Numerische Betrachtungen und Realisierung mit DUNE

Daniel Seibel

Universität des Saarlandes

3. Januar 2018

Splitting-Methode

Finite-Element-Methode
- Theoretische Einführung
- Implementierung mit DUNE

Finite-Volumen-Methode
- Theoretische Einführung
- Implementierung mit DUNE

Beispiel Vlassov-Poisson Gleichung

Die nichtrelativistische Vlassov-Poisson Gleichung

$$\frac{\partial f(t,x,v)}{\partial t} + v \cdot \nabla_x f(t,x,v) - E(t,x) \cdot \nabla_v f(t,x,v) = 0,$$
$$-\Delta_x \phi(t,x) = \rho(t,x) = 1 - \int_{\mathbb{R}^3} f(t,x,v)\, dv,$$
$$E(t,x) = -\nabla_x \phi(t,x),$$

wird in zwei Transportgleichungen “aufgesplittet”.

Für festes v betrachte man

$$\frac{\partial f}{\partial t} + v \cdot \nabla_x f = 0,$$

und für festes x

$$\frac{\partial f}{\partial t} + E \cdot \nabla_v f = 0.$$

Beide Transportgleichungen sind linear mit konstanten Koeffizienten.

Godunov-Splitting

Die beiden Transportgleichungen werden pro Zeitschritt abwechselnd gelöst.

- Löse zuerst die erste Gleichung für alle v.
- Löse anschließend mit aktualisierten Anfangsbedingungen die zweite Gleichung für alle x.

Dabei muss im Vorfeld E aus dem Poisson-Problem für das Potential berechnet werden.

Fehlerabschätzung

Betrachte abstrakt die Differentialgleichung

$$\frac{\partial u(t,x)}{\partial t} = (A+B)u(t,x),$$

und die gesplitteten Gleichungen

$$\frac{\partial u^*(t,x)}{\partial t} = Au^*(t,x),$$
$$\frac{\partial u^{**}(t,x)}{\partial t} = Bu^{**}(t,x).$$

Die Lösungen der Gleichungen in $t + \Delta t$ lauten dann

$$u(t + \Delta t) = e^{\Delta t(A+B)} u(t).$$

bzw.

$$u^*(t + \Delta t) = e^{\Delta t A} u^*(t), \quad u^{**}(t + \Delta t) = e^{\Delta t B} u^{**}(t).$$

Godunov-Splitting liefert die Lösung

$$\tilde{u}(t + \Delta t) = e^{\Delta t B} u^*(t + \Delta t) = e^{\Delta t B} e^{\Delta t A} u(t).$$

Ein Vergleich der Reihenentwicklungen zeigt

$$e^{\Delta t(A+B)} - e^{\Delta tB}e^{\Delta tA} = \frac{\Delta t^2}{2}(AB - BA) + \mathcal{O}(\Delta t^3),$$

d.h., falls die Operatoren nicht kommutieren, ist der Fehler von Ordnung 1 in der Zeit.

Numerische Behandlung

Eine Möglichkeit, die Splitting-Methode numerisch zu realisieren, ist, in jedem Zeitschritt t

- das elliptische Problem $-\Delta_x \phi(t,x) = \rho(t,x)$ mit FEM zu lösen und
- die zwei Transportgleichungen

$$\frac{\partial f}{\partial t} + v \cdot \nabla_x f = 0, \quad \frac{\partial f}{\partial t} + E \cdot \nabla_v f = 0$$

  mit FVM zu behandeln.

Im Folgenden werden beide Verfahren zusammen mit einer Implementierung in DUNE vorgestellt.

Galerkin-Formulierung

Wir betrachten als Beispiel das Poisson-Problem

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{auf } \Gamma. \end{aligned}$$

Die FEM basiert auf einer schwachen Formulierung, der Galerkin-Formulierung. Betrachten wir dazu den Raum

$$V = \left\{ v \in H^1(\Omega) \mid v = 0 \text{ auf } \Gamma \right\}.$$

Mit der ersten Greenschen Formel erhalten wir für $u, v \in V$:

$$\int_\Omega -\Delta u \, v \, dx = \int_\Omega \nabla u \cdot \nabla v \, dx,$$

Seien für $u, v \in V$ und $f \in L^2(\Omega)$

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega -\Delta u \, v \, dx,$$

und

$$\varphi(v) = \int_\Omega f \, v \, dx \, .$$

Damit lautet die Galerkin-Formulierung des Problems
Finde $u \in V$, sodass

$$a(u,v) = \varphi(v), \quad v \in V,$$

gilt.

Galerkin-Approximation

Für die FEM schränken wir die Galerkin-Formulierung auf einen endlich dimensionalen Unterraum $V_h \subset V$ ein. Diese sogenannte Galerkin-Approximation lautet:
Finde $u_h \in V_h$, sodass

$$a(u, v) = \varphi(v), \quad v \in V_h,$$

gilt.

Konstruktion Finiter Elemente

Die Idee der FEM ist es, den Raum V_h bezüglich einer Zerlegung von Ω in Dreiecke K_i zu wählen. Dabei sind die Dreiecke K_i Teil eines Tupels $(K_i, \mathcal{P}_i, \mathcal{N}_i)$, genannt Finites Element, mit den Eigenschaften:

- $K_i \subset \mathbb{R}^2$ ist abgeschlossen und beschänkt mit nichtleerem Inneren und stückweise glattem Rand.
- $\mathcal{P}_i$ ist ein endlich dimensionaler Vektorraum von Funktionen auf K_i.
- $\mathcal{N}_i = \{N_{i,1}, N_{i,2}, \ldots, N_{i,k}\}$ ist eine Basis von $\mathcal{P}_i'$.

