NUSS Projekt 3: Programmieren eines Finite Volumen Codes auf kartesischen Gittern zur Lösung der 2D Eulergleichungen

Michael Dumbser, Sabine Roller

26. Januar 2005

1 Aufgabenstellung und Lehrziele

1.1 Aufgabenstellung

Nach dem Programmieren der Riemann-Löser in Teilprojekt 2 sollen diese nun in einen vollständigen 2D Code integriert werden, damit das Zusammenspiel der einzelnen Teile eines Finite Volumen Codes klar wird. Wenngleich die Beschränkung auf kartesische Gitter eine deutliche Vereinfachung darstellt, so sind doch die wesentlichen Elemente schon klar ersichtlich:

- Riemannlöser
- Flussbilanzierung
- Zeitintegration
- Rekonstruktion und Limitierung für die 2. Ordnung
- Setzen von Randbedingungen

Das Grundgerüst des Codes wird bereits vorgegeben, so dass Routineaufgaben wie die Ausgabe der Daten in ein mit IBM Open DX kompatibles Format, sowie die Initialisierung der Anfangsbedingung nicht mehr von den Studierenden zu erledigen sind. Ausserdem wird so die Struktur des Codes, als auch ein fester Standard für die Schnittstellen zwischen den einzelnen Unterprogrammen festgelegt. Als Anwendungsbeispiel wird nach der Validierung des Codes durch einen 2D Gausspuls die Richtmyer-Meshkov Instabilität berechnet.

1.2 Lehrziele

- Programmieren eines 2D Finite Volumen Codes in Fortran 90
- Validierung des selbsterstellten Codes
- Teamarbeit

1.3 Benötigte Software

- Linux
- Emacs
- Intel Fortran 90 Compiler
- IBM Open DX

2 Grundkonzeption und Validierung des Codes Euler2D

2.1 Benötigte Zusammenhänge

2.1.1 Diskretisierung der Grundgleichungen

Die zweidimensionalen Eulergleichungen lauten

$$u_t + f_1(u)_x + f_2(u)_y = 0 \tag{1}$$

mit

$$u = \begin{pmatrix} \rho \\ \rho v_1 \\ \rho v_2 \\ e \end{pmatrix}, \qquad f_1 = \begin{pmatrix} \rho v_1 \\ \rho v_1^2 + p \\ \rho v_1 v_2 \\ v_1(e+p) \end{pmatrix} \qquad f_2 = \begin{pmatrix} \rho v_2 \\ \rho v_1 v_2 \\ \rho v_2^2 + p \\ v_2(e+p) \end{pmatrix} \tag{2}$$

Zustandsgleichung $\qquad p = (\gamma - 1)\rho\varepsilon, \qquad e = \rho\varepsilon + \frac{1}{2}\rho(v_1^2 + v_2^2)$

Enthalpie $\qquad H = \frac{e+p}{\rho}$

Diskretisiert man obige Gleichungen mit einem Verfahren in Erhaltungsform (Finite Volumen), so ergibt sich folgendes numerische Schema mit den numerischen Flüssen g und h:

$$u_{ij}^{n+1} = u_{ij}^n - \frac{\Delta t}{\Delta x}\left(g_{i+1/2,j} - g_{i-1/2,j}\right) - \frac{\Delta t}{\Delta y}\left(h_{i,j+1/2} - h_{i,j-1/2}\right) \tag{3}$$

Durch Anwendung der linearen Transformation

$$f_2(u) = M \cdot f_1(M^{-1}u) \tag{4}$$

mit der Transformationsmatrix

$$M = \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & +1 & 0 & 0 \\ 0 & 0 & 0 & +1 \end{pmatrix} \tag{5}$$

lassen sich der Fluss f_2 und der numerische Fluß h aus den Flüssen f_1 bzw. g einfach berechnen.

Für ein Verfahren erster Ordnung kann direkt der Riemannlöser benutzt werden, für Verfahren höherer Ordnung im Raum sind die Zustände an der Zellkante rechts und links geeignet zu rekonstruieren (MUSCL Technik mit Limiter).

