Skript zum

Praktikum Numerik für Partielle Differentialgleichungen I
WS 2008/2009

Bernard Haasdonk
Institut für Numerische und Angewandte Mathematik
Universität Münster

Stand 5. Februar 2009

Inhaltsverzeichnis

Zusammenfassung

Im Praktikum sollen die in der Vorlesung “Numerik partieller Differentialgleichungen I” vorgestellten numerischen Verfahren zur Lösung partieller Differentialgleichungen programmiert werden. Ziel ist die Implementierung eines effizienten, selbstadaptiven Programmpakets zur Simulation elliptischer Differentialgleichungen mit Hilfe der Finite-Elemente-Methode. Als Programmiersprache wird C/C++ verwendet, so dass Programmierkenntnisse hilfreich sind und durch das Praktikum ausgebaut werden können. Zusätzlich findet eine Einführung in die in der Arbeitsgruppe verwendeten Programmierpakete statt. Studierende, die vorhaben, in der Angewandten Mathematik ein Zulassungs- oder Diplomarbeit zu schreiben, wird die Teilnahme an dem Praktikum empfohlen.

Bemerkung zum Skript

Die Programmierung der Aufgaben erfordert die vorherige Einführung zahlreicher Verfahren, Konzepte, und Software-Pakete. Diese werden während des Praktikums in der Präsenzstunde gehalten. Dieses Skript soll genau diese Informationen sammeln, die an der Tafel präsentiert werden. Daher ist es keine ausführliche Anleitung, sondern im wesentlichen eine knappe Mitschrift der Präsenzanteile des Praktikums. Das Skript entsteht erst parallel zur Veranstaltung im aktuellen WS 2008/2009. Es werden daher aktuell zur wöchentlichen Veranstaltung immer neue Versionen Online gestellt. Für Fehlerfreiheit kann daher zunächst nicht garantiert werden. Korrekturen und Kommentare zum Skript sind daher jederzeit willkommen.

1 Einführung

Ziel der Veranstaltung ist die numerische Behandlung von PDGLn.

1.1 Beispiele:

Sei $\Omega \in \mathbb{R}^d$ ein polygonales Gebiet.

a) stationäre (d.h. zeitunabhängige) Wärmeleitung mit isolierenden Randbedingungen.
Gesucht ist $u : \Omega \to \mathbb{R}$ s.d.

$$-\nabla \cdot (\kappa \nabla u) = f \quad \text{in } \Omega \tag{1}$$
$$(\kappa \nabla u) \cdot n = 0 \quad \text{auf } \partial\Omega \tag{2}$$

b) instationäre (d.h. zeitabhängige) Konvektion-Diffusion mit Nullrandbedingungen.
Gesucht ist $u : \Omega \times [0, T] \to \mathbb{R}$ s.d.

$$\partial_t u + \nabla \cdot (cu^3 - \kappa \nabla u) = 0 \quad \text{in } \Omega \times [0, T] \tag{3}$$
$$u = 0 \quad \text{in } \partial\Omega \times [0, T] \tag{4}$$
$$u = u_0 \quad \text{für t=0} \tag{5}$$

1.2 Schritte der Numerischen Behandlung

- Diskretisierung von Ω durch ein Gitter: z.B. Dreiecke/Rechtecke
- Diskreter Funktionenraum V_h im Ort: z.B. Elementweise konstant, linear, etc.
- Zeitdiskretisierung $0 = t^0 < ... < t^K = T$
- Ziel nun: finde $u_h^k \in V_h$ mit $u_h^k(x) \approx u(x, t^k)$
- Ansatz der numerischen Lösung als Linearkombination von Basisvektoren φ_n mit Koeffizienten a_n^k (sogenannte DOFs, degrees of freedom)

$$u_h^k = \sum_{n=1}^{N} a_n^k \varphi_n$$

- Diskretisierungsverfahren der PDGL liefert Berechnungsverfahren für die DOFs: z.B. FEM, FV, LDG
- Implementation und Berechnung der numerischen Lösung durch Computer.
- Auswertung der Ergebnisse: z.B. Visualisierung, Berechnung von Zielgrößen.

Hier: Realisierung unter Unix/Linux, Programmiersprache C++, Numerik-Bibliothek Dune

2 Finite-Differenzen-Verfahren für Transport-Gleichung

2.1 Skalare Erhaltungsgleichung in 1D

Sei $\Omega = [a, b] \subset \mathbb{R}$ ein Intervall, $T > 0$ Endzeit. Gesucht ist $u(x, t)$ als Lösung von

$$\begin{aligned} \partial_t u(x,t) + \partial_x f(u(x,t)) &= 0 \quad \text{in } \Omega \times [0,T] \\ u(x,0) &= u_0(x) \text{ in } \Omega \\ u(x,t) &= u_{dir}(x,t) \text{ auf } \Gamma_{dir} \end{aligned}$$

mit Anfangsdaten u_0, Dirichlet-Randdaten u_{dir}, Flussfunktion $f(u)$, Dirichlet-Rand $\Gamma_{dir}(t) := \{x \in \partial\Omega | f'(u)n < 0\}$ mit äußerer Einheitsnormalen n.
Bemerkung: *Trotz glatter Daten können sich unstetige Lösungen entwickeln. Daher wird der Lösungsbegriff zu schwachen Lösungen mit "Entropiebedingung" erweitert. Unter gewissen Bedingungen an die Daten gilt Existenz und Eindeutigkeit von Entropielösungen, siehe [13].*

2.2 Finite-Differenzen-Diskretisierung

2.2.1 Gitter

Wir definieren

- ein Gitter $G = \{x_n\}_{n=1}^N$ auf Ω aus N Punkten mit $a = x_1 < \ldots < x_N = b$
- für die Randbehandlung die gespiegelten Punkte $x_0 = a - (x_2 - x_1)$ und $x_{N+1} = b + (x_N - x_{N-1})$
- lokale Gitterweiten $\Delta x_n := \frac{1}{2}(x_{n+1} - x_{n-1})$
- die Zwischenpunkte $x_{n+\frac{1}{2}} := \frac{1}{2}(x_{n+1} + x_n)$
- $K + 1$ Zeitpunkte durch $t^k = k\Delta t, k = 0, \ldots, K$ für Zeitschrittweite Δt

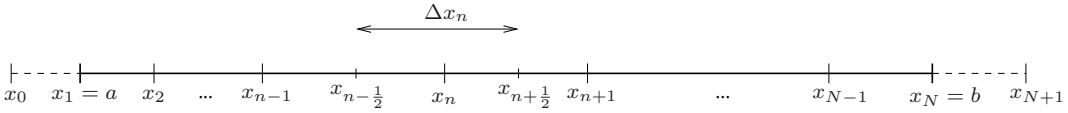

Abbildung 1: Notationen eines 1D Punktegitters mit N Knoten auf $[a, b] \subset \mathbb{R}$.

2.2.2 Diskreter Funktionenraum im Ort

Wir legen Basisfunktionen $\varphi_n : G \to \mathbb{R}$ fest durch $\varphi_n(x_m) := \delta_{nm}$. Hiermit definieren wir den diskreten Funktionenraum bzgl. des Ortes $V_h := \text{span}\{\varphi_n\}_{n=1}^N$.
Bemerkung: *Wir verzichten hier auf eine Erweiterung des Trägers auf Ω.*
Gesucht ist also $u_h^k \in V_h, k = 0, \ldots, K$ mit Darstellung $u_h^k = \sum_{n=1}^N u_n^k \varphi_n$, so dass $u_n^k \approx u(x_n, t^k)$.

2.2.3 Projektion der Anfangsdaten

Einfache Projektion der Anfangsdaten $u_h^0 := P_h(u_0)$ ist durch Punktauswertung möglich

$$(P_h(u_0))(x_n) := u_0(x_n).$$

2.2.4 Diskreter Ortsoperator

Wir verwenden eine Approximation des Orts-Differentialoperators durch 'zentrale' Differenz:

$$\begin{aligned}(\partial_x f(u))(x_n, t^k) &\approx \frac{1}{\Delta x_n}\left(f(u(x_{n+\frac{1}{2}}, t^k)) - f(u(x_{n-\frac{1}{2}}, t^k))\right) &(6)\\ &\approx \frac{1}{\Delta x_n}\left(g(u_n^k, u_{n+1}^k) - g(u_{n-1}^k, u_n^k)\right) &(7)\end{aligned}$$

mit numerischem Fluss g, der konsistent $g(u,u) = f(u)$ und Lipschitz ist.
Definiere u_{N+1}^k bzw u_0^k in Abhängigkeit des Randtyps:

$$\begin{aligned}u_0^k &:= \begin{cases} u_{dir}(a, t^k) & \text{falls } a \in \Gamma_{dir} \\ u_1^k & \text{sonst.}\end{cases} &(8)\\ u_{N+1}^k &:= \begin{cases} u_{dir}(b, t^k) & \text{falls } b \in \Gamma_{dir} \\ u_N^k & \text{sonst.}\end{cases} &(9)\end{aligned}$$

Motiviert durch unsere PDGL $\partial_t u = L(u)$ mit $L(u) := -\partial_x f(u)$ definieren wir den diskreten Ortsoperator $L_h : V_h \to V_h$

$$(L_h(u_h^k))(x_n) := -\frac{1}{\Delta x_n}\left(g(u_n^k, u_{n+1}^k) - g(u_{n-1}^k, u_n^k)\right).$$

Bemerkung: *"Erhaltungseigenschaft": Falls $u_{dir} = 0$ und $u(a,t) = u(b,t) = 0$, so gilt*

$$\int_a^b \partial_x f(u(x,t)) = f(u(b,t)) - f(u(a,t)) = 0.$$

Mit der Definition einer diskreten 'Masse' ergibt sich im Fall $u_{dir} = 0$, $u_0^k = u_N^k = 0$

$$\sum_{n=1}^{N} \Delta x_n (L_h(u_h^k))(x_n) = \sum_{n=1}^{N} \Delta x_n \left(g(u_n^k, u_{n+1}^k) - g(u_{n-1}^k, u_n^k)\right) = 0.$$

2.2.5 Zeitdiskretisierung

Für die Zeitdiskretisierung wählen wir Euler-Vorwärts. Das numerische Verfahren berechnet also nacheinander für $k = 1, \ldots, K-1$

$$u_h^{k+1} := u_h^k + \Delta t L_h(u_h^k).$$

2.2.6 Numerische Flüsse

Zentrale Differenz:

$$g(u,v) := \frac{1}{2}\left(f(u)+f(v)\right)$$

ist bei Transportproblem schlecht, weil nicht unbedingt Entropielösung gefunden wird.

Lax-Friedrichs-Fluss:

$$g(u,v) := \frac{1}{2}\left(f(u)+f(v)\right) + \frac{1}{2\lambda}(u-v)$$

mit Term für zusätzliche "numerische Viskosität". Man wählt λ möglichst groß, dass $\lambda \sup_u |f'(u)| \leq 1$.

