Praktikum Wissenschaftliches Rechnen

Stefan Girke

Wintersemester 2012/13

Zum Lösen der Aufgaben im Praktikum „Wissenschaftliches Rechnen“ werden viele verschiedene Verfahren, Konzepte und Software-Pakete benötigt. Dieses Skript soll alle Informationen bündeln und eine Orientierungshilfe beim Bewältigen der Aufgaben sein. Es handelt sich um kein vollständiges Skript, sondern nur um eine kurze Zusammenfassung der im Praktikum vorgestellten Konzepte. Dieses Skript ist eine Überarbeitung und Ergänzung des Skriptes [11], gehalten von Bernard Haasdonk im Wintersemester 2008/2009. Auch in diesem Skript stecken noch ein paar Fehler. Korrekturen, Kommentare und Verbesserungsvorschläge zum Skript sind immer willkommen.

Inhaltsverzeichnis

Kapitel 1

Hierarchische und adaptive Gitter

Möchte man eine Differentialgleichung auf einem beschränkten, polygonalen Gebiet $\Omega \subseteq \mathbb{R}^d$ mit $\dim(\Omega) = d$ lösen, so benötigt man für viele Diskretisierungsmethoden (Finite Elemente, Finite Differenzen, Finite Volumen...) eine Aufteilung des Gebietes in kleinere und „einfachere„ Teilgebiete. Im folgenden Kapitel soll ein kurzer Einblick in häufig verwendete Begriffe und Problemstellungen beim Erstellen von Gittern gegeben werden.

1.1 Referenzelement

Definition 1.1.1 (Referenzelement)**.** Ein **Referenzelement** $\hat{e} \subset \mathbb{R}^d$ ist ein konvexes Polytop, d.h. beschränkter Schnitt endlich vieler Halbräume.

Eine Menge von Referenzelementen bezeichnen wir mit $\hat{\mathcal{E}} := \{\hat{e}_i\}_{i=1}^R$.

Beispiel 1.1.2 (Referenzelement)**.** Einheitsquadrat, Einheitsdreieck, Einheitswürfel, Einheitsintervall...

Definition 1.1.3 (Entität, Referenzabbildung)**.** Eine **Entität** $e \subset \mathbb{R}^d$ ist das Bild eines Referenzelementes $\hat{e} \in \hat{\mathcal{E}}$ unter einer diffeomorphen Abbildung F_e (der **Referenzabbildung**).

Definition 1.1.4 (Subentitäten, Dimension, Kodimension, Elemente)**.** Die **Dimension** einer Entität ist $\dim \hat{e} = \dim e$. **Subentitäten** einer Entität sind die Randflächen/Kanten/Eckpunkte usw. Die **Kodimension** einer Subentität e ist $\operatorname{codim} e := d - \dim e$. Kodim-0-Entitäten bezeichnet man auch als **Elemente**.

1.2 Hierarchische Gitter

Definition 1.2.1 (Hierarchische 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

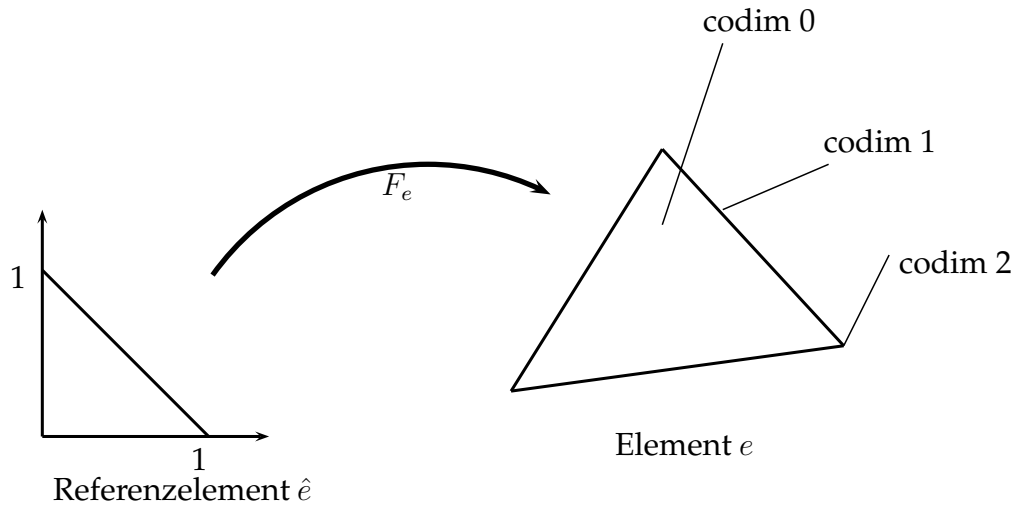


Abbildung 1.1: Referenzelement, Referenzabbildung und Subentitäten

gilt:

- die Level-0-Elemente überdecken Ω, d.h. $\bigcup_{e\in\mathcal{E}_0} e = \overline{\Omega}$,
- Elemente eines Levels überlappen sich nicht, d.h. für alle l gilt $\mathring{e} \cap \mathring{e'} = \emptyset$ für $e \neq e'$, $e, e' \in \mathcal{E}_l$,
- für alle $l > 0$, $e \in \mathcal{E}_l$ existiert ein **Eltern-Element** $e' \in \mathcal{E}_{l-1}$ mit $e \subseteq e'$, e heißt umgekehrt **Kind-Element**
- Jedes Element mit mindestend einem Kind-Element zerfällt vollständig in Kind-Elemente.

Elemente ohne Kind-Elemente nennt man Blatt-Elemente.

1.3 Gitterteile

Da man für konkrete Numerik meist nicht die komplette Gitterstruktur benötigt, beschränkt man sich auf **Gitterteile** (`GridParts`).

Definition 1.3.1 (Makroelemente)**.** Das Level-0-Gridpart besteht aus den Elementen ohne Eltern, den sogenannten **Makroelementen**.

Definition 1.3.2 (Level-Gridpart)**.** Sei $l \in \mathbb{N}_0$. Dann besteht das **Level-l-Gridpart** aus den Elementen auf Level l, d.h. $\mathcal{E}_l$.

Definition 1.3.3 (Level-Gridpart)**.** Sei $l \in \mathbb{N}_0$. Dann besteht das **Level-l-Gridpart** aus den Elementen auf Level l, d.h. $\mathcal{E}_l$. Mit $\mathcal{E}^{\text{level},l}$ bezeichnen wir die Vereinigung aller Elemente auf dem Level l.

