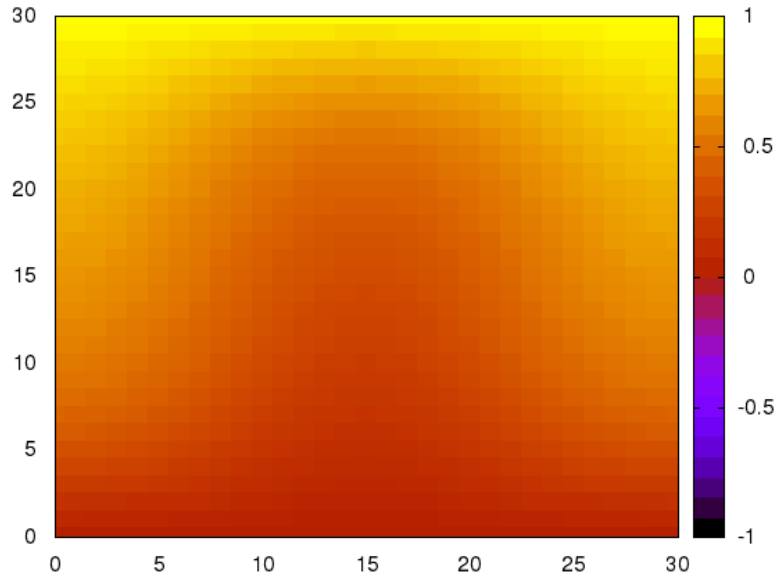
Georg-August-Universität Göttingen
Sommersemester 2016

Hausarbeit zur Vorlesung

Numerische Strömungsmechanik



Name: Roland Zimmermann
Matrikelnummer: 21426901
Mail Adresse: roland.zimmermann@stud.uni-goettingen.de
Abgabetermin: 05.08.2016

Inhaltsverzeichnis

1. Einleitung

Viele Verhaltensweisen und Effekte in der Natur können durch partielle Differenzialgleichungen beschrieben werden. Die Lösung dieser ist allerdings nur in wenigen, speziellen Fällen analytisch möglich. Für die restlichen Fälle müssen numerische Lösungsverfahren verwendet werden. Gerade diese Szenarien spielen beispielsweise in der Auto- und Luftfahrtindustrie eine relevante Rolle bei der Produktentwicklung und Optimierung. In dieser Arbeit werden hierdurch motiviert zwei verschiedene Lösungsverfahren für ein solches Problem implementiert und anschließend analysiert.

2. Fragestellung

In dieser Hausaufgabe wird die Wärmeausbreitung (siehe 3.1) in einem zweidimensionalen, rechteckigen System betrachtet werden. Diese soll durch ein extern erzeugtes Geschwindigkeitsfeld (beispielsweise durch einen Ventilator) verstärkt werden. 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. In dem betrachteten System gibt es für jeden der vier Ränder je eine Randbedingung. So soll die Temperatur auf dem oberen Rand $T = 1$ und auf dem unteren $T = 0$ betragen. An dem linken und rechten Rand soll dagegen die jeweilige Normalenableitung verschwinden. Dies wird durch eine wärmeisoliernde Wand in dem System bedingt. Zu Beginn soll eine Temperaturverteilung der Form $T_{start}(x, y) = y$ 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).
Abschließend wird noch ein kurzer Vergleich der beiden Verfahren hinsichtlich ihrer Resultate und der benötigten Rechenzeit durchgeführt.
Alle hierfür benötigten Programme und Hilfsroutinen werden in *C++* implementiert.

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 aus den Gleichungen (2) und (1) zusammen, so erhält man die *Diffusions-Advektion-Gleichung*

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

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

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

ergibt.
Durch das Einführen der *Péclet-Zahl*

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

ist eine Entdimensionalisierung der Gleichung (3) 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

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

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 zweidimensionales 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 stehen.
Die Temperatur $T(x, y)$, die hiermit beschrieben werden soll, geht dadurch 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 $T(t, x, y) = T_{i,j}(t)$ 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 (4) vorkommenden Ableitungen zu beschreiben. Hierbei wird in dem *FTCS*-Schema die zeitliche Ableitung in der ersten und die räumlichen in der zweiten Ordnung beschrieben. Die Ordnung gibt hierbei jeweils an, wie stark der Fehler von der jeweiligen Schrittweite abhängt. Bei einem Verfahren zweiter Ordnung hängt dieser Fehler von $\mathcal{O}(\Delta x^2)$ ab. Es ergibt sich für die Ableitungen

$$\frac{\partial}{\partial t}T(x,y) = \frac{T^{N+1}_{i,j} - T^N_{i,j}}{\Delta t}, \tag{5}$$

$$\frac{\partial}{\partial x}T(x,y) = \frac{T^N_{i+1,j} - T^N_{i-1,j}}{2\Delta x}, \tag{6}$$

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

Durch Einsetzen in die Gleichung (4) und Umstellen nach $T^{N+1}_{i,j}$ erhält man schließlich

$$T^{N+1}_{i,j} = T^N_{i,j} + \Delta t \left[-Pe \left(v^x_{i,j} \frac{T^N_{i+1,j} - T^N_{i-1,j}}{2\Delta x} + v^y_{i,j} \frac{T^N_{i,j+1} - T^N_{i,j-1}}{2\Delta x} \right) + \frac{T^N_{i+1,j} - 2T^N_{i,j} + T^N_{i-1,j}}{\Delta x^2} + \frac{T^N_{i,j+1} - 2T^N_{i,j} + T^N_{i,j-1}}{\Delta y^2} \right], \tag{8}$$

womit die Integration durchgeführt werden kann. Die Ausdrücke $v^x_{i,j}$ und $v^y_{i,j1}$ 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. Hierbei muss die Differenzialgleichung besonders betrachtet werden, da hier nur Nachbarpunkte in eine Richtung existieren.
Bei den *Dirichlet*-Rändern können die Werte am Rand konstante gehalten werden, sodass

$$T_{Rand}^{N+1} = T_{Rand}^{N} \Rightarrow T_{i,0}^{N+1} = T_{i,0}^{N} = 0 \wedge T_{i,N_y}^{N+1} = T_{i,N_y}^{N} = 1$$

eine geeignete mathematische Beschreibung ist. Exemplarisch für die *Neumann*-Ränder wird 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

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

In diesem Problem muss eine analoge Gleichung auch noch für den rechten Rand betrachtet werden.

