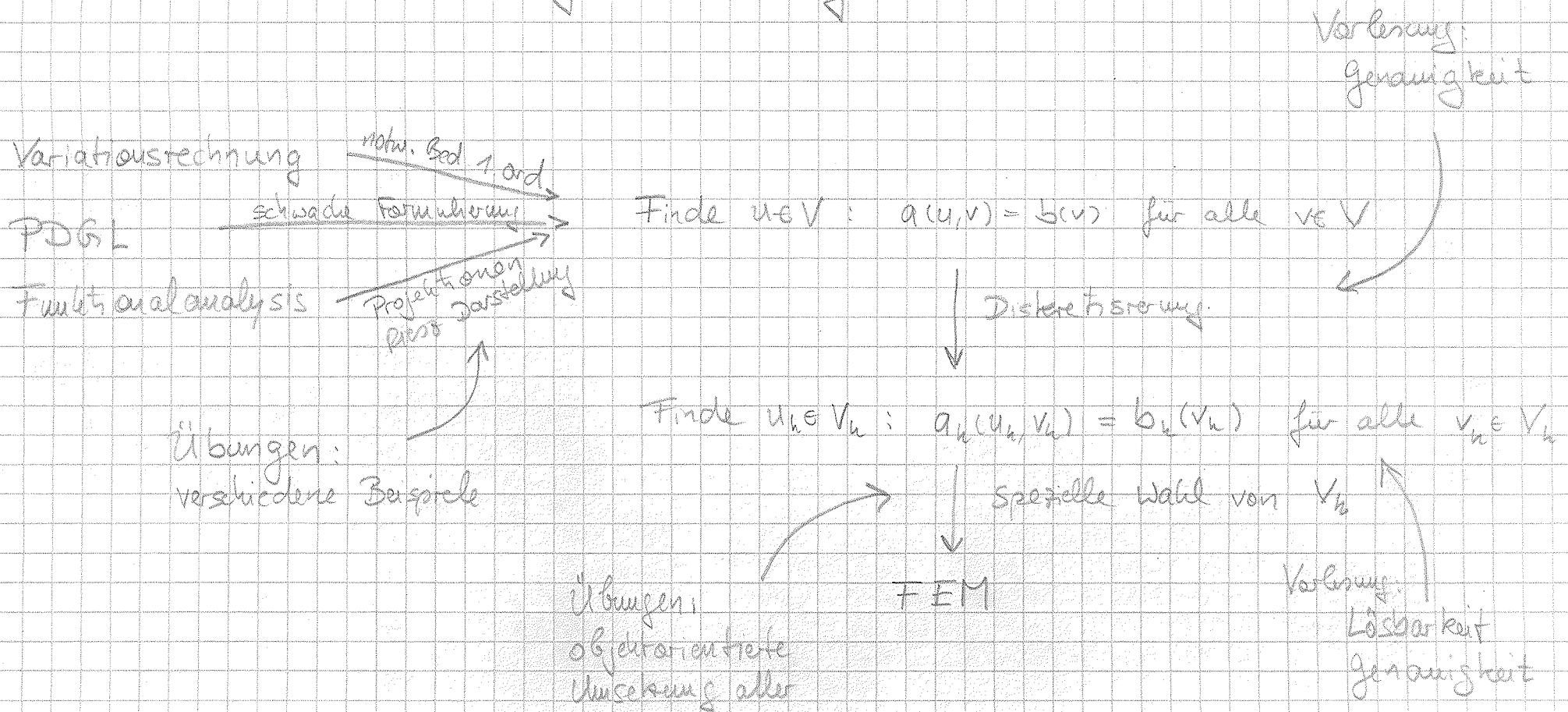
Mi 10⁰⁰

Numerik partieller DGL II – FEM

Struktur von Vorlesung und Übung



Kapitel 1: FEM-Objekte

Welche Objekte werden für eine FEM-Realisierung typischerweise benötigt?

