Fakultät für Maschinenbau,
Fahrzeugtechnik und Flugzeugtechnik

Projektarbeit

im Studiengang
Technische Berechnung und Simulation (TBM)

Implementierung der Methode der Finiten Elemente zur Lösung eindimensionaler Randwertprobleme in C++

Autor:
Andreas Beinstingel

Betreuer:
Prof. Dr.-Ing. T. Küpper

18.12.2015

Inhaltsverzeichnis

Abbildungsverzeichnis II

Tabellenverzeichnis III

Algorithmusverzeichnis IV

1. Einführung 1

1.1. Motivation . . . 1

1.2. Ziel der Arbeit . . . 1

2. Numerische Hilfsmittel 2

2.1. Romberg Verfahren . . . 2

2.2. Cholesky Zerlegung . . . 6

2.3. LU Zerlegung . . . 7

3. Konzept: Finite Elemente Methode 8

3.1. Variationsformulierung und Galerkin Ansatz . . . 8

3.2. Basisfunktionen . . . 11

3.3. FE-Gleichungssystem . . . 14

3.4. Gesamtsteifigkeitsmatrix und Randbedingungen . . . 18

3.5. Eigenwertprobleme . . . 22

3.6. Fehlerschätzung und Adaptivität . . . 23

4. Implementierung 26

5. Anwendung 32

5.1. Anwendungsbeispiel 1 . . . 33

5.2. Anwendungsbeispiel 2 . . . 33

5.3. Anwendungsbeispiel 3 . . . 34

5.4. Anwendungsbeispiel 4 . . . 34

5.5. Aufrufendes Programm . . . 35

6. Fazit 36

Literatur 37

A. hmFem.h 38

B. hmFem.cpp 39

Abbildungsverzeichnis

2.1. Sehnentrapezregel 2
2.2. Aufsummierte Sehnentrapezregel 3
2.3. Richardson Extrapolation 4

3.1. Vektorraum 8
3.2. Beispiel zur Fourierreihe 9
3.3. Galerkin Ansatz im 2D 10
3.4. Linearkombination $u_h(x)$ mit Hilfe der Hutfunktion 11
3.5. Lin. Basisfunktionen $\phi_(x)$ 12
3.6. Lin. Formfunktionen 13
3.7. Quad. Formfunktionen 13
3.9. Lokale Unterteilung in Sub-Probleme 24

5.1. Periodisch angeregtes System 32
5.2. Anwendungsbeispiel 1 33
5.3. Anwendungsbeispiel 2 33
5.4. Anwendungsbeispiel 3 34
5.5. Anwendungsbeispiel 4 34

Tabellenverzeichnis

2.1. Romberg Verfahren 4

3.1. Elementzusammenhangstabellen 20

Algorithmusverzeichnis

4.1. Reduzierte *hmFem* Deklaration 26
4.2. Klasse *linFunctions* 27
4.3. Definition linearer Koeffizienten 27
4.4. Erstellung der Matrix $\mathbf{K}$ und des Vektors $\mathbf{f}$ 28
4.5. Elementweise Integration der lokalen Steifigkeitsmatrix $\mathbf{K}^{(i)}$ 29
4.6. Elementweise Integration des lokalen Lastvektors $\mathbf{f}^{(i)}$ 29
4.7. Variationsformulierung (*Left-Hand-Side*) 29
4.8. Variationsformulierung (*Right-Hand-Side*) 29
4.9. Assemblierungsalgorithmen 30
4.10. Einbau der Neumann Randbedingungen 30
4.11. Einbau der Dirichlet Randbedingungen 31
4.12. Lösen des Gleichungssystems und Rückgabe an den Anwender 31

5.1. Aufrufendes Programm bzgl. Beispiel 1 35

1. Einführung

1.1. Motivation

Problemstellungen aus der Physik und der Ingenieurwissenschaft werden in der Mathematik sehr häufig durch Differentialgleichungen beschrieben. Hierbei unterscheidet man zwischen Gewöhnlichen und Partiellen. Erstgenannte zeichnen sich dadurch aus, dass ihre Lösung von nur einer Variablen - zum Beispiel der Zeit - abhängt. Im Gegensatz dazu kann die gesuchte Funktion bei den partiellen Differentialgleichungen von mehreren Variablen - zum Beispiel Raum und Zeit - beeinflusst werden.
Eine exakte Lösung solcher Probleme ist eher selten gegeben oder kann nur unter sehr hohem analytischen Aufwand bestimmt werden. Aus diesem Grund werden derartige Gleichungen unter Zuhilfenahme der numerischen Mathematik approximiert. Dank des enormen Fortschritts der elektronischen Datenverarbeitung können diese Algorithmen durch einen Computer relativ schnell ausgeführt werden.
Vor allem die Methode der Finiten Elemente gewann in den letzten Jahren stark an Beliebtheit und ist in der Industrie weit verbreitet. Nicht zuletzt auf Grund der sehr guten Übereinstimmung der iterierten Ergebnisse mit der Realität.
Im Rahmen dieser Projektarbeit sollen hier eindimensionale Randwertprobleme im Mittelpunkt stehen und mittels der Methode der Finiten Elemente gelöst werden.

1.2. Ziel der Arbeit

In ingenieurwissenschaftlichen Disziplinen treten vor allem inhomogene Differentialgleichungen 2. Ordnung auf. Daher sollen im Weiteren ausschließlich Randwertprobleme der folgenden Form berücksichtigt werden (”Sturm Liouville Problem”).

$$\left.\begin{array}{c} \frac{d}{dx}\left[-p(x)\cdot u'(x)\right] + q(x)\cdot u(x) = f(x) \quad x \in \Omega = (a,b) \\ \alpha_1 \cdot u(a) + \alpha_2 \cdot u'(a) = \eta \\ \beta_1 \cdot u(b) + \beta_2 \cdot u'(b) = \delta \end{array}\right\} (RWP)$$

Am Ende dieser Arbeit soll eine C++ Bibliothek zur Verfügung stehen, die es ermöglicht solche Randwertprobleme zu approximieren.

2. Numerische Hilfsmittel

In Anwendung der Methode der Finiten Elemente müssen sowohl Integrale als auch lineare Gleichungssysteme gelöst werden. Aus diesem Grund ist es sinnvoll vorab Verfahren vorzustellen, die an diesen Stellen zum Einsatz kommen.

2.1. Romberg Verfahren

Das Romberg Verfahren ist ein adaptiver Algorithmus zur numerischen Integration, das für glatte Integranden sehr gute Ergebnisse liefert. Die Methode basiert auf der aufsummierten Sehnentrapezregel und verdankt seine Adaptivität der Richardson Extrapolation.

Die Sehnentrapezregel selbst ist ein Sonderfall der geschlossenen Newton-Cotes-Formeln und integriert Polynome 1. Grades exakt. Die Idee hierbei ist es, den Integranden $f(x)$ durch eine einfacher zu integrierende Funktion $u(x)$ - vorzugsweise ein Polynom - anzunähern. Diese Integrationspolynome $u(x)$ ergeben sich aus den Lagrangeschen Interpolationsformeln und liefern im einfachsten Fall von $n = 1$ eine Gerade für $u(x)$. Womit auch die Namensherkunft geklärt sei.

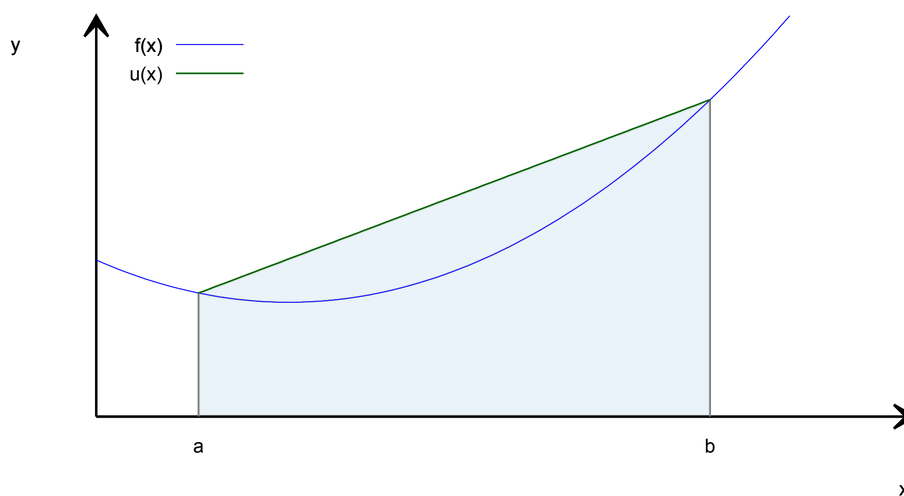

Abbildung 2.1.: Sehnentrapezregel

Die eigentliche Berechnungsformel ergibt sich schließlich zu:

$$\int_a^b f(x)dx \quad \approx \quad \frac{b-a}{2} \cdot (f(a) + f(b)) \tag{2.1}$$

Bis zum Polynomgrad $n = 8$ liefern die Newton-Cotes-Formeln auch für stärker oszilierende Integranden $f(x)$ brauchbare Ergebnisse. Darüber hinaus oszilieren die Approximationspolynome $u(x)$ selbst zu stark, so dass die dabei auftretenden Fehler außerhalb des praktikablen Bereichs liegen.
Um dieser Tatsache entgegenzuwirken wird ist gesamte Integral in eine Summe aus Teilintegralen zu zerlegen.

$$\int_a^b f(x)\,dx \quad = \quad \int_a^{\alpha_1} u_1(x)\,dx + \int_{\alpha_1}^{\alpha_2} u_2(x)\,dx + \cdots + \int_{\alpha_k}^{b} u_{k+1}(x)\,dx \quad k \in N \tag{2.2}$$

Jedes dieser Teilintegrale besitzt einen weniger oszilierenden Integranden und kann für sich durch eine Newton-Cotes-Formel approximiert werden.

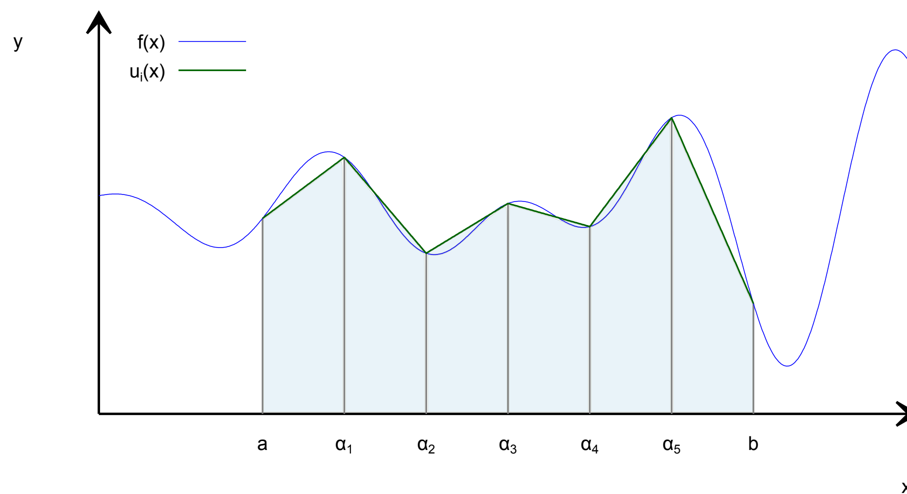

Abbildung 2.2.: Aufsummierte Sehnentrapezregel

Aus Abbildung 2.2 geht schließlich hervor, dass sowohl ein feineres Raster als auch eine höhere Ordnung der Integrationspolynome $u_i(x)$ die Genauigkeit steigern. Ein interessanter Ansatz, der diese beiden Ideen mehr oder weniger miteinander vereint, ist die Extrapolation nach Richardson. Diese besagt, dass zwei numerische Lösungen, die auf Grund zweier verschiedener Diskretisierungen entstanden sind, Anwendung finden können um eine bessere Näherung zu extrapolieren. Die Abbildung 2.3 veranschaulicht diesen Grundgedanken.