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

$$T_{i,j}^{N} = T_{i,j}^{N+1}\alpha + \beta_{i-1,j}^{-}T_{i-1,j}^{N+1} + \beta_{i+1,j}^{+}T_{i+1,j}^{N+1} + \gamma_{i,j-1}^{-}T_{i,j-1}^{N+1} + \gamma_{i,j+1}^{+}T_{i,j+1}^{N+1}. \tag{9}$$

Dabei wurden die folgenden Abkürzungen eingeführt und benutzt

$$\begin{aligned}
\alpha &= \Delta t\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta t}\right),\\
\beta_{i,j}^{\pm} &= \Delta t\left(-\frac{1}{\Delta x^2} \pm \frac{Pev_{i,j}^{x}}{2\Delta x}\right),\\
\gamma_{i,j}^{\pm} &= \Delta t\left(-\frac{1}{\Delta y^2} \pm \frac{Pev_{i,j}^{y}}{2\Delta y}\right).
\end{aligned}$$

Dies gilt allerdings nur für die inneren Punkte des Gitters. Für den linken und rechten Rand gelten, um ebenfalls die Diskretisierung in der zweiten Ordnung umsetzen zu können, die Gleichungen

$$T_{0,j}^{N+1} - \frac{4}{3}T_{1,j}^{N+1} + \frac{1}{3}T_{2,j}^{N+1} = 0, \tag{10}$$

$$T_{N_x,j}^{N+1} - \frac{4}{3}T_{N_x-1,j}^{N+1} + \frac{1}{3}T_{N_x-2,j}^{N+1} = 0. \tag{11}$$

Um das Problem nun besser beschreiben zu können, wird das zweidimensionale Feld in einem 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 den y-Wert. Dies bewirkt, dass die Zeilen des Feldes hintereinander geschrieben werden:

$$T^N_{i,j} = T^N_{i+N_y j}.$$

Nun kann die Gleichung (9) als eine Matrixoperation der Form

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

mit der Matrix $\mathbf{M}$ ausgedrückt werden. Hierbei müssen in $\vec{T^N}$ einige Einträge auf null gesetzt werden, um die Gleichungen (10) und (11) korrekt als Matrix-Gleichung darzustellen. Die Matrix $\mathbf{M}$ hat fünf Diagonalen, die ungleich null sind. $\mathbf{M}$ kann nun als Blockmatrix

$$\boldsymbol{M} = \begin{pmatrix} \mathbb{1} & 0 & 0 & 0 & . & . & . & . & 0 \\ E^- & D & E^+ & 0 & . & . & . & . & 0 \\ 0 & E^- & D & E^+ & 0 & . & . & . & 0 \\ . & & . & . & . & & & & . \\ . & & & . & . & . & & & . \\ . & & & & . & . & . & & . \\ . & & & & & . & . & . & . \\ . & . & . & . & . & 0 & E^- & D & E^+ \\ 0 & . & . & . & . & . & . & . & \mathbb{1} \end{pmatrix},$$

mit den quadratischen Matrizen $\mathbf{E}^{\pm}$, $\mathbf{D}$ und der Einheitsmatrix $\mathbb{1}$ geschrieben werden. Die beiden Einheitsmatrizen im oberen linken und unteren rechten Eck setzen hierbei die *Dirichlet*-Randbedingungen am oberen und rechten Rand um. Die Matrizen $\mathbf{D}$ und $\mathbf{E}^{\pm}$

$$\boldsymbol{E}^{\pm} = \begin{pmatrix} 0 & . & . & . & . & 0 \\ 0 & \gamma^{\pm} & 0 & . & . & 0 \\ 0 & 0 & \gamma^{\pm} & 0 & . & 0 \\ 0 & . & 0 & \gamma^{\pm} & 0 & 0 \\ 0 & . & . & 0 & \gamma^{\pm} & 0 \\ 0 & . & . & . & . & 0 \end{pmatrix}$$

$$\boldsymbol{D} = \begin{pmatrix} 1 & -\frac{4}{3} & \frac{1}{3} & . & . & 0 \\ \beta^- & \alpha & \beta^+ & 0 & & . \\ 0 & \beta^- & \alpha & \beta^+ & 0 & . \\ . & . & \beta^- & \alpha & \beta^+ & 0 \\ . & & 0 & \beta^- & \alpha & \beta^+ \\ 0 & . & . & \frac{1}{3} & -\frac{4}{3} & 1 \end{pmatrix}.$$

setzen die implizite Gleichungen (9) und (10) um, sind also das Kernstück des *BTCS-Schemas*. Ihre erste und letzte Zeile weichen hierbei vom Rest ab, da sie die *Neumann*-Randbedingungen umsetzen. Um das System nun um einen Zeitschritt weiter zu simulieren, muss die Matrix $\mathbf{M}$ invertiert werden - beziehungsweise das Gleichungssystem $\mathbf{M}\vec{x} = \vec{b}$ gelöst werden.
Hierfür hat sich das *SOR-Verfahren* (Successive Over-Relaxation), welches eine Modifikation des *Gauß-Seidel-Verfahrens* ist, als besonders geeignet herausgestellt. Dies sind beides iterative Lösungsverfahren. Um sie verwenden zu können, 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 $\boldsymbol{M}\vec{x} = \vec{b}$.
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*, welcher bei dem *Gauss-Seidel-Verfahren* gleich null ist.
Diese Matrizen-Gleichung kann in Summenschreibweise durch Eigenschaften triagonaler Matrizen aus der linearen Algebra auch als

$$x_i^{n+1} = (1-\alpha)x_i^n + \frac{\alpha}{m_{i,i}}\left(b_i - \sum_{j=1}^{i-1} m_{i,j}x_j^{n+1} - \sum_{j=i}^{N} m_{i,j}x_j^n\right) \tag{12}$$

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 nach Gleichung (8). 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. Es bietet sich zudem an, das Feld grafisch darzustellen. Dies kann in Abbildung (1) nachvollzogen werden.

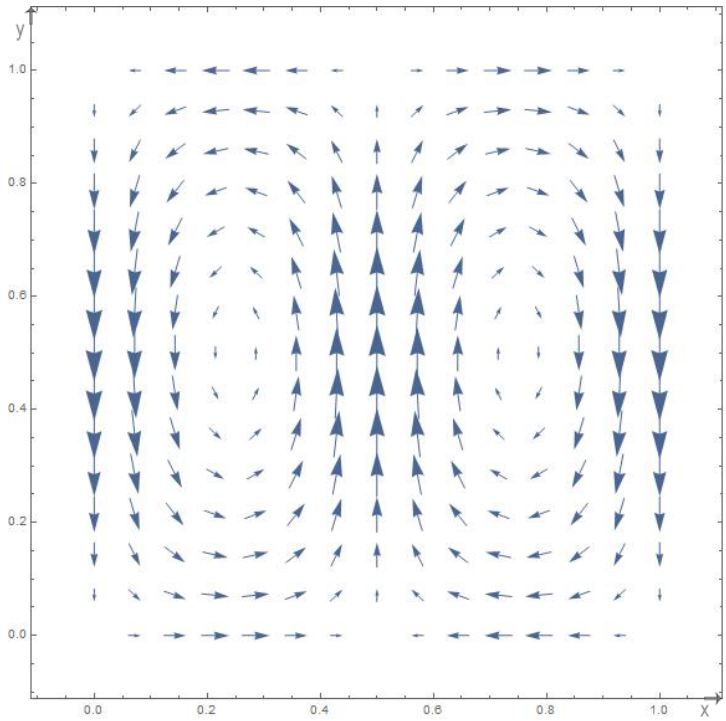


Abbildung 1: Grafische Darstellung des Geschwindigkeitsfeldes $\vec{v}$ in Abhängigkeit von x und y.

