

Protokoll: Numerische Behandlung der Navier-Stokes-Gleichungen

Clemens Anschütz
Markus Pawellek

22. April 2016

Inhaltsverzeichnis

1 Aufgaben	1
2 Mathematische und physikalische Grundlagen	2
2.1 Die Navier-Stokes Gleichungen	2
2.2 Randbedingungen	3
3 Numerik und Durchführung	4
3.1 Räumliche Diskretisierung und Gitter	4
3.2 Lösen der Impulsgleichungen	4
3.3 Ableitungs - Stencils	5
3.4 Berechnung des Drucks	7
4 Simulation und Ergebnisse	8
4.1 Ausgabe und Visualisierung	8
4.2 Zeitevolution einer Flüssigkeit	8
4.3 Einfluss der Reynoldszahl	10
4.4 Untersuchung von rechtwinkligen Geometrien	12
4.5 Periodische Anregung	12
5 Zusammenfassung	18
A Quellcode	20
B Literaturbeispiele	41

1 Aufgaben

Schreiben Sie ein Programm, welches die Navier-Stokes Gleichung für eine inkompressible Flüssigkeit in zwei Dimensionen löst. Beschäftigen Sie sich hierfür zunächst mit der Theorie der Navier-Stokes Gleichung und der Finiten-Differenzen Methode. Implementieren Sie anschließend Ihr Programm unter Berücksichtigung der Versuchsanleitung. Simulieren und visualisieren Sie mit Ihrem Programm eine Nischenströmung und vergleichen Sie ihre Resultate mit den aus der Literatur bekannten Werten.

- Überlegen Sie sich im ersten Schritt die Struktur Ihres Programmes. Es ist sinnvoll, den Quellcode in logische Einheiten zu unterteilen.
- Überlegen Sie sich, welche Parameter Sie für Ihr Programm benötigen werden. Diese sollten nicht im Quellcode gesetzt werden, sondern von einer Parameterdatei eingelesen werden.
- Überlegen Sie sich, welche Funktionen Sie benötigen, um den Algorithmus in Ihr Programm zu implementieren. Welche Übergabe- und Rückgabeparameter haben diese Funktionen?
- Kommentieren Sie Ihren Code für eine übersichtliche Dokumentation.
- Machen Sie sich Gedanken über die Ausgabe ihres Programmes. Für die Visualisierung des Geschwindigkeitsfeldes wird ein Programm benötigt, welches zweidimensionale Vektordaten darstellen kann.
- Analysieren Sie die Zeitevolution der Flüssigkeit. Achten Sie hierbei auf die Ausbildung von Wirbeln und auf den sich einstellenden stationären Endzustand.
- Studieren Sie das Verhalten unterschiedlich zäher Flüssigkeiten, indem Sie die Reynoldszahl variieren. Wie wirkt sich dies auf die Wirbelbildung und den stationären Endzustand aus?
- Zur Validierung Ihres Codes vergleichen Sie Ihre Ergebnisse mit denen aus der Literatur bekannten Simulationen.
- Nachdem Sie sich von der Richtigkeit überzeugt haben, experimentieren Sie mit unterschiedlichen Boxgeometrien. Wie verhält sich die Flüssigkeit für rechteckige Geometrien?
- Gehen Sie am oberen Rand von der konstanten Geschwindigkeitskomponente in eine sich zeitlich periodische Ändernde über.

2 Mathematische und physikalische Grundlagen

2.1 Die Navier-Stokes Gleichungen

Die Navier-Stokes Gleichungen lassen sich aus den Eulergleichungen wie sie aus der Thermodynamik bekannt sind ableiten. Für inkompressible Flüssigkeiten ($\rho = \text{const}$) bilden sie folgendes System partieller Differentialgleichungen:

$$\begin{aligned} \frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla) \vec{u} + \nabla p &= \frac{1}{Re} \nabla \vec{u} + \vec{g} \quad (\text{Impulsgleichung}) \\ \nabla \cdot \vec{u} &= 0 \quad (\text{Kontinuitätsgleichung}) \end{aligned}$$

Dieses gilt es nun mittels numerischer Verfahren zu berechnen. Re bezeichnet hierbei die Reynoldszahl, eine dimensionslose Zahl, die zur Skalierung von Modellen verwendet wird und sich wie folgt ergibt:

$$Re = \frac{\rho \cdot v \cdot d}{\eta}$$

In der Gleichung steht v für die Geschwindigkeit der Strömung, d für eine charakteristische Länge und η für die dynamische Viskosität des Fluids. Im Programm wird Re zu einer Konstanten, die Aussagen über das Strömungsverhalten liefert. Kleine Reynoldsahlen bedeuten dann sehr viskose Fluide und meist laminare Strömung, Fluide mit großen Reynoldsahlen zeigen meist turbulente Strömungen und müssen mit mehr Aufwand simuliert werden.

Da wir im vorliegenden Fall nur zweidimensionale Probleme betrachten wollen, behandeln wir die Entwicklung der Geschwindigkeit \vec{u} komponentenweise in der Form:

$$\vec{u} = u \vec{e}_x + v \vec{e}_y$$

Für diese skalaren Größen lauten die Zeitevolutionen dann:

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} &= \frac{1}{Re} \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \frac{\partial(u^2)}{\partial x} - \frac{\partial(uv)}{\partial y} + g_x \\ \frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} &= \frac{1}{Re} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{\partial(uv)}{\partial x} - \frac{\partial(v^2)}{\partial y} + g_y \end{aligned}$$

Die Kontinuumsgleichung lässt sich dann einfach in der Form schreiben:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0$$

2.2 Randbedingungen

Um die Randbedingungen zu formulieren, zerlegen wir die Geschwindigkeitsvektoren an den Rändern in ihre tangentialen und normalen Komponenten zum Rand. Wir bezeichnen diese mit φ_t und φ_n . Die Randbedingungen definieren wir dann mit Hilfe dieser Komponenten und deren Normalenbleitungen. In diesem Versuch betrachten wir nur den einfachen Fall von senkrechten und waagerechten Randstücken.

- Somit ergeben sich für senkrechte Randstücke:

$$\varphi_n = u, \quad \varphi_t = v, \quad \frac{\partial \varphi_n}{\partial n} = \frac{\partial u}{\partial x}, \quad \frac{\partial \varphi_t}{\partial n} = \frac{\partial v}{\partial x}$$

- Für waagerechte Randstücke folgt:

$$\varphi_n = v, \quad \varphi_t = u, \quad \frac{\partial \varphi_n}{\partial n} = \frac{\partial v}{\partial y}, \quad \frac{\partial \varphi_t}{\partial n} = \frac{\partial u}{\partial y}$$

Mit Hilfe dieser Definitionen lassen sich dann z.B. die Haftbedingung formulieren. Diese gilt, wenn die Grenzen undurchlässig und star sind. Flüssigkeit kann dann weder in noch aus dem Gebiet Ω fließen und ruht am Rand.

$$\varphi_n(x, y) = 0, \quad \varphi_t(x, y) = 0$$

3 Numerik und Durchführung

3.1 Räumliche Diskretisierung und Gitter

Die Simulation findet im rechteckigen Gebiet $\Omega := [0, a] \times [0, b] \subset \mathbb{R}^2$ statt. Dieses wird in $i_{max} \cdot j_{max}$ rechteckige Zellen zerlegt. Die Zellen des räumlichen Gitters besitzen die Maße $\delta x \cdot \delta y$ wobei gilt:

$$\delta x := \frac{a}{i_{max}} \quad \text{und} \quad \delta y := \frac{b}{j_{max}}$$

Die skalaren Felder u , v , und p ordnen wir auf diesem Gitter wie in Abbildung 1 zu sehen an.

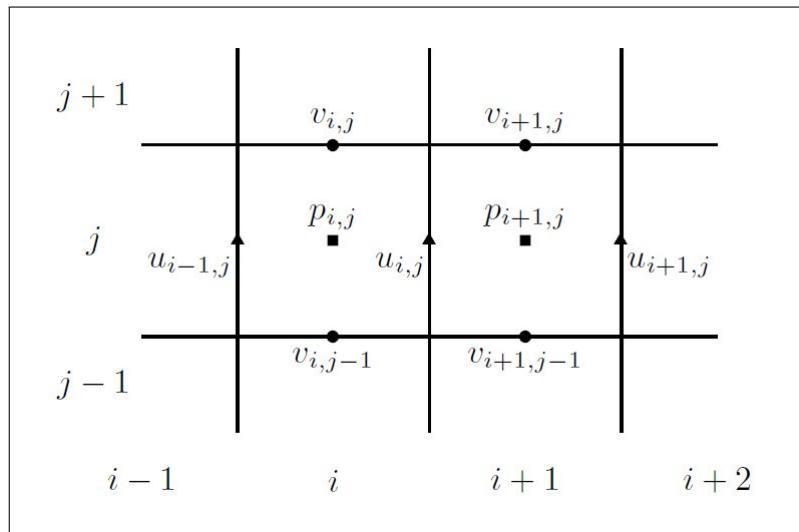


Abbildung 1: Schema für die Position der Geschwindigkeitskomponenten u , v sowie des Drucks p auf dem Gitter
Quelle: [3]

Durch die verschobenen Gitter lassen sich mögliche Oszillationen und Uneindeutigkeiten des Drucks verhindern. Die grauen Zellen in Abbildung 2 bilden die Randzellen, sie erhalten bei Initialisierung des Programms feste Werte entsprechend den jeweiligen Randwertproblemen und behalten diese während der gesamten Simulation bei.

3.2 Lösen der Impulsgleichungen

Die Variablen werden zum Zeitpunkt n betrachtet, die nachfolgenden Geschwindigkeitskomponenten werden mit Hilfe eines einfachen Euler-Vorwärts-Schrittes berechnet:

$$u^{(n+1)} = F^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial x}$$

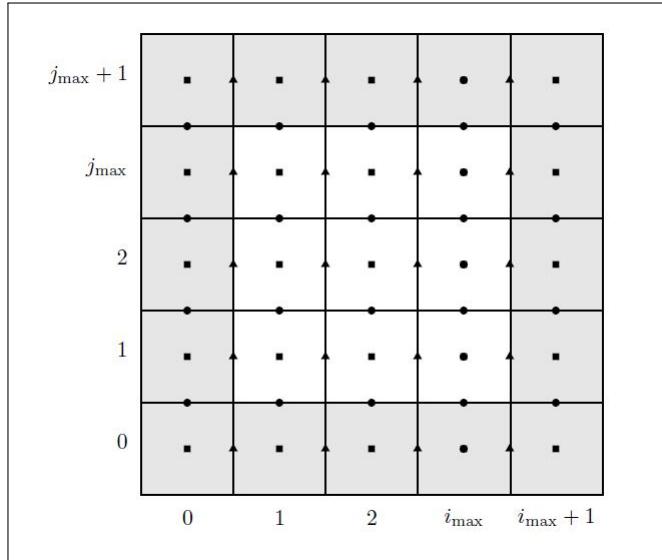


Abbildung 2: Darstellung der Randzellen mit den entsprechenden Randbedingungen
Quelle: [3]

$$v^{(n+1)} = G^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial y}$$

Dabei sind die Funktionen F und G wie folgt definiert:

$$F = u + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \frac{\partial(u^2)}{\partial x} - \frac{\partial(uv)}{\partial y} + g_x \right]$$

$$G = v + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{\partial(uv)}{\partial x} - \frac{\partial(v^2)}{\partial y} + g_y \right]$$

Das Verfahren ist wie wir sehen explizit in den Geschwindigkeitskomponenten und implizit im Druck. Die Druckberechnung wird unter Abschnitt 3.4 genauer beschreiben. Der Zeitschritt δt muss sorgfältig gewählt werden, um die Stabilität der Simulation zu gewährleisten. Nach Griebel, Dornseifer und Neuhöfer sollte er wie folgendermaßen gewählt werden:

$$\delta t := \tau \min \left(\frac{Re}{2} \frac{1}{\delta x^{-2} + \delta y^{-2}}, \frac{\delta x}{|u_{max}|}, \frac{\delta y}{|v_{max}|} \right)$$

Hierbei ist τ ein Sicherheitsfaktor zwischen 0 und 1. Im Programm wird er meist auf 0.5 gesetzt.

