NTNU
Det skapende universitet

Beregningsprogram for sammenligning av flerfase strømningsmodeller

Marius Stene

Master i produktutvikling og produksjon
Oppgaven levert: Juni 2010
Hovedveileder: Ole Jørgen Nydal, EPT

Norges teknisk-naturvitenskapelige universitet
Institutt for energi- og prosessteknikk

Oppgavetekst

Bakgrunn
Flerfasestrøm i rør er svært kompleks strømning, og endimensjonale modeller bygger i stor grad på empirisk bestemte lukningsmodeller. Det er derfor mange forslag til delmodeller for fasefraksjoner, trykkfall og strømningsmønsteroverganger i litteraturen. Det er ønskelig å ha et rammeverk for å kunne sammenligne forskjellige modeller, og hvordan disse oppfører seg ved forskjellige betingelser (fluid, rørdiameter, rørvinkel og trykk). Det er spesielt vanskelig å sammenligne forskjellige forslag til strømningsmønsterkart, da disse ofte presenteres ved hjelp av forskjellige dimensjonsløse grupper. Det er også behov for enkelt å kunne sammenligne modeller og forsøksdata.

Det er tidligere utarbeidet et rammeverk i Matlab, med et grafisk brukergrensesnitt for å forenkle tilgjengeligheten også i en undervisningssammenheng. Dette viser seg å være tungt å vedlikeholde, og det foreslås derfor en aktivitet for å etablere et nytt rammeverk i en objektorientert struktur som tillater enklere videreutvikling med flere modeller. En mulighet er å lage en implementasjon i C++ med Qt GUI bibliotek.

Mål
Det skal utvikles et løsningsforslag til et rammeverk for sammenligning av forskjellige strømningsmodeller for flerfase rørstrøm: trykkfall, væskefraksjon og strømningsmønter.

Oppgaven bearbedet ut fra følgende punkter
Oppgaven er beregningsorientert, og vil omfatte en betydelig andel programmeringsarbeid.
1. Kravspesifikasjon for programmet, med identifisering av noen modeller som skal implementeres
2. Design av en programstruktur: klasser og metoder
3. Implementering og demonstrasjon så langt det lar seg gjøre innenfor den tilgjengelige tiden for en masteroppgave

Oppgaven gitt: 18. januar 2010
Hovedveileder: Ole Jørgen Nydal, EPT

Norges teknisk-
naturvitenskapelige universitet
NTNU

Institutt for energi- og prosessteknikk

NTNU

EPT-M-2010-62

MASTEROPPGAVE

for

Stud.techn. Marius Stene

Våren 2010

Beregningsprogram for sammenligning av flerfase strømningsmodeller

Computational framework for comparisons of multiphase flow models

Bakgrunn

Flerfasestrøm i rør er svært kompleks strømning, og endimensjonale modeller bygger i stor grad på empirisk bestemte lukningsmodeller. Det er derfor mange forslag til delmodeller for fasefraksjoner, trykkfall og strømningsmønsteroverganger i litteraturen. Det er ønskelig å ha et rammeverk for å kunne sammenligne forskjellige modeller, og hvordan disse oppfører seg ved forskjellige betingelser (fluid, rørdiameter, rørvinkel og trykk). Det er spesielt vanskelig å sammenligne forskjellige forslag til strømningsmønsterkart, da disse ofte presenteres ved hjelp av forskjellige dimensjonsløse grupper. Det er også behov for enkelt å kunne sammenligne modeller og forsøksdata.

Det er tidligere utarbeidet et rammeverk i Matlab, med et grafisk brukergrensesnitt for å forenkle tilgjengeligheten også i en undervisningssammenheng. Dette viser seg å være tungt å vedlikeholde, og det foreslås derfor en aktivitet for å etablere et nytt rammeverk i en objektorientert struktur som tillater enklere videreutvikling med flere modeller. En mulighet er å lage en implementasjon i C++ med Qt GUI biblioteket.

Mål

Det skal utvikles et løsningsforslag til et rammeverk for sammenligning av forskjellige strømningsmodeller for flerfase rørstrøm: trykkfall, væskefraksjon og strømningsmønster.

Oppgaven bearbeides ut fra følgende punkter

Oppgaven er beregningsorientert, og vil omfattende en betydelig andel programmeringsarbeid.

1. Kravspesifikasjon for programmet, med identifisering av noen modeller som skal implementeres
2. Design av en programstruktur: klasser og metoder
3. Implementering og demonstrasjon så langt det lar seg gjøre innenfor den tilgjengelige tiden for en masteroppgave

" - "

Senest 14 dager etter utlevering av oppgaven skal kandidaten levere/sende instituttet en detaljert fremdrift- og eventuelt forsøksplan for oppgaven til evaluering og eventuelt diskusjon med faglig ansvarlig/veiledere. Detaljer ved eventuell utførelse av dataprogrammer skal avtales nærmere i samråd med faglig ansvarlig.

Besvarelsen redigeres mest mulig som en forskningsrapport med et sammendrag både på norsk og engelsk, konklusjon, litteraturliste, innholdsfortegnelse etc. Ved utarbeidelsen av teksten skal kandidaten legge vekt på å gjøre teksten oversiktlig og velskrevet. Med henblikk på lesning av besvarelsen er det viktig at de nødvendige henvisninger for korresponderende steder i tekst, tabeller og figurer anføres på begge steder. Ved bedømmelsen legges det stor vekt på at resultatene er grundig bearbeidet, at de oppstilles tabellarisk og/eller grafisk på en oversiktlig måte, og at de er diskutert utførlig.

Alle benyttede kilder, også muntlige opplysninger, skal oppgis på fullstendig måte. For tidsskrifter og bøker oppgis forfatter, tittel, årgang, sidetall og eventuelt figurnummer.

Det forutsettes at kandidaten tar initiativ til og holder nødvendig kontakt med faglærer og veileder(e). Kandidaten skal rette seg etter de reglementer og retningslinjer som gjelder ved alle (andre) fagmiljøer som kandidaten har kontakt med gjennom sin utførelse av oppgaven, samt etter eventuelle pålegg fra Institutt for energi- og prosessteknikk.

I henhold til "Utfyllende regler til studieforskriften for teknologistudiet/sivilingeniørstudiet" ved NTNU § 20, forbeholder instituttet seg retten til å benytte alle resultater og data til undervisnings- og forskningsformål, samt til fremtidige publikasjoner.

Ett -1 komplett eksemplar av originalbesvarelsen av oppgaven skal innleveres til samme adressat som den ble utlevert fra. Det skal medfølge et konsentrert sammendrag på maksimalt én maskinskrevet side med dobbel linjeavstand med forfatternavn og oppgavetittel for evt. referering i tidsskrifter).

Til Instituttet innleveres to - 2 komplette kopier av besvarelsen. Ytterligere kopier til eventuelle medveiledere/oppgavegivere skal avtales med, og eventuelt leveres direkte til de respektive. Til instituttet innleveres også en komplett kopi (inkl. konsentrerte sammendrag) på CD-ROM i Word-format eller tilsvarende.