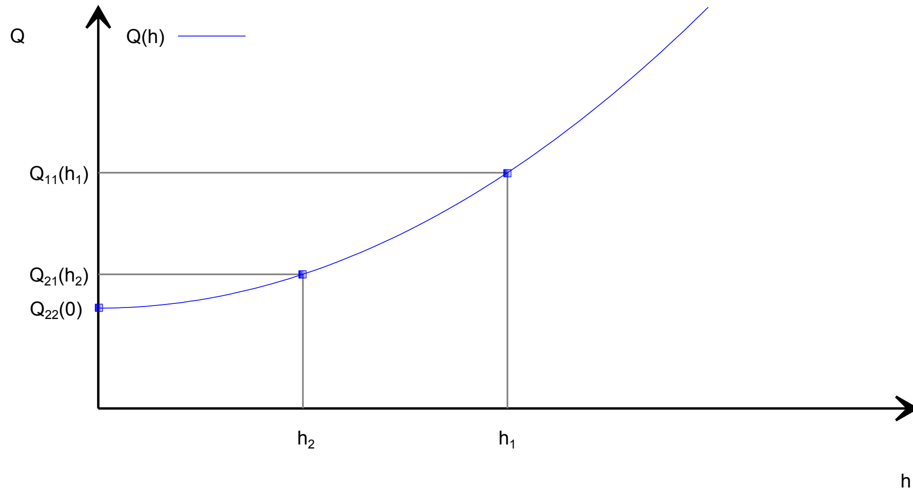

Abbildung 2.3.: Richardson Extrapolation

Hier seien $Q_{11}(h_1)$ und $Q_{21}(h_2)$ die Näherungswerte zu den Schrittweiten h_1 und h_2, wobei $h_2 < h_1$ ist. Wird anschließend ein Interpolationspolynom durch diese Stützpaare berechnet, so kann der Näherungswert $Q_{22}(h = 0)$ extrapoliert werden. Dieser Schritt ergibt Sinn, nachdem die Genauigkeit der numerischen Integration für $h \to 0$ steigt.
Liegt dem numerischem Verfahren eine besondere Form der Fehlerentwicklung zugrunde, so wird durch diesen Schritt die Ordnung des Verfahrens um zwei erhöht, was bei der Sehnentrapezregel der Fall ist.
In Anlehnung an Tabelle 2.1 ist die Extrapolation zum nächst besseren Ergebnis gegeben durch:

$$Q_{i,j} = Q_{i,j-1} + \frac{Q_{i,j-1} - Q_{i-1,j-1}}{(\frac{h_{i-1}}{h_i})^{2(j-1)} - 1} \tag{2.3}$$

Der deutsche Mathematiker Werner Romberg hat darauf aufbauend ein Schema entworfen, das dieses Extrapolationsprinzip sukzessiv auf die aufsummierte Sehnentrapezregel anwendet. Es ist in Tabelle 2.1 dargestellt und soll nachfolgend kurz erläutert werden.

Schrittw.	(2. Ord.)		(4. Ord.)		(6. Ord.)		(8. Ord.)
h_1	Q_{11}						
		$\searrow$					
$h_2 = \frac{1}{2}h_1$	Q_{21}	$\rightarrow$	Q_{22}				
		$\searrow$		$\searrow$			
$h_3 = \frac{1}{2}h_2$	Q_{31}	$\rightarrow$	Q_{32}	$\rightarrow$	Q_{33}		
		$\searrow$		$\searrow$		$\searrow$	
$h_4 = \frac{1}{2}h_3$	Q_{41}	$\rightarrow$	Q_{42}	$\rightarrow$	Q_{43}	$\rightarrow$	Q_{44}

Tabelle 2.1.: Romberg Verfahren

Beginnend mit einer beliebigen Schrittweite h_1 wird der zugehörige Näherungswert Q_{11} mit Hilfe der aufsummierten Sehnentrapezregel bestimmt. Durch Halbierung der Schrittweite kann nach dem gleichen Prinzip ein weiterer Näherungswert Q_{21} berechnet werden. Anschließend führen diese beiden Ergebnisse über den Ansatz nach Richardson auf einen genaueren Integrationswert Q_{22} höherer Ordnung. Sollte dieses Ergebnis noch nicht zufriedenstellend sein, so kann die aufsummierte Sehnentrapezregel ein weiteres Mal genutzt werden um den Wert Q_{31} zu erhalten. Dieser kann wiederum mit Q_{21} zu Q_{32} verrechnet werden, der nun mit Q_{22} auf Q_{33} schließen lässt.
Dieses Vorgehen kann nun so oft wiederholt werden bis eine gewünschte Genauigkeit erreicht wurde. Als Abbruchkriterium dient hierzu zum Beispiel die Differenz der letzten beiden Näherungswerte.

$$|Q_{i,j} - Q_{i-1,j}| \quad \leq \quad \varepsilon \tag{2.4}$$

Wurde das Kriterium erfüllt, so kann der Algorithmus verlassen werden und ein letztes mal auf den nächst besseren Näherungswert $Q_{i,i+1}$ geschlossen werden, da die Extrapolation selbst keinen großen Rechenaufwand darstellt.
Die Implementierung des Romberg-Verfahrens in C++ ist im Anhang hinterlegt.

An dieser Stelle sollte noch erwähnt werden, dass die erste Extrapolation mit der Simpson-Regel zusammenfällt. Diese erhält man ebenfalls aus dem Ansatz nach Newton-Cotes, allerdings für $n = 2$. Gleiches Gilt für den zweiten Extrapolationsschritt. Hier ergibt sich die Milne-Regel, die mit $n = 4$ bestimmt werden kann. Ab der dritten Extrapolation stimmen die Berechnungen nicht mehr exakt mit den Newton-Cotes-Formeln überein.
Nichtsdestotrotz verdeutlicht diese Tatsache nochmals sehr stark, dass sich das Romberg-Verfahren die Ordnung selbst aussucht. Es ist daher nicht zwingend notwendig den Algorithmus mit einer vermeindlich stärkeren Newton-Cotes-Formel höherer Ordnung zu beginnen.

Abschließend sei noch gesagt, dass es viele weitere Methoden zur numerischen Quadratur gibt, wobei jede für sich ihre Daseinsberechtigung hat.
Ein anderer Ansatz existiert zum Beispiel von Carl Friedrich Gauß. Dieser verteilt die Stützstellen nicht gleichmäßig über das Integrationsgebiet, sondern legt diese so fest, dass sie für den jeweiligen Integrationsansatz optimal ausgelegt sind. Hierdurch entsprechen die Ergebnisse zwar schon bei niedrigeren Ordnungen den exakten Werten, aber der Rechenaufwand ist weitaus höher.
Bei der Auswahl eines Verfahrens ist es vor allem wichtig, dass es für den jeweiligen Integranden eine ausreichende Stabilität besitzt. Das Romberg-Schema liefert - wie schon erwähnt - hervorragende Ergebnisse für ausreichend glatte Funktionen.

2.2. Cholesky Zerlegung

Die Cholesky Zerlegung ist ein direktes Verfahren zur Lösung von linearen Gleichungssystemen und liefert, im Gegensatz zu iterativen Verfahren, exakte Ergebnisse. Anders als beim Gaußschen Eliminationsverfahren kann die Cholesky Zerlegung nur für symmetrische positiv-definite Matrizen, wie sie häufig bei der Methode der Finiten Elemente entstehen, angewandt werden. Der Vorteil liegt hierbei darin, dass der rechnerische Aufwand und damit sowohl die Rechenzeit als auch der Speicherbedarf nahezu halbiert werden.
Sollten die Matrizen bzw. die Gleichungssysteme mehr als 50000 Zeilen aufweisen wird empfohlen auf iterative Verfahren zurückzugreifen, da sonst auch hier der Speicheraufwand zu hoch wird.

Ein lineares Gleichungssystem kann grundsätzlich in folgende Form gebracht werden

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.5}$$

wobei die Matrix $\mathbf{A}$ und der Vektor $\mathbf{b}$ bekannt sind. Die Matrix $\mathbf{A}$ muss - wie bereits erwähnt - symmetrisch und positiv-definit sein.
In einem ersten Schritt wird nun die Matrix $\mathbf{A}$ in ein Produkt zweier zueinander transponierter Dreiecksmatrizen zerlegt.

$$\mathbf{A} = \mathbf{R}^T \cdot \mathbf{R} \tag{2.6}$$

Es ergibt sich folgender Zusammenhang:

$$\mathbf{R}^T \cdot \mathbf{R} \cdot \mathbf{x} = \mathbf{b} \tag{2.7}$$

Ersetzt man hier den Ausdruck $\mathbf{R} \cdot \mathbf{x}$ durch einen Hilfsvektor $\mathbf{y}$, so lässt sich das Gleichungssystem umschreiben zu:

$$\mathbf{R}^T \cdot \mathbf{y} = \mathbf{b} \tag{2.8}$$

Dieses neu gewonnene Gleichungssystem kann Dank der Dreiecksform von $\mathbf{R}$ durch simples "Vorwärtseinsetzen" gelöst werden.
Abschließend wird mit Hilfe des nun bekannten Hilfsvektors $\mathbf{y}$ der noch unbekannte Vektor $\mathbf{x}$ durch "Rückwärtseinsetzen" aus dem Gleichungssystem

$$\mathbf{R} \cdot \mathbf{x} = \mathbf{y} \tag{2.9}$$

gewonnen.
Die Umsetzung des Algorithmus ist ebenfalls im Anhang gegeben.

2.3. LU Zerlegung

Nachdem sich vor allem in mathematischen Berechnungen nicht immer symmetrische positiv-definite Matrizen ergeben, soll im nachfolgenden die Gauss Elimination an Hand der LU-Zerlegung vorgestellt werden.
Der Unterschied zum Cholesky Verfahren liegt lediglich in der Zerlegung der Matrix **A**. Diese wird im Falle einer LU-Zerlegung in eine obere und in eine untere Dreiecksmatrix aufgespalten, die nicht zueinander transponiert sind. [1]

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U} \qquad mit: \quad \mathbf{L}^T \neq \mathbf{U} \tag{2.10}$$

Hier wird ersichtlich aus welchem Grund die Cholesky Zerlegung nur den halben Speicherbedarf benötigt.
Alle weiteren Schritte fallen mit denen aus dem vorherigen Kapitel 2.2 zusammen. So kann das entstandene Gleichungssystem

$$\mathbf{L} \cdot \underbrace{\mathbf{U} \cdot \mathbf{x}}_{\mathbf{y}} = \mathbf{b} \tag{2.11}$$

mit dem Hilfsvektor $\mathbf{y} = \mathbf{U} \cdot \mathbf{x}$ umgeschrieben werden zu:

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \tag{2.12}$$

Durch "Vorwärtseinsetzen" ergibt sich auch hier der Vektor **y**, der wiederum in

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \tag{2.13}$$

als neue rechte Seite fungiert. Der gesuchte Vektor **x** kann schließlich über das "Rückwärtseinsetzen" bestimmt werden.

[1] L = engl. *lower* und U = engl. *upper*

3. Konzept: Finite Elemente Methode

Neben der "Finiten Elemente Methode" (kurz: FEM) kann auch das "Schießverfahren" oder das "Finite Differenzen Verfahren" eingesetzt werden, um eindimensionale Randwertprobleme zu lösen. Allerdings stellt die FEM heutzutage das effizienteste Verfahren - vor allem für partielle Differentialgleichungen - dar und soll daher in den nächsten Kapiteln vorgestellt werden.

3.1. Variationsformulierung und Galerkin Ansatz

Der Zugang zur FEM kann grundsätzlich über verschiedene Wege erfolgen. Der hier genutzte Ansatz kommt von dem russischen Mathematiker und Ingenieur Boris Galerkin und ist ein Sonderfall der Methoden der gewichteten Residuen.

Zunächst wird - ähnlich der numerischen Integration - die gesuchte Funktion $u(x)$ einer Differentialgleichung über einen polynomialen Ansatz approximiert.

$$u(x) \quad \approx \quad u_h(x) = \sum_{i=1}^{k} \hat{u}_i \cdot \phi_i(x) \tag{3.1}$$

Der Index h soll symbolisieren, dass es sich um eine Näherungslösung handelt. Die Funktionen $\phi_i(x)$ können in Analogie zur Vektoralgebra als Basisfunktionen aufgefasst werden.

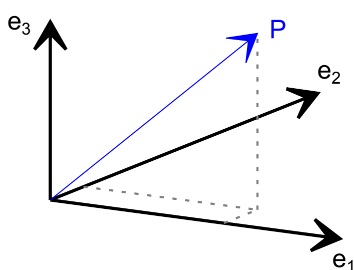

Abbildung 3.1.: Vektorraum

Stellen zum Beispiel drei linear unabhängige Vektoren $\mathbf{e_1}$, $\mathbf{e_2}$ und $\mathbf{e_3}$ eine Basis dar, so spannen diese einen Vektorraum auf. Anschließend kann durch Linearkombination dieser drei Basisvektoren jeder beliebige Punkt im Vektorraum dargestellt werden.

$$\mathbf{P} \quad = \quad \sum_{i=1}^{3} \hat{c}_i \cdot \mathbf{e_i} \tag{3.2}$$

Das Prinzip der Basisfunktionen wird zum Beispiel auch bei den Fourierreihen genutzt. Diese besagen, dass jede ausreichend glatte und stetige Funktion durch eine Linearkombination der linear unabhängigen Basisfunktionen $sin(x)$ und $cos(x)$ ausgedrückt werden kann.

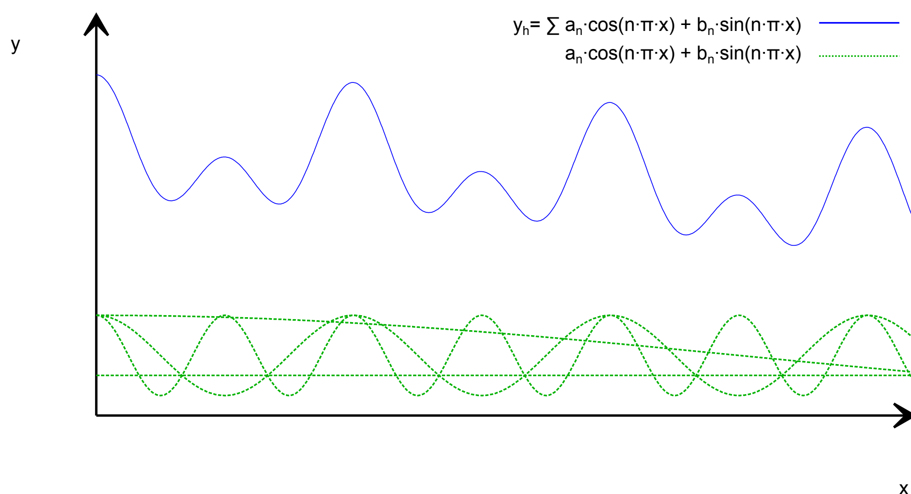

Abbildung 3.2.: Beispiel zur Fourierreihe

Der Vorfaktor $\hat{u}_i$ bzw. $\hat{c}_i$ kann als skalarer Gewichtungsparameter aufgefasst werden. Dieser gibt an, wie stark der Anteil dieser Basisfunktion bzw. dieses Basisvektors in der Linearkombination ist.

