# Analyse eines C++-Programms zur Simulation von Mehrphasenströmungen

Pascal Bähr

14. April 2016

**Zusammenfassung**

Zum besseren Verständnis des Programms soll dieses Dokument dienen. Die Funktionalitäten und die verwendeten Datenstrukturen der einzelnen Klassen werden übersichtlich und verständlich erläutert, so dass eine spätere Umstrukturierung des Programms einfacher wird. Es wird zunächst nur die Implementierung für eine Dimension betrachtet. Referenzen zu Gleichungen beziehen sich auf die zur Verfügung stehende Zusammenfassung des Themas.

## 1 Ablauf

Der grobe Ablauf des Programms.

1. Auswahl der Methode zur Flußberechnung | main
2. Auswahl Splitting/Unsplitting | numerische_methode
3. Solange keine Abbruchbedingung erfüllt worde (maximale Schritte,maximale Zeit)
   (a) Setzen der Randbedingungen (Boundary Conditions) | numerische_methode $\rightarrow$ raster
   (b) Zeitschritt über höchsten Eigenwert berechnen | numerische_methode . cflcon()
   (c) Update Schritt
      i. Fluss über gewählte Methode berechnen (U & F $\rightarrow F^{LF}$)
      ii. Updateschritt berechnen mit dem berechneten Fluss
      iii. Zellwerte aktualisieren
4. Ergebnisse exportieren

# 2 Dateien

## 2.1 main.cpp

Dient bisher nur zur Auswahl der Flußberechnungsmethode.
Zur Auswahl stehen:

a) Lax-Friedrich Methode

b) FORCE Methode

## 2.2 LaxFriedrichMethode.*

Abgeleitet von der Klasse *numerische_methode*
Verwendet Konstruktor der Basisklasse
Implementiert die abstrakte Methode *calc_method_flux()*

### 2.2.1 calc_method_flux()

Parameter *int dir* sollte implementiert werden, damit (ab 2 Dimensionen) nicht immer beide Flüsse F und G (pro Richtung) berechnet werden müssen

#### 2.2.1.1 Variablen

Zunächst Initialisierung einiger Variablen:

a) width & height: Mit Werten der Größe des Rasters

b) neqs: Anzahl der Gleichungen aus Gleichungssystem-Klasse, bei 1 Dimension -≫ 3

c) *uall/*fall/*gall:
Allokierung eines Arrays für zusammenhängenden Speicher

d) ***cs/***f/***g:
3D-Arrays der Größe 'neqs X width X height'
Mit zusammenhängen Speicher in allen Dimensionen über Array uall... + Speicherarithmetik

e) 4-facher Vektor fi:
Return Vektor für $F_i$ bei der Flussberechnung
in etwa wie: a[i][j][k][l]
i = neqs
j = Raster Width
k = Raster Heigth
l = Dimension, also 1 oder 2, später auch 3
alle Werte initialisiert mit 0.0

#### 2.2.1.2 Algorithmus

Eigentliche Fluss-Berechnung:

1. U Berechnen durch Gleichungssystem-Klasse
2. F Berechnen durch Gleichungssystem-Klasse
3. L-F Fluss $F^{LF}$ berechnen über Gleichung 5.50

$$F_i^{LF} = \frac{1}{2}\left(F_i^n + F_{i+1}^n\right) + \frac{\Delta x}{2\Delta t}\left(U_i^n - U_{i+1}^n\right) \tag{1}$$

   $F_i^{LF}$ ist eine Iteration über (in Array-Schreibweise): fi[k][i][0][0]
   k := 0 bis neqs
   i := 0 bis CELLS[0]+ordnung+1
   Besser raster.getwidth() - ordnung
   Die Raster Höhe und die Dimension bei der 1-D Berechnung sind quasi überflüssig.
4. Rückgabe des Vektorkonstruktes

## 2.3 FORCE.*

Abgeleitet von der Klasse *numerische_methode*
Verwendet Konstruktor der Basisklasse
Implementiert die abstrakte Methode *calc_method_flux()*

### 2.3.1 calc_method_flux()

Parameter *int dir* sollte implementiert werden, damit (ab 2 Dimensionen) nicht immer beide Flüsse F und G (pro Richtung) berechnet werden müssen

#### 2.3.1.1 Variablen

Zunächst Initialisierung einiger Variablen:

1. width & height: Mit Werten der Größe des Rasters
2. neqs: Anzahl der Gleichungen aus Gleichungssystem-Klasse, bei 1 Dimension → 3
   Anzahl der Gleichungen dürfte pro Dimension nicht fix sein!
3. *uall/*fall/*gall/*f_laxall/*f_rieall/*g_laxall/*g_rieall:
   Allokierung eines Arrays für zusammenhängenden Speicher
4. ***cs/***f/***g/***f_lax/***f_rie/***g_lax/***g_rie:
   3D-Arrays der Größe 'neqs X width X height'
   Mit zusammenhängen Speicher in allen Dimensionen über uall... + Speicherarithmetik