2.1.2 Verfahren 2. Ordnung in Raum und Zeit: MUSCL

Für das Verfahren 1. Ordnung wird in jeder Zelle ein konstanter Zustand angenommen, der von einer Zelle zur nächsten springt. An der Zellkante wird dann ein Riemann-Problem gelöst und damit der numerische Fluß berechnet:

$$g_{i+1/2,j} = g(u^n_{i,j}, u^n_{i+1,j}), \qquad h_{i,j+1/2} = h(u^n_{i,j}, u^n_{i,j+1}) \tag{6}$$

Nimmt man an, daß die Größen innerhalb der Zelle nicht konstant, sondern bilinear (d.h. linear in beide Richtungen) verteilt sind, dann hat man am rechten und linken bzw. am oberen und unteren Zellrand andere Werte, die in den Riemann-Löser eingehen. Der numerische Fluß berechnet sich dann aus

$$g_{i+1/2,j} = g(u^n_{i+}, u^n_{(i+1)-}), \qquad h_{i,j+1/2} = h(u^n_{j+}, u^n_{(j+1)-}) \tag{7}$$

Dafür benötigt man zunächst die Steigungen sx und sy in x- bzw. y-Richtung. Man erhält sie aus den einseitigen Ableitungen $l_{i,j}$, $r_{i,j}$ in x-Richtung, $u_{i,j}$, $o_{i,j}$ in y-Richtung:

$$l_{i,j} = \frac{u^n_{i,j} - u^n_{i-1,j}}{\Delta x} \qquad u_{i,j} = \frac{u^n_{i,j} - u^n_{i,j-1}}{\Delta y} \tag{8}$$

$$r_{i,j} = \frac{u^n_{i+1,j} - u^n_{i,j}}{\Delta x} \qquad o_{i,j} = \frac{u^n_{i,j+1} - u^n_{i,j}}{\Delta y} \tag{9}$$

Aus diesen Steigungen muß eine Auswahl getroffen werden, die die TVD-Eigenschaft erhält. Dies bekommt man durch eine Limiter-Funktion wie z.B. minmod:

$$minmod(a,b) = \begin{cases} a, & \text{falls } |a| < |b| \text{ und } ab > 0, \\ b, & \text{falls } |a| \geq |b| \text{ und } ab > 0, \\ 0, & \text{falls } ab \leq 0. \end{cases} \tag{10}$$

Die Steigungen sx und sy sind damit:

$$sx_{i,j} = minmod(l_{i,j}, r_{i,j}) \qquad sy_{i,j} = minmod(u_{i,j}, o_{i,j}) \tag{11}$$

Damit kann man die Werte vom Zellmittelpunkt an den Rand extrapolieren:

$$u^n_{i\pm} = u^n_{(i,j)} \pm \frac{\Delta x}{2} sx_{i,j} \quad \text{bzw.} \quad u^n_{j\pm} = u^n_{(i,j)} \pm \frac{\Delta y}{2} sy_{i,j} \tag{12}$$

Damit hat man bereits ein Verfahren 2. Ordnung im Raum. Um die 2. Ordnung auch in der Zeit zu bekommen, geht man prinzipiell genauso vor, man extrapoliert vom Zeitpunkt t^n auf den Zeitpunkt $t^{n+1/}$:

$$u^{n+1/2}_{(i,j)\pm} = u^n_{(i,j)\pm} + \frac{\Delta t}{2} u_t \tag{13}$$

In der Zeit kann man aber keine Steigung berechnen, da man nur die Werte zu einem Zeitpunkt hat. Man behilft sich, indem man die Zeitableitung durch Raumableitungen ersetzt:

$$u_t = -f_1(u)_x - f_2(u)_y \tag{14}$$

und damit

$$u^{n+1/2}_{(i,j)\pm} = u^n_{(i,j)\pm} - \frac{1}{2}\frac{\Delta t}{\Delta x}[f_1(u^n_{i+}) - f_1(u^n_{i-})] - \frac{1}{2}\frac{\Delta t}{\Delta y}[f_2(u^n_{j+}) - f_2(u^n_{j-})] \tag{15}$$

Diese Werte gehen nun in den Riemann-Löser ein und man bekommt die numerischen Flüsse

$$g_{i+1/2,j} = g(u^{n+1/2}_{i+}, u^{n+1/2}_{(i+1)-}), \qquad h_{i,j+1/2} = h(u^{n+1/2}_{j+}, u^{n+1/2}_{(j+1)-}) \tag{16}$$