Im einem nächsten Schritt wird der polynomiale Ansatz nun in die Differentialgleichung eingesetzt. So ergibt sich aus der exakten Gleichung

$$Lu(x) = f(x) \tag{3.3}$$

der Näherungsausdruck

$$Lu_h(x) = f(x) \tag{3.4}$$

bzw.

$$Lu_h(x) - f(x) = R_h \approx 0 \tag{3.5}$$

wobei L ein beliebiger Differentialoperator ist. Da eine ausreichend hohe Genauigkeit gefordert wird, sollte das Residuum R_h bestenfalls annähernd *Null* sein.
Nach dem Galerkin Ansatz wird hierfür das Residuum R_h mit den Basisfunktionen $\phi_i(x)$ multipliziert und die gesamte Gleichung anschließend über das interessante Gebiet integriert. Die Linearkombination der normierten Basisfunktionen $\phi_i(x)$ soll ab hier folgendermaßen abgekürzt werden.

$$\Phi(x) = \sum_{i=0}^{k} \phi_i(x) \tag{3.6}$$

Es ergibt sich folgender Sachverhalt

$$\int_a^b R_h(x,u_h) \cdot \Phi(x)\,dx \stackrel{!}{=} 0 \qquad (3.7)$$

der später auf ein lineares Gleichungssystem der Form (2.5) schließen lässt.

Die Gleichung (3.7) stellt ein Skalarprodukt zwischen zwei Funktionen in einem Funktionenraum dar und kann ebenfalls mit Hilfe der Vektoralgebra veranschaulicht werden. Das Pondon hierzu ist dementsprechend das Skalarprodukt zwischen zwei Vektoren **a** und **b**. Geometrisch kann dieses

$$\langle \mathbf{a},\mathbf{b} \rangle = \mathbf{a} \circ \mathbf{b} = |\mathbf{a}| \cdot |\mathbf{b}| \cdot cos(\alpha) \qquad (3.8)$$

als Projektion des Vektors **a** auf den Vektor **b** interpretiert werden. Stehen die Vektoren senkrecht zueinander, so hat es den Wert *Null*. Äquivalent dazu lässt sich folgern, dass bei Erfüllung der Gleichung (3.7), das Residuum R_h senkrecht auf den Basisfunktionen $\phi_i(x)$ steht.
In Abbildung 3.3 wird anhand eines zweidimensionalen Beispiels der Vektoralgebra gezeigt, wie der Ansatz nach Galerkin demnach zu deuten ist.

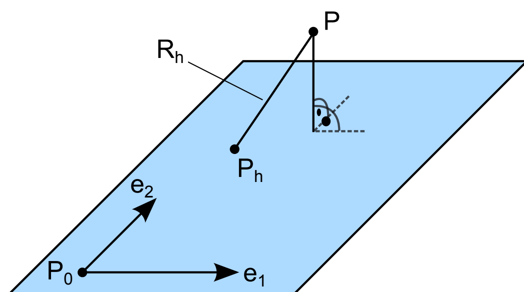

Abbildung 3.3.: Galerkin Ansatz im 2D

Hierbei soll zum Beispiel der Punkt P durch den Punkt P_h angenähert werden. Der Punkt P_h ist über eine Linearkombination der beiden Basisvektoren $\mathbf{e_1}$ und $\mathbf{e_2}$ in der Ebene gegeben. Die bestmögliche Approximation entsteht, wenn der Abstand zwischen P und P_h möglichst gering ist. Das ist der Fall, sobald das Residuum senkrecht auf den Basisvektoren steht.

Wird die Gleichung (3.7) in folgende Form

$$\int_a^b Lu_h(x) \cdot \Phi(x)\,dx = \int_a^b f(x) \cdot \Phi(x)\,dx \qquad (3.9)$$

gebracht, so nennt man (3.9) auch "schwache Formulierung" oder "Variationsformulierung". Aus dieser lässt sich später ein lineares Gleichungssystem bilden, durch dessen Lösung die Näherungsfunktion $u_h(x)$ bestimmt werden kann.

3.2. Basisfunktionen

Bevor die Ableitung des linearen Gleichungssystems aus der Gleichung (3.9) erfolgt, soll zunächst die Wahl der Basisfunktionen $\phi_i(x)$ geklärt werden. Letzten Endes unterscheidet sich die Methode der Finiten Elemente erst durch geschickte Definition dieser Funktionen von der Methode der gewichteten Residuen nach dem Galerkin Ansatz.

Die Besonderheit entsteht zum einen durch die Idee, diese Funktionen auf dem Integrationsgebiet so zu definieren, dass sie überall den Wert *Null* besitzen und nur an einer einzigen Stelle den Wert *Eins*. Zum anderen zerlegt man das Integrationsgebiet in einzelne Intervalle und setzt die besagten - von Null verschiedenen - Stellen auf die Intervallgrenzen. Dank dieser Vorgehensweise werden die Basisfunktion sozusagen *lokal* für jedes Teilintervall definiert, anstatt sie *global* über das gesamte Gebiet zu setzen. Der Vorteil hierbei ist es, dass *lokale* Basisfunktionen zunächst keine weiteren Bedingungen erfüllen müssen. Die Randbedingungen der Differentialgleichung selbst können erst später in einem anderen Schritt berücksichtigt werden. Im Gegensatz dazu müssen *globale* Basisfunktionen diese Randbedingungen bereits hier erfüllen. Wodurch jedes Ausgangsproblem ihre eigenen individuell definierten Basisfunktionen benötigt. Es ist sehr schwer und oft sogar unmöglich derartige Funktionen zu finden.

Unter Berücksichtigung der Ausgangssituation (3.1) und der oben beschriebenen Definition der Basisfunktionen $\phi_i(x)$, wird sofort ersichtlich, dass letztendlich nur die Gewichtungsparameter $\hat{u}_i$ bestimmt werden müssen, um $u_h(x)$ vollständig definieren zu können.
Der in Abbildung 3.4 gezeigte Ansatz ist auch unter dem Begriff "Hutfunktion" bekannt.

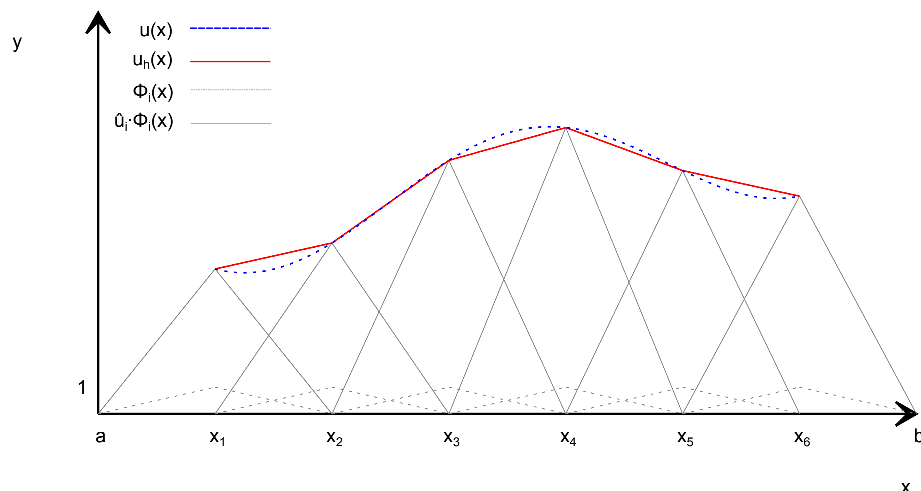

Abbildung 3.4.: Linearkombination $u_h(x)$ mit Hilfe der Hutfunktion

Hierbei wurde das Integrationsgebiet in sieben Elemente gegliedert, woraus sich acht Stützstellen ergeben. Zusätzlich wurde für jede Stützstelle eine Basisfunktion $\phi_i(x)$ der gleichen Form definiert, deren Verlauf einem Impuls mit linearen An- und Abstieg ähnelt, für den Fall, dass die x-Achse die Zeit darstellt. Über die Gewichtungsparameter $\hat{u}_i$ werden schließlich die Funktionswerte der Basisfunktionen $\phi_i(x)$ an den Stützstellen auf den gesuchten Näherungswert $u_h(x_i)$ transformiert.

In Abbildung 3.5 ist der Verlauf der Basisfunktionen $\phi_i(x)$ über ein Element verallgemeinert dargestellt.

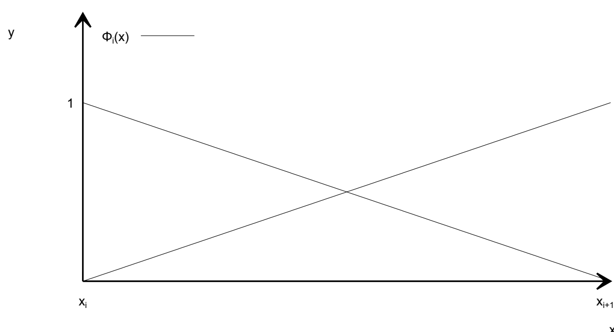

Abbildung 3.5.: Lin. Basisfunktionen $\phi_(x)$

Zu sehen ist die Basisfunktion $\phi_i(x)$ und die Basisfunktion $\phi_{i+1}(x)$. Beide besitzen einen linearen Verlauf über dem Abschnitt und erfüllen die Bedingung: [2]

$$\phi_i(x_j) \quad = \quad \delta_{ij} \qquad i,j = 1,..2 \quad (3.10)$$

Nachdem (3.10) für alle Teilintervalle gelten muss, lässt sich folgern, dass es ausreicht die Definition für $\phi_i(x)$ und $\phi_{i+1}(x)$ lediglich für ein Element aufzustellen. Anschließend können diese Funktionen auf alle weiteren Elemente abstrahiert werden.

Neben dem linearen Ansatz gibt es weitere Möglichkeiten Basisfunktionen $\phi_i(x)$ elementweise zu definieren. Für eine genauere Approximationen können Funktionen höhere Ordnung angesetzt werden, die zusätzliche Stützstellen innerhalb eines Elementes aufweisen. Auch hier muss die Bedingung (3.10) erfüllt sein und kann allgemein für eine beliebige Anzahl an Stützstellen x_i umformuliert werden zu:

$$\phi_i(x_j) \quad = \quad \delta_{ij} = \begin{cases} 1 & für \quad i = \quad j \\ 0 & für \quad i \neq \quad j \end{cases} \qquad i = 1,..n; j = 1,..k \qquad (3.11)$$

Mit n wird die Anzahl der Intervalle bzw. Elemente bezeichnet und mit k die Anzahl der der Stützstellen. Im linearen Fall gilt: $k = n + 1$.
Die Herleitung derartiger Basisfunktionen kann - wie bereits bei den Newton-Cotes-Formeln aus Kapitel 2 - ebenfalls über die Lagrangeschen Interpolationsformeln erfolgen und soll nachfolgend kurz erklärt werden.

[2] δ_{ij} = "Kronecker-Delta"

Grundsätzlich kann immer noch von Gleichung (3.11) ausgegangen werden. Lediglich um Verwechslungen mit dem Integrationsgebiet zu vermeiden, hat es sich verallgemeinert einige Namensänderungen durchzuführen.

$$\phi_\beta(\xi_\alpha) \quad = \quad \delta_{\alpha\beta} = \begin{cases} 1 & für \quad \alpha = \quad \beta \\ 0 & für \quad \alpha \neq \quad \beta \end{cases} \qquad \alpha, \beta = 1{,}2, ..., m+1 \tag{3.12}$$

Nach Lagrange lassen sich schließlich $m + 1$ Polynome m-ten Grades bilden, die diese Bedingung erfüllen:

$$\phi_\beta(\xi_\alpha) \quad = \quad \prod_{\substack{\alpha=1 \\ \alpha\neq\beta}}^{m+1} \frac{\xi - \xi_\alpha}{\xi_\beta - \xi_\alpha} \qquad mit: \beta = 1, 2, ...m+1 \tag{3.13}$$

$$= \quad \frac{\xi - \xi_1}{\xi_\beta - \xi_1} \cdot \frac{\xi - \xi_2}{\xi_\beta - \xi_2} \cdots \frac{\xi - \xi_{\beta-1}}{\xi_\beta - \xi_{\beta-1}} \cdot \frac{\xi - \xi_{\beta+1}}{\xi_\beta - \xi_{\beta+1}} \cdots \frac{\xi - \xi_{m+1}}{\xi_\beta - \xi_{m+1}} \tag{3.14}$$

In dieser Projektarbeit werden Basisfunktionen bis zu einem Polynomgrad von $m = 3$ eingesetzt.