NTNU, Institutt for energi- og prosessteknikk, 12. januar 2010

Olav Bolland
Instituttleder

Ole Jørgen Nydal
Faglig ansvarlig/veileder

Forord

Denne masteroppgaven er utført ved Institutt for energi- og prosessteknikk ved Norges teknisk-naturvitenskapelige universitet, NTNU. Arbeidet er gjort over 21 uker våren 2010. Oppgavens hovedtema var å utvikle et rammeverk og program for beregning og sammenligning av flerfase strømningsmodeller.

Arbeidet er løst basert på en tidligere masteroppgave av Kristian Kjæstad. Veilederen for masteroppgaven har vært Professor Ole Jørgen Nydal. Jeg er takknemlig for hans utmerkede bistand under arbeidet.

Trondheim, 9. Juni 2010

Marius Stene

Sammendrag

Flerfasestrømning er et komplekst forskningsområde med mange usikkerheter. Kompleksiteten gjør at litteraturen består av mange modeller med vilt forskjellige innfallsvinker for hvordan fasefraksjoner, trykkfall og strømningsmønsteroverganger i slike strømninger skal modelleres. Forskjellene mellom disse modellene gjør at det er et behov for en enkel måte å sammenligne de med hverandre og med eksperimentelle data. I denne oppgaven er et rammeverk og et program som muliggjør slike sammenligninger planlagt, skrevet og dokumentert.

Ved oppgavens start eksisterte det et program for MATLAB laget av en tidligere masterstudent for beregning av flerfase strømningsmodeller. Programmets funksjoner var nokså begrenset og kildekodens dårlige kvalitet gjorde det vanskelig å legge til nye modeller. Målet var derfor å lage et program og rammeverk fra grunnen av i C++ med større fleksibilitet, større grad av objektorientering og bruk av moduler. En tredeling ble derfor valgt med et hovedprogram som tar imot inn-data, sender disse til beregningsprogramtilleggene og viser resultatene, et sett med uavhengige programtillegg hvor alle beregninger foregår og til slutt et støttebibliotek for å koble sammen de to førstnevnte delene.

Grensesnittene mellom hovedprogrammet og programtilleggene er slike at nye programtillegg kan utvikles uten endringer i kildekoden til hovedprogrammet. Etter at et programtillegg er lagt i korrekt mappe vil hovedprogrammet automatisk laste inn dette ved neste programstart og beregninger kan utføres med en gang.

Fem forskjellige typer programtillegg for ulike beregninger er støttet, disse står for beregning av friksjon mellom fasene og veggen, slipp-forhold, utregning av regimeoverganger og beregningsmetoder for hvert av regimene. Resultatet er et fleksibelt program som raskt kan utføre mange typer beregninger og sammenligninger.

Abstract

Multiphase flow is a complex area of research and the literature consists of many models with wildly different approaches to how phase fractions, pressure loss and regime transitions should best be modelled. These differences mean that there is a need for an easy way to compare these models, both with each other and with experimental data. In this thesis an application and a framework which simplifies such comparisons has been designed, programmed and documented.

Before work on the thesis begun there already existed an application which offered some of these features. The application was written in MATLAB as part of an earlier master thesis and had limited functionality and a very hard to maintain code base which made it difficult to improve the application. The goal therefore was to design and write an application from the ground up in C++ which would be more flexible and easily extendable. To accomplish this the application was split into three parts, a main program which handles input data, dispates it to the calculation modules and displays the results, a set of modules where all the calculations are done and a support library to facilitate the communication between the two first parts.

The interfaces between the main program and calculation modules are such that new modules can be created without modifying the source code of the main program. After a new module is placed in the correct folder it will automatically be ready for use upon next application start.

Five different calculation modules are supported, these are responsible for calculation of the friction between the phases and the wall, the slip ratio, regime transitions and the calculation approach used for a specific regime.

Innhold

Figurer

Tabeller

1 Bakgrunn

1.1 Behov

Flerfasestrømning er et komplekst forskningsområde med mange usikkerheter. Kompleksiteten gjør at litteraturen består av mange modeller med vilt forskjellige innfallsvinker for hvordan fasefraksjoner, trykkfall og strømningsmønsteroverganger i slike strømninger skal modelleres. Forskjellene mellom disse modellene gjør at det er et behov for en enkel måte å sammenligne de med hverandre og med eksperimentelle data.

1.2 Tidligere arbeid

Ved prosjektets start eksisterte det allerede et program ved navn MapIT utviklet ved NTNU av Kristian Kjæstad i en masteroppgave. Dette hadde flere av de samme funksjonalitetene som var ønskelige i det nye programmet. Kildekoden til dette programmet derimot var svært kaotisk og vanskelig å videreutvikle. MapIT var laget i MATLAB, et programmeringsspråk som kan gi raske resultater for små problemstillinger men som fort kan gi uoversiktelig programlogikk for større programmer. Med 6000 kildekodelinjer må MapIT betegnes som et nokså stort MATLAB-program. Programmet bar preg av at det var utviklet med funksjonaliteten framfor kildekodedesignet i forsetet. Ønsket man for eksempel å legge til en alternativ veggfriksjonsutregning for væske måtte denne kopieres inn som et alternativ alle stedene i koden hvor den ble brukt. Brukergrensesnittkoden var også svært primitiv og så ut til å ha fremprovosert en masse kopiering av kildekode rundt omkring. I det hele skrek programmet etter en mer objektorientert framgangsmåte.

Programmet hadde også mange begrensninger som vesentlig minsket dets bruksområde, det tillot for eksempel ikke andre variabler enn væske- og gasshastighet.

En annen klar ulempe med MapIT er jo selvfølgelig at MATLAB er påkrevd for å kjøre det, et program som kan koste mange titalls tusen kroner per bruker i kommersiell sammenheng.

2 Mål

Funksjoner Målet er å lage et nytt program og et rammeverk for simulering av fasefraksjoner og trykktap i flerfasestrøm i rør, programmet skal ha en fleksibilitet som tillater at enhver av strømningsparameterne (for eksempel væsketetthet, overflatespenning eller rørvinkel) skal kunne brukes som variabel, framfor at det skal være nødvendig å gjøre beregninger med andre variabler enn strømningshastigenene manuelt. Programmet skal også tillate beregning av strømningsmønsteroverganger (regimeoverganger). Alle disse, samt importerte forsøksdata skal programmet gjøre det enkelt å sammenligne.

Modeller Programmet skal implementere et sett med beregningsmodeller som til sammen eksemplifiserer hvilke type beregninger som kan utføres. Modellene bør være velkjente modeller i litteraturen. En viss overlapping med modellene implementert i MapIT kan være fordelaktig for testing og verifisering av simuleringsresultatene.

Brukergrensesnitt Programmet skal ha et enkelt brukergrensesnitt som muliggjør presentasjon av simuleringsdataene på en effektiv måte. Det bør designes på en slik måte at brukerflyten ikke er unødvendig kompleks. Det bør gli inn i blant de andre programmene i operativsystemet uten å skille seg ut.