Beispiel Lagrange Element

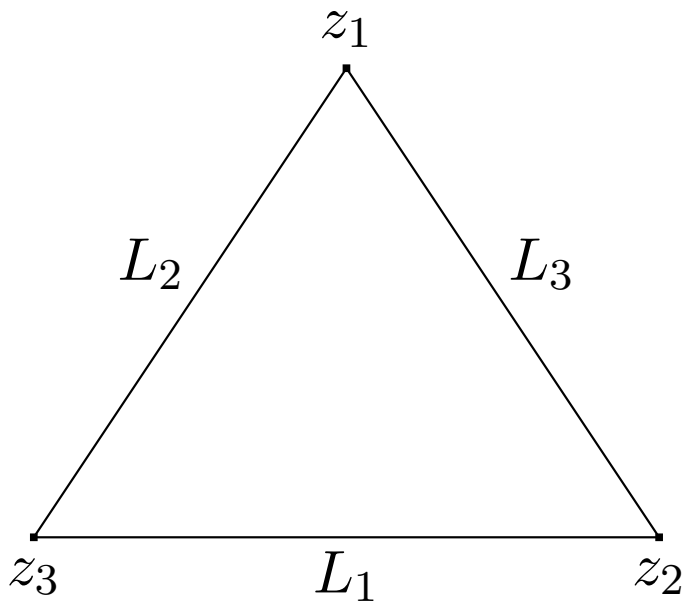


Abbildung: *Lagrange-Element für lineare Ansatzfunktionen*

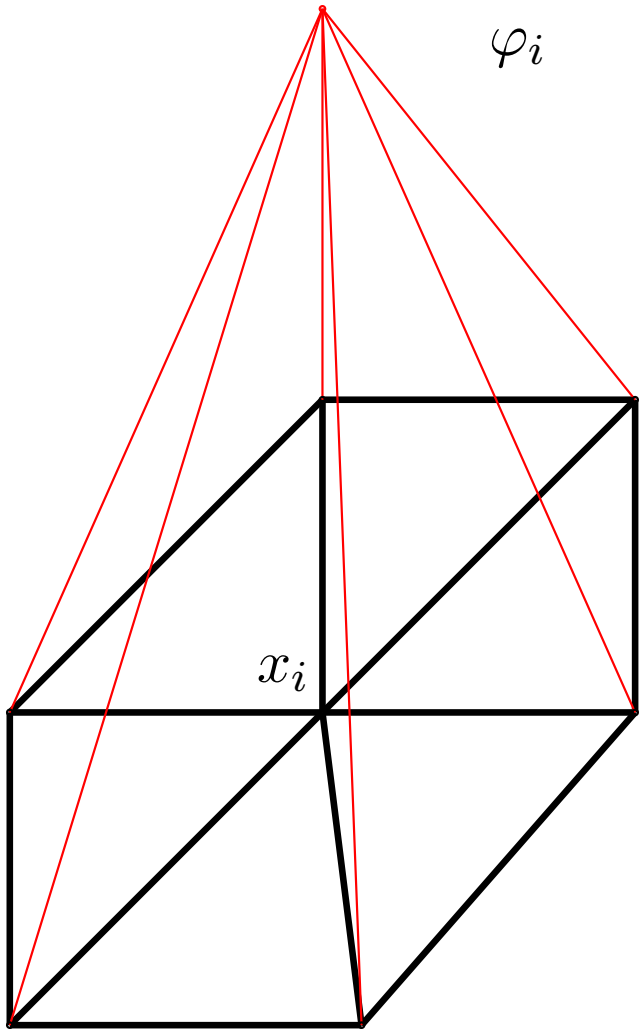


Abbildung: *Lineare Ansatzfunktion φ_i im Knoten x_i*

Interpolationsoperator

Zu jedem finiten Element $(K, \mathcal{P}, \mathcal{N})$ betrachten wir den lokalen Interpolationsoperator

$$\mathcal{I}_K v = \sum_{j=1}^{k} N_j(v)\varphi_j,$$

wobei $\{\varphi_j\}$ eine duale Basis zu $\mathcal{N}$ ist.
Den globale Interpolationsoperator definieren wir durch

$$(\mathcal{I}^h v)_{|K_i} = \mathcal{I}_{K_i} v, \quad i \in I.$$

Beispiel für Interpolation

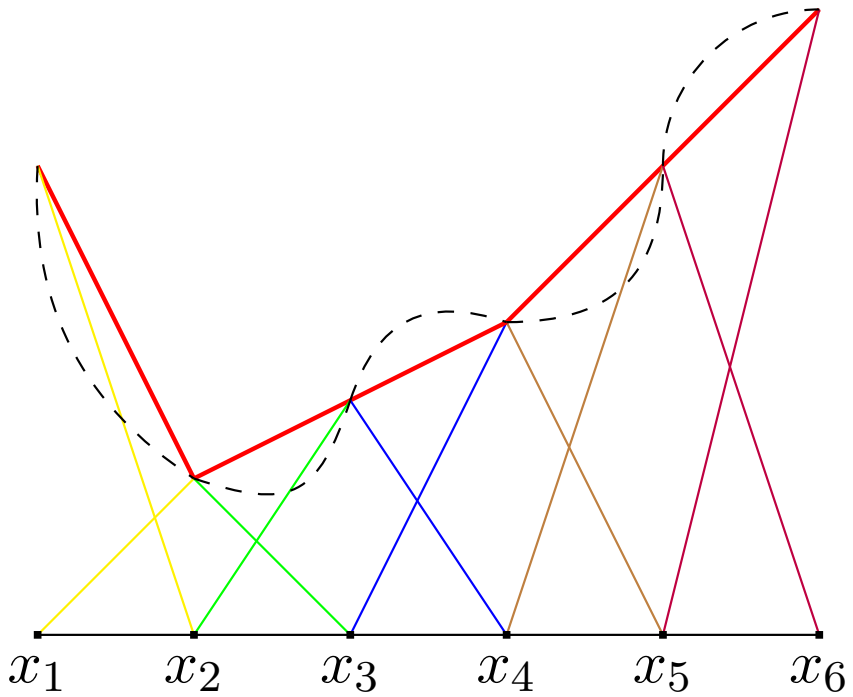


Abbildung: *Interpolation durch lineare Ansatzfunktionen*

Fehlerabschätzung

Die Formfunktionen $\mathcal{P}_i$ können so gewählt werden, dass sie die Approximationsordnung m haben, d.h.

