Numerische Strömungsmechanik
Sommersemester 2016

Hausaufgabe

Name:	Roland Zimmermann
Matrikelnummer:	21426901
Mail Adresse:	roland.zimmermann@stud.uni-goettingen.de

Abgabetermin: 05.08.2016

Inhaltsverzeichnis

1. Einleitung

2. Fragestellung

In dieser Hausaufgabe wird ein zweidimensionales quadratisches System betrachtet. In diesem soll die Wärmeausbreitung (siehe 3.1) beschrieben werden, welche durch ein extern erzeugtes Geschwindigkeitsfeld (durch einen Ventilator) hervorgerufen wird. Dieses Vektorfeld soll nicht zeitabhängig sein, und lässt sich in dimensionsloser Form durch

$$\vec{v} = (\pi \sin(2\pi x)\cos(\pi y), -2\pi \cos(2\pi x)\sin(\pi y))^t$$

beschreiben. Hierbei gibt es für jeden der vier Ränder je eine Randbedingung. So soll die Temperatur auf dem oberen Rand $T = 1$ betragen, auf dem unteren dagegen $T = 0$. An dem linken und rechten Rand soll dagegen bloß die jeweilige Normalenableitung verschwinden. Dies wird durch eine wärmeisoliernde Wand in dem System bedingt. Zu Beginn soll eine Temperaturverteilung der Form $T_{start} = y$ vorliegen.

Zur Untersuchung des System fallen mehrere Aufgaben an. So wird zuerst das Geschwindigkeitsfeld selber untersucht (siehe 4.1). Anschließend wird eine erste numerische Lösung des Problems (im *FTCS-Schema*) durchgeführt (siehe 4.2). Daraufhin wird nun die numerische Korrektheit dieser Lösung untersucht (siehe 4.3). Schließlich werden für verschiedene Parameter Simulationen durchgeführt, um unter anderem die Stabilität zu beurteilen (siehe 4.4, 4.5). Schließlich wird das Problem noch einmal gelöst, diesmal aber im *BTCS-Schema* (siehe 4.6).

3. Theorie

3.1. Die Diffusions-Advektion-Gleichung

Die Ausbreitung von Flüssigkeiten, Gasen beziehungsweise von Energie kann zum einen durch die *Diffusionsgleichung*

$$\frac{\partial}{\partial t}T = D\nabla^2 T \tag{1}$$

und zum anderen durch die *Advektionsgleichung*

$$\frac{\partial}{\partial t}T = -\nabla \cdot (\vec{v} \cdot T) \tag{2}$$

beschrieben werden. Hierbei stehen T für die Verteilung der Energie (Temperatur), t für die Zeit und D für die Diffusionskonstante. Während die *Diffusionsgleichung* die zeitliche Ausbreitung der Partikel beziehungsweise der Energie, bedingt durch zufällige

Bewegungen, beschreibt, beschreibt die *Advektionsgleichung* den Partikelfluss der durch ein Geschwindigkeitsfeld herbeigeführt wird.
Fasst man diese beiden Effekte zusammen, so erhält man die *Diffusions-Advektion-Gleichung*

$$\frac{\partial}{\partial t}T = D\nabla^2 T - \nabla \cdot (\vec{v} \cdot T) \tag{3}$$

Hierbei kann nun, wenn ein gegebenen externes, divergenzfreies Geschwindigkeitsfeld gegeben ist, dieses aus der Divergenz herausgelöst werden, sodass sich

$$\frac{\partial}{\partial t}T + \vec{v} \cdot \nabla T = D\nabla^2 T \tag{4}$$

ergibt.
Durch das Einführen der *Péclet-Zahl*

$$Pe = \frac{Lv\rho c_p}{\lambda}$$

ist eine Entdimensionalisierung der Gleichung (4) möglich. Hierbei stehen L für eine charakteristische Länge, v für eine Geschwindigkeit, rho für die Dichte, c_p für die spezifische Wärmekapazität und λ für die Wärmeleitfähigkeit des jeweiligen Mediums. Somit ergibt sich

$$\frac{\partial}{\partial t}T + Pe \cdot \vec{v} \cdot \nabla T = \nabla^2 T \tag{5}$$

3.2. Das FTCS-Schema

Eine mögliche Methode zur Lösung einer solchen partiellen Differenzialgleichung ist durch die Verfahrensgruppe der finiten Differenzen gegeben. Eine hierfür geeignete Diskretiserung des Problems ist in Form eines *FTCS*-Schemas (Forward in Time, Centered in Space) gegeben. Hierbei wird der Raum in der x-Richtung in $(N_x + 1)$- und in der y-Richtung in $(N_y + 1)$ Stützstellen unterteilt, sodass sich ein räumliches Gitter ergibt. Zwischen diesen Punkten herrscht ein jeweiliger Abstand von $\Delta x = X/N_x$ und $\Delta y = Y/N_y$, wobei X und Y für die Ausmaße des zu beschreibenden Problems in der x- und y-Richtung sind.
Die Temperatur $T(x, y)$, die hiermit beschrieben werden soll, geht somit in die Form $T_{i,j}$ über, wobei i der Index des Punktes in der x- und j der in der y-Richtung ist. Diese Indizes laufen von 0 bis N_x beziehungsweise N_y.
Um nun noch die Zeit zu erfassen, wird auch diese in Punkte diskretisiert, die einen Abstand Δ_t haben. Sodass sich insgesamt ein dreidimensionaler Würfel ergibt, dessen verschiedenen Ebenen für die verschiedenen Zeitpunkte stehen. Erneut geht die Temperatur $T(x, y) = T_{i,j}$ über in $T^N_{i,j}$, wobei N für den zeitlichen Index steht. Auch hier wird

von 0 an indiziert.

Diese Aufteilung erlaubt es nun, die in den Gleichung (5) vorkommenden Ableitungen zu beschreiben. Hierbei wird in dem *FTCS*-Schema die zeitliche Ableitung in der ersten und die räumlichen in der zweiten Ordnung beschrieben. Es ergibt sich für die Ableitungen

$$\frac{\partial}{\partial t}T(x,y) = \frac{T_{i,j}^{N+1} - T_{i,j}^{N}}{\Delta t}, \tag{6}$$

$$\frac{\partial}{\partial x}T(x,y) = \frac{T_{i+1,j}^{N} - T_{i-1,j}^{N}}{2\Delta x}, \tag{7}$$

$$\frac{\partial^2}{\partial x^2}T(x,y) = \frac{T_{i+1,j}^{N} - 2T_{i,j}^{N} + T_{i-1,j}^{N}}{\Delta x^2}. \tag{8}$$

Durch Einsetzen in die Gleichung (5) und Umstellen nach $T_{i,j}^{N+1}$ erhält man schließlich in der zweiten räumlichen Ordnung

$$\begin{aligned} T_{i,j}^{N+1} =& T_{i,j}^{N} + \Delta t \left(-Pe \left(v_{i,j}^{x} \frac{T_{i+1,j}^{N} - T_{i-1,j}^{N}}{2\Delta x} + v_{i,j}^{y} \frac{T_{i,j+1}^{N} - T_{i,j-1}^{N}}{2\Delta x} \right) \right. \\ & \left. + \frac{T_{i+1,j}^{N} - 2T_{i,j}^{N} + T_{i-1,j}^{N}}{\Delta x^2} + \frac{T_{i,j+1}^{N} - 2T_{i,j}^{N} + T_{i,j-1}^{N}}{\Delta y^2} \right) \end{aligned} \tag{9}$$

womit die Integration durchgeführt werden kann. Die Ausdrücke v^x und v^y stehen für die jeweilige x- und y-Komponente des zeitlich konstanten Geschwindigkeitsfeldes an dem jeweiligen Ort.
Des Weiteren sind allerdings noch die Randbedingungen zu beachten. Bei *Dirichlet*-Rändern muss an dem Schema nichts geändert werden. Bei *Neumann*-Rändern dagegen schon. Exemplarisch wird hierfür als Beispiels der linke Rand betrachtet, auf dem die erste Ableitung in x-Richtung verschwinden soll. Durch eine Taylor-Entwicklung und einen Koeffizientenvergleich erhält man die diskretisierte Randbedingung zweiter Ordnung