Engquist-Osher-Fluss:

$$f^+(w) := f(0) + \int_0^w \max(f'(s),0)ds \qquad f^-(w) := \int_0^w \min(f'(s),0)ds$$

$$g(u,v) := f^+(u) + f^-(v).$$

Falls $f'(s) > 0$ folgt $g(u,v) = f(u)$ "Rückwärts-Differenz".
Falls $f'(s) < 0$ folgt $g(u,v) = f(v)$ "Vorwärts-Differenz".

2.2.7 CFL-Bedingung

Für Stabilität des numerischen Verfahrens muss Δt so klein gewählt werden, dass die (Courant Friedrich Levy) Bedingung gilt. Für den Engquist Osher Fluss lautet diese

$$\sup_u \frac{\Delta t |f'(u)|}{\Delta x_n} \leq 1.$$

Für den Lax-Friedrichs-Fluss wird Δt zusätzlich durch λ eingeschränkt

$$\sup_u \frac{\Delta t |\frac{1}{2}f'(u) + \frac{1}{2\lambda}|}{\Delta x_n} \leq 1.$$

Im wesentlichen ist also Δt proportional zum kleinsten Δx_n zu wählen.

2.3 Experimental Order of Convergence (EOC)

Sei allgemein u_h eine numerische Approximation einer Funktion u in Abhängigkeit eines Diskretisierungsparameters h. Die Konvergenz $\lim_{h\to 0} u_h = u$ kann oft quantifiziert werden durch

$$\|u_h - u\| \leq Ch^\alpha$$

in einer geeigneten Norm $\|\cdot\|$ und maximalem α, der Konvergenzordnung.
Dies ergibt eine Möglichkeit, ein numerisches Verfahren zu verifizieren: Falls u bekannt ist, und zwei numerische Lösungen u_h und $u_{h'}$ berechnet sind, kann α unter Annahme des idealen Fehlergesetzes $\|u_h - u\| = Ch^\alpha$ geschätzt werden durch

$$\alpha \approx EOC(h,h') := \frac{\log(\|u_{h'} - u\| \,/\, \|u_h - u\|)}{\log(h'/h)}.$$

3 Hierarchische Gitter

Sei $\Omega \subset (\mathbb{K})^w$ ein beschränktes polygonales Gebiet mit $\mathbb{K} = \mathbb{R}, \mathbb{C}$ und $\dim(\Omega) = d$.

3.1 Referenzelemente und Entitäten

- Ein *Referenzelement* $\hat{e} \subset \mathbb{K}^n$ ist ein konvexes Polytop, d.h. beschränkter Schnitt endlichlich vieler Halbräume. (Einheitsquadrat, Einheitsdreieck, Einheitswürfel, Einheitsintervall, ...).
- Eine (kleine) *Menge von Referenzelementen* $\hat{\mathcal{E}} = \{\hat{e}_i\}_{i=1}^R$ definiert die möglichen elementaren Bestandteile eines Gitters.
- Eine *Entität* $e \subset \mathbb{K}^w$ ist das Bild eines Referenzelementes $\hat{e} \in \hat{\mathcal{E}}$ unter einer diffeomorphen Abbildung F_e, der *Referenzabbildung*.
- Die Dimension einer Entität ist $\dim(e) := \dim(\hat{e})$.
- *Subentitäten* einer Entität sind ihre Randflächen/Kanten/Eckpunkte, etc.
- Die *Codimension* einer Subentität e ist $\mathrm{codim}(e) := d - \dim(e)$.
- Codim-0 Entitäten nennt man *Elemente*.

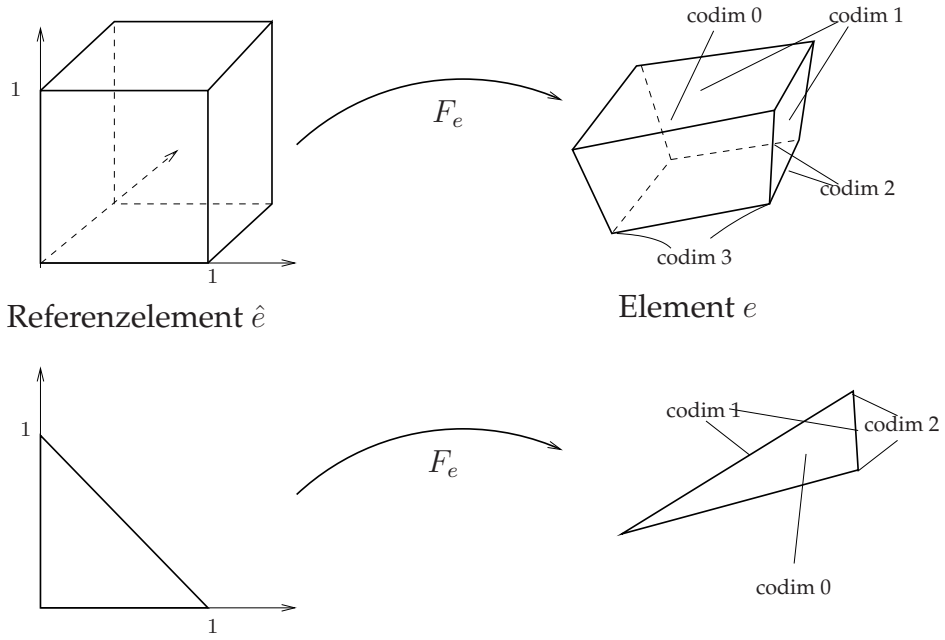

Abbildung 2: Referenzelemente, Referenzabbildungen und Subentitäten verschiedener Kodimensionen.

3.2 Hierarchisches Gitter

Ein *hierarchisches Gitter* auf Ω ist ein Tupel $G = (\mathcal{E}_l)_{l=0}^{l_{max}}$, wobei $\mathcal{E}_l$ die *Menge von Elementen auf Level l* bezeichnet, und gilt

- die Level-0 Elemente überdecken Ω, d.h. $\bigcup_{e \in \mathcal{E}_0} e = \bar{\Omega}$
- Elemente eines Levels überlappen sich nicht, d.h. für alle l gilt $\overset{\circ}{e} \cap \overset{\circ}{e'} = \emptyset$ für $e \neq e', e, e' \in \mathcal{E}_l$.
- für alle $l > 0, e \in \mathcal{E}_l$ existiert ein *Eltern-Element (Parent)* $e' \in \mathcal{E}_{l-1}$ mit $e \subset e'$, e heißt umgekehrt *Kind-Element (Child)* von e'.
- Jedes Element mit mindestens einem Kind-Element, zerfällt vollständig in Kind-Elemente.

Elemente ohne Kind-Elemente nennt man *Blatt-Elemente (Leaf-Elements)*.

3.3 Gitterteile

Für konkrete Numerik muss man sich auf *Gitterteile (Gridparts)* beschränken, siehe Fig. 3.

- Das Level-0 Gridpart besteht aus den Elementen ohne Eltern, den sogenannten *Makroelementen*.
- Das *Level-l Gridpart* besteht aus den Elementen auf Level l.
- Das *Leaf-Gridpart* besteht aus den Blatt-Elementen aller Level.

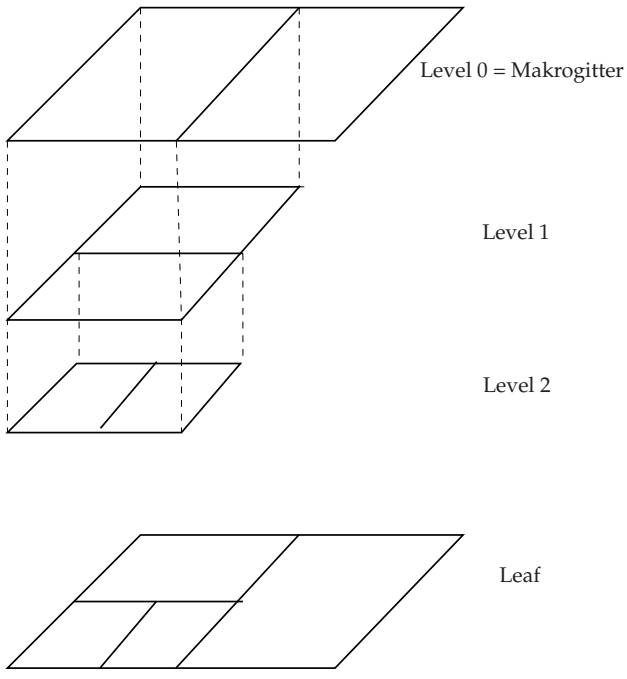

Abbildung 3: Hierarchisches Gitter mit verschiedenen Level-Gridparts und dem Leaf-Gridpart.

4 Diskrete Funktionenräume

Wir führen einige Begriffe für (teilweise triviale) mathematische Konzepte ein, die wir jedoch anschließend mit genau diesen Begriffen in Dune realisiert finden, siehe Abschnitt 7.4.

4.1 Funktionenräume

Ohne spezielle Regularitätsanforderungen und Einschränkungen, ist ein *Funktionenraum (FunctionSpace)* die Menge der Abbildungen

$$V := \{u : (\mathbb{K}_D)^d \to (\mathbb{K}_R)^n\}. \tag{10}$$

Hier ist $(\mathbb{K}_D)^d$ der *Definitionsbereich (Domain)* und $(\mathbb{K}_R)^n$ der *Wertebereich (Range)*. Hier ist also $\mathbb{K}_D$ der Koordinatentyp des Definitionsbereichs (DomainField) und $\mathbb{K}_R$ der Koordinatentyp des Wertebereichs (RangeField), z.B. reelle oder komplexe Zahlen. Die Jacobi-Matrix einer solchen Funktion ausgewertet im Punkt x $(Du)(x) \in \mathbb{K}^{n\times d}$ liegt also im *Jacobi-Wertebereich (JacobianRange)* $\mathbb{K}^{n\times d}$.

4.2 Basisfunktionen auf Referenzelementen

Auf einem Referenzelement $\hat{e} \subset \mathbb{K}_D^d$ definieren wir uns eine lokale *Menge von Basisfunktionen (BaseFunctionSet)*

$$B_{\hat{e}} := \{\hat{\varphi}_{\hat{e},1}, \ldots, \hat{\varphi}_{\hat{e},m}\} \tag{11}$$

von Funktionen $\hat{\varphi}_{\hat{e},i} : \mathbb{K}_D^d \to \mathbb{K}_R^n$ mit Träger in $\hat{e}$ und $\hat{\varphi}_{\hat{e},i}|_{\mathrm{clos}(\hat{e})} \in C^0(\mathrm{clos}(\hat{e}))$ also stetig auf dem Abschluss des Referenzelementes.

