BACHELORARBEIT

Py1AFEM: eine Implementierung adaptiver FEM in Python

ausgeführt am

Institut für Analysis und Scientific Computing
TU Wien

unter der Anleitung von

Prof. Dr. Dirk Praetorius
Dipl.-Ing. Michael Innerberger

durch

Maximilian Lackner
Matrikelnummer: 11914689

Wien, September 2023

Danksagungen

Ich möchte Martin Vetter für seine wertvolle Unterstützung während meines Projekts danken. Seine Hilfe hat mir sehr geholfen und es wäre ohne ihn schwierig gewesen, das Projekt erfolgreich abzuschließen.

Ich möchte meiner Frau und meinen Eltern für ihre anhaltende Unterstützung und Ermutigung danken, die mir bei der Fertigstellung dieser Arbeit geholfen haben.

Ich möchte mich auch bei allen anderen Personen bedanken, die dazu beigetragen haben, dieses Projekt zu einem Erfolg zu machen. Ihre Hilfe, ihre Gedanken und ihre Ideen waren wertvoll und haben zu einem besseren Ergebnis beigetragen.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am 25.09.2023

Max Lackner

Maximilian Lackner

Abstract

Ziel der Arbeit ist, eine Python-Implementierung der adaptiven Finite–Elemente–Methode (FEM) in 2D für allgemeine elliptische partielle Differentialgleichungen in Divergenzform zu präsentieren. Zunächst wird das Poisson-Problem untersucht, bevor die allgemeine P1-Galerkin FEM implementiert wird. Es wird gezeigt, dass der zu Beginn verwendete Python-Code eine quadratische Laufzeit hat, aber durch die Verwendung von Vektorisierung ein Code mit linearer Laufzeit erhalten werden kann. Das Paket enthält vektorisierte Implementierungen von der allgemeinen P1-Galerkin FEM, der Netzverfeinerung mittels Newest vertex bisection (NVB), eines Fehlerschätzers und eines adaptiven Netzverfeinerungsalgorithmus. Außerdem wird dafür eine zusätzliche Implementierung benötigt, um die Kanten zu nummerieren, welche auch in Form von `provideGeometricData` in dieser Arbeit umgesetzt ist.

Die Arbeit zeigt, dass die Implementierung von adaptiver FEM in Python eine effektive Methode zur Lösung elliptischer partieller Differentialgleichungen zweiter Ordnung darstellt. Die Verwendung von Vektorisierung zur Integration von Fehlerschätzern und Netzverfeinerung tragen dazu bei, dass der Algorithmus schnell eine sehr gut approximierte Lösung ausgeben kann. Numerische Experimente belegen, dass die Implementierungen auf optimale Konvergenzrate bzgl. der Rechenzeit führen.

Inhaltsverzeichnis

1 Einleitung

Als typisches Problem der Finite Elemente Methode (FEM) betrachten wir die Poisson-Gleichung mit gemischten Dirichlet- und Neumann-Randbedingungen: Gegeben seien $f \in L^2(\Omega), u_D \in H^{1/2}(\Gamma_D)$ und $g \in L^2(\Gamma_N)$. Hier nehmen wir an, dass $\Omega \subset \mathbb{R}^2$ ein beschränktes Lipschitz-Gebiet ist mit Rand $\Gamma := \partial\Omega$, welchen man in einen abgeschlossenen Dirichlet-Rand Γ_D mit positiver Länge und in einen Neumann-Rand $\Gamma_N := \Gamma \setminus \Gamma_D$ unterteilen kann. Wir versuchen nun eine Approximation der Lösung $u \in H^1(\Omega)$ für das folgende Problem zu berechnen:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= u_D \quad \text{auf } \Gamma_D, \\ \nabla u \cdot n &= g \quad \text{auf } \Gamma_N. \end{aligned} \tag{1}$$

Um dieses Problem zu lösen, wollen wir zuerst versuchen es zu vereinfachen. Dafür nehmen wir eine beliebige Fortsetzung $u_D \in H^1(\Omega)$ von $u_D \in H^{1/2}(\Gamma_D)$ und subtrahieren diese von der gesuchten Lösung u

$$u_0 := u - u_D \in H^1_D(\Omega) := \{v \in H^1(\Omega) \colon v = 0 \text{ auf } \Gamma_D\}. \tag{2}$$

Jetzt können wir die schwache Formulierung von Problem (1) aufstellen: Wir suchen $u_0 \in H^1_D(\Omega)$ sodass gilt:

$$\int_\Omega \nabla u_0 \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x + \int_{\Gamma_N} g v \, \mathrm{d}s - \int_\Omega \nabla u_D \cdot \nabla v \, \mathrm{d}x \quad \text{für alle } v \in H^1_D(\Omega). \tag{3}$$

Auf dieses Problem können wir wie gewohnt das Lemma von Lax–Milgram anwenden. Hierfür brauchen wir davor noch die Definition von Stetigkeit und Koerzivität [4].

Definition 1 (stetige und koerzive Bilinearform [2])**.**
Seien H ein Hilbert-Raum und $a : H \times H \to \mathbb{R}$ eine Bilinearform. Wir nennen $a(\cdot,\cdot)$ stetig, wenn eine Konstante $K > 0$ existiert, so dass gilt

$$|a(u,v)| \le K\|u\|_H\|v\|_H \quad \textit{für alle } u, v \in H.$$

Die Bilinearform heißt koerziv, wenn eine Konstante $\kappa > 0$ existiert, so dass gilt

$$a(u,u) \ge \kappa\|u\|_H^2 \quad \textit{für alle } u \in H.$$

Satz 1 (Lemma von Lax–Milgram [2])**.**
Seien H ein Hilbert-Raum mit Skalarprodukt $(\cdot,\cdot)$, $a\colon H \times H \to \mathbb{R}$ eine stetige und koerzive Bilinearform und $F \in H' := \{F\colon H \to \mathbb{R}\colon F \text{ linear}, \text{stetig}\}$. Dann existiert

genau ein $u \in H$, sodass

$$a(u, v) = F(v) \quad \textit{für alle } v \in H.$$

Mit κ aus der Definition der Koerzivität gilt ferner

$$\|u\|_H \leq \kappa^{-1} \|F\|_{H'}.$$

Durch die Anwendung der Cauchy–Schwarz-Ungleichung sowie die Linearität des Integrals kann man zeigen, dass die Bilinearform $a(u, v) := \int_\Omega \nabla u_0 \cdot \nabla v \, \mathrm{d}x$ in (3) auf $H^1(\Omega)$ linear und stetig ist. Zusätzlich kann man mit der Poincaré-Ungleichung die Koerzivität auf $H^1_D(\Omega)$ nachweisen. Da $F(v) = \int_\Omega fv \, \mathrm{d}x + \int_{\Gamma_N} gv \, \mathrm{d}s - \int_\Omega \nabla u_D \cdot \nabla v \, \mathrm{d}x$ offensichtlich linear und stetig ist, sind alle Voraussetzung von Satz 1 erfüllt und wir wissen, dass die schwache Formulierung genau eine Lösung $u_0 \in H^1_D(\Omega)$ besitzt. Da $u \in H^1(\Omega)$ genau dann eine schwache Lösung von (1) ist, wenn $u_0 := u - u_D \in H^1_D(\Omega)$ die Formulierung (3) erfüllt, folgt, dass genau eine Lösung $u := u_0 + u_D \in H^1(\Omega)$ von (1) existiert. Insbesondere sieht man, dass u nicht von der Wahl der Fortsetzung $u_D \in H^1(\Omega)$ abhängt und man so eine beliebige aber möglichst einfache Fortsetzung $u_D \in H^1(\Omega)$ von Γ_D auf Ω wählen kann [4].

2 P1-Galerkin FEM der Poisson-Gleichung

Um das Problem (1) nun mit Hilfe eines Computers lösen zu können, bedarf es ein paar Überlegungen. Wie wir wissen, kann ein Computer nicht mit unendlichdimensionalen Räumen umgehen. Deswegen versuchen wir einen endlichdimensionalen Unterraum von $H^1_D(\Omega)$ zu finden, aus dem die approximierte Lösung zu (3) sein soll. Zu diesem Zweck führen wir reguläre Triangulierungen ein.

Definition 2.
Wir nennen eine endliche Menge kompakter Dreiecke T mit positiver Fläche eine reguläre Triangulierung $\mathcal{T}$, falls die folgenden vier Voraussetzungen erfüllt sind:

- *Für alle $T \in \mathcal{T}$ gilt $T = \operatorname{conv}\{z_1, z_2, z_3\}$ mit $z_1, z_2, z_3 \in \Omega$ sowie $|T| > 0$, wobei* conv *die konvexe Hülle bezeichnet.*
- *Die Vereinigung aller Dreiecke in $\mathcal{T}$ ist der Abschluss von Ω, d.h. $\bigcup \mathcal{T} = \overline{\Omega}$.*
- *Der Schnitt zweier Dreiecke $T, T' \in \mathcal{T}$ mit $T \neq T'$ ist entweder leer, ein gemeinsamer Eckpunkt oder eine gemeinsame Kante von T und T'.*
- *Für alle Kanten $E \subset \partial\Omega$ gilt entweder $E \subset \overline{\Gamma_D}$ oder $E \subset \overline{\Gamma_N}$.*

Damit haben wir eine Triangulierung, die das Gebiet Ω in Dreiecke teilt und dabei auch den Dirichlet-Rand und Neumann-Rand berücksichtigt. Insbesonders ist jeder Knoten am Rand des Gebietes entweder auf Γ_D oder auf Γ_N. Zusätzlich definieren wir $\mathcal{E}_D$ als alle Kanten am Dirichlet-Rand, $\mathcal{E}_N$ als alle Kanten am Neumann-Rand und $\mathcal{E}_\Omega$ als alle Kanten im Inneren des Gebietes. Durch die Konformität der Triangulierung, ist jede Kante in nur einer dieser Mengen enthalten.
Jetzt können wir den endlichdimensionalen Unterraum für die FEM definieren:

$$S^1(\mathcal{T}) := \{V \in C(\Omega) : \text{für alle } T \in \mathcal{T} \quad V|_T \text{ ist affin}\}. \tag{4}$$

Der Raum $S^1(\mathcal{T})$ bezeichnet alle stetigen und auf allen Dreiecken affinen Funktionen. Durch partielle Integration, der Konformität der Triangulierung und der globalen Stetigkeitsforderung, kann man zeigen, dass die elementweise Ableitung von $V \in S^1(\mathcal{T})$ genau die schwache Ableitung ist. Diese ist offensichtlich quadratintegrabel, weil sie sogar beschränkt ist. Somit gilt, dass $S^1(\mathcal{T}) \subset H^1(\Omega)$. Als Nächstes suchen wir mit Hilfe der Knoten $\mathcal{N} = \{z_1, \dots, z_N\}$ der Triangulierung $\mathcal{T}$ eine nützliche Basis von $S^1(\mathcal{T})$. Dafür definieren wir V_ℓ als elementweise affine Funktionen mit

$$V_\ell(z_k) = \delta_{k\ell} \text{ für alle } \ell, k \in \{1, \dots, N\}, \tag{5}$$

wobei δ das Kronecker-Delta bezeichnet. Durch die Konformität der Triangulierung $\mathcal{T}$ folgt, dass alle V_ℓ stetig und dadurch in $S^1(\mathcal{T})$ sind. Da jede Funktion $V \in S^1(\mathcal{T})$ durch ihre Werte auf den Knoten $\mathcal{N}$ eindeutig festgelegt ist, folgt $V = \sum_{\ell=1}^{N} V(z_\ell)V_\ell$, d.h.

$$\mathcal{B} = \{V_1, V_2, \ldots, V_N\} \subset S^1(\mathcal{T}) \tag{6}$$

ist ein Erzeugendensystem von $S^1(\mathcal{T})$. Die Basiseigenschaft folgt direkt aus der linearen Unabhängigkeit der V_ℓ, da die Funktionen immer auf einem Knoten den Wert 1 und auf allen anderen Knoten den Wert 0 haben. Daraus folgt auch, dass $\dim(S^1(\mathcal{T})) = N < \infty$. In Abbildung 2.1 kann man einfache Beispiele der Hut-Funktionen sehen.

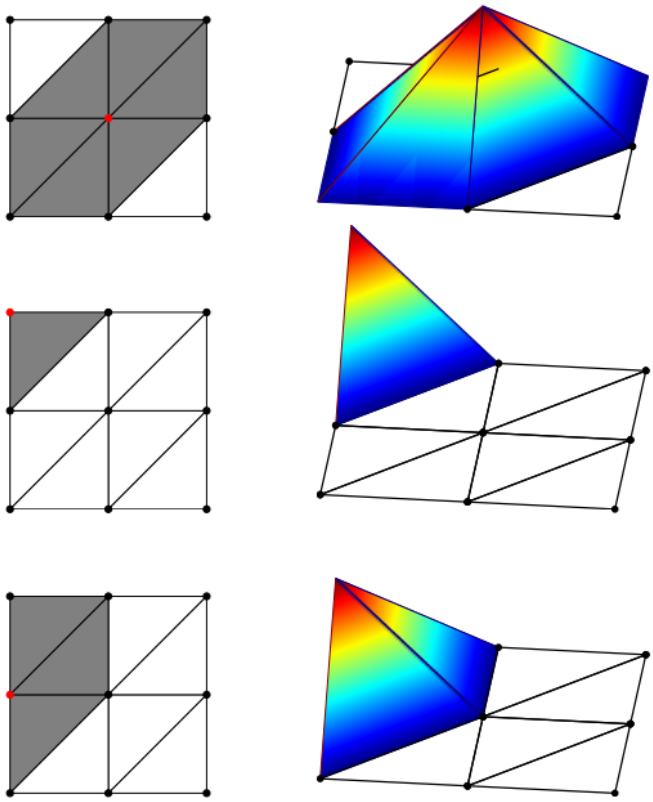


Abbildung 2.1: Beispiele von Hut-Funktionen aus [3]: Links sieht man die Triangulierung von oben und den Träger der Funktion V_ℓ in grau mit dem zugehörigen Knoten z_ℓ in rot. Rechts sieht man den Plot der jeweiligen Funktion. Alle anderen Dreiecke $T \in \mathcal{T}$ mit $V_\ell|_T = 0$ wurden weiß ausgemalt.

