IPVS – Universität Stuttgart
Abteilung Simulation großer Systeme
Prof. Miriam Mehl
Carolin Schober, Florian Lindner

WS 15/16

Numerische Simulation
Übungsblatt 1

Serielle Implementierung des Strömungslösers
Abgabe per EMail an florian.lindner@ipvs.uni-stuttgart.de bis zum 15.11.2016, 18:00

Einführung Strömungslöser

1.1 Aufgabenstellung

Im Rahmen des ersten Übungsblatts wird ein Programm zur Simulation und Visualisierung viskoser, instationärer, laminarer Strömungen erstellt. Wir betrachten hierbei nur inkompressible Medien, wie bspw. Wasser. Bei solchen Medien ändert sich die Dichte ρ über die Zeit nicht. Luft wäre ein Beispiel für ein kompressibles Medium, denn die gleiche Masse an Luft kann verschiedene Volumen ausfüllen.
Die Zähigkeit eines Mediums wird über die Viskosität definiert. Zum Beispiel hat Wasser bei 29°C eine Viskosität von $\eta = 90.891\frac{Ns}{m^2}$, Olivenöl $\eta \sim 10^2\frac{Ns}{m^2}$ und Honig $\eta \sim 10^4\frac{Ns}{m^2}$. Die dynamische Viskosität wird mit η bezeichnet und hat die Einheit $\frac{Ns}{m^2}$.
Das heißt, dass ein Medium mit steigender Viskosität zunehmend zähflüssiger ist. Laminar bedeutet, dass die Strömung keine Turbulenzen beinhaltet. Dies wird durch nicht zu hohe Geschwindigkeiten und nicht zu geringe Viskosität realisiert, resultierend in einer hinreichen geringen Reynolds-Zahl $Re = \frac{\rho \cdot u \cdot l}{\eta}$.
Eine instationäre Strömung kann sich über die Zeit verändern. Abbildung 1 zeigt ein erstes Beispiel für ein numerisches Strömungsproblem, die Nischenströmung (Driven-Cavity-Problem).

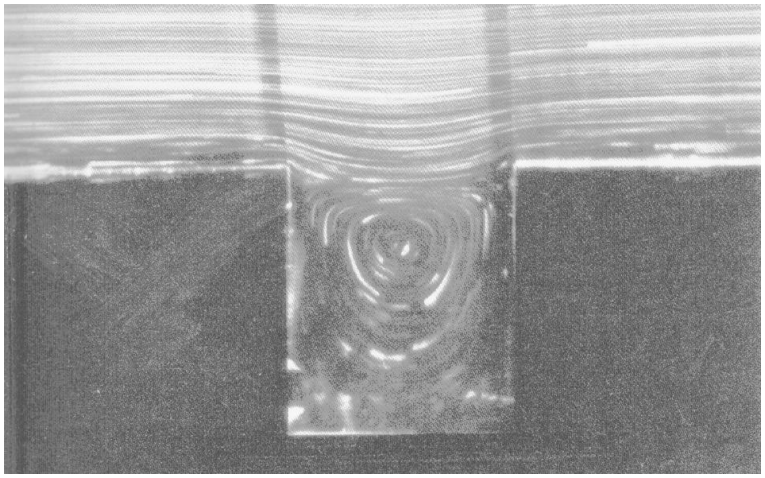


Abbildung 1: Schnappschuss einer Nischenströmung.

1.2 Entwurf

Die komplette Aufgabe wird in `C++` implementiert. Sie dürfen bei der Implementierung **keine** high-level Bibliotheken verwenden und das Programm **muss** plattformunabhängig gehalten werden, denn auf dem Testsystemen können Programme nur unter Linux (*VISSIM-Pool* oder *SGS-Pool*) kompiliert und gestartet werden. Um zwischen den beiden Datentypen *float* und *double* wechseln zu können verwenden Sie bitte einen *typedef* Realtype. Alle Teile des Programms müssen ausreichend kommentiert werden. Die Kommentare müssen für doxygen[1] [2] geeignet formatiert werden.