4.3 Diskrete Funktionenräume

Sei $\mathcal{E}$ die Menge der Elemente eines Gitterteils (GridPart). Sei $\Gamma := \cup_{e\in\mathcal{E}} \partial e$ die Menge aller Oberflächen aller Elemente. Wir nehmen an, dass eine surjektive Abbildung von elementweisen lokalen Indizes in globale Indizes $g : \mathcal{E} \times \{1, \ldots, m\} \to \{1, \ldots, N\}$ gegeben ist. Hierdurch werden globale Basisfunktionen $\varphi_j : \Omega\backslash\Gamma \to \mathbb{K}_R^n, j = 1, \ldots, N$ definiert durch

$$\varphi_j := \sum_{(e,i):g(e,i)=j} \hat{\varphi}_i \circ F_e^{-1}. \tag{12}$$

Hierbei ist F_e die (auf $\mathbb{K}_D^d$ erweiterte) Referenzabbildung eines Elementes $e \in \mathcal{E}$. Auf Kanten sind diese Funktionen zunächst nicht definiert. Wir definieren dann einen *diskreten Funktionenraum (DiscreteFunctionSpace)* durch

$$V_h := \left\{ u_h \in V : \quad u_h = \sum_{j=1}^{N} b_j \varphi \right\} \tag{13}$$

Ein Element u_h in einer solchen Funktionsmenge ist daher eine *diskrete Funktion (DiscreteFunction)*, welche global als Linearkombination von globalen Basisfunktionen mit *globalen Freiheitsgraden* $b_j \in \mathbb{K}_R$ interpretiert werden kann. Für die Numerik ist jedoch eine äquivalente lokale Darstellung wichtiger

$$V_h = \left\{ u_h \in V : \quad u_H|_e = \sum_{i=1}^{m} a_{e,i} \hat{\varphi}_{\hat{e},i} \circ F_e^{-1} \text{ mit } a_{e,i} = b_{g(e,i)} \quad \forall e \in \mathcal{E} \right\} \tag{14}$$

Ein u_h kann demnach auf jedem Element e als *lokale Funktion (LocalFunction)* $u_h|_e$ mit lokalen Freiheitsgraden (DOFs) $a_{e,i}$ dargestellt werden. Im Gegensatz zur globalen Funktion, macht auf einer lokalen Funktion eine Auswertung auf Kanten sinn, denn mit Stetigkeit der Basisfunktionen auf dem Referenzelement hat $u_h|_e$ eine stetige Erweiterung auf ∂e. Durch geeignete Wahl der Indexabbildung g können zusätzliche Eigenschaften der diskreten Funktionen garantiert werden, z.B. stetige oder differenzierbare Erweiterbarkeit auf ganz Ω.

5 Quadraturen

5.1 Integration über Gebiet

Integration über das Gebiet wird auf eine Integration über Elemente und Summierung zurückgeführt:

$$\int_\Omega f(x)dx = \sum_{e\in\mathcal{E}} \int_e f(x)dx, \qquad \int_{\partial\Omega} f(x)ds(x) = \sum_{e\in\mathcal{E}} \int_{e\cap\partial\Omega} f(x)ds(x). \tag{15}$$

5.2 Integration über Gitterelemente

Integration über Elemente wird auf Referenzelemente zurückgeführt durch Transformationssatz und Kettenregel

- Beispiel: globale Basisfunktionen:

$$\int_e f(x)\varphi_j(x)dx = \int_{\hat{e}} f(F_e(\hat{x}))\hat{\varphi}_i(\hat{x})|\det DF_e|d\hat{x} \tag{16}$$

  mit $j = g(e,i)$ und $\varphi_j = \hat{\varphi}_i \circ F_e^{-1}$. Die Größe $|\det DF_e(\hat{x})|$ wird *Integrationselement (IntegrationElement)* genannt.

- Beispiel: Ableitungen von globalen Basisfunktionen:

$$D_{\hat{x}}\hat{\varphi}_i(\hat{x}) = D_x\varphi_j(x)D_{\hat{x}}F_e(\hat{x})$$

  also

$$D\hat{\varphi}_i(\hat{x})(DF_e(\hat{x}))^{-1} = D\varphi_j(x)$$

  und

$$\nabla_x\varphi_j(x) = (D\varphi_j)^T = ((DF_e(\hat{x}))^{-1})^T\nabla_{\hat{x}}\hat{\varphi}_i(\hat{x})$$

  Mit der Abkürzung $J(\hat{x}) := ((DF_e(\hat{x}))^{-1})^T$ für diese invertierte und transponierte Jacobi-Matrix *(JacobianInverseTransposed)* folgt dann für Integrale

$$\int_e v(x)^T\nabla\varphi_j = \int_{\hat{e}} v(F_e(\hat{x}))^T(J(\hat{x})\nabla_{\hat{x}}\hat{\varphi}_i)|\det DF_e|d\hat{x}. \tag{17}$$

5.3 Approximation durch Quadraturen

Eine *Quadratur* dient zur Approximation von Integralen über Referenzelementen. Eine solche ist gegeben durch $n_p \in \mathbb{N}$ die Anzahl der Quadraturpunkte p_i und Quadraturgewichte ω_i. Die Approximation geschieht durch

$$\int_{\hat{e}} f(\hat{x})d\hat{x} \approx \sum_{i=1}^{n_p} \omega_i f(p_i).$$

6 Finite-Elemente für Elliptische Probleme

Es soll im Folgenden die Diskretisierung für das Poisson-Problem mit gemischten Randbedingungen hergeleitet werden. Als weiterführende Referenzen dienen [4, 16, 5].

6.1 Elliptisches Problem

Sei $\Omega \subset \mathbb{R}^w$ polygonales Gebiet mit Dirichlet-Rand $\Gamma_D \subset \partial\Omega$ und Neumann-Rand $\Gamma_N := \partial\Omega \backslash \Gamma_D$ und äußeren Einheitsnormalen $n(x)$. Gesucht ist $u \in C^2(\Omega) \cap C^1(\bar{\Omega})$ mit

$$\begin{aligned} -\nabla \cdot (a(x)\nabla u(x)) &= f(x) \quad \text{in } \Omega \\ u(x) &= g_D(x) \quad \text{auf } \Gamma_D \\ a(x)\nabla u(x) \cdot n(x) &= g_N(x) \quad \text{auf } \Gamma_N \end{aligned} \tag{18}$$

mit $a(x) > 0$ und alle Datenfunktionen genügend regulär.
Falls u klassische Lösung ist, so gilt für alle $\varphi \in C^\infty(\Omega) \cap C^0(\bar{\Omega})$ mit $\varphi|_{\Gamma_D} = 0$

$$\begin{aligned} \int_\Omega f(x)\varphi(x)dx &= \int_\Omega -\nabla \cdot (a(x)\nabla u(x))\varphi(x)dx \\ &= \int_\Omega a(x)\nabla u \cdot \nabla\varphi - \int_{\partial\Omega} a(x)\varphi(x)\nabla u \cdot n ds(x) \\ &= \int_\Omega a(x)\nabla u(x) \cdot \nabla\varphi(x)dx - \int_{\Gamma_D} a(x) \underbrace{\varphi(x)}_{=0} \nabla u \cdot n ds(x) \\ &\quad - \int_{\Gamma_N} \underbrace{a(x)\nabla u \cdot n}_{g_N(x)} \varphi(x) ds(x). \end{aligned}$$

6.2 Schwache Form

Raum der H^1-Funktionen mit Nullrandwerten auf Γ_D:

$$H^1_{\Gamma_D}(\Omega) := \mathrm{clos}_{H^1}(\{\varphi \in C^\infty(\Omega) \cap C^0(\bar{\Omega}) \mid \varphi|_{\Gamma_D} = 0\})$$

Funktionenraum mit inhomogenen Randwerten $g \in H^1(\Omega)$:

$$V(g) := \{v \in H^1(\Omega) \mid v - g \in H^1_{\Gamma_D}(\Omega)\}.$$

Also ist insbesondere $H^1_{\Gamma_D} = V(0)$. Gesucht ist nun $u \in V(g_D)$ mit

$$\int_\Omega a(x)\nabla u(x) \cdot \nabla\varphi(x)dx = \int_\Omega f(x)\varphi(x)dx + \int_{\Gamma_N} g_N(x)\varphi(x)ds(x) \quad \forall \varphi \in V(0). \tag{19}$$

6.3 Finite Elemente Diskretisierung

Sei $\mathcal{E}$ simpliziale Triangulierung von Ω und konform, d.h. ohne hängenden Knoten. Als Basisfunktionen auf dem Referenzsimplex $\hat{e}$ mit Knoten $\hat{v}_k, k = 1, \ldots, w+1$ werden die linearen Funktionen $\hat{\varphi}_i \in \mathbb{P}_1(\hat{e})$ gewählt mit $\hat{\varphi}_i(\hat{v}_k) = \delta_{ik}$.

Sei $v_j, j = 1, \ldots, n$ eine Aufzählung der Knoten des Gitters. Durch die Indexabbildung $g(e, i) := j$ für $F_e(\hat{v}_i) = v_j$ werden die lokalen Freiheitsgrade mit globalen Freiheitsgraden identifiziert.
Die resultierenden globalen Basisfunktionen φ_j sind stetig (erweiterbar) auf $\bar{\Omega}$ und erfüllen $\varphi_j(v_i) = \delta_{ij}$, sind also "Hütchenfunktionen".
Diskreter Funktionenraum mit inhomogenen Randwerten g:

$$V_h(g) := \{v \in \text{span}\{\varphi_j\} \mid v(v_i) = g(v_i)\ \forall v_i \in \Gamma_D\}.$$

FEM-Diskretisierung: Gesucht ist $u_h \in V_h(g_D)$ mit

$$\int_\Omega a(x)\nabla u(x) \cdot \nabla\varphi(x)dx = \int_\Omega f(x)\varphi(x)dx + \int_{\Gamma_N} g_N(x)\varphi(x)ds(x) \quad \forall\varphi \in V_h(0). \quad (20)$$

6.4 Lineares Gleichungssystem

Wegen Linearität von (20) reicht es, als Testfunktionen $\varphi = \varphi_i \in V_h(0)$ zu betrachten. Der Ansatz $u_h(x) = \sum_{j=1}^n b_j\varphi_j(x)$ liefert Bedingungen

$$\int_\Omega a(x)\sum_j b_j\nabla\varphi_j \cdot \nabla\varphi_i dx \quad = \quad \int_\Omega f\varphi_i dx + \int_{\Gamma_N} g_N\varphi_i ds(x) \quad \text{für } v_i \notin \Gamma_D. \quad (21)$$