Für die Galerkin-Methode benötigen wir jedoch, wie im vorherigen Kapitel, noch einen Raum, der die Dirichlet-Randbedingungen berücksichtigt

$$S^1_D(\mathcal{T}) := S^1(\mathcal{T}) \cap H^1_D(\Omega) = \{V \in S^1(\mathcal{T}) : V(z_\ell) = 0 \quad \text{für alle } z_\ell \in \mathcal{N} \cap \overline{\Gamma_D}, \tag{7}$$

wobei die zweite Gleichheit wieder wegen der Konformität der Triangulierung $\mathcal{T}$ gilt. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass $\mathcal{N} \cap \overline{\Gamma_D} = \{z_{n+1}, \ldots, z_N\}$. Da

die Einschränkung des Raumes die Basiseigenschaft nicht beeinflusst, müssen wir nur die überzähligen Basisfunktinen entfernen und bekommen mit $\mathcal{B}_D = \{V_1, V_2, \dots, V_n\}$ eine Basis für den Raum $S^1_D(\mathcal{T})$. Außerdem nehmen wir an, dass die Dirichlet-Funktion $u_D \in H^1(\Omega)$ stetig auf Γ_D ist und wir daher $u_D|_{\Gamma_D}$ nodale Interpolation

$$U_D := \sum_{\ell=n+1}^{N} u_D(z_\ell) V_\ell \quad \in S^1(\mathcal{T}) \tag{8}$$

approximieren können. Man bemerke, dass hier nur die Knoten $\{z_{n+1}, \dots, z_N\}$ des Dirichlet-Randes verwendet werden.
Jetzt können wir die diskrete Variante der schwachen Formulierung (3) aufstellen:
Finde $U_0 \in S^1_D(\mathcal{T})$, sodass

$$\int_\Omega \nabla U_0 \cdot \nabla V \, \mathrm{d}x = \int_\Omega fV \, \mathrm{d}x + \int_{\Gamma_N} gV \, \mathrm{d}s - \int_\Omega \nabla U_D \cdot \nabla V \, \mathrm{d}x \quad \text{für alle } V \in S^1_D(\mathcal{T}). \tag{9}$$

Hierfür betrachten wir folgendes Lemma:

Lemma 1. *Seien $S \in \mathbb{R}^{N\times N}$ und $L \in \mathbb{R}^n$ definiert durch*

$$S_{jk} = \int_\Omega \nabla V_k \cdot \nabla V_j \, \mathrm{d}x \quad \textit{für alle } j,k \in \{1,\dots,N\} \tag{10}$$

$$L_j := \int_\Omega fV_j \, \mathrm{d}x + \int_{\Gamma_N} gV_j \, \mathrm{d}s - \sum_{k=n+1}^{N} S_{jk} u_D(z_k) \quad \textit{für alle } j \in \{1,\dots,n\}. \tag{11}$$

Sei $S_n \in \mathbb{R}^{n\times n}$ eine kleinere Matrix definiert durch $(S_n)_{jk} = S_{jk}$. Dann ist S_n positiv definit und damit invertierbar, d.h. $S_n w = L$ hat eine eindeutige Lösung $w \in \mathbb{R}^n$. Sei $x \in \mathbb{R}^N$ definiert durch

$$x_j := \begin{cases} w_j & \textit{falls } j \in \{1,\dots,n\}, \\ u_D(z_j) & \textit{falls } j \in \{n+1,\dots,N\} \end{cases} \tag{12}$$

und $U_0 := \sum_{j=1}^n x_j V_j \in S^1_D(\mathcal{T})$, $U := \sum_{j=1}^N x_j V_j \in S^1(\mathcal{T})$ und $U_D := \sum_{j=n+1}^N x_j V_j \in S^1(\mathcal{T})$. Dann ist U die eindeutige Lösung von

$$\int_\Omega \nabla U \cdot \nabla V \, \mathrm{d}x = \int_\Omega fV \, \mathrm{d}x + \int_{\Gamma_N} gV \, \mathrm{d}s \quad \textit{für alle } V \in S^1_D(\mathcal{T}) \tag{13}$$

unter der Nebenbedingung $U|_{\Gamma_D} = U_D|_{\Gamma_D} \approx u_D|_{\Gamma_D}$. Außerdem gilt

$$\int_\Omega \nabla U_0 \cdot \nabla V \, \mathrm{d}x = \int_\Omega fV \, \mathrm{d}x + \int_{\Gamma_N} gV \, \mathrm{d}s - \int_\Omega \nabla U_D \cdot \nabla V \, \mathrm{d}x \quad \textit{für alle } V \in S^1_D(\mathcal{T}). \tag{14}$$

Beweis. Sei $y \in \mathbb{R}^n \setminus \{0\}$ beliebig und sei $V := \sum_{j=1}^n y_j V_j \in S^1_D(\mathcal{T})$ mit der Basis $\mathcal{B}_D$ dargestellt. Da $\mathcal{B}_D$ eine Basis von $S^1_D(\mathcal{T})$ ist, folgt zunächst $V \neq 0$ und $V|_{\Gamma_D} = 0$. Ferner

gilt $\nabla V \neq 0$, denn aus $\nabla V = 0$ würde folgen, dass V konstant ist und damit $V = 0$ wegen $V|_{\Gamma_D} = 0$. Deshalb gilt

$$\begin{aligned} y \cdot S_n y &= \sum_{j=1}^{n} y_j \sum_{k=1}^{n} (S_n)_{jk} y_k \\ &= \int_\Omega \nabla\Big(\sum_{k=1}^{n} y_k V_k\Big) \cdot \nabla\Big(\sum_{j=1}^{n} y_j V_j\Big) \mathrm{d}x \\ &= \langle \nabla V, \nabla V \rangle_{L^2(\Omega)} = \|\nabla V\|^2_{L^2(\Omega)} > 0, \text{ da } \nabla V \neq 0. \end{aligned}$$

Daraus folgt, dass S_n positiv definit und somit invertierbar ist. Weiters wollen wir zeigen, dass $S_n w = L$ äquivalent zu dem Problem (14) ist und wir so unsere Lösung bezüglich der Basis B_D darstellen können. Es gilt

$$\begin{aligned} S_n w = \left(\sum_{k=1}^{n} S_{jk} w_k \right)_{j=1}^{n} &= \left(\sum_{k=1}^{n} x_k \underbrace{\int_\Omega \nabla V_k \cdot \nabla V_j \, \mathrm{d}x}_{=S_{jk}} \right)_{j=1}^{n} \\ &= \left(\int_\Omega \nabla\Big(\sum_{k=1}^{n} x_k V_k\Big) \cdot \nabla V_j \, \mathrm{d}x \right)_{j=1}^{n} \\ &= \left(\int_\Omega \nabla U_0 \cdot \nabla V_j \, \mathrm{d}x \right)_{j=1}^{n}. \end{aligned}$$

Insbesondere folgt $y \cdot S_n w = \int_\Omega \nabla U_0 \cdot \nabla V \, \mathrm{d}x$. Analog sehen wir, dass $y \cdot L$ gleich der rechten Seite von (14) ist. Somit ist die Äquivalenz der beiden Probleme gezeigt. Da $S_n x_n = L$ eindeutig lösbar ist, bekommen wir durch $U = U_0 + U_D$ unsere eindeutige Lösung unseres Problems (13). □

Mithilfe dieses Lemmas können wir nun $U \in S^1(\mathcal{T})$ als Approximation von $u \in H^1(\Omega)$ berechnen. Für die Implementierung werden wir zuerst $S \in \mathbb{R}^{N \times N}$ aufstellen und dann $S_n x_n = L$ auf dem $n \times n$ Teilproblem der Knoten, die nicht auf dem Dirichlet-Rand liegen, lösen [4].

3 Implementierung der P1-Galerkin FEM für die 2D Poisson-Gleichung

In diesem Abschnitt wollen wir die Implementierung der P1-Galerkin FEM der Poisson-Gleichung genauer betrachten. In weiteren Kapiteln werden wir dieses Problem verallgemeinern und erweiterte Codes und ihre Laufzeiten anschauen. Der folgende Code dient nur dem ersten Verständnis unserer Ansatzes:

```
def solveLaplace(coordinates,elements,dirichlet,neumann,f,g,uD):
    nC = np.shape(coordinates)[0]
    nE = np.shape(elements)[0]
    x = np.zeros(nC)

    # Matrix aufbauen
    S = sparse.csc_matrix((nC,nC))
    for i in range(nE):
        nodesidx = elements[i,:]
        nodes = coordinates[nodesidx-1,:]
        P = np.vstack(([1,1,1],np.transpose(nodes)))
        areaT = np.linalg.det(P)/2
        grad = np.linalg.solve(P,np.array([[0,0],[1,0],[0,1]]))
        S[np.ix_(nodesidx-1, nodesidx-1)] += \
            areaT * grad @ np.transpose(grad)

    #Dirichlet-Knoten abarbeiten
    dirichlet = np.unique(dirichlet)
    for k in dirichlet:
        x[k-1] = uD(coordinates[k-1][0],coordinates[k-1][1])

    #rechte Seite berechnen
    L = -S @ x
    for i in range(nE):
        nodesidx = elements[i,:]
        nodes = coordinates[nodesidx-1,:]
        P = np.vstack(([1,1,1],np.transpose(coordinates[nodesidx-1,:])))
        areaT = np.linalg.det(P)/2
        sT = np.array(nodes).sum(axis=0)/3
        L[nodesidx-1] += areaT * f(sT[0],sT[1])/3

    for i in range(np.shape(neumann)[0]):
        nodesidx = neumann[i,:]
        nodes = coordinates[nodesidx-1,:]
        mE = np.array(nodes).sum(axis=0)/2
        lengthE = np.linalg.norm(np.array([1,-1]) @ np.array(nodes))
        L[nodesidx-1] += lengthE * g(mE[0],mE[1])/2

    #P1-FEM Berechnung
```

```
40      freenodes = np.setdiff1d(np.arange(nC),dirichlet-1)
41      x[freenodes] = sparse.linalg.spsolve(S[np.ix_(freenodes, freenodes)], \
42          L[freenodes])
43
44      # Energie
45      energy = np.transpose(x) @ S @ x
46
47      return x, energy
```

Python-Code 3.1: Eine anschauliche Implementierung der P1-Galerkin FEM der Poisson-Gleichung

3.1 Speicherung der Daten

In diesem Abschnitt präsentieren wir, wie die Speicherung der Daten abgewickelt wird.

- Die Knoten $\mathcal{N}$ sind gespeichert als $N \times 2$ array `coordinates`, wobei in der i-ten Zeile die Koordinaten des i-ten Knoten sind, d.h. $z_i = (x_i, y_i) \in \mathbb{R}^2$ ist gespeichert in

$$\texttt{coordinates}[i,:] = [x_i \quad y_i].$$

- Die Triangulierung $\mathcal{T}$ ist dargestellt als $M \times 3$ array `elements`, wobei in der i-ten Zeile das i-te Dreiecke angegeben ist, d.h. $T_i = \text{conv}\{z_j, z_k, z_\ell\} \in \mathcal{T}$ mit $z_j, z_k, z_\ell \in \mathcal{N}$ ist gespeichert in

$$\texttt{elements}[i,:] = [j \quad k \quad \ell],$$

  wobei hier die Knoten gegen den Uhrzeigersinn geordnet sind (also in mathematisch positiver Umlaufrichtung).

- Der Dirichlet-Rand Γ_D und Neumann-Rand Γ_N sind analog gespeichert. Sie werden dargestellt als $K \times 2$ bzw. $L \times 2$ arrays `dirichlet` bzw. `neumann`, wobei hier auch die i-te Zeile die i-te Kante angibt, d.h. $E_i = \text{conv}\{z_j, z_k\}$ ist gespeichert in:

$$\texttt{dirichlet}[i,:] = [j \quad k] \quad \text{bzw.} \quad \texttt{neumann}[i,:] = [j \quad k],$$

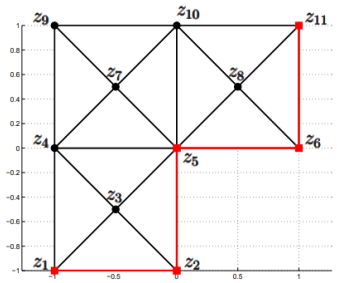
  wobei auch hier die mathematisch positive Umlaufrichtung (mit Bezug auf den Rand $\partial\Omega$) verwendet wird.

In Abbildung 3.1 ist die Speicherung der Daten für ein einfaches Beispiel noch einmal veranschaulicht.

3.2 Erklärung des Codes

In diesem Abschnitt wird erklärt, wie der Python-Code 3.1 funktioniert:

- **Zeile 1:** Wir definieren die Funktion `solveLaplace`, welche sieben Inputs hat. Die ersten vier Inputs gehören zur Triangulierung der Fläche. während die restlichen drei die Kraft f, die Neumann-Daten g und die Dirichlet-Daten u_D angeben. Diese Funktionen



coordinates

1	−1.0	−1.0
2	0.0	−1.0
3	−0.5	−0.5
4	−1.0	0.0
5	0.0	0.0
6	1.0	0.0
7	−0.5	0.5
8	0.5	0.5
9	−1.0	1.0
10	0.0	1.0
11	1.0	1.0

elements

1	1	2	3
2	2	5	3
3	5	4	3
4	4	1	3
5	4	5	7
6	5	10	7
7	10	9	7
8	9	4	7
9	5	6	8
10	6	11	8
11	11	10	8
12	10	5	8

dirichlet

1	1	2
2	2	5
3	5	6
4	6	11

neumann

1	11	10
2	10	9
3	9	4
4	4	1

Abbildung 3.1: Ein einfaches Beispiel aus [4] zur Veranschaulichung der Speicherung der Daten einer Triangulierung mit Hilfe der Arrays `coordinates`, `elements`, `dirichlet` und `neumann`

werden in Python als vorher definierte Funktionen übergeben. Jede dieser Funktionen bekommt als Input n Knoten und gibt einen Vektor von Funktionswerten zurück, der diese Funktion auf jedem Knoten auswertet, d.h. im Code erhält man mit den zwei Vektoren $\xi \in \mathbb{R}^n$ und $\psi \in \mathbb{R}^n$, die für die x- bzw. y-Koordinaten der Knoten stehen, den Vektor $z \in \mathbb{R}^n$ mit $z_j = f(\xi_j, \psi_j)$. Außderdem ist zu bemerken, dass wir dafür implizit voraussetzen, dass $f, g, u_D \in C(\overline{\Omega})$ gilt, da wir sonst keine Punktauswertung machen könnten.
Als Rückgabe der Funktion `solveLaplace` erhalten wir den Koeffizientenvektor $x \in \mathbb{R}^N$ mit $x_j = U(z_j)$ der diskreten Lösung $U \in S^1(\mathcal{T})$ gemäß Lemma 1. Außerdem erhalten wir die Energie $\|\nabla U\|^2_{L^2(\Omega)} = x \cdot Sx$

- **Zeile 8–15:** Wir berechnen die Matrix $S \in \mathbb{R}^{N\times N}_{\text{sym}}$ elementweise. Hier verwenden wir

$$S_{jk} = \int_\Omega \nabla V_k \cdot \nabla V_j \, \mathrm{d}x = \sum_{T\in\mathcal{T}} \int_T \nabla V_j \cdot \nabla V_k \, \mathrm{d}x = \sum_{T\in\mathcal{T}} |T| \, \nabla V_j|_T \cdot \nabla V_k|_T.$$

Wir beobachten, dass wir für jedes Dreieck $T \in \mathcal{T}$ nur eine 3×3 Matrix berechnen müssen, da das Produkt $\nabla V_j \cdot \nabla V_k$ immer 0 ist, wenn einer der Knoten z_j oder z_k nicht in T ist. Mit dieser Beobachtung können wir S mit Hilfe einer for-Schleife über alle Dreiecke aufbauen. Für ein fixes $T \in \mathcal{T}$ können wir mittels der Matrix $P \in \mathbb{R}^{3\times 3}$ in Zeile 11 die Fläche $|T| = \det(P)/2$ berechnen (Zeile 12). Dies beruht auf:

Lemma 2. *Für ein Dreieck $T = \mathrm{conv}\{z_1, z_2, z_3\}$ mit Knoten $z_1, z_2, z_3 \in \mathbb{R}^2$ (verstanden als Spaltenvektoren) gilt*

$$|T| = \frac{1}{2}|\det(z_2 - z_1 \quad z_3 - z_1)| = \frac{1}{2}\left|\det\begin{pmatrix} 1 & 1 & 1 \\ z_1 & z_2 & z_3 \end{pmatrix}\right|,$$

wobei die auftretenden Matrizen 2×2 und 3×3 sind.

Beweis. Ohne Beschränkung der Allgemeinheit können wir voraussetzen, dass das Dreieck T nicht degeneriert ist. Die erste Gleichheit folgt aus dem Transformationssatz, wenn man die Abbildung $\Phi_T : \Delta \to T$, $\Phi_T(x, y) = z_1 + x(z_2 - z_1) + y(z_3 - z_1)$ von

dem Referenzdreieck $\Delta = \{(x, y) \in \mathbb{R}^2 : 0 \leq x \leq 1, 0 \leq y \leq 1 - x\}$ auf T betrachtet und dann folgende Gleichheit nutzt:

$$|T| = \int_{\Phi_T(\Delta)} 1 \, \mathrm{d}y = \int_{\Delta} 1 |\det(D\Phi_T)| \, \mathrm{d}x.$$

Die linke Seite ist genau $|T|$ und die rechte Seite ergibt genau unser gewünschtes Ergebnis, da $|\Delta| = 1/2$. Die zweite Gleichheit gilt, weil

$$\det \begin{pmatrix} 1 & 1 & 1 \\ z_1 & z_2 & z_3 \end{pmatrix} = \det \begin{pmatrix} 1 & 0 & 0 \\ z_1 & z_2 - z_1 & z_3 - z_1 \end{pmatrix} = \det(z_2 - z_1 \quad z_3 - z_1) = \det D\Phi_T.$$

Damit sind die Gleichheiten gezeigt. □

In Zeile 13 berechnen wir die Gradienten der drei Hutfunktionen des aktuellen Dreiecks als `grad` $\in \mathbb{R}^{3x2}$ Matrix, wobei die einzelnen Gradienten die einzelnen Zeilen sind. Dies beruht auf:

Lemma 3. *Für ein Dreieck* $T = \mathrm{conv}\{z_1, z_2, z_3\}$ *mit Knoten* $z_1 = (x_1, y_1)^T$, $z_2 = (x_2, y_2)^T$, $z_3 = (x_3, y_3)^T \in \mathbb{R}^2$ *und* $\mathrm{grad} = (\nabla V_1, \nabla V_2, \nabla V_3)^T \in \mathbb{R}^{3x2}$, *wobei* V_1, V_2 *und* V_3 *die zu diesen Punkten gehörenden Hutfunktionen sind, gilt*

$$\begin{pmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{pmatrix} \mathrm{grad} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Beweis. Wir wissen, dass Hutfunktion affin sind, deshalb können wir sie mit gewissen $a, b, c \in \mathbb{R}$ schreiben als

$$V(x, y) = a + bx + cy \qquad \text{mit } \nabla V = \begin{pmatrix} b \\ c \end{pmatrix}.$$

In vektorieller Form heißt das

$$V(x, y) = (1, x, y) \cdot (a, b, c).$$

In Matrixschreibweise ergibt das für die erste Hutfunktion

$$Fa_1 := \begin{pmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix},$$

welches wegen $\det F = \det F^T \neq 0$ und Lemma 2 ein reguläres Gleichungssystem ist. Da wir uns jedoch nur auf den Gradienten, also $(b, c)^T$ konzentrieren, können wir schreiben

$$\begin{pmatrix} b \\ c \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} a_1 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} F^{-1} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

Wenn wir jetzt beide Seiten transponieren und nicht nur die erste Hutfunktion, sondern alle Hutfunktionen gleichzeitig betrachten, ergibt das

$$\begin{pmatrix} b_1 & c_1 \\ b_2 & c_2 \\ b_3 & c_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \left(F^T\right)^{-1} \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix},$$

wobei $\nabla V_i = (b_i, c_i)^T$ für $i = 1, 2, 3$ ist. Das lässt sich natürlich umschreiben in

$$F^T \begin{pmatrix} b_1 & c_1 \\ b_2 & c_2 \\ b_3 & c_3 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix},$$

womit wir die Aussage gezeigt haben. □

- **Zeile 18–20:** Hier werden die Einträge des Koeffizientenvektors $x \in \mathbb{R}^N$, die dem Dirichlet-Rand zugeordnet sind, initialisiert.