Es ist zu erkennen, dass das Feld zwei Zentren hat, um die sich je ein Strudel bildet. Diese laufen in der Mitte zusammen, und strömen zum oberen Rand hin.

4.2. Aufgabe 2: FTCS-Lösung

Um nun eine erste numerische Lösung erzeugen zu können, wird das zuvor in (3.2) hergeleitete Verfahren benutzt. Hierfür wird direkt eine etwas allgemeinere Implementation gewählt, welche in der Integration auch möglich Quellterme berücksichtigt. Dieser Ansatz wird gewählt, da im weiteren Verlauf dieser Arbeit (siehe 4.3) die numerische Korrektheit des Integrators überprüft werden soll. Hierfür wird eine Quelle hinzugefügt, welche eine analytische Lösung erlaubt.
Das Programm besteht aus zwei verschiedenen Routinen. Die *main*-Methode liest gewünschte Eingangsparameter, wie die Schrittweite Δt, die maximale Integrationszeit t_{max}, die *Péclet-Zahl* Pe, die Gitterauflösung N_x und N_y und einen weiteren Parameter, der angibt, ob ein Quellterm erzeugt werden soll, ein (Zeile 78-119). Zudem erzeugt sie Objekte des Types *vector<vector<double>>*, in welchem der Temperaturverlauf in dem Rechteck, und ein etwaiger Quellterm abgespeichert werden. Hierfür wird ein *vector* statt eines

Feldes genutzt, um mögliche *Acess Violation Exceptions* zu vermeiden und um Memoryleaks zu reduzieren (Zeile 121-163). Etwaige Leistungsverluste hierdurch sind zum einen begrenzt und steuern keinen großen Anteil zu der Gesamtlaufzeit bei.
In einer Schleife wird in den gewählten Zeitschritten bis zur Endzeit iteriert, und für jede Zeit ein Integrationsschritt durchgeführt (Zeile 131-163). Zuletzt wird die berechnete Temperaturverteilung in die vorher angegebene Datei ausgegeben (Zeile 199-209).
Der jeweilige Integrationsschritt wird von der *ftcs_time_step*-Methode ausgeführt. Diese legt zuerst eine Kopie des übermittelten Feldes an, um auf Grundlage dieser, die Ableitungsterme berechnen zu können. Hierfür wird in einer Schleife über das gesamte Gitter iteriert, und jeweils der $\nabla^2 T$ und der $\vec{v_0} \cdot \nabla T$ Term einzeln berechnet und anschließend nach Gleichung (13) als Änderung von T verrechnet (Zeile 19-59). Abschließend werden noch die *Dirichlet*- und *Neumann*-Ränder beachtet und die Werte passend aktualisiert (Zeile 61-76).

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 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:

$$T_{i,j}^{N+1} = T_{i,j}^N + \Delta t \left[-Pe \left(v_{i,j}^x \frac{T_{i+1,j}^N - T_{i-1,j}^N}{2\Delta x} + v_{i,j}^y \frac{T_{i,j+1}^N - T_{i,j-1}^N}{2\Delta x} \right) + \frac{T_{i+1,j}^N - 2T_{i,j}^N + T_{i-1,j}^N}{\Delta x^2} + \frac{T_{i,j+1}^N - 2T_{i,j}^N + T_{i,j-1}^N}{\Delta y^2} + Q_{i,j} \right], \tag{13}$$

Die Berechnung und Zuweisung dieses Quellterms (Zeile 148-161) wurde im Quellcode (1) implementiert. Nun kann zu jedem Zeitpunkt die Differenz zwischen der stationären Lösung $T^*_{i,j}$ und der numerischen Lösung $T^N_{i,j}$ berechnet werden als

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

Diese Abweichung wird nun alle 100 Zeitschritte in der *main*-Methode (Zeile 175-196) ausgegeben.
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.

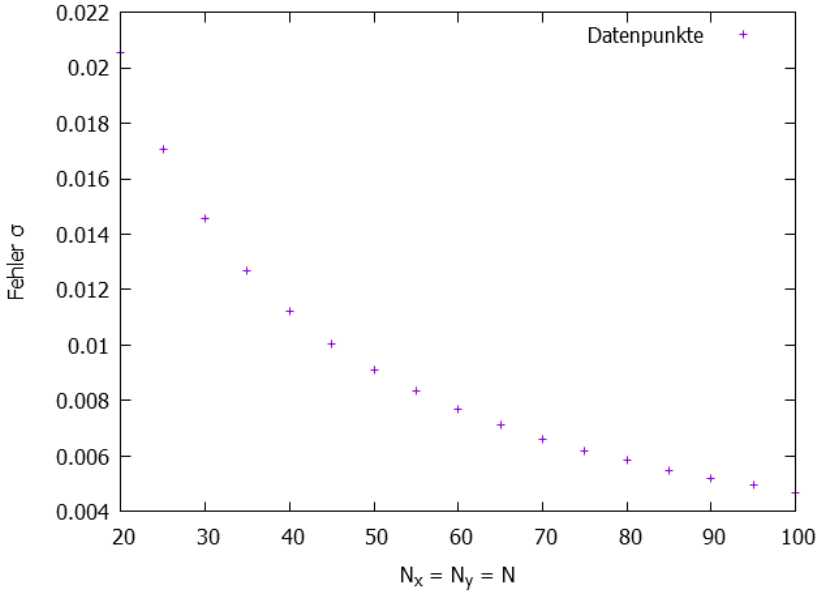


Abbildung 2: Grafische Auftragung der numerischen Abweichung σ in Abhängigkeit von der Gittergröße N.

In Abbildung (2) zeigt den 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,