5. 4-facher Vektor fi:
   Return Vektor für $F_{FORCE}$ bei der Flussberechnung
   wie a[i][j][k][l]
   i = neqs
   j = Raster Width
   k = Raster Heigth
   l = Dimension, also 1 oder 2
   alle Werte initialisiert mit 0.0

#### 2.3.1.2 Algorithmus

Eigentliche Fluss-Berechnung:

1. U Berechnen durch Gleichungssystemklasse
2. F Berechnen durch Gleichungssystemklasse
3. Lax-Friedrichs Fluss $F^{LF}$ berechnen über Gleichung 5.50

$$F_i^{LF} = \frac{1}{2}\left(F_i^n + F_{i+1}^n\right) + \frac{\Delta x}{2\Delta t}\left(U_i^n - U_{i+1}^n\right) \tag{2}$$

   $F_i^{LF}$ ist eine Iteration über (in Array-Schreibweise): fi[k][i][0][0]
   k := 0 bis neqs
   i := 0 bis CELLS[0]+ordnung+1
   Besser -> raster.getwidth() - ordnung
   Fluss wird ein zweites mal über eine Vektor Länge (aus Gleichungssystem) berechnet, wahrscheinlich ein Artefakt!
4. Richtmeyer Vektor $U^{RI}$ (als Raster aufgebaut) berechnen über Gleichung 5.53

$$U_{i+1/2}^{RI} = \frac{1}{2}\left[U_i^n + U_{i+1}^n\right] + \frac{\Delta t}{2\Delta x}\left(F_i^n - F_{i+1}^n\right) \tag{3}$$

   Jeweils für jede Zelle: d (Dichte), ux (Geschwindigkeit in x-Richtung), uxr (Relative Geschwindigkeit in x-Richtung)
   Von Vektor U aus Gleichung 5.24:

$$U = \begin{bmatrix} \rho \\ \rho u \\ u_r \end{bmatrix}$$

   bzw im Programm für die Werte der Zelle:

$$U = \begin{bmatrix} \rho \\ \frac{\rho u}{\rho} \\ u_r \end{bmatrix} = \begin{bmatrix} \rho \\ u \\ u_r \end{bmatrix} => \begin{bmatrix} d \\ ux \\ uxr \end{bmatrix}$$

   zusätzlich den Druck p (Druck) pro Zelle über Gleichung 5.15

$$P = K_g \rho^{\gamma}, \tag{4}$$

5. Richtmeyer Fluss $F^{RI}$ berechnen durch Gleichungssystemklasse. Veranschaulicht durch Gleichung 5.54

$$F^{RI}_{i+1/2} = F(U^{RI}_{i+1/2}). \tag{5}$$

6. FORCE Fluss $F^{FORCE}$ berechnen durch Mittelwert Berechnung zwischen den zwei Flüssen. Gleichung 5.55

$$F^{FORCE}_{i+1/2} = \frac{1}{2}\left[F^{LF}_{i+1/2} + F^{RI}_{i+1/2}\right] \tag{6}$$

   Gespeichert in f_force Vektor

7. Rückgabe des Vektorkonstruktes

## 2.4 numerische_methode.cpp

Basisklasse für die spezialisierten Flußberechnungsmethoden.
Beinhaltet die meiste Funktionalität des Programms.

### 2.4.1 Konstruktor

#### 2.4.1.1 Variablen

Zunächst Initialisierung einiger Variablen durch auslesen aus Werten der Konstanten.
Konstanten Klasse hat in der Implementierung keinen höheren Nutzen.

a) name: für Protokollierung mit write() 2.4.5 zur Unterscheidung zwischen Lax-Friedrich und FORCE Methode

b) ordnung: Ordnung des Verfahrens / Wie viele zusätzliche Zellen an den Rändern miteinbezogen werden / bisher nur 1 implementiert

c) dimension: Dimension in der gerechnet wird. Bisher 1D (x) und 2D (x,y)

d) CELLS: Zeigt Anzahl der Zellen in entsprechender Dimension
CELLS[0] für Zellen in X; CELLS[1] für Zellen in Y