3.3 Ableitungs - Stencils

Die Ableitungen werden mit Hilfe der Finiten-Differenzen-Methode berechnet. Dabei werden die ersten Ortsableitungen beispielsweise des Drucks durch den einfachen rechtseitigen Differenzenquotienten approximiert.

$$\left[\frac{\partial p}{\partial x} \right]_{i,j} = \frac{p_{i+1,j} - p_{i,j}}{\delta x}, \quad \left[\frac{\partial p}{\partial y} \right]_{i,j} = \frac{p_{i,j+1} - p_{i,j}}{\delta y}$$

Für die zweiten Ableitungen werden standardmäßig zentrierte Differenzen verwendet.

$$\left[\frac{\partial^2 u}{\partial x^2} \right]_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\delta x)^2}, \quad \left[\frac{\partial^2 u}{\partial y^2} \right]_{i,j} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\delta y)^2}$$

Analoge Vorschriften liegen für die Ableitungen von v vor. Die Ableitungen der gemischten und nichtlinearen Terme werden durch eine Linearkombination von zentrierten Differenzen und Donor-Cell-Stencils ermittelt.

$$\begin{aligned} \left[\frac{\partial(u^2)}{\partial x} \right]_{i,j} &= \frac{1}{4\delta x} \left((u_{i,j} + u_{i+1,j})^2 - (u_{i-1,j} + u_{i,j})^2 \right) \\ &\quad + \gamma \frac{1}{4\delta x} (|u_{i,j} + u_{i+1,j}| (u_{i,j} - u_{i+1,j}) - |u_{i-1,j} + u_{i,j}| (u_{i-1,j} - u_{i,j})) \\ \left[\frac{\partial(v^2)}{\partial y} \right]_{i,j} &= \frac{1}{4\delta y} \left((v_{i,j} + v_{i,j+1})^2 - (v_{i,j-1} + v_{i,j})^2 \right) \\ &\quad + \gamma \frac{1}{4\delta y} (|v_{i,j} + v_{i,j+1}| (v_{i,j} - v_{i,j+1}) - |v_{i,j-1} + v_{i,j}| (v_{i,j-1} - v_{i,j})) \\ \left[\frac{\partial(uv)}{\partial x} \right]_{i,j} &= \frac{1}{4\delta x} ((u_{i,j} + u_{i,j+1})(v_{i,j} + v_{i+1,j}) - (u_{i-1,j} + u_{i-1,j+1})(v_{i-1,j} + v_{i,j})) \\ &\quad + \gamma \frac{1}{4\delta x} (|u_{i,j} + u_{i,j+1}| (v_{i,j} - v_{i+1,j}) - |u_{i-1,j} + u_{i-1,j+1}| (v_{i-1,j} - v_{i,j})) \\ \left[\frac{\partial(uv)}{\partial y} \right]_{i,j} &= \frac{1}{4\delta y} ((u_{i,j} + u_{i,j+1})(v_{i,j} + v_{i+1,j}) - (u_{i,j-1} + u_{i,j})(v_{i,j-1} + v_{i+1,j-1})) \\ &\quad + \gamma \frac{1}{4\delta y} (|v_{i,j} + v_{i,j+1}| (u_{i,j} - u_{i,j+1}) - |v_{i,j-1} + v_{i+1,j-1}| (u_{i,j-1} - u_{i,j})) \end{aligned}$$

Der Vorfaktor γ gewichtet hierbei die Anteile von zentrierten Differenzen und Donor-Cell-Stencils. Er sollte nach Hirt so gewählt werden, dass er folgende Bedingung erfüllt:

$$\gamma \geq \max_{ij} \left(\frac{|u_{i,j}| \delta t}{\delta x}, \frac{|v_{i,j}| \delta t}{\delta y} \right)$$

Jedoch wird er in unserem Programm zu Beginn auf einen konstanten Wert gesetzt, wie er auch in anderen Simulationen bereits verwendet wurde.

3.4 Berechnung des Drucks

Zur Bestimmung des Drucks verwenden wir die Kontinuitätsgleichung.

$$\frac{\partial u^{n+1}}{\partial x} + \frac{\partial v^{n+1}}{\partial y} = 0$$

Zusammen mit der unter 3.2 angegebenen Gleichung für die Ermittlung der Geschwindigkeiten lässt sich folgende Poisson-Gleichung für den Druck ableiten:

$$\frac{\partial^2 p^{(n+1)}}{\partial x^2} + \frac{\partial^2 p^{(n+1)}}{\partial y^2} = \frac{1}{\delta t} \left(\frac{\partial F^{(n)}}{\partial x} + \frac{\partial G^{(n)}}{\partial y} \right)$$

Die zweiten und ersten Ortsableitungen werden analog zu den unter 3.3 beschriebenen Ableitungen gebildet. Zum lösen der diskreten Poissons-Gleichung wird hier das Successive-Over-Relaxation Verfahren (SOR) angewendet. Bei diesem iterativen Verfahren setzen wir den Startwert zu $p_{i,j}^{(n)}$, jeder weitere Iterationsschritt ergibt sich dann zu:

$$p_{i,j}^{it+1} = (1-\omega)p_{i,j}^{it} + \frac{\omega}{2(\delta x)^{-2} + 2(\delta y)^{-2}} \left(\frac{p_{i+1,j}^{it} + p_{i-1,j}^{it+1}}{(\delta x)^2} + \frac{p_{i,j+1}^{it} + p_{i,j-1}^{it+1}}{(\delta y)^2} - \text{RHS}_{i,j} \right)$$

Wobei RHS die rechte Seite der Poissons-Gleichung bezeichnet. In jeder Iteration wird das Residuum r berechnet

$$r_{i,j}^{it+1} = \frac{p_{i+1,j}^{it+1} - 2p_{i,j}^{it+1} + p_{i-1,j}^{it+1}}{(\delta x)^2} + \frac{p_{i,j+1}^{it+1} - 2p_{i,j}^{it+1} + p_{i,j-1}^{it+1}}{(\delta y)^2} - \text{RHS}_{i,j}$$

bis die Norm des Residuumms eine vorgegebene relative Toleranzgrenze unterschreitet.

$$\|r^{it+1}\| < \epsilon \|p^{it}\|$$

Wobei die Norm im Versuch durch die Maximumsnorm gebildet wird und ϵ eine festgelegte kleine Konstante darstellt. Die Randzellen werden bei jedem Iterationsschritt einfach durch Kopieren der jeweiligen Nachbarzellen gefüllt.

4 Simulation und Ergebnisse

4.1 Ausgabe und Visualisierung

Für die Visualisierung wurde mithilfe der Bibliothek *Qt v5.1.1* eine Klasse erstellt, welche durch die Übergabe eines diskretisierten Vektorfeldes, dieses automatisch darstellt. Dabei wird eine bestimmte Anzahl an Position zufällig gewählt. An diesen Position werden dann Teilausschnitte der Strömungslinien des Vektorfeldes angezeigt. Die Länge stellt ein natürliches Maß für die Stärke des Vektorfeldes an den jeweiligen Punkten dar. In Abbildung 3 ist als Beispiel das folgende Vektorfeld \vec{v} auf dem Bereich $[0, 1]^2$ dargestellt worden.

$$\vec{v}(x, y) := x\vec{e}_x + \sqrt{x} \sin(3\pi x) \sin(\pi y)\vec{e}_y$$



Abbildung 3: Visualisierung des Beispielvektorfeldes \vec{v} durch das erstellte Programm.
Vergleiche mit [3].

Für diskretisierte skalare Felder, wie den Druck, ist die selbe Klasse zuständig. Dabei werden verschiedene Farbwerte verwendet um die Stärke des Feldes zu verdeutlichen. Im Programm selbst wurden stärkere Werte durch rötlichere Farben und schwächere durch bläuliche Farben dargestellt. Als Beispiel sei die Visualisierung des skalaren Feldes v_y durch das Programm in Abbildung 4 gegeben.

4.2 Zeitevolution einer Flüssigkeit

Im Folgenden wurden für das Programm die unten stehenden Parameter verwendet.

$$Re = 100 \quad u = 1.28 \quad i_{max} = 64 \quad j_{max} = 64$$

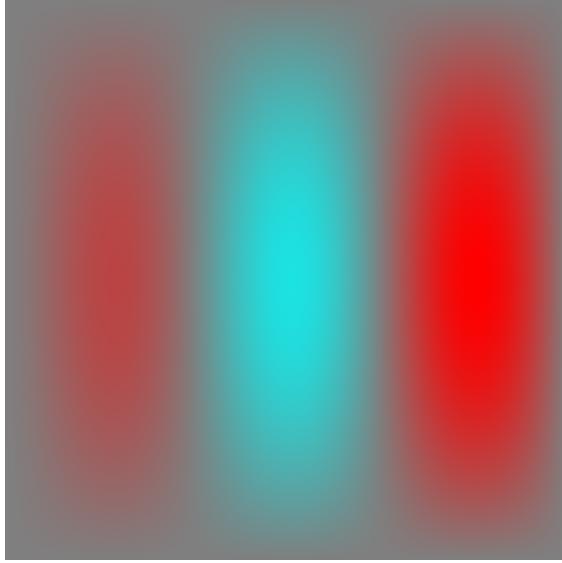


Abbildung 4: Visualisierung des skalaren Feldes v_y durch das Programm.
Vergleiche mit [3].

$$a = b = 1.0 \quad \tau = 0.5 \quad \gamma = 0.5 \quad \epsilon = 0.001 \quad \omega = 0.5$$

Diese Parameter werden beibehalten, sollte keine weitere Angabe zu Ihnen erfolgen.

Die Nischenströmung konnte mithilfe des Programmes erfolgreich simuliert werden. Es wurden für vier verschiedene Zeiten die Evolution der Flüssigkeit in der Zelle aufgenommen. Diese sind in Abbildung zu sehen.

Diese Ergebnisse entsprechen den Ergebnissen aus [3] und decken sich mit der Erfahrung. Zu Beginn sind deutliche Druckunterschiede zu beobachten. Im oberen rechten Bereich entsteht durch die Aufstauung der Flüssigkeit an der rechten Wand eine Druckerhöhung. Analog dazu lässt sich der Druck im linken Bereich durch einen Sog erklären. Im Laufe der Zeit gleichen sich die Druckunterschiede aus, da die Flüssigkeit inkompressibel ist. Im stationären Endzustand, der nach circa 3.5 s erreicht war, ist der Druck im gesamten Bereich minimal (hellblaue Farbe).

Das Geschwindigkeitsfeld bildet bereits kurz nach dem Start der Simulation einen Wirbel aus. Dieser ist zuerst, wie in Abbildung zu erkennen, sehr flach im oberen Bereich zu sehen. Mit der Zeit wird die gesamte träge Flüssigkeit aufgrund innerer Reibung beschleunigt. Der Wirbel wird größer bis er im Endzustand fast die gesamte Zelle einnimmt. Dabei verschiebt sich das Zentrum des Wirbels leicht nach rechts. Aufgrund der nicht rotationssymmetrischen Boxgeometrie kommt es zu Deformierungen des Wirbels an den Ecken der Zelle. In den unteren beiden Ecken der Zellen sind die Beiträge des Geschwindigkeitsfeldes zu gering, um eine nähere Untersuchung zuzulassen. Es lässt sich jedoch erkennen, dass der Wirbel dort abbricht.