$$T_{i,j}^{N+1} - \frac{4}{3}T_{i+1,j}^{N+1} + \frac{1}{3}T_{i+2,j}^{N+1} = 0.$$

3.3. Das BTCS-Schema

Ein alternativer Ansatz zur Lösung ist das *BTCS*-Schema (Backward in Time, Centered in Space), bei der räumlich erneut eine Diskretisierung zweiter Ordnung durchgeführt wird, diese allerdings zum Zeitpunkt $(N+1)$ ausgewertet wird. Zeitlich wird auch wieder ein Verfahren erster Ordnung verwendet. Somit ergibt sich der Ausdruck

$$T_{i,j}^{N} = T_{i,j}^{N+1}\alpha + \beta_{i-1,j}^{-}T_{i-1,j}^{N+1} + \beta_{i+1,j}^{+}T_{i+1,j}^{N+1} + \gamma_{i,j-1}^{-}T_{i,j-1}^{N+1} + \gamma_{i,j+1}^{+}T_{i,j+1}^{N+1}. \tag{10}$$

Dabei wurden die folgenden Abkürzungen eingeführt und benutzt

$$\alpha = \Delta t \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta t} \right), \tag{11}$$

$$\beta_{i,j}^{\pm} = \Delta t \left(-\frac{1}{\Delta x^2} \pm \frac{Pev_{i,j}^{x}}{2\Delta x} \right), \tag{12}$$

$$\gamma_{i,j}^{\pm} = \Delta t \left(-\frac{1}{\Delta y^2} \pm \frac{Pev_{i,j}^{y}}{2\Delta y} \right). \tag{13}$$

Um das Problem nun besser beschreiben zu können, wird das zweidimensionale Feld in einen eindimensionalen Spaltenvektor zusammengefasst. Hierfür ist es notwendig, die Indizes umzubenennen. Hierbei wird ab jetzt die folgende Konvention benutzt: durchlaufe erst alle x-Werte zu einem festen y-Wert, und erhöhe das y. Dies bewirkt, dass die Zeilen des Feldes hintereinander geschrieben werden:

$$T_{i,j}^{N} = T_{i+N_y j}^{N}.$$

Nun kann die Gleichung (10) als eine Matrixoperation der Form

$$\vec{T}^{N} = M \cdot \vec{T}^{N+1},$$

mit der Matrix M ausgedrückt werden. Diese hat die fünf Diagonalen, die ungleich null sind in ihrem Hauptteil, und zwei weitere Submatrizen, die fast nur eine Hauptdiagonale haben, die ungleich null ist. Somit schreibe M nun als Blockmatrix

$$\boldsymbol{M} = \begin{pmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & C \end{pmatrix},$$

mit der Matrix A

$$\boldsymbol{A} = \begin{pmatrix} 2 & -4/3 & 1/3 & . & 1/3 & -4/3 & 2 & -4/3 & 1/3 & . \\ 0 & 1 & 0 & . & & & & & & . \\ 0 & 0 & 1 & 0 & . & & & & & . \\ . & . & . & . & . & . & & & & . \\ . & & . & . & . & . & . & & & . \\ . & & & . & . & . & . & . & . & . \\ . & & & & . & . & . & . & . & . \\ . & & & & & . & . & . & . & . \\ . & & & & & & . & 0 & 1 & 0 \\ 0 & . & . & . & . & . & . & . & 0 & 1 \end{pmatrix},$$

welche die Randbedingungen am oberen, linken und rechten Rand umsetzt; und der Matrix C

$$\boldsymbol{C} = \begin{pmatrix} 1 & 0 & . & . & 0 \\ 0 & 1 & 0 & . & . \\ . & . & . & . & . \\ . & & 0 & 1 & 0 \\ 0 & . & . & 0 & 1 \end{pmatrix},$$

welche die Randbedingungen am unteren Rand umsetzt. Die Matrix B

$$\boldsymbol{B} = \begin{pmatrix} \gamma^- & . & \beta^- & \alpha & \beta^+ & . & \gamma^+ & 0 & . & . & 0 \\ 0 & \gamma^- & . & \beta^- & \alpha & \beta^+ & . & \gamma^+ & 0 & . & 0 \\ 0 & 0 & \gamma^- & . & \beta^- & \alpha & \beta^+ & . & \gamma^+ & . & 0 \\ . & & . & . & . & . & . & . & . & . & . \\ . & & & . & . & . & . & . & . & . & . \\ . & & & & . & . & . & . & . & . & . \\ . & & & & & . & . & . & . & . & . \\ . & & & & & & . & . & . & . & . \\ . & & & & & & & . & . & . & . \\ 0 & . & 0 & \gamma^- & . & \beta^- & \alpha & \beta^+ & . & \gamma^+ & . \\ 0 & . & . & 0 & \gamma^- & . & \beta^- & \alpha & \beta^+ & . & \gamma^+ \end{pmatrix},$$

setzt die implizite Gleichung (10) um. Um nun einen Zeitschritt vorzugehen, muss also nur noch die Matrix M invertiert werden, beziehungsweise das Gleichungssystem $M\vec{x} = \vec{b}$ gelöst werden.
Hierfür hat sich das *SOR-Verfahren*, welches eine Modifikation des *Gauß-Seidel-Verfahrens* ist, als besonders geeignet herausgestellt. Dies sind beides iterative Lösungsverfahren. Um diese zu benutzen, wird die Matrix zuerst in drei Teilmatrizen $\boldsymbol{M} = \boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U}$ zerlegt, wobei D eine Diagonal-, L eine untere Dreiecks- und U eine obere Dreiecksmatrix ist. Zudem wird der *Residuen-Vektor* $\vec{r_n} = \boldsymbol{M}\vec{x_n} - \vec{b}$ eingeführt, wobei der Index den jeweiligen Iterationsschritt angibt. Dieser Vektor beschreibt die Differenz zwischen der korrekten und der iterativen Lösung der Gleichung $Mx = b$.
Durch diese Zerlegung ist es möglich eine rekursive Bestimmungsformel [2, S. 25ff]

$$\vec{x}^n = \vec{x}^{n-1} - \alpha(\boldsymbol{L} + \boldsymbol{D})^{-1}\vec{r}^{n-1}$$

aufzustellen. Hierbei ist α der so genannte *Relaxationsparameter*. Bei dem *Gauss-Seidel-Verfahren* ist er gleich null.
Diese Matrizen-Gleichung kann in Summenschreibweise durch Eigenschaften triagonaler Matrizen aus der linearen Algebra auch als

$$x_i^{n+1} = (1-\alpha)x_i^n + \frac{\alpha}{m_{i,i}}\left(b_i - \sum_{j=1}^{i-1} m_{i,j}x_j^{n+1} - \sum_{j=i}^{N} m_{i,j}x_j^n\right) \tag{14}$$

aufgefasst werden [2, S.30], wobei N die Dimensionen des Problems angibt.

4. Durchführung der Aufgaben

4.1. Aufgabe 1: Untersuchung von $\vec{v}$

Für die numerische Betrachtung des Problems spielt der Einfluss von $\vec{v}$ eine entscheidende Rolle. Ist dieses Geschwindigkeitsfeld nämlich divergenzfrei, so ergibt sich eine mögliche, vereinfachende Betrachtung in diskretisierter Form (siehe). Für das externe Feld

$$\vec{v} = (\pi \sin(2\pi x)\cos(\pi y), -2\pi \cos(2\pi x)\sin(\pi y))^t$$

gilt

$$\nabla \cdot \vec{v} = \frac{\partial}{\partial x}(\pi \sin(2\pi x)\cos(\pi y) + \frac{\partial}{\partial y}(-2\pi \cos(2\pi x)\sin(\pi y)) = 0;$$

es ist somit divergenzfrei. Dies kann man auch grafisch nachvollziehen.