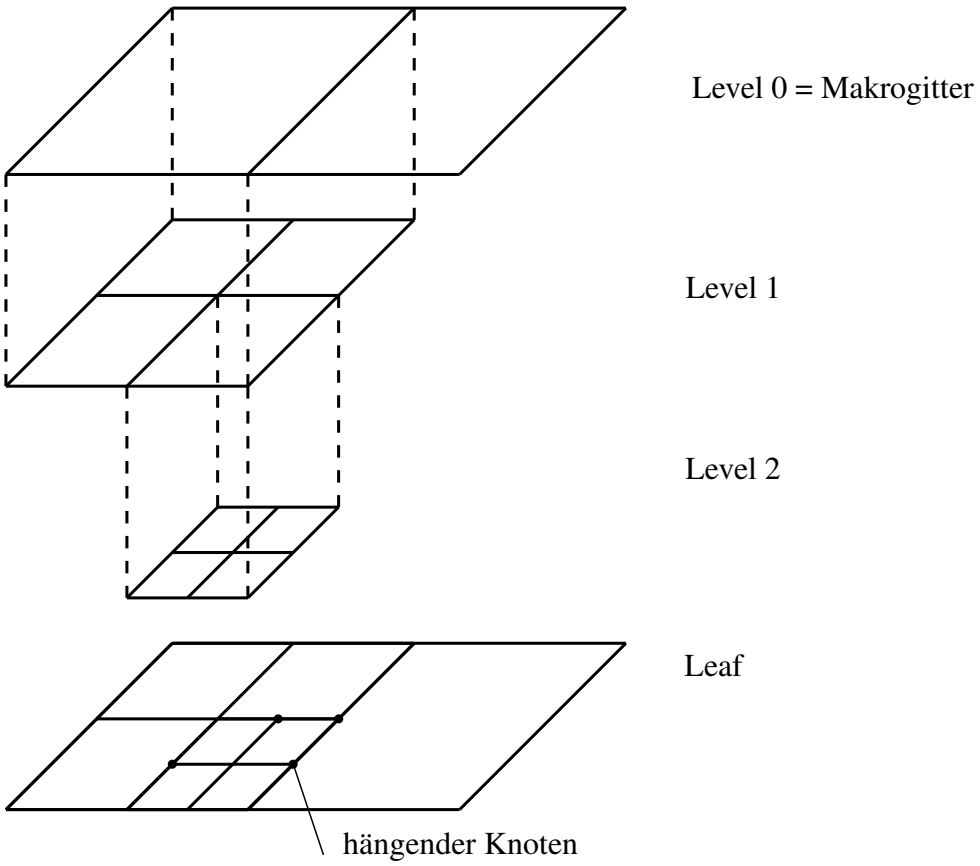


Abbildung 1.2: Hierarchisches Gitter mit Leaf- und Level-Sicht

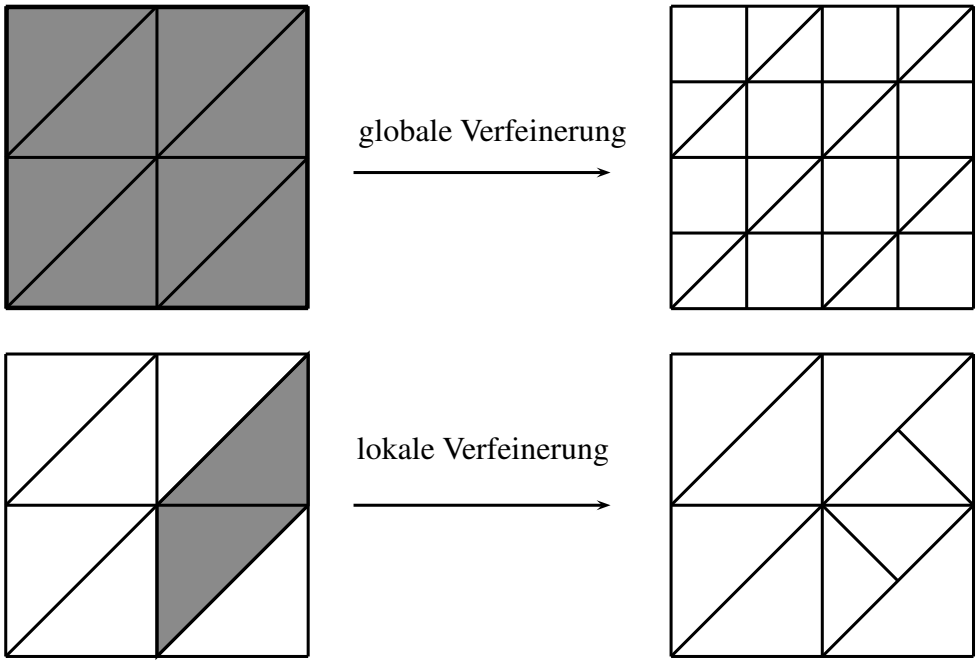


Abbildung 1.3: globale/lokale Verfeinerungen

Definition 1.3.4 (Leaf-Gridpart)**.** Das **Leaf-Gridpart** besteht aus den Blatt-Elementen aller Level. Mit $\mathcal{E}^{\text{leaf}}$ bezeichnen wir die Vereinigung aller Blatt-Elemente.

Definition 1.3.5 (Hängender Knoten)**.** Ein **hängender Knoten** ist eine Kodim-d-Entität, die nicht auf dem Rand einer Kodim-$d-1$-Entität des Leaf-Gridparts liegt.

1.4 Indexmengen und lokale Verfeinerungen

Viele Diskretisierungsmethoden sind darauf ausgelegt, dass Informationen, die lokal auf einer Entität im Gitter gespeichert sind, auf einen globalen Kontext abgebildet werden müssen. Dazu benötigt man Indexmengen, die jeder Entität eine global eindeutige Nummer zuweisen. Indexmengen können sich auf verschiedene Mengen von Entitäten beziehen. Während das Erstellen einer Indexmenge für ein Makrogitter einfach ist, benötigt man für hierarchische Gitter, die lokal verfeinert werden können, zusätzliche Funktionalität, um die Eigenschaften der Indexmenge nicht zu zerstören. Wir definieren mit

$$\mathcal{E}^c := \{e \in \mathcal{E} \mid e \text{ hat die Kodimension } c\}$$

die Menge aller Kodim-c-Entitäten eines Gitters. Analog definieren wir

$$\mathcal{E}^{c,\text{leaf}} := \mathcal{E}^c \cap \mathcal{E}^{\text{leaf}}$$

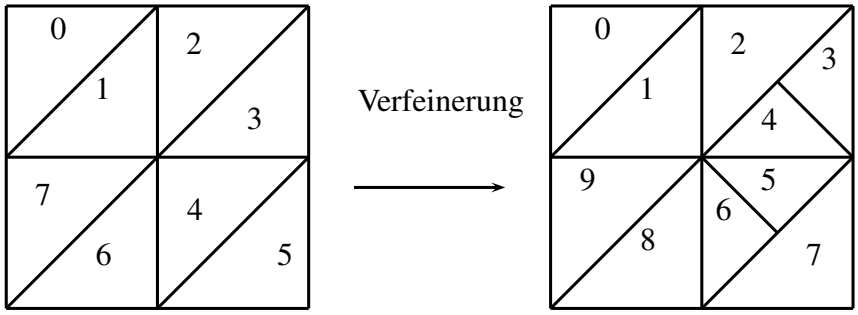


Abbildung 1.4: Beispiel einer Indexmenge für Elemente eines Leaf-Gridparts vor und nach einem Adaptionsschritt

und

$$\mathcal{E}^{c,\mathrm{level},l} := \mathcal{E}^{c} \cap \mathcal{E}^{\mathrm{level},l}.$$

Definition 1.4.1 (Indexmengen für Gitter, Indexabbildungen)**.** Indexmengen definieren eine Indexabbildung $m : \mathcal{E} \to \mathbb{N}_0$, die eine Entität eines Gitters auf die natürlichen Zahlen abbildet. Die Indexabbildung m hat folgenden Eigenschaften:

- sie ist eindeutig innerhalb der Untermenge $\mathcal{E}^c$,
- sie ist konsekutiv und startet bei 0, d.h. $0 \leq m(e) < |\mathcal{E}^c|$.

Definition 1.4.2 (Indexmengen für Level- und Leaf-Gridparts)**.** Analog definiert man Indexmengen für das Level- und Leaf-Gridpart:

- Die Indexabbildung $m : \mathcal{E}^{\mathrm{leaf}} \to \mathbb{N}_0$ bzw. hat folgenden Eigenschaften:
  - sie ist eindeutig innerhalb der Untermenge $\mathcal{E}^{c,\mathrm{leaf}}$,
  - sie ist konsekutiv und startet bei 0, d.h. $0 \leq m(e) < |\mathcal{E}^{c,\mathrm{leaf}}|$.
- Die Indexabbildung $m : \mathcal{E}^{\mathrm{level},l} \to \mathbb{N}_0$ bzw. hat folgenden Eigenschaften:
  - sie ist eindeutig innerhalb der Untermenge $\mathcal{E}^{c,\mathrm{level},l}$,
  - sie ist konsekutiv und startet bei 0, d.h. $0 \leq m(e) < |\mathcal{E}^{c,\mathrm{level},l}|$.

Bemerkung 1.4.3 (Indexmengen und lokale Verfeinerungen)**.** Mit jedem Verfeinerungsschritt ändert sich die Indexmenge.

Kapitel 2

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 6.2.