Unter Berücksichtigung der weiteren Übungsblätter und der einfacheren Fehlerbehandlung müssen Sie sich bei der Implementierung an den hier vorgestellten Entwurf halten. Nach der

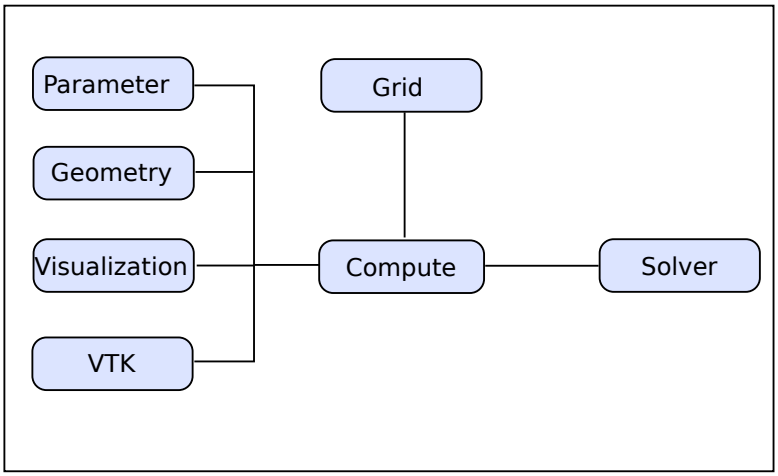


Abbildung 2: Erstes UML-Klassendiagramm nach der Analyse

ersten strukturierten Analyse des Algorithmus lässt sich folgendes UML-Klassendiagramm (Abbildung 2) erstellen. In den nächsten Abschnitten werden die einzelnen Komponenten kurz erklärt und im Gesamtentwurf eingeordnet. Zum Einstieg erhalten Sie ein Rahmenprogramm, das die Header Dateien aller Klassen und Module enthält. Eine genauere Dokumentation und die geforderte Funktionalität der jeweiligen Komponenten entnehmen Sie bitte den Header Dateien des Rahmenprogramms.
Darüberhinaus verfügt das Rahmenprogramm über eine live-Visualisierung der Simulationsergebnisse, deren Implementierung Ihnen bereitgestellt wird. Eine Bedienungsanleitung zur live-Visualisierung finden Sie weiter unten.

Im Rahmenprogramm finden Sie in der Datei `typedef.hpp` eine Anzahl an **typedefs**, die Sie bei Ihrer Implementierung verwenden sollen.

Grid

Die Klasse `Grid` fasst die Funktionalität für das der Diskretisierung zugrundeliegende Gitter zusammen. Dazu gehören Datenzugriffe zum lesen und schreiben von Werten an den Gitterpunkten sowie die Bereitstellung von Operatoren zur Berechnung räumlicher Ableitungen erster

[1] http://www.stack.nl/~dimitri/doxygen/
[2] http://www.stack.nl/~dimitri/doxygen/docblocks.html

und zweiter Ordnung. Zur Diskretisierung der Ableitungen wird ein kompakter Stencil verwendet der nur die Information an den benachbarten Gitterpunkten verwendet. Die interne Repräsentation des Gitters ist ein ein-dimensionales Feld und Datenzugriffe sind ausschließlich über geeignete Iteratoren möglich. Die diskreten Simulationsgrößen u, v, p und andere diskrete Felder werden als Instanzen der Grid Klasse als Member in Compute angelegt. Implementieren Sie die in der Header Datei definierten Funktionen und Schnittstellen der Grid Klasse.

Iterator

Um auf bestimmte Stellen der diskreten Größen bequem und sicher zugreifen zu können wird ein Iterator definiert. Die abstrakte Klasse `Iterator` verfügt zu diesem Zweck über eine ganze Reihe an Funktionen. So kann man mit `First()` und `Next()` über den gesamten Definitionsbereich eines Iterators laufen oder mit den Funktionen `Up()`, `Down()`, `Right()` und `Left()` auf Nachbarelemente zugreifen. Die Iteratoren `InteriorIterator` und `BoundaryIterator` spezialisieren die Iterator Klasse indem sie den Definitionsbereich des Iterators auf das Innere des Simulationsgebiets bzw. den Rand des Simulationsgebiets einschränken.