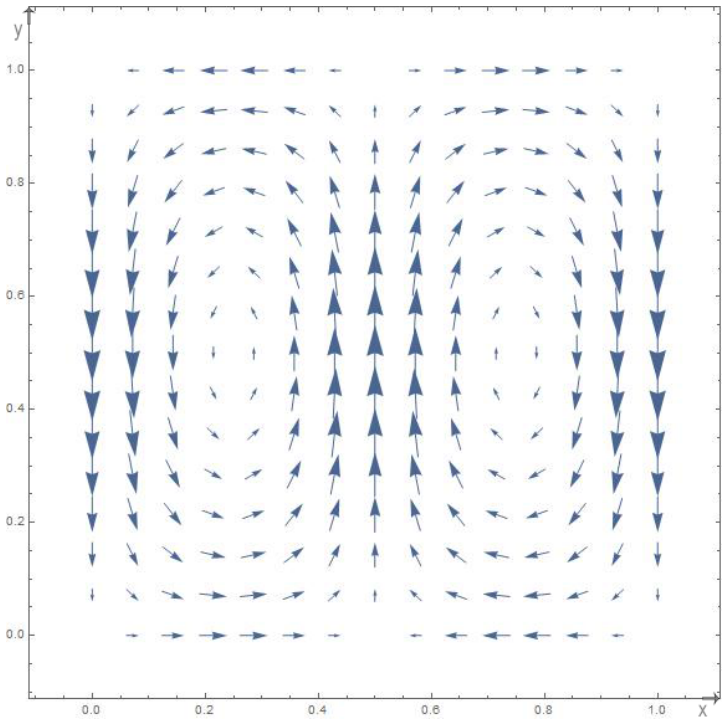

Abbildung 1: Grafische Darstellung des Geschwindigkeitsfeldes $\vec{v}$ in Abhängigkeit von x und y.

4.2. Aufgabe 2: FTCS-Lösung

4.3. Aufgabe 3: Numerische Korrektheit der FTCS-Lösung

Zur Beurteilung der Korrektheit der numerischen Lösung wird normalerweise erstmal versucht, das Ergebnis mit der analytischen Lösung zu vergleichen. Da dieses Problem allerdings nicht analytisch lösbar ist, muss hier anders verfahren werden. Als Ansatz hierbei wird eine mögliche stationäre Lösung geraten, und die Differenzialgleichung um einen Quellterm so ergänzt, dass dies die exakte Lösung des Problems ist.
Für diese konstruierte stationäre Lösung wird der Ansatz

$$T^* = \cos(\pi x)\sin(\pi y) + y$$

gewählt, da dies sowohl die Startverteilung umsetzt, als auch die vier Randbedingungen erfüllt. Um den Quellterm Q zu korrigieren, wird T^* in die stationäre Gleichung

$$Pe \cdot \vec{v}\nabla T^* = \nabla^2 T^* + Q$$

eingesetzt, welche auch Quellen berücksichtigt. Es ergibt sich somit

$$Q = Pe \cdot \vec{v}\nabla T^* - \nabla^2 T^*,$$

was sich nach Einsetzen des Ansatzes und des gegebenen Geschwindigkeitsfeldes zu

$$Q = \pi^2 \left(2\cos(\pi x)\sin(\pi y) - Pe \cdot \cos(\pi x)^3 \sin(2\pi y)\right) - 2Pe \cdot \pi \cos(2\pi x)\sin(\pi y)$$

ergibt.
Berücksichtigt man diesen Quellterm in dem *FTCS-Schema*, folgt:

$$\begin{aligned} T_{i,j}^{N+1} =& T_{i,j}^N + \Delta t \left(-Pe \left(v_{i,j}^x \frac{T_{i+1,j}^N - T_{i-1,j}^N}{2\Delta x} + v_{i,j}^y \frac{T_{i,j+1}^N - T_{i,j-1}^N}{2\Delta x} \right) \right. \\ & \left. + \frac{T_{i+1,j}^N - 2T_{i,j}^N + T_{i-1,j}^N}{\Delta x^2} + \frac{T_{i,j+1}^N - 2T_{i,j}^N + T_{i,j-1}^N}{\Delta y^2} + Q_{i,j} \right) \end{aligned}$$

Dies führt zu Änderungen des Quellcodes im Vergleich zur vorherige Aufgabe in der Methode ... in den Zeilen ...-... . Nun kann zu jedem Zeitpunkt die Differenz zwischen der stationären Lösung $T_{i,j}^*$ und der numerischen Lösung $T_{i,j}^N$ berechnet werden als

$$\sigma^N = \sqrt{(\Sigma_{i,j}(T_{i,j}^N - T_{i,j}^*)^2)}.$$

Diese Abweichung wird nun alle 100 Zeitschritte ausgegeben. Dies führt zu Veränderungen in der Methode ... in Zeile ... bis ... .
Setzt man nun noch $N_x = N_y$, und variiert diese Werte für ein festes Δt, so können

Aussagen getroffen werden, über die Abweichungen der stationären numerischen T und der tatsächlichen stationären Lösung T^* in Abhängigkeit von der Gittergröße.

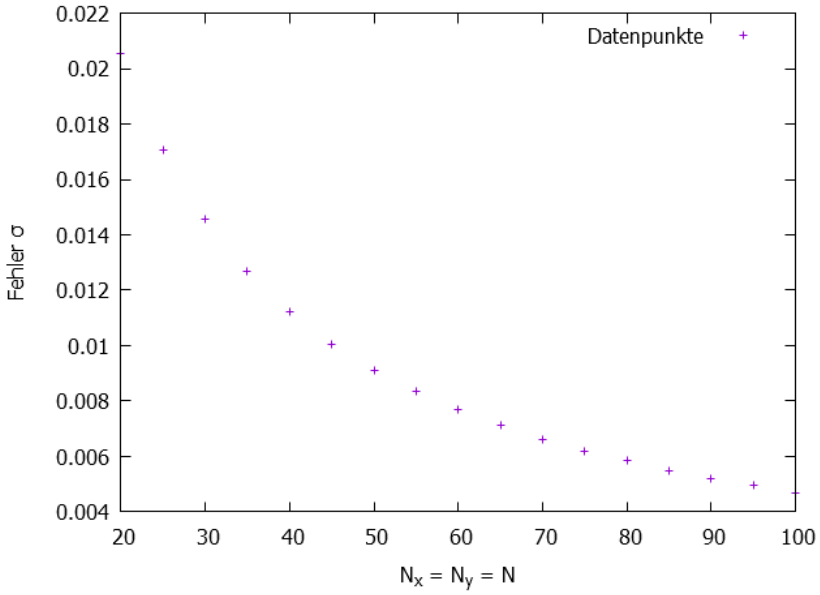

Abbildung 2: Grafische Auftragung der numerischen Abweichung σ in Abhängigkeit von der Gittergröße N.

In Abbildung (2) ist der Verlauf des Fehlers σ in Abhängigkeit von $N_x = N_y$ für $Pe = 2$ für einen Zeitschritt von $\Delta t = 2 \cdot 10^{-5}$ aufgetragen. Zudem wurde eine eine Regression durchgeführt, welche eine Abhängigkeit der Form $\sigma = N^{-1.2}$ ergeben hat. Dazu ist noch anzumerken, dass für größere N-Werte (größer als 115) der Fehler nicht mehr weiter sinkt, sondern sogar stark anwächst, da die numerische Lösung divergent wird.

4.4. Aufgabe 4: Auswertung verschiedener FTCS-Lösungen

Im weiteren Verlauf wird der zuvor eingeführte Quellterm $Q = 0$ gesetzt. Nun kann das Temperaturfeld T für zwei verschiedene *Péclet-Zahlen* $Pe = 2$ und $Pe = 10$ simuliert, und zu den drei Zeitpunkten $t = 0.005$, $t = 0.05$ und $t = 0.5$ visualisiert werden. Hierfür wird $\Delta t = 2 \cdot 10^{-}4$ und $N = N_x = N_y = 30$ angenommen. Die Darstellungen sind in den Abbildungen (3) und (4) zu finden.