Damit $u_h \in V_h(g_D)$, fordern wir

$$b_i = g_D(v_i) \quad \text{für } v_i \in \Gamma_D. \quad (22)$$

Dies ergibt ein $n \times n$ lineares Gleichungssystem (LGS) $\mathbf{Sb} = \mathbf{r}$, eine Zeile pro Testfunktion und eine Zeile pro Dirichletknoten, z.B. falls $v_1, v_n \notin \Gamma_D$ und $v_i \in \Gamma_D$:

$$\begin{pmatrix} \int_\Omega a\nabla\varphi_1 \cdot \nabla\varphi_1 & \ldots & \ldots & \ldots & \int_\Omega a\nabla\varphi_n \cdot \nabla\varphi_1 \\ \vdots & & & & \\ 0 & & \ldots 0 \quad 1 \quad 0 \ldots & & 0 \\ \vdots & & & & \\ \int_\Omega a\nabla\varphi_1 \cdot \nabla\varphi_n & \ldots & \ldots & \ldots & \int_\Omega a\nabla\varphi_n \cdot \nabla\varphi_n \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} \int_\Omega f\varphi_1 + \int_{\Gamma_N} g_N\varphi_1 \\ \vdots \\ g_D(v_i) \\ \vdots \\ \int_\Omega f\varphi_n + \int_{\Gamma_N} g_N\varphi_n \end{pmatrix}$$

6.5 Algorithmische Aspekte

6.5.1 LGS-Eigenschaften

Die Steifigkeitsmatrix $\mathbf{S}$ ist im allgemeinen

- groß
- dünn besetzt (sparse)
- eventuell strukturiert (Band- / Blockstruktur)
- unsymmetrisch (falls $v_j \notin \Gamma_D, v_i \in \Gamma_D$, so ist $S_{ji} \neq 0$ aber $S_{ij} = 0$)

$\Rightarrow$ Verwendung von Sparse-Matrix-Klassen und iterative LGS-löser für unsymmetrische Systeme.

6.5.2 Assemblierung

Jeder Eintrag von $\mathbf{S}, \mathbf{r}$ beruht auf Element/Randintegrale, erfordern Gitterdurchläufe zur Quadratur. Statt vielen teuren Gitterdurchläufen für die einzelnen Einträge erfolgt Assemblierung der Matrix und der rechten Seite in einem (oder zwei) Gitterdurchläufen:

- Initialisiere $\mathbf{S} = 0, \mathbf{r} = 0$.
- Für alle Elemente $e \in \mathcal{E}$ und alle lokalen Basisfunktionen φ_i, φ_j berechne die *lokalen Elementbeiträge*

$$\int_e a\nabla\varphi_i \cdot \nabla\varphi_j dx \quad \text{und} \quad \int_e f\varphi_i dx + \int_{\partial e \cap \Gamma_N} g_N \varphi_i ds(x)$$

  und verteile diese durch Addition zu den richtigen Einträgen in $\mathbf{S}, \mathbf{r}$ (globale Spalten/Zeilenindizes $g(e,i)$,$g(e,j)$).
- Für jeden Dirichlet-Knoten v_i, erzeuge Einheitszeile in $\mathbf{S}$ und setze i-ten Eintrag in $\mathbf{r}$ auf Randwert $g_D(v_i)$.

6.5.3 Symmetrisierung

Optional: Addiere für alle Paare $v_j \notin \Gamma_D, v_i \in \Gamma_D$ das $(-\int_\Omega a\nabla\varphi_j \cdot \nabla\varphi_i)$-fache der i-ten Zeile zur j-ten Zeile im LGS:

$$\begin{pmatrix} \int_\Omega a\nabla\varphi_1 \cdot \nabla\varphi_1 & \dots & 0 & \dots & \int_\Omega a\nabla\varphi_n \cdot \nabla\varphi_1 \\ \vdots & & & & \\ 0 & & 0 & 1 & 0 & & 0 \\ \vdots & & & & \\ \int_\Omega a\nabla\varphi_1 \cdot \nabla\varphi_n & \dots & 0 & \dots & \int_\Omega a\nabla\varphi_n \cdot \nabla\varphi_n \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{pmatrix} = \mathbf{r} - \begin{pmatrix} \sum_{v_j \in \Gamma_D} g_D(v_j) \int_\Omega a\nabla\varphi_j \cdot \nabla\varphi_1 \\ \vdots \\ 0 \\ \vdots \\ \sum_{v_j \in \Gamma_D} g_D(v_j) \int_\Omega a\nabla\varphi_j \cdot \nabla\varphi_n \end{pmatrix}$$

LGS ist symmetrisch und positiv definit $\Rightarrow$ Verwendung von iterativen Gleichungssystemlöser für symmetrische, positiv definite Systeme.

6.6 Fehlerschätzer für Adaptivität

Für den Fall obiger Gleichung mit $g_D = 0$ kann man a posteriori Fehlerschätzer herleiten. Gegeben eine Lösung $u_h \in V_h(g_D) = V_h(0)$ der schwachen Form, definieren wir die Elementresiduen

$$r := f + \nabla \cdot (a\nabla u_h)$$

welche elementweise L^2 sind falls f in L^2, a elementweise in C^1 ist und wir elementweise differenzierbare Ansatzfunktionen in V_h haben. Weiter definieren wir die Kantenresiduen

$$R := \begin{cases} g_N - (a\nabla u_h) \cdot n & \text{auf} \quad \Gamma_N \\ 0 & \text{auf} \quad \Gamma_D \\ -\frac{1}{2}[a\nabla u_h] & \text{auf} \quad \partial\mathcal{E} \backslash \partial\Omega. \end{cases}$$

Hierbei definieren wir den Sprung des Flusses in Normalenrichtung auf inneren Kanten $\gamma = \partial e \cap \partial e'$ zwischen den Elementen $e, e' \in \mathcal{E}$ als

$$[a\nabla u_h] := a|_e \nabla(u_h|_e) \cdot n_e + a|_{e'} \nabla(u_h|_{e'}) \cdot n_{e'}.$$

Mit der Notation $h_e := \mathrm{diam}(e)$ kann man Elementfehlerschätzer definieren durch

$$\eta_e^2 := h_e^2 \left\| r \right\|_{L^2(e)}^2 + h_e \left\| R \right\|_{L^2(\partial e)}^2 .$$

Wir definieren die Energienorm für Funktionen $v \in V(0)$ als $|||v|||^2 := \int_\Omega a(\nabla v)^2$. Man kann dann zeigen, dass eine von h_e unabhängige Konstante C existiert (aber im allgemeinen unbekannt ist), welche den Fehler beschränkt durch

$$|||u - u_h|||^2 \leq C \sum_{e \in \mathcal{E}} \eta_e^2.$$

Für eine Herleitung siehe z.B. [1]. Hiermit läßt sich eine adaptive Strategie zur Gitterverfeinerung formulieren, welche eine Gleichverteilung der Fehlerschätzer zum Ziel hat. Sei hierzu $\varepsilon > 0$ und $\gamma \in (0, 1)$, z.B. typischerweise $\gamma = 0.5$.

1. Starte mit $i = 0$ und dem vorgegebenen Gitter $\mathcal{E}^{(0)} := \mathcal{E}$.
2. Berechne auf dem Gitter $\mathcal{E}^{(i)}$ eine numerische Lösung $u_h^{(i)}$
3. Ermittle für alle $e \in \mathcal{E}^{(i)}$ die Elementfehlerschätzer η_e^2, das maximum $\eta_{max}^2 := \max_e \eta_e^2$ und die Summe $\eta^2 := \sum_e \eta_e^2$.
4. Falls $\eta > \varepsilon$
   (a) markiere alle Elemente zum verfeinern, welche $\eta_e^2 \geq \gamma \eta_{max}^2$ erfüllen.
   (b) Verfeinere das Gitter und erhalte $\mathcal{E}^{(i+1)}$
   (c) setze $i := i + 1$ und wiederhole Schritt 2.

Man kann zeigen, dass für einfache Probleme und genügend feinem Anfangsgitter, solche Verfahren konvergieren [7].
Bemerkung: *Bei Arbeiten mit Finiten Elementen muss also nicht nur das Makrogitter konform sein, sondern ebenfalls die lokale Verfeinerungsregel ein konformes Gitter erzeugen. Sonst ist eine separate Behandlung der Freiheitsgrade zu hängenden Knoten erforderlich (Interpolation). In Dune sind für ALUCubeGrid und ALUSimplexGrid keine konforme lokale Verfeinerung implementiert. Daher wird hierfür ein AlbertaGrid empfohlen.*

7 Hinweise zur Arbeitsumgebung

7.1 Programmier-Werkzeuge

Es folgen ein paar Hinweise auf empfohlene Programme zur Software-Entwicklung unter Unix/Linux. Die Grundlegenden Befehle zur Verwendung dieser Betriebssysteme finden sich in entsprechenden Kurzanleitungen, z.B. [15]. Für weitere Details der folgenden Programme siehe ebenfalls die zahlreichen Anleitungen im Internet.

emacs: Leistungsfähiger Texteditor mit C++ Modus. Dient als Oberfläche für externe Programme: Compiler, Debugger, Shell, etc. Bedienung vollkommen ohne Maus durch entsprechende Tastenkürzel möglich. Siehe auch die Reference-Card [10]. Beispiele:

M-x help : (Drücken von `esc` gefolgt von `x` gefolgt von Texteingabe `help` und abschließender Bestätigungstaste) Zeigt die Hilfefumktionalität des emacs

M-x apropos : Suchen von Kommandos anhand eines Suchwortes.

g++: GNU C++ Compiler, Erzeugung von Object-Dateien aus C++ Quelldateien, bzw. Linken von Objektdateien zu zu einem ausführbaren Programm. Beispiel:

```
g++ mytest.cc -o mytest
```

erzeugt ausführbares Programm mytest aus dem Quellcode mytest.cc. Wichtige Optionen:

-I/mein/pfad/zu/includes : Angabe von Verzeichnissen, in denen Header-Files gesucht werden sollen, welche mit `#include <myheader.hh>` im Programm verwendet werden.

-L/meine/lib : Spezifikation eines Pfades mit Bibliotheken

-g Bewahrt beim Kompilieren symbolische Informationen (Variablennamen, etc.), die späteres Debugging des erzeugten Programmes erlaubt.