Computation

Die Klasse `Compute` ist die zentrale Einheit des Strömungslösers, hier findet die eigentliche Simulation und Berechnung statt. Der Konstruktor initialisiert die Simulation und benötigt Information über das Simulationsgebiet und dessen Geometrie sowie über die Simulationsparameter. Die in der Header-Datei `Compute.hpp` gegebenen Funktionen sind nach Kapitel 3.2.6 im Skript und den dort referenzierten Gleichungen zu implementieren.

Solver

Das entstehende lineare Gelichungssystem für den Druck soll iterativ gelöst werden. Kapitel 4 im Skript erläutert die Funktion eines iterativen Lösers am Beispiel der Gauß-Seidel Iteration und beschreibt die Erweiterung zum SOR Löser. Die Klasse `Solver` unterstützt die Grundfunktionalität eines iterativen Lösers für lineare Gelichungssysteme. Der `SOR`-Solver erbt von der abstrakten `Solver` Klasse und implementiert die `Cycle` Methode gemäß Gelichung (4.1) im Skript.

Geometry

Die `Geometry` Klasse verfügt über Methoden zum laden und schreiben einer rechteckigen Geometrie von und in eine Datei. Hierbei werden die Parameter `length`, `size` und `velocity` für jede Dimension gelesen und geschrieben. `length` gibt die Ausdehnung des Simulationsgebiet in dieser Raumrichtung an, `size` die räumliche Auflösung des Gitters und `velocity` den Wert der initialen Geschwindigkeit.
Darüberhinaus verfügt die `Geometry` Klasse über Methoden, die die Randwerte für u, v und p setzen. Die Randwerte müssen zu Beginn jedes Zeitschrittes neu gesetzt werden. Hinweise zum Setzen der Randwerte finden Sie für die Fließgeschwindigskomponenten u und v im Skript, Kapitel 3.1.3. Das Setzen der Randwerte des Drucks p und der "vorläufigen" Geschwindigkeitskomponenten F und G erläutert Kapitel 3.2.3.

Parameter

Die `Parameter` Klasse liest Werte aus einer Parameter-Datei ins Programm ein, speichert diese und macht sie über Get-Funktionen für andere Komponenten öffentlich. Die notwendigen Parameter sind in Tabelle 1 gegeben:

tend	Endzeit t_{end}
tau	Sicherheitsfaktor Zeitschritt τ
dt	Zeitschrittweite
itermax	Maximale Anzahl an Iterationen
eps	Toleranz für Druckiteration ε
omega	Relaxationsfaktor ω
alpha	Upwind-Differencing-Faktor α
re	Reynoldszahl Re

Tabelle 1: Eingabeparameter für die Simulation

VTK IO

Die Klasse `VTK` organisiert das rausschreiben der Simulationsergebnisse in `.vtk`-Format zur späteren Visualisierung mit ParaView. Diese Klasse wird Ihnen vollständig bereitgestellt und muss nicht modifiziert werden. Sie sollten sich jedoch mit der Funktionalität vertraut machen um die entsprechenden Funktionen korrekt einzusetzen.

1.3 Beispiel

Zu Beginn simulieren Sie ein typisches Beispielproblem aus der numerischen Strömungsmechanik, die Nischenströmung (Driven-Cavity-Problem). Abbildung 3 zeigt die Randwertbedingungen auf unserem Gebiet $\Omega := [0,1] \times [0,1] \subset \mathbb{R}^2$. Simuliert wird ein mit dem Medium gefülltes qua-

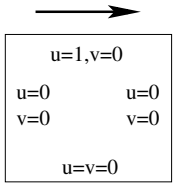


Abbildung 3: Randwertbedingungen für das Driven-Cavity-Problem.