2.1 Funktionenräume

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

$$V := \{u : (\mathbb{K}_D)^d \to (\mathbb{K}_R)^n\}. \tag{2.1}$$

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`) $\mathbb{K}^{n\times d}$.

2.2 Basisfunktionen auf Referenzelementen

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

$$B_{\hat{e}} := \{\hat{\varphi}_{\hat{e},1}, \ldots, \hat{\varphi}_{\hat{e},m}\} \tag{2.2}$$

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.

2.3 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 $g : \mathcal{E} \times \{1,\dots,m\} \to \{1,\dots,N\}$ gegeben ist. Hierdurch werden globale Basisfunktionen $\varphi_j : \Omega\backslash\Gamma \to \mathbb{K}_R^n, j = 1,\dots,N$ definiert durch

$$\varphi_j := \sum_{(e,i):g(e,i)=j} \hat{\varphi}_i \circ F_e^{-1}. \tag{2.3}$$

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

$$V_h := \left\{ u_h \in V : \quad u_h = \sum_{j=1}^{N} b_j \varphi \right\} \tag{2.4}$$

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

$$V_h = \left\{ u_h \in V : \quad u_H|_e = \sum_{i=1}^{m} a_{e,i}\hat{\varphi}_{\hat{e},i} \circ F_e^{-1} \text{ mit } a_{e,i} = b_{g(e,i)} \quad \forall e \in \mathcal{E} \right\} \tag{2.5}$$

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.

Kapitel 3

Quadraturen

3.1 Integration über Gebiet

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

$$\int_\Omega f(x)dx = \sum_{e\in\mathcal{E}} \int_e f(x)dx, \qquad \int_{\partial\Omega} f(x)ds(x) = \sum_{e\in\mathcal{E}} \int_{e\cap\partial\Omega} f(x)ds(x). \quad (3.1)$$

3.2 Integration über Gitterelemente

Integration über Elemente wird auf Referenzelemente zurückgeführt durch Transformationssatz und Kettenregel

- Beispiel: globale Basisfunktionen:

$$\int_e f(x)\varphi_j(x)dx = \int_{\hat{e}} f(F_e(\hat{x}))\hat{\varphi}_i(\hat{x})|\det DF_e|d\hat{x} \quad (3.2)$$

  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

$$\int_e v(x)^T \nabla \varphi_j = \int_{\hat{e}} v(F_e(\hat{x}))^T (J(\hat{x}) \nabla_{\hat{x}} \hat{\varphi}_i) |\det DF_e| d\hat{x}. \tag{3.3}$$

3.3 Approximation durch Quadraturen

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

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

Kapitel 4

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, 12, 5].

4.1 Elliptisches Problem

Sei $\Omega \subset \mathsf{R}^w$ 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

$$\begin{aligned} -\nabla \cdot (a(x)\nabla u(x)) &= f(x) \quad \text{in } \Omega \\ u(x) &= g_D(x) \quad \text{auf } \Gamma_D \\ a(x)\nabla u(x) \cdot n(x) &= g_N(x) \quad \text{auf } \Gamma_N \end{aligned} \tag{4.1}$$

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 $\varphi|_{\Gamma_D} = 0$

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

4.2 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

$$\int_\Omega a(x)\nabla u(x)\cdot\nabla\varphi(x)dx = \int_\Omega f(x)\varphi(x)dx + \int_{\Gamma_N} g_N(x)\varphi(x)ds(x) \quad \forall\varphi \in V(0). \tag{4.2}$$

4.3 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 $\hat{\varphi}_i \in \mathcal{P}_1(\hat{e})$ 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:

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

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

$$\int_\Omega a(x)\nabla u(x)\cdot\nabla\varphi(x)dx = \int_\Omega f(x)\varphi(x)dx + \int_{\Gamma_N} g_N(x)\varphi(x)ds(x) \quad \forall\varphi \in V_h(0). \tag{4.3}$$

4.4 Lineares Gleichungssystem

Wegen Linearität von (4.3) 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

$$\int_\Omega a(x)\sum_j b_j\nabla\varphi_j\cdot\nabla\varphi_i dx \;=\; \int_\Omega f\varphi_i dx + \int_{\Gamma_N} g_N\varphi_i ds(x) \quad \text{für } v_i \notin \Gamma_D. \tag{4.4}$$

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

$$b_i = g_D(v_i) \quad \text{für } v_i \in \Gamma_D. \tag{4.5}$$

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 $v_i \in \Gamma_D$:

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

4.5 Algorithmische Aspekte

4.5.1 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.

4.5.2 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 $g_D(v_i)$.

4.5.3 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.

4.6 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

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

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 $\gamma = 0.5$.

(i) Starte mit $i = 0$ und dem vorgegebenen Gitter $\mathcal{E}^{(0)} := \mathcal{E}$.

(ii) Berechne auf dem Gitter $\mathcal{E}^{(i)}$ eine numerische Lösung $u_h^{(i)}$

(iii) 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 $\eta^2 := \sum_e \eta_e^2$.

(iv) 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.

Kapitel 5

Programmieren unter Linux

5.1 Editoren

Welcher Editor unter Linux zum Programmieren verwendet wird, hängt im Wesentlichen von den Vorlieben des Programmierers ab. Möchte er in der Konsole arbeiten oder benötigt er eine komplette Gui? Hier sollen kurz ein paar Editoren mit ihren Vor- und Nachteilen vorgestellt werden. In diesem Skript wird zum Editieren von Textdateien vim verwendet.

5.1.1 vim

vim ist ein leistungsstarker, konsolenbasierter Editor, der dazu ausgelegt ist, Befehle mit möglichst wenig Tastenanschlägen zu erreichen. Ungewohnt sind für Anfänger die unterschiedlichen Befehlsmodi, denn nur im Editiermodus kann man seinen Text editieren:

Befehlsmodus vim startet automatisch im Befehlsmodus. Mit `i` wechselt man in den Editiermodus, mit `:` in die Kommandozeile und mit `v` in den visuellen Modus. Ziel ist es, immer wieder in den Befehlsmodus zu kommen (mit Esc).

Editiermodus Hier lassen sich wie gewohnt Editieroptionen am Text durchführen.

Kommandozeilenmodus Dies ist eine Unterform des Befehlsmodus.

Visueller Modus Zum Markieren von Textstellen.

Zum Speichern und Schließen tippt man
`:w`
und
`:q`

Man kann in der Datei `.vimrc` alle persönlichen Einstellungen speichern. Hier sind eine kurze Auswahl von nützlichen Einstellungen. Es gibt wesentlich mehr.

```
"syntax highlighting
syntax on
set showmatch
set showcmd
"indention
set autowrite
set smartindent
"incremental search
set incsearch
" Better command-line completion
set wildmenu
" Use case insensitive search, except when using capital letters
set ignorecase
set smartcase
" Allow backspacing over autoindent, line breaks and start of insert action
set backspace=indent,eol,start
" Display the cursor position on the last line of the screen or in the status
" line of a window
set ruler
" Enable use of the mouse for all modes
set mouse=a
" Display line numbers on the left
set number
" number of blanks for a tab
set tabstop=2
" mininmal distance between cursor and border while scrolling
set scrolloff=2
" indicator for a new line when a line is wrapped
set showbreak==>\ \ \
"global search
set hlsearch
```

Nur, wer die Default-Einstellungen von vim ändert, kann sich über die gesamte Leistungsfähigkeit von vim bewusst werden. Wer eine Gui für vim benutzen möchte, sollte gvim ausprobieren.
Ein wichtiger Vorteil von vim ist das flüssige Arbeiten per ssh.

5.1.2 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.

Ein wesentlicher Nachteil von emacs sind die etwas weniger intuitiven Tastenkürzel. Möchte man remote arbeiten, so benötigt man eine gute Verbindung, um flüssig arbeiten zu können.

5.1.3 qtcreator

5.2 Versionkontrollsysteme

Jeder, der an einem Projekt für längere Zeit gearbeitet hat, kennt das Problem: Aus kleinen Projekten können schnell große, unübersichtliche Projekte werden. Möchte man neue Ideen in solche Projekte einbauen, so passiert es häufig, dass man beim Implementieren die neue Idee verwerfen muss. Hier bieten Versionskontrollsysteme die einfache Möglichkeit, alte Zustände wieder herzustellen.