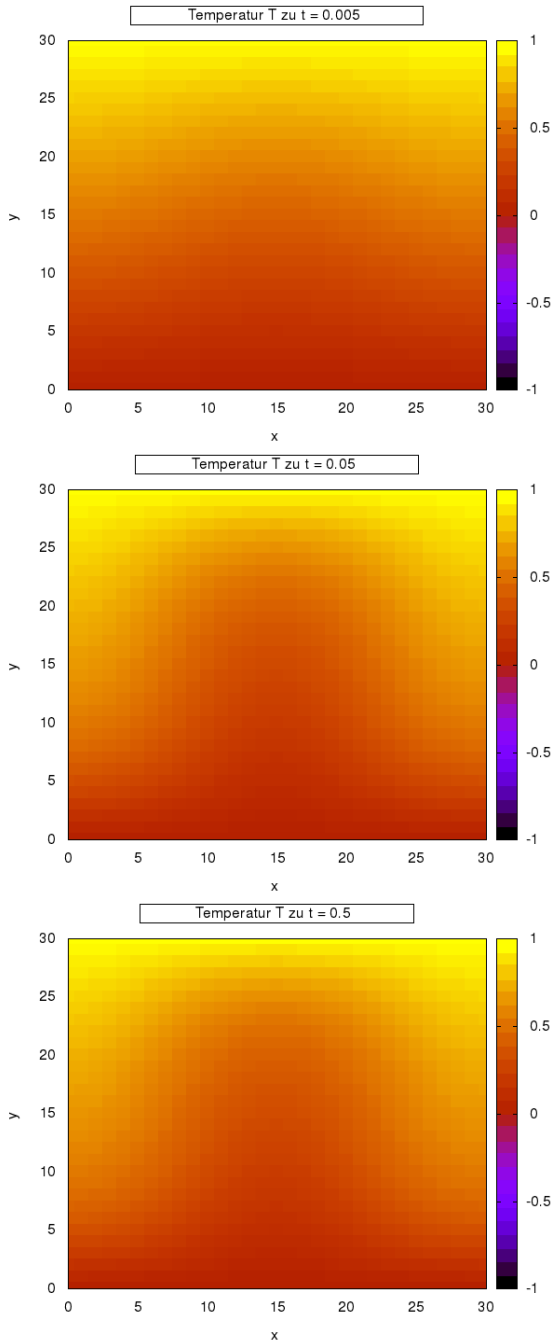

Abbildung 3: Temperatur für $Pe = 2$.

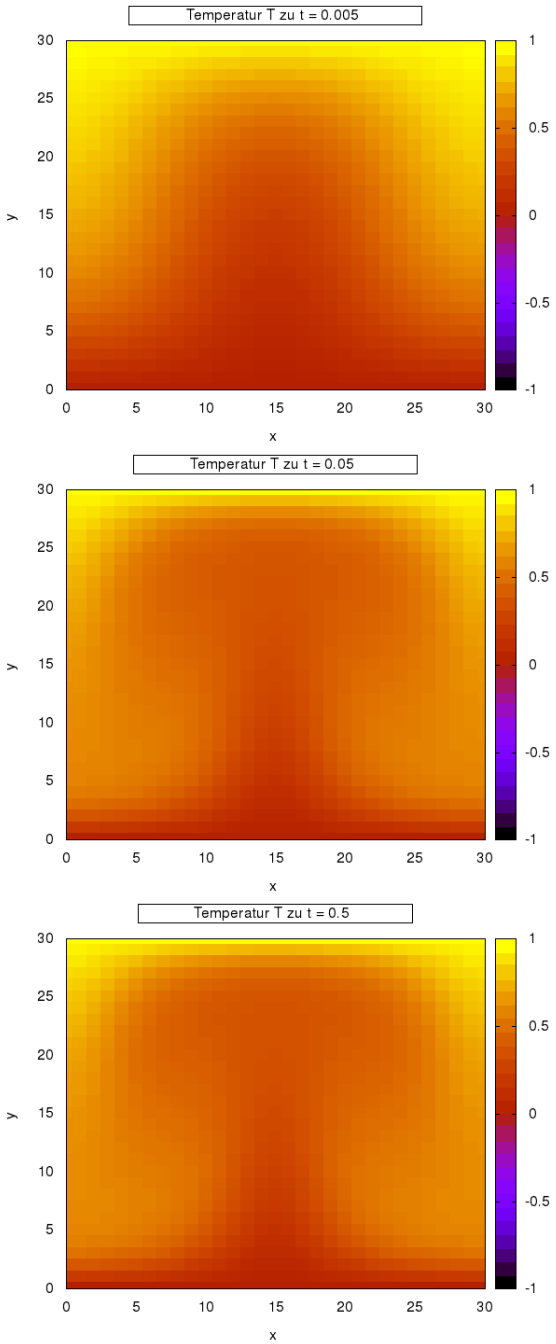

Abbildung 4: Temperatur für $Pe = 10$.

4.5. Aufgabe 5: Beurteilung der Stabilität der FTCS-Lösung

Um weitere Aussagen über die numerische Lösung treffen zu können, wird nun noch die Stabilität betrachtet. Hierfür wird zuerst für $N = N_x = N_y = 30$ die maximale Zeitschrittgröße Δt_{max}^{Pe} für die *Péclet-Zahlen* $Pe \in \{0.1, 1.0, 10\}$ bestimmt.
Diese ergeben sich zu

$$\begin{aligned}\Delta t_{max}^{0.1} &= 2.785 \cdot 10^{-4} 2.788,\\ \Delta t_{max}^{1.0} &= 2.787 \cdot 10^{-4} 2.789,\\ \Delta t_{max}^{10} &= 2.795 \cdot 10^{-4}.\end{aligned}$$

Dies liefert einen ersten Eindruck: Die Stabilitätsgrenze hängt von Pe ab, und nimmt für steigende Pe in gewissen, geringen Maßen zu.
Für eine genauere Aussage wird nun eine *von-Neumann-Stabilitätsanalyse* durchgeführt. Hierfür wird für die Lösung ein Ansatz der generellen Form

$$T_{i,j}^N = \xi^N e^{i(k_x \Delta x \cdot i + k_y \Delta y \cdot j)}$$

gewählt. Durch Einsetzen in Gleichung (9) und Kürzen ergibt sich der Ausdruck

$$\begin{aligned}\xi = 1 - \frac{\Delta t Pe v^x}{2\Delta x}\left(e^{ik_x\Delta x} - e^{-ik_x\Delta x}\right) - \frac{\Delta t Pe v^y}{2\Delta y}\left(e^{ik_y\Delta y} - e^{-ik_y\Delta y}\right) - \frac{2\Delta t}{\Delta x^2} - \frac{2\Delta t}{\Delta y^2}\\ + \frac{\Delta t}{\Delta x^2}\left(e^{ik_x\Delta x} + e^{-ik_x\Delta x}\right) + \frac{\Delta t}{\Delta y^2}\left(e^{ik_y\Delta y} + e^{-ik_y\Delta y}\right),\end{aligned}$$

welcher durch trigonometrische Identitäten in

$$\begin{aligned}\xi = 1 - i\Delta t Pe\left(\frac{v^x}{\Delta x}\sin(k_x\Delta x) + \frac{v^y}{\Delta y}\sin(k_y\Delta y)\right) + \frac{2\Delta t}{\Delta x^2}\cos(k_x\Delta x) + \frac{2\Delta t}{\Delta y^2}\cos(k_y\Delta y)\\ - \frac{2\Delta t}{\Delta x^2} - \frac{2\Delta t}{\Delta y^2}\end{aligned}$$

übergeht. Damit das Verfahren konvergiert, muss ξ kleiner 1 ein. Nähert man hier den Sinus und den Cosinus mit den Maximalwerten, und schätzt die Komponenten v^x und v^y nach oben durch π und 2π ab, so ergibt sich

$$\lambda^2\left(64 + \frac{Pe 2\pi^2}{\Delta x^2}\right) - 16\lambda < 0, \tag{15}$$

mit der *CFL-Zahl* $\lambda = \Delta t/\Delta x^2$. Eine Lösung dieser Ungleichung ergibt folgenden Bereich, sodass das *FTCS-Schema* konvergiert:

$$0 < \lambda = \frac{\Delta t}{\Delta x^2} < \frac{8\left(\frac{Pe}{\Delta x}\right)^2}{32\left(\frac{Pe}{\Delta x}\right)^2 + \pi^2}. \tag{16}$$

Dies ist durch die Abschätzungen nach oben bedingt, eine obere Schranke. Liegt λ außerhalb des Bereiches, aber nahe an der Grenze, so ist ein Konvergieren nicht sicher ausgeschlossen. Liegt λ allerdings in dem Bereich, so konvergiert das Verfahren sicher. Dieses Verhalten konnte an den zuvor empirisch ermittelten Grenzen für Δt verifiziert werden.