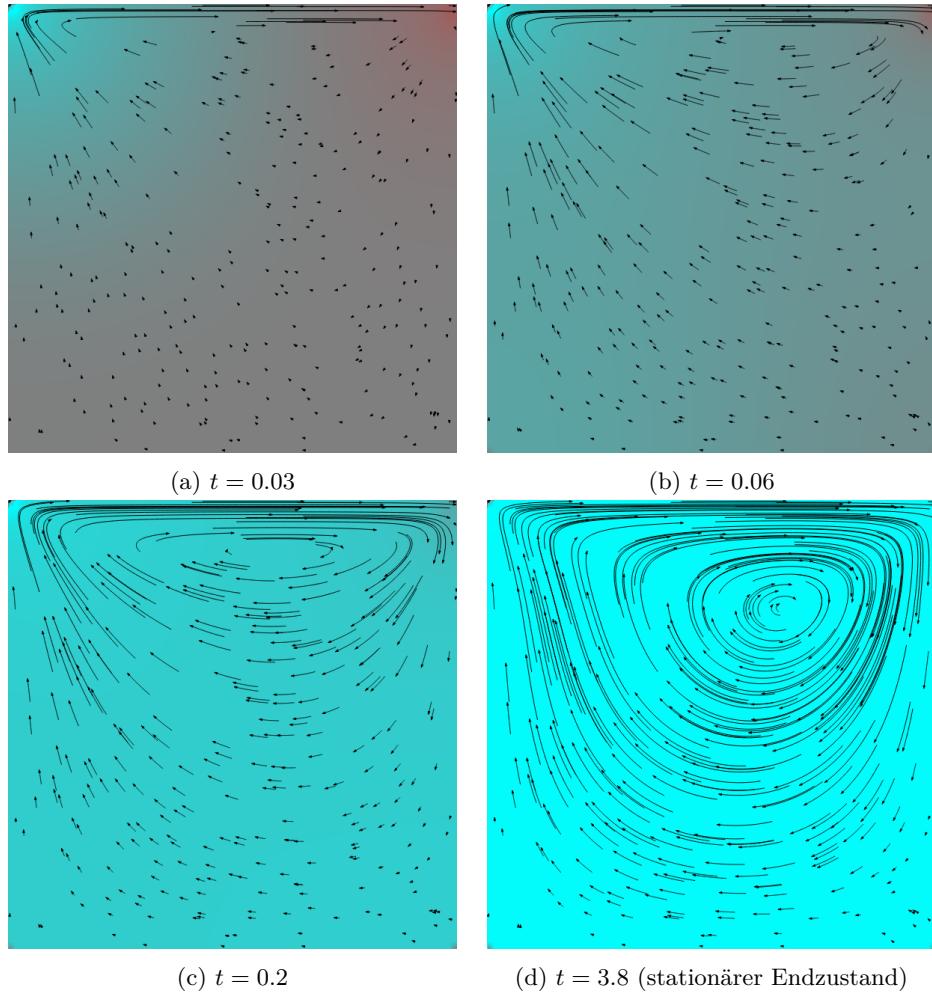


Abbildung 5: simulierte Zeitevolution des Geschwindigkeits- und Druckfeldes für $Re = 100$

In der rechten oberen Ecke verlassen einige Pfeile die Box. Dies würde den Haftbedingungen widersprechen. Jedoch sind diese Fehler auf numerische Rundungsfehler bei der Darstellung und endliche Ortsauflösung zurückzuführen, da die Gesamtlösung dadurch nicht verändert wird.

4.3 Einfluss der Reynoldszahl

Die Simulationen wurden nun mit steigender Reynoldszahl durchgeführt. Alle weiteren Parameter wurden konstant gelassen. Im Folgenden sollen nur noch die stationären Endzustände der Geschwindigkeitsfelder betrachtet werden.

Während für Reynoldszahlen von 1 bis 10 die Ergebnisse qualitativ gleich sind, erkennt eine deutliche Verschiebung und Deformierung des Wirbels ab

$Re = 100$. Kleinere Reynoldszahlen bedeuten eine höhere Viskosität. In den ersten beiden Fällen ist also die innere Reibung der Flüssigkeiten so groß, dass asymmetrische Turbulenzen verhindert werden.

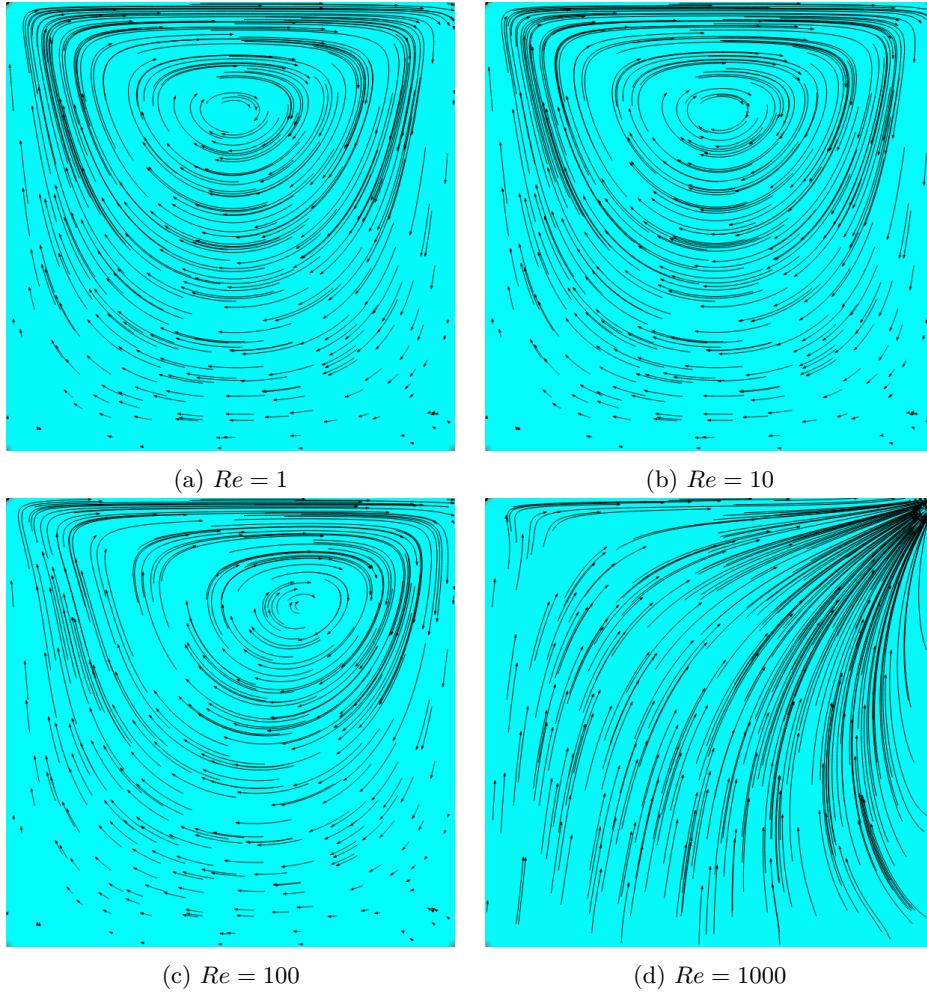


Abbildung 6: simulierte Strömungslinien im stationären Endzustand für verschiedene Reynoldszahlen

Im Falle von $Re = 1000$ ist deutlich erkennbar, dass ein Fehler in der Berechnung aufgetreten ist. Hier stößt man mit der Ortsauflösung von 64×64 Gitterpunkten und der daraus folgenden Zeitauflösung an die Grenzen der Simulation. In dem Programm wird eine 32-bit Auflösung der reellen Zahlen verwendet. Diese besitzt eine relative Genauigkeit von rund 10^{-6} . Mit einer 64-bit Auflösung könnte in diesem Falle nun ein besseres Ergebnis erzielt werden. Auch eine zu groß gewählte Randgeschwindigkeit kann zu einer numerischen Instabilität führen. Für $Re = 1000$ haben wir nun die Anzahl der Gitterpunkte auf 512×512 erhöht und die Geschwindigkeit auf 1.0 gesetzt. Es war somit möglich eine stabile numerische Berechnung

zu realisieren. Dies ist jedoch mit einem erheblich größeren Rechen- und Zeitaufwand verbunden. Die genauen Ergebnisse der Simulation sind in den Abbildungen 7 und 8 zu sehen. Diese Zeitevolution stimmt mit den Ergebnissen aus [3] ohne Einschränkungen überein (siehe Anhang).

4.4 Untersuchung von rechtwinkligen Geometrien

Im Folgenden werden wieder nur stationäre Zustände des Strömungsfeldes betrachtet. Dabei sei wieder $Re = 100$ und die Größe des Gitters 64×64 .

Die Simulationen für rechteckige Boxgeometrien sind in den Abbildungen 9 und 10 zu sehen. Es wird deutlich, dass sich durch die Wahl von $a \geq b$ die Lösung qualitativ kaum von der ursprünglichen Lösung unterscheidet. Der entstehende Wirbel scheint einfach mit dem Parameter a skaliert zu werden. Ein wesentlicher Unterschied besteht hier in der Deformation des Wirbels, welche aber eine logische Konsequenz aus der Änderung der Boxgeometrie ist.

Für den Fall $a < b$ ist aber nun in Abbildung 9 eine etwas andere Strömung zu sehen. Der gesamte Wirbel wird weder deformiert noch skaliert. Er behält in etwa seine Größe bei.

4.5 Periodische Anregung

Für diesen Versuchsteil wurde für die Randgeschwindigkeit die folgende periodische Funktion verwendet.

$$\cos 3t$$

Dabei beschreibt t wieder den Zeitparameter der Simulation. Die Ergebnisse sind in den Abbildungen 11 und 12 zu sehen.

Wie zu erwarten war wechseln die Drehrichtungen des Wirbels, je nachdem in welche Richtung die Randgeschwindigkeit zeigt. Es ist wichtig zu bemerken, dass beim Wechsel der Geschwindigkeitsrichtung ein enormer Druckanstieg zu beobachten ist.

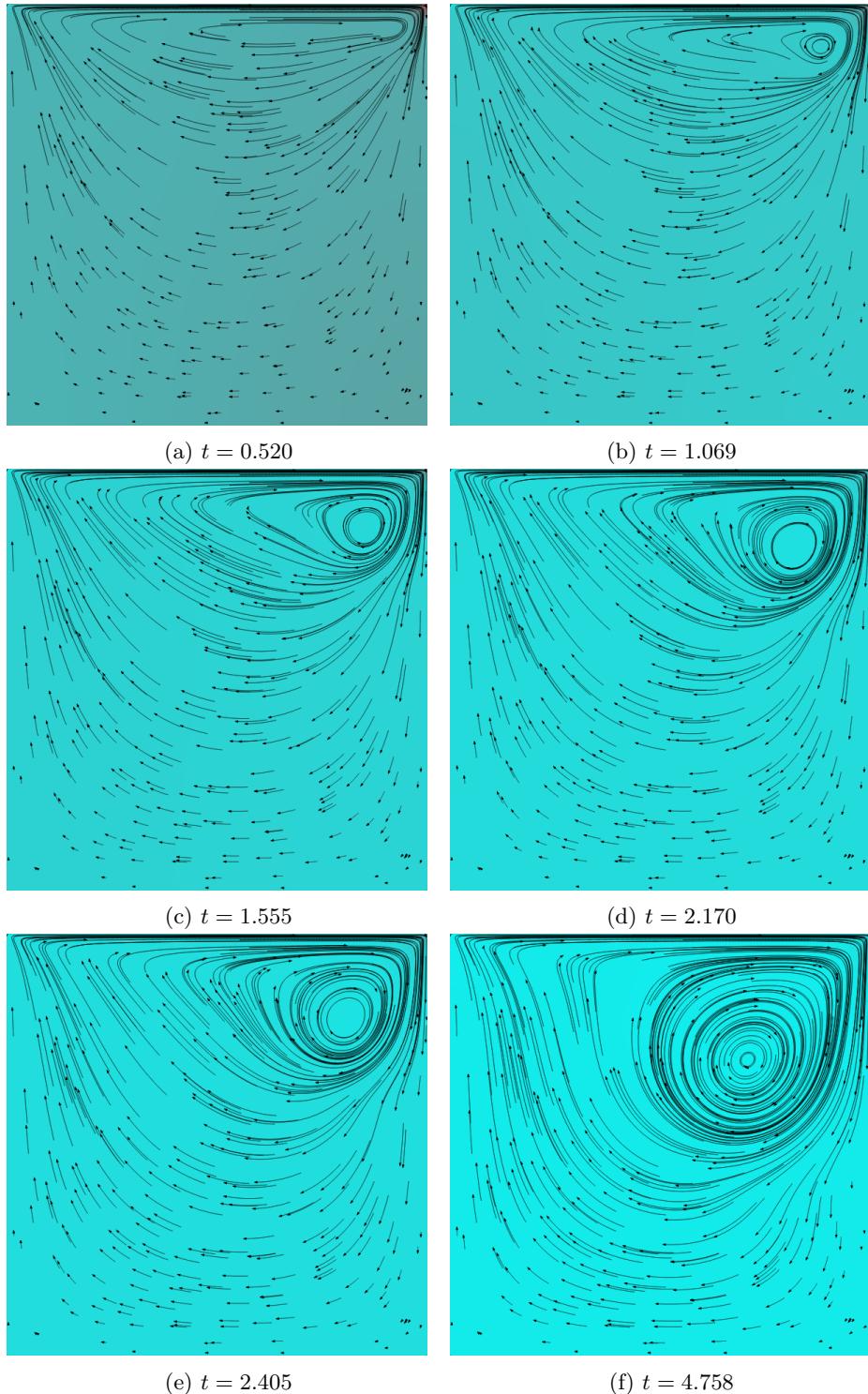


Abbildung 7: simulierte Zeitevolution der Strömungslinien und des Druckfeldes für $Re = 1000$ auf einem 512×512 -Gitter für verschiedene Zeitparameter t