- **Zeile 23–37:** Der Vektor $L \in \mathbb{R}^N$ aus Lemma 1 wird berechnet. Zuerst verwenden wir dazu die Dirichlet-Daten in Zeile 23. Dann addieren wir elementenweise die Kraft f dazu. Hier werden die auftretenden Integrale mit Hilfe einer Quadratur mit dem Schwerpunkt $s_T \in T$ approximiert, wobei (aufgrund der elementweisen Affinität der Hutfunktionen) $V_j(s_T) = 1/3$ gilt. Es folgt:

$$\int_\Omega f V_j \mathrm{d}x = \sum_{T \in \mathcal{T}} \int_T f V_j \mathrm{d}x \approx \sum_{T \in \mathcal{T}} |T| f(s_T) V_j(s_T) = \sum_{T \in \mathrm{supp}(V_j)} \frac{|T|}{3} f(s_T).$$

  Für jedes Dreieck T werden somit drei Einträge in L verändert. Schlussendlich (Zeile 32–37) werden dann noch die Neumann-Daten hinzugefügt. Um das Integral zu approximieren, wird auch hier Quadratur mit dem Mittelpunkt $m_E \in E$ angewandt, wobei (aufgrund der kantenweise Affinität der Hutfunktionen) $V_j(m_E) = 1/2$ gilt. Außerdem beschreibt h_E die Länge der Kante. Es folgt:

$$\int_{\Gamma_N} g V_j \mathrm{d}x = \sum_{E \in \mathcal{E}_N} \int_E g V_j \mathrm{d}x \approx \sum_{E \in \mathcal{E}_N} h_E g(m_E) V_j(m_E) = \sum_{E \in \mathrm{supp}(V_j)} \frac{h_E}{2} g(m_E).$$

  Hier werden für jede Kante zwei Einträge in L geändert.

- **Zeile 40–42:** Zuerst erstellen wir die Menge aller Knoten, für die gilt $z_j \notin \Gamma_D$ (Zeile 40). Dann lösen wir das lineare Gleichungssystem aus Lemma 1 für den Vektor x für die Knoten $z_j \notin \Gamma_D$. Beachte, dass hier die Koeffizineten $x_k = u_D(z_k)$ für die Knoten $z_k \in \Gamma_D$ nicht verändert werden, sodass $x \in \mathbb{R}^N$ nun der Koeffizientenvektor der P1-FEM Lösung $U \in S^1(\mathcal{T})$ ist.

4 Allgemeine P1-Galerkin FEM in 2D

Die letzten Kapitel haben sich damit beschäftigt, die Lösung $u \in H^1(\Omega)$ des Problems

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= u_D \quad \text{auf } \Gamma_D, \\ \nabla u \cdot n &= g \quad \text{auf } \Gamma_N. \end{aligned} \tag{15}$$

zu approximieren. Jedoch wollen wir die Problemstellung nicht bei der Poisson-Gleichung belassen, sondern die allgemeine Form einer linearen elliptischen partiellen Differentialgleichung zweiter Ordnung anschauen: Gegeben seien wieder $f \in L^2(\Omega), u_D \in H^{1/2}(\Gamma_D)$ und $g \in L^2(\Gamma_N)$. Wir nehmen wieder an, dass $\Omega \subset \mathbb{R}^2$ ein beschränktes Lipschitz-Gebiet ist, welches einen Rand $\Gamma := \partial\Omega$ besitzt (mit dem äußeren Normalvektor n), den wir in einen abgeschlossenen Dirichlet-Rand Γ_D und in einen Neumann-Rand $\Gamma_N := \Gamma \setminus \Gamma_D$ unterteilen. Hierbei wird vorausgesetzt, dass Γ_D positive Länge hat. Zusätzlich seien Koeffizienten $\mathcal{A}, b, c \in L^\infty(\Omega)$ mit $\mathcal{A}(x) \in \mathbb{R}^{2\times 2}_{\text{sym}},\ b(x) \in \mathbb{R}^2,\ c(x) \in \mathbb{R}$ gegeben, sodass

$$\begin{aligned} -\operatorname{div}(\mathcal{A}\nabla u) + b \cdot \nabla u + cu &= f \quad \text{in } \Omega, \\ u &= u_D \quad \text{auf } \Gamma_D, \\ (\mathcal{A}\nabla u) \cdot n &= g \quad \text{auf } \Gamma_N. \end{aligned} \tag{16}$$

Wir können wie im vorherigen Kapitel verfahren und mit einer beliebigen Erweiterung

$$u_0 = u - u_D \in H^1_D(\Omega) := \{v \in H^1(\Omega) \colon v = 0 \text{ auf } \Gamma_D\} \tag{17}$$

die schwache Formulierung aufstellen: Mit der Bilinearform

$$a(u, v) := \int_\Omega (\mathcal{A}\nabla u \cdot \nabla v + b \cdot \nabla u v + cuv)\ \mathrm{d}x \tag{18}$$

suchen wir $u_0 \in H^1_D(\Omega)$ sodass gilt:

$$a(u_0, v) = \int_\Omega fv\ \mathrm{d}x + \int_{\Gamma_N} gv\ \mathrm{d}s - a(u_D, v) \quad \text{für alle } v \in H^1_D(\Omega). \tag{19}$$

Wir müssen zusätzlich fordern, dass (19) alle Voraussetzung des Lemmas von Lax–Milgram erfüllt, sodass die schwache Formulierung genau eine Lösung $u_0 \in H^1_D(\Omega)$ besitzt. Somit gilt auch, dass genau eine Lösung $u := u_0 + u_D \in H^1(\Omega)$ von (16) existiert.
Die Umsetzung des allgemeinen Problems am Computer unterscheidet sich von der der Poisson-Gleichung kaum. Die Triangulierung und das Aufstellen der diskreten Variante der schwachen Formulierung funktionieren analog. Wir suchen $U_0 \in S^1_D(\mathcal{T})$ mit

$$a(U_0, V) = \int_\Omega fV\ \mathrm{d}x + \int_{\Gamma_N} gV\ \mathrm{d}s - a(U_D, V) \quad \text{für alle } V \in S^1_D(\mathcal{T}). \tag{20}$$

Wenn man dies mit (9) vergleicht, sieht man, dass sich nur die Bilinearform geändert hat. Somit betrachten wir wieder folgendes Lemma:

Lemma 4. *Seien* $A,\ B,\ C \in \mathbb{R}^{N\times N}$ *für alle* $j,k \in \{1,\dots,N\}$ *und* $L \in \mathbb{R}^n$ *für alle* $j,k \in \{1,\dots,n\}$ *wie folgt definiert:*

$$A_{jk} = \int_\Omega \mathcal{A}\nabla V_k \cdot \nabla V_j \ \mathrm{d}x, \tag{21}$$

$$B_{jk} = \int_\Omega b \cdot \nabla V_k \ V_j \ \mathrm{d}x, \tag{22}$$

$$C_{jk} = \int_\Omega cV_k \ V_j \ \mathrm{d}x, \tag{23}$$

$$L_j = \int_\Omega fV_j \ \mathrm{d}x + \int_{\Gamma_N} gV_j \ \mathrm{d}x - \sum_{k=n+1}^{N} S_{jk}u_D(z_k). \tag{24}$$

Seien $S := A + B + C \in \mathbb{R}^{N\times N}$ *und* $S_n \in \mathbb{R}^{n\times n}$ *eine kleinere Matrix definiert durch* $(S_n)_{jk} = S_{jk}$. *Sei die Bilinearform* $a(\cdot,\cdot)$ *definiert durch*

$$a(u,v) := \int_\Omega (\mathcal{A}\nabla u \cdot \nabla v + b \cdot \nabla uv + cuv) \ \mathrm{d}x. \tag{25}$$

Ist $a(\cdot,\cdot)$ *elliptisch auf* $H^1_D(\mathcal{T})$, *dann ist* S_n *positiv definit und damit invertierbar, d.h.* $S_n w = L$ *hat eine eindeutige Lösung* $w \in \mathbb{R}^n$. *Sei* $x \in \mathbb{R}^N$ *definiert durch*

$$x_j := \begin{cases} w_j & \text{falls } j \in \{1,\dots,n\}, \\ u_D(z_j) & \text{falls } j \in \{n+1,\dots,N\} \end{cases} \tag{26}$$

und $U_0 := \sum_{j=1}^n x_j V_j \in S^1_D(\mathcal{T})$, $U := \sum_{j=1}^N x_j V_j \in S^1(\mathcal{T})$ *und* $U_D := \sum_{j=n+1}^N x_j V_j \in S^1(\mathcal{T})$. *Dann ist* U *die eindeutige Lösung von*

$$a(U,V) = \int_\Omega fV \ \mathrm{d}x + \int_{\Gamma_N} gV \ \mathrm{d}s \quad \text{für alle } V \in S^1_D(\mathcal{T}) \tag{27}$$

unter der Nebenbedingung $U|_{\Gamma_D} = U_D|_{\Gamma_D} \approx u_D|_{\Gamma_D}$. *Außerdem gilt*

$$a(U_0,V) = \int_\Omega fV \ \mathrm{d}x + \int_{\Gamma_N} gV \ \mathrm{d}s - a(U_D,V) \quad \text{für alle } V \in S^1_D(\mathcal{T}). \tag{28}$$

Der Beweis dieses Lemmas ist analog zu dem Beweis von Lemma 1. Mithilfe dieses Lemmas können wir nun U als Approximation von $u \in H^1(\Omega)$ berechnen. Für die Implementierung werden wir zuerst $A,\ B,\ C$ und L aufstellen und dann $S_n x_n = L$ auf dem $n \times n$ Teilproblem der Knoten, die nicht auf dem Dirichlet-Rand liegen, lösen.

5 Implementierung der allgemeinen P1-Galerkin FEM in 2D

In diesem Abschnitt betrachten wir die Implementierung der P1-Galerkin FEM für das allgemeine Problem (16). Noch ist diese Implementierung nicht effizient, sondern dient nur zur Illustration. Aber im nächsten Kapitel wird die Optimierung der Laufzeit diskutiert.

```
def solve0(coordinates,elements,dirichlet,neumann,f,g,uD,a,b,c):
    nC = np.shape(coordinates)[0]
    nE = np.shape(elements)[0]
    x = np.zeros(nC)

    # Matrix aufbauen
    A = sparse.csc_matrix((nC,nC))
    B = sparse.csc_matrix((nC,nC))
    C = sparse.csc_matrix((nC,nC))
    for i in range(nE):
        nodesidx = elements[i,:]
        nodes = coordinates[nodesidx-1,:]
        P = np.vstack(([1,1,1],np.transpose(nodes)))
        areaT = np.linalg.det(P)/2
        grad = np.linalg.solve(P,np.array([[0,0],[1,0],[0,1]]))
        sT = np.array(nodes).sum(axis=0)/3
        A[np.ix_(nodesidx-1, nodesidx-1)] += (areaT * \
            (grad @ a) @ np.transpose(grad))
        B[np.ix_(nodesidx-1, nodesidx-1)] += np.transpose( \
            (areaT * (grad @ b(sT[0],sT[1])) /3) * np.ones(3))
        C[np.ix_(nodesidx-1, nodesidx-1)] += areaT * \
            c(sT[0],sT[1])/9 * np.ones((3,3))
    S = A + B + C

    #Dirichlet-Knoten abarbeiten
    dirichlet = np.unique(dirichlet)
    for k in dirichlet:
        x[k-1] = uD(coordinates[k-1][0],coordinates[k-1][1])

    #rechte Seite berechnen
    L = -S @ x
    for i in range(nE):
        nodesidx = elements[i,:]
        nodes = coordinates[nodesidx-1,:]
        P = np.vstack(([1,1,1],np.transpose(nodes)))
        areaT = np.linalg.det(P)/2
        sT = np.array(nodes).sum(axis=0)/3
        L[nodesidx-1] += areaT * f(sT[0],sT[1])/3
    for i in range(np.shape(neumann)[0]):
        nodesidx = neumann[i,:]
```

```
41          nodes = coordinates[nodesidx-1,:]
42          mE = np.array(nodes).sum(axis=0)/2
43          lengthE = np.linalg.norm(np.array([1,-1]) @ np.array(nodes))
44          L[nodesidx-1] += lengthE * g(mE[0],mE[1])/2
45
46      #P1-FEM Berechnung
47      freenodes = np.setdiff1d(np.arange(nC),dirichlet-1)
48      x[freenodes] = sparse.linalg.spsolve(S[np.ix_(freenodes, freenodes)], \
49          L[freenodes])
50
51      return x
```

Python-Code 5.1: Eine anschauliche, aber ineffiziente Implementierung der allgemeinen P1-Galerkin FEM

5.1 Erklärung des Codes

Wenn wir diesen Code mit dem Code 5.1 der Poisson-Gleichung vergleichen, können wir sehen, dass sich nur der Aufbau der Matrix verändert hat. Durch die geänderte Bilinearform müssen wir nun drei Matrizen aufstellen und diese zusammenfügen.

- **Zeile 1:** Im Vergleich zu `solveLaplace` haben wir hier nun zusätzlich die Inputs $\mathcal{A}$, b und c. Dazu setzen wir voraus, dass $\mathcal{A}$ konstant ist und aufgrund der späteren Punktauswertung am Schwerpunkt der Dreiecke auch implizit, dass $b,\ c \in C(\overline{\Omega})$. Als Ausgabe der Funktion `solve0` erhalten wir diesmal den Koeffizientenvektor $x_j = U(z_j)$ von der diskreten Lösung $U \in S^1(\mathcal{T})$ der allgemeinen linearen elliptischen partiellen Differentialgleichung (16) zweiter Ordnung.

- **Zeile 7–23:** Um die drei Matrizen aus Lemma 4 aufzustellen, gehen wir analog zum Poisson-Problem vor. Mit der Beobachtung

$$A_{jk} = \int_\Omega \mathcal{A}\nabla V_k \cdot \nabla V_j \,\mathrm{d}x = \sum_{T\in\mathcal{T}} \int_T \mathcal{A}\nabla V_k \cdot \nabla V_j \,\mathrm{d}x \quad \text{für alle } j,k \in \{1,\dots,N\} \tag{29}$$

  können wir die Einträge der Matrix $A \in \mathbb{R}^{N\times N}_{\mathrm{sym}}$ berechnen:

$$A_{jk} = \sum_{T\in\mathcal{T}} |T| \big(\mathcal{A}\nabla V_k|_T\big) \cdot \nabla V_j|_T. \tag{30}$$

  Wir nutzen wieder aus, dass für jedes Element $T \in \mathcal{T}$ die beiden Summanden $\nabla V_j|_T$ und $\nabla V_k|_T$ nur dann gleichzeitig nicht-null sind, wenn z_j und z_k beide Knoten von T sind. Hier ist zu beachten, dass in der Matrix `grad` $\in \mathbb{R}^{3\times 2}$ die Gradienten der Huntfunktionen als Zeilen gespeichert sind. Dies gilt natürlich auch für die Matrizen B und C analog. Hier gilt wieder mit $V_k(s_T) = V_j(s_T) = 1/3$ (im Code entspricht das

dem Vektor $\frac{1}{3}(1,1,1)$) und mittels der Schwerpunkt-Quadratur:

$$\begin{aligned} B_{jk} = \sum_{T\in\mathcal{T}} \int_T b\cdot\nabla V_k|_T \; V_j|_T \,\mathrm{d}x &\approx \sum_{T\in\mathcal{T}} |T|\big(b(s_T)\cdot\nabla V_k|_T\big)V_j(s_T) \\ &= \sum_{T\in\mathrm{supp}(V_j)} \frac{|T|}{3} b(s_T)\cdot\nabla V_k|_T, \\ C_{jk} = \sum_{T\in\mathcal{T}} \int_T cV_k|_T \; V_j|_T \,\mathrm{d}x &\approx \sum_{T\in\mathcal{T}} |T|c(s_T)V_k(s_T)V_j(s_T) \\ &= \sum_{T\in\mathrm{supp}(V_j)\,\cap\,\mathrm{supp}(V_k)} \frac{|T|}{9} c(s_T). \end{aligned} \tag{31}$$

Somit können wir diese Matrizen aufstellen und in Zeile 23 addieren. Damit haben wir S berechnet und können nun wie bei der Poisson-Gleichung fortfahren.