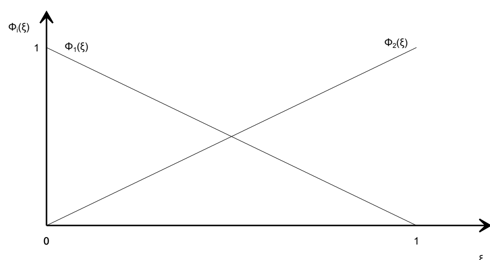

Abbildung 3.6.: Lin. Formfunktionen

$$\begin{aligned} \phi_1(\xi) &= \quad 1 - \xi \\ \phi_2(\xi) &= \quad \xi \end{aligned} \tag{3.15}$$

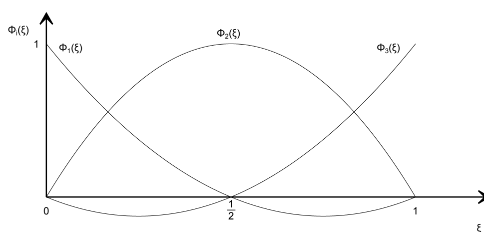

Abbildung 3.7.: Quad. Formfunktionen

$$\begin{aligned} \phi_1(\xi) &= \quad 2\xi^2 - 3\xi + 1 \\ \phi_2(\xi) &= \quad -4\xi^2 + 4\xi \\ \phi_3(\xi) &= \quad 2\xi^2 - \xi \end{aligned} \tag{3.16}$$

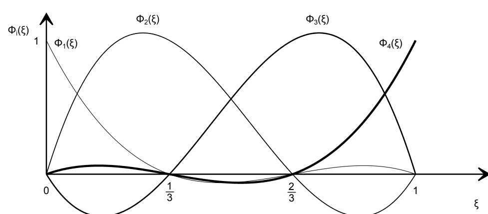

Abbildung 3.8.: Kub. Formfunktionen

$$\begin{aligned}
\phi_1(\xi) &= -\frac{9}{2}\xi^3 + 9\xi^2 - \frac{11}{2}\xi + 1 \\
\phi_2(\xi) &= \frac{27}{2}\xi^3 - \frac{45}{2}\xi^2 + 9\xi \\
\phi_3(\xi) &= -\frac{27}{2}\xi^3 + 18\xi^2 - \frac{9}{2}\xi \\
\phi_4(\xi) &= \frac{9}{2}\xi^3 - \frac{9}{2}\xi^2 + \xi
\end{aligned} \quad (3.17)$$

Es ist bereits hier ersichtlich, dass die Genauigkeit der FEM - ähnlich der numerischen Integration - über zwei Wege gesteigert werden kann. Zum einen über eine feinere Diskretisierung der Intervalle (*"h-Methode"*) und zum anderen über eine höhere Ordnung in den Basisfunktionen $\phi_i(x)$ (*"p-Methode"*).

3.3. FE-Gleichungssystem

Ausgehend von der Variationsformulierung (3.9) soll in diesem Kapitel das charakteristische Gleichungssystem der Finiten Elemente bestimmt werden. Dieses wird am Ende in folgender Form vorliegen:

$$\mathbf{K} \cdot \hat{\mathbf{u}} = \mathbf{f} \quad (3.18)$$

Während die Matrix $\mathbf{K}$ die sogenannte *"Steifigkeitsmatrix"* darstellt, beinhaltet der Vektor $\hat{\mathbf{u}}$ die Gewichtungsparameter $\hat{u}_i$ und wird daher als *"Verschiebungsvektor"* bezeichnet. Der Vektor $\mathbf{f}$ ist auf Grund seiner Beschaffenheit auch unter dem Begriff *"Lastvektor"* bekannt und enthält die äußeren Anregungen des Differentialgleichungssystems.

Unverkennbar ergibt sich der Lastvektor $\mathbf{f}$ aus der rechten Seite der Variationsformulierung (3.9) ergeben wird. Demzufolge ist die linke Seite für die Steifigkeitsmatrix $\mathbf{K}$ zuständig. An dieser Stelle muss jedoch von der allgemeinen Formulierung abgewichen werden, da die Steifigkeitsmatrix $\mathbf{K}$ von dem Differentialoperator L abhängt. Der Aufwand eine allgemein gültige Steifigkeitsmatrix $\mathbf{K}$ während der Programmlaufzeit zu definieren ist wenig praktikabel und mathematisch nahezu unmöglich. Aus diesem Grund sind in gängigen FE Programmen sogenannte Elementbibliotheken enthalten, die je nach Differentialoperator L, die Steifigkeitsmatrix $\mathbf{K}$ zum vorgenannten Differentialgleichungsproblem beinhalten.

Schlussendlich sollen in dieser Projektarbeit Differentialgleichungssysteme der Form (RWP) behandelt werden. Schließlich decken diese einen Großteil der auftretenden Probleme in den Natur- und Ingenieurwissenschaften ab.
Um späteren Anwendern des Lösungsverfahrens die Eingabe zu erleichtern, wird das Randwertproblem in eine etwas gängigere Form gebracht:

$$\begin{aligned} -p(x)\cdot u''(x) + q(x)\cdot u'(x) + r(x)\cdot u(x) = f(x) \quad x \in \Omega = (a,b) \\ \alpha_1 \cdot u(a) + \alpha_2 \cdot u'(a) = \eta \\ \beta_1 \cdot u(b) + \beta_2 \cdot u'(b) = \delta \end{aligned} \tag{3.19}$$

In einem weiteren Schritt werden zur Herleitung der Steifigkeitsmatrix **K** konstante Koeffizienten verwendet. Der spätere Einbau der linearen Koeffizienten ist vergleichsweise einfach und soll am Ende dieses Kapitels diskutiert werden.
Das nun entstandene Ausgangsproblem

$$-p\cdot u''(x) + q\cdot u'(x) + r\cdot u(x) = f(x) \tag{3.20}$$

führt unter der Anwendung des Galerkin Ansatzes zu folgender Variationsformulierung:

$$\int_a^b -p\cdot u''(x)\cdot\Phi(x) + q\cdot u'(x)\cdot\Phi(x) + r\cdot u(x)\cdot\Phi(x)dx = \int_a^b f(x)\cdot\Phi(x)dx \tag{3.21}$$

Mit Hilfe der partiellen Integration

$$\int_a^b w'(x)\cdot v(x)dx = [w(x)\cdot v(x)]_a^b - \int_a^b w'(x)\cdot v(x)dx \tag{3.22}$$

kann der erste Term der linken Seite so umformuliert werden, dass die zweite Ableitung von $u(x)$ verschwindet bzw. eine Differentiation auf die Basisfunktion $\Phi(x)$ übertragen wird.

$$\begin{aligned} \int_a^b +p\cdot u'(x)\cdot\Phi'(x) + q\cdot u'(x)\cdot\Phi(x) + r\cdot u(x)\cdot\Phi(x)dx \\ - p\cdot[u'(x)\cdot\Phi(x)]_a^b = \int_a^b f(x)\cdot\Phi(x)dx \end{aligned} \tag{3.23}$$

Der durch die partielle Integration entstandene Ausdruck außerhalb des Integrals wird vorerst vernachlässigt, weil es sich hierbei bereits um das Einsetzen von Randwerten handeln würde. Der Einbau der Randwerte kann aber - wie in der FEM üblich - erst in einem späteren Schritt erfolgen. Hierzu sei auf das nächste Kapitel verwiesen.

Werden nun die exakten Ausdrücke $u(x)$ durch den polynomialen Ansatz (3.1) ausgetauscht wird sofort ersichtlich, dass mit Hilfe der partiellen Integration die Basisfunktionen $\phi_j(x)$ nur noch aus dem Raum C^1 und nicht mehr aus dem Raum C^2 stammen müssen. Aus diesem Grund ist es möglich das unter (3.19) definierte Problem mit linearen Ansatzfunktionen zu lösen. [3]

$$\begin{aligned}\int_a^b &+ p \cdot (\sum_{j=1}^{k} \hat{u}_j \cdot \phi_j(x))' \cdot \Phi'(x) \\ &+ q \cdot (\sum_{j=1}^{k} \hat{u}_j \cdot \phi_j(x))' \cdot \Phi(x) \\ &+ r \cdot (\sum_{j=1}^{k} \hat{u}_j \cdot \phi_j(x)) \cdot \Phi(x) \qquad dx &= \int_a^b f(x) \cdot \Phi(x) dx\end{aligned} \tag{3.24}$$

Abschließend ist die Linearkombination $\Phi(x)$ aus dem Galerkin Ansatz durch die Summenschreibweise zu ersetzen. Außerdem ist das Integral über dem Intervall $[a,b]$ in eine Summe aus Teilintegralen über den einzelnen Elementen zu zerlegen.

$$\begin{aligned}\sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} &+ p \cdot (\sum_{j=1}^{k} \hat{u}_j \cdot \phi_j(x))' \cdot (\sum_{l=1}^{k} \phi_l(x))' \\ &+ q \cdot (\sum_{j=1}^{k} \hat{u}_j \cdot \phi_j(x))' \cdot (\sum_{l=1}^{k} \phi_l(x)) \\ &+ r \cdot (\sum_{j=1}^{k} \hat{u}_j \cdot \phi_j(x)) \cdot (\sum_{l=1}^{k} \phi_l(x)) \qquad dx &= \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} f(x) \cdot (\sum_{l=1}^{k} \phi_l(x)) \qquad dx\end{aligned} \tag{3.25}$$

Gemäß der Bedingung (3.11) sind bei linearen Ansatzfunktionen nur $\phi_{i-1}(x)$ und $\phi_i(x)$ über dem Teilintervall $[x_{i-1},x_i]$ von *Null* verschieden. [4] Daher reduzieren sich die Summen innerhalb der Integrale zu:

$$\begin{aligned}\sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} &+ p \cdot (\sum_{j=i-1}^{i} \hat{u}_j \cdot \phi_j(x))' \cdot (\sum_{l=i-1}^{i} \phi_l(x))' \\ &+ q \cdot (\sum_{j=i-1}^{i} \hat{u}_j \cdot \phi_j(x))' \cdot (\sum_{l=i-1}^{i} \phi_l(x)) \\ &+ r \cdot (\sum_{j=i-1}^{i} \hat{u}_j \cdot \phi_j(x)) \cdot (\sum_{l=i-1}^{i} \phi_l(x)) \qquad dx &= \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} f(x) \cdot (\sum_{l=i-1}^{i} \phi_l(x)) \qquad dx\end{aligned} \tag{3.26}$$

[3] Neben den Lagrange'schen Ansatzfunktionen werden für DGL höherer Ordnung u. a. sogenannte hermitische Polynome eingestzt (z.B. Bernoulli-Balkenbiergung)

[4] quadratische Ansatzfunktionen: $\phi_{i-1}(x)$, $\phi_{i-\frac{1}{2}}(x)$ und $\phi_i(x)$
kubische Ansatzfunktionen: $\phi_{i-1}(x)$, $\phi_{i-\frac{2}{3}}(x)$, $\phi_{i-\frac{1}{3}}(x)$ und $\phi_i(x)$

Das Faktorisieren von $\hat{u}$ sowie das Ausmultiplizieren der Summen der Basisfunktionen - für lineare Ansatzfunktionen - führt zu folgendem Sachverhalt:

$$\begin{aligned}\sum_{i=1}^{n} \Bigg[\quad & \hat{u}_{i-1} \cdot \int_{x_{i-1}}^{x_i} p \cdot \phi'_{i-1}(x) \cdot \phi'_{i-1}(x) + q \cdot \phi'_{i-1}(x) \cdot \phi_{i-1}(x) + r \cdot \phi_{i-1}(x) \cdot \phi_{i-1}(x) \quad dx \\ & + \hat{u}_{i-1} \cdot \int_{x_{i-1}}^{x_i} p \cdot \phi'_{i-1}(x) \cdot \phi'_{i}(x) + q \cdot \phi'_{i-1}(x) \cdot \phi_{i}(x) + r \cdot \phi_{i-1}(x) \cdot \phi_{i}(x) \quad dx \\ & + \hat{u}_{i} \cdot \int_{x_{i-1}}^{x_i} p \cdot \phi'_{i}(x) \cdot \phi'_{i-1}(x) + q \cdot \phi'_{i}(x) \cdot \phi_{i-1}(x) + r \cdot \phi_{i}(x) \cdot \phi_{i-1}(x) \quad dx \\ & + \hat{u}_{i} \cdot \int_{x_{i-1}}^{x_i} p \cdot \phi'_{i}(x) \cdot \phi'_{i}(x) + q \cdot \phi'_{i}(x) \cdot \phi_{i}(x) + r \cdot \phi_{i}(x) \cdot \phi_{i}(x) \quad dx \Bigg] \\ & = \sum_{i=1}^{n} \left[\int_{x_{i-1}}^{x_i} f(x) \cdot \phi_{i-1}(x) + f(x) \cdot \phi_{i}(x) \quad dx \right] \end{aligned} \tag{3.27}$$

Hieraus lassen sich nun die Koeffizienten der Steifigkeitsmatrix $\mathbf{K}^{(i)}$ und des Lastvektors $\mathbf{f}^{(i)}$ für ein Element ablesen.