e) cref: Konstante in Equation of States; $K_2$

f) done: Dichte der Phase 1; $\rho_1$

g) ccl: Massenanteil in Simulation

h) mor: x-Wert rechte Grenze - oben rechts

i) mol: x-Wert linke Grenze - oben links

j) mur: y-Wert untere Grenze - unten rechts

k) mul: y-Wert obere Grenze - unten links

l) timeou: Zeit Output - maximale Zeit als Abbruchbedingung

m) steps: Zum Festhalten der gemachten Schritte zur Protokollierung

n) maxnt: Gesetztes Maximum als Abbruchbedingung bei Fehler

o) teilerend: für wieviel Schritte dt geteilt wird

p) teiler: um wieviel dt zuerst geteilt wird
klein Anfang für Stabilität

q) variante: Variante der EOS
andere varianten nicht getestet

r) g: Gamma Konstante für EOS

s) dx: Delta x ((rechter Rand-linker Rand)/Anzahl Zellen in x) / dy: Delta y

t) dt: Delta t bei Berechnung des Time Steps mit CFL
zum Debuggen mit 0 initialisiert, sonst nicht vorher in Verwendung

u) rhol: Initialwert der Dichte der Phase 2, links

v) alfll: Alpha-Berechnung (Volumenanteil) über

$$\alpha = 1 - \rho/\rho_1 + c\rho/\rho_1$$

bzw im Programm

$$alfll = 1.0 - rhol/done + ccl * rhol/done$$

Als Einzelschritt eigentlich unnötig, da nur für den nächsten Schritt benötigt wird

w) dll: Berechnung von Dichte $\rho_2$ über

$$\rho_2 = (c\rho)/\alpha,$$

bzw im Programm

$$dll = ccl * rhol/alfll$$

Als Einzelschritt eigentlich unnötig, da nur für den nächsten Schritt benötigt wird

x) pll: Berechnung von Druck $P$ über

$$P = K_2\rho_2^{\gamma}$$

bzw im Programm

$$pll = cref * dll^g$$

Als Einzelschritt eigentlich unnötig, da nur für den nächsten Schritt benötigt wird

y) ct: Berechnung der CT bzw $K_g$ Größe über (Umformung von Gleichung 5.15)

$$K_g = \frac{P}{\rho^g}$$

bzw im Programm

$$ct = \frac{pll}{rhol^g}$$

z) splitting: Ob mit Unsplitting oder Splitting gerechnet werden soll

#### 2.4.1.2 Funktion

Der Konstruktor lässt entscheiden, ob mit splitting oder unsplitting gerechnet wird.
Könnte Teil der Main Funktion werden

### 2.4.2 start_method()

#### 2.4.2.1 Variablen

a) timDelta x ((rechter Rand-linker Rand)/Anzahl Zellen in x) / dy: Delta ye: Variable für den Timestep (über cflcon() 2.4.3)
startet bei 0.0

b) timetol: Untere Grenze für die Zeit

c) timedif: Nach jedem Berechnungsschritt aktualisiert auf $|time - timeou|$

d) step_output: Gibt an ob Zwischenschritte protokolliert werden sollen

#### 2.4.2.2 Funktion

1. Abfrage ob Updates mit Splitting oder Unsplitting kalkuliert werden sollen
   Nur für/ab 2D, in etwa: wie Mittelwerte der Zellen gebildet werden
2. (Eventuell protokollieren mit write())
3. Schleife läuft bis zu einem Maximum maxnt, damit es bspw im Fehlerfall nicht unendlich läuft
   und solange timedif $>$ timetol
   Also bis Schritt- oder Zeitüberschreitung
   (a) Setzen der Rand- und Anfangswertbedingungen im Raster über die Rasterfunktion: bcondi()
   (b) (Eventuell protokollieren)
   (c) Zeitschritt berechnen über Funktion cflcon() 2.4.3
   (d) Falls Splitting, dann update() 2.4.4 mit entsprechender Flussberechnungsmethode (2.2.1 oder 2.3.1) und Parameter dir = 1
   Für 1D Standard!
   (e) Falls Unsplitting, dann update() mit Flußberechnung und Parameter dir = 1
   neue Setzung der Boundary Conditions (bcondi() **??**)
   Erneutes Update mit Parameter dir = 2
   Unsplitting nur ab 2D.

(f) timedif setzen auf

$$|time - timeou| \tag{7}$$

(g) Variable *steps* auf *n* erhöhen für Protokollierung

4. Ende der Berechnung protokollieren mit write()

### 2.4.3 cflcon()

Berechnung des nächsten Zeitschrittes über die CFL-Condition.