6 Effizientere Methode der allgemeinen P1-Galerkin FEM

6.1 Speicherung von csc-Matrizen

Auf den ersten Blick sehen die Python-Codes 3.1 und 5.1 sehr effizient aus. Doch wenn man den Code empirisch untersucht, sieht man, dass das Aktualisieren der csc-Matrizen(*Compressed sparse column*) — oder auch ccs-Matrizen genannt — einen quadratischen Aufwand verursacht. In Abbildung 6.1 kann man gut erkennen, dass der Aufwand des `solve0` Algorithmus

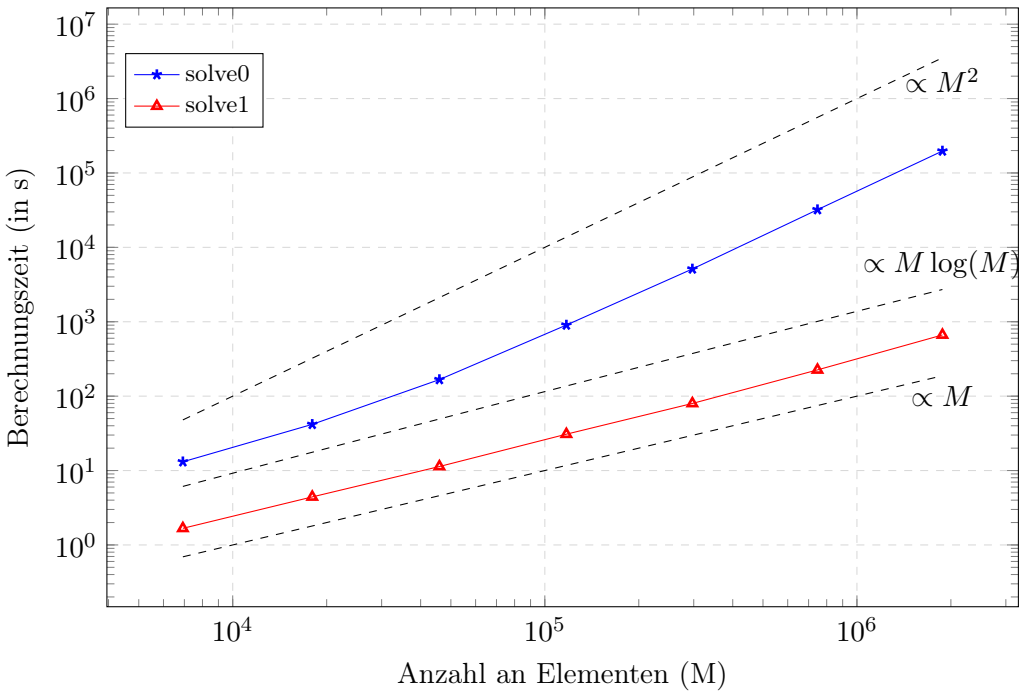


Abbildung 6.1: Die Laufzeit der beiden Algorithmen `solve0` und `solve1` in Abhängigkeit von der Anzahl der Elemente. Man kann sehen, dass der langsame Python-Code 5.1 quadratischen Aufwand hat, hingegen der neue effizientere Python-Code 6.1 linearen Aufwand besitzt.

aus dem Python-Code 5.1 quadratische Laufzeit hat, wobei im Vergleich dazu der effizientere Algorithmus `solve1` aus dem Python-Code 6.1, den wir im Folgenden analysieren werden, linearen Aufwand besitzt. Die hohe Komplexität kommt daher, dass eine csc-Matrix intern in Form von drei Arrays gespeichert wird: der Index des ersten Elements jeder Zeile, die Spaltenindizes der Elemente und die Werte der Elemente. Beim Hinzufügen von Einträgen kann es daher sein, dass man die Indizes der ersten Elemente oder die Spaltenindizes aller

nachfolgenden Elemente in der gleichen Spalte aktualisieren muss. Dies führt dann insgesamt zu einen quadratischen Aufwand, da wir ja für jede Aktualisierung i einen Aufwand von $\mathcal{O}(i)$ haben und somit insgesamt auf $\mathcal{O}(N^2)$ kommen mit $N = \#T$. Das nachfolgende Beispiel soll dies veranschaulichen:

Beispiel	**interne Speicherung als csc-Matrix**
$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 3 & 5 & 7 \\ 4 & 0 & 0 & 8 \\ 0 & 2 & 0 & 0 \end{pmatrix}$	$val = (1, 4, 3, 2, 5, 7, 8)$ $rowInd = (0, 2, 1, 3, 1, 1, 2)$ $colPtr = (0, 2, 4, 5, 7)$

Tabelle 6.1: Ein einfaches Beispiel zur Veranschaulichung der internen Speicherung von Python für csc-Matrizen.

Tabelle 6.1 zeigt, wie eine beispielhafte csc-sparse-Matrix in den drei Vektoren *val*(*„values“*), *rowInd*(*„row indices“*) und *colPtr*(*„column pointer“*) abgespeichert wird. Die Arrays *val* und *rowInd* enthalten 7 Elemente, dies entspricht genau der Anzahl der Nichtnullelemente in A. Das Array *colPtr* enthält 5 Elemente, was immer die Anzahl der Spalten addiert mit 1 ist. Um die Matrix aus den drei Vektoren zu rekonstruieren geht man den Vektor *colPtr* durch und sieht durch die ersten beiden Einträge, dass zwei Elemente in der ersten Spalte (Spaltenindex 0) enthalten sind. Man würde dann mit *rowInd* die einzelnen Einträge von *val* an die Position setzen können. Zum Beispiel würde hier die 1 an die Stelle $(0, 0)$ und die 4 an die Stelle $(0, 2)$ der Matrix gehören. Danach würde man weiter im Vektor *colPtr* die Zahlen 2 und 4 betrachten und erkennen, dass wieder zwei Zahlen in Spalte 2 (Spaltenindex 1) sind. Die Form der Matrix $A \in \mathbb{R}^{M \times N}$ wird zusätzlich in den Metadaten gespeichert, wobei die Anzahl der Spalten auch in dem Vektor *colPtr* abgelesen werden kann, da gilt $M = length(colPtr) - 1$.
Um nun das Problem des Aufwandes zu beschreiben, fügen wir der Matrix eine Zahl hinzu: Angenommen wir wollen an die Stelle $(0, 2)$ die Zahl 8 setzen, dann würde sich die interne Speicherung wie folgt verändern:

Beispiel	**interne Speicherung als csc-Matrix**
$A = \begin{pmatrix} 1 & 0 & 8 & 0 \\ 0 & 3 & 5 & 7 \\ 4 & 0 & 0 & 8 \\ 0 & 2 & 0 & 0 \end{pmatrix}$	$val = (1, 4, 3, 2, 8, 5, 7, 8)$ $rowInd = (0, 2, 1, 3, 0, 1, 1, 2)$ $colPtr = (0, 2, 4, 6, 8)$

Tabelle 6.2: Ein einfaches Beispiel zur Veranschaulichung der internen Speicherung von Python für csc-Matrizen.

Wir können sehen, dass durch das Hinzufügen einer Zahl alle Vektoren verändert werden

und vor allem die relevanten Einträge danach verschoben werden müssen. [1].
Im folgenden Abschnitt werden wir die Assemblierung der Matrix $S = A+B+C$ optimieren, um eine (nahezu) lineare Laufzeit zu erreichen, selbst wenn wir das effiziente csc-Format verwenden.

6.2 Vektorisierung des Problems

In diesem Abschnitt werden wir eine effizienteren Methode zur Assemblierung der Steifigkeitsmatrix betrachten.

```
1  I = np.zeros(9*nE)
2  J = np.zeros(9*nE)
3  A = np.zeros(9*nE)
4
5  for i in range(nE):
6      nodesidx = elements[i,:]
7      nodes = coordinates[nodesidx-1,:]
8      P = np.vstack((np.ones(3), np.transpose(nodes)))
9      areaT = np.linalg.det(P)/2
10     grad = np.linalg.solve(P, np.array([[0, 0], [1, 0], [0, 1]]))
11     sT = np.array(nodes).sum(axis=0) / 3
12     idx = np.arange(9*i, 9*i+9)
13     I[idx] = np.repeat(nodesidx, 3)
14     J[idx] = np.tile(nodesidx, 3)
15     A[idx] = np.reshape((areaT * (grad @ a @ np.transpose(grad))),(9,))
16     A[idx] += np.tile(np.transpose( \
17         (areaT * (grad @ b(sT[0], sT[1])) / 3))[0],3)
18     A[idx] += np.repeat((areaT * c(sT[0], sT[1]) / 9),9)
19 S = sparse.csc_matrix((A, (I-1, J-1)), shape=(nC, nC))
```

Python-Code 6.1: Eine fast-lineare Implementierung der Assemblierung der Steifigkeitsmatrix

Um das Bearbeiten der csc-Matrix effizienter zu machen, verwenden wir Methoden zur Vektorisierung. Das bedeutet, dass wir anstatt jedesmal die csc-Matrix zu verändern, die Matrix zuerst in ein Koordinatenformat bringen, d.h. alle Einträge in einen Vektor speichern und erst am Ende eine csc-Matrix erstellen. Im Folgenden wird der Code 6.1 Schritt für Schritt erklärt:

- **Zeile 1–3:** Im Vergleich zu vorher werden jetzt nicht drei csc-Matrizen erstellt, sondern 3 Vektoren. Da wir für jedes Dreieck in der Triangulierung 9 Einträge verändern, haben die Vektoren die Länge $9\#\mathcal{T}$.
- **Zeile 12–14:** Hier stellen wir die einzelnen Vektoren auf, mit denen wir am Ende die Matrix zusammen setzen wollen. In die Vektoren I und J schreiben wir die jeweiligen Knoten, die gerade betroffen sind. Das bedeutet, wenn das erste Dreieck aus den Knoten z_1, z_2, z_3 besteht, bekommen wir für $i = 0$

$$\begin{aligned} idx &= [0,1,2,3,4,5,6,7,8], \\ I[idx] &= [1,1,1,2,2,2,3,3,3], \\ J[idx] &= [1,2,3,1,2,3,1,2,3]. \end{aligned}$$

Somit treffen wir mit $I[idx]$ und $J[idx]$ dieselben Einträge, die wir vorher mit der Matrix treffen wollten.

- **Zeile 15–18:** Jetzt berechnen wir die Werte exakt wie im vorherigen Code nur speichern wir sie diesmal in Vektoren ab. Das bedeutet, die 3×3 Matrix wird zeilenweise in einen Vektor gespeichert. Es es ist gut zu sehen, dass wir exakt dieselben Matrizen berechnen wie im vorherigen Python-Code 5.1 in Zeile 17–22.

- **Zeile 19:** Am Ende wir durch den Befehl *sparse.csc_matrix* die Matrix so zusammengesetzt, dass gilt:
$$S[I[k], J[k]] = A[k].$$
Außerdem legen wir die Größe der Matrix $nC \times nC$ explizit fest. Hierbei ist zu beachten, dass mehrfache Indexpaare
$$A(i,j) := \{k = 0, \ldots, 9\#\mathcal{T} - 1 : (I(k), J(k)) = (i,j)\}$$
bei der Assemblierung der Matrix auf Addition der Einträge führt, d.h.
$$S_{ij} = \sum_{k \in A(i,j)} A[k].$$

Mit dieser Implementierung der Berechnung der Steifigkeitsmatrix ist der Aufwand des ganzen Verfahren nahezu linear, wie wir in den empirischen Experiment unten beobachten. Im Weiteren werden wir die effizientere Umsetzung 6.1 zusammen mit dem restlichen Code von 5.1 `solve1` nennen.

7 Lokale Netzverfeinerung

Da die Genauigkeit der diskreten Approximation U von u von der Triangulierung $\mathcal{T}$ abhängig ist, betrachten wir in diesem Abschnitt lokale Netzverfeinerung. Durch die Triangulierung gilt es f, g, u_D und die Singularitäten von u aufzulösen. Mit einem adaptiven Algorithmus, der in einem späteren Kapitel besprochen wird, kann das automatisch passieren. Zunächst werden wir *„newest vertex bisection“* (NVB) genauer betrachten. Dies ist ein Algorithmus, um konforme Triangulierungen lokal zu verfeinern.

7.1 Geometrische Daten erstellen

Um das Netz lokal zu verfeinern, braucht der NVB-Algorithmus mehr Informationen als nur `coordinates, elements, dirichlet, neumann`. Konkret brauchen wir eine Nummerierung der Kanten von $\mathcal{T}$. Außerdem müssen wir wissen, welche Kanten zu welchem Dreieck (`elements`) und welche Knoten zu welcher Kante gehören. Der folgende Code soll dieses Problem lösen und zusätzliche Datenstruktur schaffen.

```
def provideGeometricData(elements, *bnds):
    nE = elements.shape[0]
    nB = len(bnds)
    I = np.hstack((elements[:, 0], elements[:, 1], elements[:, 2]))
    J = np.hstack((elements[:, 1], elements[:, 2], elements[:, 0]))
    pointer = np.hstack((1, 3*nE, np.zeros(nB, dtype=int)))
    for j, boundary in enumerate(bnds):
        if boundary.size > 0:
            I = np.hstack((I, boundary[:, 1]))
            J = np.hstack((J, boundary[:, 0]))
        pointer[j+2] = pointer[j+1] + boundary.shape[0]

    #Nummerierung der Kanten
    idxIJ = np.where(I < J)[0]
    edgeNumber = np.zeros(I.shape, dtype=int)
    edgeNumber[idxIJ] = np.arange(idxIJ.shape[0])+1
    idxJI = np.where(I > J)[0]
    number2edges = csc_matrix((np.arange(1, idxIJ.shape[0]+1), \
        (I[idxIJ], J[idxIJ])))
    numberingIJ = find(number2edges)[2]
    idxJI2IJ = find(csc_matrix((idxJI+1, (J[idxJI], I[idxJI]))))[2]
    edgeNumber[idxJI2IJ-1] = numberingIJ

    #Erstelle element2edges, edge2nodes und boundary2edges
    element2edges = edgeNumber[:3*nE].reshape((nE, 3),order='F')
    edge2nodes = np.column_stack((I[idxIJ], J[idxIJ]))
    bnds2edges = []
    for j in range(nB):
        bnds2edges.append(edgeNumber[pointer[j+1]:pointer[j+2]])
```

```
30      return edge2nodes, element2edges, bnds2edges
```

Python-Code 7.1: Eine schnelle Berechnung geometrischer Daten

Um ein besseres Verständnis für diese Implementierung zu bekommen, wird sie im Folgenden erklärt:

- **Zeile 1:** Beispielsweise wird die Funktion mit dem Befehl:

  `provideGeometricData(elements,dirichlet,neumann)`

  aufgerufen, wobei `dirichlet` und `neumann` in dem Argument `*bnds` versteckt sind. Das bedeutet wir könnten den Rand des Gebietes Γ auch in mehrere Teile unterteilen. Im Gesamten soll die Funktion mehrere Arrays zurückgeben. Als erstes soll gelten, dass für eine Kante E_ℓ der Vektor `edge2node[`ℓ`,:]` die Indizes j, k enthält, wobei $z_j, z_k \in \mathcal{N}$, sodass $E_\ell = \operatorname{conv}\{z_j, z_k\}$. Ferner gilt `element2edge[i,`$\ell$`]` die Nummer der Kante an die zwischen `element[i,`ℓ`]` und `element[i,`ℓ`+1]` liegt, wobei $\ell + 1 = 3 \equiv 0$ interpretiert wird. Als letztes soll das Programm auch die Nummern der Kanten des Randes zurückgeben. Dies wird mit `bnds2edges` realisiert. In diesem Argument kann zum Beispiel `dirichlet2edge[`ℓ`]` gespeichert werden, was die Nummer der ℓ-ten Kante des Dirichlet-Randes ausgibt.