4.6. Aufgabe 6: BTCS-Lösung

Nachdem zuvor die explizite Diskretisierung untersucht wurde, wird das Problem nun implizit im *BTCS-Schema* betrachtet (siehe 3.3). Hierfür wird zuerst die zuvor beschriebene Matrix $\mathbf{M}$ implementiert. Dies wird in (...) durchgeführt. Nun muss nur noch in jedem Integrationsschritt diese Matrix invertiert werden. Dafür wurde das *SOR-Verfahren* in (...) nach Gleichung (14) implementiert. Dabei werden in jedem einzelnen Interationsschritt die Residuen $\vec{r} = \mathbf{M}\vec{x} - \vec{b}$ für die Lösung von $\mathbf{M}\vec{x} = \vec{b}$ berechnet, und gefordert, dass der Betrag dieser kleiner als $|\vec{r}| < 10^{-4}$ ist. Dies wird als Abbruchbedingung im dem *SOR-Verfahren* genutzt. Um die Vorteile des *SOR* wirklich nutzen zu können, muss zuerst ein geeigneter Relaxationsparameter α ermittelt werden. Hierzu wird $Pe = 10$, $N_x = N_y = 30$ und $\Delta t = 100$ gesetzt. Nun wird α mit einer Schrittweite von 0.05 ab 1.0 variiert und die benötigten Iterationen ermittelt.

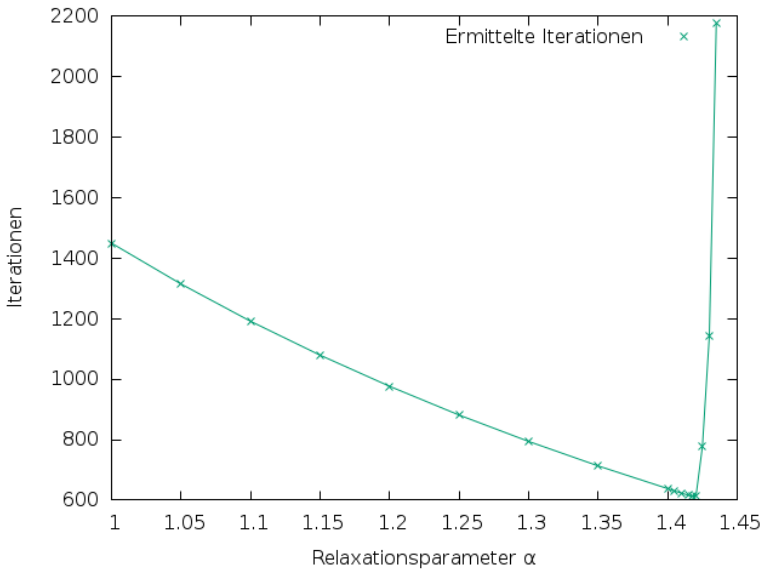

Abbildung 5: Grafische Darstellung der benötigten Iterationen zur Konvergenz in Abhängigkeit vom Relaxationsparameter α.

Die grafische Auftragung der so ermittelten Ergebnisse ist in Abbildung (5) zu finden. Es ist gut erkennbar, dass das bei $\alpha = 1.41$ die geringste Anzahl an Iterationen benötigt wurde. Somit wird im weiteren Verlauf dieser Parameter genutzt.

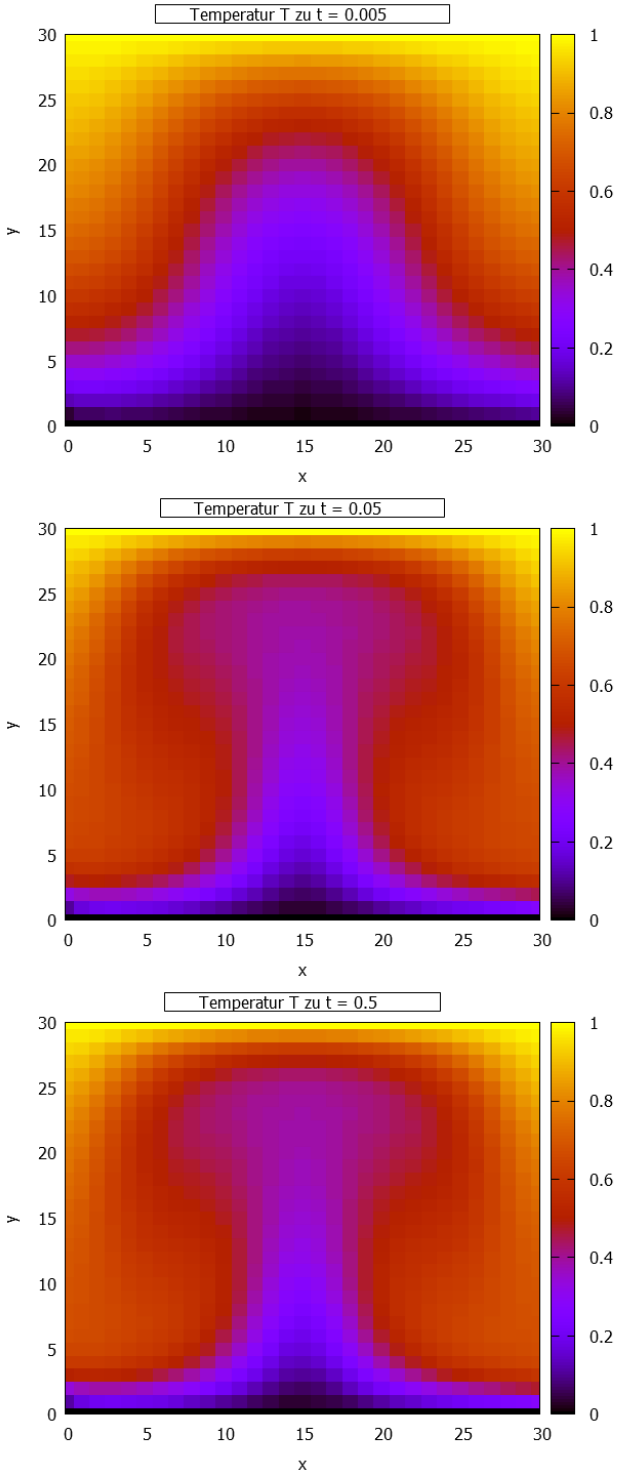

Abbildung 6: Temperatur nach dem *BTCS-Schema* für $Pe = 10$ und $\Delta t = 2 \cdot 10^{-4}$.

Schließlich wird das zuvor implementierte und getestete Verfahren noch einmal für die Parameterwahl aus Aufgabenpunkt (4.4) angewendet, um die Ergebnisse miteinander zu vergleichen. Hierfür wurde also erneut $Pe = 10$, $\Delta t = 2 \cdot 10^{-4}$ und $t \in \{0.005, 0.05, 0.5\}$ gewählt. Die hierdurch erhaltenen Temperaturverteilungen sind erneut als Heatmap in Abbildung (6) dargestellt.