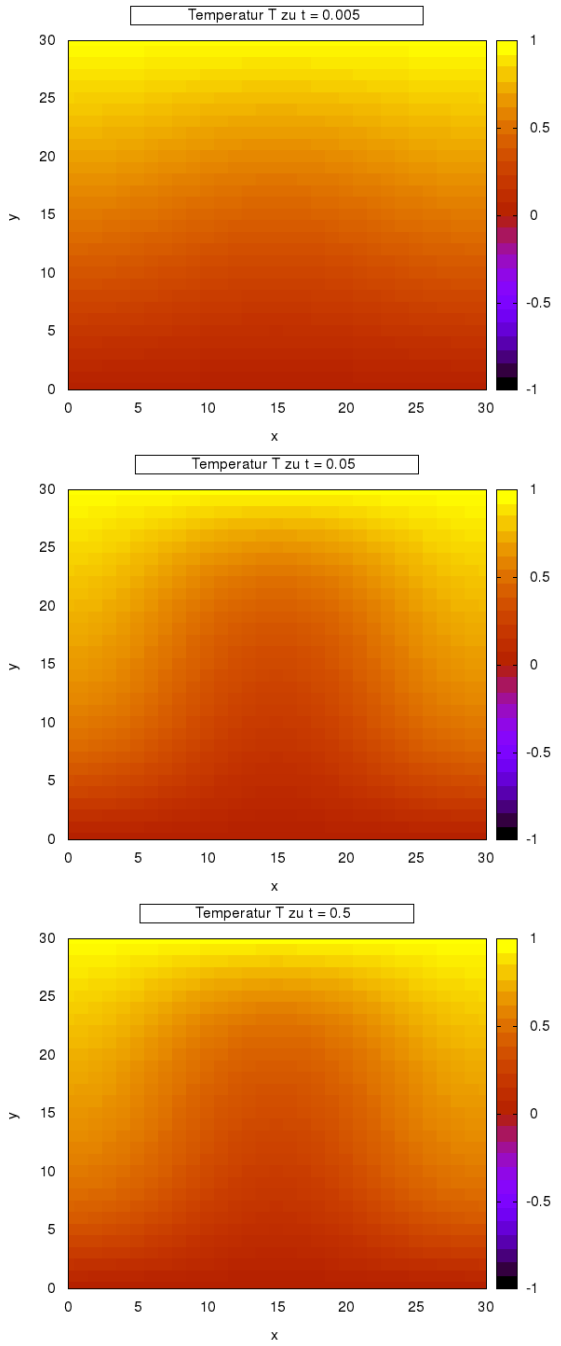


Abbildung 3: Temperatur für $Pe = 2$.

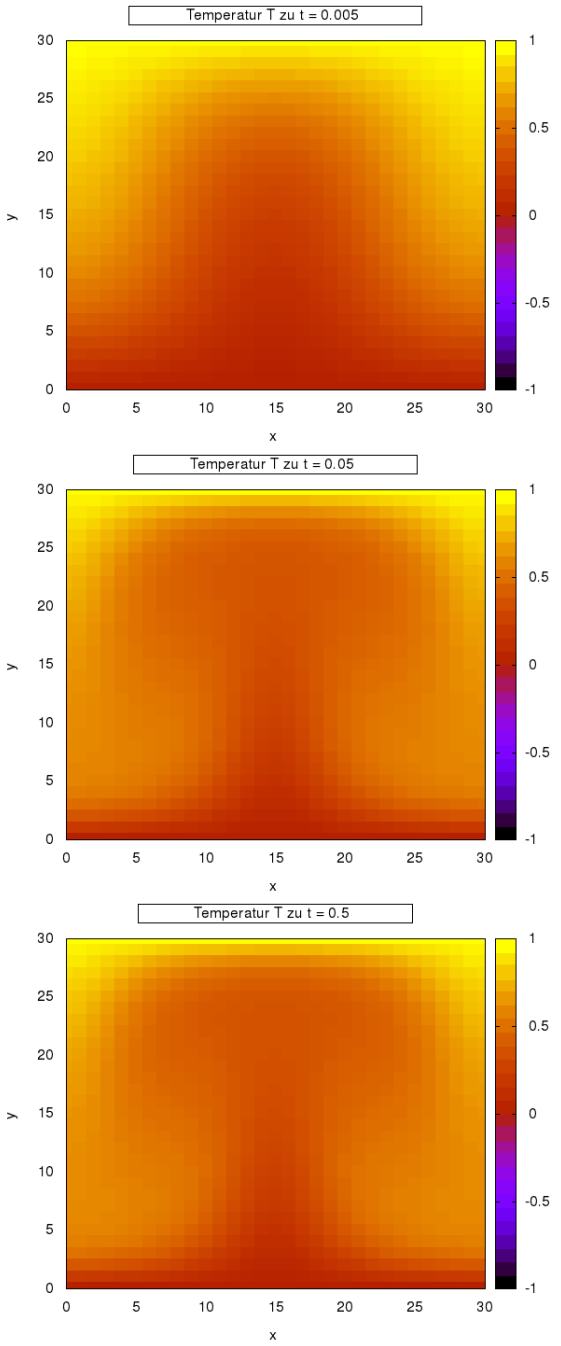


Abbildung 4: Temperatur für $Pe = 10$.

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.
Hierbei ergibt sich unabhängig von der *Péclet-Zahl* eine ähnliche Temperaturverteilung. So ist sie im unteren Bereich deutlich geringer, als im oberen Bereich, was durch die *Dirichlet*-Randbedingungen hervorgerufen wird. Zudem ist sie in der Mitte deutlich geringer, als rechts und links daneben. Dies lässt sich gut durch die Verteilung durch das Geschwindigkeitsfeld erklären. Diese transportiert anschaulich die vorhandene Wärme aus der Mitte hinweg, und verteilt sich im oberen Bereich. Bei der größeren *Péclet-Zahl* $Pe = 10$ ergibt sich eine stärkere Verteilung, sodass ein stärker ausgeprägtes Gefälle zwischen warm und kalt entsteht. Für beide Parameter geht das System allerdings schon nach der kurzen Zeitspanne $t = 0.05$ in eine annähernd stationäre Lösung über - die Unterschiede zwischen den Verteilung für $t = 0.05$ und $t = 0.5$ sind minimal.

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^{Pe}_{max} für die *Péclet-Zahlen* $Pe \in \{0.1, 1.0, 10\}$ empirisch ermittelt.
Diese ergeben sich zu

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

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

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

gewählt. Durch Einsetzen in Gleichung (8) und Kürzen ergibt sich der Ausdruck

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

welcher durch trigonometrische Identitäten in

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

übergeht. Damit das Verfahren konvergiert, muss $|\vec{\xi}|$ 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

$$\lambda^2 \left(64 + (2\pi Pe \Delta x)^2\right) - 16\lambda < 0, \tag{14}$$

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

$$0 < \lambda = \frac{\Delta t}{\Delta x^2} < \frac{4}{16 + (\Delta_x Pe \pi)^2}. \tag{15}$$

Hierdurch kann der maximale Zeitschrit, für den das Verfahren konvergiert durch

$$\Delta t_{max} = \frac{4\Delta x^2}{16 + (\pi Pe \Delta_x)^2} \tag{16}$$

ausgedrückt werden.

Péclet-Zahl	Δt_{max} (empirisch)	Δt_{max} (theoretisch)
0.1	$2.788 \cdot 10^{-4}$	$2.778 \cdot 10^{-4}$
1	$2.789 \cdot 10^{-4}$	$2.776 \cdot 10^{-4}$
10	$2.795 \cdot 10^{-4}$	$2.600 \cdot 10^{-4}$

Tabelle 1: Vergleich der empirischen und theoretischen abgeschätzten Werte für Δt_{max}.

Ein Vergleich zwischen den empirisch ermittelten Grenzzeitschritten und den maximalen Zeitschritten, für die nach der nach oben abgeschätzten theoretischen Betrachtung noch ein konvergentes Verhalten zu erwarten ist, bietet die Tabelle (1) Es ist zu erkennen, die theoretische Abschätzung für die kleinste *Péclet*-Zahl die beste Übereinstimmung mit dem empirisch gefunden Wert hat - dies hebt den Charakter der bei der Herleitung angenommenen Abschätzungen hervor.