5.2.1 git

git kann mehr als nur Versionkontrolle: Es lassen sich mehrere Ideen (“Zweige“) parallel entwickeln, Projekte lassen sich einfach auf Servern sichern und das gemeinsame Arbeiten mehrere Programmierer an einem Projekt wird erleichtert.
Man kann zum einen git über die Konsole oder mit einer Gui wie z.B. gitk, git gui oder git cola arbeiten.
Zu git gibt es viele Tutorials, z.B. ist http://git-scm.com/book für Anfänger sehr gut geeignet. Es sollen im folgenden die wichtigsten Grundlagen von git kurz vorgestellt werden. Dies ist weder ein Tutorial noch vollständig.

Ein erstes Projekt

Um die Funktionsweise von git zu verstehen, sollte man zunächst lokal auf seinem Rechner einen Ordner erstellen und den mit git verwalten, d.h. man gibt die Befehle

```
mkdir myfirstgit
cd myfirstgit/
```

`git init`

ein. Beim letzten Befehl erhält man die Ausgabe

`Initialized empty Git repository in /myfirstgit/.git/`

oder eine ähnliche Meldung. Wir haben somit unser erstes git repository erstellt. Mit

`ls -a`

listet man alle (auch versteckte) Dateien und Verzeichnisse in dem aktuellen Ordner auf. Es sollte die Ausgabe

`. .. .git`

erscheinen. In diesem Ordner liegen also nun alle benötigten Daten, um git starten zu können. Hier sollte man besser nichts ändern, wenn man nicht weiß, was man tut. Nun legt man eine Testdatei an, um git zu testen

`touch test`

und schreibt etwas in die Datei.

Dateien werden nicht automatisch zur Versionskontrolle hinzugefügt, sondern müssen manuell hinzugefügt werden. Mit

`git status`

zeigt man an, welche Dateien im Ordner `myfirstgit` nicht unter Versionkontrolle stehen, welche geändert wurden. In unserem Fall erhalten wir

```
# On branch master
#
# Initial commit
#
# Untracked files:
# (use “git add <file>...“ to include in what will be committed)
#
# test
```

Betrachten wir den unteren Abschnitt, so sehen wir, dass die Datei `test` nicht unter Versionkontrolle steht. Wir folgen obigem Vorschlag und rufen

`git add test`

auf. Nun liefert `git status`

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
# (use “git rm -cached <file>...“ to unstage)
#
# new file: test
```

Die Datei `test` steht nun unter Versionkontrolle. Zum Rückgängig machen ruft man

(wo wie es da steht)

```
git rm -cached test
```

auf. Die Datei steht jetzt zwar unter Versionskontrolle, aber die Änderungen an der Datei wurden noch nicht eingetragen. Wichtig: Nur wenn eine Änderung eingetragen ist, kann man zu einem späteren Zeitpunkt wieder in diesen Zustand wechseln.

```
git commit -m ‘‘my first commit‘‘
```

macht die Eintragung und gibt dieser Eintragung den Namen „my first commit“. Die Anwort ist

```
[master (root-commit) 5ca2714] my first commit
1 file changed, 1 insertion(+)
create mode 100644 test
```

oder ähnliches. Nun liefert

```
git status
```

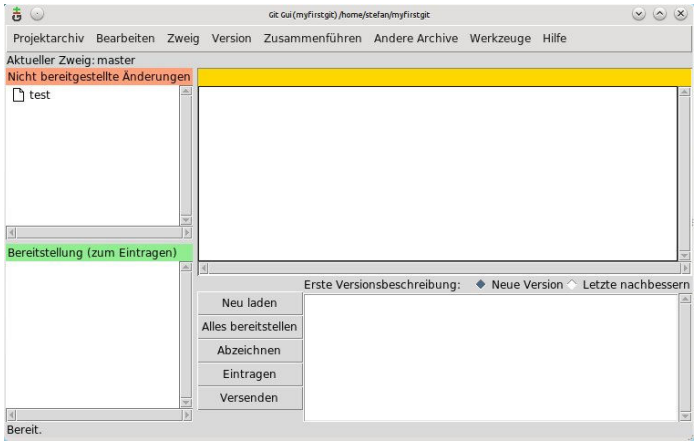
das Ergebnis

```
# On branch master nothing to commit (working directory clean)
```

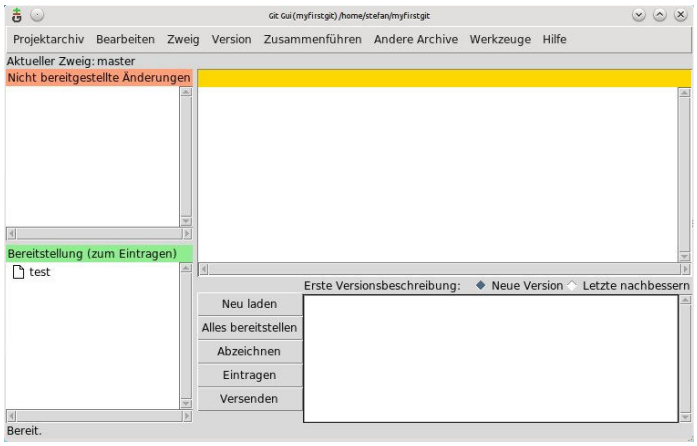
Das heißt, dass alle Änderungen eingetragen worden sind. Wer lieber in einer Gui arbeiten möchte, ruft nach dem Erstellen des git-Repositories mit

```
git gui
```

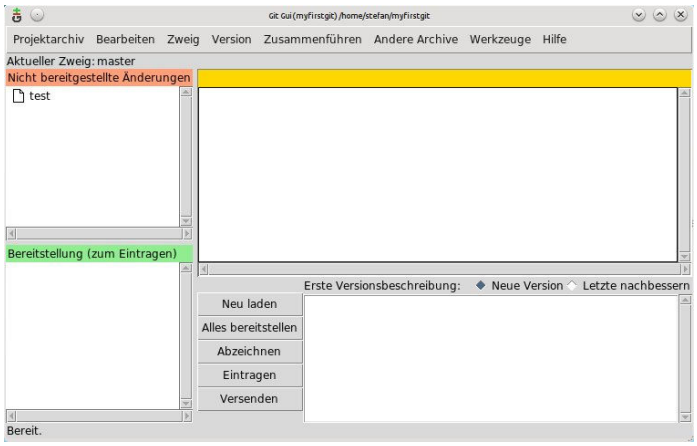
die Gui auf.



Mit einem Klick auf die Datei „test“ links oben, stellt man die Datei unter Versionskontrolle. Das sieht dann so aus:



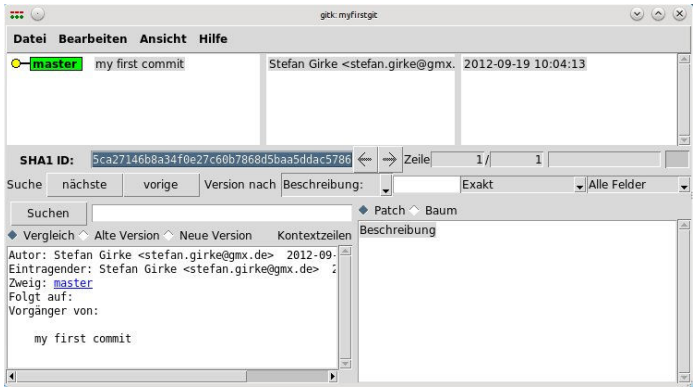
Nun trägt man eine Nachricht in das Textfeld unten rechts ein und klickt auf „Eintragen“. Man erhält wieder:



Schließt man das Programm und ruft

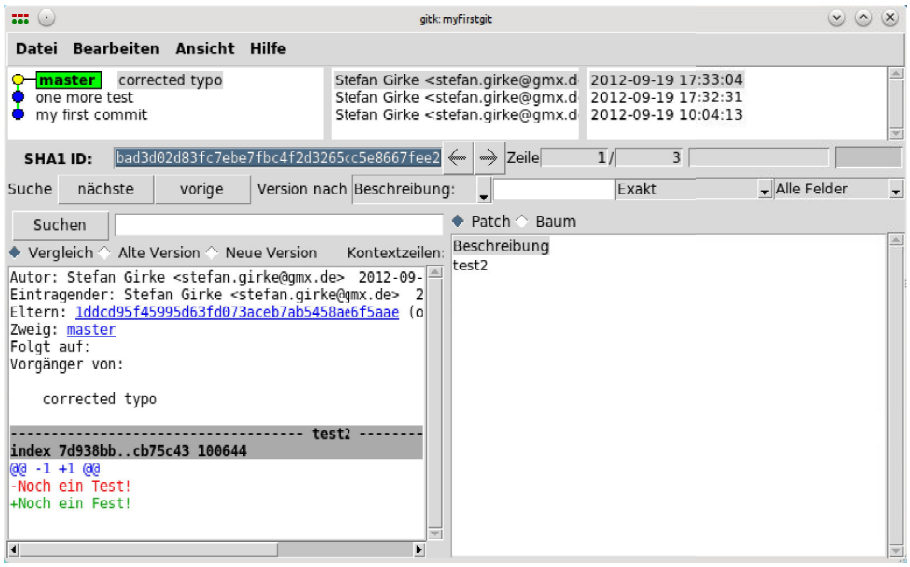
`gitk`

auf, so erhält man ein Programm, in dem man den Verlauf aller Änderungen betrachten kann. Noch sieht es ein bisschen langweilig aus; bei komplexeren Programmen ist es aber sehr hilfreich.



Zweige

Erstellt man nun ein paar weitere Commits, so könnte unser Projekt z.B. so aussehen.



Bisher haben wir nur eine lineare Versionskontrolle verfolgt. Man kann allerdings von jedem Commit einen neuen Zweig erstellen. Aktuell gibt es einen Zweig, den man mit