5. Auswertung der Ergebnisse

5.1. Numerische Korrektheit der FTCS-Lösung

Durch den Vergleich in Abschnitt (4.3) ist erkennbar, dass der numerische Fehler für größere Gitter deutlich geringer wird. Dieser Abfall hat allerdings auch eine bestimmte Grenze. Überschreitet $N = N_x = N_y$ diesen Wert, so kann die numerische Lösung nicht mehr konvergieren. Den Grund für dieses Verhalten (Abweichung der *CFL-Zahl*) ist in Abschnitt (4.5) beschrieben und erklärt. Anzumerken ist noch, dass der Fehler noch rapider gefallen wäre, wenn man den Gesamtfehler durch die Anzahl der zu berechnenden Gitterpunkte zwecks einer Normalisierung geteilt hätte.

Literatur

[1] Andres Honecker, Thomas Pruschke, Salvatore R. Manmana,
Scriptum zur Vorlesung: Computergestütztes wissenschaftliches Rechnen.
Göttingen, Sommersemester 2016

[2] W. Auzinger,
Lecture Notes Iterative Solution of Large Linear Systems.
Wien, 2011

A. Quellcode

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cstdlib>
#include <iomanip>
#include <cmath>

using namespace std;

//custom type which discribes a 2d grid. use this instead of arrays for access violation safety
typedef vector<vector<double > > grid_t;


//integrates the ODE with the FTCS scheme. expects the parameters:
//  values: a grid_t object which contains current distribution of the temperature T
//  v0x, v0y: a grid_t object which contains the x/y component of the velocity v0
//  delta_t, delta_x, delta_y, pe: the time-step-size, the step size in x/y direction and the Péclet number
void ftcs_time_step(grid_t& values, grid_t v0x, grid_t v0y, const double delta_t, const double delta_x, const double delta_y, const double pe)
{
  //get the dimensions of the grid
  size_t dimension_x = values.size();
  size_t dimension_y = values.at(0).size();

  //create copy of the grid with the same dimensions
  //this will be used for the calculation of the derivatives
  grid_t old_values;
  for (size_t x=0; x<dimension_x; x++)
  {
    //add a new row
    old_values.push_back(vector<double>());
    for (size_t y=0; y<dimension_y; y++)
    {
      //copy value
      old_values.at(x).push_back(values.at(x).at(y));
    }
  }
```

```
37
38  //loop over the entire inner grid (ignore the 4 outer sides of
       the rectangle) and calculate the new value using the
       FTCS-scheme
39  for (size_t x=1; x<dimension_x-1; x++)
40  {
41    for (size_t y=1; y<dimension_y-1; y++)
42    {
43      values.at(x).at(y) += delta_t*
44                  (
45                    1.0/pow(delta_x,2)*(old_values.at(x+1).at(y) -
                         2.0*old_values.at(x).at(y) +
                         old_values.at(x-1).at(y))
46                  + 1.0/pow(delta_y,2)*(old_values.at(x).at(y+1) -
                      2.0*old_values.at(x).at(y) +
                      old_values.at(x).at(y-1))
47                    -pe*
48                      (
49                        v0x.at(x).at(y)/(2.0*delta_x)*(old_values.at(x+1).at(y)-ol
50                      +
                            v0y.at(x).at(y)/(2.0*delta_y)*(old_values.at(x).at(y+1)-o
51                      )
52                  );
53    }
54  }
55
56  //now take care of the four boundary conditions
57
58  //left/right boundaries (Neumann boundaries)
59  for (size_t y=0; y<dimension_y; y++)
60  {
61    values.at(0).at(y) = 1.0/3.0*(4.0*values.at(0+1).at(y) -
         values.at(0+2).at(y));
62    values.at(dimension_x-1).at(y) =
         -2.0/3.0*(-2.0*values.at(dimension_x-2).at(y) +
         1.0/2.0*values.at(dimension_x-3).at(y));
63  }
64
65  //bottom/top boundaries (Dirichlet boundaries)
66  for (size_t x=0; x<dimension_x; x++)
67  {
68    values.at(x).at(0) = 0.0;
69    values.at(x).at(dimension_y-1) = 1.0;
70  }
```

```
71
72 }
73
74 int main(int argc, char* argv[])
75 {
76   //print information for the usage
77   cout << "Use this program to calculate the temperature with the
       FTCS-scheme in the rectangle." << endl;
78   cout << "Expects the follwing set of parameters:" << endl;
79
80   cout << "\toutput file:\tThe path to the file in which the
       results will be stored." << endl;
81   cout << "\tN_x:\t\tThe amount of grid points in the x-direction."
       << endl;
82   cout << "\tN_y\t\tThe amount of grid points in the y-direction."
       << endl;
83   cout << "\tPe:\t\tThe Péclet-Number" << endl;
84   cout << "\tMax t\t\tThe maximum time until the system will be
       simulated." << endl;
85   cout << "\tDelta t\t\tThe time step size for the integration." <<
       endl;
86
87   //read entered parameters
88
89   //create output stream
90   ofstream outputFile;
91   outputFile.open(argv[1]);
92   outputFile << fixed << setprecision(5);
93
94   //the amount of grid points in the x direction
95   const size_t dimension_x = (atoi)(argv[2])+1;
96   //the distance between to grid points
97   const double delta_x = 1.0/(dimension_x-1);
98
99   //the amount of grid points in the x direction
100  const size_t dimension_y = (atoi)(argv[3])+1;
101  //the distance between to grid points
102  const double delta_y = 1.0/(dimension_y-1);
103
104  //Péclet-number
105  const double pe = atof(argv[4]);
106  //maximum time until which the system will be simulated
107  const double max_t = atof(argv[5]);
108  //the time step size
```

```
109   const double delta_t = atof(argv[6]);
110
111   cout << "Entered parameters:" << endl;
112   cout << "\tNx:" << dimension_x << "\tNy:" << dimension_y <<
          "\tPe:" << pe << "\tMax t" << max_t << "\tDelta t:" << delta_t
          << endl;
113
114   //to store the x component of the velocity v0
115   grid_t v0x;
116   //to store the y component of the velocity v0
117   grid_t v0y;
118   //to store the current temperature
119   grid_t values;
120
121   //create the new grids
122   for (size_t x=0; x<dimension_x; x++)
123   {
124     //add a new row
125     v0x.push_back(vector<double>());
126     v0y.push_back(vector<double>());
127     values.push_back(vector<double>());
128
129     for (size_t y=0; y<dimension_y; y++)
130     {
131       //add the v0 values
132       v0x.at(x).push_back(M_PI*sin(2*M_PI*x*delta_x)*cos(M_PI*y*delta_y));
133       v0y.at(x).push_back(-2.0*M_PI*cos(2*M_PI*x*delta_x)*sin(M_PI*y*delta_y));
134
135       //add the start temperature of the rectangle
136       values.at(x).push_back(y*delta_y);
137     }
138   }
139
140   //loop over the time, until we have reached the maximum time
141   for (double t=0.0; t<max_t; t+=delta_t)
142   {
143     ftcs_time_step(values, v0x, v0y, delta_t, delta_x, delta_y, pe);
144   }
145
146   //print the final grid's values in the givven output file in a
          matrix form
147   for (size_t y=0; y<dimension_y; y++)
148   {
149     for (size_t x=0; x<dimension_x; x++)
```

```
150     {
151       outputFile << values.at(x).at(y) << "\t";
152     }
153     outputFile << "\n";
154   }
155
156   return 1;
157 }
```

source/task02.cpp