dratisches Gefäß. An allen Rändern sind Haftbedingungen gewählt. Die oberste Schicht wird mit einer konstanten Geschwindigkeit bewegt, so dass sich das Medium in Bewegung gesetzt. Dieser Effekt wird durch Änderung der Randwertbedingungen realisiert:

$$u_{i,jMax+1} = 2.0 - u_{i,jMax}, \qquad i = 1, \ldots, iMax$$

Für die Simulation werden folgende Parameter verwendet:

Parameter	Wert
$xSize = ySize$	128
$xLength = yLength$	1.0

Tabelle 2: Geometrie Parameter Driven-Cavity

Parameter	Wert
Re	1000
ω	1.7
τ	0.5
eps	0.001
α	0.9
$itermax$	100

Tabelle 3: Simulationsparameter Driven-Cavity

Starten Sie die Simulation und visualisieren Sie die Ergebnisse mit ParaView für die Zeitschritte $t = 0.0 + i \cdot 0.2$ mit $i \in [0, 82]$. Eine Anleitung zum Laden der Ergebnisse in ParaView finden Sie im nächsten Kapitel.

1.4 Build-System

Um Ihren Code zu bauen, haben Sie 3 Möglichkeiten:

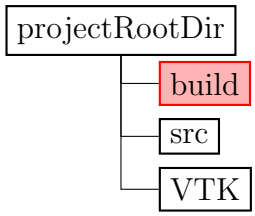
a) Aufsetzen Ihres eigenen Makefiles

b) Verwenden der vorgegebenen CMake[3] Umgebung

c) Verwenden der vorgegebenen SCons[4] Umgebung

Im folgenden wird die Verwendung der beiden Buildsysteme kurz vorgestellt.

1.4.1 CMake



CMake ist ein populäres Build-System, das von diversen großen Open-Source Projekten eingesetzt wird, bspw. Deal.II (FEM-Bibliothek), Gmsh (Bibliothek zum Erstellen unstrukturierter Dreiecksgitter) oder LAPACK (**L**inear **A**lgebra **Pack**age).
Um ein Projekt mit CMake zu bauen, können Sie optional ein Buildverzeichnis erstellen. In diesem oder dem Projekt-Wurzelverzeichnis rufen Sie cmake auf. Daraufhin wird von CMake ein makefile erzeugt, das Sie mit `make -j` *Anzahl Prozesse* aufrufen können. Zum Konfigurieren von cmake können Sie entweder die Kommandozeile verwenden

`cmake -DDEBUG_VISU=ON bzw -DDEBUG_VISU=OFF` $*projectRootDir*

oder das ncurses interface

`ccmake` $*projectRootDir*

Der gezeigte Parameter setzt ein Flag, ob die in Kapitel 1.5 vorgestellte Visualisierung gebaut werden und angezeigt werden soll (im Quellcode mit Präprozessor-Direktiven realisiert). Außerdem können Sie wählen, ob Sie eine Version mit oder ohne Debug-Information bauen wollen.

1.4.2 SCons

Alternativ zu CMake können Sie SCons benutzen. Im Gegensatz zu CMake, wo das Build-System in der Datei `CMakeLists.txt` definiert wird, besteht das Build-System bei SCons aus Python Datei(-en). Die Hauptdatei heißt `SConstruct`, in dieser wird die Build-Umgebung eingerichtet, d.h. Compiler-Flags werden gesetzt. Anschließend wird die Datei `SConscript` aufgerufen und dabei automatisch ein Build-Verzeichnis erstellt. In dieser Datei werden alle nötigen Source-files zusammen gesucht, ein Name für die gebaute Datei definiert und schließlich das Programm gebaut. Zwei Dateien sind nötig, um gleichzeitig ein Build-Verzeichnis haben zu können und die Dateien nicht bzgl. des Build-Verzeichnisses sondern ihres tatsächlichen Orts im Speicher zusammen suchen zu können; für weitere Informationen: https://bitbucket.org/scons/scons/wiki/VariantDir%28%29.

