} \cdot \nabla u,\ v)$, bzw. $(\text{Term} \cdot u,\ \nabla v)$, wenn der `value_type` von `Term` einem `WorldVector<double>` entspricht, bzw. $(\mathbf{1}\text{Term} \cdot \nabla u,\ v)$, bzw. $(\mathbf{1}\text{Term} \cdot u,\ \nabla v)$, mit $\mathbf{1} = (1,1,1)^\top$, wenn der `value_type` von `Term` einem `double` entspricht.
`addFOT(Operator, Term, i, Flag)`	Ein 1.-Ordnungs Term $(\text{Term} \cdot \partial_i u,\ v)$, bzw. $(\text{Term} \cdot u,\ \partial_i v)$
`addSOT(Operator, Term)`	Ein 2.-Ordnungs Term $(\text{Term} \cdot \nabla u,\ \nabla v)$, wobei `value_type` von `Term` einem `WorldMatrix<double>`, bzw. `double` entsprechen muss.
`addSOT(Operator, Term, i, j)`	Ein 2.-Ordnungs Term $(\text{Term} \cdot \partial_j u,\ \partial_i v)$, wobei `value_type` von `Term` einem `double` entsprechen muss.

Fr die Berechnung eines Integrals ber einen Ausdruck von DOFVectoren kann folgender Befehl verwendet werden: `integrate(Term);` und fr die Transformation eines DOFVectors folgen-

des: `DV << Term;` bzw. `transformDOF(Term, DV);` Wobei jeweils DV kein DOFVector-Pointer sein kann.

5.3 Beispiele

5.3.1 Bilinearform

Folgende Bilinearform soll assembliert werden:

$$a(c,\vartheta) := \left\langle \frac{1}{\epsilon}(\phi^2 - 1)c,\ \vartheta \right\rangle + \left\langle \max(10^{-5},\ (\phi + 1))\nabla c,\ \nabla\vartheta \right\rangle$$

Mit Hilfe der oben definierten Ausdrcke lsst sich dies nun wie folgt schreiben:

```
Operator op(feSpace, feSpace);
addZOT(op, (1.0/epsilon) * (pow<2>(valueOf(phi)) - 1.0) );
addSOT(op, max(1.e-5, valueOf(phi) + 1.0) )
prob->addMatrixOperator(op, 0, 0);
```

5.3.2 Integration und Interpolation

Danach sollen folgende Integrale und ein Double-well berechnet werden:

$$mean = \int \frac{1}{2}(c+1)\ dx, \quad length = \int \|\nabla c\|\ dx, \quad DW = \frac{1}{2}(c^2\,(1-c)^2 + \frac{1}{\epsilon}\|\nabla c\|^2)$$

Dazu knnen wieder die Expressions verwendet werden:

```
double mean = integrate( 0.5 * (valueOf(c) + 1.0) );
double length = integrate( two_norm(gradientOf(c)) );

DOFVector<double> DW(c.getFeSpace(), "Double-Well");
DW << 0.5*( pow<2>(valueOf(c)) * pow<2>(1.0 - valueOf(c)) +
 ↪  unary_dot(gradientOf(c)) );
```

5.3.3 Funktoren

Komplexe Ausdrcken knnen auch zu einem Funktor zusammen gefasst werden. Dazu wird erst das Funktor-Objekt erstellt und dieses dann ber den Term `func` in einem Expression eingebunden:

$$V = \sum_{i=0}^{d-1} \alpha_i U_i$$

Mit α_i sind Koeffizienten und $U = \{U_i\}$ ein DOFVector¡WorldVector¡double¿, ¿. Dieser Term kann umgesetzt werden, mittels

```
struct F : FunctorBase
{
  typedef double value_type;
  F(WorldVector<double>& alpha_) : alpha(alpha_) {}

  int getDegree(int d0) const {
    return d0;
  }
  value_type operator()(const WorldVector<double>& U) const {
    double result = 0.0;
    for (int i = 0; i < U.getSize(); i++)
      result += alpha[i]*U[i];
    return result;
  }
private:
  WorldVector<double> alpha;
};

int main() {
  // ... //
  DOFVector<WorldVector<double> > U(feSpace, "U");
  DOFVector<double> V(feSpace, "V");

  WorldVector<double> alpha = (...);
  U << (...)

  V << func(F(alpha), valueOf(U));
}
```

Bisher wird in AMDiS anstelle von Objekten zur Basis `FunctorBase` die AbstractFunction verwendet. Um diese nutzen zu knnen, gibt es Wrapper, die diese implizit zu einem Funktor Konvertieren. Fr den Spezialfall der Auswertung an Koordinaten gibt es zustzlich einen Shortcut:

```
struct F : AbstractFunction<double, WorldVector<double> >
{
  F() : AbstractFunction<double, WorldVector<double> >(1) {}

  double operator()(const WorldVector<double>& x) const {
    return cos(x[0]);
  }
};

int main() {
```

```
  // ... //
  DOFVector<double> V(feSpace, "V");

  V << function_(wrap(new F), X());
  // bzw.
  V << eval(new F)

  // und auch verschobene Auswertungen sind mglich:
  WorldVector<double> shift = (...)
  V << function_(wrap(new F), X() + shift);
}
```

5.3.4 Differentiation

Insbesondere in Newtonverfahren fr nichtlineare PDEs und Rosenbrock-Verfahren fr die Zeitintegration (nichtlinearer Gleichungen), wird die Jacobimatrix eines Differentialoperators bentigt. Den kann man hufig auf die Ableitung der Koeffizientefunktionen zurckfhren. Siehe aus Motivation auch das Nonlinear Demo [[AMDiS:Demo:Nonlinear example]]. Grundlage der Implementierung knnte sein, dass die Terme automatisch differenziert werden Als Beispiel soll hier ein einfaches Polynom als Term dienen:

$$expr(u) = u^4$$

Um nach u zu differenzieren, mssen wir dem zugrundeliegenden DOFVector einen Namen geben:

```
struct u_ {};
struct v_ {};
...
DOFVector<double> U(...);
DOFVector<double> V(...);
auto expr = pow<4>(valueOf<u_>(U));

auto diff_expr = diff<u_>(expr);  // = 4 * u^3
auto diff_expr2 = diff<u_>(expr, valueOf<v_>(V)); // 4 * u^3*v
```

CHAPTER 6

Applications

Abstract

CHAPTER 7

Conclusion and outlook