#### 2.4.3.1 Variablen

a) cref: Wie 2.4.1
Wird hier noch mal instantiiert, könnte anders mit Konstanten gelöst werden

b) cfl: CFL Condition aus Konstanten

c) ccl: Wie 2.4.1
Wird hier noch mal instantiiert, könnte anders mit Konstanten gelöst werden

d) done: Wie 2.4.1
Wird hier noch mal instantiiert, könnte anders mit Konstanten gelöst werden

e) gi: invertierte Gamma Konstante $\gamma^{-1} = \frac{1}{\gamma}$

f) maxd: Bekommt Dichte aus Zellen zugewiesen
Nur für analytische Methode, kaum Funktion

g) maxu: Bekommt x-Geschwindigkeit aus Zellen zugewiesen
Nur für analytische Methode, kaum Funktion

h) maxur: Bekommt Relative x-Geschwindigkeit aus Zellen zugewiesen
Nur für analytische Methode, kaum Funktion

i) maxuy: Bekommt y-Geschwindigkeit aus Zellen zugewiesen
Nur für analytische Methode, kaum Funktion

j) maxuyr: Bekommt Relative y-Geschwindigkeit aus Zellen zugewiesen
Nur für analytische Methode, kaum Funktion

k) smax: Höchster Eigenwert als $v_{max}$ für Gleichung 2.19 bzw. Ungleichung 2.20

l) maxs: Kalkulierter Eigenwert zum überprüfen auf Maximum mit smax

m) n_eqns: Anzahl der Formeln in Datei 'formeln' in einer Zeile
Dürfte nicht fix an Dimension gebunden sein

n) uone: Ergebnis des 1. Wertes von Vektor U

$$U = \begin{bmatrix} \rho \\ \rho u \\ u_r \end{bmatrix}$$

o) utwo: Ergebnis des 2. Wertes aus Vektor U

p) uthree: Ergebnis des 3. Wertes aus Vektor U

q) ufour: Ergebnis des 4. Wertes aus Vektor U für 2D

$$U = \begin{bmatrix} \rho \\ \rho u_x \\ \rho u_y \\ u_{r,x} \\ u_{r,y} \end{bmatrix}$$

r) ufive: Ergebnis des 5. Wertes aus Vektor U für 2D

s) p: Druck / ux: x-Geschw / d: Dichte $\rho$ / uxr: x-rel Geschw / dtwo: Dichte der Phase 2; $\rho_2$ / uy: y-Geschw / uyr: y-rel. Geschw
für Kalkulationen

#### 2.4.3.2 Funktion

1. n_eqns über Dimension festlegen
2. Passende Methode gewählt, je nach Dimension und Analytische Lösung (Konstante *calceigv* = 0) oder über Jacobi Matrix (= 1)

- Weitere Schritte nur für 1D und Jacobi analysiert

3. Anlegen von Vektoren und Matrix mit Lapack++
4. Maxima finden von 0 bis CELLS[0]+2*ordnung+1
   Eventuell auch über Raster Weite statt mit CELLS

   (a) Variante zur Berechnung des Drucks auswählen, bisher nur Variante 1 (Gleichung 5.15) implementiert
   Raster anpassen
   $\rho_2$ über umgeformte Gleichung berechnen
   okay mit CT bzw $K_g$?

   (b) Werte des Vektors U über Zellenwerte setzen

   (c) Werte in Jacobi Matrix einsetzen über Funktion matrix_1d() (s. 2.4.6)

   (d) Eigenwerte über lapackpp Library berechnen

   (e) Höchsten Eigenwert in Matrix suchen

   (f) Zeitschritt dt über Gleichung 2.19/2.20 berechnen

(g) *dt* für *teilerend* Schritte teilen um 'teiler'
teilerend und teiler haben numerische Bedeutung

(h) Zeit mit maximaler Zeit 'timeou' als obere Schranke abgleichen und ggfs. 'dt' anpassen

(i) Gesamtzeit anpassen (um 'dt' erhöhen)

5. Aktuelle Zeit zurückgeben

#### 2.4.4 update()

Aktualisieren aller Zellen mit berechnetem Fluss

##### 2.4.4.1 Variablen

a) dtodx: $= \frac{dt}{dx}$

b) dtody: $= \frac{dt}{dy}$ für 2 Dimensionen

c) d,ux,uy,uxd,uyd,uxr,uyr: Variablen für Update Vektor U
ux nur einmalig verwendet
uym uyd, uyr für nur für 2 Dimensionen

d) width: Raster Breite

##### 2.4.4.2 Funktion

1. Passende Methode je nach Dimension wählen (nur Dimension 1 analysiert)
2. Update von 1 (=Ordnung) bis $<$ CELLS[0]+ordnung+1
   Eventuell auch über Raster Weite statt mit CELLS
   also: Zellränder werden zwar mit einbezogen, aber nicht neu berechnet

   (a) d,ux,uxd,uxr auf Werte der aktuellen Zellenwerten setzen (uxd kalkulieren)

   (b) Vektor U des Updateschritts berechnen, analog zu 5.51

$$U_i^{n+1} = U_i^n + \frac{\Delta t}{\Delta x}\left(F_{i-1}^{n,LF} - F_i^{n,LF}\right) \tag{8}$$