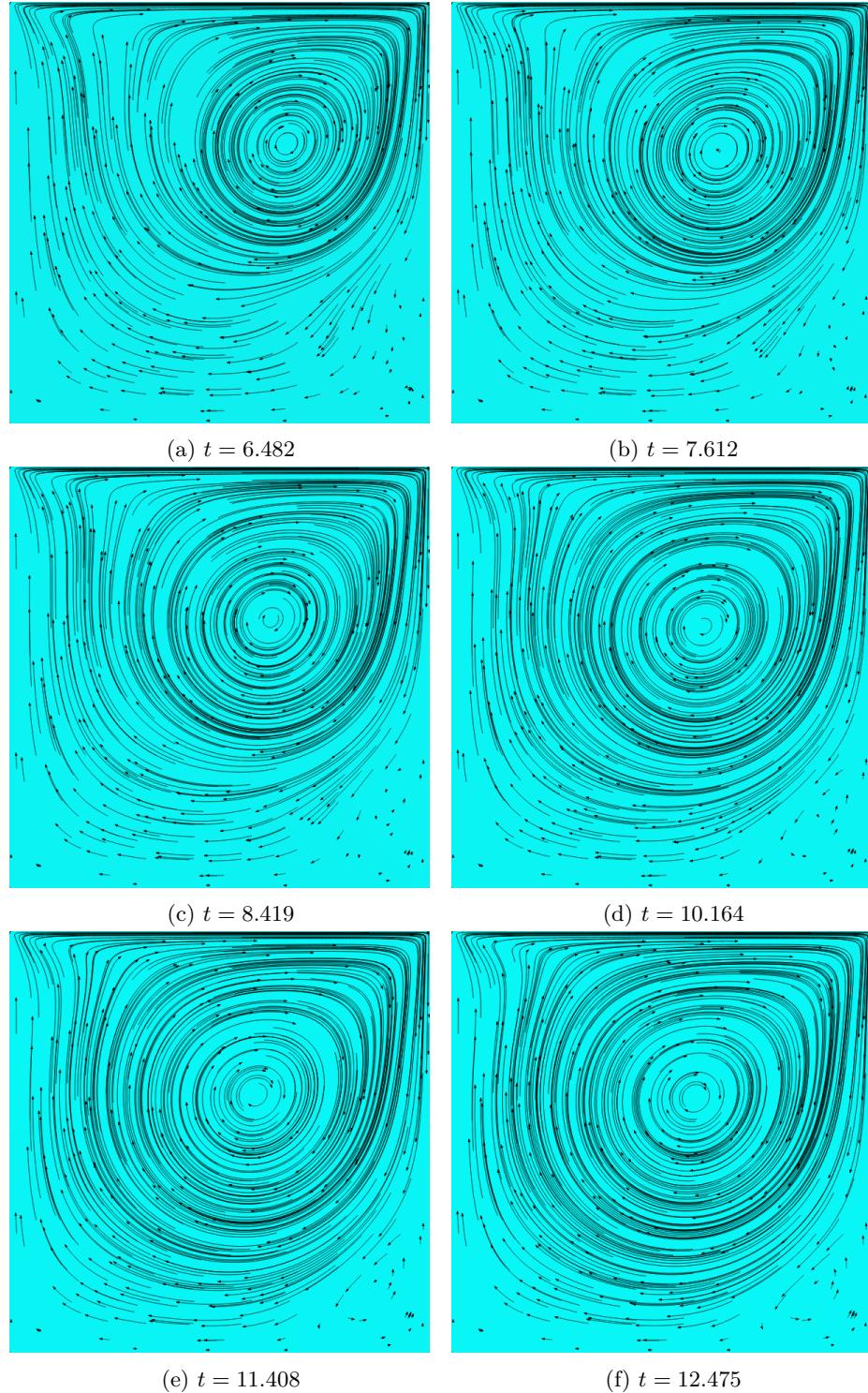


Abbildung 8: simulierte Zeitevolution der Strömungslinien und des Druckfeldes für $Re = 1000$ auf einem 512×512 -Gitter für verschiedene Zeitparameter t

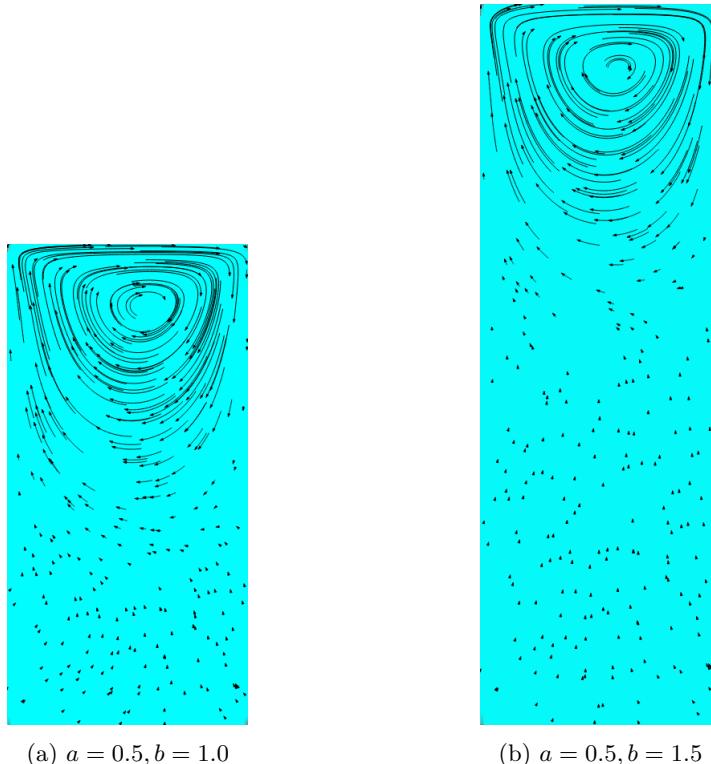


Abbildung 9: Simulation des stationären Strömungsfeldes für verschiedene Boxgeometrien

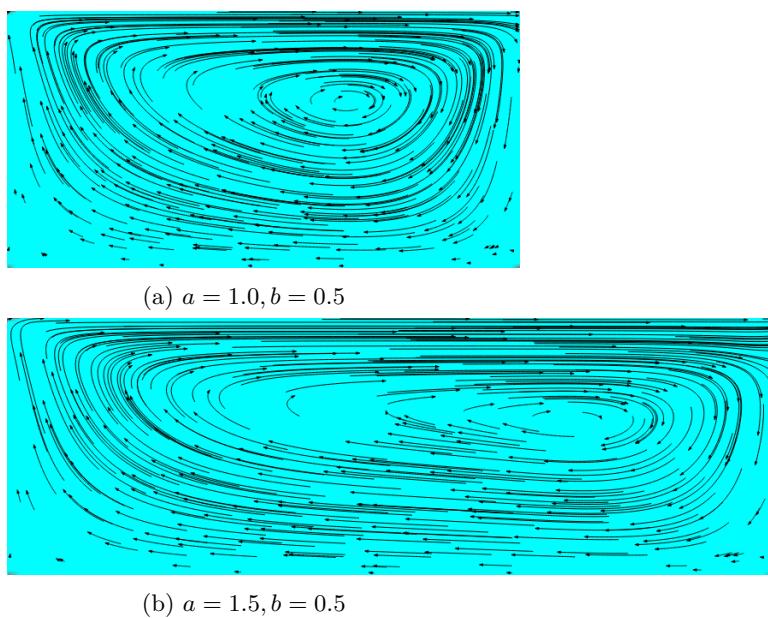


Abbildung 10: Simulation des stationären Strömungsfeldes für verschiedene Boxgeometrien

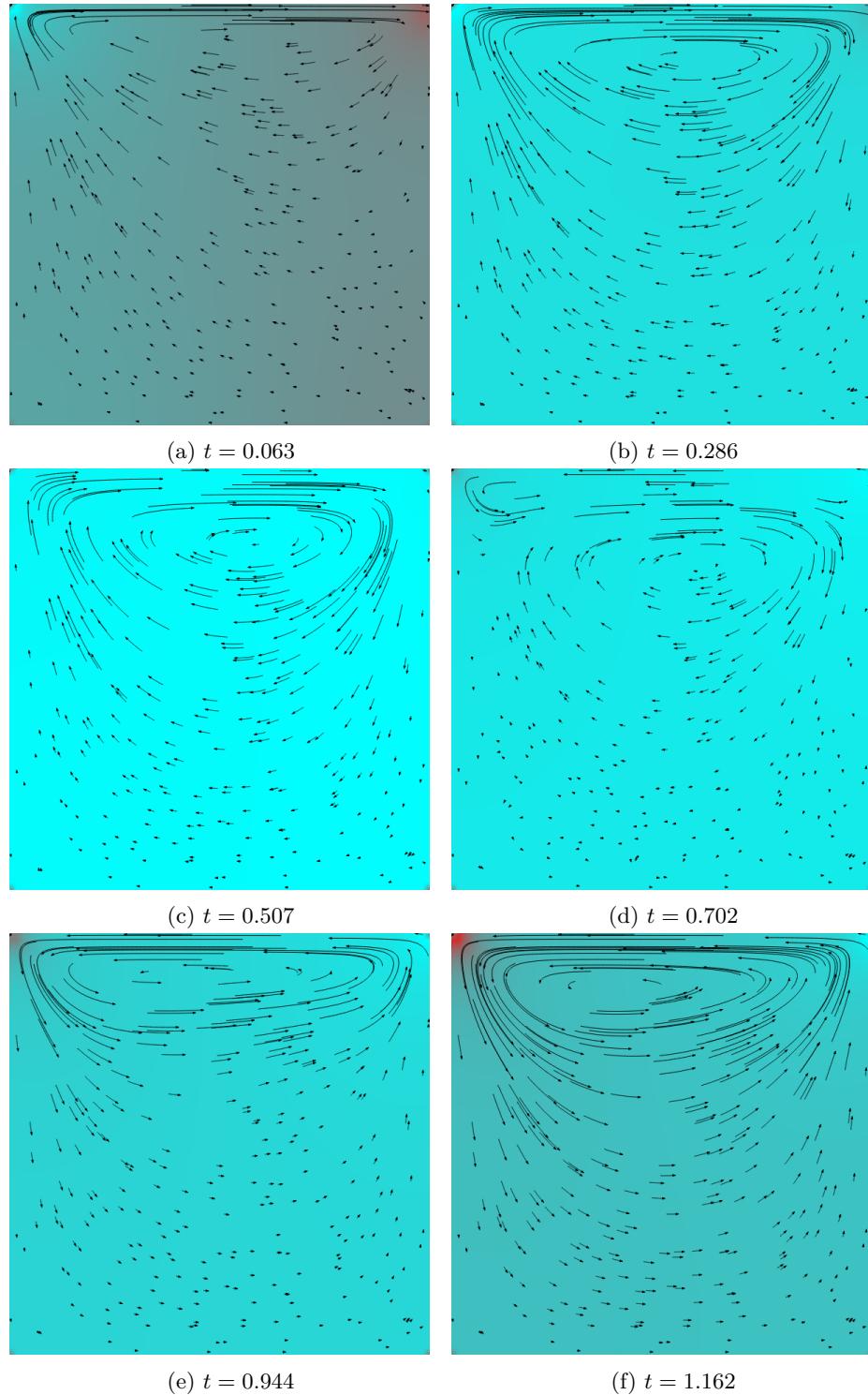


Abbildung 11: simulierte Zeitevolution Strömungslinien und des Druckfeldes, welche periodisch angeregt werden, für verschiedene Zeitparameter t