2.1.3 Bestimmung der Zeitschrittweite

Die CFL Zeitschrittbedingung zur Gewährleistung einer stabilen Diskretisierung lautet:

$$\Delta t = CFL \cdot \frac{\min(\Delta x, \Delta y)}{\max(|c| + |v|)} \tag{17}$$

$$CFL < \frac{1}{2} \tag{18}$$

2.1.4 Transformation der konservativen und primitiven Variablen

Die Umwandlung der konservativen Variablen $[\rho, \rho u, \rho v, e]^T$ in primitive Variable $[\rho, u, v, p]^T$ funktioniert unter Zuhilfenahme der Zustandsgleichung wiefolgt:

$$\rho = \rho \tag{19}$$

$$u = \frac{\rho u}{\rho} \tag{20}$$

$$v = \frac{\rho v}{\rho} \tag{21}$$

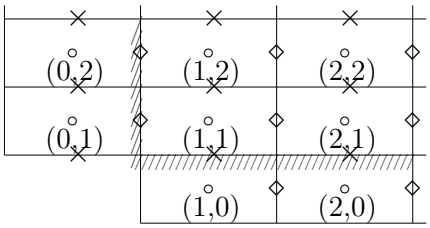
$$p = (\gamma - 1)\left(e - \frac{1}{2\rho}\left((\rho u)^2 + (\rho v)^2\right)\right) \tag{22}$$

Die umgekehrte Transformation lautet:

$$\rho = \rho \tag{23}$$
$$\rho u = \rho \cdot u \tag{24}$$
$$\rho v = \rho \cdot v \tag{25}$$
$$e = \frac{p}{\gamma - 1} + \frac{\rho}{2}\left(u^2 + v^2\right) \tag{26}$$

2.1.5 Setzen von Randbedingungen

Um Randbedingungen einzuhalten, werden um das Rechengebiet sogenannte Ghostcells gelegt, in denen Werte so vorgeschrieben werden, daß auf dem Rand die verlangten Bedingungen gelten.



Man kann die Werte der inneren Zellen in die Ghostzellen extrapolieren,

$$u_{0,j} = u_{1,j} \qquad u_{IMAX+1,j} = u_{IMAX,j} \tag{27}$$
$$u_{i,0} = u_{i,1} \qquad u_{i,JMAX+1} = u_{i,JMAX} \tag{28}$$

Hat man ein Beispiel, bei dem man nur einen Ausschnitt aus einem periodisch wiederkehrenden Verhalten simuliert, setzt man periodische Randbedingungen, indem man die inneren Werte vom gegenüberliegenden Rand kopiert:

$$u_{0,j} = u_{IMAX,j} \qquad u_{IMAX+1,j} = u_{1,j} \tag{29}$$
$$u_{i,0} = u_{i,JMAX} \qquad u_{i,JMAX+1} = u_{i,1} \tag{30}$$

Dies muß man für alle Größen machen, insbesondere auch für die Gradienten! Andernfalls würde das Verfahren am Rand auf 1. Ordnung zurückschalten.

2.2 Modularer Aufbau

Der Code Euler2D ist in Fortran 90 programmiert und nutzt die Möglichkeit des modularen Programmierens, welches im Vergleich zur klassischen Programmierweise Sicherheitsvorteile im Umgang mit Subroutinen und Variablen durch die PRIVATE und PUBLIC Merkmale bietet. Der Code beinhaltet die folgenden Dateien:

- **Euler2D.f90**: Hauptprogramm, welches vom Benutzer an der Shell aufgerufen wird. Wird den Studierenden fertig zur Verfügung gestellt.
- **TypesDef.f90**: Hier werden die Datentypen festgelegt. Wird zur Verfügung gestellt, um eine einheitliche Schnittstelle zwischen den verschiedenen Subroutinen herzustellen.
- **IniEuler2D.f90**: Subroutine zur Initialisierung der Simulationsparameter; wird zur Verfügung gestellt und kann von den Studierenden je nach Problemstellung verändert werden
- **CreateMesh.f90**: Erstellt das kartesische Gitter. Wird fertig zur Verfügung gestellt.
- **InitialCondition.f90**: Setzt die Anfangsbedingung je nach Problemstellung. Wird fertig zur Verfügung gestellt.
- **PrimCons.f90**: Umwandlung der konservativen in primitive Variable. Der Modul-Header wird bereitgestellt, die Funktion ist zu programmieren.
- **CalcTimeStep.f90**: Berechnet den Zeitschritt aus der CFL Bedingung. Der Modul-Header wird bereitgestellt, die Funktion ist zu programmieren.
- **DataOutput.f90**: Schreibt die Feldgrössen in einem von Tecplot lesbaren Format in eine Datei. Wird vorgegeben.
- **TimeDisc.f90**: Hier erfolgt die Zeitdiskretisierung.
- **CloseEuler2D.f90**: Gibt den Speicherplatz wieder frei. Wird vorgegeben.
- **SetBoundaryCondition.f90**: Routine zum Setzen der Randbedingungen. Ist von den Studierenden zu programmieren.
- **ComputeGradients.f90**: Berechnung der Gradienten und Anwendung eines Limiters für räumliche Diskretisierungsordnung 2. Ist von den Studenten zu erstellen.
- **Reconstruction.f90**: Rekonstruktion der linken und rechten Zustände für das Riemannproblem an jeder Seite und MUSCL Technik in der Zeit. Ist für beide Raumordnungen von den Studierenden zu programmieren.
- **FluxCalculation.f90**: Hier werden die 2D Euler Flüsse berechnet indem die Feldgrössen ins jeweilige seitenlokale Koordinatensystem gedreht werden und der so berechnete Fluss danach wieder ins globale System zurück gedreht wird. Wird von den Studierenden programmiert.
- **FluxBalance.f90**: Bilanzierung der Flüsse über jede Zelle. Liefert die Zeitableitung der Feldgrössen in jedem Volumenelement. Ist von den Studenten zu programmieren.

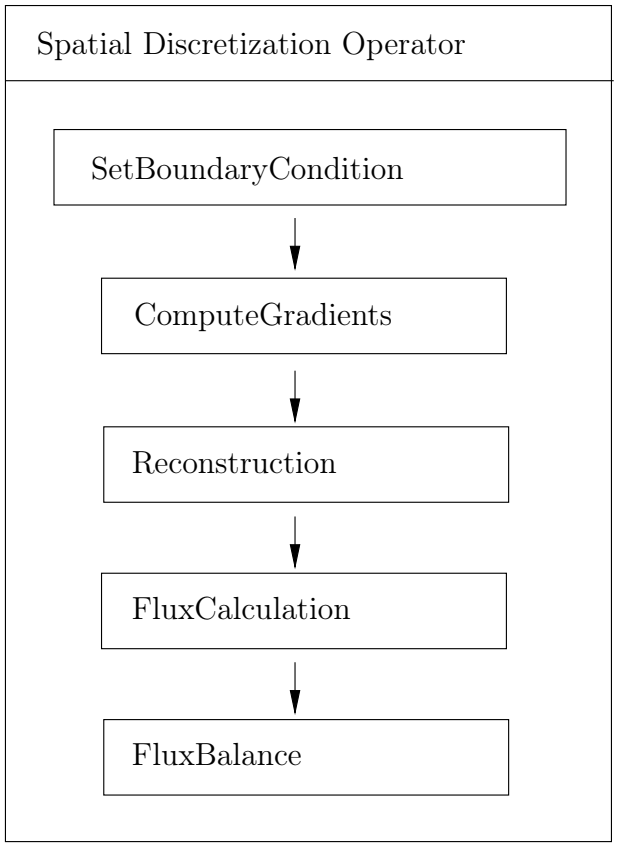


Abbildung 1: Flussdiagramm des Raumdiskretisierungsoperators im Code Euler2D

- **FluxGodunov.f90**: Berechnet den numerischen Fluss nach Godunov. Wurde bereits in Teilprojekt 3 programmiert.
- **ExactRiemann.f90**: Löst ein Riemannproblem exakt durch Fixpunktiteration. Wird zur Verfügung gestellt.

Kern des Codes ist, wie üblich in jedem Finite Volumen Code, die Flussfunktion, der in Abbildung 1 als Flussdiagramm dargestellte Raumdiskretisierungsoperator und die Schleife über die Zeitintegrationsschritte.

2.3 Validierung

Die Validierung erfolgt mittels folgender Testbeispiele:

- 1D Gausspuls nur in der Dichte mit reiner Konvektionsgeschwindigkeit u in x-Richtung. Der Puls muss mit der Geschwindigkeit u transportiert werden.
- 1D Gausspuls nur im Druck, der sich in zwei gleiche nach rechts und links laufende Gausspulse aufteilen muss, die sich je nach Stärke des Druckpulses aufsteilen und sich für sehr kleine Störungen mit der Schallgeschwindigkeit $c = \sqrt{\gamma p/\rho}$ bewegen (vgl. Abbildung 2).
- 2D Gausspuls nur im Druck, der kreisförmige, konzentrische Wellenfronten erzeugen muss, siehe Abbildung 3.