Objektorientert Erfaringer fra en kort studie av kildekoden til MapIT har vist at riktig form på kildekoden og hvordan den er satt sammen har mye å si for hvor oversiktlig og effektiv den er. Det er derfor et krav at kildekoden skal utnytte objektorientering i den grad det gir mening for å hindre unødvendig kopiering av kildekode og for å gjøre kildekoden mer oversiktlig. Dette vil også gjøre det enklere for andre i framtiden å gjøre endringer i kildekoden uten at terskelen er for høy og at man nødvendigvis må sette seg inn i og gjøre endringer over hele koden.

Programmet bør også kunne utvides med nye beregningsmodeller uten at det heller krever for store endringer i selve programkoden, helst bør det kunne gjøres uten at programkoden behøver å endres i det hele tatt.

Bruksområder Det er ønskelig at programmet er så enkelt som mulig å ta i bruk. Dette vil gjøre at det kan tas i bruk i utdanningssammenhenger uten for mye arbeid. Dette betyr at det ikke bør gjøre seg avhengig av alt for mange andre programmer eller biblioteker. Spesielt gjelder det å unngå slike store og dyre programmer som MATLAB.

Plattform Et krav for programmet er at det fungerer på moderne utgaver av Microsoft Windows. Om programmet i tillegg til dette fungerer på Apples Mac OS X og Linux vil dette være en bonus.

3 Flerfasestrømning

3.1 Generelt

Flerfasestrømning refererer til strømmen av to eller flere forskjellige faser sammen. Det kan blant annet være væske-faststoff, væske-væske, gass-faststoff eller det vi her skal se på, gass-væske. Slike strømmer ser man i mange sammenhenger i industrien. Her til lands gjør det seg mest gjeldende i utvinningen og transporten av olje og gass fra offshore utvinningsplattformer[11]. Dette er et felt som får stadig større oppmerksomhet ettersom det blir vanligere med subsea-installasjoner for petroliumsutvinning med tilhørende flerfasetransport[2].

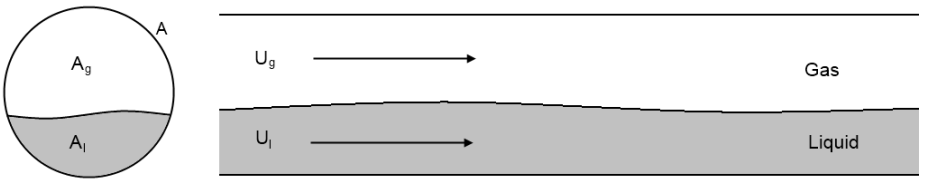
Flerfasestrømning skiller seg ut fra énfasestrømning i at strømningsbildet er mye mer komplekst og uforutsigbart. Mens det i enfasestrømning er et begrenset antall variabler å forholde seg til, er det i flerfasestrømning X ganger så mange variabler avhengig av hvor mange faser det er. I tillegg til dette, og kanskje enda viktigere, kommer interaksjonene mellom fasene. Hvilke(n) av disse interaksjonene som gjør seg dominerende kan variere med blant annet hastigheten til de forskjellige fasene og hvilke faser strømningen består av.

Symbol	Navn	Enhet
U_{sg}	Tilsynelatende væskehastighet	$[m/s]$
U_{sl}	Tilsynelatende gasshastighet	$[m/s]$
ρ_l	Væsketetthet	$[kg/m^3]$
ρ_g	Gasstetthet	$[kg/m^3]$
μ_l	Væskeviskositet	$[kg/sm]$
μ_g	Gassviskositet	$[kg/sm]$
σ	Overflatespenning	$[N/m]$
D	Rørdiameter	$[m]$
ϵ	Rørruhet	$[m]$
ϕ	Rørvinkel	$[°]$

Tabell 1: Karakteristiske parametre for tofasestrøm[10] som bør være tilgjengelig som variable i programmet

3.2 Viktige definisjoner og begreper

I flerfasestrømning er det en del sentrale begreper og definisjoner som er nødvendige for å kunne gi en bedre språklig og matematisk beskrivelse. Under er en figur som viser lagdelt flerfasestrømning men disse definisjonene gjelder alle strømningstyper. Ligningene er hentet fra Nydals forelesningsnotater om emnet.[12]



Figur 1: Flerfasestrømning i horisontalt rør

Tverrsnittet (A) kan deles opp i to deler, arealet som tas opp av væskefasen (A_l) og arealet som tas opp av gassfasen (A_g).

$$A_g + A_l = A \tag{1}$$

Andelen av rørtverrsnittet som blir tatt opp av gassfasen kalles «void fraction» (α) eller gassfraksjon, mens andelen som blir tatt opp av væskefasen kalles «holdup» (H) eller væskefraksjon.

$$\alpha = \frac{A_g}{A} \tag{2}$$

$$H = \frac{A_l}{A} \tag{3}$$

Ettersom det bare er to faser vil summen av gass- og væskefraksjonen naturlignok måtte være én.

$$\alpha + H = 1 \tag{4}$$

Ved beskrivelse av flerfasestrømning er det vanlig å oppgi hastighetene som tilsynelatende hastigheter, det vil si hastigheten hver fase hadde hatt ved den nåværende volumstrømmen dersom den var alene i røret. Verdien får man ved å multiplisere den faktiske fasehastigheten med arealandelen til fasen.

$$U_{sg} = \alpha U_g = (\frac{A_g}{A})U_g \tag{5}$$

$$U_{sl} = HU_l = (\frac{A_l}{A})U_l \tag{6}$$

Slipp-forholdet er ganske enkelt forholdet mellom gass og væskehastigheten

$$S = \frac{U_g}{U_l} \tag{7}$$

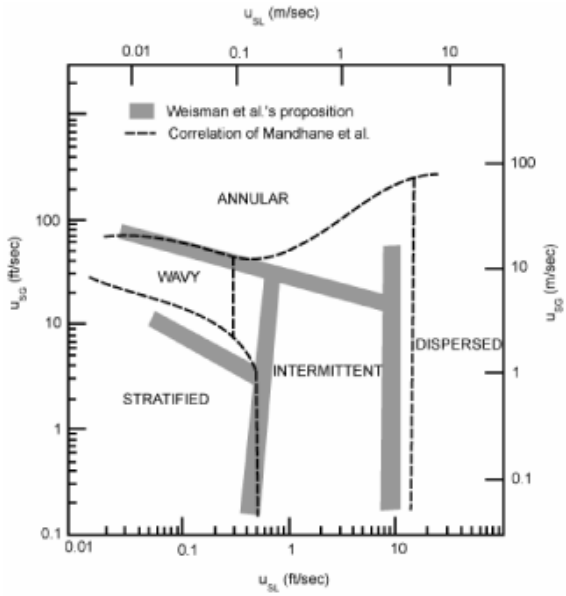
3.3 Regimer

3.3.1 Om regimer

Mens det i énfasestrømning er to typer strømning, laminær og turbulent, er det i flerfasestrømning mange flere. Lokalisering av under hvilke forhold de forskjellige regimene oppstår vil være én av oppgavene programmet er ment å assistere i.