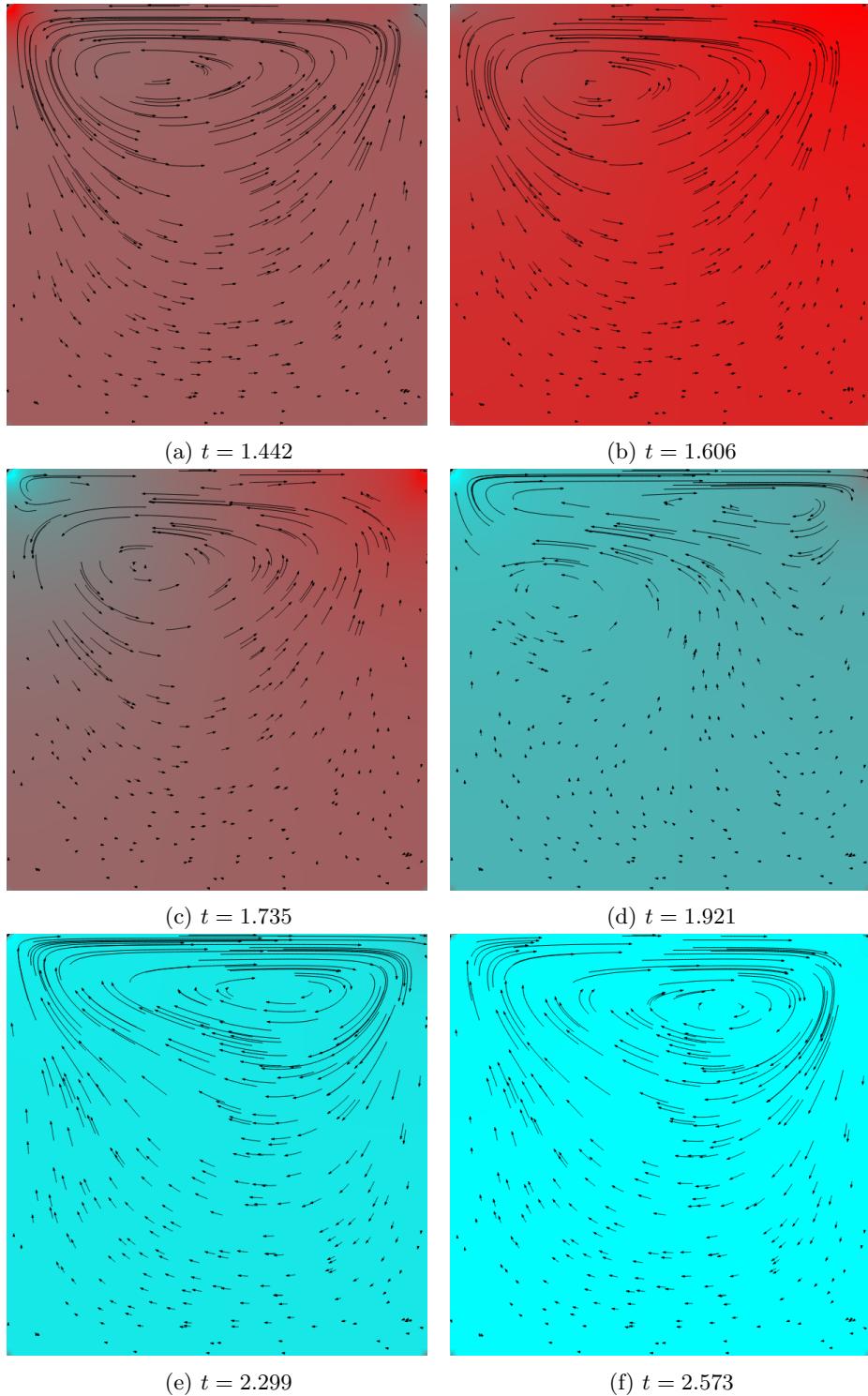


Abbildung 12: simulierte Zeitevolution Strömungslinien und des Druckfeldes, welche periodisch angeregt werden, für verschiedene Zeitparameter t

5 Zusammenfassung

Der Versuch diente dazu numerische Verfahren zu erlernen, um Differentialgleichungen wie die Navier-Stokes Gleichung für inkompressible Flüssigkeiten zu lösen und simulieren. Die hierfür verwendeten Techniken wie finite Differenzen, Donor-Cell-Stencils oder das SOR Verfahren lieferten gute Ergebnisse bei der Berechnung von Druck und Geschwindigkeit. Die damit simulierten Nischenströmungen deckten sich für kleine Reynoldszahlen (1 bis 100) sehr gut mit den Erwartungen aus der eigenen Erfahrung, sowie mit den bekannten Simulationen und Beispielen aus der Literatur. Auch konvergierte die Lösung relativ schnell, innerhalb weniger Sekunden gegen den stationären Endzustand, wenn man die Berechnung auf einem 64×64 Gitter durchführte. Für größere Reynoldszahlen wie z.B. $Re = 1000$ entstanden hier bereits beträchtliche Instabilitäten, die keine sinnvolle Simulation mehr möglich machten. Zur Berechnung derartiger Flüssigkeiten musste die Ortsauflösung stark erhöht werden, was zwar wieder zu guten Übereinstimmungen mit Literaturwerten führte, den Zeitaufwand aber natürlich erheblich ansteigen ließ. So dauerte das Errechnen des Endzustandes in diesem Fall schon über eine halbe Stunde in Echtzeit. Der höhere Rechenaufwand für größere Reynoldszahlen ist gewissermaßen eine notwendige Konsequenz, da mit der Erhöhung von Re die innere Reibung der Flüssigkeit abnimmt und damit die einzelnen Teilgebiete sich unabhängig von den Nachbarzellen entwickeln können, was zu mehreren kleinen Wirbeln und Turbulenzen führt, wie es ja auch in natürlichen, realen Strömungen der Fall ist. Die Simulationen, die unter veränderten Randbedingungen wie rechteckige Boxgeometrie oder zeitlich wechselnder Geschwindigkeit erstellt wurden, zeigten ebenfalls sehr gute Ergebnisse, die den Erwartungen für reale Flüssigkeiten entsprachen. Zusammenfassend kann man sagen, dass die numerische Behandlung der Navier-Stokes Gleichung wie sie hier durchgeführt wurde, ein sehr effizientes Verfahren zur Simulation des Verhaltens inkompressibler Flüssigkeiten darstellt. Mittels einfacher Rechentechnik können hiermit bereits nach kurzer Zeit für die meisten Fluide sehr gute Ergebnisse erzielt werden, die aufwändige Versuche in Wind- oder Strömungskanälen ersparen. Erweiterungen zum Simulieren hoher Reynoldszahlen oder kompressibler Substanzen lassen sich durch zusätzliche Terme in der Differentialgleichung und höhere Rechenleistung z.B. mittels Computercluster und paralleler Programmierung bewerkstelligen, wodurch das Verfahren universell einsetzbar wird.

Literatur

- [1] Durst. *Grundlagen der Strömungsmechanik*. 2006.
- [2] Ferziger/Perić. *Computational Methods for Fluid Dynamics*. 2002.
- [3] Griebel/Dornseifer/Neunhoeffer. *Numerical Simulation in Fluid Dynamics*. 1998.

A Quellcode

vec.h

```
#ifndef __VEC_H__
#define __VEC_H__

#include <math.h>

template <class T>
struct vec2{
    union{
        struct{ T x,y; };
        T v[2];
    };

    vec2(T vec_x = 0, T vec_y = 0): x(vec_x), y(vec_y) {}
    vec2(const T vec[2]): x(vec[0]), y(vec[1]) {}

    // get reference to array element
    T& operator [](int idx) { return v[idx]; }
    const T& operator [](int idx) const { return v[idx]; } // 
        const call

    // set vector from coordinates
    void set(T vec_x, T vec_y) { x = vec_x; y = vec_y; }

    // set both coordinates to equal value
    void set2(T value) { x = y = value; }

    // set vector from array
    void setv(const T vec[2]) { x = vec[0]; y = vec[1]; }

    // set vector from vector
    void setvec(const vec2<T> &vec) { x = vec.x; y = vec.y; }

    // normalize vector
    void setnorm() { const T tmp = 1.0f/mag(); x *= tmp; y *=
        tmp; }

    // negate vector
    void setneg() { x = -x; y = -y; }

    // set coordinates to invers
    void setinv() { x = 1/x; y = 1/y; }

    // cross product in 2d
    void setcross() { x = -y; y = x; }

    // set component product
}
```

```

void setcomp(const vec2<T> &vec) { x *= vec.x; y *= vec.y;
}

// set component division
void setcdiv(const vec2<T> &vec) { x /= vec.x; y /= vec.y;
}

// add vector
void setadd(const vec2<T> &vec) { x += vec.x; y += vec.y;
}
void operator +=(const vec2<T> &vec) { setadd(vec); }

// subtract vector
void setsub(const vec2<T> &vec) { x -= vec.x; y -= vec.y;
}
void operator -=(const vec2<T> &vec) { setsub(vec); }

// multiply with scalar value
void setmult(T value) { x *= value; y *= value; }
void operator *=(T value) { setmult(value); }

// divide by scalar value
void setdiv(T value) { x /= value; y /= value; }
void operator /=(T value) { setdiv(value); }

// set to maximum coordinates
void setmaxvec(const vec2<T> &vec) { x = ((x < vec.x)?(vec
    .x):(x)); y = ((y < vec.y)?(vec.y):(y)); }

// set to minimum coordinate
void setminvec(const vec2<T> &vec) { x = ((x > vec.x)?(vec
    .x):(x)); y = ((y > vec.y)?(vec.y):(y)); }

// get magnitude
T mag() const { return sqrt(x*x + y*y); }

// get squared magnitude
T sqmag() const { return x*x + y*y; }

// get product of all components
T prod() const { return x*y; }

// get normalized vector
vec2<T> norm() const { const T tmp = 1.0f/mag(); return
    vec2<T>(tmp*x, tmp*y); }

// get negative vector
vec2<T> neg() const { return vec2<T>(-x, -y); }
vec2<T> operator -() const { return neg(); }

// get vector of inverse values
vec2<T> inv() const { return vec2<T>(1/x, 1/y); }

```

```

// get dot product of vectors
friend T dot(const vec2<T> &vec_1, const vec2<T> &vec_2) {
    return vec_1.x*vec_2.x + vec_1.y*vec_2.y; }
friend T operator *(const vec2<T> &vec_1, const vec2<T> &
    vec_2) { return dot(vec_1, vec_2); }

// get cross product in 2d
friend vec2<T> cross(const vec2<T> vec) { return vec2<T>(-
    vec.y, vec.x); }

// get component product
friend vec2<T> comp(const vec2<T> &vec_1, const vec2<T> &
    vec_2) { return vec2<T>(vec_1.x*vec_2.x, vec_1.y*vec_2
    .y); }

// get component division
friend vec2<T> cdiv(const vec2<T> &vec_1, const vec2<T> &
    vec_2) { return vec2<T>(vec_1.x/vec_2.x, vec_1.y/vec_2
    .y); }

// add vectors
friend vec2<T> add(const vec2<T> &vec_1, const vec2<T> &
    vec_2) { return vec2<T>(vec_1.x+vec_2.x, vec_1.y+vec_2
    .y); }
friend vec2<T> operator +(const vec2<T> &vec_1, const vec2
    <T> &vec_2) { return add(vec_1, vec_2); }

// subtract vectors
friend vec2<T> sub(const vec2<T> &vec_1, const vec2<T> &
    vec_2) { return vec2<T>(vec_1.x-vec_2.x, vec_1.y-vec_2
    .y); }
friend vec2<T> operator -(const vec2<T> &vec_1, const vec2
    <T> &vec_2) { return sub(vec_1, vec_2); }

// multiply vector with scalar value
friend vec2<T> mult(T value, const vec2<T> &vec) { return
    vec2<T>(value*vec.x, value*vec.y); }
friend vec2<T> mult(const vec2<T> &vec, T value) { return
    mult(value, vec); }
friend vec2<T> operator *(T value, const vec2<T> &vec) {
    return mult(value, vec); }
friend vec2<T> operator *(const vec2<T> &vec, T value) {
    return mult(value, vec); }

// divide vector by scalar value
friend vec2<T> div(const vec2<T> &vec, T value) { return
    vec2<T>(vec.x/value, vec.y/value); }
friend vec2<T> operator /( const vec2<T> &vec, T value) {
    return div(vec, value); }

// get maximal vector