   Gleichung auf alle Flüsse anwendbar, nur Unterschiede je nach Dimension

   (c) Zellwerte d, ux, uxr auf Werte des Updateschritts U setzen

#### 2.4.5 write()

Logs für die Ergebnisse
Namen anpassen, ggbf. kürzen
Unterordner benutzen

#### 2.4.6 matrix_1d()

Jacobi-matrix erstellen
Je nach EOS Variante, aber nur Variante 1 funktional implementiert

##### 2.4.6.1 Funktion

Setzt 1 dimensionales Array auf:

$$J = \begin{pmatrix} 0 & 1 & 0 \\ -\frac{u_2^2}{u_1^2} + c(1-c)u_3^2 + \gamma K_\rho u_1^{\gamma-1} & \frac{2u_2}{u_1} & u_1 c(1-c)2u_3 \\ -\frac{u_2}{u_1^2}u_3 + \left(\frac{K_2}{K_\rho}\right)^{1/\gamma} \gamma K_\rho u_1^{\gamma-2} - \frac{\gamma K_\rho u_1^{\gamma-1}}{\rho_1} & \frac{u_3}{u_1} & \frac{u_2}{u_1} + (1-2c)u_3 \end{pmatrix} \tag{9}$$

Index wird zeilenweise erhöht.

### 2.5 raster.*

Bietet ein Raster aus Zellen. Speicherung für alle Dimensionen in einem 1D-Array.

#### 2.5.1 Konstruktoren

Mögliche Parameter / Überladungen

a) int x: 1D Raster - leer
Nie verwendet!?

b) int x, int y: 2D Raster - leer
Nie verwendet!?

c) Konstanten konstanten, string save_in: Konstaten zur Initialisierung, Zwischenspeichern der Schritte
save_in nur in Einzelfall verwendet, anders implementierbar? Fest einprogrammieren?

##### 2.5.1.1 Variablen

a) choice: Wahl für die Rasterinitialisierung (je nach Dimension)

b) *zelle: 1D Zellen-Array

c) dimension: Dimension des Rasters (bis zu 3, aber nur bis 2 implementiert)

d) width: Breite des Rasters

e) height: Höhe des Rasters

f) cells[2]: Anzahl der Zellen - [0] für x / [1] für y

#### 2.5.1.2 Funktion

Nur bei letzter Variante des Konstruktors

1. Sämtliche Variablen aus Konstanten einlesen
   (a) cells[1] je nach Dimension
   (b) Width/Height über
   $$cells[d] + 2 * ordnung + 1$$
   für Position im 1D-Zellenarray
   (c) dx über ((rechter Rand-linker Rand)/Anzahl Zellen in x)
   (d) zelle als Array der Größe width*height
2. Je nach Dimension entsprechende Wahl für Raster Initiierung ausgeben
   Hier: 1D und Riemann-Problem analysiert
3. Von 'ordnung' (nur 1 bisher) bis $<$ cells[0]+ordnung+1
   also von erstem Zellwert bis einschließlich dem letzten Zellwert ohne Ränder
   (a) xpos über mol + (mor-mol)*((n-ordnung)/cells[0])
   Immer eine Zelle weiter, angefangen bei mol (x ganz links)
   (b) Wenn xpos $<=$ 0.0 (also linke Hälfte), dann Zellenwerte (d,ux,uxr) auf rhol, vl, vrl
   (c) Sonst (in rechter Hälfte) Zellenwerte auf rhor, vr, vrr

### 2.5.2 Getter & Setter

Getter und Setter für Raster Variablen und Zellenwerte
bis auf wenige Ausnahmen (dienen nur der Übersicht/Einfachheit) wahrscheinlich überflüssig

a) get-width/-height/-dim:
   Geben unveränderte Werte wieder
   Einziger Zweck, vermeidung von Settern. Bei Vorsicht auch kein Problem!?

b) get_Zelle(...):
   Getter mit Parameter je nach Dimension (x,y,z)
   Getter machen (halbwegs) Sinn, wegen Zugriff auf eindimensionales Zell-Array
   Nur der für 3 Dimensionen wurde ordentlich implementiert, die anderen rufen den dritten auf
   Entweder alle 3 ordentlich implementieren / Default Paramter verwenden / Zugriff auf Zell-Array jeweils direkt implementieren)

c) set_Zelle_*:
   Setter einzelne Zellwerte (ux,uxr,uy,uyr,d,p) je nach Dimension (nur für 1-2 Dimensionen implementiert)
   Setter machen (halbwegs) Sinn wegen Zugriff auf 1D-Array der Zellen
   Anzahl an Settern absurd, für Performance wahrscheinlich am besten Inline, sonst für alle Dimensionen einen, oder gar einen für ganzen Vektor?