$$\begin{aligned} K_{1,1}^{(i)} &= \int_{x_{i-1}}^{x_i} p \cdot \phi'_{i-1}(x) \cdot \phi'_{i-1}(x) + q \cdot \phi'_{i-1}(x) \cdot \phi_{i-1}(x) + r \cdot \phi_{i-1}(x) \cdot \phi_{i-1}(x) \quad dx \\ K_{2,1}^{(i)} &= \int_{x_{i-1}}^{x_i} p \cdot \phi'_{i-1}(x) \cdot \phi'_{i}(x) + q \cdot \phi'_{i-1}(x) \cdot \phi_{i}(x) + r \cdot \phi_{i-1}(x) \cdot \phi_{i}(x) \quad dx \\ K_{1,2}^{(i)} &= \int_{x_{i-1}}^{x_i} p \cdot \phi'_{i}(x) \cdot \phi'_{i-1}(x) + q \cdot \phi'_{i}(x) \cdot \phi_{i-1}(x) + r \cdot \phi_{i}(x) \cdot \phi_{i-1}(x) \quad dx \\ K_{2,2}^{(i)} &= \int_{x_{i-1}}^{x_i} p \cdot \phi'_{i}(x) \cdot \phi'_{i}(x) + q \cdot \phi'_{i}(x) \cdot \phi_{i}(x) + r \cdot \phi_{i}(x) \cdot \phi_{i}(x) \quad dx \end{aligned} \tag{3.28}$$

$$\begin{aligned} f_1^{(i)} &= \int_{x_{i-1}}^{x_i} f(x) \cdot \phi_{i-1}(x) \quad dx \\ f_2^{(i)} &= \int_{x_{i-1}}^{x_i} f(x) \cdot \phi_{i}(x) \quad dx \end{aligned} \tag{3.29}$$

Damit ist die Herkunft der Gleichung (3.18) grundsätzlich gegeben durch:

$$\begin{bmatrix} K_{1,1}^{(i)} & K_{1,2}^{(i)} \\ K_{2,1}^{(i)} & K_{2,2}^{(i)} \end{bmatrix} \cdot \begin{bmatrix} \hat{u}_{i-1} \\ \hat{u}_{i} \end{bmatrix} = \begin{bmatrix} f_1^{(i)} \\ f_2^{(i)} \end{bmatrix} \tag{3.30}$$

Die Steifigkeitsmatrix $\mathbf{K}^{(i)}$ in der Gleichung (3.30) ist eine sogenannte *lokale* Steifigkeitsmatrix, symbolisiert durch den Index $^{(i)}$. Die Tatsache zeigt sich über die Integration der einzelnen Elemente. Gleiches gilt dementsprechend für den Lastvektor $\mathbf{f}^{(i)}$.
Nachdem bei der Herleitung von linearen Ansatzfunktionen ausgegangen wurde, ist das Ergebnis eine 2×2-Matrix bzw. ein zweidimensionaler Vektor. Bei einem quadratischen Ansatz hätte sich eine 3×3-Matrix bzw. ein dreidimensionaler Vektor ergeben. Die Dimension lässt sich folglich mit Hilfe des Lagrange'schen Polynomansatzes zu $DIM = m + 1$ definieren.

Ein äußerst interessanter Aspekt ist, dass auf Grund der konstanten Koeffizienten alle lokalen Steifigkeitsmatrizen $\mathbf{K}^{(i)}$ identisch sind - vorausgesetzt das Intervall $[a,b]$ wurde in äquidistante Teilgebiete zerlegt. Daher würde es in diesem Fall genügen, die lokale Steifigkeitsmatrix $\mathbf{K}^{(i)}$ einmalig zu bestimmen.
Im Umkehrschluss lässt sich folgern, dass bei linearen Koeffizienten die *lokale* Elementsteifigkeitsmatrix $\mathbf{K}^{(i)}$ tatsächlich für jedes einzelne Element zu integrieren ist. Der Rechenaufwand steigt daher mit der Elementanzahl rasant an.
Zur Bestimmung dieser Matrizen genügt es in den Gleichungen (3.28) die konstanten Koeffizienten durch lineare Koeffizienten zu ersetzen. [5]

Abschließend muss hier noch auf die in Kapitel 3.2 angedeuteten Transformationen der Basisfunktionen $\phi_i(x)$ auf das Intervall $[a, b]$ aufmerksam gemacht werden.
Dort wurden die Basisfunktionen (3.15), (3.16) und (**??**) über dem Intervall $[0, 1]$ definiert. Solange nun das Integrationsgebiet $[x_{i-1}, x_i]$ in (3.28) und (3.29) mit dem Intervall übereinstimmt - sprich $x_{i-1} = 0$ und $x_i = 1$ - treten keine Komplikationen auf. Andernfalls muss entweder das Integrationsgebiet $[x_{i-1}, x_i]$ auf das Gebiet $[0, 1]$ transformiert werden oder dementsprechend umgekehrt das Gebiet $[0, 1]$ auf das Integrationsgebiet $[x_{i-1}, x_i]$ abstrahiert werden.
Als Abbildungsvorschriften ergeben sich folgende Zusammenhänge:

$$\xi^{(i)}(x) = \frac{x - x_{i-1}}{x_i - x_{i-1}} \tag{3.31}$$

$$x^{(i)}(\xi) = (x_i - x_{i-1}) \cdot \xi + x_{i-1} \tag{3.32}$$

3.4. Gesamtsteifigkeitsmatrix und Randbedingungen

Das Kapitel 3.4 veranschaulicht den Aufbau der Gesamtsteifigkeitsmatrix $\mathbf{K}$ und des Gesamtlastvektors $\mathbf{f}$. Aufgezeigt wird ebenso die Berücksichtigung der Randwerte in dem Gesamtgleichungssystem.

Die einfachste Möglichkeit zur Aufstellung der Gesamtsteifigkeitsmatrix $\mathbf{K}$ bzw. des Gesamtlastvektors $\mathbf{f}$ ist es, das in Gleichung (3.27) gezeigte Schema starr anzuwenden. Auf Grund der dort unterschiedlich auftretenden Integrale bei linearen Koeffizienten ist dies prinzipiell auch ein praktikabler Ansatz.

[5] Der Schritt der vorangegangenen partiellen Integration ist vertretbar, da durch die Numerik aus dem kontinuierlichem Intervall ein diskretes Intervall erzuegt wird.

Mehr Freiheit in der Gestaltung bzw. der Modellierung ist aber durch ein elementweises Einordnen der *lokalen* Steifigkeitsmatrizen in die *globale* Gesamtsteifigkeitsmatrix gegeben. Dieser zusätzlich Aufwand lohnt sich insofern, da es ermöglicht verschiedene Elementtypen frei miteinander zu kombinieren. Zum Beispiel können in Bereichen starker Oszillationen Elemente höherer Ordnung definiert werden.
Des Weiteren bringt es Vorteile im Bezug auf die Rechenzeit. So kann im Falle konstanter Koeffizienten die lokale Steifigkeitsmatrix $\mathbf{K}^{(i)}$ nur einmal integriert und anschließend beliebig oft in das Gesamtsystem eingesetzt werden.
Weitaus interessanter ist der Gedanke vor allem dahingehend, dass auf diese Art und Weise unterschiedliche Differentialgleichungssysteme relativ einfach miteinander gekoppelt werden können. Das ist unter anderem einer der Hauptgründe für den hohen Beliebtheitsgrad der Finiten Elemente in der Industrie. So kann zum Beispiel das Verhalten eines Balkens äußerst realitätsnah durch das Zusammenführen der Differentialgleichung eines Zugstabes mit der Differentialgleichung der Bernoulli Biegetheorie beschrieben werden. Es ist sogar gängige Praxis gewöhnliche Differentialgleichungen über die Knotenpunkte mit partiellen Differentialgleichungen zu verbinden. Als Beispiel sei die Auslegung eines Tisches genannt. Hierbei kann das Verhalten der vier Tischbeine mit je einem Balkenelement - wie es oben angedeutet wurde - beschrieben werden und die Tischplatte als Plattenelement über die partielle Differentialgleichung der Kirchhoff'schen Plattentheorie.

Je nach Anwendungsfall kommen hierfür verschiedene Assemblierungsverfahren zum Einsatz. Im Rahmen dieser Projektarbeit werden ausschließlich Elementsteifigkeitsmatrizen $\mathbf{K}^{(i)}$ für Differentialgleichungen der Form (3.19) der gleichen Ansatzfunktion berücksichtigt. Zur Herleitung des Algorithmus wird die Gleichung (3.27) exemplarisch ausgeführt:

$$\mathbf{K} = \begin{bmatrix} K_{1,1}^{(1)} & K_{1,2}^{(1)} & 0 & 0 & \cdots & 0 \\ K_{2,1}^{(1)} & K_{2,2}^{(1)} + K_{1,1}^{(2)} & K_{1,2}^{(2)} & 0 & \cdots & 0 \\ 0 & K_{2,1}^{(2)} & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & K_{1,2}^{(n-1)} & 0 \\ 0 & \cdots & 0 & K_{2,1}^{(n-1)} & K_{2,2}^{(n-1)} + K_{1,1}^{(n)} & K_{1,2}^{(n)} \\ 0 & \cdots & 0 & 0 & K_{2,1}^{(n)} & K_{2,2}^{(n)} \end{bmatrix} \tag{3.33}$$

$$\mathbf{f} = \begin{bmatrix} f_1^{(1)} \\ f_2^{(1)} + f_1^{(2)} \\ \vdots \\ 0 \\ f_2^{(n-1)} + f_1^{(n)} \\ f_2^{(n)} \end{bmatrix} \tag{3.34}$$

Es ist ersichtlich, dass der letzte Knoten eines Elementes mit dem ersten Knoten des angrenzenden Elementes auf der Diagonalen zusammenfällt. Die Aufgabe des Assemblierungsalgorithmus ist es daher die einzelnen lokalen Elementsteifigkeitsmatrizen sukzessiv auf der Hauptdiagonalen aneinanderzureihen. In der Literatur wird dieser Zusammenhang sehr gerne über sogenannte "Elementzusammenhangstabellen" veranschaulicht.

Elementnummer	globale Knotennummern im Bezug auf die lokale Knotennummer	
	1	2
1	0	1
2	1	2
3	2	3
⋮	⋮	⋮
n-1	n-1	n

Tabelle 3.1.: Elementzusammenhangstabellen

Lineare Ansatzfunktionen führen dementsprechend zu einer schwach besetzten Drei-Diagonalmatrix, quadratische zu einer Fünf-Diagonalmatrix und kubische zu einer Sieben-Diagonalmatrix. Der Lastvektor ist verhältnismäßig einfach aufgebaut und kann in einer ähnlichen Schleife assembliert werden.

Nach dem erfolgreichen Aufbau des Gesamtgleichungssystems ist es an der Zeit die Randwerte zu berücksichtigen. Hierfür wird zunächst der vernachlässigte Term aus der partiellen Integration in Kapitel 3.3 aufgegriffen. Durch Einsetzen der Grenzen und einer einfachen Umformung wird der Term auf die rechte Seite der Variationsformulierung gebracht.

$$\int_a^b \cdots \quad dx = \int_a^b f(x) \cdot \Phi(x) dx + p \cdot u'(b) \cdot \Phi(b) - p \cdot u'(a) \cdot \Phi(a) \tag{3.35}$$

Des Weiteren ergibt sich aus der Bedingung (3.12), dass $\Phi(a) = \Phi(b) = 1$ ist. Im Gesamtlastvektor wird dieser Umstand folgendermaßen berücksichtigt.

$$\mathbf{f} = \begin{bmatrix} f_1^{(1)} - p \cdot u'(a) \\ f_2^{(1)} + f_1^{(2)} \\ \vdots \\ 0 \\ f_2^{(n-1)} + f_1^{(n)} \\ f_2^{(n)} + p \cdot u'(b) \end{bmatrix} \tag{3.36}$$

Damit ist die Vorgehensweise zum Einbau sogenannter *Neumann* Randbedingungen gegeben. Zur Berücksichtigung der *Dirichlet* Randwerte wird zunächst das Gesamtgleichungssystem betrachtet. Für drei Elemente - ohne *Neumann* Bedingungen - gilt zum Beispiel:

$$\begin{bmatrix} K_{1,1}^{(1)} & K_{1,2}^{(1)} & 0 & 0 \\ K_{2,1}^{(1)} & K_{2,2}^{(1)} + K_{1,1}^{(2)} & K_{1,2}^{(2)} & 0 \\ 0 & K_{2,1}^{(2)} & K_{2,2}^{(2)} + K_{1,1}^{(3)} & K_{1,2}^{(3)} \\ 0 & 0 & K_{2,1}^{(3)} & K_{2,2}^{(3)} \end{bmatrix} \cdot \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \end{bmatrix} = \begin{bmatrix} f_1^{(1)} \\ f_2^{(1)} + f_1^{(2)} \\ f_2^{(2)} + f_1^{(3)} \\ f_2^{(3)} \end{bmatrix} \tag{3.37}$$

Ziel ist es nun das Gleichungssystem so zu modifizieren, dass am Ende gilt:

$$\hat{u}_1 = \eta \tag{3.38}$$

$$\hat{u}_4 = \delta \tag{3.39}$$

Es gibt viele Möglichkeiten das zu bewerkstelligen. Eine Möglichkeit ist das Streichen derjenigen Zeilen und Spalten in der Gesamtsteifigkeitsmatrix **K**, an denen der Knotenpunkt einen bestimmt Wert annehmen soll. In diesem Fall sind das die Knoten 1 und 4. Darüber hinaus erhalten die betroffenen Diagonalelemente den Wert 1. Werden abschließend die dazugehörigen Elemente im Gesamtlastvektor **f** durch den gewünschten Randwert ersetzt, ergibt sich folgender Aufbau:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & K_{2,2}^{(1)} + K_{1,1}^{(2)} & K_{1,2}^{(2)} & 0 \\ 0 & K_{2,1}^{(2)} & K_{2,2}^{(2)} + K_{1,1}^{(3)} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \end{bmatrix} = \begin{bmatrix} \eta \\ (f_2^{(1)} + f_1^{(2)}) - (\eta \cdot K_{2,1}^{(1)}) \\ (f_2^{(2)} + f_1^{(3)}) - (\delta \cdot K_{3,4}^{(3)}) \\ \delta \end{bmatrix} \tag{3.40}$$

Hierbei wurde der "Verlust" der Steifigkeit über eine manuelle Matrixmultiplikationsoperation im Gesamtlastvektor **f** berücksichtigt.[6]

[6] Der Schritt der manuellen Multiplikation könnte vernachlässigt werden, indem nur Zeilen gestrichen werden. Allerdings würde eine evtl. vorhandene Symmetrie verloren gehen.