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). Dazu werden im Hauptprogramm im Quellcode (2) vier Routinen benötigt. Die *main*-Methode liest die eingegebene Parameter erneut ein (Zeile 50-88), und befüllt die zuvor eingeführte Matrix $\mathbf{M}$. Dazu wird ein Objekt der Klasse *square_matrix* aus Quellcode (3) benutzt (Zeile 90-155). Diese erbt von der Klasse *matrix*, welche im Quellcode (4) und (6) definiert wird. Hierbei werden die Routinen *beta* und *gamma* benutzt, welche die so benannten Variablen für den jeweiligen Gitterpunkt berechnen (Zeile 21-48). Weitergehend wird das Temperaturfeld nicht mehr in einem zwei sondern in einem eindimensionalen Objekt gespeichert. Dieses wird zunächst mit den gewünschten Startwerten initialisiert (Zeile 155-170).
Nun wird über das zu betrachtende Zeitintervall iteriert, und in jedem Integrationsschritt die zuvor aufgestellte Matrix invertiert. Dafür wurde das *SOR-Verfahren* im Quellcode (7) und (5) nach Gleichung (12) 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.

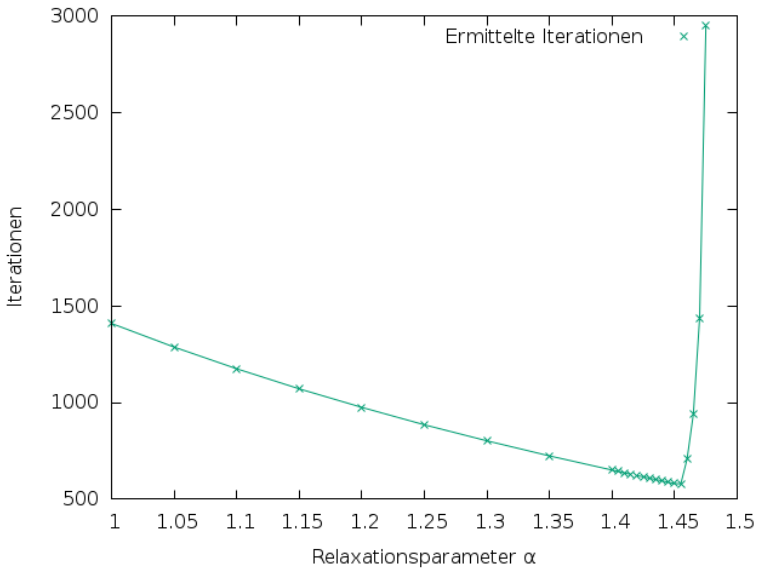


Abbildung 5: Grafische Darstellung der benötigten Iterationen zur Konvergenz in Abhängigkeit vom Relaxationsparameter α.

Die grafische Auftragung der so ermittelten Ergebnisse ist in Abbildung (5) zu finden. Es ist gut erkennbar, dass das bei $\alpha = 1.455$ die geringste Anzahl an Iterationen benötigt wurde. Die Anzahl der benötigten Iterationen für höhere Relexationsparameter α sind nicht mehr graphisch dargestellt, da hier für leicht erhöhte Werte bereits mehrere Zehntausend Iterationen benötigten wurden. Somit wird im weiteren Verlauf dieser Parameter genutzt. 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.

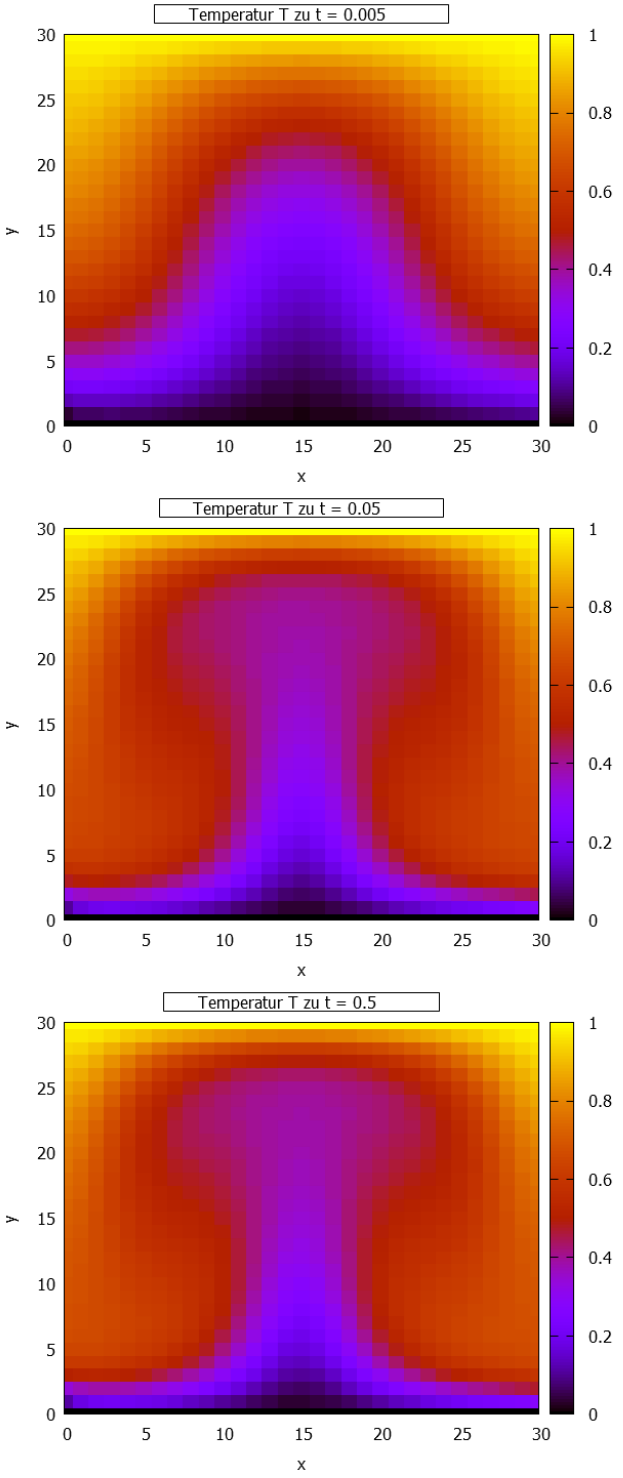


Abbildung 6: Temperatur nach dem *BTCS-Schema* für $Pe = 10$ und $\Delta t = 2 \cdot 10^{-4}$.

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.

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

Die zuvor in Abschnitt (4.5) hergeleitete Gleichung für den maximal gültigen Zeitschritt Δt_{max} ist durch die Abschätzungen nach oben, 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. Hierbei zeigte sich allerdings, dass die Aussagen der Gleichung (16) für geringe *Péclet-Zahlen* zu stark sind, und noch zu viele Werte ausschließen, für die das Verfahren noch konvergiert. Allerdings konnte durch exemplarische Tests bei höheren Werten ein Verhalten festgestellt werden, welches dem durch die Formel beschriebenen entspricht: je höher die Zahl wird, desto feiner muss die Zeitintegration gewählt werden.