1.1 Formen und Vektoren

Aufgabenstellung: Finde $u \in V_h$ so dass $a(u,v) = b(v)$ für alle $v \in V_h$

offensichtlich benötigte Objekte:
- V_h endlich-dimensionaler K-VR ($K \in \{\mathbb{C}, \mathbb{R}\}$)
- Elemente von V_h
- $a: V_h \times V_h \to K$ bilinear
- $b: V_h \to K$ linear

naheliegende Bestückung der Objekte:

V_h: Auswahl einer Urbasis $B = (\varphi_1, \dots, \varphi_N)$

Element von V_h: Koordinatenspalte $\in K^{N\times 1}$ bezüglich B

a: Koordinatendarstellung bezüglich B (Gewichtsmatrix $\in K^{N\times N}$ im SP auf $K^{N\times 1}$)

b: Koordinatendarstellung $\in K^{1\times N}$ bezüglich B' (duale Basis)

Umsetzung von mathematischen Objekten in Form von Klassen

Klassenname

```
classdef   basis
    {
    }
end
```

← Klassenbeschreibung

```
classdef   vector
    {
    }
end
```

Entscheidende Frage ist nicht: "wie programmiere ich eine Basis / einen Vektor?", sondern "was soll eine Basis / ein Vektor können?"

(Philosophisch: die Frage "was ist ein Vektor?" beantworten wir letztlich mit einer Liste von Fähigkeiten / Regeln die Vektoren haben / erfüllen)

Fähigkeiten werden durch Funktionen in der Klassenbeschreibung programmiert sog Methoden

Beispiel: Fähigkeiten einer Basis B

- beantwortet Frage: Welche Dimension hast du? Antwort: B.dim
- beantwortet Frage: Bist du identisch mit anderer Basis E? Antwort: B.gleich(E)
- beantwortet Frage: Unterscheidest du dich von einer anderen Basis E? Antwort: B.ungleich(E)

Methodennamen

mit einer Basis B

wollen wir Vektoren definieren, z.B. so

Klassenname

u = vector(B, [0,1,1])

Koordinaten bzgl. Basis B

man sagt: u ist eine Instanz (ein konkretes Objekt) der Klasse vector

v = vector(B, [2,1,1])

Fähigkeiten eines Vektors u:

Methodennamen

- hat additiven Inversen Vektor: u.invers alternativer Aufruf invers(u)
- kann mit passendem Vektor v addiert werden: u.plus(v) alternativ plus(u,v)

etc.

Programmierung von Methoden:

```
classdef vector
    methods
        function v = vector(bar, coo)
            {
        end
        function w = plus(u, v)
            {
        end
        function w = invers(u)
            {
        end
    end
end
```

spezielle Methode "Konstruktor"
mit Methodenname = Klassenname zum Konstruieren konkreter Objekte

hat immer Typ der Klasse (hier vector)

klar: das konkrete vector-Objekt muss sich seine Basis und Koordinaten merken ...

allgemein benötigen Klassen oft Daten um die Methoden bereitzustellen

sog. Eigenschaften (properties)

```
classdef vector
    properties (Access = private)
        B   % Basis
        C   % Koordinaten
    end
    methods
        {
    end
end
```

Daten können außerhalb der Klassendefinition nicht gelesen oder geschrieben werden ... Zugriff nur durch Methoden möglich

Beispiel "Konstruktor"

```
function   v = vector(bas, coo)
  if  ~ isa(bas, 'basis')
    error('First argument must be a basis.');
  end
  if ~ isnumeric(coo)
    error('Coordinates (2nd arg) must be of numeric type.');
  end
  if  numel(coo) ~= bas.dim
    error('Coordinate vector has wrong dimension.');
  end                    Zugriff auf Klasseneigenschaften
  v.B = bas;
  v.c = reshape(coo, bas.dim, 1);
end
```

Beispiel: display

Jede Klasse hat automatisch eine display Methode, die aufgerufen wird, wenn ein Objekt der Klasse als Ergebnis in der Konsole dargestellt wird

Diese Methode kann man neu programmieren (überladen)

```
function display(this)
    fprintf('Coordinates w.r.t ');
    this.B          % display Funktion der Basis aufrufen
    c = this.c      % display Funktion des Vektors this.c aufrufen
end
```

Beispiel: plus

```
function w = plus(u,v)
    if ~isa(u,'vector') || ~isa(v,'vector') || u.B ~= v.B
        error('Vectoraddition not possible');
    end
    w = vector(u.B, u.c + v.c);
end
```