- **Zeile 4–5:** Hier erstellen wir zwei Arrays I und J, sodass die Kanten von $\mathcal{T}$ so beschrieben werden, dass $(i, j) \in \{(I_\ell, J_\ell) : \ell = 0, 1, 2, \dots\} =: G$ jeweils die Knotenindizes einer Kante sind. Dies wird umgesetzt indem wir alle Knoten der Elemente spaltenweise speichern, jedoch für J in der zweiten Spalte beginnen.

- **Zeile 6–11:** Wir beachten, dass bis jetzt alle Kanten im Inneren des Gebietes zweimal in G vorkommen, einmal als Paar $(i, j) \in G$ und einmal als Paar $(j, i) \in G$. Also erweitern wir die Arrays mit den zusätzlichen Paaren des Randes, sodass auch diese zweimal in G vorkommen.

- **Zeile 14–22:** Jetzt erstellen wir eine Nummerierung, sodass `edgeNumber[`ℓ`]` die Kante (I_ℓ, J_ℓ) zurückgibt. Um dies einfach zu machen, betrachten wir zuerst alle Paare (i, j) mit $i < j$ und nummerieren diese (Zeile 14-16). Hier ist zu beachten, dass wir bei 1 zu zählen beginnen. Als letztes müssen wir beachten, dass wir für die Paare (j, i) dieselben Nummern vergeben wie für (i, j). Dafür speichern wir G in einer csc-Matrix ab und haben so eine obere Dreiecksmatrix, wobei die Einträge bereits die richtige Nummerierung enthalten (Zeile 18–19). Als Nächstes verwenden wir die Dimensionen der Transponierten G^T in `idxJI2IJ`, wobei die Einträge die Indizes im Bezug auf I und J sind. Beachte, dass wir in Zeile 21 zu `idxJI` eine 1 addieren müssen, da sonst die Funktion `find` den Index 0 nicht "findet". Somit erhöhen wir hier alle Indizes um 1 und verringern beim Initialisieren die Indizes wieder. Mit diesen Informationen kann man dann die richtige Nummerierung für alle Paare (j, i) mit $i < j$ abspeichern (Zeile 22).

- **Zeile 27–29:** Hier werden die Outputs für den Rand wie oben beschrieben erstellt.

Mit diesen gewonnenen geometrischen Daten können wir jetzt den Netzverfeinerungsalgorithmus *newest vertex bisection* (NVB) besprechen.

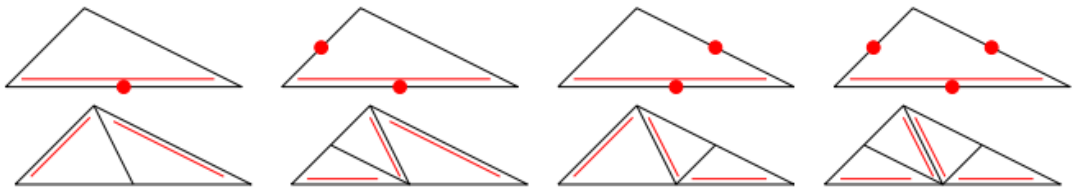


Abbildung 7.1: Illustration von NVB aus [4]: Für jedes Dreieck $T \in \mathcal{T}$ gibt es eine Referenzkante, welche hier mit der roten Linie markiert ist. In der oberen Reihe sieht man die Mittelpunkte für alle Fälle, wie das Dreieck unterteilt werden kann, eingezeichnet. In der unteren Reihe sieht man die resultierende Verfeinerung und die Markierung der neuen Referenzkanten.

7.2 Newest vertex bisection (NVB)

Bevor wir die Implementierung genauer anschauen, wollen wir die Idee hinter NVB betrachten. Dafür beginnen wir mit einer Triangulierung $\mathcal{T}_0$. Für jedes Dreieck $T \in \mathcal{T}_0$ wählt man eine sogenannte *Referenzkante*. Hier könnte man zum Beispiel einfach die längste Kante des Dreieck wählen. Jetzt folgt NVB einer induktiven Verfeinerungsregel, die im Folgenden beschrieben wird:
Wir nehmen an, dass $\mathcal{T}_\ell$ schon eine Verfeinerung von $\mathcal{T}_0$ ist, die mit der Methode von *newest vertex bisection* durchgeführt wurde.

- Um eine Kante $T \in \mathcal{T}_\ell$ zu verfeinern, nehmen wir den Mittelpunkt x_T der Referenzkante E_T und bekommen so einen neuen Knoten. Jetzt kann T entlang x_T und dem gegenüberliegenden Knoten in zwei Dreiecke geteilt werden (sog. Bisektion).

- Dadurch werden die gegenüberliegenden Kanten des neuen Knoten x_T zu den Referenzkanten der neuen Dreiecke T_1 und T_2 (siehe Abbildung 7.1).

- Nachdem alle markierten Dreiecke halbiert wurden, bleiben üblicherweise *hängende Knoten* zurück. Das sind Knoten, die auf einer Kante eines Dreiecks liegen, aber nicht Eckpunkte dieses Dreiecks sind. Hierfür führen wir zusätzliche Teilungen hinzu, um am Ende auf eine reguläre Triangulierung $\mathcal{T}_{\ell+1}$ zu kommen (siehe Abbildung 7.2).

Mit ein paar Überlegungen erkennt man, dass der letzte Schritt, der zu einer regulären Triangulierung führt, nur endlich viele zusätzliche Halbierungen zur Folge hat:

- Wir betrachten anstelle von markierten Elementen markierte Kanten. Das bedeutet, dass sobald ein Dreieck markiert ist, alle Kanten des Dreiecks zur Verfeinerung markiert werden.

- Wenn wir ein beliebiges Dreieck T teilen wollen, stellen wir sicher, dass die Referenzkante markiert ist. Dies geschieht rekursiv in höchstens $3\#\mathcal{T}_\ell$ Rekursionen, da dann alle Kanten markiert wären.

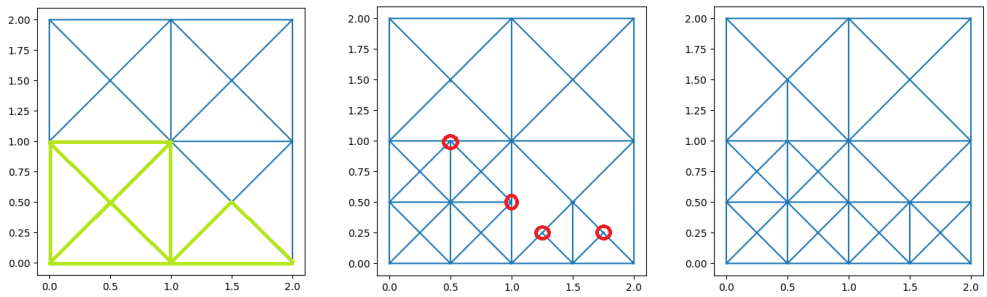


Abbildung 7.2: Die zu verfeinernde Triangulierung (links) enthält 16 Dreiecke. Wenn man jetzt die ersten 5 Elemente für die NVB-Methode markiert und die Verfeinerung durchführt, würde das Ergebnis die Triangulierung in der Mitte ergeben. Diese ist jedoch nicht regulär, da sie hängende Knoten enthält (mit roten Kreisen markiert). Deswegen führen wir weitere Bisektionen durch, um auf die richtige Triangulierung (rechts) zu kommen.

- Wenn ein Dreieck halbiert wird, wird die Referenzkante halbiert, während die beiden anderen Kanten zu den Referenzkanten der beiden neuen Dreiecke werden. Die Verfeinerung von T in 2, 3 oder 4 Dreiecke kann dann in einem Schritt erfolgen, siehe [4].

7.3 Implementierung von NVB

Der Python-Code 7.2 zeigt eine Implementierung des NVB Algorithmus, wobei wir eine bestimmte Konvention verwenden: Für jedes Dreieck T_ℓ, welches gespeichert ist als

$$\texttt{elements}[\ell, :] = [i \quad j \quad k],$$

gilt, dass z_k der *newest vertex* (neuer Scheitelpunkt) und die Referenzkante $E = \mathrm{conv}\{z_i, z_j\}$ sind.

```
def refineNVB(coordinates, elements, marked, *bnds):
    nE = elements.shape[0]

    # Geometrische Daten bekommen
    edge2nodes, element2edges, *boundary2edges = \
        provideGeometricData(elements, *bnds)

    # Die Elemente der Verfeinerung markieren
    edge2newNode = np.zeros(np.max(np.max(element2edges)))
    edge2newNode[element2edges[marked]-1] = 1
    swap = np.array(1)
    while swap.size>0:
        markedEdge = edge2newNode[element2edges-1]
        swap = np.nonzero(~markedEdge[:, 0].astype(int) & \
            (markedEdge[:, 1].astype(int) | \
            markedEdge[:, 2].astype(int)))[0]
        edge2newNode[element2edges[swap, 0]-1] = 1

    # Neue Knoten erstellen
    edge2newNode[edge2newNode != 0] = (coordinates.shape[0] + \
```

```
        np.arange(np.count_nonzero(edge2newNode))+1)
    idx = np.nonzero(edge2newNode)[0]
    coordinates = np.concatenate((coordinates, \
        np.zeros((idx.shape[0], 2))), axis=0)
    coordinates[edge2newNode[idx].astype(int)-1, :] = \
        (coordinates[edge2nodes[idx, 0]-1, :] + \
        coordinates[edge2nodes[idx, 1]-1, :]) / 2

    # Die Kanten der Randbedingungen verfeinern
    newbnds = []
    for j in range(len(bnds)):
        boundary = bnds[j]
        if boundary.size > 0:
            newNodes = edge2newNode[boundary2edges[j]-1]
            markedEdges = np.nonzero(newNodes)[0]
            if markedEdges.size > 0:
                boundary = np.concatenate((boundary[~newNodes.astype(bool) \
                    , :], np.array([boundary[markedEdges, 0], \
                    newNodes[markedEdges]], ndmin=2).T, \
                    np.array([newNodes[markedEdges], boundary[markedEdges, \
                    1]], ndmin=2).T), axis=0).astype(int)
        newbnds.append(boundary)

    # Neue Knoten der Verfeinerung bereitstellen
    newNodes = edge2newNode[element2edges-1]
    # Die Art der Verfeinerung jedes Elemetes definieren
    markedEdges = (newNodes != 0)
    none = ~markedEdges[:, 0]
    bisec1 = (markedEdges[:, 0] & ~markedEdges[:, 1] & ~markedEdges[:, 2])
    bisec12 = (markedEdges[:, 0] & markedEdges[:, 1] & ~markedEdges[:, 2])
    bisec13 = (markedEdges[:, 0] & ~markedEdges[:, 1] & markedEdges[:, 2])
    bisec123 = (markedEdges[:, 0] & markedEdges[:, 1] & markedEdges[:, 2])

    # Platz schaffen fuer die neuen Elemente
    idx = np.ones(nE, dtype=int)
    idx[bisec1] = 2 # NVB an der ersten Kante
    idx[bisec12] = 3 # NVB an der ersten und zweiten Kante
    idx[bisec13] = 3 # NVB an der ersten und dritten Kante
    idx[bisec123] = 4 # NVB an allen Kanten
    idx = np.concatenate(([0], np.cumsum(idx)))
    # Neue Elemente erstellen
    newElements = np.zeros((idx[-1], 3), dtype=int)
    idx=idx[:-1]

    # Definiere neue Elemente fuer bisec1
    newElements[idx[none], :] = elements[none, :]
    newElements[[idx[bisec1], 1+idx[bisec1]], :] = np.transpose( \
        np.array([[elements[bisec1, 2], elements[bisec1, 0], \
    newNodes[bisec1,0]], elements[bisec1, 1], elements[bisec1, 2], \
        [newNodes[bisec1, 0]]]), (0, 2, 1))
    # Definiere neue Elemente fuer bisec12
    newElements[[idx[bisec12], 1+idx[bisec12], 2+idx[bisec12]], :] = \
        np.transpose(np.array([[elements[bisec12, 2], \
        elements[bisec12, 0], newNodes[bisec12, 0]], [newNodes[bisec12, 0],\
        elements[bisec12, 1], newNodes[bisec12, 1]], [elements[bisec12, 2],\
```

```
76          newNodes[bisec12, 0], newNodes[bisec12, 1]]]), (0, 2, 1))
77      # Definiere neue Elemente fuer bisec13
78      newElements[[idx[bisec13], 1+idx[bisec13], 2+idx[bisec13]], :] = \
79          np.transpose(np.array([[newNodes[bisec13, 0], elements[bisec13, 2],\
80          newNodes[bisec13, 2]], [elements[bisec13, 0], newNodes[bisec13, 0],\
81          newNodes[bisec13, 2]], [elements[bisec13, 1], elements[bisec13, 2],\
82          newNodes[bisec13, 0]]]), (0, 2, 1))
83      # Definiere neue Elemente fuer bisec123
84      newElements[[idx[bisec123], 1+idx[bisec123], 2+idx[bisec123], \
85          3+idx[bisec123]], :] = \
86          np.transpose(np.array([[newNodes[bisec123, 0], \
87          elements[bisec123, 2], newNodes[bisec123, 2]], \
88          [elements[bisec123, 0], newNodes[bisec123, 0], \
89          newNodes[bisec123, 2]], [newNodes[bisec123, 0], \
90          elements[bisec123, 1], newNodes[bisec123, 1]], \
91          [elements[bisec123, 2], newNodes[bisec123, 0], \
92          newNodes[bisec123, 1]]]), (0, 2, 1))
93
94      return coordinates, newElements, *newbnds
```

Python-Code 7.2: NVB Verfeinerung

- **Zeile 1:** Die Funktion `refineNVB` bekommt die Vektoren `coordinates` und `elements`, welche die Koordinaten der Knoten bzw. die Dreiecke angeben, sowie den Vektor `marked`, der die Elemente enthält, die verfeinert werden sollen. Außerdem muss man der Funktion noch weitere Vektoren der Randbedingungen übergeben. Am Ende wollen wir eine reguläre Triangulierung bekommen, in welcher zumindest die markierten Dreiecke geteilt wurden. Beispielsweise wird die Funktion mit dem Befehl

  `refineNVB(coordinates,elements,marked,dirichlet,neumann)`

  aufgerufen, wobei wieder `dirichlet` und `neumann` in dem Argument `*bnds` versteckt sind.

- **Zeile 5–6:** Wir beginnen damit die geometrischen Daten, die wir in Abschnitt 7.2 mit der Funktion 7.1 aufgestellt haben, zu erstellen.

- **Zeile 9–10:** Als Nächstes erstellen wir einen Vektor `edge2newNode`, wobei `edge2newNode[`ℓ`]` genau dann nicht-null ist, wenn die ℓ-te Kante verfeinert werden soll. In Zeile 10 markieren wir alle Kanten von den markierten Dreiecken, um sie zu verfeinern. Dies führt dazu, dass markierte Dreiecke mit drei Bisektionen verfeinert werden. Man könnte hier auch nur die Referenzkante markieren, was dazu führen würde, dass jedes markierte Dreieck nur einmal unterteilt wird.

- **Zeile 11–17:** Hier müssen wir sicherstellen, dass, wenn eine Kante eines Dreiecks für die Verfeinerung markiert ist, auch dessen Referenzkante markiert wird. Diese Schleife endet im schlechtesten Fall nach $\#\mathcal{T}$ Schritten.

- **Zeile 20–27:** Der Mittelpunkt jeder markierten Kannte wird zu einem neuen Knoten. Die Anzahl an neuen Knoten finden wir durch die Nicht-Null-Einträge im Vektor `edge2newNode`.

- **Zeile 30–42:** Die ℓ-te Kante auf dem Rand ist genau dann markiert, wenn `newNodes[`ℓ`]` nicht-null ist. In diesem Fall beinhaltet der Vektor den Mittelpunkt der Kante. Wenn mindestens eine Kante vom Rand markiert ist, wird die Randbedingung aktualisiert (Zeile 35–42).

- **Zeile 45–52:** Wir generieren hier ein neues Array, sodass `newNodes[i,`ℓ`]` nicht-null ist genau dann, wenn die ℓ-te Kante des Dreiecks T_i für die Verfeinerung markiert ist. In diesem Fall beinhaltet es wieder den Mittelpunkt der Kante (Zeile 45). Um den Code zu beschleunigen, verwenden wir die logische Indizierung und berechnen ein logisches Array `markedEdges`, dessen Eintrag `markedEdges[i,`ℓ`]` nur angibt, ob die ℓ-te Kante von Element T_i zur Verfeinerung markiert ist oder nicht. Die Vektoren `none, bisec1, bisec12, bisec13` und `bisec123` enthalten die Indizes der Elemente gemäß der jeweiligen Verfeinerungsregel. Zum Beispiel enthält `bisec12` alle Elemente, bei denen die erste und die zweite Kante zur Verfeinerung markiert sind. Hierbei sei daran erinnert, dass entweder keine oder zumindest die erste Kante (Referenzkante) markiert ist.