5.3. Aufgabe 6: Auswertung und Vergleich der BTCS-Lösung

Nachdem nun beide Verfahren umgesetzt und getestet worden sind, folgt nun ein Vergleich dieser. Hierbei werden zuerst die dadurch erhaltenen Lösungen miteinander verglichen. Betrachtet man die Ergebnisse für die *Péclet*-Zahl $Pe = 10$ des $BTCS$-Verfahrens aus Abbildung (6) mit denen des $FTCS$-Verfahrens in Abbildung (4), so kann man eine hohe Übereinstimmung des Verlaufes festellen. Hierbei ist für diese gegebene Parameterwahl kein Unterschied feststellbar.

Simulierte Zeitspanne t	$FTCS$ [sec]	$BTCS$ [sec]
1	< 1	65
10	3	79
50	15	83
100	33	89
200	68	91
400	135	94

Tabelle 2: Exemplarischer Laufzeitvergleich zwischen dem expliziten ($FTCS$) und dem impliziten ($BTCS$) Lösungsverfahren.

Da zuvor das explizite Lösungsverfahren auf numerische Korrektheit überprüft wurde, kann also auch davon ausgegangen werden, dass das implizite Lösungsverfahren korrekt umgesetzt worden ist.
Betrachtet man allerdings die Laufzeit der beiden Verfahren, so können große Unterschiede sowohl in der Größenordnung, als auch im Verlauf festgestellt werden. Diese Laufzeiten wurden exemplarisch gemessen und in Tabelle (5.3) festgehalten. Es ist anzumerken, dass die gemessene Laufzeit von dem verwendeten Rechner und anderen äußeren Einflüssen abhängen kann, sodass diese Daten nur als Anhaltspunkt für das grobe Verhalten gesehen werden können. Zudem hätten die verschiedenen Routinen auch noch weiter hinsichtlich ihrer Laufzeit optimiert werden können. Da dies den Rahmen des Themas der Arbeit überschreiben würde, wurde darauf verzichtet.
Es ist zu erkennen, dass die Laufzeit des expliziten Verfahrens in etwa linear mit der simulierten Zeitspanne t anwächst, wohingegen die des impliziten Verfahrens deutlich geringer anwächst, und näherungsweise fast konstant bleibt ab einer bestimmten Zeitspanne t.
Somit zeigt sich, dass das explizite Verfahren zweiter Ordnung in diesem Fall seine Stärken bei kurzen und das implizite Verfahren langen Simulationen hat.

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

A.1. Hauptprogramme

```
#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 and returns the error, compared with the stationary solution. 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
//  source: a grid_t object which contains the source values
//  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, grid_t source, 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));
    }
  }

  double gradient_t_times_v0;
  double laplacian_t;

  //loop over the entire inner grid (ignore the 4 outer sides of the rectangle) and calculate the new value using the FTCS-scheme
  for (size_t x=1; x<dimension_x-1; x++)
  {
    for (size_t y=1; y<dimension_y-1; y++)
    {
      //calculates the 2nd ordner laplacian of the temp.
      laplacian_t = 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))
          + 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));

```

```
52        //calculates the inner product of the gradient of the temp. and v0
53        gradient_t_times_v0 =
              v0x.at(x).at(y)/(2.0*delta_x)*(old_values.at(x+1).at(y)-old_values.at(x-1).at(y))
54            +
                  v0y.at(x).at(y)/(2.0*delta_y)*(old_values.at(x).at(y+1)-old_values.at(x).at(y-1));
55
56        //integration step according to eq. (8)
57        values.at(x).at(y) += delta_t * (laplacian_t - pe*gradient_t_times_v0 +
              source.at(x).at(y));
58      }
59    }
60
61    //now take care of the four boundary conditions
62
63    //left/right boundaries (Neumann boundaries)
64    for (size_t y=0; y<dimension_y; y++)
65    {
66      values.at(0).at(y) = 1.0/3.0*(4.0*values.at(0+1).at(y) - values.at(0+2).at(y));
67      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));
68    }
69
70    //bottom/top boundaries (Dirichlet boundaries)
71    for (size_t x=0; x<dimension_x; x++)
72    {
73      values.at(x).at(0) = 0.0;
74      values.at(x).at(dimension_y-1) = 1.0;
75    }
76  }
77
78  int main(int argc, char* argv[])
79  {
80    //print information for the usage
81    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;
82    cout << "Expects the follwing set of parameters:" << endl;
83
84    cout << "\toutput file:\tThe path to the file in which the results will be stored."
          << endl;
85    cout << "\tN_x:\t\tThe amount of grid points in the x-direction." << endl;
86    cout << "\tN_y\t\tThe amount of grid points in the y-direction." << endl;
87    cout << "\tPe:\t\tThe Péclet-Number" << endl;
88    cout << "\tMax t\t\tThe maximum time until the system will be simulated." << endl;
89    cout << "\tDelta t\t\tThe time step size for the integration." << endl;
90    cout << "\tUse external heat source\t\tIf set to 1, then the source Q will be used."
          << endl;
91
92    //read entered parameters
93
94    //create output stream
95    ofstream outputFile;
96    outputFile.open(argv[1]);
97    outputFile << fixed << setprecision(5);
98
99    //the amount of grid points in the x direction
100   const size_t dimension_x = (atoi)(argv[2])+1;
101   //the distance between to grid points
102   const double delta_x = 1.0/(dimension_x-1);
103
104   //the amount of grid points in the x direction
105   const size_t dimension_y = (atoi)(argv[3])+1;
106   //the distance between to grid points
```

```
107   const double delta_y = 1.0/(dimension_y-1);
108
109   //Péclet-number
110   const double pe = atof(argv[4]);
111   //maximum time until which the system will be simulated
112   const double max_t = atof(argv[5]);
113   //the time step size
114   const double delta_t = atof(argv[6]);
115
116   const bool use_source = atoi(argv[7]) == 1;
117
118   cout << "Entered parameters:" << endl;
119   cout << "\tNx: " << dimension_x << "\tNy: " << dimension_y << "\tPe: " << pe <<
          "\tMax t: " << max_t << "\tDelta t: " << delta_t << endl;
120
121   //to store the x component of the velocity v0
122   grid_t v0x;
123   //to store the y component of the velocity v0
124   grid_t v0y;
125   //to store the current temperature
126   grid_t values;
127   //to store the source of the temperature
128   grid_t sources;
129
130   //create the new grids
131   for (size_t x=0; x<dimension_x; x++)
132   {
133     //add a new row
134     v0x.push_back(vector<double>());
135     v0y.push_back(vector<double>());
136     values.push_back(vector<double>());
137     sources.push_back(vector<double>());
138
139     for (size_t y=0; y<dimension_y; y++)
140     {
141       //add the v0 values
142       v0x.at(x).push_back(M_PI*sin(2*M_PI*x*delta_x)*cos(M_PI*y*delta_y));
143       v0y.at(x).push_back(-2.0*M_PI*cos(2*M_PI*x*delta_x)*sin(M_PI*y*delta_y));
144
145       //add the start temperature of the rectangle
146       values.at(x).push_back(y*delta_y);
147
148       //add the source of the temperature of the rectangle
149       if (use_source)
150       {
151         sources.at(x).push_back(
152           pow(M_PI,2)*(cos(M_PI*delta_x*x)*sin(M_PI*delta_y*y)*2
153           - pe*pow(cos(x*delta_x*M_PI),3)*sin(2*M_PI*delta_y*y))
154           -pe*2*M_PI*cos(2*M_PI*delta_x*x)*sin(M_PI*y*delta_y)
155         );
156       }
157       else
158       {
159         //no heat source is desired => set it to 0 for all points
160         sources.at(x).push_back(0.0);
161       }
162     }
163   }
164
165   //to print just every 100th error comparison
166   size_t iteration = 0;
167   //is being used to calculate the numerical error compared to stat.solution
168   double error;
```

```
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     ftcs_time_step(values, v0x, v0y, sources, delta_t, delta_x, delta_y, pe);
173
174     //the heat source is beeing used => calcuate the difference
175     if (use_source)
176     {
177       //now calculate the difference, compared to the stationary solution
178       //loop over all elements and compare them
179       error = 0.0;
180       for (size_t x=0; x<dimension_x; x++)
181       {
182         for (size_t y=0; y<dimension_y; y++)
183         {
184           error += pow(
185                       values.at(x).at(y) -
                                (cos(M_PI*delta_x*x)*sin(M_PI*delta_y*y)+y*delta_y)
186                     ,2);
187         }
188       }
189
190       //print every 100th error value
191       if (iteration++ % 100 == 0)
192       {
193           cout << "time:\t" << t << "\terror:\t" << error << "\n" << flush;
194           iteration = 0;
195       }
196     }
197   }
198
199   //print the final grid's values in the givven output file in a matrix form
200   for (size_t y=0; y<dimension_y; y++)
201   {
202     for (size_t x=0; x<dimension_x; x++)
203     {
204       outputFile << values.at(x).at(y) << "\t";
205     }
206     outputFile << "\n";
207   }
208
209   return 1;
210 }
```