```
git branch
```

anzeigen lassen kann. Unser Zweig heißt „master“. Mit

```
git branch newbranch
```

legt man einen neuen Zweig „newbranch“ an.

```
git branch
```

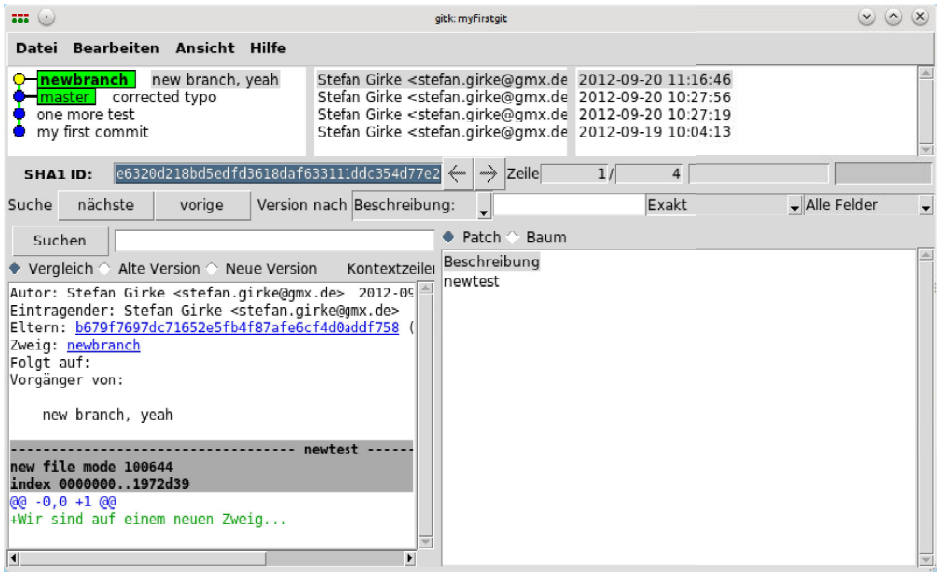
liefert nun

```
* master
newbranch
```

Wir haben ein neuen Zweig erstellt und können mit

```
git checkout newbranch
```

auf diesen Zweig wechseln. Erstellen wir nun eine Änderung und tragen sie ein, so würde das wie folgt aussehen:



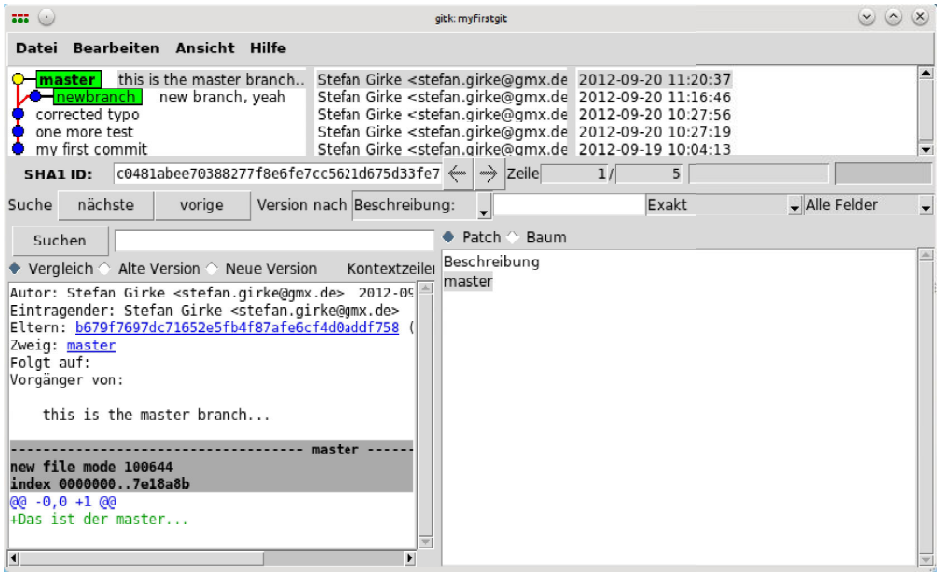
Da wir mit mehreren Zweigen arbeiten, rufen wir gitk mit

```
gitk --all
```

auf, um alle Zweige anzuzeigen. Der aktuelle Zweig steht auf dem neuen Commit, der „master„-Zweig steht auf dem alten Zweig. Mit

```
git checkout master
```

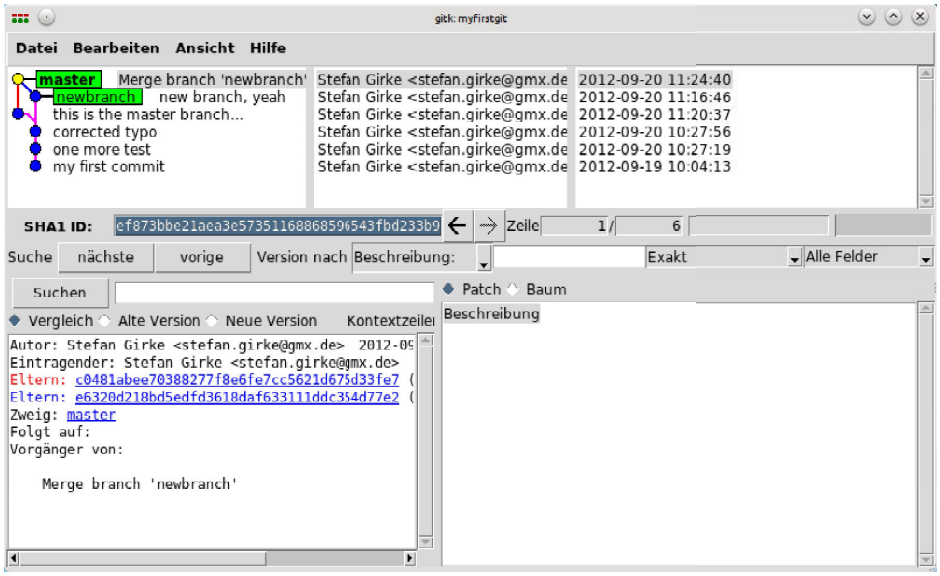
kommt man wieder auf den alten Zweig und kann dort Änderungen durchführen. Hat man dort einen Commit durchgeführt, so sieht das wie folgt aus:



Möchte man beide Zweige zusammenfügen, weil beide Zweige nützliche Features besitzen, so funktioniert das mit