- **Zeile 55–63:** Unser Ziel ist es, die Nummerierung der Dreiecke in dem Sinne zu erhalten, dass die neu generierten Dreiecke fortlaufende Nummern haben. Die Elemente von `bisec1` werden zu zwei Elementen verfeinert, die Elemente von `bisec12` und `bisec13` werden zu drei Elementen verfeinert, die Elemente von `bisec123` werden zu vier Elementen verfeinert. Das bedeutet, dass wir in dem Array `newElements` extra Platz machen für die neuen Elemente (Zeile 62). Zu beachten ist auch, dass wir in Zeile 63 die letzte Zahl des Index löschen, um später mit den gleichlangen Boolean-Arrays `none, bisec1, bisec12, bisec13` und `bisec123` zu initialisieren.

- **Zeile 66–92:** Bei der Verfeinerung beachten wir eine bestimmte Reihenfolge der neuen Dreiecke: Wenn T durch die NVB-Methode verfeinert wird, zerlegt es sich in T_ℓ und $T_{\ell+1}$. Dabei ist T_ℓ das linke Element im Bezug auf die Bisektionskante.

- **Zeile 94:** Am Ende geben wir die Vektoren `coordinates` und `newElements` zurück, die die neuen Koordinaten bzw. die neuen Dreiecke enthalten. Außerdem werden auch die neuen Randbedingungen zurückgegeben.

8 A posteriori Fehlerschätzer und adaptive Netzverfeinerung

In der Regel sind Rechenzeiten und Speicherplatz limitiert für numerische Berechnungen. Deswegen brauchen wir eine bestimmte „Abbruchbedingung“, bei der der Algorithmus aufhört noch weiter zu verfeinern. Meistens wird dies in der Numerik mit einer Grenze an Elementen umgesetzt. Das bedeutet wir suchen eine Triangulierung $\mathcal{T}$, sodass die Anzahl der Elemente $M = \#\mathcal{T} \leq M_{max}$ unter einer gewissen maximalen Anzahl bleibt und zusätzlich der Fehler $\|u - U\|_{H^1(\Omega)}$ quasi-minimal ist, d.h. der Fehler ist bis auf eine generische multiplikative Konstante minimal.
Ein solches Netz $\mathcal{T}$ wird meistens auf iterative Weise gewonnen. Hierfür sei $\eta_T \in \mathbb{R}$ für jedes Element $T \in \mathcal{T}$ der Verfeinerungsindikator, welcher (zumindest heuristisch) folgende Bedingung erfüllt:

$$\eta_T \approx \|u - U\|_{H^1(T)} \quad \text{für alle } T \in \mathcal{T}. \tag{32}$$

Insbesondere gilt dann für den Fehlerschätzer $\eta = \left(\sum_{T\in\mathcal{T}} \eta_T^2\right)^{1/2}$ auch $\eta \approx \|u - U\|_{H^1(\Omega)}$. Wichtig zu wissen ist, dass man auch ohne u zu kennen η_T berechnen und so den lokalen Fehler $\|u - U\|_{H^1(T)}$ zumindest schätzen kann [4].

8.1 Adaptiver Algorithmus

Mit einem gegebenen Verfeinerungsindikator $\eta_T \approx \|u-U\|_{H^1(T)}$ können wir Elemente $T \in \mathcal{T}$ für die Verfeinerung markieren. Dies machen wir, indem wir versuchen eine kleinste Menge $\mathcal{M} \subseteq \mathcal{T}$ zu finden, für die gilt

$$\theta \sum_{T\in\mathcal{T}} \eta_T^2 \leq \sum_{T\in\mathcal{M}} \eta_T^2, \tag{33}$$

wobei $\theta \in (0, 1]$, d.h. die Indikatoren der markierten Elemente kontrollieren einen fixen Prozentsatz θ aller Indikatoren. Dann können wir eine neue Triangulierung $\mathcal{T}'$ basierend auf $\mathcal{T}$ erzeugen, indem wir zumindest die markierten Elemente $T \in \mathcal{M}$ verfeinern, um den Fehler $\|u - U\|_{H^1(\Omega)}$ zu reduzieren. Zu beachten ist, dass $\theta \to 1$ einer nahezu gleichmäßigen Netzverfeinerung entspricht, d.h. die meisten Elemente sind zur Verfeinerung markiert, während $\theta \to 0$ zu stark adaptierten Netzen führt [4].

```
def adaptiveAlgorithm(coordinates, elements, dirichlet, neumann, f, g, uD, \
    a, b, c, nEmax, theta):
    while True:
        # Berechne die diskrete Loesung
        x = solve1(coordinates,elements,dirichlet,neumann,f,g,uD,a,b,c)
        # Berechne die Verfeinerungsindikatoren
        indicators = computeEtaR(x, coordinates, elements, dirichlet, \
            neumann, f, g, a, b, c)
```

```
9           # Abbruchbedingung
10          if elements.shape[0] >= nEmax:
11              break
12          # Markiere die Elemente zur Verfeinerung
13          idx = indicators.argsort()[::-1]
14          sumeta = np.cumsum(indicators[idx])
15          m = np.argmax(sumeta >= sumeta[-1] * theta) + 1
16          marked = idx[:m]
17          # Netz verfeinern
18          coordinates, elements, dirichlet, neumann = \
19              refineNVB(coordinates, elements, marked, dirichlet, neumann)
20
21      return x, coordinates, elements, indicators
```

Python-Code 8.1: Adaptiver Algorithmus

- **Zeile 1–2:** Die Funktion nimmt neben den üblichen Daten des Netzes auch die Daten des Problems $f, g, u_D, \mathcal{A}, b$ und c. Außerdem muss der Benutzer seine Präferenzen für die maximale Anzahl an Elementen `nEmax` und den adaptiven Parameter θ von (33) angeben. Am Ende wollen wir den Koeffizientenvektor x von der finalen Galerkin-Lösung $U \in \mathcal{S}^1_D(\mathcal{T})$, die finalen Daten der Triangulierung und den zugehörigen Vektor `indicators`, der die Fehlerindikatoren der einzelnen Elementen angibt, bekommen.
- **Zeile 3–19:** Solange die Anzahl der Elemente $M = \#\mathcal{T}$ nicht die maximale Anzahl an Elementen `nEmax` überschreitet, machen wir das Folgende: In Zeile 5 berechnen wir den Koeffizientenvektor der diskreten Lösung. Dann verwenden wir den Fehlerschätzer, der im j-ten Koeffizienten den Wert $\eta_j^2 := \eta_{T_j}^2$ speichert (siehe nächstes Abschnitt 8.2). Dann wird die Abbruchbedingung überprüft, ob die maximale Anzahl der Elemente schon überschritten wurde (Zeile 10–11). In Zeile 13 finden wir eine Permutation π auf $\{1, \dots, M\}$, sodass die Indikatoren $(\eta^2_{\pi(j)})_{j=1}^M$ absteigend sortiert sind. Dann berechnen wir die Summen $\sum_{j=1}^m \eta^2_{\pi(j)}$ für jedes m und suchen den kleinsten Index m, für den gilt $\theta \sum_{j=1}^M \eta_j^2 = \theta \sum_{j=1}^M \eta^2_{\pi(j)} \leq \sum_{j=1}^m \eta^2_{\pi(j)}$ (Zeile 15). In Zeile 16 markieren wir noch die Menge an Dreiecken, die wir verfeinern wollen $\mathcal{M} = \{T_{\pi(j)} : j = 1, \dots, m\}$. Am Ende verfeinern wir die Triangulierung mit der `refineNVB` Funktion und bekommen so ein adaptiertes Netz.

Um diesen adaptiven Algorithmus ausführen zu können, braucht es jetzt noch einen Fehlerschätzer, den wir im folgenden Abschnitt betrachten wollen.

8.2 Residualer Fehlerschätzer

Wir betrachten den residualen Fehlerschätzer $\eta := \left(\sum_{T\in\mathcal{T}} \eta_T^2\right)^{1/2}$ mit dem Verfeinerungsindikator

$$\begin{aligned}\eta_T^2 &= |T| \|f - b\cdot\nabla U - cU\|^2_{L^2(T)} \\ &\quad + |T|^{1/2} \|[\mathcal{A}\nabla U \cdot n]\|^2_{L^2(\partial T\cap\Omega)} + |T|^{1/2}\|g - \mathcal{A}\nabla U\cdot n\|^2_{L^2(\partial T\cap\Gamma_N)},\end{aligned} \tag{34}$$

wobei wir betonen müssen, dass $\operatorname{div}\mathcal{A}\nabla U = 0$ auf T verschwindet, da $\mathcal{A}\nabla U$ nach Voraussetzung $\mathcal{T}$-stückweise konstant ist. Dabei notiert $[(\cdot)\cdot n]|_E$ den Sprung in normaler Richtung über eine innere Kante $E \in \mathcal{E}_\Omega$. Für benachbarte Elemente $T_\pm \in \mathcal{T}$ mit dem äußeren Normalvektor $n_\pm$ ist der Sprung der $\mathcal{T}$-stückweisen konstanten Funktion ∇U über die gemeinsamen Kante $E = T_+ \cap T_- \in \mathcal{E}$ definiert als

$$[\mathcal{A}\nabla U \cdot n]|_E := \mathcal{A}\nabla U|_{T_+} \cdot n_+ + \mathcal{A}\nabla U|_{T_-} \cdot n_-. \tag{35}$$

Für die Implementierung verwenden wir wieder die Schwerpunkt- bzw. Mittelpunkt-Quadratur und ersetzen $f|_T \approx f(s_T)$, $b|_T \approx b(s_T)$, $c|_T \approx c(s_T)$ und $g|_E \approx g(m_E)$. Mit der Definition $\mathcal{E}_T$ als die Kanten eines Dreiecks erhalten wir somit insgesamt die Formel

$$\begin{aligned}\eta_T^2 = |T|^2 \left(f(s_T) - b(s_T)\cdot\nabla U|_T - c(s_T)U(s_T)\right)^2 \\ + |T| \sum_{E\in\mathcal{E}_T\cap\mathcal{E}_\Omega} [\mathcal{A}\nabla U|_T]|_E^2 + |T| \sum_{E\in\mathcal{E}_T\cap\mathcal{E}_N} \left(g(m_E) - \mathcal{A}\nabla U|_T\cdot n\right)^2.\end{aligned} \tag{36}$$

Zu beachten ist wieder, dass in der ersten Summe, wie in (35) beschrieben, zwei Summanden existieren und dass durch die Approximationen der Funktionen die Faktoren der Flächen und Kanten noch dazukommen. Die folgende Implementierung des Fehlerschätzers im Python-Code 8.2 retourniert einen Vektor von quadratischen Verfeinerungsindikatoren $(\eta_{T_1}^2, \ldots, \eta_{T_M}^2)$, wobei $\mathcal{T} = \{T_1, \ldots, T_M\}$:

```
def computeEtaR(x,coordinates,elements,dirichlet,neumann,f,g,a,b,c):
    edge2nodes, element2edges, dirichlet2edges, neumann2edges = \
        provideGeometricData(elements,dirichlet,neumann)
    dirichletedges = set(dirichlet2edges)
    neumannedges = dict(zip(neumann2edges,neumann))

    nE=np.shape(elements)[0]
    etaR = np.zeros((nE,))
    etaR_edges = np.zeros((edge2nodes.shape[0],))
    area = np.zeros((nE,))
    sT = np.zeros((nE,2))
    gradU = np.zeros((nE,2))

    # Vorbereitung pro Dreieck
    for i in range(nE):
        nodes = elements[i,:]
        c1 = coordinates[nodes[0]-1]
        c2 = coordinates[nodes[1]-1]
        c3 = coordinates[nodes[2]-1]
        area[i] = ((c2-c1)[0]*(c3-c1)[1] - (c2-c1)[1]*(c3-c1)[0])/2
        gradV = np.array([[c2[1]-c3[1],c3[1]-c1[1],c1[1]-c2[1]], \
            [c3[0]-c2[0],c1[0]-c3[0],c2[0]-c1[0]]])/(area[i]*2)
        gradU[i] = gradV @ x[nodes-1]
        sT[i] = np.array(coordinates[nodes-1,:]).sum(axis=0)/3

        # Berechnung des Fehlers pro Kante
        for j in element2edges[i]:
            if j in dirichletedges:
                continue
            elif j in neumannedges.keys():
```

```
31                  nodes = neumannedges[j]-1
32                  p = coordinates[nodes[1],1]-coordinates[nodes[0],1]
33                  q = coordinates[nodes[0],0]-coordinates[nodes[1],0]
34                  n = np.array([p,q]) / (np.sqrt(p**2 + q**2))
35                  mE = np.array(coordinates[edge2nodes[j-1,:]-1,:]).\
36                      sum(axis=0)/2
37                  gmE = g(mE[0],mE[1])
38                  etaR_kanten[j-1] += gmE - gradU[i] @ a @ n
39              else:
40                  nodes = edge2nodes[j-1,:]-1
41                  p = coordinates[nodes[1],1]-coordinates[nodes[0],1]
42                  q = coordinates[nodes[0],0]-coordinates[nodes[1],0]
43                  n = np.array([p,q]) / (np.sqrt(p**2 + q**2))
44                  if etaR_edges[j-1] == 0:
45                      etaR_edges[j-1] += gradU[i] @ a @ n
46                  else:
47                      etaR_edges[j-1] += gradU[i] @ a @ -n
48
49      # Berechnung des Fehlers pro Dreieck
50      for i in range(nE):
51          nodes = elements[i,:]
52          bsT = b(sT[i][0],sT[i][1])
53          csT = c(sT[i][0],sT[i][1])
54          fsT = f(sT[i][0],sT[i][1])
55          etaR[i] = np.sum(area[i] * \
56              etaR_edges[element2edges[i]-1]**2, axis=0)
57          UsT = np.sum(x[nodes-1]/3,axis=0)
58          etaR[i] += (area[i] * (fsT- gradU[i] @ bsT - csT*UsT))**2
59
60      return etaR
```

Python-Code 8.2: Residualer Fehlerschätzer

Der Python-Code 8.2 ist folgendermaßen aufgebaut:

- **Zeile 1:** Diesmal bekommt unsere Funktion den Koeffizientenvektor, alle Daten der Triangulierung und zusätzlich die Funktionen f, g, $\mathcal{A}$, b und c, wobei $\mathcal{A} \in \mathbb{R}^{d\times d}_{\text{sym}}$ als konstant vorausgesetzt wird. als
- **Zeile 4–5:** Um die Laufzeit des Programms zu verbessern, wandeln wir die Dirichlet- und Neumann-Kanten in ein Set bzw. in ein Dictionary um.
- **Zeile 15–24:** Wir beginnen damit für jedes Dreieck die Fläche, den Gradienten von U und den Schwerpunkt zu berechnen. Für den Gradienten von U müssen wir zuerst die Gradienten der V_j berechnen. Dafür sagen wir zur Veranschaulichung z_1, z_2, z_3 sind die Knoten eines Dreiecks $T \in \mathcal{T}$ gegen den Uhrzeigersinn geordnet. V_j sind wieder die Hutfunktionen zugehörig zu ihren Knoten $z_j = (x_j, y_j) \in \mathbb{R}^2$. Mit der Identifikation $z_4 \equiv z_1$ und $z_5 \equiv z_2$, Lemma 2 und dem Beweis von Lemma 3 können wir den Gradienten von V_j explizit berechnen und erhalten

$$\nabla V_j|_T = \frac{1}{2|T|}(y_{j+1} - y_{j+2}, x_{j+2} - x_{j+1}). \tag{37}$$

Damit folgt $\nabla U|_T = \sum_{j=1}^{3} x_j \nabla V_j$ aus $U|_T = \sum_{j=1}^{3} x_j V_j$ und wir können in Zeile 23 den Gradienten von U auf dem jeweiligen Dreieck berechnen.