$$\|v - \mathcal{I}^h v\|_{H^1(\Omega)} \leq C h^{m-1} |v|_{H^m(\Omega)}, \quad \forall v \in V \cap C^k(\Omega).$$

Wählt man nun den Unterraum V_h als

$$V_h = \mathcal{I}^h(V \cap C^k(\Omega)) \cap C(\Omega),$$

so folgt aus Céas Lemma direkt

$$\|u - u_h\|_{H^1(\Omega)} \leq C h^{m-1} |u|_{H^m(\Omega)}.$$

Ist $\{\psi_j\}_{j=1}^N$ eine Basis von V_h, so lässt sich die Galerkin-Approximation als lineares Gleichungssystem

$$Au_h = f_h$$

schreiben. Dabei ist die Matrix $A \in \mathbb{R}^{N \times N}$ durch

$$A_{jk} = a(\psi_j, \psi_k) = \int_\Omega \nabla\psi_j \cdot \nabla\psi_k \, dx,$$

und die rechte Seite $f_h \in \mathbb{R}^N$ durch

$$(f_h)_j = \varphi(\psi_j) = \int_\Omega f\psi_j \, dx$$

gegeben.

Überblick

DUNE, kurz für *Distributed and Unified Numerics Environment*, ist ein in C++ geschriebener, modularer Baukasten zum Lösen von partiellen Differentialgleichungen mit gitterbasierten Methoden.
Er besteht aus

- *core modules*,
- *grid modules* und
- *discretization modules*.

Der Fokus dieses Vortrags liegt auf dem discretization module *dune-fem*.

Das Programm

Im Kern besteht das Programm aus sechs Teilen, nämlich

- der Klasse **Probleminterface**,
- der Klasse **Model**,
- der Klasse **EllipticOperator**,
- der Funktion **assembleRHS()**,
- der Klasse **FemScheme** und
- den Funktionen **main()** und **algorithm()**.

ProblemInterface und Model

Die Klassen ProblemInterface und Model beschreiben das Problem, d.h., sie enthalten die Koeffizienten und die Randbedingungen des allgemeineren Problems

$$\begin{aligned} -\mathrm{div}(D(x) \cdot \nabla u(x)) + m(x)u(x) &= f(x), \quad x \in \Omega, \\ u(x) &= g(x), \quad x \in \Gamma_D, \\ D(x)\nabla u(x) \cdot \nu + \alpha(x)u(x) &= t(x), \quad x \in \Gamma_N. \end{aligned}$$

Beispiel für Model

```
1  template< class FunctionSpace, class GridPart >
2  struct DiffusionModel
3  {
4    template< class Entity, class Point >
5    void diffusiveFlux ( const Entity &entity,
6                         const Point &x,
7                         const RangeType &value,
8                         const JacobianRangeType &
       gradient,
9                         JacobianRangeType &flux ) const
10   {
11     flux = gradient;
12   }
```

EllipticOperator

Die Klasse EllipticOperator kümmert sich um den Aufbau der Matrix A, d.h., sie berechnet die Einträge

$$A_{jk} = \int_\Omega \nabla\psi_j \cdot \nabla\psi_k \, dx = \sum_{i \in I} \int_{K_i} \nabla\psi_j \cdot \nabla\psi_k \, dx \, .$$

Dazu betrachtet man lokale Matrizen A^i,$i \in I$, die den Beitrag auf dem Element K_i umfassen.

Referenzelement

Es ist üblich, die Elemente K_i mit einem Referenzelement K vermöge einer Abbildung $F_i : K \to K_i$ zu identifizieren.
Somit folgt

$$\begin{aligned} A^i_{jk} &= \int_{K_i} \nabla\psi_j \cdot \nabla\psi_k \, dx \\ &= \int_K |\det DF_i| \; (\nabla\psi_j \cdot \nabla\psi_k) \circ F_i \, d\tilde{x} \\ &\approx |K| \sum_{\alpha=1}^{q} w_\alpha |\det DF_i(\tilde{x}_\alpha)| \; (\nabla\psi_j^K \cdot \nabla\psi_k^K)(\tilde{x}_\alpha), \end{aligned}$$

für eine geeignete Quadratur und Basisfunktionen $\left\{\psi_j^K\right\}$ auf K.

Beispiel für Berechnung der Einträge

```
1  for( IteratorType it = rangeSpace.begin(); it !=
       rangeSpace.end(); ++it )
2  {
3    const EntityType &entity = *it;
4    const GeometryType &geometry = entity.geometry();
5
6    LocalMatrixType jLocal = jOp.localMatrix( entity,
       entity );
7
8    QuadratureType quadrature( entity, order );
9    for( size_t pt = 0; pt < quadrature.nop(); ++pt )
10   {
11     const double weight = quadrature.weight( pt ) *
       geometry.integrationElement( quadrature.point( pt
       ) );
```

Berechnung der Matrixeinträge

Die Matrix A^i wird spaltenweise aufgebaut, d.h., in

$$A^i_{jk} = |K| \sum_{\alpha=1}^{q} w_\alpha |\det DF_i(\tilde{x}_\alpha)| \ (\nabla \psi_j^K \cdot \nabla \psi_k^K)(\tilde{x}_\alpha),$$

werden für festes k und alle j die Einträge simultan berechnet. Durch Caching werden die Basisfunktionen ψ_j^K nur einmalig in den Quadraturpunkten ausgewertet.