[3] http://cmake.org/
[4] http://scons.org/

Zur Konfiguration, ob Sie eine Visualisierung bauen möchten oder nicht, verwenden Sie die Datei `custom.py`. Debug-Informationen zur Verwendung des gdb können Sie wiefolgt über einen Kommandozeilen-Parameter steuern

`scons debug=1 -j` *AnzahlProzesse*.

Die Installation von SCons erfolgt nach[5]. Anschließend müssen Sie den Ort der Installation in die `${PATH}` Umgebungsvariable aufnehmen:

```
export PATH=${HOME}/bin:${PATH}
```

Damit Sie den Befehl nicht bei jedem Einloggen erneut eingeben müssen, können Sie eine Datei `.bashrc` anlegen und den Befehl dort hinein schreiben. Damit es keinen Unterschied gibt, wenn Sie sich per ssh einloggen, müssen Sie zusätzlich eine Datei `.bash_profile` anlegen und dort folgendes hinein schreiben:

```
source .bashrc
```

Das führt dazu, dass unabhängig davon, ob Ihre aktuelle Shell eine Login-Shell (via ssh) oder eine interaktive Shell (wenn Sie aus einer beliebigen Desktopumgebung heraus ein Terminal öffnen), die gleichen Befehle ausgeführt werden.

Im Gegensatz zu CMake, wo das Build-System in der Datei `CMakeLists.txt` definiert wird, besteht das Build-System bei SCons aus Python Datei(-en). Die Hauptdatei heißt `SConstruct`, in dieser wird die Build-Umgebung eingerichtet, d.h. Compiler-Flags werden gesetzt. Anschließend wird die Datei `SConscript` aufgerufen und dabei automatisch ein Build-Verzeichnis erstellt. In dieser Datei werden alle nötigen Source-files zusammen gesucht, ein Name für die gebaute Datei definiert und schließlich das Programm gebaut. Zwei Dateien sind nötig, um gleichzeitig ein Build-Verzeichnis haben zu können und die Dateien nicht bzgl. des Build-Verzeichnisses sondern ihres tatsächlichen Orts im Speicher zusammen suchen zu können; für weitere Informationen: https://bitbucket.org/scons/scons/wiki/VariantDir%28%29.
Zur Konfiguration, ob Sie eine Visualisierung bauen möchten oder nicht, verwenden Sie die Datei `custom.py`. Die Entscheidung, ob Sie zusätzliche Debug-Informationen zur Verwendung des gdb generieren lassen wollen, können Sie wiefolgt über einen Kommandozeilen-Parameter treffen:

`scons debug=1 -j` *AnzahlProzesse*.

[5] http://scons.org/doc/production/HTML/scons-user.html#idm139970758218448

1.5 Visualisierung

Zur Visualisierung der Simulationsergebnisse sind zwei Varianten möglich. Zum einen gibt es die Möglichkeit die vom Programm geschriebenen `.vtk`-Dateien mit den Simulationsdaten mit dem OpenSource Programm ParaView[6] einzulesen und zu visualisieren. Weiter unten geben wir hierzu eine kleine Anleitung.
Zusätzlich verfügt das Rahmenprogramm über eine SDL basierte live Visualisierung der Simulationsdaten.

Live-Visualisierung

Die vollständige Implementierung ist in der Klasse `Renderer` gegeben und muss nicht weiter modifiziert werden. Damit die Visualisierung fehlerfrei funktioniert müssen jedoch verwendete Funktionen wie `Grid::Interpolate()` u. a. korrekt implementiert werden. Die live Visualisierung wird in der main Funktion in `app.cpp` gestartet.