```

```

friend vec2<T> maxvec(const vec2<T> &vec_1, const vec2<T>
    &vec_2) { return vec2<T>((vec_1.x < vec_2.x)?(vec_2.x)
    :(vec_1.x), (vec_1.y < vec_2.y)?(vec_2.y):(vec_1.y));
}

// get minimal vector
friend vec2<T> minvec(const vec2<T> &vec_1, const vec2<T>
    &vec_2) { return vec2<T>((vec_1.x > vec_2.x)?(vec_2.x)
    :(vec_1.x), (vec_1.y > vec_2.y)?(vec_2.y):(vec_1.y));
}
};

typedef vec2<float> vec2f;
typedef vec2<double> vec2d;
typedef vec2<unsigned char> vec2b; // byte vector
typedef vec2<unsigned int> vec2u;
typedef vec2<int> vec2i;
typedef vec2<long long> vec2l;
typedef vec2<unsigned long long> vec2ul;

// template <class T, class U>
// vec2<T> toVec2<T>(vec2<U> vec) { return vec2<T>((T)vec.x,
// (T)vec.y); }

// template <class T, class U>
// vec2<T> operator T(const vec2<U> &vec) { return vec2<T>((
// T)vec.x, (T)vec.y); }

#endif // __VEC_H__

```

grid.h

```

#ifndef __GRID_H__
#define __GRID_H__

#include "vec.h"

template <class T>
class Grid2{
    public:
        // constructors
        Grid2(): _size(vec2u(2,2)), _min(vec2<T>(0,0)), _max(
            vec2<T>(1,1)), _scale(vec2<T>(1,1)), inv_scale(vec2<
            T>(1,1)) {}
        Grid2(vec2<T> grid_min, vec2<T> grid_max, vec2u
            grid_size): Grid2() {
            setSize(grid_size);
            setArea(grid_min, grid_max);

```

```
}

// set grid size in x, y direction - minimum is 2
void setSize(const vec2u &grid_size){
    _size = grid_size;
    _size.setmaxvec(vec2u(2,2));
    calcScale();
}

// set grid area
void setArea(const vec2<T> &grid_min, const vec2<T> &
    grid_max) {
    const vec2<T> tmp_max = maxvec(grid_max, grid_min);
    const vec2<T> tmp_min = minvec(grid_max, grid_min);
    const vec2<T> tmp = tmp_max - tmp_min;

    if (tmp.x == 0 || tmp.y == 0)
        return;

    _max = tmp_max;
    _min = tmp_min;
    calcScale();
}

// get grid dimension (size)
vec2u size() const { return _size; }
// get count of grid points
unsigned int count() const { return _size.prod(); }

// get parameters for grid area on plane
vec2<T> min() const { return _min; }
vec2<T> max() const { return _max; }
// get diameter vector of used area
vec2<T> diam() const { return _max-_min; }

// get distance between two grid points
vec2<T> scale() const { return _scale; }
// get inverse scale
vec2<T> invScale() const { return inv_scale; }

// get position in plane to given grid point
vec2<T> pos(const vec2u &idx_vec) const { return _min +
    comp(_scale, vec2<T>(idx_vec.x, idx_vec.y)); }
vec2<T> pos(unsigned int x, unsigned int y) const {
    return pos(vec2u(x,y)); }
vec2<T> pos(const vec2<T> &con_idx_vec) const { return
    _min + comp(_scale, c_idx_vec); }

// get continuous index of a (near) grid point to given
position in plane (no rounding)
```

```

vec2<T> conIdx(const vec2<T> &pos_vec) const { return
    comp(pos_vec - _min, inv_scale); }
vec2<T> idx(const vec2<T> &pos_vec) const { return
    conIdx(pos_vec); }

// get approximated grid point to given position in
// plane (normal rounding)
vec2u rIdx(const vec2<T> &pos_vec) const { const vec2<T>
    tmp = conIdx(pos_vec); return vec2u(tmp.x, tmp.y);
}

private:
// grid dimensions
vec2u _size;
// position and scale
vec2<T> _min;
vec2<T> _max;

// temporary vector for distance between grid points
vec2<T> _scale;
vec2<T> inv_scale;

// calculate temporary scale parameter
void calcScale(){
    const vec2u tmp = _size - vec2u(1,1);
    _scale = cdiv(_max-_min, vec2<T>((T)tmp.x, (T)tmp.y) );
    ;
    inv_scale = _scale.inv();
}
};

typedef Grid2<float> Grid2f;
typedef Grid2<double> Grid2d;

#endif // __GRID_H__

```

scalar-field.h

```

#ifndef __SCALAR_FIELD_H__
#define __SCALAR_FIELD_H__


template <class T>
class ScalarField2{
public:
    // constructors

```

```
ScalarField2(const Grid2<T> &field_grid): _grid(field_grid
    ), _field(0) { init(); }
// destructor
~ScalarField2() { delete[] _field; }

// get reference to vector at given grid point
T& operator ()(const vec2u &grid_point) { return _field[
    _grid.size().y * grid_point.x + grid_point.y]; }
T& operator ()(unsigned int x, unsigned int y) { return (*
    this)(vec2u(x,y)); }
const T& operator ()(const vec2u &grid_point) const {
    return _field[_grid.size().y * grid_point.x +
        grid_point.y]; }
const T& operator ()(unsigned int x, unsigned int y) const
{ return (*this)(vec2u(x,y)); }

// get reference to vector element for given linear
// coordinate
T& operator [](unsigned int idx) { return _field[idx]; }
const T& operator[](unsigned int idx) const { return
    _field[idx]; }

// get constant reference to grid
const Grid2<T>& grid() const { return _grid; }

// set current grid
void setGrid(const Grid2<T>& new_grid) { grid = new_grid;
    init(); }

// set field values
void set(const vec2u &grid_point, T value) { (*this)(
    grid_point) = value; }
void set(unsigned int x, unsigned int y, T value) { (*this
    )(x,y) = value; }

// get linear interpolated value of field for given
// position in plane
T operator ()(const vec2<T> &pos_vec) const{
    const vec2<T> idx = _grid.idx(pos_vec);
    const T i = floor(idx.x);
    const T j = floor(idx.y);

    if (idx.x <= 0 || idx.x >= _grid.size().x - 1 || idx.y
        <= 0 || idx.y >= _grid.size().y - 1)
        return (*this)(idx.x, idx.y);

    const T tmp_x1 = idx.x - i;
    const T tmp_x2 = 1 - tmp_x1;
    const T tmp_y1 = idx.y - j;
    const T tmp_y2 = 1 - tmp_y1;
```

```
        return tmp_x2 * tmp_y2 * (*this)(i,j)
            + tmp_x1 * tmp_y2 * (*this)(i+1,j)
            + tmp_x2 * tmp_y1 * (*this)(i,j+1)
            + tmp_x1 * tmp_y1 * (*this)(i+1,j+1);
    }

// get maximal magnitude of field values
T maxMag() const{
    T max_sqmag = 0;

    for (unsigned int i = 0; i < _grid.count(); i++){
        const T tmp = fabs(_field[i]);

        if (tmp > max_sqmag)
            max_sqmag = tmp;
    }

    return max_sqmag;
}

private:
// array with values for field
T *_field;
// grid of field
Grid2<T> _grid;

// initializing array of vector
void init(){
    if (_field != 0)
        delete[] _field;

    _field = new T[_grid.count()];

    for (unsigned int i = 0; i < _grid.count(); i++)
        _field[i] = 0;
};

typedef ScalarField2<float> ScalarField2f;
typedef ScalarField2<double> ScalarField2d;

#endif // __SCALAR_FIELD_H__
```

vec-field.h

```
#ifndef __VEC_FIELD_H__
#define __VEC_FIELD_H__


#include "vec.h"
#include "grid.h"


template <class T>
class VecField2{
public:
    // constructors
    VecField2(const Grid2<T> &field_grid): _grid(field_grid)
        , _field(0), _offset(vec2<T>()) { init(); }
    // destructor
    ~VecField2() { delete[] _field; }

    // get reference to vector at given grid point
    vec2<T>& operator ()(const vec2u &grid_point) { return
        _field[_grid.size().y * grid_point.x + grid_point.y
    ]; }
    vec2<T>& operator ()(unsigned int x, unsigned int y) {
        return (*this)(vec2u(x,y)); }
    const vec2<T>& operator ()(const vec2u &grid_point)
        const { return _field[_grid.size().y * grid_point.x
        + grid_point.y]; }
    const vec2<T>& operator ()(unsigned int x, unsigned int
    y) const { return (*this)(vec2u(x,y)); }

    // get reference to vector element for given linear
    // coordinate
    vec2<T>& operator [](unsigned int idx) { return _field[
        idx]; }
    const vec2<T>& operator[](unsigned int idx) const {
        return _field[idx]; }

    // get constant reference to grid
    const Grid2<T>& grid() const { return _grid; }

    // set current grid
    void setGrid(const Grid2<T>& new_grid) { grid = new_grid
        ; init(); }

    // set offset
    void setOffset(const vec2<T> &offset_vec) { _offset =
        offset_vec; }

    // set field values
    void set(const vec2u &grid_point, const vec2<T> &value)
        { (*this)(grid_point) = value; }
```

```
void set(unsigned int x, unsigned int y, const vec2<T> &
         value) { (*this)(x,y) = value; }

// get offset
vec2<T> offset() const { return _offset; }

// get linear interpolated value of field for given
// position in plane
vec2<T> operator ()(const vec2<T> &pos_vec) const{
    const vec2<T> idx = _grid.idx(pos_vec);

    const T i = floor(idx.x);
    const T j = floor(idx.y);

    if (idx.x <= 0 || idx.x >= _grid.size().x - 1 || idx.y
        <= 0 || idx.y >= _grid.size().y - 1)
        return (*this)(idx.x, idx.y);

    const T tmp_x1 = idx.x - i;
    const T tmp_x2 = 1 - tmp_x1;
    const T tmp_y1 = idx.y - j;
    const T tmp_y2 = 1 - tmp_y1;

    return tmp_x2 * tmp_y2 * (*this)(i,j)
        + tmp_x1 * tmp_y2 * (*this)(i+1,j)
        + tmp_x2 * tmp_y1 * (*this)(i,j+1)
        + tmp_x1 * tmp_y1 * (*this)(i+1,j+1);
}

// get maximal magnitude of field values
T maxMag() const{
    T max_sqmag = 0;

    for (unsigned int i = 0; i < _grid.count(); i++){
        const T tmp = _field[i].sqmag();

        if (tmp > max_sqmag)
            max_sqmag = tmp;
    }

    return sqrt(max_sqmag);
}

private:
    // array with values for field
    vec2<T> *_field;
    // grid of field
    Grid2<T> _grid;
    // offset in x and y direction for staggered grids
    vec2<T> _offset;
```

```
// initializing array of vector
void init(){
    if (_field != 0)
        delete[] _field;

    _field = new vec2<T>[_grid.count()];
}
};

typedef VecField2<float> VecField2f;
typedef VecField2<double> VecField2d;

#endif // __VEC_FIELD_H__
```