-O1, -O2, -O3,... : Spezifikation eines Optimierungslevels. Höhere Nummer: Längere Compilierzeit, dafür schnelleres Programm.

make: Steuerung von Dateierzeugungsprozessen durch Regeln, die in einer Datei namens `Makefile` enthalten sind. Beispieleintrag einer Regel in einem Makefile:

```
mytest: my_main.o my_sub.o
        g++ my_main.o ma_sub.o -o mytest
```

Wichtig ist hier, dass die zweite Zeile ein Tab als erstes Zeichen vor dem Kommando enthält, und dies keine Leerzeichen sind. Die erste Zeile spezifiziert das Ziel (`mytest`), die notwendigen Eingabedateien (`my_main.o`,

`my_sub.o`). Die zweite (und eventuell weitere) Zeile spezifiziert den Befehl der ausgefuehrt wird, umd as Ziel zu erzeugen. Ein anschließendes `make mytest` überprüft das vorhandensein der Quellen `my_main.o` und `my_sub.o`, erzeugt diese gegebenenfalls durch weitere Regeln, und anschließend wird der Befehl zur Konstruktion des Ziels ausgeführt.

gdb: Gnu-Debugger: Schrittweise ausführen von Programmen, Variableninspektion, -Manipulation, etc. Wird über Kommandozeilen-Befehle gesteuert. Siehe auch die Reference Card [11].

ddd: Grafische Oberfläche für den gdb.

gnuplot: Gnu Visualisierungsprogramm. Beispiel: gegeben eine Textdatei `data.dat` mit x y Koordinatenpaaren pro Zeile, führt
`plot "data.dat" with lines`
zu einer stückweise linear interpolierten Darstellung der Daten.

paraview: Visualisierung von Gitter und Daten, siehe Abschnitt 7.6.

7.2 Remote Computing

Das Arbeiten mit Dune erfolgt auf eine der 64-Bit Linux-Rechner `scorpion` oder `vortex`.
Möglichkeiten des Arbeitens sind

ssh: Mittels `ssh -XC scorpion` wird eine Konsolen-basierte SSH-Verbindung mit Grafik-Weiterleitung und Kompression gestartet.

vnc:
- Auf dem remote Rechner (z.B. vortex) wird ein VNC-Server gestartet: `nice vncserver`. Ergebnis ist eine Display-Nummer, z.B. `vortex:1`. Ausloggen aus dem Rechner erfolgt durch `exit`.
- Auf dem lokalen Rechner (z.B. sklavenhaendler) wird ein VNC-Client gestartet, z.B. mittels `vncviewer vortex:1`. Dies öffnet die interaktive Arbeitsoberfläche auf dem Host. Alternative Clients sind `tightvnc, xtightvnc`.
- Auf dem remote Rechner muss der VNC-Server beendet werden: `vncserver -kill :1`.

Wichtig ist bei Arbeiten in einem Rechnernetz, dass Prozesse “geniced” werden. Dies bedeutet, dass allen Benutzern gemäß einer Prioritätenliste ein entsprechender Anteil der Prozessorzeit zugeteilt wird. Damit wird verhindert, dass Rechner durch Rechnungen einzelner Benutzer komplett blockiert werden. Statt

```
./my_prog my_arg1 my_arg2
```

werden rechenintensive Programme daher gestartet mit

```
nice ./my_prog my_arg1 my_arg2
```

Da Dune-Kompilate und Datenfiles sehr schnell sehr groß werden, existiert für alle Praktikumsteilnehmer ein Unterverzeichnis in `/share/numeric/src/dune_praktikum`. Die Quota ist hier wesentlich höher als in den beschränkten Home-Verzeichnissen. Es wird daher empfohlen, dort zu arbeiten.

7.3 Dune-grid

Das Kern-Modul Dune-grid stellt eine abstrakte Schnittstelle für hierarchische Gitter bereit. Hierdurch werden Daten und Algorithmen getrennt. Durch Gitterzugriff über die Schnittstellen-Methoden, können numerische Algorithmen mit Variation der Gitterimplementationen verwendet werden. Für weitere Details siehe die Online Dokumentation [9] unter `http://www.dune-project.org/doc-1.0/doxygen/html/classes.html` und das Grid-Howto [3] der Kursseite.

7.3.1 Gitter-Implementationen

Konkrete Implementationen der Gitter-Schnittstelle sind für einfache strukturierte Gitter implementiert, und es existieren Implementationen der Schnittstelle für verschiedene Gittermanager-Pakete:

Klasse	dim	adaptiv	parallel
SGrid	$\mathbb{N}$	-	-
YaspGrid	$\mathbb{N}$	-	X
OneDGrid	1	X	-
AlbertaGrid	2,3	X	-
ALUSimplexGrid	2,3	X	X
ALUCubeGrid	2,3	X	X
UGGrid	2,3	X	X

7.3.2 Wichtige Klassen

Einige wichtige Klassen sind die folgenden:

FieldVector: Vektor-Klasse mit mit Arithmetik, für kleine Vektoren, Dimension wird zur Compile-Zeit spezifiziert.

FieldMatrix: Matrix-Klasse mit mit Arithmetik, für kleine Matrizen, Dimension wird zur Compile-Zeit spezifiziert.

Entity: Die Entity Klasse enthält topologische Informationen über eine Entität eines Gitters. Erlaubt nur lesenden Zugriff.

Geometry: Die Geometry einer Entität enthält geometrische Informationen: Referenzabbildung, Koordinaten, Volumen, etc.

LeafGridPart: Blatt-Gitterteil des hierarchischen Gitters.

LevelGridPart: bestimmte Ebene als Teil des hierarchischen Gitters.

LevelIndexSet, LeafIndexSet: Konsekutive Numerierung von Entitäten eines Gridparts. Wichtig für Verwaltung von diskreten Funktionen.

GrapeGridDisplay: Klasse, welche die Visualisierung eines Gitters mit der Umgebung GRAPE erlaubt.

GrapeDataDisplay: Klasse, welche die Visualisierung eines Gitters mit Daten in der Umgebung GRAPE erlaubt.

VTKWriter: Ermöglicht das Schreiben eines Gitters im VTK-Format, um dies z.B. mit Paraview zu visualisieren.

VTKIO: Ermöglicht das Schreiben eines Gitters mit Daten im VTK-Format, um dies z.B. mit Paraview zu visualisieren.

7.3.3 Iteratoren

Da Gitter als Container von Elementen gesehen werden können, wurde eine STL-ähnlicher Zugriff auf Elementen/Entitäten realisiert. In der STL würde man z.B: eine Schleife über alle Einträge eines Vektors durchführen mit

```
vector<double> v(6);
for (vector<double>::iterator it=v.begin(); it!=v.end(); it++)
     cout << (*it) << " ";
```

Ähnliche Funktionalität haben Iteratoren in Dune, insbesondere Initialisierung und Abfrage des Endes durch entsprechende Methoden und Dereferenzieren eines Iterators für den Zugriff auf das Element. Einige Klassen sind

LevelIterator: erlaubt Iteration über die Elemente eines bestimmten Level eines Gitters.

HierarchicIterator: erlaubt Iteration über die Kinder eines Elementes

LeafIterator: erlaubt Iteration über die Blatt-Elemente eines Gitters.

IntersectionIterator: erlaubt Iteration über die Randentitäten, d.h. Codim 1 Entitäten eines Elementes. Ein IntersectionIterator ermöglicht Zugriff auf Gebietsrand-Information, Normalen, Nachbarelemente, etc.

Diese Gitterspezifischen Typen sind in einer Template-Struktur `Codim` im Gitter verfügbar. Beispiel Initialisierung eines LevelIterators:

```
GridType::template Codim<0>::LevelIterator
          lit = grid.template lbegin<0>(level)
```

Alternativ können die Iteratoren auch aus einem GridPart extrahiert werden

```
typedef GridPartType::Codim<0>::IteratorType IteratorType;
typedef GridPartType::IntersectionIteratorType
                                  IntersectionIteratorType;
```

Entsprechende Methoden sind dann `begin<0>()`, `end<0>()` und `ibegin(entity)`, `iend(entity)` auf dem Gridpart.

7.3.4 Gitter-Verfeinerung

Bei adaptiven Gittern gibt es im wesentlichen zwei Methoden zur Verfeinerung

a) Globale Verfeinerung: Methode `globalRefine(reflevel)` verfeinert ein Gitter `reflevel` mal.

b) Lokale Verfeinerung: Methode `mark(reflevel, entity)` markiert ein Element zum Verfeinern (`reflevel=1`) oder Vergröbern (`reflevel=-1`). Eine anschließende Adaption des Gitters erfolgt mit der `adapt` Methode des Gitters.

Für Details zu Gitteradaption, siehe Abschnitt 7 des Grid-Howto [3].

7.3.5 Dune Grid Parser

Für die Verschiedenen Gitter-Typen benötigte man zunächst individuelle Makrogitter-Dateien zur Initialisierung, z.B. `*.tetra`, `*.hexa` und `*.al` für ALUSimplexGrid, ALUCubeGrid und AlbertaGrid Gitter. Um hier eine Vereinheitlichung zu schaffen, wurde das Dune Grid Format eingeführt mit einem Parser, der diese Files einliest und in ein gewünschtes Gitterformat verwandelt. Eine Beispieldatei `cube.dgf`:

```
DGF
Interval
0   0 0              % first corner
1.0 1.0 1.0          % second corner
3   3  3             %3 cells in three directions

# now we define the boundary
BOUNDARYDOMAIN
default 1               % all other boundarys have id 1
2    0  0  0    0  1  1  % x = 0 -> id 2
3    1  0  0    1  1  1  % x = 1 -> id 3
```

Die erste Zeile identifiziert die Datei als Dune Grid Format. In weiteren Blöcken sind Inhalte definiert. Ein Interval-Block definiert eine äquidistante Zerlegung eines rechtwinkliges Parallelogramms. Alternativ kann man auch Punktelisten und Elemente durch die Punkte definieren. Die Randelemente können mit ganzzahligen Labels versehen werden. Eine Zeile in in einem Boundarydomain Block besteht aus einem Index und zwei Tupel. Hierdurch werden die Randelemente des Gitters, die in dem durch die beiden Tupel spezifizierten rechteckigen Bereich liegen, die erste Zahl in der Zeile als Markierung bekommen. Weitere Blöcke sind möglich, auch teilweise nur von bestimmten Gittern umsetzbar. Insbesondere sind vielfältige herkömmliche 3D-Datenformate verwendbar.
Einlesen eines solchen Files ist mittels eines Grid-Pointers möglich, wobei angenommen wird, dass GridType definiert ist:

```
Dune::GridPtr<GridType> gridptr(filename);
GridType& grid = *gridptr;
```

Für weitere Informationen zu dem Dune Grid Format und der Verwendung von GridPtr, siehe die Online-Dokumentation (Modules → I/O → Dune Grid Format).

7.4 Dune-fem

Dune-fem ist ein in Freiburg entwickeltes Dune-Modul, welches PDE-Diskretisierungskomponenten zur Verfügung stellt. Dies umfasst Klassen für Funktionen, Funktionenräumen, diskrete Funktionen, FEM/FV/LDG-Operatoren, lineare Gleichungsystemlöser, Quadraturen, etc. Dokumentation findet sich unter [8].

7.4.1 Diskrete Funktionen

Das Konzept von Funktionenräumen aus Abschnitt 4 ist in Dune umgesetzt. Die wichtigsten Klassen sind:

FunctionSpace: In Abhängigkeit von `DomainFieldType`, `RangeFieldType` und den Dimensionen d und n von Definitions- und Wertebereich wird hierdurch ein Funktionenraum definiert im Sinne von (10).