Det er vanlig å dele opp horisontal flerfasestrømning i fire hovedstrømningsregimer[12]. Annulær-, boble-, lagdelt (stratifisert) og slugstrøm. Overgangen mellom regimene er ikke alltid like markerte og strømningene kan ha aspekter av flere regimer.

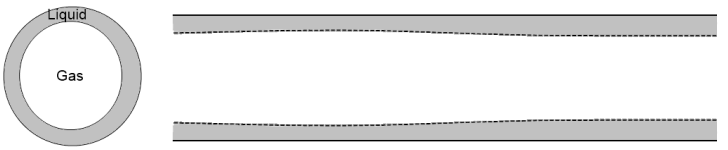
Disse regimene blir ofte plottet inn i et regimekart for å vise når hver av de vil oppstå. En type ofte brukt regimekart går ut på å plotte den tilsynelatende væskehastigheten (U_{sl}) mot den tilsynelatende gasshastigheten (U_{sg}), **figur 2** viser et eksempel på et slikt regimekart. Der ser vi at for lave tilsynelatende væskehastigheter får vi for det meste lagdelt strøm, mens det oppstår boblestrøm (dispersed) ved høye tilsynelatende væskehastigheter. Ved mellomhøye tilsynelatende væskehastigheter vil det ved høye tilsynelatende gasshastigheter være annulærstrøm og ved lavere tilsynelatende gasshastigheter slugstrøm.



Figur 2: Eksempel på regimekart med U_{sl} mot U_{sg} [13]

3.3.2 Annulær

Ved svært høye gasshastigheter vil flerfasestrømningen havne i det annulære regimet. Her domineres strømningen av gasstransporten og væsken blir liggende inntil veggen. I tillegg kan det være store mengder væske i form av dråper i gassfasen. Ved ekstreme gasshastigheter kan «tåkestrøm» oppstå hvor væskesjiktet langs veggen praktisk talt er borte.[1]

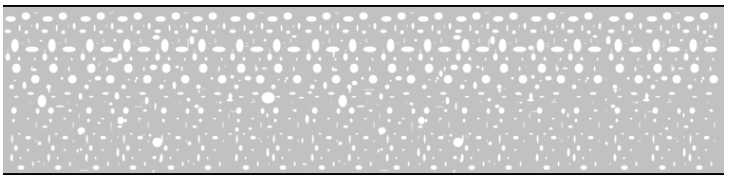


Figur 3: Annulær flerfasestrømning

I horisontale rør vil ikke tverrsnittet alltid se så idealisert ut som på bildet over, tyngdekraften vil bidra til at væskesjiktet nederst i røret vil være tykkere enn det på toppen.[3]

3.3.3 Boble

Ved høye væskehastigheter vil bevegelsene i røret være dominert av væskefasen, dette vil gi en strøm hvor gassfasen er distribuert rundt i væskefasen i form av bobler. For horisontale rør vil bobletettheten være høyere ved toppen av røret enn ved bunnen på grunn av oppdriften, for vertikale rør vil distribusjonen naturlig nok være tilnærmet uniform.



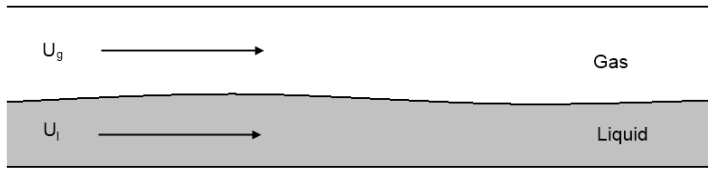
Figur 4: Bobleflerfasestrømning

Størrelsen på dråpene vil avhenge av flere faktorer, for vann vil det normalt være større dråper enn for olje ettersom vann har en høyere overflatespenning.[14]

3.3.4 Lagdelt

Lagdelt strøm er regimet som oppstår ved de laveste hastighetene. Lav U_{sl} er det viktigste kriteriet, kombinert med en ikke alt for høy U_{sg}. Ikke overraskende er det tyngdekraften som dominerer dette regimet, dette resulterer i et skille med den tyngste fasen på bunnen og den letteste fasen oppå.

Ved lav diameter og/eller sterk overflatespenning vil det oppstå tilfeller der overflatespenningen delvis tar over for tyngdekraften. Regimet vil da ligne litt mer på annulærstrøm med en overflate som buer oppover ved sideveggene.[7]



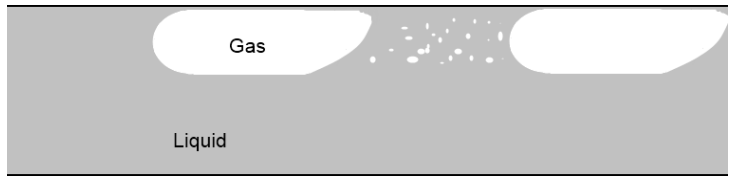
Figur 5: Lagdelt flerfasestrømning

En økt væskehastighet vil gi stadig større bølger på overflaten. Kombinert med høyere gasshastighet vil strømningen til slutt gå over til slugstrøm.[9]

3.3.5 Slug

Slugstrøm er en serie væske og gass-«slugs», det vil si områder som er nesten bare væske og så nesten bare gass. Dette regimet oppstår ved middels høye gass og væskehastigheter. Slugstrømmens karakteristikk er avhengig av rørformen. Mange opp og nedoverbakker i røret vil kunne gi en mer utagerende slugstrøm.

Dette er det regimet som oftest oppstår i petroliumsrørledninger, noe som blir reflektert i det store antallet publikasjoner om modellering av nettopp dette regimet.[11]



Figur 6: Slug flerfasestrømning

Lengden på sluggene varierer, men en typisk lengde er 15-40 ganger rørdiameteren.[4]

4 Programbeskrivelse

4.1 Arkitektur

Målene i **kapittel 2** gjorde det tydelig at programmet måtte være objektorientert og med løs kobling mellom beregningsfunksjonene og programlogikken. Som følge av dette har programmet blitt delt opp i tre separate hoveddeler.

Den første er selve programmet, «*MultiPhasePlot*», dette står for all vindusbehandlingen og brukergrensesnittet, lagrer og tar imot data fra bruker og plotter.

Den andre hoveddelen er et sett med programtillegg (Eng: plug-ins) i form av dll-filer (dynamic link libraries). Dette er hvor selve beregningene foregår. Hver enkelt fil står for én type beregning, for eksempel kan det være en fil som utfører beregninger av veggfriksjon med Blasius' metode, en annen som beregner trykktap og væskefraksjon for slugstrøm med «unit cell»-modellen og en siste fil som finner regimeovergangen mellom boble og slugstrøm etter Radovicich & Moissis' kriterier. Alle disse programtilleggene lastes inn automatisk av hovedprogrammet hvis de er plassert i riktige mapper. De får all informasjonen som er nødvendig for å utføre simuleringene fra hovedprogrammet; resultatene sendes deretter tilbake.