#### 2.5.3 bcondi

Anwendung der Randbedingungen (Boundary Conditions)

##### 2.5.3.1 Variablen

a) up-,down-,right-,left-bc: Randbedingungen aus Konstanten für alle Richtungen

##### 2.5.3.2 Funktion

1. Je nach Dimension entsprechende Zellen (d,ux,uxr,...) setzen.
2. Bei 1D: Linker Rand (Zelle 0) und Rechter Rand (Zelle CELLS[0]+ordnung+1)
   Für 2. (bzw. x-te) Ordnung nur die äußersten Zellen oder 2 bzw x äußersten Zellen?
3. Je nach Option (0 = durchlässige; 1 = reflektierende)
   Bei Reflektierend (immer? bei Dia 'oft') nur Geschwindigkeit reflektiert.
   Weitere Boundary Conditions implementieren? Periodisch?

### 2.6 zelle.*

Fasst sämtliche Werte einer Zelle als Objekt zusammen
Fix nur als 2D Zelle implementiert
Dynamisch, oder fix für alle Dimensionen implementieren, nicht zu kompliziert

#### 2.6.1 Konstruktor

Initialisiert sämtliche Werte mit 0.0

##### 2.6.1.1 Variablen

a) ux,uy: Geschwindigkeit in x/y Richtung

b) uxr,uyr: Relative Geschwindigkeit in x/y Richtung

c) d: Dichte

d) p: Druck

### 2.7 Konstanten.*

Fasst alle für die Berechnung verwendeten Konstanten zusammen
Liest Konstanten aus Datei ein (Dateiname fix in main einprogrammiert, eventuell Auswahl?)
Statische Klasse für gesamten Programmablauf macht mehr Sinn für Konstanten

### 2.7.1 Konstruktor

Liest sämtliche Werte aus übergebener Datei ein.

#### 2.7.1.1 Variablen

a) Vektoren const_-*: Werden nicht verwendet!

b) Konstanten sind fix für alle alle (bis zu 2) Dimensionen

c) Konstanten sind fix für alle Raster-Varianten
Nur die Erstellen/Lesen die nötig sind

## 2.8 Gleichungssystem.*

Verwaltung der Formelvektoren u, f_u und s_u

#### 2.8.0.1 Variablen

a) Vektoren u, f_u, s_u und uback nicht mehr in Verwendung

b) Konstanten Instanz wird nicht (wirklich) verwendet

c) cref, done, ccl, gc: Konstanten $K_2$, $\rho_2$, c (Massenanteil), $\gamma$ = ???

d) ccl12: $(1 - 2c)$

e) ccl12h: $\frac{(1-2c)}{2}$ für Gleichung 5.14

f) gcinv: $\gamma$ invertiert, also $\frac{1}{\gamma}$

g) gcdgcm1: $\frac{\gamma}{\gamma-1}$ für Gleichung 5.13

h) gcm1dgc: $\frac{\gamma-1}{\gamma}$ für Gleichung 5.13

i) cclm1: $c * (1 - c)$ für 5.26

j) powcref: $K_2^{\frac{1}{\gamma}}$ als $\Psi$ bei Lösung von Dia (s.19)

### 2.8.1 Konstruktoren

Variablen Initialisieren
neqs (Anzahl Gleichungen) über dim
'dim' nicht wirklich benötigt

#### 2.8.1.1 Variablen

a) Vektoren const_-*: Werden nicht verwendet!

b) Konstanten fix für alle alle (bis zu 2) Dimensionen

c) Konstanten fix für alle Raster-Varianten
Nur die Erstellen/Lesen die nötig sind

### 2.8.2 compute_u_1d

Berechnung der Werte U für 1 Dimension
Bekommt 3d Array für Vektor u
Raster für alle Zellwerte
'cells' für Gitterbreite könnte weggelassen werden, da über Raster indirekt vorhanden

#### 2.8.2.1 Funktion

a) Für alle Zellen des Gitters inkl. Rand (0 bis cells[0]+2*ordnung+1)

   (a) u[0][i][0] auf $\rho$-Wert der Zelle

   (b) u[1][i][0] auf $\rho * u$-Wert der Zelle

   (c) u[2][i][0] auf $u_r$-Wert der Zelle

### 2.8.3 compute_u_1d

Berechnung der Werte U für 1 Dimension
Bekommt 3d Array für Vektor u
Raster für alle Zellwerte
'cells' für Gitterbreite könnte weggelassen werden, da über Raster indirekt vorhanden
Im Endeffekt werden Zellwerte nur umkopiert
Entspricht 5.24

$$U = \begin{bmatrix} \rho \\ \rho u \\ u_r \end{bmatrix} \tag{10}$$

#### 2.8.3.1 Funktion

a) Für alle Zellen des Gitters inkl. Rand (0 bis $<$ cells[0]+2*ordnung+1)