Function: Ist eine allgemeine Klasse, welche eine (analytische) Funktion aus einem FunctionSpace repräsentiert. Wichtigster Bestandteil ist eine `evaluate()` Methode.

LagrangeDiscreteFunctionSpace: Eine Implementation eines diskreten Funktionenraums, welches elementweise polynomial und global stetige Funktionen repräsentiert. Die Klasse benötigt als Template-Parameter den FunctionSpaceType, den GridPartType und eine Polynomordnung $p \geq 1$. Die lokalen Basisfunktionen sind Lagrange-Basisfunktionen, d.h. es ist eine nodale Basis, bei denen die DOFs direkt mit Funktionswerten an Lagrange-Knoten übereinstimmen. Diese Lagrange-Knoten sind ebenfalls verfügbar über den Typ `LagrangePointSetType` und der Methode `lagrangePointSet(entity)`. Ein LagrangePointSet hat Methoden `nop()` für die Anzahl der Punkte, und `point(i)` für Zugriff auf die Punkte.

DiscontinuousGalerkinSpace: Eine Implementation eines diskreten Funktionenraums, welches elementweise polynomiale Funktionen ohne Stetigkeitsbedingung repräsentiert. Die Klasse benötigt als Template-Parameter den FunctionSpaceType, den GridPartType und eine Polynomordnung $p \geq 0$. Die lokalen Basisfunktionen sind orthonormiert bezüglich der L^2-norm auf dem Referenzelement. Es gibt daher keine eindeutige Zuordnung von Funktionswerten und DOFs.

AdaptiveDiscreteFunction: Dies ist eine Implementation eines Diskreten Funktionstyps. Als einziger Template-Parameter wird der DiskreteFunctionType benötigt. Die Klasse stellt Speicherverwaltung der globalen DOFs und Unterstützung von Gittersadaptivität zur Verfügung.

BaseFunctionSet: Statt Auswertung von globalen Basisfunktionen φ_j, ist mit dieser Klasse Auswertung von lokalen Basisfunktionen $\hat{\varphi}_{\hat{e},i}$ mittels `evaluate(...)` und deren Ableitungen mittels `jacobian` möglich.

Der Rückgabetyp von letzterem ist `JacobianRangeType`, welches eine `FieldMatrix` ist, d.h. jede Zeile ein `FieldVector`. Ein Zugriff

auf das `BaseFunctionSet` eines Elementes ist durch die Methode `baseFunctionSet(entity)` des diskreten Funktionenraumes möglich.

Anlegen von Diskreten Funktionen

Die Template-Abhängigkeiten der Hilfsklassen implizieren bereits die Schritte zum Anlegen einer diskreten Funktion: Nach dem Initialieren eines Gitters und GridParts wird hierauf ein Funktionenraum und hiermit ein Diskreter Funktionenraum definiert. Bei Vorliegen einer Instanz eines diskreten Funktionenraumes `dfspace` kann eine diskrete Funktion einfach angelegt werden mittels `DiscreteFunctionType df("my_function",dfspace)`.

Zugriff auf Diskrete Funktionen

Im allgemeinen soll man globale Auswertungen von diskreten Funktionen vermeiden, weil dies immer mit einem teuren Gitter-Suchdurchlauf verbunden ist, in dem das Element zum Auswertepunkt bestimmt wird.
Problemlos ist eine Iteration über die globalen DOFs b_j in (13) zum Lesen und Schreiben. Hierzu gibt es in der diskreten Funktionsklasse einen `DofIteratorType` und die Methoden `dbegin()` und `dend()`. Zusätzlich ist die Abbildung $g(e,i)$ der lokalen in globalen DOF-Indizes in (14) realisiert durch die Methode `mapToGlobal(entity,i)` des diskreten Funktionenraumes.
Ein lokaler Zugriff ist über den Typ `LocalFunctionType` und der Methode `LocalFunctionType localFunction(entity)` der diskreten Funktion möglich. Eine LocalFunction erlaubt lesenden und schreibenden Zugriff auf die lokalen DOFs $a_{e,i}$ aus (14) durch den `operator[]`, wobei man einfach $i-1$ in der eckigen Klammer angibt (Zählung beginnt in C++ ja bei 0).
Eine localFunction,auf einem Element initialisiert, liefert auch die Möglichkeit mit der `evaluate` Methode eine lokale Auswertung einer diskreten Funktion durchzuführen. Die Koordinaten müssen dann auch lokale Koordinaten (d.h. bezüglich dem Referenzelement) sein.

7.4.2 Quadraturen

Die folgenden Dune-fem Klassen ermöglichen Integration auf Entitäten mittels Quadraturen, siehe Abschnitt 5.

- `CachingQuadrature<GridPartType,0>` kann für Elementintegration verwendet werden.
- `CachingQuadrature<GridPartType,1>` kann für Integration über Intersections verwendet werden. Diese Klasse enthält einen enum, der die Konstanten `INSIDE` und `OUTSIDE` definiert. Diese sind im Konstruktur der Quadratur zu verwenden, um anzugeben, ob man die Quadratur bzgl. dem innen oder außen liegenden Element orientieren will.

Die Methoden liefern die Quadraturinformationen:

- `nop()`: Anzahl der Quadraturpunkte n_p

- `point(i)`: Quadraturpunkt p_{i+1} , $i = 0, \ldots, nop() - 1$.
- `weight(i)`: Quadraturgewicht ω_{i+1}.

In den Integrationsformeln tauchen häufig bestimmte Ableitungen der Referenzabbildung auf. Diese stehen in der `Geometry` eines Elementes zur Verfügung:

- $|\det DF_e|$ erhältlich via Methode `integrationElement(...)`
- $((DF_e)^{-1})^T$ erhältlich via Methode `jacobianInverseTransposed(...)`

7.4.3 Operatoren

Übereinstimmend zur mathematischen Verwendung des Begriffs ist ein `Operator` in Dune eine Realisierung einer Abbildung zwischen Funktionenräumen.

- Die Template Parameter in der Deklaration `Operator< DFieldType, RFieldType, DType, RType >` spezifizieren die Typen der Eingangsfunktionen und der Ergebnisfunktionen.
- Durch die Methode `apply(arg,dest)` und `operator()(arg,dest)` wird der Operator auf eine Funktion `DType& arg` angewendet und das Ergebnis in `RType& dest` gespeichert.

Operatoren auf diskreten Funktionen, deren Anwendung durch einen Gitterdurchlauf mit elementweiser Operation beschrieben werden kann, können durch die Klassen `DiscreteOperator` und `LocalOperator` realisiert werden.

7.4.4 Iterative LGS-Löser

Die Klasse der *orthogonal error methods (OEM)* bezeichnet Verfahren zum Iterativen Lösen von Gleichungssystemen $\mathbf{Ax} = \mathbf{b}$. Der Bekannteste Vertreter ist das Konjugierte Gradienten (CG) Verfahren. Die folgenden Verfahren sind in Dune-fem als Operatoren realisiert, für Details verweisen wir auf [2, 6].

Lösungsoperator	$\mathbf{A}$ sym., p.d.	$\mathbf{A}$ non-sym, non-pd
`OEMCGOp`	ja	nein
`OEMBICGSTABOp`	ja	ja
`OEMBICGSQOp`	ja	ja
`OEMGMRESOp`	ja	ja

Verwendung dieser Klassen:

- `oemsolver.hh` einbinden
- Typdefinition des Lösers, z.B.

```
typedef OEMBICGSTABOp <DiscreteFunctionType,MyOperatorType>
        InverseOperatorType;
```

Hierbei ist `MyOperatorType` ein Klassentyp, der die Matrixmultiplikation $\mathbf{A}x$ realisiert. (Details zu Anforderungen siehe weiter unten).

- Initialisierung des Lösers z.B. mit

```
double redEps = 0.0, absLimit = 1e-15, maxIter=20000;
bool verbose = true;
InverseOperatorType solver(myOp, redEps, absLimit,
                           maxIter,verbose);
```

  wobei `myOp` eine existierende Instanz der Klasse `MyOperatorType` ist, `redEps` die relative Toleranz des Residuums, `absLimit` die absolute Lösungstoleranz des Residuums, `maxIter` die maximale Anzahl an Iterationen und `verbose` ein Flag zur Bildschirm-Detailausgabe.
  Achtung: Nicht alle Parameter sind in allen Lösern realisiert.

- Konkretes Lösen eines Gleichungssystems durch `solver(b,x)` zu einer gegebenen diskreten Funktion `b` und Ziel in der diskreten Funktion `x`.
  Achtung: Der vor dem Aufruf vorhandene Wert von `x` *dient zugleich als Anfangswert der Iteration. Eventuell ist also Null-Initialisierung sinnvoll.*

- Die Klasse `MyOperatorType` muss zur Verwendung mit den OEM-Methoden nicht notwendigerweise von einem `Dune::Operator` abgeleitet werden. Es reicht, wenn diese Klasse eine Methode

```
void multOEM(const double* & arg, double* & dest)
{
 ... // do your matrix multiplication
};
```

  und eine Methode

```
MyOperatorType& systemMatrix()
{
  return *this;
};
```

  besitzt.

7.5 GRAPE

GRAPE ist eine interaktive Visualisierungsumgebung, die durch eine funktionale Schnittstelle auf das durch die Numerik vorhandene Gitter und Daten zugreift. Daher erfolgt kein Speichern des Gitters, sondern an gewünschter Stelle im eigenen Programm, kann das Gitter interaktiv dargestellt werden. Für Details zu GRAPE siehe die Dokumentation [12].

7.5.1 Oberfläche

Es werden zwei Fenster geöffnet, der Manager und das Output-Fenster, siehe Fig. 4. Wichtigste Bestandteile des Managers sind die Reiter in der ersten Zeile:

manag: Haupt-Rubrik: Wahl von Visualisierungsmethoden (scene display method), Visualisierungsmodus (Switch → grid/patch/texture mode)

trans: Anwendung von geometrischen Transformationen auf Ausgabe: Objektposition, Ansichtsrichtung. Einstellung entweder über Schieber (ruler) oder interaktiv (plane, sphere). Praktisch ist hier "fit-to-window", um die Szene zu zentrieren.

opts: Dynamischer Bereich mit Optionen zu den momentanen Objekten, Visualisierungsmethoden, etc. Z.B. Wahl der darzustellenden Funktion, Colorbar, Interaktives Inspizieren des Gitters, Einstellen von Isolinien, etc. Die Vielzahl von Reglern wird über die Bereiche opt0 bis opt5 verteilt.

exit: Beenden von Grape, Kontrolle wird an die aufrufende Numerik-Anwendung zurückggegeben.