Den tredje og siste hoveddelen er et støttebibliotek som ligger som grunnlag for kommunikasjonen mellom hovedprogrammet og programtilleggene. Det spesifiserer grensesnitt som programtilleggene må følge for å kunne bli lastet inn av hovedprogrammet samt et sett med andre funksjoner som både hovedprogrammet og programtilleggene er avhengige av, for eksempel en klasse for behandling og lagring av data. Dette støttebiblioteket kalles «*MultiPhaseSupport*».

Denne framgangsmåten gjør at nye programtillegg kan utvikles uten å røre hovedprogrammet. Å tillate en slik enkel utvidelse av programmet med nye beregningsmetoder var et av hovedmålene for det nye programmet. Dette tillater også enkelt distribusjon og deling av nye programtillegg blant kolleger eller andre som ønsker å samarbeide. Med denne tilnærmingen vil det være langt færre grunner til å grave i kildekoden til hovedprogrammet, det vil kun være for å rette feil eller legge til helt nye programfunksjoner at dette vil være nødvendig.

Ettersom programtilleggene er de som utfører selve beregningen og de er uavhengige av hovedprogrammet, vil det ikke være noe i veien for å kunne gjenbruke disse gjennom å lage et nytt program med et annet brukergrensesnitt som fokuserer på andre aspekter av simuleringene hvis det skulle være ønskelig. Altså å erstatte *MultiPhasePlot* men å beholde *MultiPhaseSupport* og programtilleggene. Dette kommer selvfølgelig i tillegg til muligheten til å utbedre *MultiPhasePlot*.

4.2 Rammeverk

4.2.1 Qt

Programmet har blitt utviklet med Qt-bibliotekene som grunnlag. Qt er et programvareutviklingsrammeverk for C++ utviklet av Nokia (Trolltech) som tilbyr klasser som forenkler utviklingen av brukergrensesnittet, for eksempel vinduer, knapper og lister, i tillegg til klasser for utføring av ikke-brukergrensesnittrelaterte operasjoner som skriving til filer, nettverkskommunikasjon og trådbehandling. Det er tilgjengelig på en rekke plattformer, dette gjør at programmer utviklet med basis i Qt i mange tilfeller kan gjøres tilgjengelig for flere plattformer uten mye ekstraarbeid.

Qt støtter (offisielt) følgende plattformer[6]:

- Windows
- Mac OS X
- Linux/X11 & Embedded
- Windows CE
- Symbian
- Maemo

Qt er gratis og åpen kildekode (GNU LGPL & GPL) så programmer laget med hjelp av Qt vil ikke være avhengige av dyre ekstraprogrammer for å kjøre.

For å kjøre et program laget med Qt trenger ikke Qt å være installert, programmet er kun avhengig av at de riktige dll-filene er vedlagt, i dette tilfellet vil det dreie seg om ni filer på tilsammen omtrent 13 MB i komprimert form, en overkommelig størrelse.

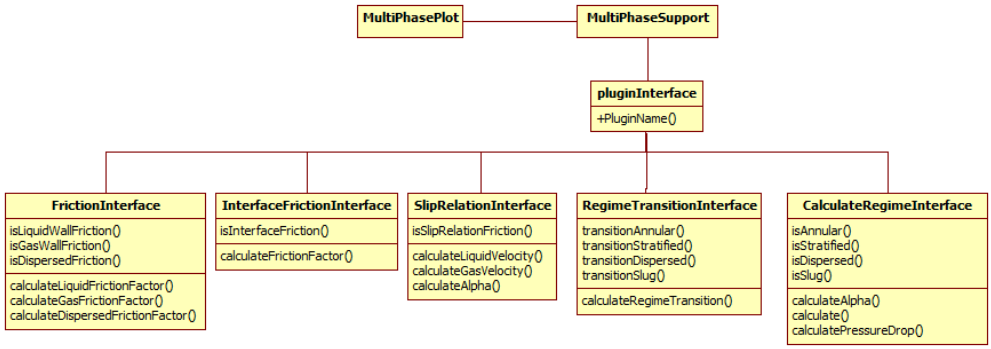
4.2.2 Qwt

Til tross for at Qt kommer med mange funksjonaliteter så tilbyr det ikke plottefunksjoner som er gode nok for et program som i så stor grad er avhengig av det som dette programmet. Derfor ble også Qwt benyttet. Qwt står for «Qt Widgets for Technical Applications» og er som navnet tilsier et sett brukergrensesnittenheter (Eng: widgets) spesiallaget for Qt og tekniske bruksområder, deriblant plotting. Ettersom det er laget for Qt så fungerer det nesten som om det var en del av selve Qt. Qwt er også åpen kildekode men i motsetning til Qt så er det ikke noe selskap som står bak det, i stedet er det en gruppe entusiaster som har utviklet det over de siste 10 årene.

De funksjonene fra Qwt som ble brukt i programmet er flerfarget plotting med zooming, tegnforklaring, logaritmisk/linear akseskala og eksportering av plott til bildefiler.

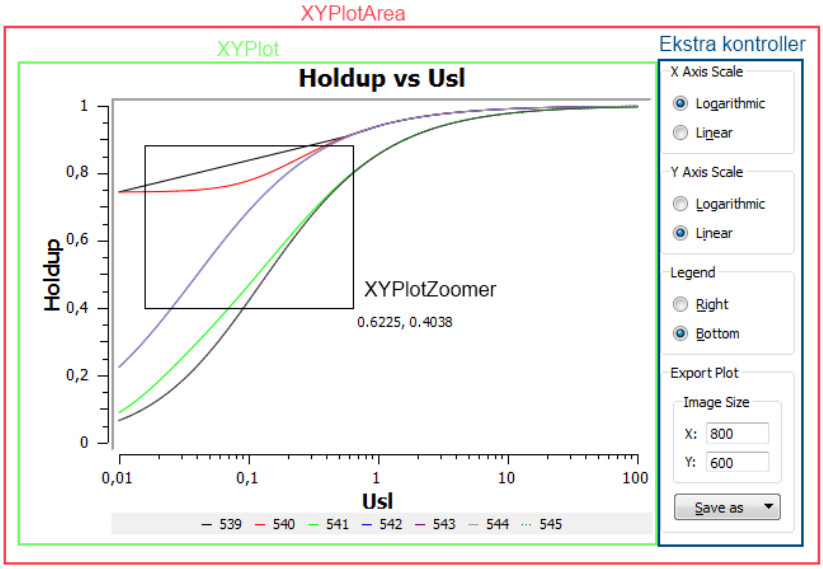
4.3 Objektorientering og arv

I programmet er det brukt objektorientering med arv hvor det gir fordeler. Det mest åpenbare eksempelet er programtilleggene. Her vil et implementert programtillegg måtte arve én av de fem «interface»-klassene som igjen arver den generelle *pluginInterface*-klassen.