- **Zeile 27–47:** Jetzt beginnen wir für jede Kante $E \in \mathcal{E}$ den Sprungterm $[\mathcal{A}\nabla U \cdot n]|_E$ oder $[g(m_E) - \mathcal{A}\nabla U \cdot n]|_E$ zu berechnen, abhängig davon, ob es eine innere Kante oder eine Kante des Randes ist. In den Zeilen 28, 30 und 39 überprüfen wir wo die betrachtete Kante liegt. Liegt sie auf dem Dirichlet-Rand, überspringen wir sie, da diese nicht in unserem Fehlerschätzer berücksichtigt werden. Liegt sie auf dem Neumann-Rand berechnen wir den Normalvektor und den Mittelpunkt der Kante, um g zu approximieren und berechnen den Fehler. Da wir die Neumann-Randbedingungen extra gegen den Uhrzeigersinn gespeichert haben, zeigen die Normalvektoren auch alle nach außen. Falls die Kante aber im Inneren unseres Gebietes liegt, berechnen wir den Fehler exakt zweimal pro Kante, da wir in der äußeren Schleife (Zeile 15) über jedes Dreieck und in der inneren Schleife (Zeile 27) über alle Kanten des Dreiecks iterieren. Weil $n_+ = -n_-$ gilt, berechnen wir einmal den Fehler mit n_+ und mit dem anderen Dreieck einmal mit n_-. Aufgrund der Tatsache, dass wir nachher das Ergebnis quadrieren ist das exakte Vorzeichen nicht relevant.

- **Zeile 50–58:** Jetzt müssen wir nur noch die restlichen Ergebnisse berechnen und alles zusammenführen. Somit iterieren wir nochmal über alle Dreiecke und approximieren dabei b, c und f mittels Schwerpunkt-Quadratur. Zusätzlich addieren wir jetzt die quadrierten Fehler der Kanten jedes Dreieck und multiplizieren mit $|T|$. Am Ende müssen wir nur noch den ersten Summanden $|T|^2(f(s_T) - b(s_T) \cdot \nabla U - c(s_T)U(s_T))^2$ unserer Formel (36) hinzu addieren. Dazu nutzen wir $V_j(s_T) = 1/3$ im Schwerpunkt und es gilt

$$U(s_T) = \sum_{j=1}^{3} x_j V_j(s_T) = \sum_{j=1}^{3} \frac{x_j}{3}. \tag{38}$$

9 Numerische Experimente

Um die Effizienz des angegebenen Codes zu belegen, konzentriert sich dieses Kapitel auf einige numerische Beispiele. Für das Visualisieren der diskrete Lösung $U = \sum_{j=1}^{N} x_j V_j$ kann man in Python den folgenden Befehl verwenden:

```
import plotly.figure_factory as ff
fig = ff.create_trisurf(coordinates[:,0], coordinates[:,1], x, \
    simplices = elements[:,0:3]-1)
fig.show()
```

Python-Code 9.1: Eine Möglichkeit um die berechnete Lösung zu visualisieren

9.1 Erstes Beispiel

Für das erste Beispiel betrachten wir

$$u(x,y) = x(1-x)y(1-y) \quad \text{in } \Omega = (0,1)^2, \tag{39}$$

welche die Bedingungen $u \in C^\infty(\Omega)$ und $u|_{\partial\Omega} = 0$ erfüllt. Wir definieren $\Gamma_D = \partial\Omega$ und $\Gamma_N = \emptyset$. Somit erhalten wir $f := -\Delta u = -2(-x + x^2 - y + y^2)$ sowie $u_D = 0$. Außerdem gilt $A = I, b = 0$ und $c = 0$. Da die Lösung u glatt ist, muss uniforme Verfeinerung schon die optimale Rate ausgeben. In Abbildung 9.2 können wir die optimale Rate $\Theta(M^{-1/2})$ für die adaptive ($\theta = 0.25$) aber auch uniforme Strategie beobachten. Außerdem ist gut zu sehen, dass der Fehler und der Fehlerschätzer parallel verlaufen. Zusätzlich können wir in Abbildung 9.3 auch die Fehler gegenüber der gebrauchten Laufzeit sehen, welche auch mit der optimalen Rate fallen.

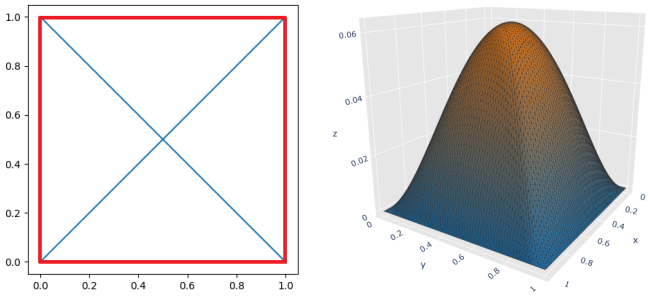


Abbildung 9.1: Das Anfangsgitter von Ω aus Beispiel 1 mit dem Dirichlet-Rand Γ_D in rot markiert (links). Die Lösung $u(x,y) = x(1-x)y(1-y)$ (rechts).

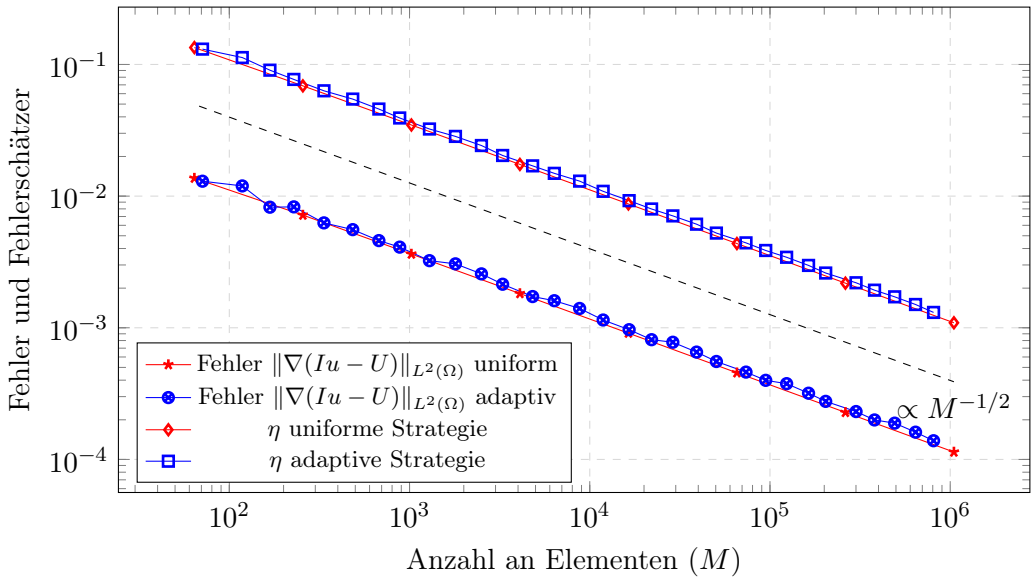


Abbildung 9.2: Der Galerkin Fehler (mit Iu als nodale Interpolation von u) und der residuale Fehlerschätzer η des ersten Beispiels in Abhängigkeit zur Anzahl an Elementen. Wir betrachten sowohl uniforme Netzverfeinerung als auch den adaptiven Algorithmus dem Python-Code 8.1.

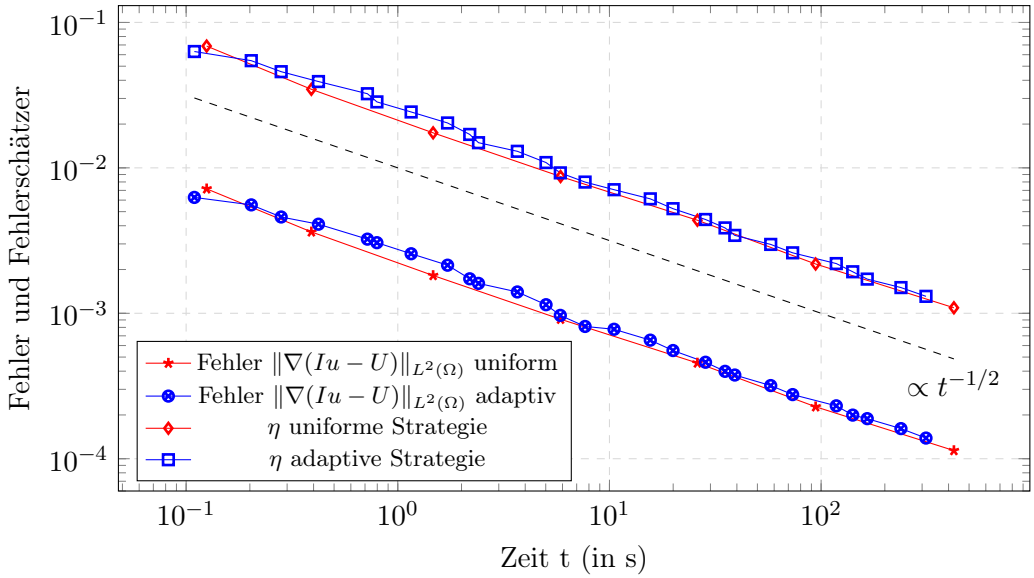


Abbildung 9.3: Der Galerkin Fehler (mit Iu als nodale Interpolation von u) und der residuale Fehlerschätzer η des ersten Beispiels in Abhängigkeit zur Anzahl an Elementen. Wir betrachten sowohl uniforme Netzverfeinerung als auch den adaptiven Algorithmus dem Python-Code 8.1.

9.2 Zweites Beispiel

Im nächsten Beispiel betrachten wir dieselbe Lösung u, aber verändern unsere Randbedingungen. Wir betrachten diesmal $\Gamma_D = \{(x,y) \in \partial\Omega : x = 0 \vee y \in \{0,1\}\}$ und $\Gamma_N = \{(x,y) \in \partial\Omega : x = 1\}$. Somit haben wir $g = (1-2x)y(1-y)$. Und wieder sehen wir in Abbildung 9.5 und in Abbildung 9.6 die optimale Rate für adaptive ($\theta = 0.25$) und uniforme Strategie.

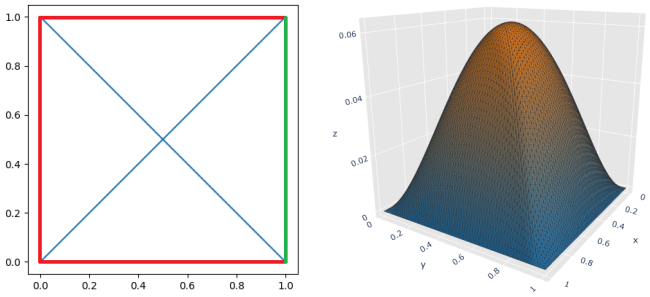


Abbildung 9.4: Das Anfangsgitter von Ω aus Beispiel 2 mit dem Dirichlet- und Neumann-Rand in rot bzw. grün markiert (links). Die Lösung $u(x,y) = x(1-x)y(1-y)$ (rechts).

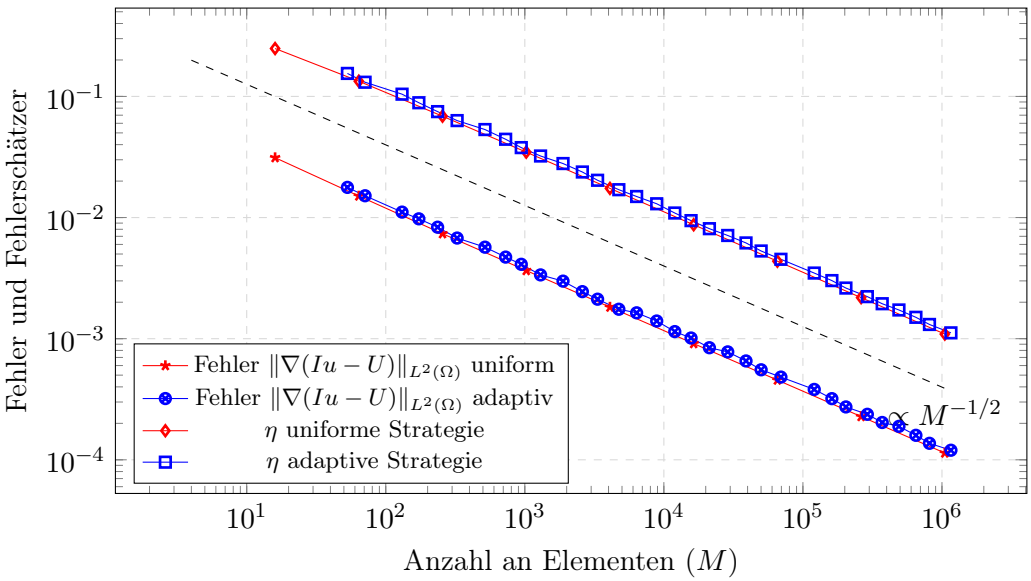


Abbildung 9.5: Der Galerkin Fehler (mit Iu als nodale Interpolation von u) und der residuale Fehlerschätzer η des zweiten Beispiel in Abhängigkeit zur Anzahl an Elementen. Wir betrachten sowohl uniforme Netzverfeinerung als auch den adaptiven Algorithmus aus Python-Code 8.1.

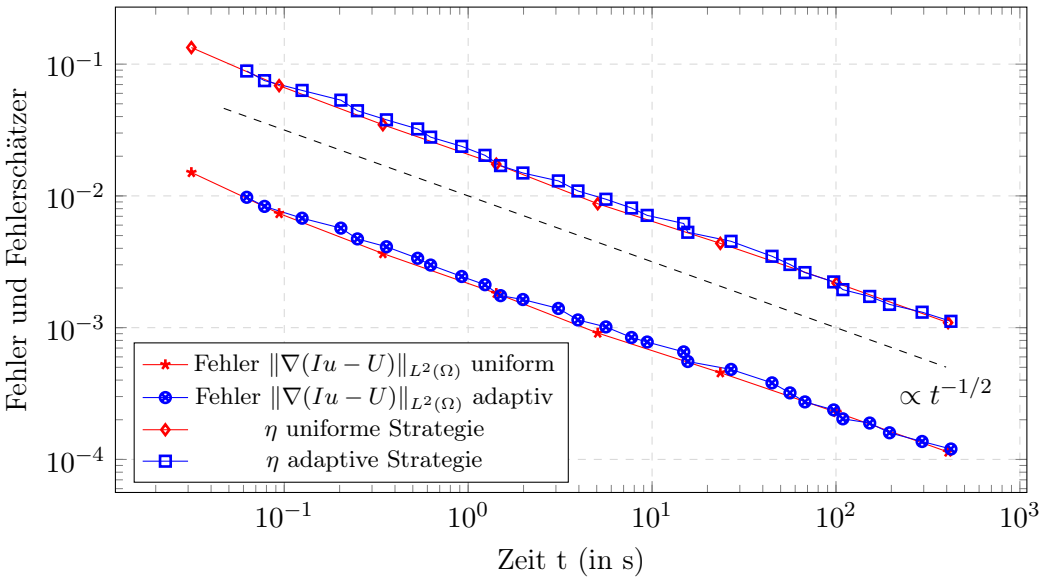


Abbildung 9.6: Der Galerkin Fehler (mit Iu als nodale Interpolation von u) und der residuale Fehlerschätzer η des zweiten Beispiel in Abhängigkeit zur gebrauchten Zeit. Wir betrachten sowohl uniforme Netzverfeinerung als auch den adaptiven Algorithmus aus Python-Code 8.1.

9.3 Drittes Beispiel

Für das dritte Beispiel betrachten wir

$$u(x,y) = xy \quad \text{in } \Omega = (-1,1)^2 \setminus ([0,1]\times[-1,0]). \tag{40}$$

Diesmal wählen wir den Drichlet-Rand und Neumann-Rand wie folgt:

$$\Gamma_D = \{(x,y)\in\partial\Omega : (x=0 \wedge y\in(-1,0)) \vee (x\in(0,1)\wedge y=0)\}, \quad \Gamma_N = \partial\Omega\setminus\Gamma_D. \tag{41}$$

Zusätzlich wählen wir diesmal $A = I, b = (1,0)^T$ und $c = 0$. Somit erhalten wir für $f(x,y) = y, u_D = 0$ und $g = \partial_n u$. Wieder sehen wir in Abbildung 9.8, dass die Rate der uniformen und die der adaptiven Strategie ($\theta = 0.25$) $\Theta(M^{1/2})$ ist.

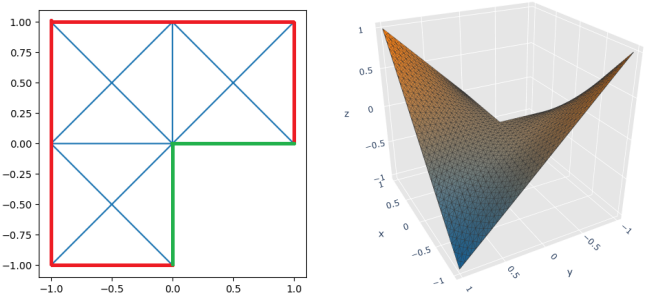


Abbildung 9.7: Das Anfangsgitter von Ω aus Beispiel 3 mit dem Dirichlet- und Neumann-Rand in rot bzw. grün markiert (links). Die Lösung $u(x,y) = xy$ (rechts).