Beispiel für Berechnung der Einträge

```
1 domainBaseSet.jacobianAll( quadrature[ pt ], dphi );
2
3 RangeJacobianRangeType adphi( 0 );
4 for( unsigned int localCol = 0; localCol <
      domainNumBasisFunctions; ++localCol )
5 {
6   model().DiffusiveFlux( entity, quadrature[ pt ],
      dphi[ localCol ], adphi );
7
8    jLocal.column( localCol ).axpy( dphi, adphi, weight
       );
9 }
```

Das restliche Programm

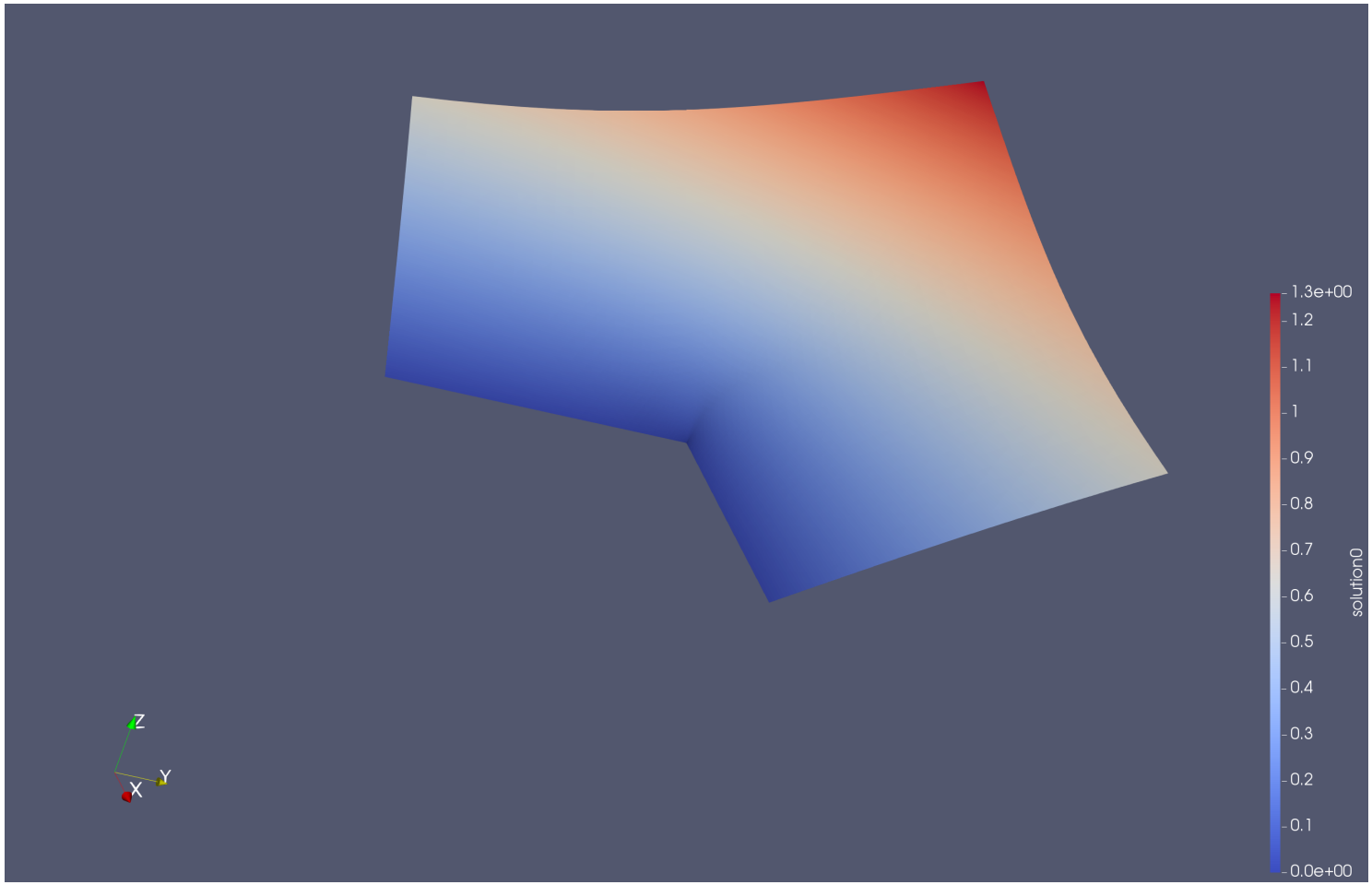
- Der Aufbau der rechten Seite f_h in assembleRHS() verläuft analog zur Berechnung der Matrix A.
- Die Klasse FemScheme wählt den Raum V_h und lässt das Gleichungssystem aufstellen und lösen.
- Die Funktionen algorithm() und algorithm erzeugen ein Objekt der Klasse FemScheme für das gewählte Problem und Gitter. Außerdem lesen sie die Parameterdatei und schreiben die Ergebnisse in eine Datei.

Beispiel

Wir betrachten das Poisson-Problem mit rechter Seite $f = 0$ und Dirichletbedingung

$$g(x) = \|x\|_2^{2/3} \sin\left(\frac{2\alpha}{3}\right),$$

wobei $\alpha = \arctan(x_1/x_2)$ ist.


1.3e+00
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0e+00
solution0
Z
Y
X

Bemerkungen

- Die Behandlung von Randbedingungen erfolgt in der Klasse DirichletConstraints.
- Es können verschiedene Löser für das Gleichungssystem verwendet werden.
- Durch leichte Modifikationen kann das Programm auch nichtlineare Probleme lösen.

Problemstellung

Wir betrachten das eindimensionale hyperbolische Problem

$$\frac{\partial q}{\partial t} + \frac{\partial f(q)}{\partial x} = 0,$$

in $\Omega \times \mathbb{R}_{>0}$.

Diskretisierung

Wir zerlegen Ω in Intervalle

$$\mathcal{C}_i = (x_{i-1/2}, x_{i+1/2}) \subset \Omega$$

der Länge Δx und die Zeitachse in Intervalle

$$(t_n, t_{n+1}) \subset \mathbb{R}_{>0}$$

der Länge Δt.