Figur 7: Programtilleggarkitekturen med offentlige funksjonskall

Et annet eksempel på objektorientering i programmet er *XYPlotArea*-klassen som brukes fire steder i programmet. Denne tar seg av alt rundt plottingen av grafer. Den tillater endring av akseskaleringen, plassering av tegnforklaring og exportering av grafer til bildefiler. Ved å plassere alle disse funksjonene i én klasse (ett objekt) så vil alle stedene i programmet hvor denne klassen blir brukt automatisk få alle disse funksjonene. Klassen består av *XYPlot*, et plotteområde arvet fra Qwt sin *QwtPlot* med visse endringer, *XYPlotZoomer*, en spesialversjon av zoomefunksjonen, arvet fra Qwt sin *QwtPlotZoomer*, samt en del knapper og skrivefelt fra Qt for å konfigurere plottingen.



Figur 8: Objektorienteringseksempel, XYPlotArea

4.4 Programtillegg

4.4.1 Om programtilleggene

Det er fem ulike typer programtillegg. Hver av disse har en egen mal som må følges for å være et fullgodt programtillegg som kan lastes inn av hovedprogrammet.

- Friksjonsfaktor (f)
  - Friksjonsfaktor vegg
  - Friksjonsfaktor overflate
  - Slipp-forhold
- Regimeovergang ($U_{sl} = f(U_{sg})$)
- Beregningsmetode (α og ΔP)

Tilleggene er laget på en slik måte at de er agnostiske ovenfor hverandre. For eksempel vil ethvert regimeovergangtillegg kunne benytte seg av et hvilket som helst friksjonsfaktor-tillegg. Dette gjelder også tillegg som blir lagt til i ettertid.

Ved svært spesielle tilfeller vil det kunne være nødvendig å lage et programtillegg med så spesialiserte krav at ikke alle programtilleggene vil fungere sammen med det, i så tilfelle vil programmet gi beskjed om at de valgte programtilleggene ikke fungerer sammen.

4.4.2 Friksjonsfaktorprogramtillegg

Det eksisterer mange måter å beregne friksjonsfaktoren på. I programmet er det lagt til rette for tre ulike typer programtillegg som beregner friksjonsfaktoren. Den første beregner friksjonsfaktoren mot vegg for gassfasen, væskefasen og bobler. Den andre beregner friksjonsfaktoren mellom de to fasene og den tredje regner strengt tatt ikke ut noen friksjonsfaktor, men slipp-forholdet, men den blir like fullt gruppert i friksjonsfaktortilleggruppen på grunn av andre likheter.

4.4.3 Regimeovergangprogramtillegg

Regimeovergangprogramtilleggrensesnittet er laget slik at alle typer regimeoverganger skal kunne beregnes. Det er opp til programtilleggene selv å identifisere hva slags overganger de simulerer. Programmet vil gruppere programtilleggene i brukergrensesnittet basert på dette. Et eksempel på dette kan sees i **figur 9a** på side 17.

4.4.4 Beregningsmetodeprogramtillegg

Beregningsmetodeprogramtilleggene står for det meste av beregningene for de ulike regimene. Koblingen mot programtilleggene er laget slik at den er så fleksibel som mulig. Dette gir programtilleggene tilgang på all informasjon om strømmen samt tilgang til å benytte de andre valgte programtilleggene. Dette er viktig siden det er mange forskjellige framgangsmåter å gjøre de nødvendige beregningene på.

Som resultat fra disse beregningene gir programtilleggene væskefraksjon og trykktap for strømmen. I tillegg har programtilleggkoblingen en ekstra funksjon som lar programmet hente ut all ekstra informasjon fra beregningene for spesielle bruksområder. Det er bruken av denne veldig fleksible funksjonen som vil kunne gi programtillegg som ikke fungerer sammen med alle andre programtillegg slik som omtalt i **avsnitt 4.4.1**.

4.5 Funksjoner

4.5.1 Strømningssimulering

En av programmets to hovedfunksjoner er å kunne plotte kurver for væskefraksjon og trykktap for en valgfri variabel. Alle simuleringsparametrene kan fungere som denne variabelen. Det vil si:

- U_{sl} (Tilsynelatende væskehastighet)
- U_{sg} (Tilsynelatende gasshastighet)
- Gass- og væsketetthet
- Gass- og væskeviskositet

- ~~Gasstemperatur og trykk~~ [1]
- Overflatespenning
- Rørvinkel
- Rørdiameter
- Rørruhet.

Å kunne velge variabel på denne måten gir stor fleksibilitet i forhold til hvilke utregninger og undersøkelser man kan gjøre med programmet.

For hver simulering kan det velges hvilke programtillegg som skal brukes og hvilket regime beregningene skal foretas for. Det gjelder både hvilke friksjonsfunksjoner som skal brukes, hvilket regime det skal beregnes for og hvilken tilnærming som skal brukes for å utføre beregningene for dette regimet.

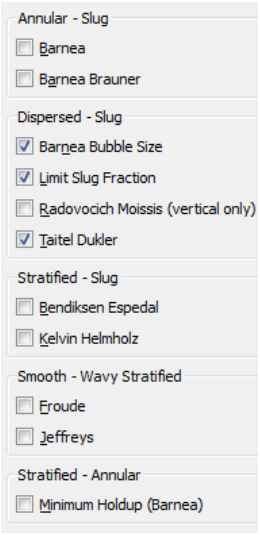
I stedet for å velge et spesifikt regime er det også mulig å velge «Minimum holdup» sammen med hvilke av regimene Minimum holdup-funksjonen skal benytte. Simuleringen vil da for hver enkel variabelverdi gi resultatet for det regimet med den laveste væskefraksjonen.

4.5.2 Regimeoverganger

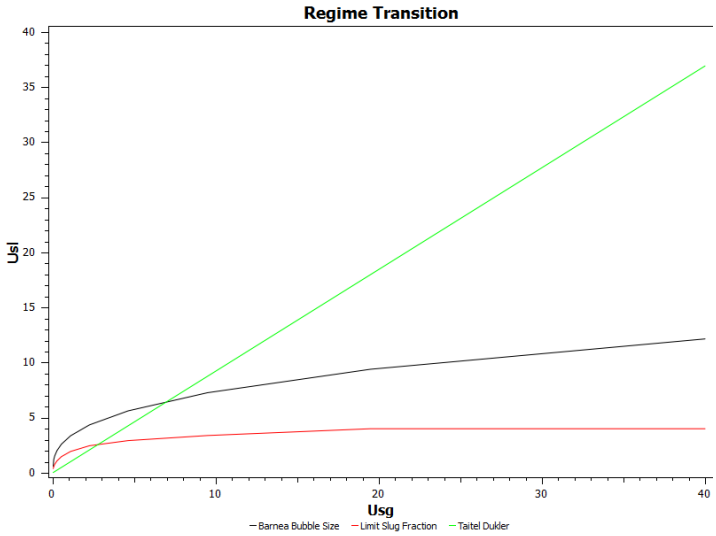
Den andre hovedfunksjonen til programmet er å plotte regimeoverganger mellom to forskjellige regimer. Programmet er videre lagt opp slik at det er lett å sammenligne disse forskjellige overgangene. Dette er praktisk å kunne gjøre fordi det ofte er stor usikkerhet i hvilken av regimeovergangskriteriene som gir rett resultat for en spesifikk regimeovergang.