Quellcode 1: Implementation des FTCS-Schemas

```
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 //unit matrix
102 //main diagonal for the dirichlet boundaries again
103 for (size_t i = 0; i<dimension_x; i++)
104 {
105   coeff_matrix.set_value(i, i, 1.0);
106 }
107
108 //Matrix M
109 //contains the biggest part of the BTCS-scheme
110 for (size_t i = dimension_x+1; i<n - dimension_x-1; i++)
111 {
112   //ingore this diagonals, as they affect the boundaries
113   if (i % dimension_x == 0 || i % dimension_x == dimension_x-1)
114   {
115     continue;
116   }
117
118   //i % dimension_x gives the x coordinate of the field-value on which the current
        matrix's item operates
119   //i / dimension_x gives the y coordinate of the field-value on which the current
        matrix's item operates
120   coeff_matrix.set_value(i, i - dimension_x, gamma(false, delta_t, delta_x, delta_y,
        pe, i % dimension_x, i/dimension_x));
121   coeff_matrix.set_value(i, i - 1, beta(false, delta_t, delta_x, delta_y, pe, i %
        dimension_x, i/dimension_x));
122   coeff_matrix.set_value(i, i, alpha);
123   coeff_matrix.set_value(i, i + 1, beta(true, delta_t, delta_x, delta_y, pe, i %
        dimension_x, i/dimension_x));
124   coeff_matrix.set_value(i, i + dimension_x, gamma(true, delta_t, delta_x, delta_y,
        pe, i % dimension_x, i/dimension_x));
125 }
```

```
126
127   //now set the repeating line which take care of the neumann boundaries
128   //first left boundary
129   coeff_matrix.set_value(dimension_x, dimension_x+2, 1.0/3.0);
130   coeff_matrix.set_value(dimension_x, dimension_x+1, -4.0/3.0);
131   coeff_matrix.set_value(dimension_x, dimension_x, 1);
132   for (size_t i=2; i<dimension_x-1; i++)
133   {
134     //right
135     coeff_matrix.set_value(i*dimension_x-1, i*dimension_x-3, 1.0/3.0);
136     coeff_matrix.set_value(i*dimension_x-1, i*dimension_x-2, -4.0/3.0);
137     coeff_matrix.set_value(i*dimension_x-1, i*dimension_x-1, 1);
138
139     //left
140     coeff_matrix.set_value(i*dimension_x, i*dimension_x, 1);
141     coeff_matrix.set_value(i*dimension_x, i*dimension_x+1, -4.0/3.0);
142     coeff_matrix.set_value(i*dimension_x, i*dimension_x+2, 1.0/3.0);
143   }
144
145   //last right boundary
146   coeff_matrix.set_value((dimension_x)*(dimension_x-1)-1,
          (dimension_x)*(dimension_x-1)-3, 1.0/3.0);
147   coeff_matrix.set_value((dimension_x)*(dimension_x-1)-1,
          (dimension_x)*(dimension_x-1)-2, -4.0/3.0);
148   coeff_matrix.set_value((dimension_x)*(dimension_x-1)-1,(dimension_x)*(dimension_x-1)-1,
          1);
149
150   //unit matrix
151   //main diagonal for the dirichlet boundaries again
152   for (size_t i = n - dimension_x; i<n; i++)
153   {
154     coeff_matrix.set_value(i, i, 1.0);
155   }
156
157
158   //create two vectors to store the temperature and to calculate the new one
159   vector<double> values;
160   vector<double> old_values;
161
162   //fill them with the starting temperature
163   for (size_t y = 0; y<dimension_y; y++)
164   {
165     for (size_t x = 0; x<dimension_x; x++)
166     {
167       old_values.push_back(y*delta_y);
168       values.push_back(y*delta_y);
169     }
170   }
171
172   //start to measure the runtime now
173   clock_t start_time = clock();
174
175   //loop over the time, until we have reached the maximum time
176   for (double t = 0.0; t<max_t; t += delta_t)
177   {
178     //set the right hand side of the equation (set some values to zero for the neuman
          boundaries)
179     old_values.at(dimension_x) = 0;
180     for (size_t i=2; i<dimension_x-1; i++)
181     {
182       old_values.at(i*dimension_x-1) = 0;
183       old_values.at(i*dimension_x) = 0;
184     }
```

```
185     old_values.at((dimension_x)*(dimension_x-1)-1) = 0;
186
187     //invert the matrix M/solve the linear system to get the new temperature(value)
188     cout << "t: " << t << "\titerations: " << linear_system_solve_sor(coeff_matrix,
            values, old_values, 1.455, 1e-4) << endl;
189
190     //copy the new values into the old_values vector for the next iteration
191     for (size_t i = 0; i<n; i++)
192     {
193       old_values.at(i) = values.at(i);
194     }
195   }
196
197   //measure the end time
198   clock_t end_time = clock();
199
200   //print the runtime
201   cout << "Integration finished after " << (end_time - start_time) / CLOCKS_PER_SEC <<
        " seconds.";
202
203   //print the final grid's values in the givven output file in a matrix form
204   for (size_t i = 0; i<n; i++)
205   {
206     outputFile << values.at(i) << "\t";
207     if (i > 0 && (i+1) % dimension_x == 0)
208     {
209       outputFile << endl;
210     }
211   }
212
213   return 1;
214 }
```