Umformulierung

Wir integrieren die Gleichung über jede Zelle $\mathcal{C}_i$:

$$\frac{\partial}{\partial t}\int_{\mathcal{C}_i} q\,dx = -\int_{\mathcal{C}_i}\frac{\partial f(q)}{\partial x}\,dx = f(q(x_{i-1/2})) - f(q(x_{i+1/2}))$$

und über jedes Zeitintervall (t_n, t_{n+1}):

$$\int_{\mathcal{C}_i} q(x,t_{n+1})\,dx - \int_{\mathcal{C}_i} q(x,t_n)\,dx = \int_{t_n}^{t_{n+1}} f(q(x_{i-1/2},t))\,dt - \int_{t_n}^{t_{n+1}} f(q(x_{i+1/2},t))\,dt$$

Mit

$$\mathcal{Q}_i^n \approx \frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} q(x, t_n)\, dx$$

und

$$F_{i-1/2}^n \approx \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} f(q(x_{i-1/2}, t))\, dt$$

erhalten wir das numerische Schema

$$\mathcal{Q}_i^{n+1} = \mathcal{Q}_i^n - \frac{\Delta t}{\Delta x}(F_{i+1/2}^n - F_{i-1/2}^n).$$

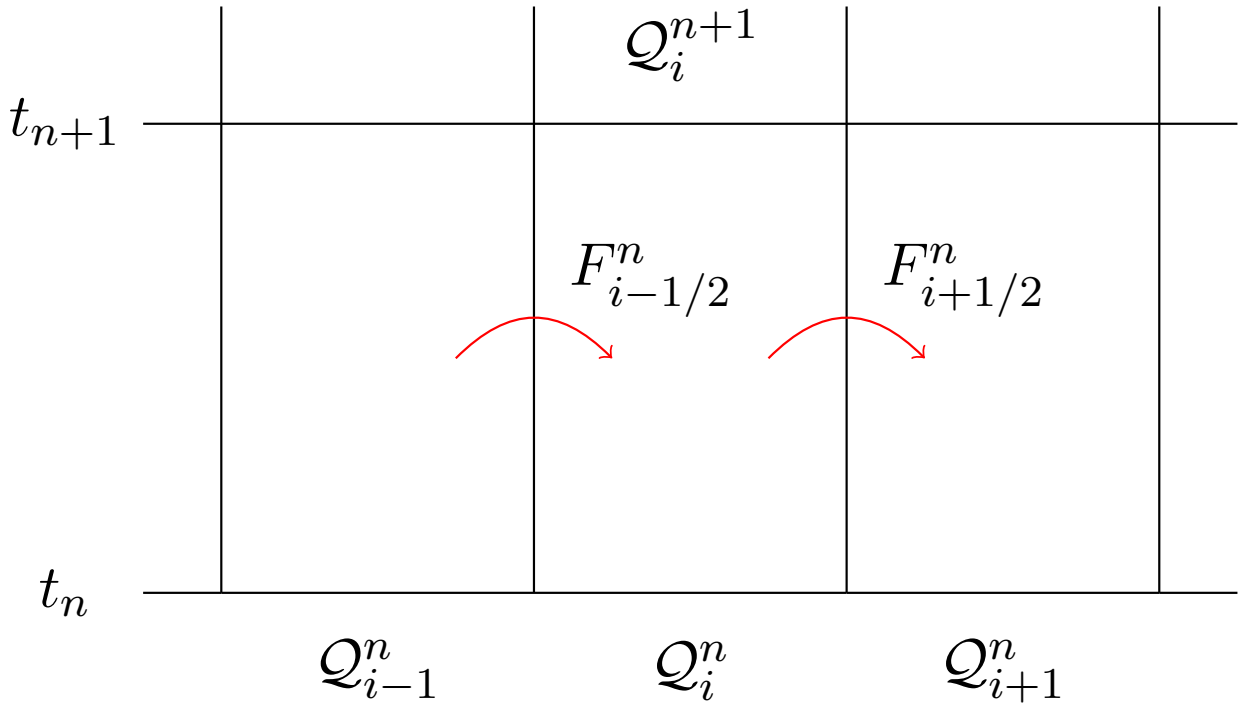


Abbildung: *Motivation für das numerische Schema.*

Godunovs Methode

1. Approximiere q im Zeitschritt t_n durch die stückweise konstante Funktion $\tilde{q}^n$, gegeben durch

$$\tilde{q}^n(x, t_n) = \mathcal{Q}_i^n \quad \text{für } x \in \mathcal{C}_i.$$

2. Berechne die Lösung q^{n+1} des hyperbolischen Problems $q_t + f(q)_x = 0$ mit Anfangsbedingung $\tilde{q}^n$ zur Zeit t_{n+1}.
3. Berechne die neuen Zellmittelwerte

$$\mathcal{Q}_i^{n+1} = \frac{1}{\Delta x} \int_{\mathcal{C}_i} \tilde{q}^n(x, t_{n+1})\, dx$$

und gehe zu Schritt 1.