I tillegg til disse funksjonene har hvert regimeovergangprogramtillegg en funksjon som returnerer regimet ved en gitt væske- og gasshastighet. Denne funksjonen blir ikke benyttet av programmet slik det er nå, men den vil være fin å ha hvis det senere skulle være ønskelig å implementere en funksjonalitet hvor regimet velges automatisk for hver enkelt variabelverdi basert på hvilke programtillegg som er valgt. Minimum holdup-funksjonen kan sees på som en begrenset og litt mer manuelt utført versjon av dette.

[1] Ikke implementert, beregningene utføres basert på gassviskositet og tetthet



(a) Liste over tilgjengelige regimeovergangs-kriterier



(b) Plot av valgte regimeovergangskriterier

Figur 9: Tre regimeoverganger mellom boble- og slugstrøm plottet sammen

4.5.3 Innlasting av tidligere utførte simuleringer

Alle simuleringer foretatt blir lagret til senere bruk. Simuleringer gjort i siste arbeidsøkt vil automatisk dukke opp i en liste med en oppsummering av parametrene brukt i simuleringen. Herfra kan et ubegrenset antall simuleringer markeres for plotting. Det er dermed veldig lett å sammenligne forskjellige gjennomførte simuleringer.

Ved å merke av «Show old simulations»-avkryssningsboksen vil listen fylles med simuleringer fra alle tidligere arbeidsøkter også. Det er selvfølgelig mulig å slette unna simuleringer som ikke lenger er ønsket i listen.

Sammen med resultatene fra simuleringene blir også parametrene lagret. Ved å markere en simulering i simuleringshistorikklisten og trykke «Load parameters»-knappen blir disse lastet inn til bruk i nye simuleringer.

4.5.4 Importering og eksportering av data

Måten simuleringer blir lagret på og lastet inn gjør det enkelt å eksportere simuleringsdata til annen bruk og å importere eksperimentelle data for sammenligning med simuleringsresultatene. En slik sammenligning vil kunne være en sentral del av analysen av resultatene

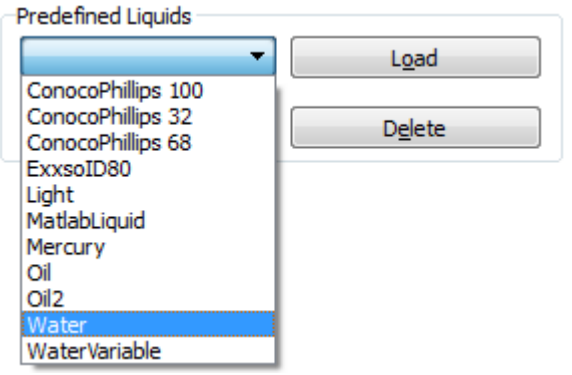
og det er derfor viktig at denne funksjonen er tilgjengelig. For å importere eksperimentelle data behøver de bare å være lagret i en tekstfil i den riktige mappen og de vil automatisk bli lastet inn.

Eksportering av data er enda enklere da dette skjer automatisk som en del av lagringen av gamle simuleringer til framtidig bruk i programmet. For å bruke disse dataene i en annen sammenheng er det bare å navigere seg til riktig mappe, simuleringsdataene ligger der i filer i et populært tekstformat beregnet for akkurat slike data kalt «comma-separated values». Data som skal importeres må bruke det samme formatet. Formatet ble valgt fordi det støttes av langt flere programmer enn for eksempel Excel-formatet xls og er lettere å arbeide med siden det er ren tekst.

4.6 Tilgjengelighet & brukervennlighet

Programmet inneholder en rekke funksjoner som forenkler og effektiviserer bruken av programmet og som gjør det mest mulig selvforklarende og tilgjengelig for alle.

Forhåndsdefinerte gasser og væsker Å raskt kunne bytte mellom forskjellige gasser og væsker effektiviserer arbeidet i programmet så derfor er en slik funksjon implementert. Den tillater at man lagrer de nåværende dataene for en gass eller en væske ved å gi den et navn. Alle lagrede gasser og væsker kan hentes fram ved hjelp av et par tastetrykk. Programmet kommer med et sett vanlige gasser og væsker allerede definert.



Figur 10: Skjermbilde, forhåndsdefinerte og brukerlagrede væsker

Tilstandsbevaring Noe av det mest irriterende med brukergrensesnittet til MapIT var hvordan tilstanden til programmet ikke ble bevart til neste gang programmet ble åpnet, enten etter en krasj eller vanlig lukking av programmet. *MultiPhasePlot* ble derfor laget slik at alle verdier og konfigurasjoner blir bevart på tvers av programomstarter. Tidligere simuleringer blir også bevart slik omtalt i **avsnitt 4.5.3**.

Informasjonsboble Ved å holde musen over en del av programmet som framstår uforståelig eller man ønsker mer informasjon om vil en informasjonsboble dukke opp. Denne gir informasjon om hva for eksempel en knapp gjør eller i regimeovergangavkryssningsboks-tilfellene, litt bakgrunnsinformasjon om beregningsmetoden og litteraturreferanser.

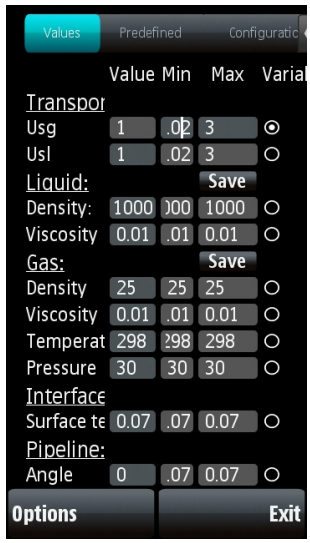
Statusfelt Gir informasjon om simuleringsstatusen og også et sammendrag av informasjonsbobleteksten for raskere tilgang.

Tastatursnarveier Det er flere grunnen til at programmet har implementert tastatursnarveier over hele programmet, ikke bare gjør det bruken raskere for erfarne brukere, det åpner også for at programmet kan brukes uten mus, bra for effektiv bruk på bærbare datamaskiner og for funksjonshemmede. Med tastatursnarveier menes ikke bare de vanlige menysnarveiene som mange er kjent med som **Ctrl+S** for å lagre, men også snarveier for alle knapper og avkryssningsbokser i hele programmet. For eksempel hvis vi ser på understrekingen av «o» og «e» i teksten på knappene på **figur 10** på side 18 betyr det at ved å trykke **Alt+O** vil «Load»-knappen bli trykket inn og **Alt+E** vil trykke inn «Delete»-knappen.

En oppmerksom leser vil kanskje legge merke til at til og med regimeovergangavkryssningsboksene i **figur 9a** på side 17, som ikke er en del det faste brukergrensesnittet men som legges til automatisk ved hver oppstart basert på hvilke regimeovergangprogramtillegg som blir lastet inn, har slike tastatursnarveier. Dette er fordi programtilleggrensesnittet lar programtilleggene spesifisere hurtigtasten som programmet skal gi til avkryssningsboksen.