```
git merge newbranch
```

Man fügt also immer den aktuellen Zweig mit dem im letzten Argument angegebenen Zweig zusammen. Das sieht dann wie folgt aus:



Man kann auch Zweige

```
git branch -d newbranch
```

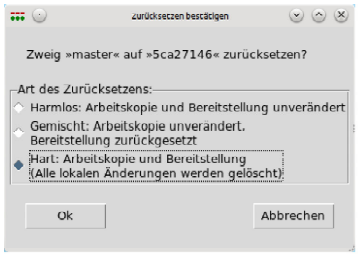
löschen. Wichtig: Auf jedes Zweigende muss ein Zweig stehen, wenn man noch darauf zugreifen will.

Alte Versionen wiederherstellen

Ein alter Commit kann nun wiederhergestellt. Dazu sollte man unbedingt einen neuen Zweig anlegen und anschließend den Befehl

```
git reset --hard
```

anwenden. Achtung: Dabei gehen alle Änderungen, die nicht eingetragen wurden, verloren! Mit gitk wählt man rechts oben „my first commit“ mit Rechtsklick aus und wählt „Zweig ’master’ hierher zurückversetzen“. Danach wählt man im nächsten Fenster „Hart“



Alternativ kann man natürlich auch lokale Änderungen bestehen lassen.

Verteiltes Arbeiten

Hat man ein schon existierendes git-Repository auf einem Server und kennt die Adresse, so kann man mit dem Befehl
`git clone adressewoauchimmer.git`
eine lokale Kopie auf seinem Rechner erstellen. Mit dem Befehl
`git pull`
werden Änderungen, die auf dem Server passiert sind, im lokalen Verzeichnis aktualisiert. Hat man Schreibrechte und möchte seine Änderungen auf den Server laden, so genügt ein
`git push`
nachdem man alle lokalen Änderungen eingetragen hat. Im übrigen kann man mit
`git branch -a`
alle Zweige (lokale und remote [auf dem Server]) anzeigen lassen.

5.3 Programmierwerkzeuge

5.3.1 g++

Der GNU C++-Compiler, Erzeugung von Object-Dateien aus C++-Quelldateien, bzw. Linken von Objektdateien zu einem ausführbaren Programm. Beispiel:
`g++ mytest.cc -o mytest`
erzeugt eine ausführbares Programm „mytest“ aus dem Quellcode mytest.cc.

- `-I/pfad/zu/includes/`: Gibt die Verzeichnisse an, in denen Header-Dateien gesucht werden sollen, welche mit
  `#include <myheader.hh>`
  im Programm verwendet werden.
- `-L/meine/lib`: Gibt den Pfad von Bibliotheken an.
- `-g`: Bewahrt beim Kompilieren symbolische Informationen (Variablennamen etc.). So lässt sich ein späteres Debuggen des Programms erlauben.
- `-ggdb`: Produziert für gdb optimierte Informationen.
- `-O1,-O2,-O3`: Spezifikation des Optimierungslevels. Je höher die Nummer, desto länger ist die Compilezeit, aber die Ausführung des Programms kürzer.
- `-std=gnu++0x` und `-std=gnu++11`: Aktiviert aktuelle C++-Standards.
- `-DDEFINITION=1`: Dies definiert eine Konstante DEFINITION mit dem Wert 1. Man kann also verschiedene Versionen für seinen Code schreiben.

Mit
`g++ --version`
erhält man die aktuelle Version.

5.3.2 make

Das Programm make ist für die Steuerung von Dateierzeugungsprozessen durch Regeln, die in einer Datei namens Makefile enthalten sind, zuständig. Insbesondere können Makefiles auch beim Erstellen von C++-Programmen helfen sein. Ein Beispiel ist

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

Dabei muss die zweite Zeile mit genau einem Tab beginnen. Die erste Datei spezifiziert das Ziel (`mytest`), die notwendigen Eingabedaten (`my_main.o`, `my_sub.o`). Die folgenden Zeilen spezifizieren den Befehl, der ausgeführt wird, um das Ziel zu erzeugen. Ruft man nun

```
make mytest
```

auf, so wird das Vorhandensein der Quellen `my_main.o` und `my_sub.o`, erzeugt diese gegebenenfalls durch weiter Regeln, und anschließend wird der Befehl zur Konstruktion des Ziels ausgeführt.

5.3.3 gdb

gdb ist ein Gnu-Debugger zum schrittweisen Ausführen von Programmen, Variableninspektionen und -manipulationen etc. Hat man sein Programm `myprog` mit der Compileroption

```
-g
```

bzw.

```
-ggdb
```

erstellt, so kann man mit dem Befehl

```
gdb ./myprog
```

den Debugger starten. Mit

```
r
```

startet der Debugger und durchläuft das Programm. Möchte man einzeln Quellcodezeilen durchlaufen, so muss man einen Breakpoint setzen. Mit

```
b 13
```

setzt man den Breakpoint in Zeile 13 der aktuellen Datei. Mit

```
b foo
```

setzt man den Breakpoint in der aktuellen Datei auf die Funktion `foo`. Mit

```
b file.hh:13
```

setzt man den Breakpoint in der Datei file.hh in Zeile 13. Startet man den Debugger, so hält er bei jedem Breakpoint an. Mit

```
n
```

geht man eine Zeile weiter. Mit

```
s
```

folgt man z.B. einem Funktionsaufruf. Man kann sich mit

```
p var
```

den Wert der Variablen `var` anzeigen lassen. Mit

```
bt
```

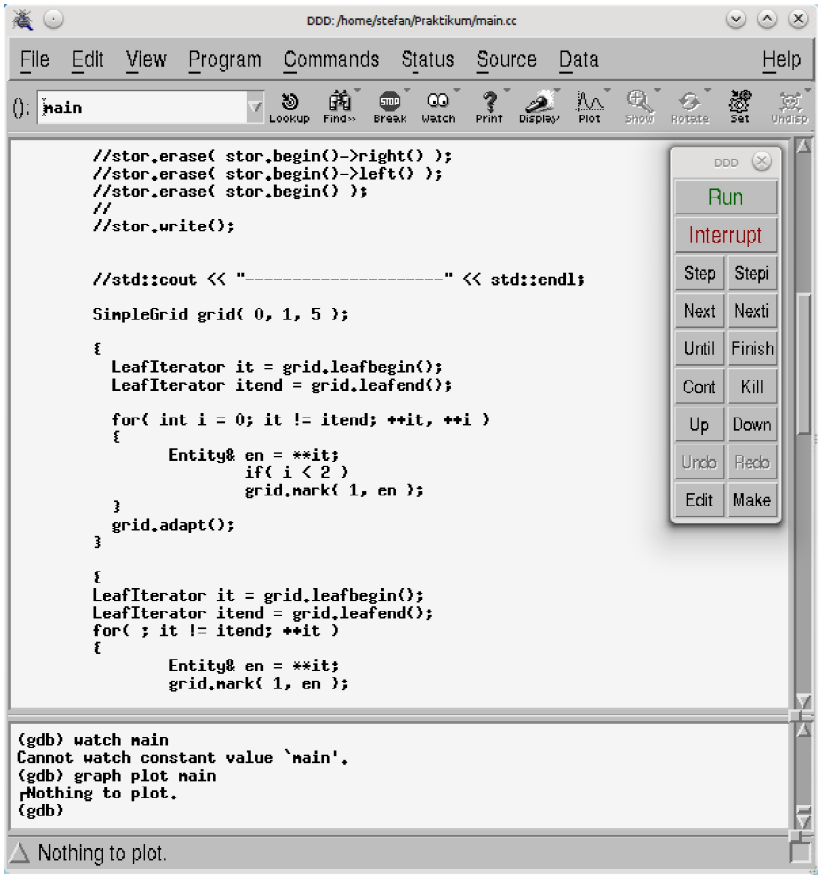
erhält man den aktuellen Stack. Mit