3.5. Eigenwertprobleme

Bisher wurde der Ansatz der Finiten Elemente genutzt, um das Verhalten einer Differentialgleichung im Bezug auf eine gegebene Störung zu beschreiben. In den technisches Anwendung ist es aber oftmals interessant das Eigenverhalten des Systems ohne Störung zu betrachten. So spielen zum Beispiel in Schwingungsproblemen die sogenannten "Eigenfrequenzen" eine große Rolle.
In diesem Kapitel soll daher ein kurzer Ausblick in die Problematik der Eigenwertprobleme einer Differentialgleichung gegeben werden, bevor ein adaptiver Algorithmus vorgestellt wird.

Liegt die Differentialgleichung in der *Normalform* vor, so kann das Eigenwertproblem folgendermaßen definiert werden:

$$-u''(x) + q(x) \cdot u'(x) + r(x) \cdot u(x) = \lambda \cdot u(x) \tag{3.41}$$

Die Anwendung des Galerkin Ansatzes und die anschließenden Umformungen aus Kapitel 3.3 führen hierbei auf ein ähnliches Gleichungssystem. Ein Unterschied ist lediglich auf der rechten Seite gegeben.
Nachdem hier anstatt der gegebenen Funktion $f(x)$ nochmals die gesuchte Funktion $u(x)$ auftritt, entsteht dadurch eine weitere Matrix $\mathbf{M}$ im Gesamtgleichungssystem. Diese ist auch unter dem Begriff "Massenmatrix" bekannt.

$$\mathbf{K} \cdot \hat{\mathbf{u}} \;=\; \lambda \cdot \mathbf{M} \cdot \hat{\mathbf{u}} \tag{3.42}$$

Eine Umstellung der Gleichung ergibt das für Matrizen bekannte Eigenwertwertproblem.

$$\left(\mathbf{M}^{-1} \cdot \mathbf{K} - \lambda \cdot \mathbf{E}\right) \cdot \hat{\mathbf{u}} \;=\; 0 \tag{3.43}$$

Die sich daraus ergebenden Eigenwerte und Eigenvektoren stellen die verschiedenen Eigenzustände der Differentialgleichung dar. In der Schwingungslehre werden die Eigenwerte λ gerne über die Kreiszahl ω beschrieben. Hierfür gilt:

$$\omega_0 = 2 \cdot \pi \cdot f_0 \;=\; \sqrt{\lambda} \tag{3.44}$$

3.6. Fehlerschätzung und Adaptivität

Neben der Richtigkeit einer numerischen Berechnung ist vor allem die erreichte Qualität von großer Bedeutung. Ein Maß hierfür ist der gemachte Fehler während der Simulation. Um diesen zu bestimmen, gibt es in der Numerik grundsätzlich zwei verschiedene Arten. Zum einen die *a-priori* Fehlerabschätzung und zum anderen die *a-posteriori* Fehlerabschätzung. Die erste Variante schätzt den zu erwartenden Fehler bereits im Vorfeld ab. Als Kriterien dienen hierfür zum Beispiel die Diskretisierung des Intervalls oder die Beschaffenheit des Ausgangsproblems. Leider sind die berechneten Ergebnisse nicht immer verlässlich.
Attraktiver sind die *a-posteriori* Fehlerabschätzungen. Diese Beruhen auf bereits berechneten Ergebnissen und geben daher signifikante Aussagen über die Qualität der gefundenen Lösung.

Der Sinn einer Adaptivität ist es nun, den Fehler während der Programmlaufzeit zu minimieren. Hierfür werden bei der FEM die *h-Methode* und die *p-Methode* genutzt. Die Idee dahinter ist, das Problem zunächst zweimal zu berechnen. Einmal mit beliebig gewählten Annahmen - im Bezug auf die Diskretisierung und der verwendeten Ansatzfunktionen - und ein zweites mal mit vermeintlich besseren numerischen Eigenschaften. Aus den beiden gewonnen Ergebnissen kann anschließend ein Fehler ermittelt werden.
Entspricht dieser nicht den Genauigkeitsanforderungen, so wird eine weitere Iteration mit noch feineren Eigenschaften durchgeführt. Darüber hinaus besteht die Option den Fehler nicht global für das Gesamtintervall zu berechnen, sondern lokal für die einzelnen Teilelemente. Auf diese Weise ist es möglich die Verfeinerungen nur in den betroffenen Teilgebieten durchzuführen.
Es ist offenkundig, dass bei einem adaptiven Algorithmus weitaus höhere Rechenzeiten zu erwarten sind.

Der in dieser Projektarbeit angewandte Algorithmus basiert ausschließlich auf der *h-Methode* mit linearen Ansatzfunktionen und kann schließlich in folgende Hauptaufgaben gegliedert werden.

1. Berechnung des Hauptproblems
2. Berechnung weiterer lokaler Unterprobleme
3. Durchführung der lokalen Fehlerabschätzungen
4. Markierung schlechter Elemente
5. Verfeinerung der markierten Elemente

Der erste Punkt wurde bereits in den vorherigen Kapiteln ausführlich diskutiert und wird daher nicht weiter beachtet. In einem nächsten Schritt soll die aktuell vorliegende Lösung in kleinere Unterprobleme aufgespalten werden. Der Grundgedanke hierbei ist es, um jeden Knotenpunkt i ein eigenes Randwertproblem auf dem Gebiet $[x_{i-1},x_{i+1}]$ zu formen. Als Dirichlet-Randwerte können jeweils die "alten" Lösungen $\hat{u}_{i-1}$ bzw. $\hat{u}_{i+1}$ angenommen werden.
Sowohl der erste als auch der letzte Knoten nehmen hier eine Sonderstellung ein, da es in der Ausgangslösung keinen Knoten x_{a-1} bzw. x_{b+1} gibt. Hier werden die Intervalle auf die Bereiche $[a,x_{a+1}]$ und $[x_{b-1}, b]$ begrenzt. In Abbildung 3.9 wird dieser Sachverhalt veranschaulicht.

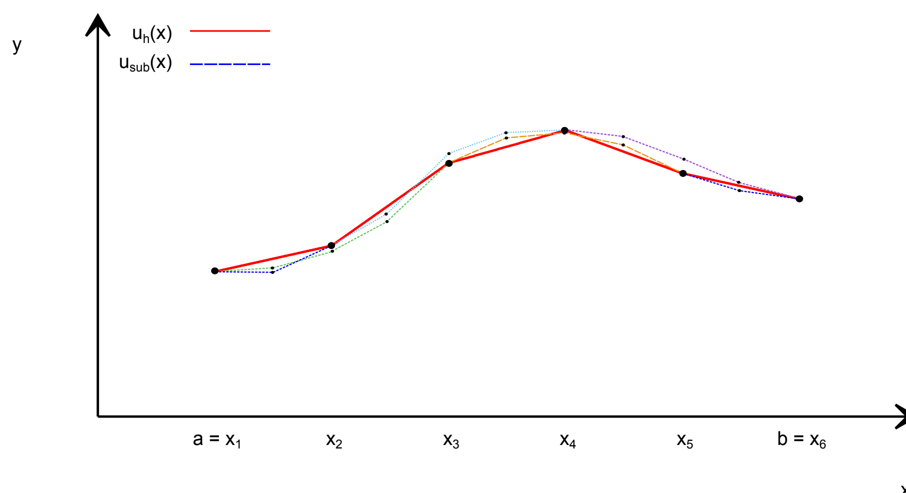

Abbildung 3.9.: Lokale Unterteilung in Sub-Probleme

Es ist zu sehen, dass auf diese Weise für jedes Ausgangselement zwei Vergleichslösungen entstehen. Die Elementanzahl einer neu gewonnenen Unterlösung kann hierbei beliebig variieren. In der Abbildung 3.9 sind das vier Elemente pro Unterproblem und damit zwei Elemente pro "altes" Element.
Wird nun die Abweichung zwischen jedem Ausgangselement und den zwei Unterproblemen bestimmt, so ergibt sich daraus eine Art Fehler, der als Qualitätsmerkmal herangezogen werden kann.
Es gibt verschiedene Möglichkeiten derartige Abweichungen zu bestimmen. In dieser Projektarbeit soll das über die Funktionsnorm des sogenannten *Sobolev*-Raumes $H^1(a, b)$ geschehen.
Dieser Raum stellt eine Menge aller Funktionen des Raumes $L_2(a, b)$ [7] dar, die eine erste verallgemeinerte Ableitung besitzen, welche ebenfalls Element des $L_2(a, b)$ sind.
Eine detailliertere Beschreibung der Herkunft dieser Funktionsräume würde den Rahmen einer Projektarbeit übersteigen. Daher sei lediglich erwähnt, dass die Basisfunktionen $\phi_i(x)$ in der FEM Elemente des Sobolev-Raumes sind.

[7] Lebesgue-Raum L_p: Menge aller p-fach integrierbaren Funktionen

Eine Funktionsnorm ist eine charakteristische positive skalare Größe, die einer Funktion aus einem Funktionenraum eine gewisse "Größe" zuordnet.[8] Die H^1-Norm $\| \bullet \|_{1,(a,b)}$ auf dem Intervall $[a, b]$ ist folgendermaßen definiert:

$$\|f(x)\|_{1,(a,b)} = \sqrt{\int_a^b \left[(f(x))^2 + (f'(x))^2 \right] dx} \tag{3.45}$$

Eine elementare Eigenschaft von Normen wird über die Dreiecksungleichung beschrieben. Diese besagt ganz allgemein für Funktionsnormen:

$$\left| \; \|f(x)\| - \|g(x)\| \; \right| \leq \|f(x) - g(x)\| \tag{3.46}$$

Abstrahiert auf die obige Situation ergibt sich, dass

$$\epsilon_i = \sqrt{\int_{x_{i-1}}^{x_i} \left[(u_h(x) - u_{h,sub}(x))^2 + \left(u'_h(x) - u'_{h,sub}(x) \right)^2 \right] dx} \tag{3.47}$$

der maximal auftretende Fehler zwischen den beiden gefundenen Lösungen liegt. Übersteigt der nun ermittelte Fehler ϵ_i bei einem Element die geforderte Toleranz, so wird dieser Bereich markiert und später in mindestens zwei Elemente unterteilt. Anschließend startet der Algorithmus von vorne und es wird eine neue Ausgangslösung mit der nun neu definierten Diskretisierung berechnet. Dieser Zyklus wird so oft durchlaufen, bis keine Elemente mehr zu markieren sind.

[8] Analogie zur Vektoralgebra: Die Norm eines Vektors in einem Vektorraum ist seine Länge.

4. Implementierung

Nachdem in den vorangegangen Kapiteln das Konzept der Finiten Elemente und deren Hilfsmittel ausführlich diskutiert wurden, ist es Zeit die Theorie in der Praxis anzuwenden. Hierfür sind die numerischen Methoden als Algorithmen in der Programmiersprache C++ implementiert worden. Das Ergebnis und damit der Programmcode selbst, ist im Anhang abgedruckt. Das nachfolgende Kapitel dient daher lediglich zur Orientierung und beschreibt dahingehend die Grundstruktur der Bibliothek und deren wichtigsten Eigenschaften.