Quellcode 2: Implementation des BTCS-Schemas

A.2. Hilfsbibliotheken

```
#include "matrix.h"

#ifndef SQUAREMATRIX_H
#define SQUAREMATRIX_H
namespace numerical
{
  class square_matrix :  public matrix
  {
    public:
      //creates a nXn matrix
      square_matrix(int n) : matrix(n,n) {}
        //creates a nXn matrix and initializes all values with defaultValue
      square_matrix(int n, double defaultValue) : matrix(n,n, defaultValue) {};
  };
}
#endif
```

Quellcode 3: square_matrix.h

```
#include <fstream>

#ifndef MATRIX_H
#define MATRIX_H
namespace numerical{
  class matrix
  {
    public:
      //creates a new nXm matrix
      matrix(size_t n, size_t m);
      //creates a new nXm matrix and initializes all values with defaultValue
      matrix(size_t n, size_t m, double defaultValue);
      //deconstructor
      ~matrix();

      //returns the number of rows/columns
      size_t get_n();
      size_t get_m();

      //returns the value in row n, column m
      double get_value(size_t n, size_t m);
      //sets the value in row n, column m
      void set_value(size_t n, size_t m, double value);
    private:
      double** data;
      size_t n;
      size_t m;
  };
}
#endif
```

Quellcode 4: matrix.h

```
#include "square_matrix.h"
#include <vector>

using namespace std;

#ifndef LINEARSYSTEM_H
#define LINEARSYSTEM_H
namespace numerical
```

```
{
  size_t linear_system_solve_sor(square_matrix& coeff_matrix, vector<double> &x,
      vector<double> &b, double alpha, double error_threshold);
  double linear_system_solve_sor_step(square_matrix& coeff_matrix, vector<double> &x,
      vector<double> &b, double alpha);

  void multiply_matrix_vector(matrix& matrix, vector<double> &x, vector<double>
      &result);
}
#endif
```

Quellcode 5: linear_system.h

```
#include "matrix.h"
#include <iostream>

namespace numerical{
  //creates a new nXm matrix
  matrix::matrix(size_t n, size_t m)
  {
    this->n = n;
    this->m = m;
    data = new double*[m];
    for (size_t i=0;i<m;i++)
    {
      data[i] = new double[n];
    }
  }

  //creates a new nXm matrix and initializes all values with defaultValue
  matrix::matrix(size_t n, size_t m, double defaultValue)
  {
    this->n = n;
    this->m = m;
    data = new double*[m];
    for (size_t i=0;i<m;i++)
    {
      data[i] = new double[n];
      for (size_t j=0;j<n;j++)
      {
        data[i][j] = defaultValue;
      }
    }
  }

  //deconstructor
  matrix::~matrix()
  {
    delete data;
  }

  //sets the value in the nth row, mth column
  void matrix::set_value(size_t n, size_t m, double value)
  {
    data[n][m] = value;
  }

  //returns the value in the nth row, mth column
  double matrix::get_value(size_t n, size_t m)
  {
    return data[n][m];
  }

```

```
51    //returns the number of rows
52    size_t matrix::get_n()
53    {
54        return this->n;
55    }
56
57    //returns the number of columns
58    size_t matrix::get_m()
59    {
60      return this->m;
61    }
62  }
```

Quellcode 6: matrix.cpp

```
1  #include "linear_system.h"
2  #include "square_matrix.h"
3  #include <iostream>
4  #include <fstream>
5  #include <cmath>
6  #include <vector>
7  #include <float.h>
8
9  using namespace std;
10
11 namespace numerical
12 {
13   void multiply_matrix_vector(matrix& matrix, vector<double> &xVec, vector<double>
         &result)
14   {
15     result.clear();
16     double val;
17     for (size_t y = 0; y<matrix.get_n(); y++)
18     {
19       val = 0.0;
20       for (size_t x = 0; x<matrix.get_m(); x++)
21       {
22         val += matrix.get_value(y, x) * xVec.at(x);
23       }
24       result.push_back(val);
25     }
26   }
27
28   //calculates one SOR step the for system coeff_matrix * x = b with the relaxation
         parameter alpha
29   double linear_system_solve_sor_step(square_matrix& coeff_matrix, vector<double> &x,
         vector<double> &b, double alpha)
30   {
31     //the new value of the current item
32     double new_value;
33     //a helper variable to store a needed sum of items in the loop
34     double sum;
35
36     //loop over all items and calculate the next elemt of x
37     for (size_t k = 0; k<coeff_matrix.get_n(); k++)
38     {
39       sum = 0.0;
40
41       //splitted the one loop into two seperate loops for better performance.
42       //one of the loops is already using the new values while the other one is using
             the old one
43       for (size_t i = 0; i<k; i++)
44       {
```

```
45          sum += coeff_matrix.get_value(k, i)*x.at(i);
46        }
47        for (size_t i = k + 1; i<coeff_matrix.get_n(); i++)
48        {
49          sum += coeff_matrix.get_value(k, i)*x.at(i);
50        }
51        new_value = (1.0 - alpha)*x.at(k) + alpha / coeff_matrix.get_value(k,
              k)*(b.at(k) - sum);
52
53        //update the new x value
54        x.at(k) = new_value;
55      }
56
57      //calculate the residue now
58      double residue = 0.0;
59      vector<double> testVector;
60      multiply_matrix_vector(coeff_matrix, x, testVector);
61      for (size_t k = 0; k<coeff_matrix.get_n(); k++)
62      {
63        residue += pow(testVector.at(k) - b.at(k), 2);
64      }
65
66      //return the length of the residue vector
67      return sqrt(residue);
68    }
69
70    //tries to solve the linear equation coeff_matrix * x = b for x iterative with the
          SOR method untill either:
71    //1) the residues are lower than the specified error_threshold
72    //2) the residues of 50 following iterations have been increasing => propably no
          convergence for this system
73    //returns the needed iterations
74
75    size_t linear_system_solve_sor(square_matrix& coeff_matrix, vector<double> &x,
          vector<double> &b, double alpha, double error_threshold)
76    {
77      //contains the value of the residues of the current iteration. start with a high
            number because of the if-comparison below
78      double error = DBL_MAX;
79      //contains the value of the residues of the previous iteration. start with a high
            number because of the if-comparison below
80      double old_error;
81      //counter of the needed iterations
82      size_t iteration = 0;
83
84      //contains how often the residues of two following iterations have been increasing
85      size_t residue_increased_counter = 0;
86
87      //loop, until a solution has been found
88      while (true)
89      {
90        //swap the old and the current error value
91        old_error = error;
92        //increase the number of needed iterations
93        iteration++;
94
95        //calculate the next SOR step
96        error = linear_system_solve_sor_step(coeff_matrix, x, b, alpha);
97
98
99        //check for exit condition 1)
100       if (error < error_threshold)
101       {
```

```
102           return iteration;
103         }
104         else if (error > old_error)
105         {
106           //exit condition 2
107           if (residue_increased_counter > 50){
108             return iteration;
109           }
110           residue_increased_counter++;
111         }
112         else
113         {
114           //reset the counter for the second exit condition.
115           residue_increased_counter=0;
116         }
117       }
118     }
119 }
```

Quellcode 7: linear_system.cpp