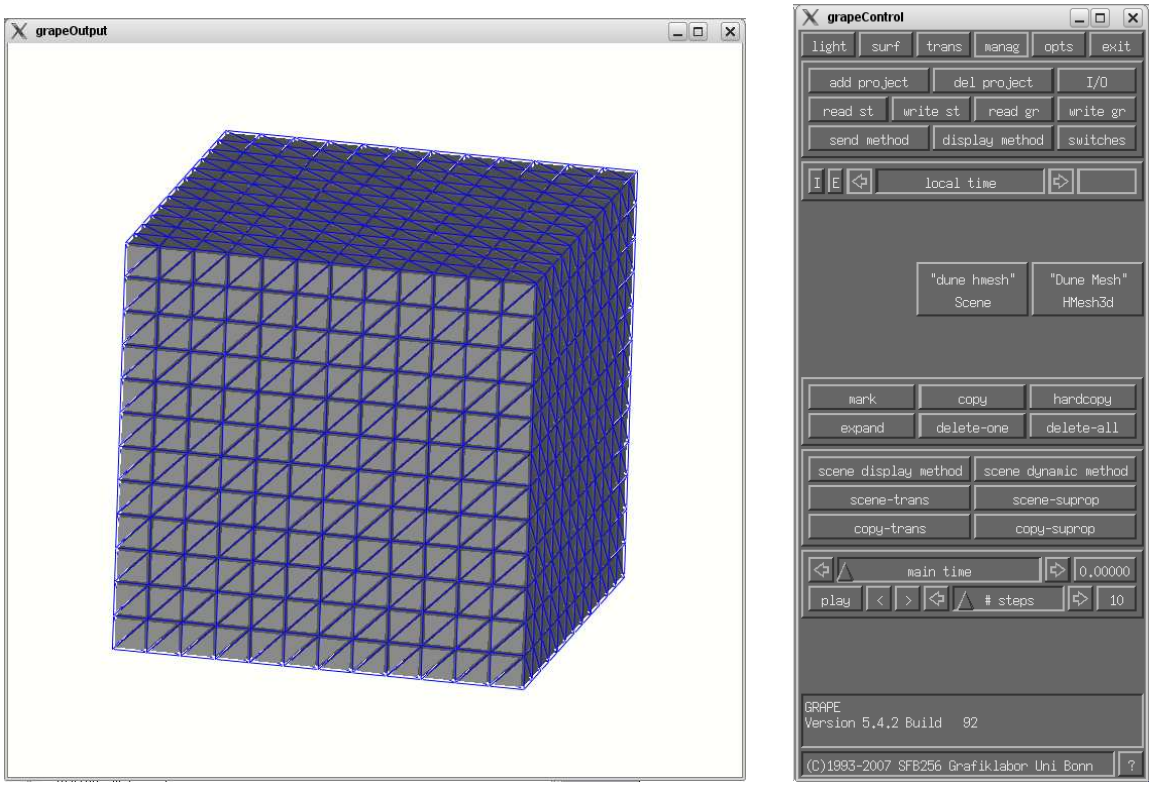

Abbildung 4: Grape Output und Manager Fenster.

7.5.2 Visualisierung eines Gitters

Eine Darstellung eines Dune-Gitters wird erreicht mit der GrapeGridDisplay Klasse. Nach Öffnen der Grape-Fenster ist ein Einstellen der Display Methode (scene display method → Doppelklick auf GenMesh3D) als "display" oder "inspect"

möglich. Anschließend kann unter “opts” entweder verschiedene Boundary-Ids zur Darstellung ausgewählt werden oder ein Durchschreiten des Gitters durchgeführt werden.

7.5.3 Visualisierung von Daten

Eine Darstellung von Dune Daten auf einem Gitter wird erreicht mit der Klasse `GrapeDataDisplay`. Mit der `addData(...)` Methode werden diskrete Funktionen hinzugefügt, mit `display()` schließlich die Visualisierung gestartet. Nach Öffnen der Grape-Fenster ist ein Einstellen der Display-Methode (scene display method → Doppelklick auf GenMesh3D) als “bnd-isoline” oder “clip-isoline” sinnvoll. Anschließend kann unter “opts” → HMesh3D: “Dune Mesh” im mittleren Pop-up-Menü die Funktion ausgewählt werden. Neben eigenen Datenfunktionen sind einige Standard-Funktionen vorhanden, z.B. einfach die Koordinatenkomponenten $f(x, y, z) = x$, etc. Die Visualisierungseinstellungen sind in einem entsprechenden Bereich unter opts regelbar, der Bereich wird entsprechend der Display-Methode benannt, z.B. “isoline“.

7.5.4 Voreinstellungen in .graperc Dateien

Beim Starten von Grape werden Einstellungsdateien gelesen, naemlich `~/.graperc` (allgemeine Einstellungen) und `./.graperc` (zusätzliche Problemabhängige Einstellungen) dort können Abkürzungen und globale Einstellungen gespeichert werden. Eine Beispiel-Datei ist

```
######## Settings of grape-window (does not work under linux)
grapeOutput =
{
 geometry = 800x800+100+100;
}

####### Settings of Manager
Manager = {

# general flags:
  log_events = 1;
  replay_delay = 1;
  replay_scale = 0;
  ask_on_exit = 0;

# Key-Shortcuts:
  hotkey = {
    key = "A-x";
#-- mouse speed: nice animation
    mouse_delay = 1;
#-- click: when no method exists: simulate a left mouse click
    click = "/main/top-row/exit";
  }

  hotkey = {
```

```
    key = "w";
#-- send: send method xxx-send to current object
    send = "fit2window";
    send = "draw-once";
  }

 hotkey = {
  key = "i";
  click = "/main/manag/scene/scene-display-method";
  typein = "isoline";
  click = "/scene-display-method/method/Ok";
 }
}
```

Hiermit werden insbesondere mit `w` ein "fit-to-window" und mit `i` ein "isoline" durchgeführt. Für weitere Informationen zu den Möglichkeiten dieser Resource-Dateien siehe `http://www.mathematik.uni-freiburg.de/IAM/Research/grape/DOC/HTML/node70.html` und `node400.html`.

7.5.5 Makros Aufzeichnen

Die Visualisierung in Grape ermöglicht eine Vielzahl von Einstellungen, die sich oft von Programmlauf zu Programmlauf wiederholen. Diese können aufgezeichnet werden, und beim Start automatisch durchgeführt werden. Hierzu definiert man in der `.graperc` Datei das flag `log_events=1`. Dies erzeugt während des Arbeitens mit Grape eine Datei `manager.*.log`, wobei `*` die Prozessnummer ist. Diese Datei wird nach Beenden von Grape wieder gelöscht. Um die Folge von Einstellungen vom Start von Grape bis zum momentanen Zustand zu speichern, kopiert man diese Datei in `manager.replay`. Diese Datei wird beim nächsten Start von Grape automatisch ausgeführt.

7.6 ParaView

Schritte zur Visualisierung mit ParaView, vgl. Fig. 5:

- Mit der Dune Klasse VTKWriter in wird ein Gitter in eine `*.vtu` Datei geschrieben.

  ```
  #include<dune/grid/io/file/vtk/vtkwriter.hh>
  ...
  Dune::VTKWriter<GridType> vtkwriter(grid);
  vtkwriter.write(vtk_outfn.c_str());
  ```

- Alternativ können zusätzliche Daten gespeichert werden, indem die Klasse VTKIO verwendet wird, z.B. Visualisierung von `df`:

  ```
  #include <dune/fem/io/file/vtkio.hh>
    ...
    Dune::VTKIO<GridPartType> vtkio( gridpart );
  ```

```
#if POLYDEG == 0
  vtkio.addCellData( df );
#elif POLYDEG == 1
  vtkio.addVertexData(df );
#endif
  vtkio.write(df.name().c_str());
#endif
```

- ParaView starten, z.B. liegt eine 64-Bit Linux-Version unter `/share/dune/Modules/modules_x86_64/paraview/bin/paraview`.
- File → Open und entweder das Gitter-File oder das Datenfile → `*.vtu` einladen.
- Im Object Inspector (Bereich links unten) → Display → "visible" aktivieren.
- Interaktive Ansicht des Gitters oder Daten im Grafikfenster.

Für Details zu ParaView siehe die Projekt-Webseite [14].

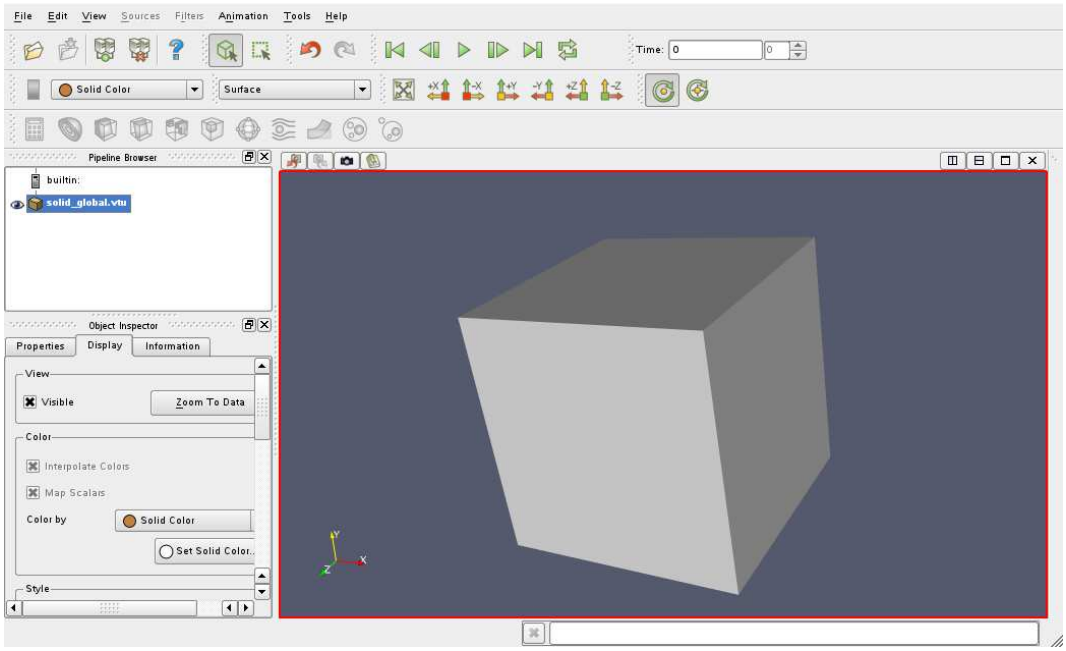

Abbildung 5: Paraview Visualisierungsoberfläche.

8 Hinweise zur Programmierung mit C++

Hier folgt eine unsortierte Liste von Hinweise zur Programmierung mit C++. Einiges ist hiervon ist im Grunde in C++ Programmier-Handbüchern auffindbar. Weiter enthält die Liste Empfehlungen zu Programmierstil, die sich insbesondere in dem Dune Projekt durchgesetzt haben.