Ganz im Stil der objektorientierten Programmierung, sind alle Algorithmen als Methoden in der Klasse *hmFem* hinterlegt. Des Weiteren besitzt *hmFem* unter anderem die private Membervariable *lins* des Typs *linFunctions*, die ausschließlich über den Konstruktor bei der Objektinstanzierung gesetzt werden kann und gesetzt werden muss.
Im Algorithmus 4.1 ist eine auf das Wesentliche reduzierte Form des Deklarationsteils der Klasse *hmFem* gegeben.

```
typedef vector<long double> vec1d;
typedef vector<vec1d> vec2d;

class hmFem {
  // M E M B E R
  private:
    linFunctions&   lins; double xim1, xi, h; int feOrder, m, n;

  // K O N S T R U K T O R
  public:
    hmFem( linFunctions& linf );

  // M E T H O D E N
  public:
    vec2d fem1d_sturmLiouvilleAdp(double a, double b, int bcType[2], double bcVal[2]);
    vec2d fem1d_sturmLiouville(   double a, double b, int ffeOrder, int N,
                    int bcType[2], double bcVal[2], const vec1d &xVals = vec1d() );
  private:
    double  phi( double x, int node, int der, int order );
    double  quadR( double (hmFem::*f) (double), double a, double b, double eps, int maxIt);
    vec1d   GLS( vec2d A, vec1d b);
    vec2d   addToGSM( vec2d A_l, int e, vec2d A );
    vec1d   addToGSV( vec1d v_l, int e, vec1d v );
    vec2d   getK_loc( double (hmFem::*LHSvariationsF)(double x)  );
    vec1d   getF_loc( );
    double  RHS_f(double x);
    double  LHS_sturmLiouville(double x);
};
```

Algorithmus 4.1: Reduzierte *hmFem* Deklaration

Der Datentyp *linFunctions* ist ebenfalls eine für das *hmFem* - Projekt geschriebene Klasse. Diese dient sozusagen als Sammelstelle für Funktionen, die während der Laufzeit ausgeführt werden sollen, aber erst noch von einem späteren Anwender definiert werden müssen. Es stellt vereinfacht gesagt das Interface für die linearen Koeffizienten $p(x)$, $q(x)$, $r(x)$ und $f(x)$ dar.

```
class linFunctions {
  public:
    virtual double funPx(double x)    = 0;
    virtual double funQx(double x)    = 0;
    virtual double funRx(double x)    = 0;
    virtual double funFx(double x)    = 0;
};
```

Algorithmus 4.2: Klasse *linFunctions*

Über diesen Umweg ist es nun für den Benutzer möglich eigene Koeffizienten zu bestimmen. Hierzu muss lediglich eine weitere Klasse in einem aufrufenden Programm definiert werden, die von der Klasse *linFunctions* abgeleitet ist. Dank der Vererbung können so die vier Funktionen überladen werden. Bei der darauffolgenden Objektinstanzierung der Klasse *hmFem* wird die eben erzeugte Kindklasse an den Konstruktor übergeben.

```
int main(){
  //Erzeugung einer Kindklasse, abgeleitet von linFunctions
  class Koeff : public linFunctions {
    public:
      //Ueberladen der Funktionen
      double funPx(double x) {  return -20; }
      double funQx(double x) {  return x; }
      double funRx(double x) {  return 100*x; }
      double funFx(double x) {  return 0;}
  };

  //Objektinstanzierung der Kindklasse = "Sammelstelle"
    Koeff    coeff;          // Instanz mit: p(x), q(x), r(x) und f(x)

  //Objektinstanzierung von hmFem
    hmFem fe(coeff);

  return 0;
}
```

Algorithmus 4.3: Definition linearer Koeffizienten

Als eigentliche Schnittstelle nach außen fungieren letztendlich die öffentlichen Methoden *fem1d_sturmLiouville()* und *fem1d_sturmLiouvilleAdp()*. Alle weiteren Methoden und Variablen sind für den User nicht sichtbar und damit auch nicht modifizierbar.

Das Kernstück der FEM besteht aus dem Aufbau der Gesamtsteifigkeitsmatrix $\mathbf{K}$ und des Gesamtlastvektors $\mathbf{f}$. Daher sollen anhand des Ablaufs von *fem1d_sturmLiouville()* die hierfür verwendeten privaten Hauptmethoden kurz vorgestellt werden. [9]
Beim Einstieg in die Routine werden nach einer kurzen Plausibilitätsprüfung einige Hilfsvariablen aus den Übergabewerten bestimmt. Dazu gehören zum Beispiel die Anzahl der lokalen bzw. globalen Freiheitsgrade *DOF_l* bzw. *DOF_g* und die Ordnung der zu benutzenden Ansatzfunktionen *feOrder*. Die Ordnung wird hierbei in einer privaten Membervariablen abgelegt, da diese Information später an unterschiedlichen Stellen von Bedeutung ist. Anschließend läuft eine Schleife über die Anzahl der gewünschten Elemente N und integriert pro Element die lokale Steifgkeitsmatrix $\mathbf{K}^{(i)} = K_l$ und den lokalen Lastvektor $\mathbf{f}^{(i)} = f_l$. Diese können innerhalb der Schleife direkt über den Assemblierungsalgorithmus *addToGSM()* bzw. *addToGSV()* in die globale Gesamtsteifigkeitsmatrix $\mathbf{K} = K$ bzw. in den globalen Gesamtlastvektor $\mathbf{f} = f$ eingeordnet werden.

```
this->feOrder   = ffeOrder;                 //Ansatzfunktion: 1=lin, 2=quad, 3=kub
DOF_l           =  this->feOrder + 1;       //lokale Anzahl an Freiheitsgraden
DOF_g           = (this->feOrder * N) + 1;  //globale Anzahl an Freiheitsgraden

for ( int e = 1; e <= N; e++ ) {
  if ( xVals.size() > 0  ) {
    this->xim1  = xVals[e-1];
    this->xi    = xVals[e];
  }
  else {
    this->xim1  = a+((e-1)  *  (b-a)/N  );
    this->xi    = a+( e     *  (b-a)/N  );
  }
  this->h     = fabs( this->xi - this->xim1 );

  K_l = this->getK_loc( this->LHS_sturmLiouville );
  K   = this->addToGSM( K_l, e, K  );

  f_l = this->getF_loc(  );
  f   = this->addToGSV( f_l, e, f  );
}
```

Algorithmus 4.4: Erstellung der Matrix $\mathbf{K}$ und des Vektors $\mathbf{f}$

Zudem werden in der Schleife die Koordinaten $x_{i-1} = xim1$ und $x_i = xi$ des aktuellen Elementes bestimmt. Hierbei wird zunächst geprüft ob der optionale Parameter $xVals$ vorhanden ist, denn dieser ermöglicht es die Knotenkoordinaten und damit eine individuelle Diskretisierung über einen Vektor vorzugeben. Andernfalls wird mit Hilfe der gewünschten Elementanzahl N und den Intervallgrenzen a und b ein äquidistantes Gitter errechnet.

[9] Alle weiteren Methoden sind im Bezug auf den Code selbsterklärend und im Anhang gegeben.

In den Methoden *getK_loc()* und *getF_loc()* wird folglich für jedes Element die numerische Quadratur von Romberg *quadR()* aufgerufen. Diese integriert jeweils die übergebene Funktion auf dem Intervall $[0,1]$ mit einer gewünschten Genauigkeit von 0.0001. Hierfür werden maximal fünf Extrapolationsschritten gestattet.
Die Funktion *LHSvariationsForm* wurde im Algorithmus 4.4 an *getK_loc()* übergeben und stellt in dieser Projektarbeit das *Sturm-Liouville* Probelm dar.

```
for (int i=0; i<(feOrder+1); i++) {
  for (int j=0; j<(feOrder+1); j++) {
    this->m = i+1;  //Array geht bei Null los, aber Ansatzfunktionen bei 1!
    this->n = j+1;
    K_l[i][j] = K_l[i][j] + this->quadR( LHSvariationsForm, 0, 1, 0.0001, 5);
  }
}
```

Algorithmus 4.5: Elementweise Integration der lokalen Steifigkeitsmatrix $\mathbf{K}^{(i)}$

```
for (int i=0; i<(feOrder+1); i++) {
  this->m = i+1;  //Array geht bei Null los, aber Ansatzfunktion bei 1!
  f_l[i] = f_l[i] + this->quadR( this->RHS_f, 0, 1, 0.0001, 5);
}
```

Algorithmus 4.6: Elementweise Integration des lokalen Lastvektors $\mathbf{f}^{(i)}$

In den privaten Membervariablen m und n werden die aktuellen Elementindizes abgespeichert. Der Grund wird in der nachfolgenden Variationsformulierung ersichtlich.

```
double hmFem::LHS_sturmLiouville(double x) {
  double p   =  this->lins.funPx(  (x*this->h)+this->xim1  );
  double q   =  this->lins.funQx(  (x*this->h)+this->xim1  );
  double r   =  this->lins.funRx(  (x*this->h)+this->xim1  );

  return
     (1/this->h) * p *
        this->phi(x, this->n, 1, this->feOrder) * this->phi(x, this->m, 1, this->feOrder) +
     q *
        this->phi(x, this->n, 1, this->feOrder) * this->phi(x, this->m, 0, this->feOrder) +
     this->h *  r *
        this->phi(x, this->n, 0, this->feOrder) * this->phi(x, this->m, 0, this->feOrder);
}
```

Algorithmus 4.7: Variationsformulierung (*Left-Hand-Side*)

```
double hmFem::RHS_f(double x) {
  double f =  this->lins.funFx(  (x*this->h)+this->xim1   );
  return      this->h * f * this->phi(x, this->m, 0, this->feOrder);
}
```

Algorithmus 4.8: Variationsformulierung (*Right-Hand-Side*)

Die Funktionen für die linke Seite *LHSvariationsForm* und für die rechte Seite *RHS_f* wurden aus den Gleichungen (3.28) und (3.29) abgeleitet.

Auf Grund der Integration über das Intervall $[0,1]$, müssen die in *lins* enthaltenen Koeffizientenfunktionen *p(x), q(x), r(x) und f(x)* mit Hilfe der Gleichung (3.32) aus dem Gebiet $[a,b]$ auf $[0,1]$ transformiert werden.
Anstatt dessen werden über die Methode *phi()* die zugehörigen Ansatzfunktionen direkt an der Stelle $x \in [0,1]$ ausgeführt. Die Selektion der entsprechenden Ansatzfunktion erfolgt hierbei zum einen über die Ordnung der Ansatzfunktion *feOrder* und zum anderen über die jeweilig Knotennummer m bzw. n und der gewünschten Differentiationsstufe.

Die entstandene lokale Steifigkeitsmatrix *K_l* und der entstandene lokale Lastvektor *f_l* werden nun über die Assemblierungsverfahren *addToGSM()* und *addToGSV()* in ihre jeweiligen globalen Endpositionen eingeordnet.

```
  vec2d hmFem::addToGSM( vec2d K_l, int e, vec2d K ) {
    int DOF_l = K_l.size();
    for (int i=0; i<DOF_l; i++) {
      for (int j=0; j<DOF_l; j++) {
        K[i+((e-1)*this->feOrder)][j+((e-1)*this->feOrder)] =
          K[i+((e-1)*this->feOrder)][j+((e-1)*this->feOrder)] + K_l[i][j];
      }
    }
    return K;
  }


  vec1d hmFem::addToGSV( vec1d f_l, int e, vec1d f ) {
    for (int i=0; i<(this->feOrder+1); i++) {
      f[i+((e-1)*this->feOrder)] = f[i+((e-1)*this->feOrder)] + f_l[i];
    }
    return f;
  }

}
```

Algorithmus 4.9: Assemblierungsalgorithmen

Damit ist der Algorithmus 4.4 abgeschlossen und der Einbau der Randbedingungen kann in Anlehnung an Kapitel 3.4 beginnen.
Für die *Neumann* Randbedingung am linken bzw. rechten gilt:

```
  //links
  if (bcType[0] == 1) {
    f[0] = f[0] - (this->lins.funPx(  (a*this->h)+this->xim1  )*bcVal[0]);
  }
  //rechts
  if (bcType[1] == 1) {
    f[DOF_g-1] = f[DOF_g-1] + (this->lins.funPx(  (b*this->h)+this->xim1  )*bcVal[1]);
  }
```

Algorithmus 4.10: Einbau der Neumann Randbedingungen

Im Falle der *Dirichlet* Bedingungen:

```
//links
if (bcType[0] == 0) {
  f[0] = bcVal[0];
  f[1] = f[1] - bcVal[0] * K[1][0];
  if (DOF_g > 2) {
    f[2] = f[2] - bcVal[0] * K[2][0];
  }
  if (DOF_g > 3) {
    f[3] = f[3] - bcVal[0] * K[3][0];
  }
  for ( int i=0; i<DOF_g; i++) {
    K[0][i] = 0.0; K[i][0] = 0.0;
  }
  K[0][0] = 1;
}

//rechts
if (bcType[1] == 0) {
  f[DOF_g-1] = bcVal[1];
  f[DOF_g-2] = f[DOF_g-2] - bcVal[1] * K[DOF_g-2][DOF_g-1];
  if (DOF_g > 2) {
    f[DOF_g-3] = f[DOF_g-3] - bcVal[1] * K[DOF_g-3][DOF_g-1];
  }
  if (DOF_g > 3) {
    f[DOF_g-4] = f[DOF_g-4] - bcVal[1] * K[DOF_g-4][DOF_g-1];
  }
  for ( int i=0; i<DOF_g; i++) {
    K[DOF_g-1][i] = 0.0; K[i][DOF_g-1] = 0.0;
  }
  K[DOF_g-1][DOF_g-1] = 1;
}
```

Algorithmus 4.11: Einbau der Dirichlet Randbedingungen

Zu guter Letzt wird das nun entstandene Gleichungssystem über den Aufruf von *GLS()* gelöst. Je nach der Beschaffenheit von **K** wird hierfür entweder die LU- oder die Cholesky Zerlegung herangezogen.
Für den Anwender wird im einem letzten Schritt das Ergebnisfeld *rtrn* aufgebaut.

```
u = this->GLS( K, f);

for ( int i=0; i<DOF_g; i++ ) {
  rtrn[i][0] = x[i];
  rtrn[i][1] = u[i];
}
return rtrn;
```

Algorithmus 4.12: Lösen des Gleichungssystems und Rückgabe an den Anwender

5. Anwendung

Abschließend sollen hier noch Anwendungsbeispiele gezeigt werden, die mit Hilfe der C++ Bibliothek gelöst werden können. Darüber hinaus werden die gefundenen Ergebnisse mit denen der numerischen Berechnungssoftware *MATLAB* verglichen.
Zum Lösen von eindimensionalen Randwertproblemen bietet das Unternehmen *The MathWorks* die Funktion *bvp4c* an. Dahinter verbirgt sich ein adaptiver Algorithmus, der ebenfalls auf der Methode der gewichteten Residuen beruht. Allerdings wird das Residuum hier nicht über den Galerkin Ansatz minimiert, sondern über ein sogenanntes *Kollaktionsverfahren.*

Differentialgleichungen zweiter Ordnung beschreiben in der Physik häufig Schwingungsprobleme. Abbildung 5.1 zeigt schematisch die Herleitung der allgemeinsten Form der Schwingungsdifferentialgleichung.