Besonderheit des Methodennamens 'plus'

statt u.plus(v) oder plus(u,v) kann man auch u+v schreiben
(man sagt der +-Operator ist überladen)

weitere überladbare Operatoren:

==	eq
~=	ne
-	uminus (als Vorzeichen)
*	mtimes

(Komplette Liste per doc-search Stichwort Overloading)

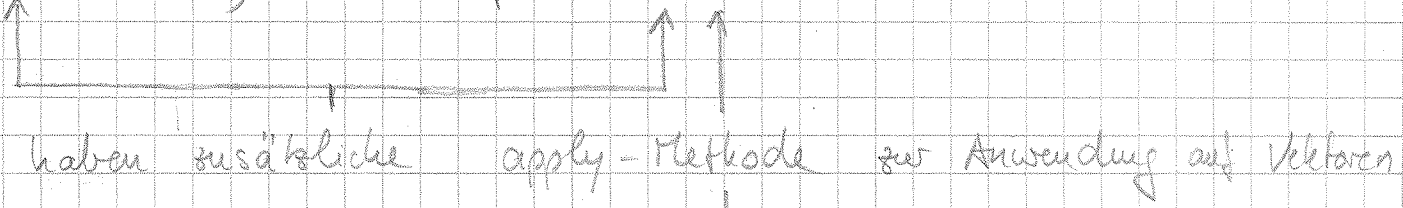
(vorläufige) Realisation von basis

```
properties (Access = private)
    id
end
properties (SetAccess = private)
end                                  dim
methods
    function B = basis(d)
        if ~isnumeric(d) || d<=0 || d~=floor(d)
            error('Dimension must be a positive integer.');
        end
        B.dim = d;
        B.id = d;
    end
    function display(this)
        fprintf('Basis %d\n', this.id);
    end
```

nur Schreibzugriff von außen verboten, Lesezugriff erlaubt

```
function   b = eq(this, B)
    b =   isa(B, 'basis')  &&  B.id = this.id;
end
function   b = ne (this, B)
    b = ~ (this == B);
end
end
```

entsprechende Realisierung weiterer Methoden von vector und von classdef linearForm, classdef bilinearForm



$u = a \setminus b$ ist Lösung von: finde zulässiges u, so dass $a.apply(u,v) = b.apply(v)$
für alle zulässigen v

Problem: a muss auf Koordinaten von u,v zugreifen können (für apply)
— " — b — " — (für $\setminus$)
a muss jeweils auf Basis zugreifen können zur Gültigkeitsprüfung

Ausweg 1: nur Schreibzugriff auf B, c einschränken (SetAccess = private)

Ausweg 2: gemeinsamkeit "Basisdarstellung" als Oberklasse
programmieren

classdef basisRepresentation

properties (SetAccess = private, GetAccess = protected)

abgeleitete Klassen können Eigenschaften lesen

B

C

end

methods

function V = basisRepresentation (bas, coo)

Kopie von vector: Überprüfung ob bas basis und coo numerisch (mit evtl Fehlermeldung)

bR.B = bas;

% Achtung C kann bei gleicher Basis Spaltenvektor, Zeilenvektor, Matrix Tensor, ... sein

%

% Überprüfung ob coo zulässig ist kann nicht hier entschieden werden

% sondern muss durch Methode der jeweils spezielleren Version der Klasse

C = bR.checkedCoordinates (coo);

if isempty (C)

error ('Coordinates not admissible.');

end

```
        bR.c = c;
    end
    function display (this)
        wie bei vector
    end
end
```

von Klassen mit abstrakten Methoden können keine Objekte erzeugt werden; abstrakte Methoden können in abgeleiteten Klassen implementiert werden

```
methods (Abstract, Access = protected)
    C = checkedCoordinates (this, coo);
end
end
```

Änderung von vector: vector ist abgeleitet von basisRepresentation (Elternklasse) und erbt daten/Eigenschaften/Methoden

```
classdef vector < basisRepresentation
    methods
        function v = vector (bas, coo)
            v@basisRepresentation (bas, coo);
        end
```