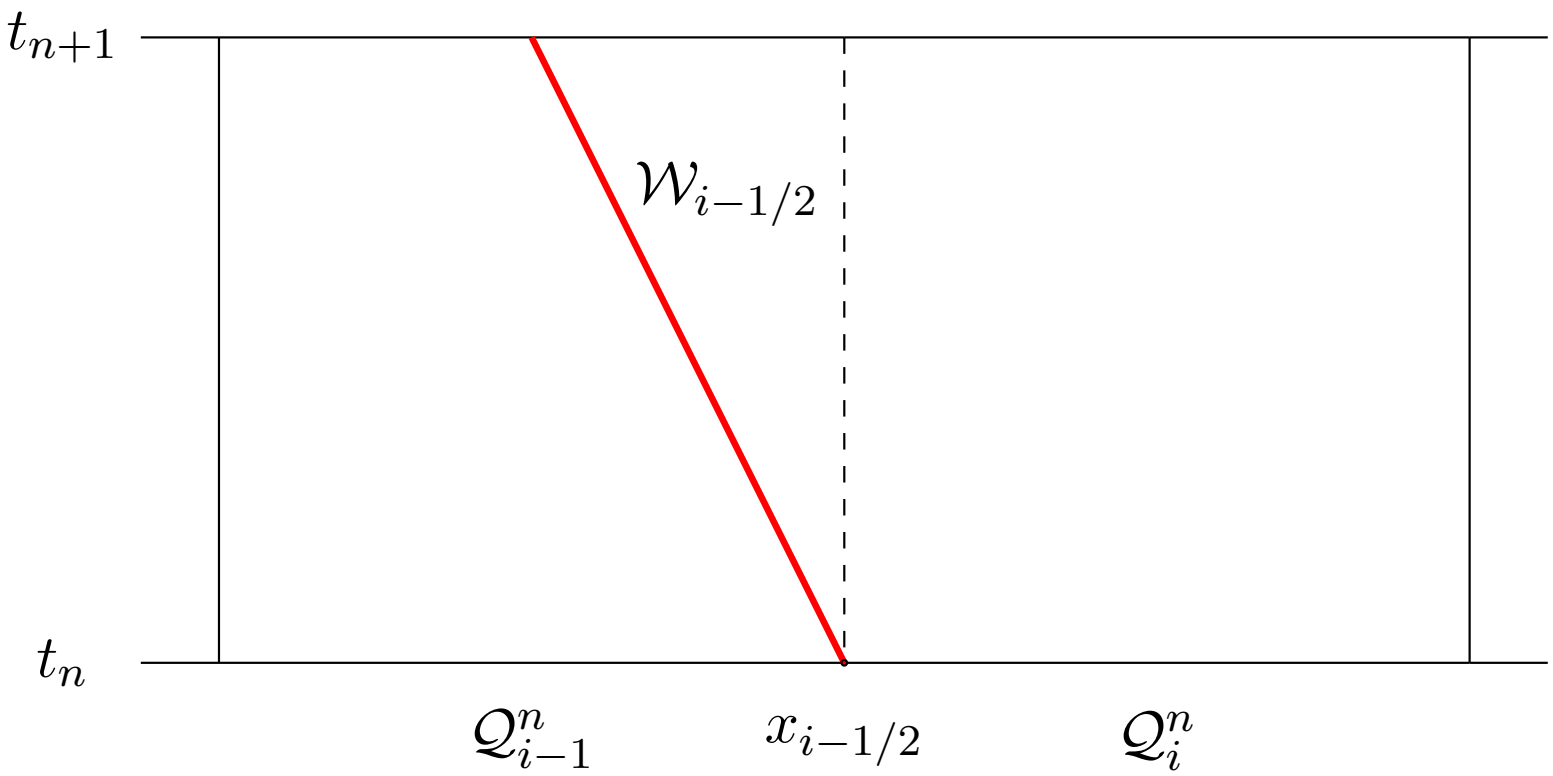


Abbildung: *Riemann-Problem im Punkt* $x_{i-1/2}$.

Riemann-Problem

Die Funktion $\tilde{q}^n(x_{i-1/2}, \cdot)$ ist konstant auf (t_n, t_{n+1}) und ihr Wert $q^{\downarrow}(\mathcal{Q}_{i-1}, \mathcal{Q}_i)$ kann durch Lösen des Riemann-Problems

$$q_t + f(q)_x = 0, \quad q(x, t_n) = \begin{cases} \mathcal{Q}_{i-1}^n, & x < x_{i-1/2}, \\ \mathcal{Q}_i^n, & x > x_{i-1/2}, \end{cases}$$

bestimmt werden. Der numerische Fluss ist dann gegeben durch

$$F_{i-1/2}^n = \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} f(q^{\downarrow}(\mathcal{Q}_{i-1}^n, \mathcal{Q}_i^n))\, dt = f(q^{\downarrow}(\mathcal{Q}_{i-1}^n, \mathcal{Q}_i^n)).$$