8.1 Namensgebung

- Wir schreiben Klassennamen durchgehend gross, Instanzen einer Klasse klein. Methodennamen (außer Konstruktor und Destruktor) werden ebenfalls klein geschrieben.

```
// Klassendefinition
class MyClass
{
public:
 MyClass();
 void myMethod();
 ~MyClass();
};
// Objekt der Klasse:
MyClass myclass;
```

- Membervariablen bekommen grundsätzlich ein _ angehängt, damit man auf den ersten Blick in einer Methode sieht, was Membervariablen und was lokale Variablen sind. Auch ist eine Initialisierung der Membervariablen im Konstruktor dann sehr generisch machbar, weil die Einkommenden Variablen einfach identisch (nur ohne _) wie die Membervariablen gewählt werden können.

```
class MyClass
{
public:
   MyClass(const double a, const int t): a_(a), t_(t)
     {}
private:
   double a_;
   int t_;
};
```

8.2 Header Files

- Wir nennen C++Header-Files grundsätzlich `*.hh` zur Abgrenzung von C-Header Files (`*.h`).

- Zwecks Verhindern von Kompiler-Fehlermeldungen bei Mehrfach-Einbindung wird in Header-Files mit Defines gearbeitet. Voraussetzung ist

ein möglichst eindeutiger Bezeichner, der aus dem Dateinamen generiert werden kann. Beispiel `myheader.hh`:

```
// myheader.hh: example file
#ifndef MYHEADER_HH
#define MYHEADER_HH
...
// hier der eigentliche Header Code ...
...
#endif
```

So wird der Header-Code also genau einmal in einem Objectfile eincompiliert, unabhängig wieviele Quelldateien dieses Header-File einbinden.

- In Header Files sollten keine Komponenten enthalten sein, welche Object-Code erzeugen, wie z.B. Implementationen von Klassenmethoden, oder statische Datenstrukturen, etc. Falls diese Header-Datei von zwei Quelldateien eingebunden wird, diese beiden Quelldateien in einzelne Objectfiles kompiliert werden, und versucht wird, diese Objectfiles zu linken, wird der Linker einen Fehler erzeugen wegen doppeltem Vorhandensein von Implementationen. Stattdessen sollten diese Implementationen in einer separaten `*.cc` Datei erfolgen. Es ist jedoch möglich, Inline-Implementationen in Header Files zu halten, weil diese schließlich ohne Funktionskopf in die einzelnen Objectfilen hineincompiliert werden, also keine Probleme erzeugen.

8.3 Dynamischer Polymorphismus, Virtuelle Methoden

Objektorientierte Implementation einer einfachen Klassenhierarchie mit Hilfe von virtuellen Funktionen:

```
// Base class
class VectorInterface
{
  public:
    virtual void print() { cout << "Base class, no data!\n"; }
};

// Implementation derived from base class
class VectorImpl1: public VectorInterface
{
  public:
    virtual void print() {
      for (int i=0;i<50;i++) cout << data_[i] << " ";
      cout << "\n";
  }
  private:
    double data_[50];
};

// some routine that uses the interface
```

```
void do_something(VectorInterface& vec) {
  vec.print();
}

void main(..) {
  VectorImpl1 vec;
  do_something(vec);
}
```

Ohne die Schlüsselworte `virtual` würde die Ausgabe der Basisklasse erfolgen. Durch die Verwendung der virtuellen Routinen wird die korrekte `print()` Methode der abgeleiteten Klasse aufgerufen, trotz Verwendung der Schnittstellenklasse in `do_something()`. Solche virtuellen Aufrufe sind jedoch immer mit einem Nachschlagen eines Funktionspointers in einer Tabelle und einem Funktionsaufruf verbunden. Bei kleinen und häufig aufgerufenen Funktionen ist dies sehr teuer. Eine Möglichkeit, diese virtuellen Funktionen zu umgehen ist das CRTP im folgenden Abschnitt.

8.4 Statischer Polymorphismus, CRTP

Das *Curiously Recurrent Template Pattern* ermöglicht ein imitieren von dynamischer Bindung durch Template-Techniken ohne Verwendung von virtuellen Funktionen. Manchmal wird es auch (fälschlicherweise) als Barton-Nackman-Trick bezeichnet. Dasselbe Beispiel wie oben mit CRTP Technik:

```
// Base class "knowing the later derived class" as template argument
template <class VectorImp>
class VectorInterface
{
  public:
    // forwarding of interface method to the derived object
    void print() {
       asImp().print();
    }
  protected:
    // change of current object type from base class to derived class
    VectorImp& asImp() {
       return static_cast<VectorImp&>(*this);
    }
};

// Implementation derived from base class
class VectorImpl1: public VectorInterface<VectorImp1>
{
  public:
    void print() {
      for (int i=0;i<50;i++) cout << data_[i] << " ";
      cout << "\n";
  }
  private:
    double data_[50];
```

```
};

// some routine that uses an interface Routine
template <class VectorImp>
void do_something(VectorImp& vec) {
  vec.print();
}

void main(..) {
  VectorImpl1 vec;
  do_something(vec);
}
```

Die Basisklasse "kennt" die später abgeleitete Klasse in Form eines Template-Argumentes. Daher kann die "Sicht" auf ein vorliegendes Objekt der Basisklasse erweitert werden durch einen entsprechenden Cast. In der Interface-Klasse müssen alle Schnittstellen-Methoden weitergeleitet werden an die abgeleitete Klasse durch `asImp()`. Diese Technik ermöglicht dem Compiler, optimalen Code zu produzieren durch z.B. inlining. Der Geschwindigkeitsgewinn wird sichtbar, wenn entsprechend hohe Optimierungslevel beim Compilieren eingestellt sind.

8.5 Interface, Defaultimplementation und Implementation

Ein in Dune häufig auffindbares Programmmuster mittels CRTP ist folgende Zerlegung:

Interface: Eine Basisklasse deklariert eine Reihe von Schnittstellenmethoden, die eine abgeleitete und instanzierte Klasse implementiert haben muss. Diese Klasse wird selbst nicht instanziert.

DefaultImplementation: Einige der Schnittstellenmethoden können mit Hilfe von weiteren Schnittstellen-Methoden manchmal default-implementiert werden, d.h. eine funktionierende, aber eventuell langsame Version kann bereitgestellt werden. Eine hiervon abgeleitete Klasse kann diese eventuell durch effizientere Versionen ersetzen. Falls die DefaultImplementation-Klasse bereits alle Schnittstellenmethoden implementiert, ist sie instanzierbar. Meist wird aber weiter abgeleitet.

Implementation: Eine Spezialimplementation einer Schnittstelle kann von der DefaultImplementation-Klasse abgeleitet werden. Damit stehen die Default-Implementationen zur Verfügung oder können überladen werden. Diese Klasse muss die noch nicht implementierten Schnittstellenmethoden bereitstellen, damit Objekte instanziert werden können.

Ein Beispielprogramm ist `crtp.cc` auf der Kursseite.

8.6 Zeitmessung in C++

Mittels der `ctime` Bibliothek kann Laufzeit sehr einfach gemessen werden

```
#include <ctime>
...
clock_t start = clock();
... // do some computations
clock_t finish = clock();
double time = (double(finish)-double(start))/CLOCKS_PER_SEC;
```

8.7 Typdefinitionen

Die Verwendung von Typdefinitionen erleichtert spätere Austauschbarkeit von Klassentypen, indem nur an einer Stelle eine Typdefinition geändert werden muss und nicht an zahlreichen Stellen. Dieses Prinzip ist bei Templatebasierter Programmierung sehr zu empfehlen.

```
typedef VectorImp1 VectorType;
\\ typedef VectorImp2 VectorType;
VectorType vec;
Matrix<VectorType> mat;
```

8.8 Assertions

Ein sehr praktisches Konzept zum Debuggen ist die Verwendung von sogenannten Assertions in einem Programm. Durch einbinden von <assert.h> kann man an beliebigen Programmstellen überprüfen, ob bestimmte Bedingungen erfüllt sind. Beispiel:

```
double* meinpointer = new double[10000000]
// Test der Initialisierung vor dem Schreiben:
assert(meinpointer != 0);
meinpointer[500] = 10.0
```

Die Assertions werden zur Laufzeit überprüft. Ist die Assertion erfüllt, läuft das Programm einfach weiter. Ist die Assertion nicht erfüllt, bekommt man eine Fehlermeldung: “assert ’meinpointer != 0’ failed” welche wesentlich informativer ist, als ein nichtssagendes “Segmentation Fault”. Hierdurch findet man die Stelle im Programmcode sehr schnell.
Ist ein Programm lauffähig, ohne dass es Abbrüche durch Assertions gibt, kann man diese alle ausschalten durch das Precompiler-flag `#define NDEBUG` und anschließendem Neucompilieren. Die Empfehlung lautet daher, solche asserts(...) in beliebiger ausgiebiger Anzahl in eigenen Programmen verwenden.

Literatur

[1] M. Ainsworth and J.T. Oden. *A Posteriori Error Estimation in Finite Element Analysis*. Wiley Interscience, 2000.

[2] S.F. Ashby, T.A. Manteuffel, and P.E. Saylor. A taxononmy for conjugate gradient methods. *SIAM J Numer Anal*, 27:1542–1568, 1990.

[3] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. The distributed and unified numerics environment (dune) grid interface howto, 2008, http://www.dune-project.org/doc/grid-howto/grid-howto.pdf.

[4] D. Braess. *Finite Elemente*. Springer, 2003.

[5] A. Dedner and M. Ohlberger, Skriptum zur Vorlesung Wissenschaftliches Rechnen SS06, Universität Freiburg, 2006.

[6] W. dörfler: Orthogonale fehler-methoden. universität freiburg, http://www.mathematik.uni-freiburg.de/iam/homepages/willy/papero1.html, 1997.

[7] W. Dörfler. A convergent adaptive algorithm for poisson's equation. *SIAM J. Numer. Anal.*, 33:1106–1124, 1996.

[8] DUNE-fem Projektwebseite: http://dune.mathematik.uni-freiburg.de.

[9] DUNE Projektwebseite: www.dune-project.org.

[10] Emacs reference card, http://refcards.com/docs/gildeas/gnu-emacs/emacs-refcard-a4.pdf.

[11] Gdb reference card, http://www.digilife.be/quickreferences/qrc/gdb%20quick%20reference.pdf.

[12] GRAPE webseite: http://www.mathematik.uni-freiburg.de/iam/research/grape/doc/html/manual.html.

[13] D. Kröner. *Numerical Schemes for Conservation Laws*. John Wiley & Sons and Teubner, 1997.

[14] ParaView webseite: www.paraview.org.

[15] Unix reference card, http://comp.chem.umn.edu/chem8021/unix_ref_card.pdf.

[16] R. Verfürth. *A review of a posteriori error estimation and adaptive mesh-refinement techniques*. Wiley-Teubner, 1996.