```
1 #include <iostream>
2 #include <fstream>
3 #include <vector>
4 #include <cstdlib>
5 #include <iomanip>
6 #include <cmath>
7
8 using namespace std;
9
10 //custom type which discribes a 2d grid. use this instead of arrays
     for access violation safety
11 typedef vector<vector<double >> grid_t;
12
13
14 //integrates the ODE with the FTCS scheme and returns the error,
     compared with the stationary solution. expects the parameters:
15 //  values: a grid_t object which contains current distribution of
     the temperature T
16 //  v0x, v0y: a grid_t object which contains the x/y component of
     the velocity v0
17 //  source: a grid_t object which contains the source values
18 //  delta_t, delta_x, delta_y, pe: the time-step-size, the step
     size in x/y direction and the Péclet number
19 double ftcs_time_step(grid_t& values, grid_t v0x, grid_t v0y,
     grid_t source, const double delta_t, const double delta_x, const
     double delta_y, const double pe)
20 {
21   //get the dimensions of the grid
22   size_t dimension_x = values.size();
23   size_t dimension_y = values.at(0).size();
24
25   //create copy of the grid with the same dimensions
26   //this will be used for the calculation of the derivatives
27   grid_t old_values;
```

```
28
29  for (size_t x=0; x<dimension_x; x++)
30  {
31    //add a new row
32    old_values.push_back(vector<double>());
33    for (size_t y=0; y<dimension_y; y++)
34    {
35      //copy value
36      old_values.at(x).push_back(values.at(x).at(y));
37    }
38  }
39
40  //loop over the entire inner grid (ignore the 4 outer sides of
       the rectangle) and calculate the new value using the
       FTCS-scheme
41  for (size_t x=1; x<dimension_x-1; x++)
42  {
43    for (size_t y=1; y<dimension_y-1; y++)
44    {
45      values.at(x).at(y) += delta_t*
46                (
47                  1.0/pow(delta_x,2)*(old_values.at(x+1).at(y) -
                        2.0*old_values.at(x).at(y) +
                        old_values.at(x-1).at(y))
48                + 1.0/pow(delta_y,2)*(old_values.at(x).at(y+1) -
                     2.0*old_values.at(x).at(y) +
                     old_values.at(x).at(y-1))
49                  -pe*
50                    (
51                      v0x.at(x).at(y)/(2.0*delta_x)*(old_values.at(x+1).at(y)-ol
52                    +
                        v0y.at(x).at(y)/(2.0*delta_y)*(old_values.at(x).at(y+1)-o
53                    )
54                    + source.at(x).at(y)
55                );
56    }
57  }
58
59  //now take care of the four boundary conditions
60
61  //left/right boundaries (Neumann boundaries)
62  for (size_t y=0; y<dimension_y; y++)
63  {
```

```
64      values.at(0).at(y) = 1.0/3.0*(4.0*values.at(0+1).at(y) -
            values.at(0+2).at(y));
65      values.at(dimension_x-1).at(y) =
            -2.0/3.0*(-2.0*values.at(dimension_x-2).at(y) +
            1.0/2.0*values.at(dimension_x-3).at(y));
66    }
67
68    //bottom/top boundaries (Dirichlet boundaries)
69    for (size_t x=0; x<dimension_x; x++)
70    {
71      values.at(x).at(0) = 0.0;
72      values.at(x).at(dimension_y-1) = 1.0;
73    }
74
75    //now calculate the error, compared to the stationary solution
76    //loop over all elements and compare them
77    double error = 0.0;
78    for (size_t x=0; x<dimension_x; x++)
79    {
80      for (size_t y=0; y<dimension_y; y++)
81      {
82        error += pow(
83                    values.at(x).at(y) -
                        (cos(M_PI*delta_x*x)*sin(M_PI*delta_y*y)+y*delta_y)
84                  ,2);
85      }
86    }
87
88    return sqrt(error);
89  }
90
91  int main(int argc, char* argv[])
92  {
93    //print information for the usage
94    cout << "Use this program to calculate the temperature with the
          FTCS-scheme in the rectangle with a source, and to compare the
          stationary solution with the numerical." << endl;
95    cout << "Expects the follwing set of parameters:" << endl;
96
97    cout << "\toutput file:\tThe path to the file in which the
          results will be stored." << endl;
98    cout << "\tN_x:\t\tThe amount of grid points in the x-direction."
          << endl;
```

```
99  cout << "\tN_y\t\tThe amount of grid points in the y-direction."
       << endl;
100 cout << "\tPe:\t\tThe Péclet-Number" << endl;
101 cout << "\tMax t\t\tThe maximum time until the system will be
       simulated." << endl;
102 cout << "\tDelta t\t\tThe time step size for the integration." <<
       endl;
103
104 //read entered parameters
105
106 //create output stream
107 ofstream outputFile;
108 outputFile.open(argv[1]);
109 outputFile << fixed << setprecision(5);
110
111 //the amount of grid points in the x direction
112 const size_t dimension_x = (atoi)(argv[2])+1;
113 //the distance between to grid points
114 const double delta_x = 1.0/(dimension_x-1);
115
116 //the amount of grid points in the x direction
117 const size_t dimension_y = (atoi)(argv[3])+1;
118 //the distance between to grid points
119 const double delta_y = 1.0/(dimension_y-1);
120
121 //Péclet-number
122 const double pe = atof(argv[4]);
123 //maximum time until which the system will be simulated
124 const double max_t = atof(argv[5]);
125 //the time step size
126 const double delta_t = atof(argv[6]);
127
128 cout << "Entered parameters:" << endl;
129 cout << "\tNx: " << dimension_x << "\tNy: " << dimension_y <<
       "\tPe: " << pe << "\tMax t: " << max_t << "\tDelta t: " <<
       delta_t << endl;
130
131 //to store the x component of the velocity v0
132 grid_t v0x;
133 //to store the y component of the velocity v0
134 grid_t v0y;
135 //to store the current temperature
136 grid_t values;
137 //to store the source of the temperature
```

```
138   grid_t sources;
139
140   //create the new grids
141   for (size_t x=0; x<dimension_x; x++)
142   {
143     //add a new row
144     v0x.push_back(vector<double>());
145     v0y.push_back(vector<double>());
146     values.push_back(vector<double>());
147     sources.push_back(vector<double>());
148
149     for (size_t y=0; y<dimension_y; y++)
150     {
151       //add the v0 values
152       v0x.at(x).push_back(M_PI*sin(2*M_PI*x*delta_x)*cos(M_PI*y*delta_y));
153       v0y.at(x).push_back(-2.0*M_PI*cos(2*M_PI*x*delta_x)*sin(M_PI*y*delta_y));
154
155       //add the start temperature of the rectangle
156       values.at(x).push_back(y*delta_y);
157
158       //add the source of the temperature of the rectangle
159       sources.at(x).push_back(
160         pow(M_PI,2)*(cos(M_PI*delta_x*x)*sin(M_PI*delta_y*y)*2
161         - pe*pow(cos(x*delta_x*M_PI),3)*sin(2*M_PI*delta_y*y))
162         -pe*2*M_PI*cos(2*M_PI*delta_x*x)*sin(M_PI*y*delta_y)
163       );
164     }
165   }
166
167   //to print just every 100th error comparison
168   size_t iteration = 0;
169   //loop over the time, until we have reached the maximum time
170   for (double t=0.0; t<max_t; t+=delta_t)
171   {
172     double err = ftcs_time_step(values, v0x, v0y, sources, delta_t,
          delta_x, delta_y, pe);
173
174     //print error value
175     if (iteration++ % 100 == 0)
176     {
177         cout << "time:\t" << t << "\terror:\t" << err << "\n" <<
              flush;
178         iteration = 0;
179     }
```

```
180   }
181
182   //print the final grid's values in the givven output file in a
          matrix form
183   for (size_t y=0; y<dimension_y; y++)
184   {
185     for (size_t x=0; x<dimension_x; x++)
186     {
187       outputFile << values.at(x).at(y) << "\t";
188     }
189     outputFile << "\n";
190   }
191
192   return 1;
193 }
```

source/task03.cpp