Anleitung ParaView

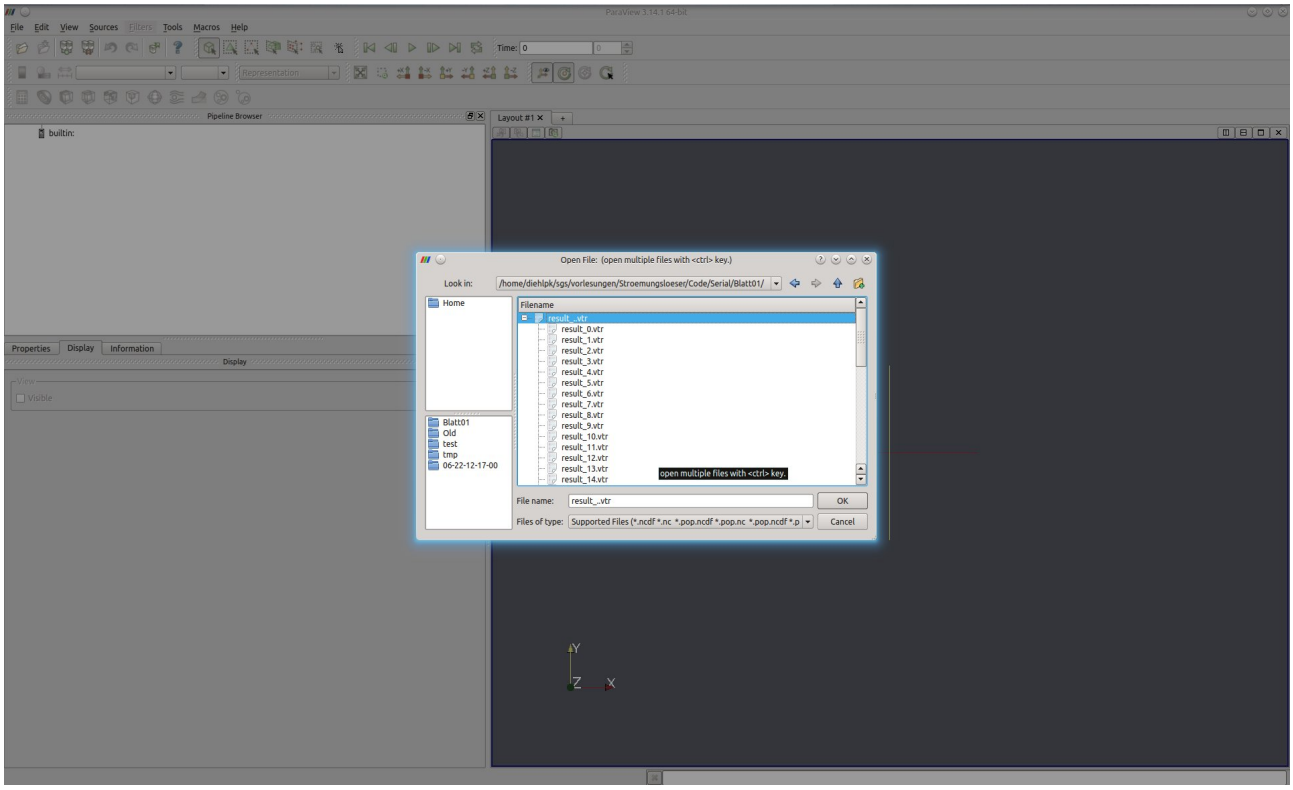


Abbildung 4: Um die Simulationsergebnisse gebündelt zu öffnen klicken sie auf *_..vts.