```
c
```

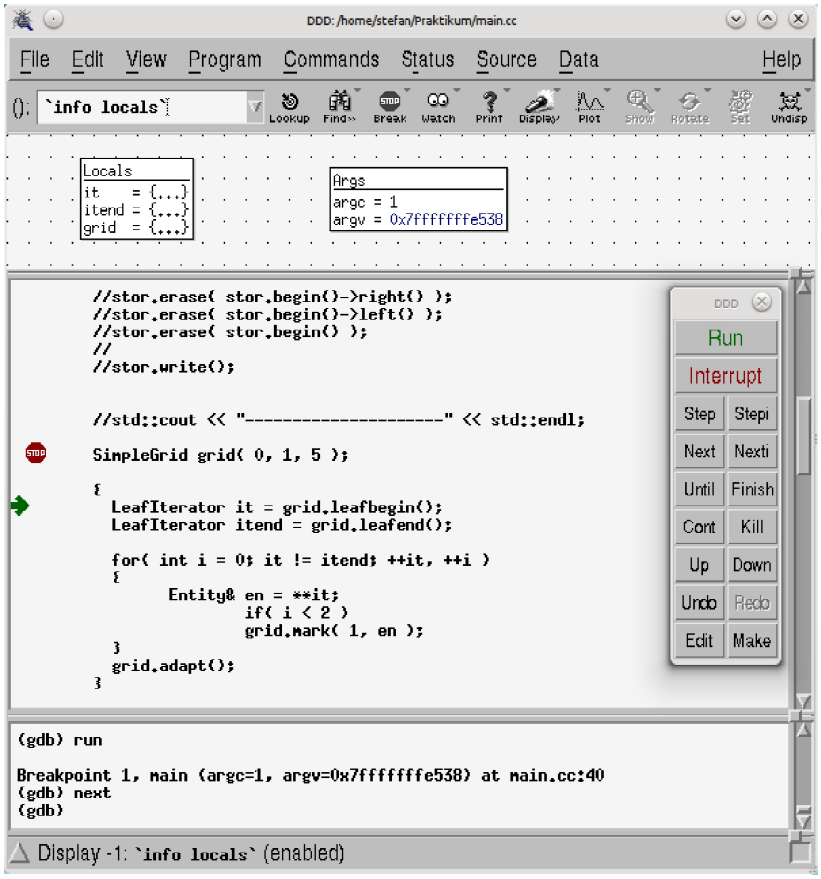
lässt man das Programm weiterlaufen.

5.3.4 ddd

Da nicht jeder gerne in der Konsole arbeitet, gibt es mit ddd eine graphische Oberfläche für gdb.



Man kann einfach durch den Quellcode scrollen und Breakpoints setzen. Alternativ kann man zusätzlich die gdb-Konsole benutzen.



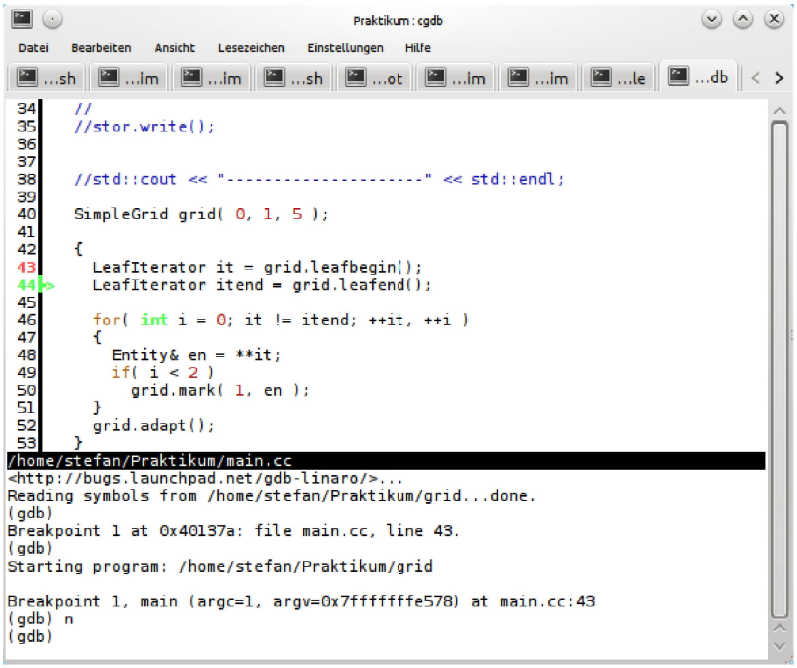
Zusätzlich bietet ddd die Möglichkeit viele Variablen gleichzeitig zu überwachen und beliebig anzuordnen.

5.3.5 cgdb

Wer mit vim arbeitet, möchte vielleicht eine etwas konsolennähere, aber graphische Umgebung für gdb ausprobieren. Folgende Einstellung sollten beim Arbeiten helfen

```
:set tabstop=2
:set winsplit=top_big
:set syntax=c
:set print pretty on
```

Man kann sie auch in die Datei .cgdb/cgdbrc schreiben, damit sie bei jedem Start geladen werden.



5.3.6 valgrind

Wer mit C++ programmiert und mit Zeigern arbeitet, muss immer ein Auge auf die Speicherverwaltung werfen. Speicherlecks und Segmentation Faults sind schwierig zu finden. Hierbei kann valgrind helfen und zeigt potentielle Kandidaten für ein Speicherleck an. Wer schon einmal lange nach einem Zugriffsfehler gesucht hat, weiß dieses Programm zu schätzen.

5.4 Datenvisualisierung

5.4.1 paraview

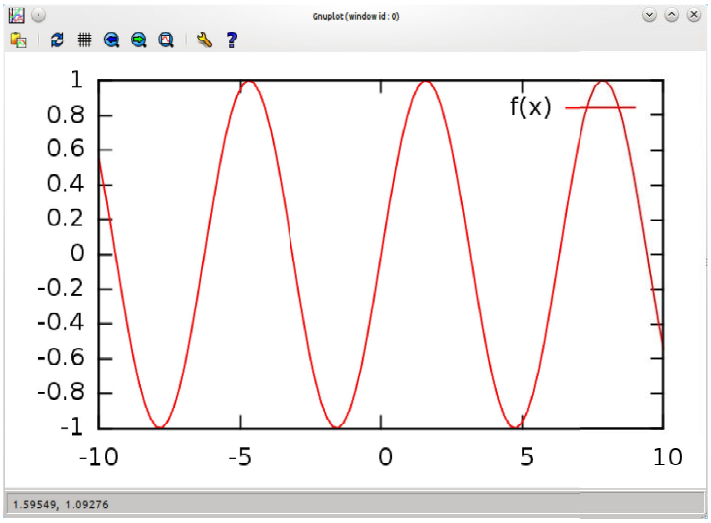
5.4.2 gnuplot

Mit gnuplot erhält man ein einfaches, konsolenbasiertes Programm zum Plotten von Daten und Funktionen. Mit

```
gnuplot
```

startet man die interaktive Shell. Nun kann man sich z.B. beliebige Funktionen und Variablen definieren. Zum Darstellen der Sinusfunktion schreibt man z.B.