```
1 #include <iostream>
2 #include <fstream>
3 #include <vector>
4 #include <cstdlib>
5 #include <iomanip>
6 #include <cmath>
7 #include "../../../lib/linear_system.h"
8 #include "../../../lib/square_matrix.h"
9
10 using namespace std;
11 using namespace numerical;
12
13 //custom type which discribes a 2d grid. use this instead of arrays
       for access violation safety
14 typedef vector<vector<double > > grid_t;
15
16 /*calculates the variable beta. expects the parameters:
17   plus: indicates whether beta plus or beta minus will be calculated
18   delta_t, delta_x, delta_y, pe: the time-step-size, the step size
         in x/y direction and the Péclet number
19   x,y: the x/y coordinate of the point on which beta is desired
20 */
21 double beta(bool plus, double delta_t, double delta_x, double
       delta_y, double pe, double x, double y)
22 {
23   if (plus)
24   {
```

```
25      return delta_t*(-1.0 / pow(delta_x, 2) + (pe*(M_PI*sin(2 *
            M_PI*x*delta_x)*cos(M_PI*y*delta_y))) / (2.0*delta_x));
26    }
27    else
28    {
29      return delta_t*(-1.0 / pow(delta_x, 2) - (pe*(M_PI*sin(2 *
            M_PI*x*delta_x)*cos(M_PI*y*delta_y))) / (2.0*delta_x));
30    }
31  }
32
33  /*calculates the variable ganna. expects the parameters:
34    plus: indicates whether gamma plus or gamma minus will be
          calculated
35    delta_t, delta_x, delta_y, pe: the time-step-size, the step size
          in x/y direction and the Péclet number
36    x,y: the x/y coordinate of the point on which gamma is desired
37  */
38  double gamma(bool plus, double delta_t, double delta_x, double
        delta_y, double pe, double x, double y)
39  {
40    if (plus)
41    {
42      return delta_t*(-1.0 / pow(delta_y, 2) + (pe*(-2.0*M_PI*cos(2 *
            M_PI*x*delta_x)*sin(M_PI*y*delta_y))) / (2.0*delta_y));
43    }
44    else
45    {
46      return delta_t*(-1.0 / pow(delta_y, 2) - (pe*(-2.0*M_PI*cos(2 *
            M_PI*x*delta_x)*sin(M_PI*y*delta_y))) / (2.0*delta_y));
47    }
48  }
49
50  int main(int argc, char* argv[])
51  {
52    //print information for the usage
53    cout << "Use this program to calculate the temperature with the
          BTCS-scheme in the rectangle." << endl;
54    cout << "Expects the follwing set of parameters:" << endl;
55
56    cout << "\toutput file:\tThe path to the file in which the
          results will be stored." << endl;
57    cout << "\tN_x:\t\t.The amount of grid points in the
          x-direction." << endl;
```

```
58  cout << "\tN_y\t\tThe amount of grid points in the y-direction."
       << endl;
59  cout << "\tPe:\t\tThe Péclet-Number" << endl;
60  cout << "\tMax t\t\tThe maximum time until the system will be
       simulated." << endl;
61  cout << "\tDelta t\t\tThe time step size for the integration." <<
       endl;
62
63  //read entered parameters
64
65  //create output stream
66  ofstream outputFile;
67  outputFile.open(argv[1]);
68  outputFile << fixed << setprecision(5);
69
70  //the amount of grid points in the x direction
71  const size_t dimension_x = (atoi)(argv[2])+1;
72  //the distance between to grid points
73  const double delta_x = 1.0/(dimension_x-1);
74
75  //the amount of grid points in the x direction
76  const size_t dimension_y = (atoi)(argv[3])+1;
77  //the distance between to grid points
78  const double delta_y = 1.0/(dimension_y-1);
79
80  //Péclet-number
81  const double pe = atof(argv[4]);
82  //maximum time until which the system will be simulated
83  const double max_t = atof(argv[5]);
84  //the time step size
85  const double delta_t = atof(argv[6]);
86
87  cout << "Entered parameters:" << endl;
88  cout << "\tNx: " << dimension_x << "\tNy: " << dimension_y <<
       "\tPe: " << pe << "\tMax t: " << max_t << "\tDelta t: " <<
       delta_t << endl;
89
90  //the length of the value vector
91  size_t n = dimension_x*dimension_y;
92
93  //create a new square matrix (nXn) to store the matrix M
94  square_matrix coeff_matrix(n, 0);
95
96  //create const value for the variable alpha
```

```
97  const double alpha = delta_t * (2.0 / pow(delta_x, 2) + 2.0 /
        pow(delta_y, 2)) + 1.0;
98
99  //now set the values of the three matrices
100
101 //Matrix A
102 //main diagonal for the neuman boundaries
103 for (size_t i = 0; i<dimension_x; i++)
104 {
105   coeff_matrix.set_value(i, i, 1.0);
106 }
107
108 //first row (for the dirichlet boundaries)
109 coeff_matrix.set_value(0, 0, 2.0);
110 coeff_matrix.set_value(0, 1, -4.0 / 3.0);
111 coeff_matrix.set_value(0, 2, 1.0 / 3.0);
112
113 for (size_t i = dimension_x; i<n; i += dimension_x)
114 {
115   coeff_matrix.set_value(0, i - 2, 1.0 / 3.0);
116   coeff_matrix.set_value(0, i - 1, -4.0 / 3.0);
117   coeff_matrix.set_value(0, i, 2.0);
118   coeff_matrix.set_value(0, i + 1, -4.0 / 3.0);
119   coeff_matrix.set_value(0, i + 2, 1.0 / 3.0);
120 }
121
122 coeff_matrix.set_value(0, dimension_x - 1, -4.0 / 3.0);
123 coeff_matrix.set_value(0, dimension_x - 2, 1.0 / 3.0);
124
125 //Matrix B
126 //contains the biggest part of the BTCS-scheme
127 for (size_t i = dimension_x; i<n - dimension_x; i++)
128 {
129   coeff_matrix.set_value(i, i - dimension_x, gamma(false,
        delta_t, delta_x, delta_y, pe, i % dimension_x,
        i/dimension_x));
130   coeff_matrix.set_value(i, i - 1, beta(false, delta_t, delta_x,
        delta_y, pe, i % dimension_x, i/dimension_x));
131   coeff_matrix.set_value(i, i, alpha);
132   coeff_matrix.set_value(i, i + 1, beta(true, delta_t, delta_x,
        delta_y, pe, i % dimension_x, i/dimension_x));
133   coeff_matrix.set_value(i, i + dimension_x, gamma(true, delta_t,
        delta_x, delta_y, pe, i % dimension_x, i/dimension_x));
134 }
```

```
135
136   //Matrix C
137   //main diagonal for the neuman boundaries
138   for (size_t i = n - dimension_x; i<n; i++)
139   {
140     coeff_matrix.set_value(i, i, 1.0);
141   }
142
143   //create two vectors to store the temperature and to calculate
          the new one
144   vector<double> values;
145   vector<double> old_values;
146
147   //fill them with the starting temperature
148   for (size_t y = 0; y<dimension_y; y++)
149   {
150     for (size_t x = 0; x<dimension_x; x++)
151     {
152       old_values.push_back(y*delta_y);
153       values.push_back(y*delta_y);
154     }
155   }
156
157   //loop over the time, until we have reached the maximum time
158   for (double t = 0.0; t<max_t; t += delta_t)
159   {
160     //invert the matrix M/solve the linear system to get the new
            temperature(value)
161     linear_system_solve_sor(coeff_matrix, values, old_values, 1.44,
            1e-4);
162
163     //copy the new values into the old_values vector for the next
            iteration
164     for (size_t i = 0; i<n; i++)
165     {
166       old_values.at(i) = values.at(i);
167     }
168   }
169
170   //print the final grid's values in the givven output file in a
          matrix form
171   for (size_t i = 0; i<n; i++)
172   {
173     outputFile << values.at(i) << "\t";
```

```
174       if (i > 0 && (i+1) % dimension_x == 0)
175       {
176         outputFile << endl;
177       }
178     }
179
180     return 1;
181 }
```

source/task06.cpp