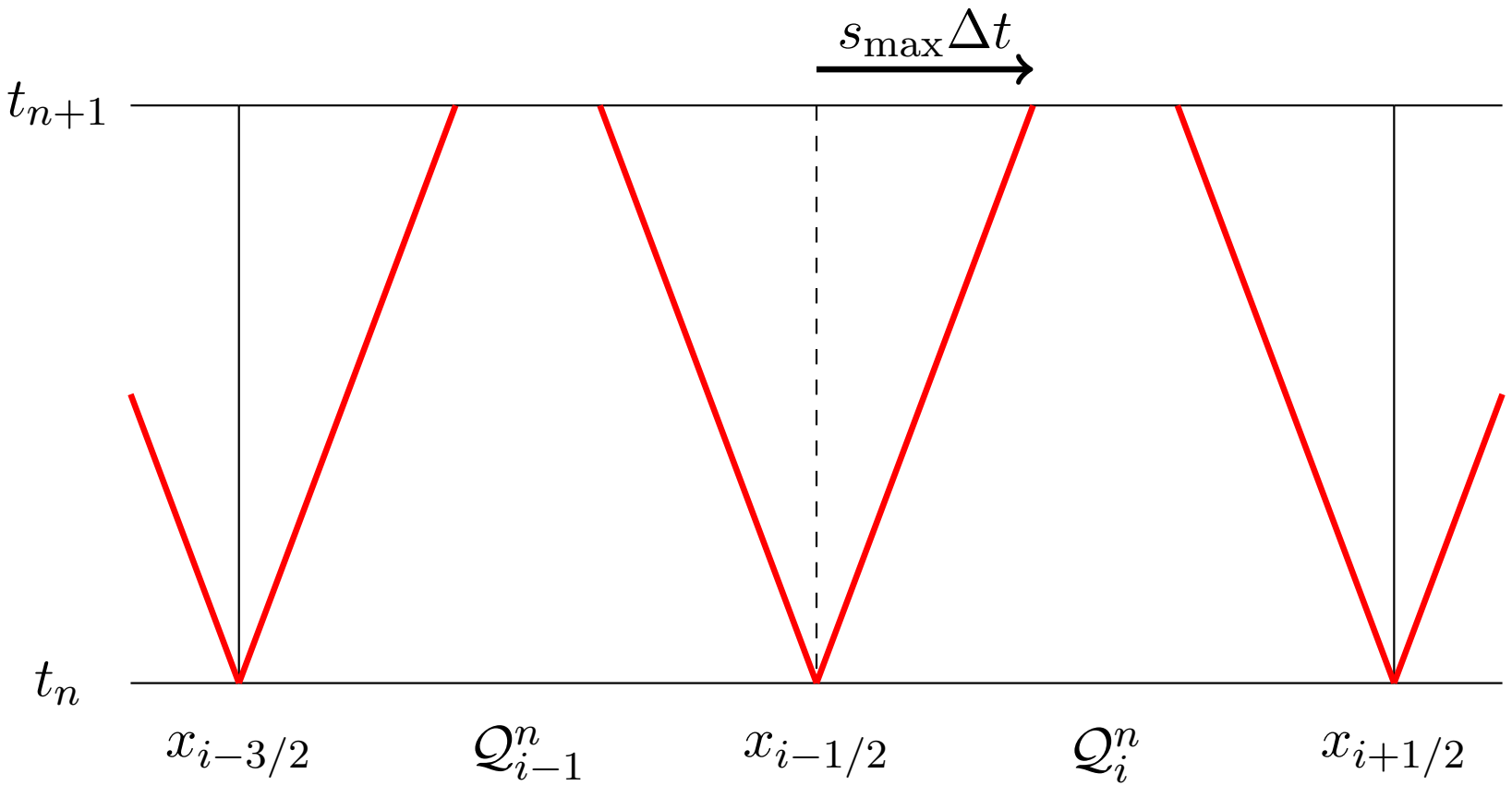


Abbildung: *Stabilitätsbedingung für Godunovs Methode.*

Stabilität und Konvergenz

- Godunovs Methode ist im Allgemeinen stabil für Courantzahl

$$\frac{s_{\max}\Delta t}{\Delta x} \leq 1,$$

  wobei $s_{\max}$ die größte Wellengeschwindigkeit ist.
- Godunovs Methode ist von erster Ordnung.
- Anstelle von konstanten Funktionen können auch lineare Funktionen $\tilde{q}$ verwendet werden. Dies bildet die Grundlage für höher aufgelöste Methoden.

Upwind-Methode

Wir betrachten die Gleichung $q_t + aq_x = 0$ mit $a \in \mathbb{R}$ konstant. Dann gilt offenbar für $t \in (t_n, t_{n+1})$

$$\tilde{q}^n(x_{i-1/2}, t) = \tilde{q}^n(x_{i-1/2} - a(t - t_n), t_n) = \begin{cases} \mathcal{Q}^n_{i-1}, & a > 0, \\ \mathcal{Q}^n_i, & a < 0. \end{cases}$$