Aufruf des Konstruktors der Elternklasse

```
    %
    weitere Methoden wie vorher
    außer display (schon geerbt)
    %
end
methods (Access = protected)          % Methode nur für abgeleitete Klassen sichtbar
    function C = checkedCoordinates(this, coo);
        if numel(coo) ~= this.B.dim
            C = [];
        else
            C = reshape(coo, this.B.dim, 1);
        end
    end
end
```

Implementierung der abstrakten Methode

1.2. Triangulierung

Im FEM-Fall ist $V_h \subset \mathcal{F}(\Omega_h, K)$ ← Funktionen $\Omega_h \to K$

$\uparrow$

$\emptyset \neq \Omega_h \subset \mathbb{R}^d$ beschrieben durch Triangulierung

Definition 1.1: Ein Tupel $\mathcal{T} = (T_1, \ldots, T_M)$ von abgeschlossenen Polytopen $T_i \subset \mathbb{R}^d$ heißt Triangulierung, wenn $\mathring{T}_i \cap \mathring{T}_j = \emptyset$ $i \neq j$.
Die Menge $\Omega = (\bigcup_{i=1}^{M} T_i)^\circ$ wird (durch $\mathcal{T}$) triangulierte Menge genannt

$\mathcal{T}$ heißt mit affinen Transformationen erzeugt, wenn es ein Referenzpolytop $T_{ref} \subset \mathbb{R}^d$ gibt und invertierbare affine Transformationen $F_j : \mathbb{R}^d \to \mathbb{R}^d$ so dass $T_j = F_j(T_{ref})$ $j = 1, \ldots, M$.

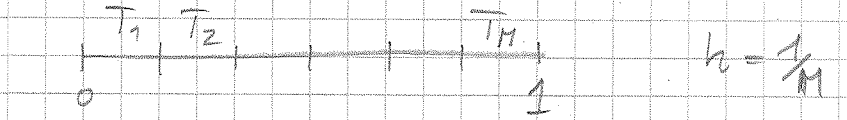
Beispiel: 1D

$T_{ref} = [0,1]$

$F_j(s) = h(s+j-1)$

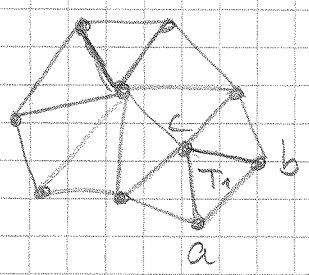
$T_j = [h_j, h_{(j+1)}]$

$\Omega = (0,1)$

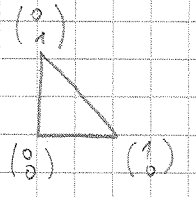


$h = 1/M$

2D

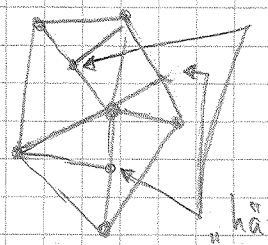


$T_{ref} =$



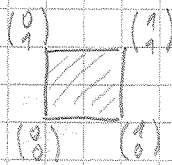
$F_1(s) = a + (b-a \quad c-a)\,s$

"Dreiecksgitter"



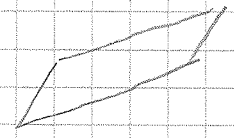
sog. „hängende" Knoten

anderes Referenzpolytop:



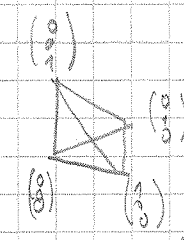
F_j

affin erzeugt

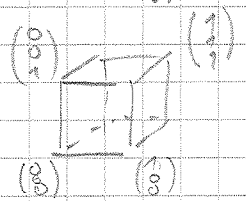


"Parallelogrammgitter"

3D: Referenzpolytope



"Tetraedergitter"



"Spatgitter"

Umsetzung
- classdef → polytope
- clandef → affine Triangulation

Methoden ergeben sich durch Anforderungen an Klassen (später)

Simplex, abgeleitet von polytope: $\text{simplex}(d) = \{ x \in \mathbb{R}^d \mid x_i \geq 0, \sum_{i=1}^{d} x_i \leq 1 \}$

cube, abgeleitet von polytope: $\text{cube}(d) = \{ x \in \mathbb{R}^d \mid 0 \leq x_i \leq 1 \}$