[6] http://www.paraview.org/

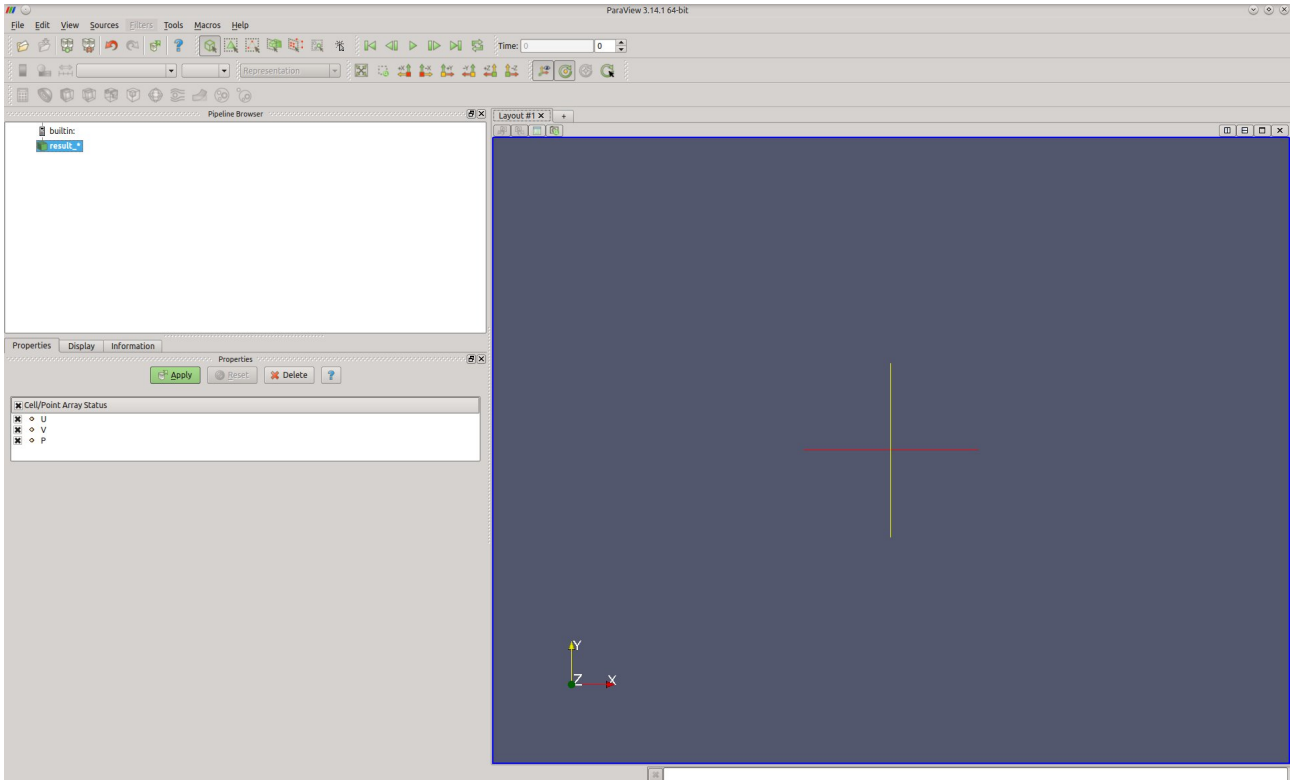


Abbildung 5: Im Reiter *Properties* klicken sie auf den Button Apply um die Ergebnisse zu visualisieren.