nsg.h

```
#ifndef __NSG_H__
#define __NSG_H__

#include "vec.h"
#include "grid.h"
#include "scalar-field.h"
#include "vec-field.h"
#include "utils.h"

#include <iostream>
#include <algorithm>

class NSG{
    public:
        NSG(const Grid2f &grid_def, const vec2f &g_vec, float
            re_number):
            grid(grid_def), p(grid_def), u(grid_def), v(grid_def),
            velocity(grid_def),
            F(grid_def), G(grid_def), RHS(grid_def), res(grid_def),
            g(g_vec), Re(re_number), Re_inv(1.0f/re_number), gamma
            (0.5f), omega2(0.5f), omega1(0.5f), t(0.0f), step(0)
            {
                epsilon = 0.001;
                dt_tmp = 0.5f * Re * (1.0f / grid.invScale().sqmag());
            }

        // initialize, set dirichlet boundaries
        void init() {}

        void calc(){
            for (unsigned j = 0; j < grid.size().x; j++) {
```

```

u(j,0) = 0.99f * cos(3.0f*t);
// nsg.u(j,grid.size().y-1) = 0.0f;
}

const float u_max = u.maxMag();
const float v_max = v.maxMag();

dt = fmin(grid.scale().x / u_max, grid.scale().y /
           v_max );
dt = 0.5 * fmin(dt, dt_tmp);
inv_dt = 1.0f / dt;

t += dt;
step += 1;

// calculate F,G
for (unsigned int i = 1; i < grid.size().x - 1; i++){
    for (unsigned int j = 1; j < grid.size().y - 1; j++)
    {
        const float F_tmp1 = Re_inv * ( (u(i+1,j) - 2*u(i,
                j) + u(i-1,j))/(grid.scale().x * grid.scale().
                x) + (u(i,j+1) - 2*u(i,j) + u(i,j-1))/(grid.
                scale().y * grid.scale().y) );
        const float F_tmp2 = grid.invScale().x * ( sq( 0.5
                f * (u(i,j) + u(i+1,j)) ) - sq( 0.5f * (u(i-1,
                j) + u(i,j)) ) + gamma * grid.invScale().x * 0.25f * (
                fabs(u(i,j) + u(i+1,j)) * (u(i,j) -
                u(i+1,j)) - fabs(u(i-1,j) + u(i,j)) *
                (u(i-1,j) - u(i,j)) );
        const float F_tmp3 = grid.invScale().y * 0.25f * (
                (v(i,j) + v(i+1,j))*(u(i,j) + u(i,j+1)) - (v(
                i,j-1) + v(i+1,j-1))*(u(i,j-1) + u(i,j)) )
                + gamma * grid.invScale().y * 0.25f * (
                fabs(v(i,j) + v(i+1,j))*(u(i,j) - u(
                i,j+1)) - fabs(v(i,j-1) + v(i+1,j-1))
                *(u(i,j-1) - u(i,j)) );
        F(i,j) = u(i,j) + dt * ( F_tmp1 - F_tmp2 - F_tmp3
                + g.x );
    }

    const float G_tmp1 = Re_inv * ( (v(i+1,j) - 2*v(i,
            j) + v(i-1,j))/(grid.scale().x * grid.scale().
            x) + (v(i,j+1) - 2*v(i,j) + v(i,j-1))/(grid.
            scale().y * grid.scale().y) );
    const float G_tmp2 = grid.invScale().y * ( sq( 0.5
            f * (v(i,j) + v(i,j+1)) ) - sq( 0.5f * (v(i,j-
            1) + v(i,j)) ) + gamma * grid.invScale().y * 0.25f * (
            fabs(v(i,j) + v(i,j+1)) * (v(i,j) -
            v(i,j+1)) - fabs(v(i,j-1) + v(i,j)) *
            (v(i,j-1) - v(i,j)) );
    const float G_tmp3 = grid.invScale().x * 0.25f * (
            (v(i,j) + v(i+1,j))*(u(i,j) + u(i,j+1)) - (u(
            i,j) + v(i+1,j))*(u(i,j) + u(i,j+1)) );
}

```

```
i-1,j) + u(i-1,j+1))*(v(i-1,j) + v(i,j)) )  
+ gamma * grid.invScale().x * 0.25f * (  
    fabs(u(i,j) + u(i,j+1))*(v(i,j) - v(  
        i+1,j)) - fabs(u(i-1,j) + u(i-1,j+1)  
       )*(v(i-1,j) - v(i,j)) );  
G(i,j) = v(i,j) + dt * ( G_tmp1 - G_tmp2 - G_tmp3  
+ g.y );  
  
RHS(i,j) = inv_dt * ( grid.invScale().x * (F(i,j)  
- F(i-1,j)) + grid.invScale().y * (G(i,j) - G(  
    i,j-1)) );  
}  
}  
  
float r;  
int step = 0;  
  
const float p_max = p.maxMag();  
  
do{  
    for (unsigned int i = 1; i < grid.size().x - 1; i++)  
    {  
        p(i,0) = p(i,1);  
        p(i, grid.size().y - 1) = p(i, grid.size().y - 2);  
    }  
  
    for (unsigned int j = 1; j < grid.size().y - 1; j++)  
    {  
        p(0,j) = p(1,j);  
        p(grid.size().x - 1, j) = p(grid.size().y - 2, j);  
    }  
  
    for (unsigned int i = 1; i < grid.size().x - 1; i++)  
    {  
        for (unsigned int j = 1; j < grid.size().y - 1; j  
            ++){  
            p(i,j) = omegal * p(i,j) + 0.5f * omega2 * (1.0f  
                / grid.invScale().sqmag()) * ( sq(grid.  
                    invScale().x) * (p(i+1,j) + p(i-1,j)) + sq(  
                        grid.invScale().y) * (p(i,j+1) + p(i,j-1)) -  
                        RHS(i,j));  
  
        }  
    }  
    r = 0.0f;  
    for (unsigned int i = 1; i < grid.size().x - 1; i++)  
    {  
        for (unsigned int j = 1; j < grid.size().y - 1; j  
            ++){  
            res(i,j) = sq(grid.invScale().x) * ( p(i+1,j) -  
                2*p(i,j) + p(i-1,j) ) + sq(grid.invScale().y
```

```
        ) * ( p(i,j+1) - 2*p(i,j) + p(i,j-1) ) - RHS
        (i,j);
    const float tmp = fabs(res(i,j));

    if (tmp > r)
        r = tmp;
    }

step += 1;

}while(r >= epsilon*p_max && step < 100);

for (unsigned int i = 1; i < grid.size().x - 1; i++){
    for (unsigned int j = 1; j < grid.size().y - 1; j++){
        {
            u(i,j) = F(i,j) - dt * grid.invScale().x * ( p(i
                +1,j) - p(i,j) );
            v(i,j) = G(i,j) - dt * grid.invScale().y * ( p(i,j
                +1) - p(i,j) );

            velocity(i,j) = vec2f(u(i,j),v(i,j));
        }
    }
}

// private:
public:
    ScalarField2f p;
    ScalarField2f u, v;
    VecField2f velocity;
    Grid2f grid;
    vec2f g;
    float Re;
    float Re_inv;
    float gamma;
    float epsilon;
    float omegal, omega2;
    float t;
    unsigned int step;

    // temporary
    // ScalarField2f tmp_u, tmp_v, tmp_p;
    ScalarField2f F,G,RHS, res;
    float dt, inv_dt, dt_tmp;
};

#endif // __NSG_H__
```

render-w.h

```
#ifndef __RENDER_W_H__
#define __RENDER_W_H__


// #include <vector>
#include <stdlib.h>

#include <QTimer>
#include <QWidget>
#include <QPainter>
#include <QKeyEvent>

#include "grid.h"
#include "vec-field.h"
#include "scalar-field.h"
#include "nsg.h"


class RenderW : public QWidget{
    Q_OBJECT

public:
    // constants
    static const float STEP_SIZE_PAR = 1.01f;

    // constructors
    RenderW(QWidget *parent = 0): vec_field(0), rand_pos(0),
        rand_pos_size(300) {
        timer = new QTimer(this);
        connect(timer, SIGNAL(timeout()), this, SLOT(update()))
    };
    timer->start(1);

    running = true;
}

// destructor
~RenderW(){
    delete timer;
    delete[] rand_pos;
}

// get pointer to current vector field
VecField2f* field() { return vec_field; }

// get count of random positions
unsigned int randPosSize() { return rand_pos_size; }
```

```
public slots:
    // set vector field currently showing
    void setVecField(VecField2f* field){
        if (field == 0)
            return;

        vec_field = field;
        pixel_grid.setArea(vec_field->grid().min(), vec_field
            ->grid().max());
        initRandPos();
        step_size = STEP_SIZE_PAR * vec_field->grid().scale().
            mag();
    }

    // set count of random positions
    void setRandPosSize(unsigned int count) { rand_pos_size
        = count; initRandPos(); }

protected:
    void paintEvent(QPaintEvent *event){
        if (vec_field == 0)
            return;

        if (running)
            nsg->calc();

        printf("%u: %f\n", nsg->step, nsg->t);

        QPainter painter(this);
        painter.setRenderHints(QPainter::Antialiasing, true);

        QImage map(width(), height(), QImage::Format_RGB32);
        const float inv_max_mag = 1.0f / (s->maxMag() + 0.001)
        ;

        for (unsigned int i = 0; i < width(); i++){
            for (unsigned int j = 0; j < height(); j++){
                const vec2f pos = pixel_grid.pos(i, j);
                const float col_par = 0.5f * ((*s)(pos) *
                    inv_max_mag + 1.0f);

                const QRgb value = qRgb(255.0f * col_par, 255.0f *
                    (1.0f - col_par), 255.0f * (1.0f - col_par));
                map.setPixel(i, j, value);
            }
        }

        painter.drawImage(QRect(0, 0, width(), height()), map,
            QRect(0, 0, width(), height()));
    }
```

```
for (unsigned int i = 0; i < rand_pos_size; i++) {
    // random position in plane
    vec2f pos = rand_pos[i];

    // construct vector path
    QPainterPath path;
    vec2f pixel_pos = pixel_grid.idx(pos);
    path.moveTo(pixel_pos.x, pixel_pos.y);

    for (int t = 0; t < 50; t++){
        pos += step_size * (*vec_field)(pos);

        if ( (pos.x <= vec_field->grid().min().x) || (pos.
            x >= vec_field->grid().max().x) || (pos.y <=
            vec_field->grid().min().y) || (pos.y >=
            vec_field->grid().max().y) )
            break;

        pixel_pos = pixel_grid.idx(pos);
        path.lineTo(pixel_pos.x, pixel_pos.y);
    }

    painter.setBrush(Qt::NoBrush);
    painter.drawPath(path);

    // draw arrow at the end of path
    QPainterPath arrow_path;
    vec2f dir = (*vec_field)(pos);
    dir.setnorm();
    const float par = 2.5f;

    arrow_path.moveTo(pixel_pos.x, pixel_pos.y);
    arrow_path.lineTo(pixel_pos.x - 1.5f*par*dir.x - 0.5
                      f*par*dir.y, pixel_pos.y - 1.5f*par*dir.y + 0.5f
                      *par*dir.x);
    arrow_path.lineTo(pixel_pos.x - 1.5f*par*dir.x + 0.5
                      f*par*dir.y, pixel_pos.y - 1.5f*par*dir.y - 0.5f
                      *par*dir.x);
    arrow_path.closeSubpath();

    painter.setBrush(Qt::black);
    painter.drawPath(arrow_path);
}

painter.end();
}

void resizeEvent(QResizeEvent *event) { pixel_grid.
    setSize(vec2u(width(), height())); }

void keyPressEvent(QKeyEvent *event){
    if (event->key() == Qt::Key_Space)
```

```
        running = !running;
    }

public:
    // grid of pixels over plane
    Grid2f pixel_grid;
    // vector field
    VecField2f *vec_field;
    // random position vector
    vec2f *rand_pos;
    unsigned int rand_pos_size;
    ScalarField2f *s;

    // temporary
    float step_size;

public:
    NSG *nsg;
    QTimer *timer;
    bool running;

private:
    // initializes random positions for vector arrows
    void initRandPos(){
        if (vec_field == 0)
            return;

        if (rand_pos != 0)
            delete[] rand_pos;

        rand_pos = new vec2f[rand_pos_size];

        for (unsigned int i = 0; i < rand_pos_size; i++){
            const float rx = (float)rand() / (float)RAND_MAX;
            const float ry = (float)rand() / (float)RAND_MAX;
            rand_pos[i] = comp(vec2f(rx, ry), vec_field->grid() .
                diam()) + vec_field->grid().min();
        }
    }
};

#endif // __RENDER_W_H__
```