Der Riemannlöser wurde bereits in Teilprojekt 2 implementiert und validiert, so dass obige Tests ausreichen, um die Funktionalität des Gesamtcodes zu überprüfen. Programmierfehler würden sich schnell in Instabilitäten und falschen Ausbreitungsgeschwindigkeiten der Wellen äussern.

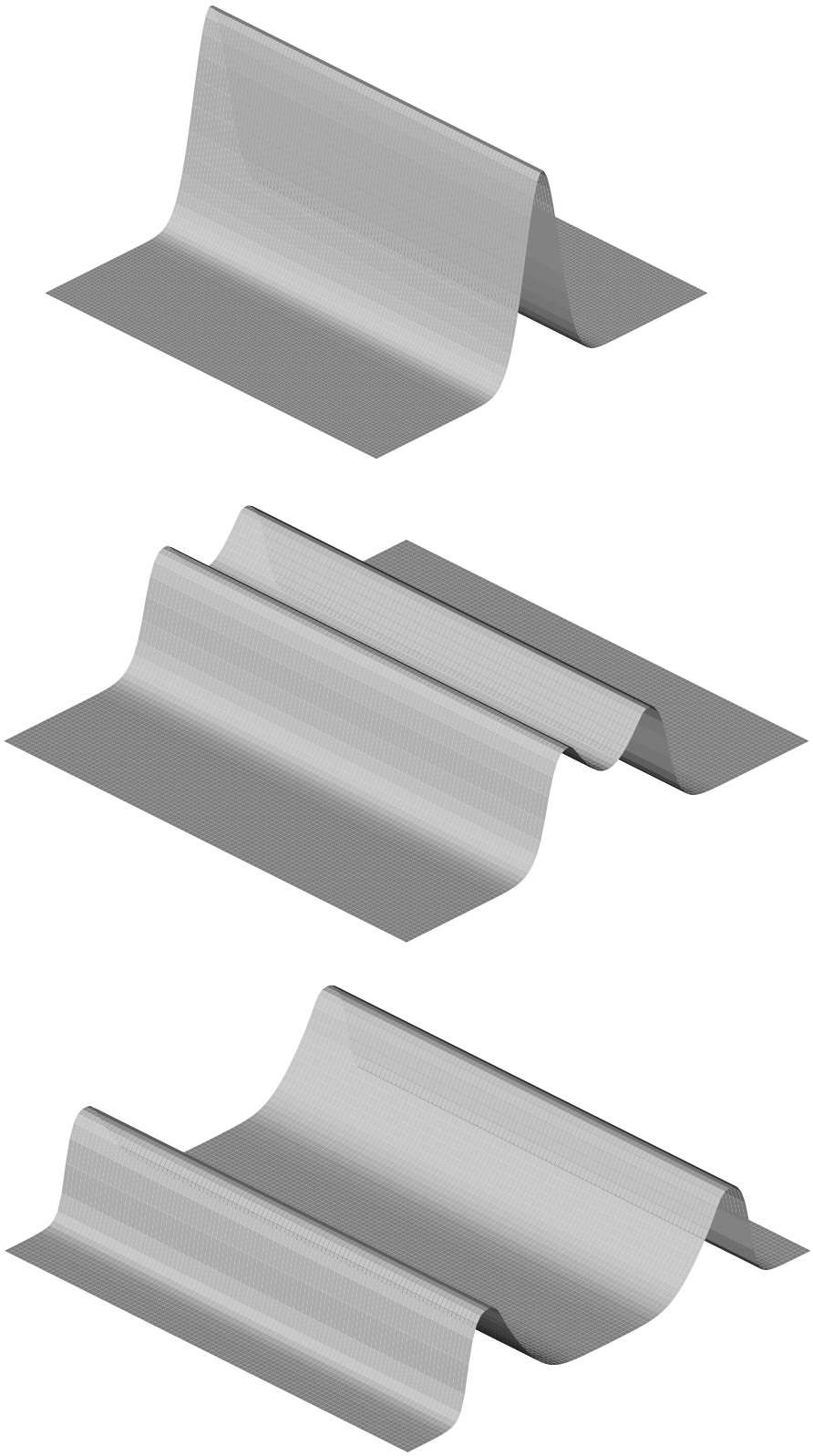


Abbildung 2: 1D Gausspuls in p, der sich in eine links- und rechtslaufende Welle aufteilt