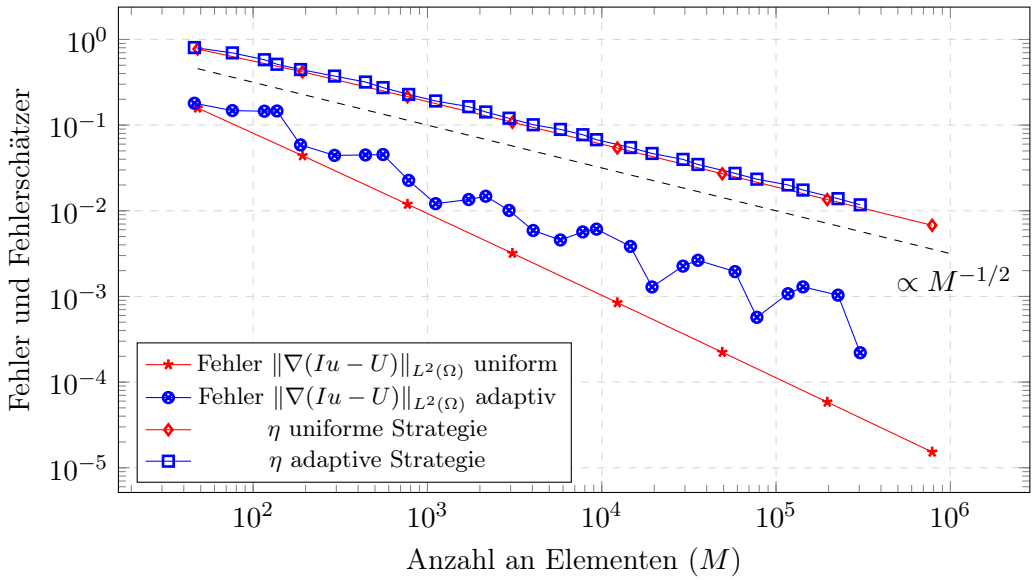


Abbildung 9.8: Der Galerkin Fehler (mit Iu als nodale Interpolation von u) und der residuale Fehlerschätzer η des dritten Beispiel in Abhängigkeit zur Anzahl an Elementen. Wir betrachten sowohl uniforme Netzverfeinerung als auch den adaptiven Algorithmus aus Python-Code 8.1.

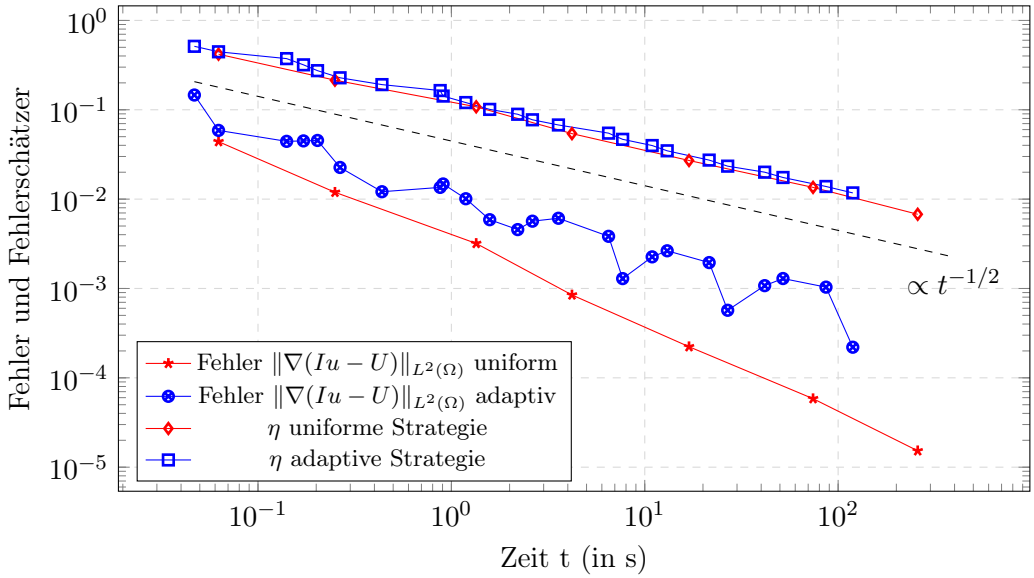


Abbildung 9.9: Der Galerkin Fehler (mit Iu als nodale Interpolation von u) und der residuale Fehlerschätzer η des dritten Beispiel in Abhängigkeit zur gebrauchten Zeit. Wir betrachten sowohl uniforme Netzverfeinerung als auch den adaptiven Algorithmus aus Python-Code 8.1.

9.4 Viertes Beispiel

Als letztes Beispiel betrachten wir noch den Fall, wenn wir die Lösung nicht vorgegeben haben. Wir wählen:

$$\begin{aligned} \Omega &:= (-1,1)^2 \setminus ([0,1] \times [-1,0]) \\ \Gamma_D &:= \{(x,y) \in \partial\Omega : (x = 0 \wedge y \in (-1,0)) \vee (x \in (0,1) \wedge y = 0)\} \end{aligned} \tag{42}$$

mit demselben Angfangsgitter wie in Abbildung 9.7 und betrachten

$$\begin{aligned} -\operatorname{div}(\mathcal{A}\nabla u) + b \cdot \nabla u + cu = 1 = f \quad &\text{in } \Omega \\ u = 0 \quad &\text{auf } \Gamma_D \\ (\mathcal{A}\nabla u) \cdot u = 0 \quad &\text{auf } \partial\Omega \setminus \Gamma_D \end{aligned} \tag{43}$$

mit $\mathcal{A} = I, b = (1,1)^T$ und $c = 1$. Hier sehen wir in Abbildung 9.10 die Rate $\Theta(M^{-1/2})$ für die adaptiven (mit $\theta = 0.25$) und die Rate $\Theta(M^{-1/3})$ für die uniforme Strategie. Für dieses Beispiel sehen wir in Abbildung 9.12 einige adaptiv verfeinerte Gitter und in Abbildung 9.13 die Ausgabe des Python-Codes 9.1, in der wir eine diskrete Lösung U auf einem uniformen Gitter visualisieren.

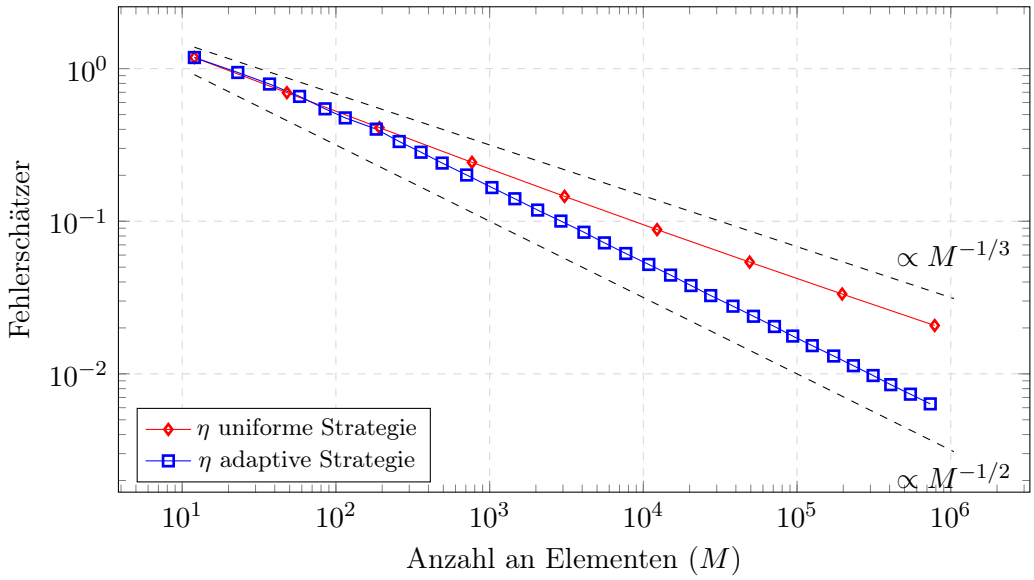


Abbildung 9.10: Der residualen Fehlerschätzer η des vierten Beispiel in Abhängigkeit zur Anzahl an Elementen. Wir betrachten sowohl uniforme Netzverfeinerung als auch den adaptiven Algorithmus aus Python-Code 8.1.

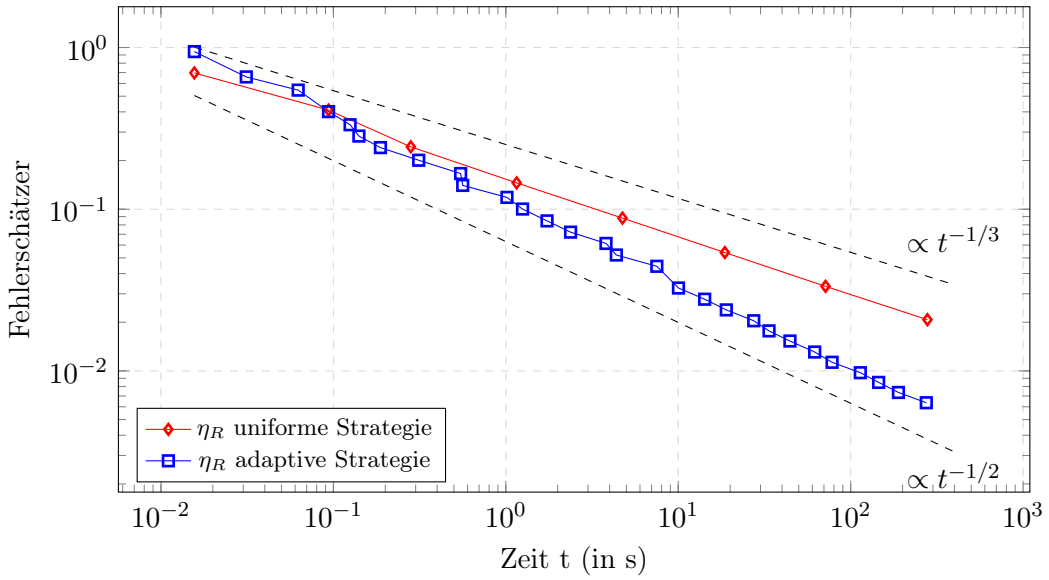


Abbildung 9.11: Der residualen Fehlerschätzer η des vierten Beispiel in Abhängigkeit zur gebrauchten Zeit. Wir betrachten sowohl uniforme Netzverfeinerung als auch den adaptiven Algorithmus aus Python-Code 8.1.

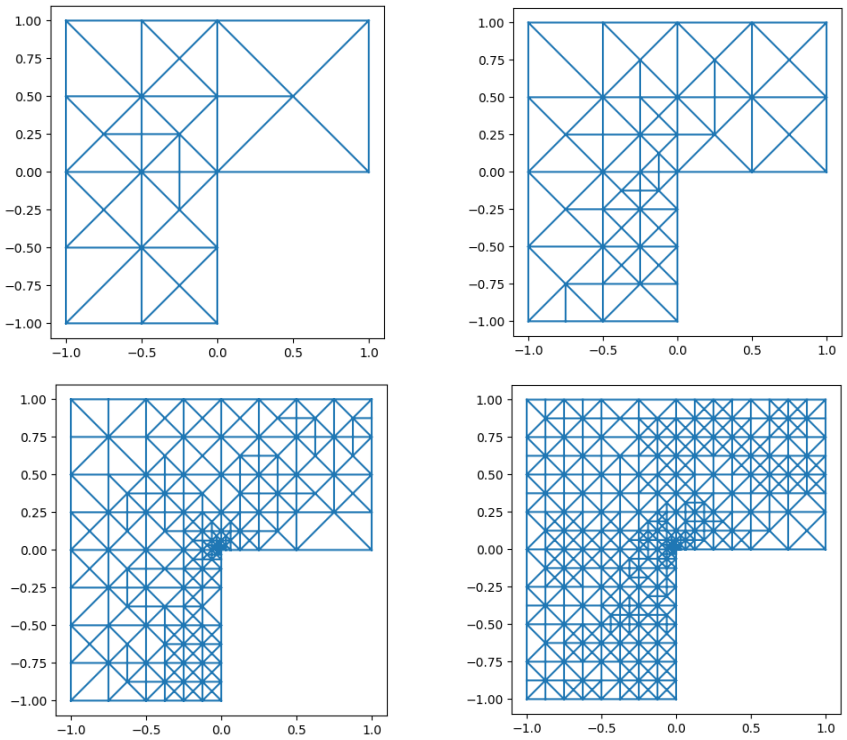


Abbildung 9.12: Einige adaptive Gitter von Beispiel 4. Wir sehen $\#\mathcal{T}_3$=37 (links oben), $\#\mathcal{T}_5 = 85$ (rechts oben), $\#\mathcal{T}_9 = 358$ (links unten) und $\#\mathcal{T}_{11} = 709$ (rechts unten).

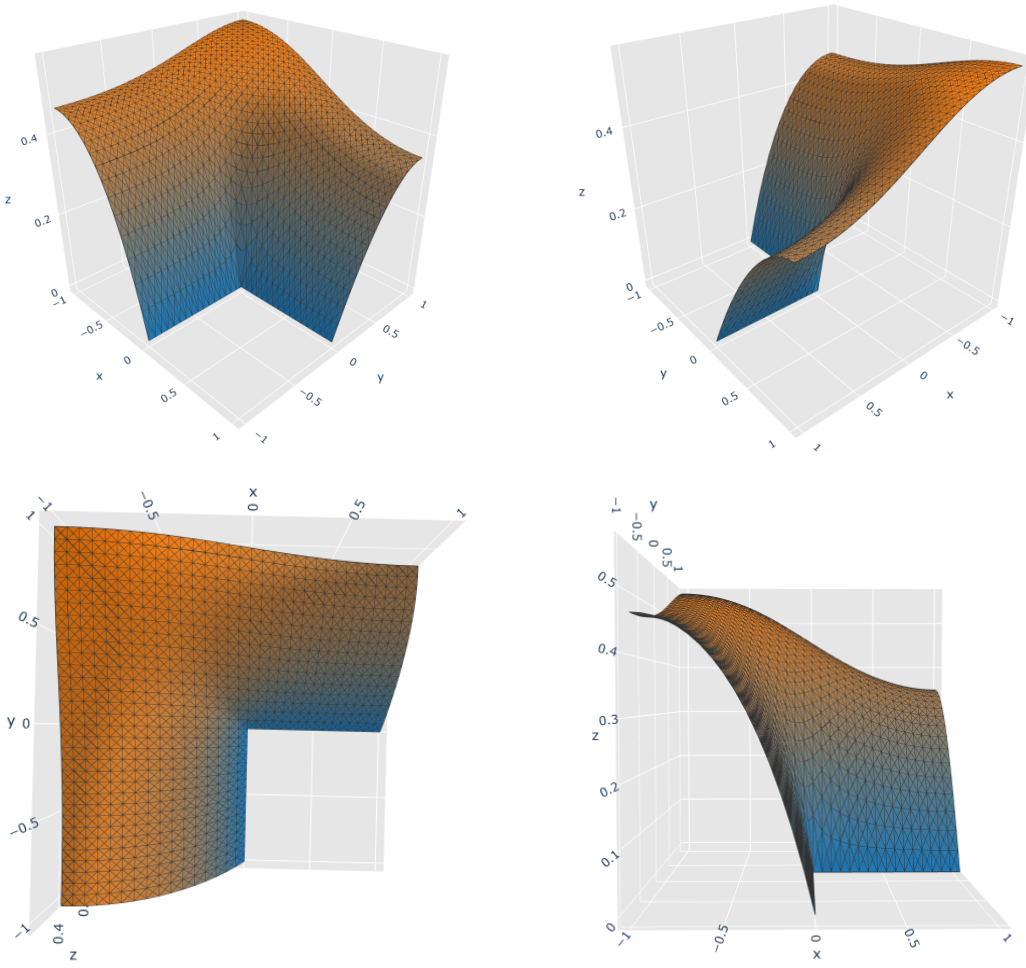


Abbildung 9.13: Die Lösung u aus Beispiel 4.

Literaturverzeichnis

[1] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and van der Vorst, H. (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics.

[2] Jüngel, A. (2022). Partielle Differentialgleichungen, Vorlesungsskript, TU Wien. https://www.asc.tuwien.ac.at/~juengel/scripts/PDE.pdf. [Zugriff: 20-September-2022].

[3] Michael Feischl and Dirk Praetorius (2020). Numerics of Partial Differential Equations: Stationary Problems, Vorlesungsskript, TU Wien. https://www.asc.tuwien.ac.at/feischl/downloads/FEM_VO.pdf. [Zugriff: 20-September-2022].

[4] Stefan Funken, Dirk Praetorius, Philipp Wissgott (2011). Efficient Implementation of Adaptive P1-FEM in Matlab. https://www.asc.tuwien.ac.at/~praetorius/matlab/p1afem/p1afem.pdf. [Zugriff: 20-September-2022].