```
f(x) = sin(x)
plot f(x)
```

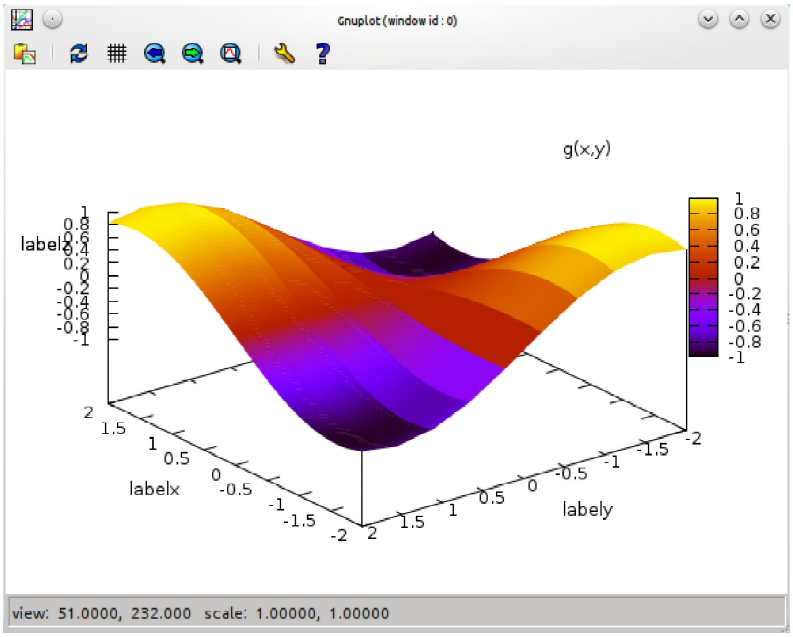


Man kann auch Oberflächen plotten und Beschriftungen setzen. Mit

```
g(x,y)=sin(x)*sin(y)
set xrange[-2:2]
set yrange[-2:2]
set xlabel 'labelx'
set ylabel 'labely'
set zlabel 'labelz'
```

```
splot g(x,y) with pm3d
```

erhält man



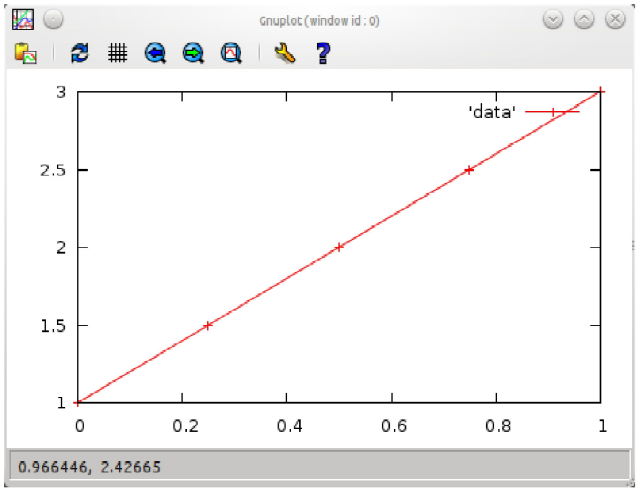
Die wichtigste Eigenschaft von gnuplot ist für uns allerdings das Darstellen von Daten. Legt man eine Datei `data` mit dem Inhalt

```
0.0 1.0
0.25 1.5
0.5 2.0
0.75 2.5
1.0 3.0
```

so kann man mit

```
plot 'data' with linesp
```

die Daten visualisieren:



Auch mehrere Funktionen in einem Plot sind möglich. Mit

`help`

liefert eine Hilfe.

`q`

verlässt man die Shell.

5.5 remote arbeiten

5.5.1 ssh

Mit

```
ssh -XC nutzerkennung@servername.uni-muenster.de
```

kann man sich über die Konsole auf einem Rechner an der Uni Münster einloggen. Dazu ersetzt man die `nutzerkennung` durch seine eigene Nutzerkennung und

`servername` durch einen Computernamen (z.B. `schaf01`, `schaf02`...), der an der Uni Münster steht. Wem dieser Befehl zu lang ist, kann den Ordner `./ssh` mit einer Datei `config` anlegen, falls sie noch nicht existieren und folgende Einträge machen:

```
Host servername
Hostname servername.uni-muenster.de
User nutzerkennung
```

wobei `servername` und `nutzerkennung` wieder zu ersetzen sind. Mit

```
ssh servername
```

kann man sich nun schneller einloggen.
Möchte man Dateien von einem Uni-Computer auf den eigenen Computer kopieren, so verwendet man

```
scp nutzerkennung@servername.uni-muenster.de: /quellpfad/datei
zielpfad/datei
```

oder

```
scp quellpfad/datei nutzerkennung@servername.uni-muenster.de: /zielpfad/datei
```

für den umgekehrten Vorgang. Mit

```
scp -r quellpfad/datei nutzerkennung@servername.uni-muenster.de: /zielpfad/datei
```

kopiert man rekursiv.
Wer sich nicht immer sein Passwort eingeben möchten, sollte sich `ssh-keygen` und `keychain` anschauen.

Kapitel 6

Programmieren mit DUNE

6.1 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/doxygen/dune-grid-html/index.html`
und das Grid-Howto [3] der Kursseite.

6.1.1 Installation

Zunächst erstellt man einen eigenen Ordner
`mkdir prakikum`
`cd praktikum`
Um DUNE-GRID zu verwenden, benötigt man zumindest die beiden weiteren Module DUNE-COMMON und DUNE-GEOMETRY. Es gibt verschiedene Wege, um an ein DUNE-Modul zu kommen. Als Anfänger sollte man sich für eine stabile Version entscheiden, aktuell Version 2.2.0. Hierzu reicht es, die gezippten Datei `dune-common-2.2.0.tar.gz`, `dune-geometry-2.2.0.tar.gz` und `dune-grid-2.2.0.tar.gz` von der DUNE-Homepage
`http://www.dune-project.org/download.html`
herunterzuladen und zu entpacken
`tar xfz dune-common-2.2.0.tar.gz`
`tar xfz dune-geometry-2.2.0.tar.gz`
`tar xfz dune-grid-2.2.0.tar.gz`
Alternativ kann man sich auch auf dem aktuellen Entwicklungszweig arbeiten („trunk“). Möchte man ein git-Repository verwenden, so gibt man den Befehl

```
git svn clone https://svn.dune-project.org/svn/dune-common/trunk
git svn clone https://svn.dune-project.org/svn/dune-geometry/trunk
git svn clone https://svn.dune-project.org/svn/dune-grid/trunk
```

Das Erstellen der Repositorys dauert allerdings etwas länger.
Im Verzeichnis `praktikum` ruft man nun den Befehl
`./dune-common/bin/dunecontrol all`
auf. Möchte man ein eigenes DUNE-Modul erstellen, so ruft man
`./dune-common/bin/duneproject`
auf und folgt den Anweisungen.

6.1.2 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

6.1.3 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.

`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.

6.1.4 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.

6.1.5 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].

6.1.6 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).

6.2 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].

6.2.1 Diskrete Funktionen

Das Konzept von Funktionenräumen aus Abschnitt 2 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 (2.1).

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 (2.4) 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 (2.5) 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 (2.5) 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.

6.2.2 Quadraturen

Die folgenden DUNE-FEM Klassen ermöglichen Integration auf Entitäten mittels Quadraturen, siehe Abschnitt 3.

- `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(...)`

6.2.3 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.

6.2.4 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	**A** sym., p.d.	**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 **Ax** 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.

Kapitel 7

Programmierkonzepte in 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.

7.1 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_;
};
```

7.2 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 Object-Datei 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 Object-Dateien kompiliert werden, und versucht wird, diese Object-Datei 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 Object-Dateien hineincompiliert werden, also keine Probleme erzeugen.

7.3 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.

7.4 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.

7.5 Typdefinitionen/typename

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;
```

Sind Typdefinitionen in einer Klasse öffentlich definiert, so kann Typdefinitionen von diesen Typen erstellen. Ist z.B. `FieldType` eine Typedefinition in `VectorType`, so kann man schreiben

```
typedef typename VectorImp1 VectorType::FieldType;
```

In diesem Fall stellt man das Schlüsselwort `typename` voran, damit der Compiler weiß, dass es sich hinter „`VectorType::`“ um einen Typen handelt und nicht um eine statische Methode. Fehlermeldungen bei Typdefinition entstehen meistens durch Vergessen von `typename` oder aufgrund anderer fehlender Typdefinitionen.

7.6 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.

7.7 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;
```

7.8 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.

Literaturverzeichnis

[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] Bernard Haasdonk. Praktikum Numerik Partielle Differentialgleichungen I WS 2008/2009.

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