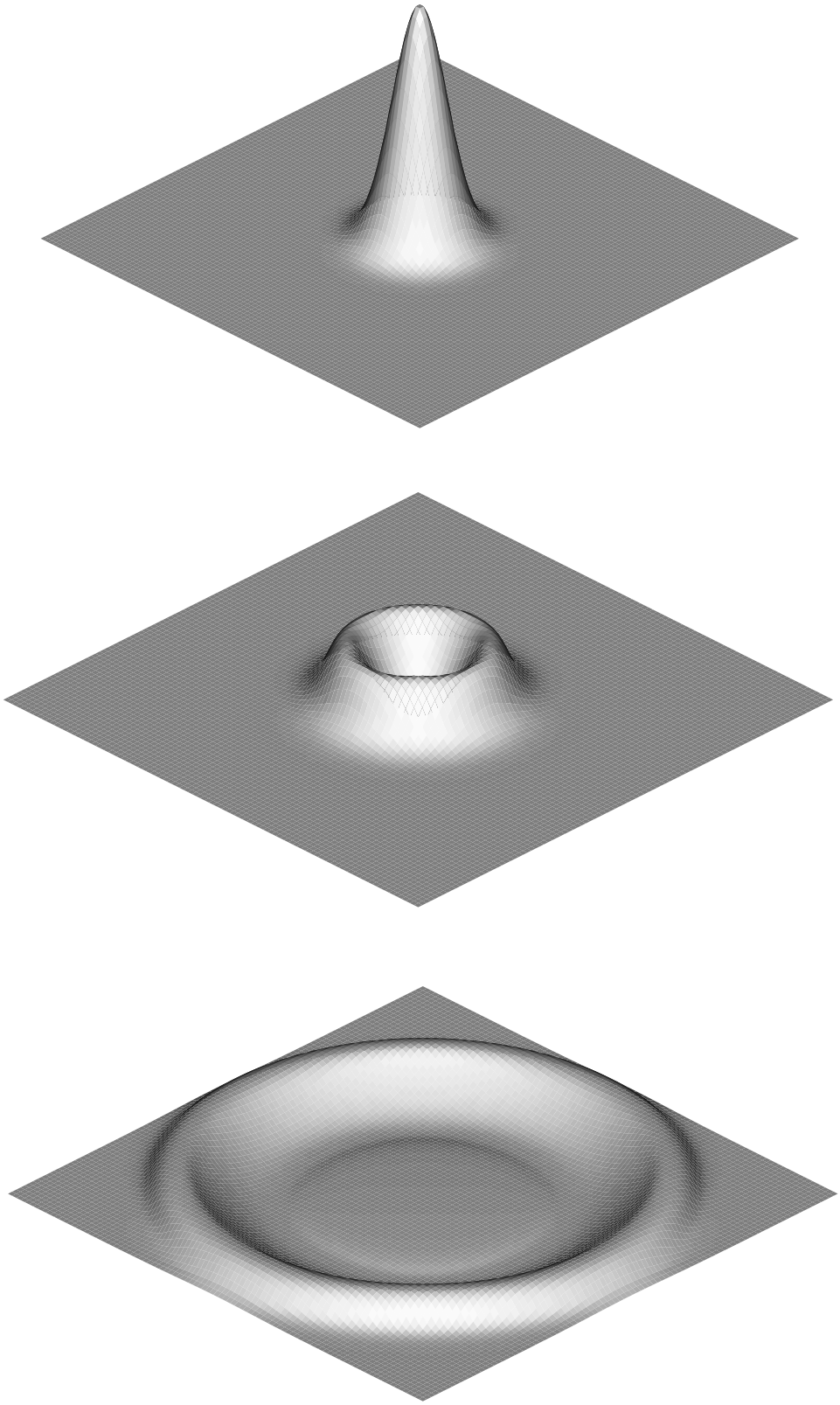


Abbildung 3: 2D Gausspuls in p, der konzentrische Wellenfronten hervorruft