   (a) u[0][i][0] auf $\rho$-Wert der Zelle

   (b) u[1][i][0] auf $\rho * u$-Wert der Zelle

(c) u[2][i][0] auf $u_r$-Wert der Zelle

#### 2.8.4 compute_f_1d

Berechnung die Lösung der Formel für den Fluss in 1 Dimension
Bekommt 3d Array für Vektor von Funktionen F(U)
Raster für alle Zellwerte
'cells' für Gitterbreite könnte weggelassen werden, da über Raster indirekt vorhanden
Entspricht 5.24

$$F(U) = \begin{bmatrix} \rho u \\ \rho u^2 + \rho c(1-c)u_r^2 + P \\ u u_r + \frac{1-2c}{2} u_r^2 + \Psi(\rho, P)) \end{bmatrix} \tag{11}$$

Hier die 'anpassbaren' Gleichungen?
mit $\Psi$ von Dias Lösung

$$\Psi(P) = \frac{\gamma}{\gamma - 1} K_2^{1/\gamma} P^{(\gamma-1)/\gamma} - \frac{P}{\rho_1}$$

##### 2.8.4.1 Funktion

a) Für alle Zellen des Gitters inkl. Rand (0 bis < cells[0]+2*ordnung+1)
   (a) f[0][i][0] auf Ergebnis der 1. Komponente des Vektors
   (b) f[1][i][0] auf Ergebnis der 2. Komponente des Vektors
   (c) f[2][i][0] auf Ergebnis der 3. Komponente des Vektors

# 3 Überarbeitung

## 3.1 Namenskonventionen

a) Allgemein
   (a) Nur englische Bezeichner
   (b) Einzelne Worte mit _-Zeichen trennen
b) Variablen/Parameter
   (a) Lowercase → cell_width
   (b) Außerhalb von Formeln: Sprechende Bezeichner → number_of_equations

(c) “Boolsche“ Variablen: sollten true/false implizieren → with_splitting

(d) Mathematisch: Namen wie in Formeln, gr. Buchstaben ausschreiben → x, rho_1, ...

c) Makros: Uppercase → CIRCLE(x,y)

d) Methoden: wie Variablen → set_cells

e) Klassen: Pascal case → Cells, Numerical_Method, ...

f) Außerhalb

(a) Dateien: Lowercase, quasi wie Methoden

(b) Ordner: Pascal case, quasi wie Klassen

## 3.2 Zeit Tests

### 3.2.1 Beobachtung

Bei der original Version von Herrn Hensel, welche die exprtk-Bibliothek verwendet, konnten durch einfache Zeitstempel schnell die Bottlenecks ausgemacht werden.
Das Problem liegt bei der Berechnungs des Updateschritts bzw. in den Aufrufen zum Lösen des Gleichungssystems.
Die einzelnen Aufrufe der jeweiligen solve-Methode aus der Gleichungssystem-Klasse benötigen zwar vereinzelt nicht lange (ungefähr 2e-04 (0,0002) Sekunden pro Aufruf). Bei den gegebenen Input Daten (30x30 Raster) sind das allerdings bereits 16335 Aufrufe pro Berechnungsschritt nur für die Flussberechnung, also mindestens 3,267 Sekunden. Ein gesamter Schritt dauert ungefähr 5 Sekunden (auf meinem System).
Das umgeschriebene Programm (ohne Verwendung von exprtk) benötigt für einen Einzelschritt gerade mal knapp 8e-04 (0.0008) Sekunden. Die exprtk-Variante ist also 6250 mal langsamer.
Vernachlässigbar wäre eine dynamische Variante die maximal X mal langsamer ist.

### 3.2.2 Idee

Viele aufwendige Funktionen der exprtk-Bibliothek werden repetitiv aufgerufen.
Wahrscheinlich könnte man zahlreiche dieser Funktionen in einem einmaligen Initialaufruf unterbringen, wodurch die eigentlichen Berechnungsschritte dramatisch verkürzt werden könnten.

### 3.2.3 Durchführung

Sämtliche Funktionen der exprt-Bibliothek außer *expression.value()* wurden im Konstruktorder Gleichungssystem-Klasse initialisiert, die Klasse wurde dafür um Attribute der entsprechenden Typen erweitert. Das Ergebnis ist trotz einer unsauberen Lösung gut. Die exprt-Variante ist so nur noch um einen Faktor von ca. 2-3 langsamer.
Die Bibliothek kann also in der neuen Version weiter verwendet werden, wodurch dynamische Anpassungen möglich sein werden.

Weitere Varianten, z. B. Parsing von Gleichungen über Embedded Python, werden aufgrund des relativ zufriedenstellenden Ergebnisses nicht weiter betrachtet.