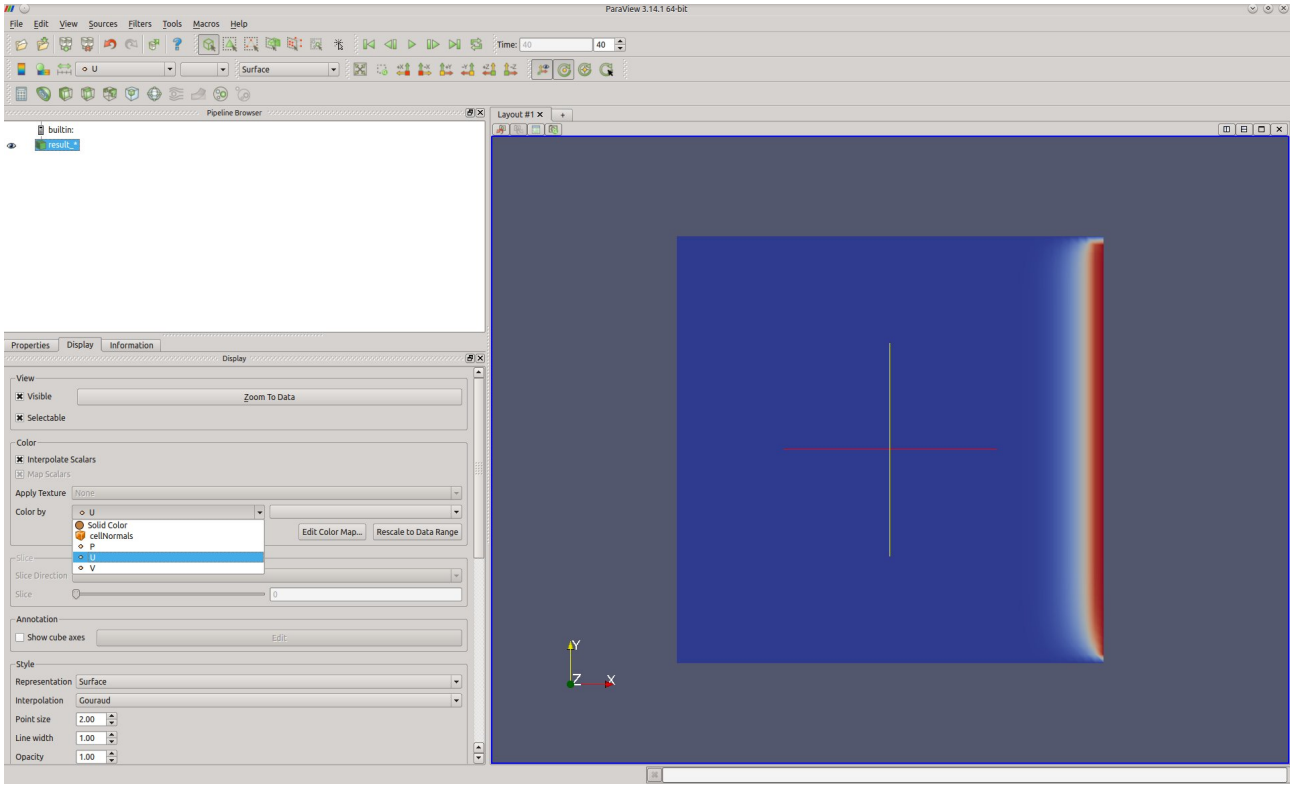


Abbildung 6: Im Reiter *Display* können sie unter *Color by* auswählen welcher Datensatz für das einfärben des rechteckigen Gitters verwendet werden soll.

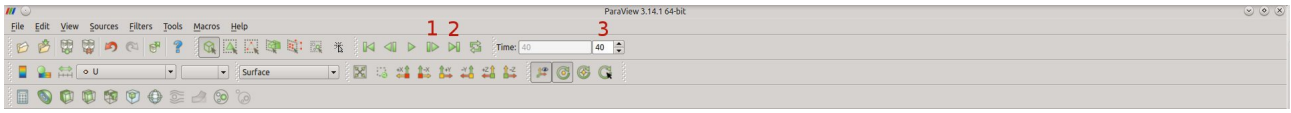


Abbildung 7: Mit dem Button 1 wird ein Zeitschritt weiter gesprungen, Button 2 springt zum letzten Zeitschritt und Textfeld 3 zeigt den akutellen Zeitschritt an.

1.6 Fragen

- Wählen Sie die Zeitschritte manuell. Testen Sie, für welche Zeitschrittweiten dt der Algorithmus stabil ist.
- Was passiert wenn die Reynoldszahl verändert wird ($Re = 100, 500, 2000, 10000$)?
- Wir haben in der Vorlesung behauptet, dass unsere Diskretisierung im Raum zweiter Ordnung (bei Verwendung zentraler Differenzen) beziehungsweise erster Ordnung (bei Verwendung des Donor-Cell Schemas), in der Zeit erster Ordnung genau ist. Wie könnten Sie das mit Hilfe Ihres Simulationsprogramms überprüfen?
- Erstellen Sie eine Tabelle der Laufzeiten für unterschiedliche Gitterauflösungen. Stimmen diese Ergebnisse mit unseren theoretischen Überlegungen aus der Vorlesung überein? Warum / warum nicht?