Mit $a = a_+ + a_-$ erhalten wir den numerischen Fluss

$$F^n_{i-1/2} = a_+ \mathcal{Q}^n_{i-1} + a_- \mathcal{Q}^n_i.$$

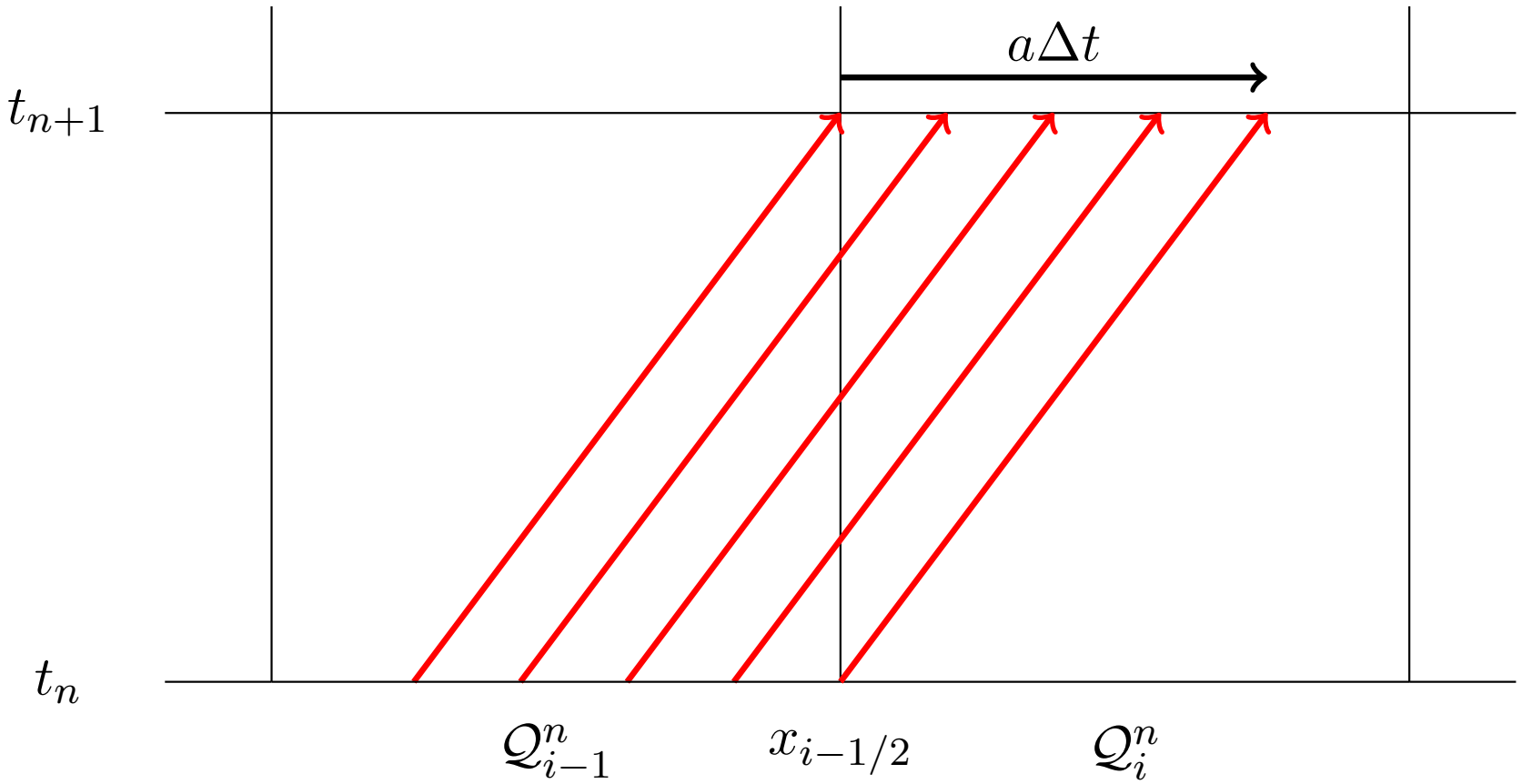


Abbildung: *Veranschaulichung der Upwind-Methode*

Das Programm

Das Programm besteht aus

- der Klasse **ProblemData**,
- der Klasse **TransportModel**
- der Klasse **FiniteVolumeScheme** und
- den Funktionen **main()** und **algorithm()**.

ProblemData

Wie zuvor beschreibt ProblemData ein Problem der Form

$$\begin{aligned} \frac{\partial q(x,t)}{\partial t} + a \cdot \nabla q(x,t) = 0, \qquad & x \in \Omega, t \in (0,T), \\ q(x,t) = h(x,t), \quad & x \in \Gamma_D, t \in (0,T), \\ q(x,0) = q_0(x), \quad & x \in \Omega. \end{aligned}$$

TransportModel

Ausgehend von einem Objekt der Klasse ProblemData wird hier der numerische Fluss F berechnet. Für eine einfache Upwind-Methode ist er gegeben durch

$$F(q_\ell, q_r, n) = \begin{cases} (aq_\ell) \cdot n, & a \cdot n < 0, \\ (aq_r) \cdot n, & a \cdot n \geq 0. \end{cases}$$

FiniteVolumeScheme

Hier wird die Lösung für den Zeitschritt t_n nach t_{n+1} berechnet.
Seien $\mathcal{C}_i \subset \Omega$ Zellen mit

- Nachbarn $\mathcal{C}_j$ mit gemeinsamen Randflächen S_{ij}, $j \in N(i)$,
- Randflächen S_j mit $S_j \subset \Gamma$, $j \in B(i)$,
- und äußerem Normalenvektor n_i.

Damit lautet das lokale Update:

$$\mathcal{Q}_i^{n+1} = \mathcal{Q}_i^n - \frac{\Delta t}{|\mathcal{C}_i|} \left(\sum_{j \in N(i)} |S_{ij}| F(\mathcal{Q}_i^n, \mathcal{Q}_j^n, n_i) + \sum_{j \in B(i)} |S_j| F(\mathcal{Q}_i^n, h(\cdot, t_n), n_i) \right)$$

Implementierung

```
1 const IteratorType end = space.end();
2 for( IteratorType it = space.begin(); it != end; ++it
      )
3 {
4   const EntityType &entity = *it;
5   const double enVolume = 1.0 / geo.volume();
6
7   const LocalFunctionType lfSolEn = solution.
      localFunction( entity );
8   LocalFunctionType lfUpdEn = update.localFunction(
      entity );
```

```
1  const IntersectionIteratorType iitend = gridPart.iend(
       entity );
2  for( IntersectionIteratorType iit = gridPart.ibegin(
     entity ); iit != iitend; ++iit )
3  {
4  const IntersectionType &intersection = *iit;
5
6  const GlobalCoordinateType  normal = intersection.
     centerUnitOuterNormal();
7  LocalCoordinateType pointEn = intersection.
     geometryInInside().center();
8  const double faceVolume = intersection.geometry().
     volume();
9
10 RangeType qLeft;
11 lfSolEn.evaluate( pointEn, qLeft );
```

```
1  if( intersection.neighbor() )
2  {
3    const EntityType neighbor = intersection.outside();
4    const double nbVolume = 1.0 / neighbor.geometry().
       volume();
5
6
7    LocalFunctionType lfSolNb = solution.localFunction(
       neighbor );
8    LocalFunctionType lfUpdNb = update.localFunction(
       neighbor );
9
10   LocalCoordinateType pointNb = intersection.
       geometryInOutside().center();
```

```
1  RangeType qRight;
2  lfSolNb.evaluate( pointNb, qRight );
3
4  RangeType flux;
5  model().numericalFlux( normal, qLeft, qRight, flux );
6
7  RangeType enFlux = flux;
8  enFlux *= -enVolume * faceVolume;
9  lfUpdEn.axpy( pointEn, enFlux );
10
11 RangeType nbFlux = flux;
12 nbFlux *= nbVolume * faceVolume;
13 lfUpdNb.axpy( pointNb, nbFlux );
14 }
```

Beispiel 1

Wir betrachten das Transportproblem für $a = (1.25, 1.25)^\top$ und

$$q_0 = \mathbb{1}_{B_{1/2}(0)}.$$

Bemerkungen

- Die zweite Summe für die Randbedingung wird ähnlich berechnet.
- Es können alternativ verschiedene Runge-Kutta-Verfahren verwendet werden.

Zusammenfassung

- Wir haben die FEM und FVM am Beispiel des Poisson Problems und der Transportgleichung kennengelernt.
- Die FEM baut auf der Galerkin-Formulierung auf, während die FVM die Erhaltungsgleichung als Grundlage nimmt.
- Wir haben die Methoden mit Hilfe der Bibliothek DUNE realisiert. Die Vorteile von DUNE bestehen in der Modularisierung und einfachen Handhabung.