## 3.3 Neue Version

In der neuen Version sollten die einzelnen Klassen nachfolgende Funktionalitäten bieten.
Generell sollten einige Klassen nur einmal instanziiert werden (statisch).
Switch Cases zum Großteil durch umlagern in Unterfunktionen kürzen/überschauberer machen

### 3.3.1 main

a) Sämtliche Entschiedungen sollten hier getroffen werden können
Entscheidungen könnten Teil von "Setter"-Methoden der jeweiligen Klasse sein.

b) Aufruf der eigentlichen Methode zur Flußberechnung

### 3.3.2 Flußberechnungsmethoden - Lax-Friedrich/FORCE

a) Die Klasse könnte von der Numerischen MethodeKlasse unabhängig werden (keine Vererbung)

b) Der Speicher könnte fix reserviert werden (Konstruktor) und müsste nicht für jede Flussberechnung neu vergeben werden

c) Der Rückgabe-Vektor könnte, um Dimensions-unabhängig zu bleiben, ein 1D Array nach einheitlichem Schema sein.
Da die Flußberechnungsmethoden Dimensions-abhängig sind, könnten für jede Dimension auch unterschiedliche Methoden erstellt werden, mit einem Array der entsprechenden Dimension als Rückgabewert .

### 3.3.3 Numerische Methode

a) Könnte weiterhin als zentrale Klasse für die eigentliche Lösung dienen

b) exprtk Verwendung auslagern in spezielle Klasse

c) Bleibt es bei Splitting/Unsplitting für 2D? Wie würde es bei 3D aussehen?

d) Je nach Anzahl könnte es sinnvoller sein für jede Variante (Un-/Splitting, ...) eine Subklasse zu erstellen.

e) Lange Switch Cases können vermutlich reduziert/übersichtlicher werden, durch Auslagerung in Teilfunktionen

### 3.3.4 Gleichungssystem

a) Eine Klasse für alle Operationen mit exprtk-Bibliothek

b) Statisch vorhanden

c) Eventuell nur eine solve()-Methode nötig mit Parameter

d) Könnte cflcon und update Methode direkt beinhalten, falls solve Aufrufe

### 3.3.5 Raster

a) Entscheidung über Initialisierung wird in der main-Funktion getroffen und als Parameter angenommen

b) Statisch vorhanden, da immer (?) nur ein Raster benötigt wird

c) Eventuell auch eher ein Raster pro Dimension, um unnötige Aufrufe einzuschränken.

d) Raster-Initialisierungen im Konstruktor in Teilfunktionen umlagern / eventuell auch eine abstrakte Grundklasse

e) Getter/Setter werden entfernt, falls sie nur einen unveränderten Wert zurückgeben.

### 3.3.6 Raster

a) Entscheidung über Initialisierung wird in der main-Funktion getroffen und als Parameter angenommen

b) Statisch vorhanden, da immer (?) nur ein Raster benötigt wird

c) Eventuell auch eher ein Raster pro Dimension, um unnötige Aufrufe einzuschränken.
Da muss ich mir noch einen genauen Überblick machen, wie man eine Raster Klasse effizient für alle Dimensionen verwenden könnte.

d) Raster-Initialisierungen im Konstruktor in Teilfunktionen umlagern / eventuell auch eine abstrakte Grundklasse

e) Getter/Setter werden entfernt, falls sie nur einen unveränderten Wert zurückgeben.

### 3.3.7 Raster

a) Entscheidung über Initialisierung wird in der main-Funktion getroffen und als Parameter angenommen

b) Statisch vorhanden, da immer (?) nur ein Raster benötigt wird

c) Eventuell auch eher ein Raster pro Dimension, um unnötige Aufrufe einzuschränken.
Da muss ich mir noch einen genauen Überblick machen, wie man eine Raster Klasse effizient für alle Dimensionen verwenden könnte.

d) Raster-Initialisierungen im Konstruktor in Teilfunktionen umlagern / eventuell auch eine abstrakte Grundklasse

e) Getter/Setter werden entfernt, falls sie nur einen unveränderten Wert zurückgeben.

### 3.3.8 Zelle

a) Pro Dimension ein Zellentyp

b) Eine dynamische Variante wäre die Verwendung eines Zell-Arrays (2D), welches in der ersten Dimension wie im bisherigen Programm die einzelne Zelle des Rasters anspricht (dimensions unabhängig) und in der zweiten Dimension PLatz für die nötigen Variablen bietet (ux, uy, ...). Die Variablen könnten, damit es sprechender bleibt, über Enums angesprochen werden: z.B. cells[0][CELL_UX]

Generell sind die Klassen, die Herr Hensel in seinem Programm erstellt hatte, dieselben auf die meine Version aufbauen würde.
Einige Grundfunktionen, wie die Flußberechnung würde ich allerdings in einzelne abstrakte Klassen ausgliedern, damit das Programm besser um eben solche zu erweitern ist.
Hier wäre es also wichtig zu wissen, welches die Stellen sind, an denen Gleichungen/Varianten ersetzt werden müssten.