main.cpp

```
#include <iostream>
#include <stdlib.h>
using namespace std;

#include <QApplication>

#include "vec-field.h"
#include "nsg.h"
#include "render-w.h"

int main(int argc, char *argv[]){
    // srand(time(NULL));

    const vec2f min(0.0f, 0.0f);
    const vec2f max(1.0f, 1.0f);
    const vec2u size(1<<6, 1<<6);
    const Grid2f grid(min, max, size);

    VecField2f v(grid);
    ScalarField2f s(grid);

    for (unsigned int i = 0; i < v.grid().size().x; i++){
        for (unsigned int j = 0; j < v.grid().size().y; j++){
            vec2f offset;
            // offset = vec2f(0.5f, 0.5f);

            const vec2f pos = v.grid().pos(i, j) - offset;

            // v(i, j) = vec2f(pos.x, sqrt(pos.x) * sin(3*M_PI*pos.x) * sin(M_PI*pos.y));
            // v(i, j) = vec2f(pos.x*pos.x + pos.y*pos.y, pos.x*pos.x - pos.y*pos.y);
            s(i, j) = sqrt(pos.x) * sin(3*M_PI*pos.x) * sin(M_PI*pos.y);
            // s(i, j) = cos(10.0f * pos.x) * cos(10.0f * pos.y);
            // v(i, j) = vec2f(-pos.x*pos.y, pos.x*pos.x + pos.y*pos.y);
            // v(i, j) = vec2f(-pos.y, pos.x);
        }
    }

    NSG nsg(grid, vec2f(), 100);

    //
    for (unsigned j = 0; j < grid.size().x; j++){
        nsg.u(j, 0) = 1.0f;
        nsg.u(j, grid.size().y-1) = 0.0f;
    }
    for (unsigned j = 0; j < grid.size().y; j++){
        nsg.v(0, j) = 0.0f;
        nsg.v(grid.size().x-1, j) = 0.0f;
    }
}
```

```
nsg.v(0,j) = 0.0f;
nsg.v(grid.size().x-1, j) = 0.0f;
}

for (unsigned int i = 0; i < grid.size().x; i++){
    for (unsigned int j = 0; j < grid.size().y; j++){
        nsg.p(i,j) = 0.0f;
    }
}

// float t = 0.0f;

// for (int i = 0; i < 1000; ++i){
//     nsg.calc();
//     cout << (t+=nsg.dt) << "s" << endl;
// }

QApplication app(argc, argv);

RenderWindow *render = new RenderW;
render->resize(max.x * 600, max.y * 600);
render->setVecField(&(nsg.velocity));
// render->setVecField(&v);
render->nsg = &nsg;
render->s = &(nsg.p);
render->show();

app.exec();

delete render;

return 0;
}
```

config.pro

```
TARGET = prg.exe

TEMPLATE = app
CONFIG += qt thread warn_on release
QT = core gui

greaterThan(QT_MAJOR_VERSION, 4) {
    QT += widgets
}

DEFINES +=
QMAKE_CXXFLAGS += -O2 -fPIC -restrict -fno-fnalias -fno-
rtti -fno-exceptions -static -std=c++11
QMAKE_LFLAGS +=
```

```
INCLUDEPATH +=  
  
LIBS +=  
  
HEADERS = \  
    ../../src/render-w.h \  
    ../../src/vec.h \  
    ../../src/grid.h \  
    ../../src/vec-field.h \  
    ../../src/scalar-field.h \  
    ../../src/nsg.h \  
    ../../src/utils.h  
  
SOURCES = ../../src/main.cpp  
  
unix:OBJECTS_DIR      = obj  
unix:MOC_DIR           = moc  
  
unix:QMAKE_POST_LINK=strip $(TARGET)
```

B Literaturbeispiele

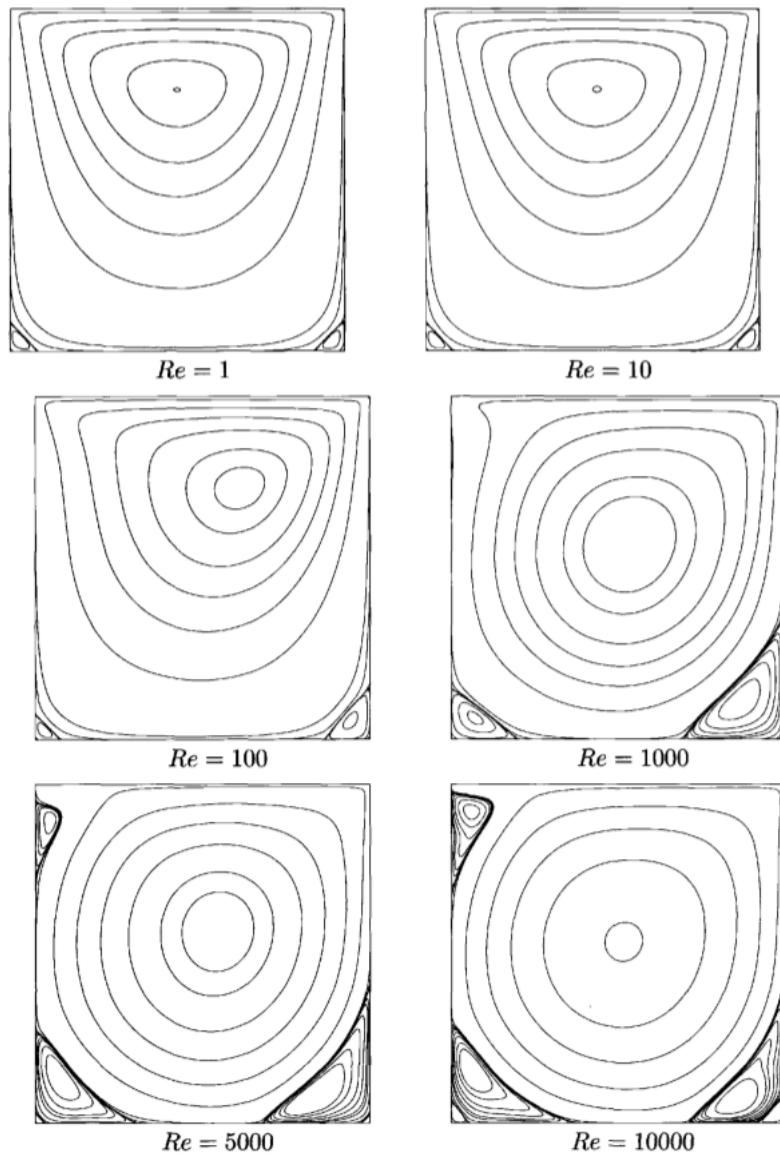


FIG. 5.7. *Driven cavity, steady state streamlines at different Reynolds numbers.*

Abbildung 13: Simulation des stationären Geschwindigkeitsfeldes für verschiedene Reynoldszahlen
Quelle: [3]

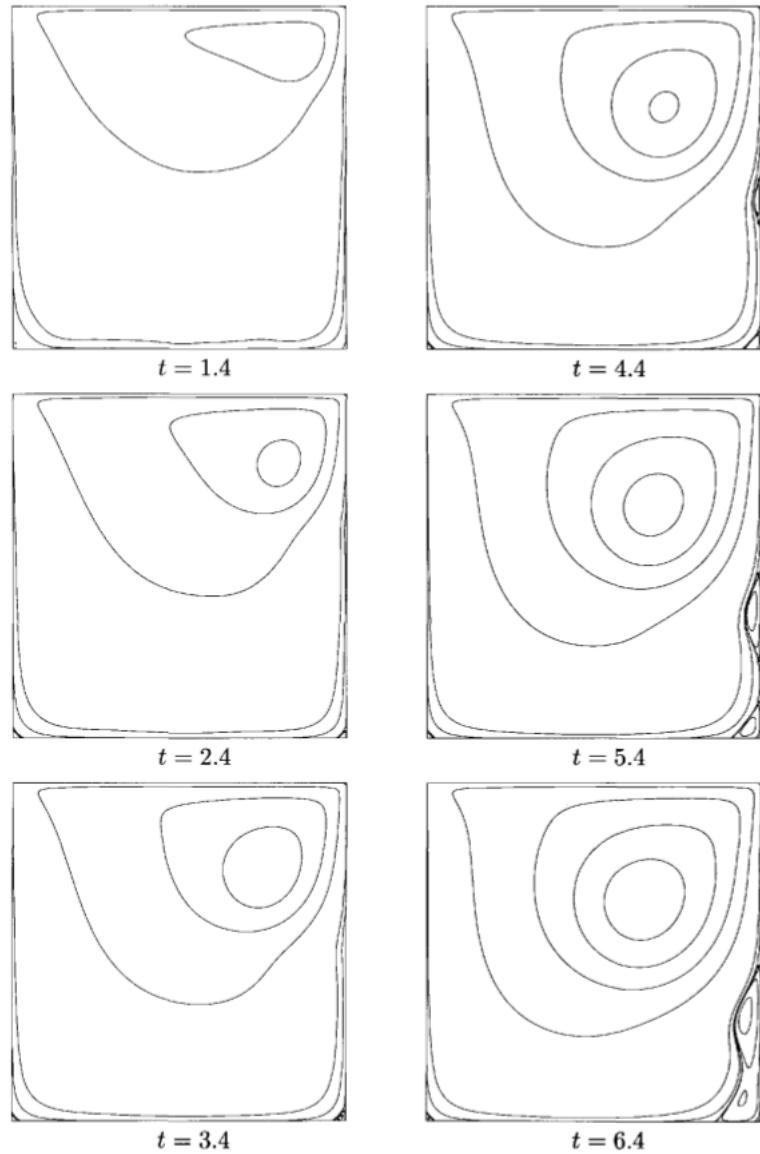


FIG. 5.4. *Cavity flow, streamlines, time evolution at $Re = 1000$.*

Abbildung 14: Zeitevolution des Geschwindigkeitsfeldes für $Re = 1000$
Quelle: [3]

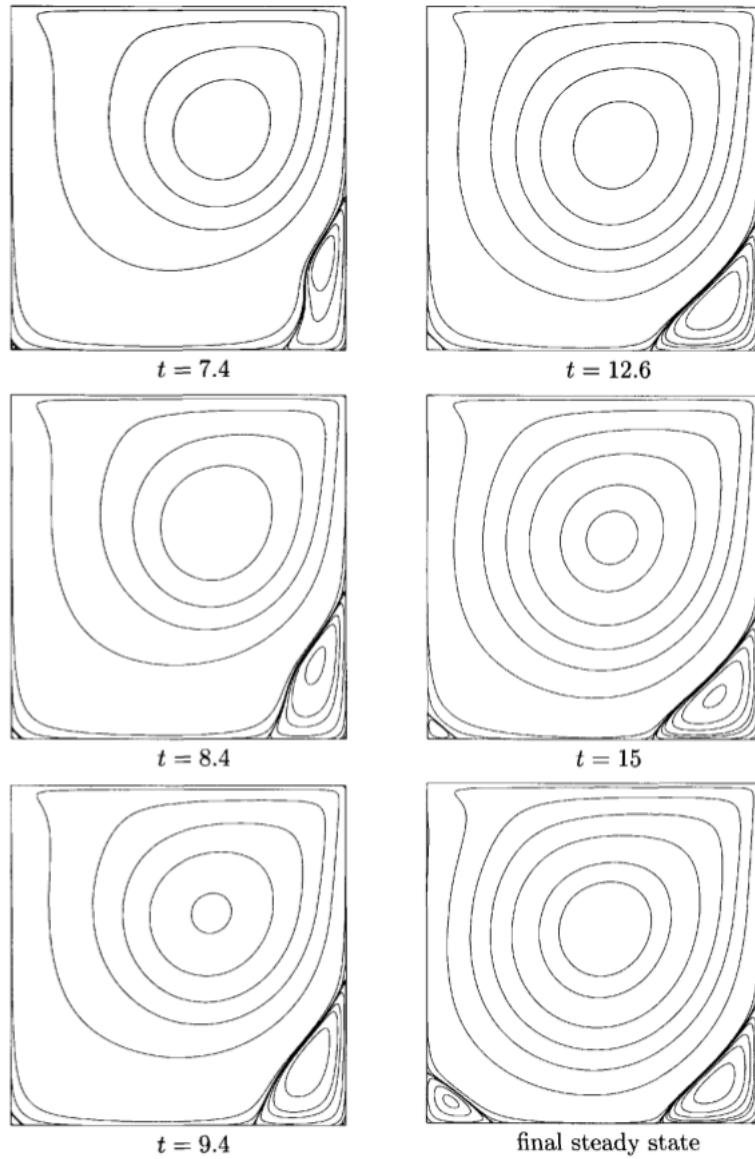


FIG. 5.5. *Driven cavity, streamlines, time evolution at $Re = 1000$.*

Abbildung 15: Zeitevolution des Geschwindigkeitsfeldes für $Re = 1000$
Quelle: [3]