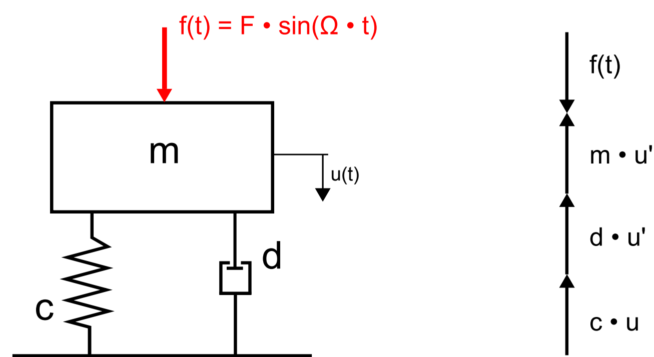

Abbildung 5.1.: Periodisch angeregtes System

Hieraus ergibt sich eine Sonderform des bekannten *Sturm-Liouville* Problems:

$$m \cdot u''(t) + d \cdot u'(t) + c \cdot u(t) \quad = \quad F \cdot sin(\Omega \cdot t) \tag{5.1}$$

m	$[kg]$	Masse	F	$[N]$	Kraftamplitude
d	$[Ns/m]$	Dämpfungskonstante	Ω	$[1/s]$	Anregungsfrequenz
c	$[N/m]$	Federkonstante	t	$[s]$	Zeit

Auf der nachfolgenden Seite sind die Ergebnisse vier willkürlich gewählter Beispiele zu sehen. Des Weiteren ist der aufrufende Programmcode des ersten Beispiels abgedruckt.

5.1. Anwendungsbeispiel 1

$$
\begin{gathered}
20 \cdot u''(t) + 25 \cdot u'(t) + 100 \cdot u(t) = 3000 \cdot sin(4 \cdot t) \quad t \in [0, 5] \\
u(0) = 8 \qquad u(5) = -1
\end{gathered}
\tag{5.2}
$$

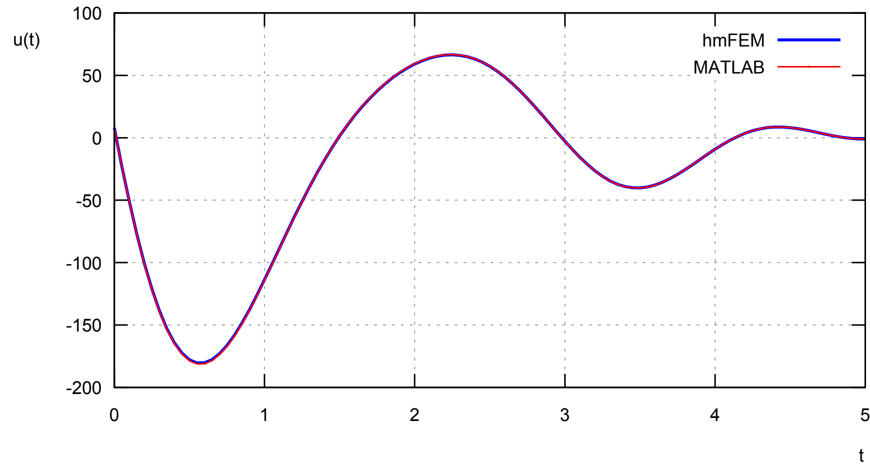

Abbildung 5.2.: Anwendungsbeispiel 1

5.2. Anwendungsbeispiel 2

$$
\begin{gathered}
20 \cdot u''(t) + (100 - t) \cdot u(t) = 3000 \cdot sin(4 \cdot t) \quad t \in [0, 5] \\
u(0) = 15 \qquad u(5) = -10
\end{gathered}
\tag{5.3}
$$

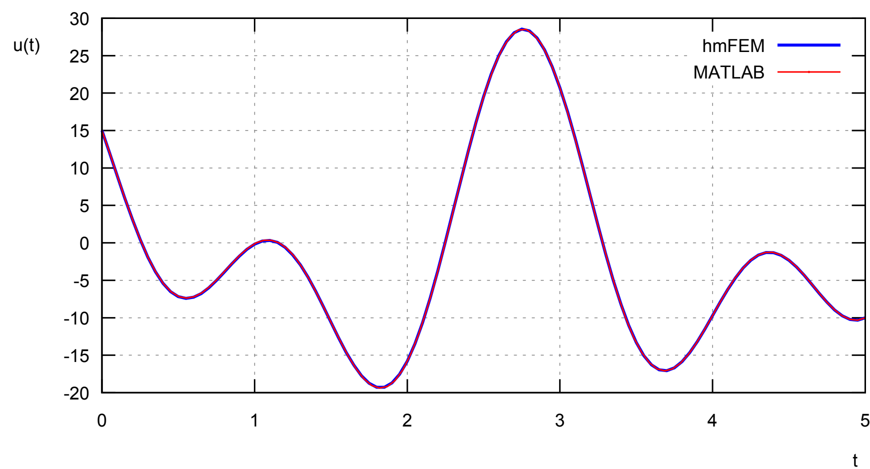

Abbildung 5.3.: Anwendungsbeispiel 2

5.3. Anwendungsbeispiel 3

$$20 \cdot u''(t) + (t^2 - t) \cdot u'(t) + (100 + t^3) \cdot u(t) = 3000 \cdot sin(4 \cdot t) \quad t \in [0, 5]$$
$$u(0) = 15 \qquad u'(5) = 0 \tag{5.4}$$

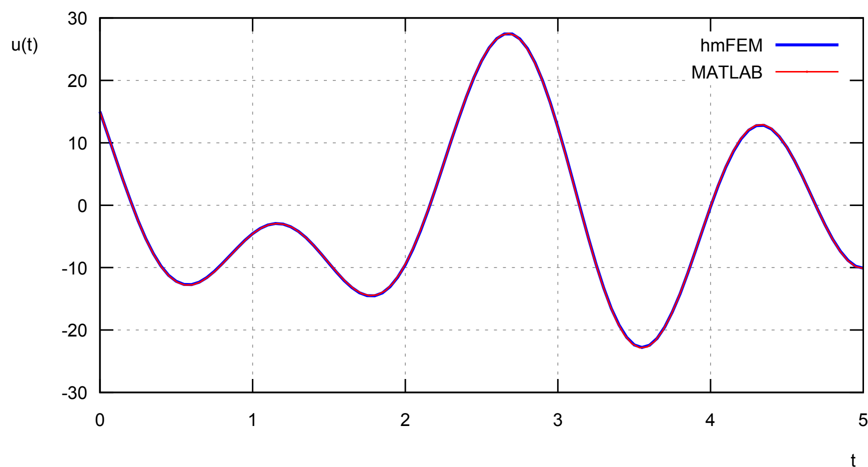

Abbildung 5.4.: Anwendungsbeispiel 3

5.4. Anwendungsbeispiel 4

$$20 \cdot u''(t) + t \cdot u'(t) + (100 \cdot t) \cdot u(t) = 0 \qquad t \in [0, 5]$$
$$u'(0) = 10 \qquad u'(5) = 0 \tag{5.5}$$

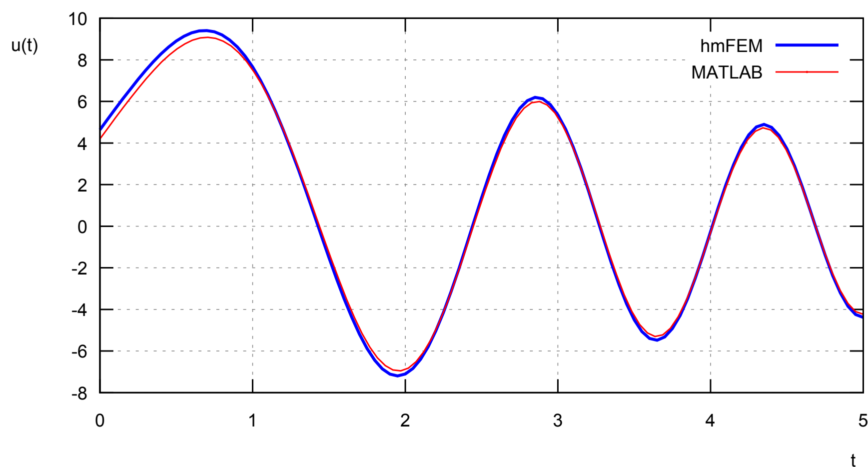

Abbildung 5.5.: Anwendungsbeispiel 4

5.5. Aufrufendes Programm

```
#include <math.h>
#include <fstream>
#include <iostream>
#include "../../hmFEM/hmFEM.h"
using namespace std;
typedef vector<long double> vec1d;
typedef vector<vec1d> vec2d;

int main(){
    class Koeff : public linFunctions {
        public:
            double funPx(double x) {  return -20; }
            double funQx(double x) {  return 25; }
            double funRx(double x) {  return 100; }
            double funFx(double x) {  return 3000*sin(4*x);}
    };

    cout << "hmFem: Beispiel-Anwendung" << endl;

    double  a           = 0;    // Intervallgrenze (Start)
    double  b           = 5;    // Intervallgrenze (Ende)
    Koeff   coeff;              // Klasse mit: p(x), q(x), r(x) und f(x)
    int     feOrder     = 1;    // Ansatzfunktion: 1=lin, 2=quad, 3=kub
    int     numE        = 100;  // Anzahl der Elemente
    int     bcT[2];             // Typ der Randbedingung
    double  bcV[2];             // Wert der Randbedingung
    vec2d   sol;                // Ergebnisfeld

    // Randbedingungen setzen:
    //links/a           rechts/b
    bcT[0] = 0;         bcT[1] = 0;     //0=dirichlet, 1=neumann
    bcV[0] = 8;         bcV[1] = -1;    //wert=wert

    //Berechnung
    hmFem fe(coeff);
    sol = fe.fem1d_sturmLiouville( a, b, feOrder, numE, bcT, bcV);

    return 0;
}
```

Algorithmus 5.1: Aufrufendes Programm bzgl. Beispiel 1

6. Fazit

Die Projektarbeit hat gezeigt, dass die Methode der Finiten Elemente ein sehr anschauliches und stabiles Verfahren ist. So ist am Ende eine C++ Bibliothek entstanden, die Sturm Liouville Probleme zuverlässig löst. Darüber hinaus wurden die ersten Grundbausteine für eine Adaptivität - basierend auf der *h-Methode* - gesetzt. Auch wenn diese im Falle linearer Koeffizienten den Aufwand nicht rechtfertigt.

Dessen ungeachtet ist das Kontingent der FEM bei weitem noch nicht erschöpft. So können für weiterführende Projektarbeiten an der Hochschule München folgende Themengebietem in Betracht gezogen werden:

1. Lösen von Eigenwertproblemen (Modal- und Stabilitätsanalysen)
2. Berücksichtigung nicht-linearer Problemstellungen
3. Erstellung weiterer Variationsformulierungen für DGL höherer Ordnung
4. Erweiterung der Ansatzfunktionen in Hinblick auf hermitische Polynome
5. Verbesserung der Adaptivität (*hp-Methode*)
6. Implementierung partieller Differentialgleichungen

Abschließend möchte ich mich bei Herrn Prof. Dr.-Ing. Küpper für die Betreuung und Unterstützung während dieser Arbeit bedanken.

Literatur

[1] K.J. Bathe und P. Zimmermann. *Finite-Elemente-Methoden -*. 2. vollst. neu bearb. u. erw. Aufl. 2002. Berlin, Heidelberg: Springer, 2002. ISBN: 978-3-540-66806-0.

[2] Michael Jung und Ulrich Langer. *Methode der finiten Elemente für Ingenieure - Eine Einführung in die numerischen Grundlagen und Computersimulation*. 2. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2012. ISBN: 978-3-658-01101-7.

[3] Bernd Klein. *FEM - Grundlagen und Anwendungen der Finite-Element-Methode im Maschinen- und Fahrzeugbau*. 10. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2014. ISBN: 978-3-658-06054-1.

[4] Prof. C. Rhode. *Skript: Numerik II*. Universität Bielefeld, 2005.

[5] Prof. Dr.-Ing. K. Warendorf. *Skript: Numerische Verfahren*. Hochschule München, 2015.

A. hmFem.h

B. hmFem.cpp