4.7 Plattformer

Programmet ble utviklet på og for Microsoft Windows og fungerer uten problemer på alle nyere utgaver av Windows. I tillegg, ettersom Qt er tilgjengelig for MacOS X og Linux og programmet er skrevet på en plattformagnostisk måte så fungerer programmet, etter noen små endringer, på disse også. Faktisk fungerte tidlige utgaver av programmet på moderne Nokia og Samsung-telefoner også, men videre eksperimentering med dette ble nedprioritert fordi bruksområdet er nærmest ikke-eksisterende.



Figur 11: Skjermbilde, programmet kjører på Nokia 5800

5 Modeller

Under utviklingen av programmet var det viktig å ha programtillegg som kunne brukes til å teste og å feilsøke programmet. De gjorde det også enklere å se hvilke fasiliteter programtilleggrensesnittene måtte ha for å støtte alle mulige framtidige modeller. Tilsammen 24 programtillegg ble laget, hvorav de fleste også var tilgjengelige i MapIT. Dette gjorde det mulig å sammenligne simuleringsresultatene for å verifisere at det ikke var noen implementeringsfeil.

Selv om programmet er lagt til rette for at det skal være veldig enkelt å produsere egne programtillegg er det viktig at det tilbyr god funksjonalitet «ut av boksen» også. Derfor er alle disse programtilleggene lagt ved programmet.

En oppskrift på hvordan man lager egne programtillegg er tilgjengelig i **avsnitt 8.1**

Veggfriksjon	Overflatefriksjon	Slipp-forhold	Beregnings-metode	Regimeovergang
Blasius	Haaland Ext.	Malnes	Annular	Barnea
Colebrook	Russel	Slip 1	Dispersed	Barnea Brauner
Haaland		Slip 2	Stratified	Barnea Bubble Size
Moody			Slug	Bendiksen Espedal
				Froude
				Jeffreys
				Kelvin Helmholz
				Limit Slug Fraction
				Minimum Holdup
				Radovicich Moissis
				Taitel Dukler

Tabell 2: Vedlagte programtillegg

6 Bruk

En kort video som viser *MultiPhasePlot* i bruk er gjort tilgjengelig på Internett for å eksemplifisere ovenfor potensielle framtidige brukere hva programmet kan brukes til. [2]

6.1 Strømningssimulering

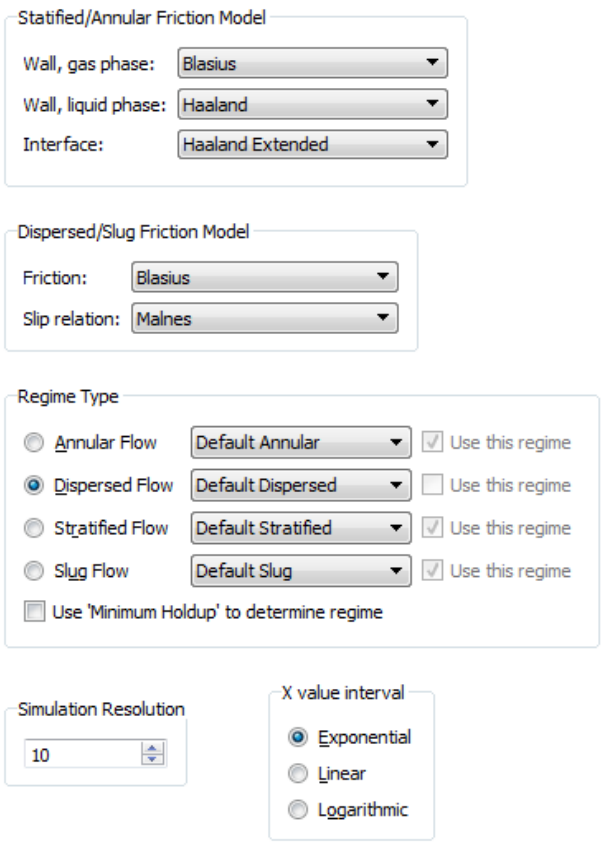
Første steg for å gjennomføre en simulering er å taste inn strømningsparametrene i *Values*-fanen, denne fanen er vist i mobiltelefonutgave i **figur 11**. Hvis væsken og gassen som det skal simuleres for allerede er lagt inn som forhåndsdefinerte faser, så kan disse lastes inn, hvis ikke må verdiene tastes inn manuelt sammen med de resterende parametrene som rørdiameter og vinkel. I tillegg må variabelen velges og maksimum og minimumsverdier for denne tastes inn.

Neste steg er å bevege seg til *Configuration*-fanen. Denne er vist i **figur 12**. Her er det innstillinger for tre aspekter av simuleringen. Øverst velges friksjonsberegningsmetodene, totalt fem forskjellige typer, noen av disse utføres av samme programtillegg men de kan likevel velges uavhengig. Deretter må et av de fire regimene velges eller man kan velge *Minimum Holdup*, noe som betyr at for hver variabelverdi blir det av de valgte regimene som gir den laveste væskefraksjonen brukt. Når *Minimum Holdup* er valgt vil avkryssningsboksene til høyre i grupperingsboksen bli aktivert slik at det er mulig å velge hvilke regimer *Minimum Holdup*-funksjonen skal ha som alternativer. I tillegg til å velge regime må også beregningsmetoden for dette regimet velges. Programmet kommer bare med én beregningsmetode for hvert regime, så dette er lite aktuelt med mindre den som bruker programmet har laget et alternativt beregningsmetodeprogramtillegg.

Det siste steget er valg av simuleringsoppløsning, for vanlig strømningssimulering har dette lite å si ettersom en oppløsning på for eksempel 200 går raskt å simulere, simulering av regimeovergang derimot er mer krevende så en oppløsning på rundt 30 er der å anbefale, i alle fall i eksperimenteringsfasen. I tillegg kan det velges hvordan denne oppløsningen skal være fordelt. *Exponential* anbefales med mindre det er veldig store endringer nært variabelens maksimum.

Til slutt er det bare å trykke *Simulate*-knappen i *Flow Simulation*-grupperingsboksen.

[2] http://www.youtube.com/watch?v=iQW-FyxDfEM&hd=1



Figur 12: Skjermbilde, konfigurasjonsmenyen

6.2 Regimeoverganger

Ved simulering av regimeoverganger er programbruken ganske lik som for vanlig strømningssimulering. Alle simuleringsparametrene må tastes inn på samme måte, men U_{sg} må velges som variabel, dette fordi programmet bare støtter plotting av regimekart for U_{sg} mot U_{sl}. Ved forsøk på å simulere regimeoverganger med en annen variabel vil det bli gitt beskjed om at dette ikke er støttet.

Deretter må simuleringen konfigureres på samme måte som strømningssimuleringen men valget av regime vil naturlig nok ikke ha noen effekt ettersom vi skal finne regimeovergangen mellom regimene spesifisert av de valgte regimeovergangprogramtillegget.

Siste skritt er å gå til *Regime Transition*-fanen. Her velges de regimeovergangene som skal simuleres. **Figur 9a** viser hvordan denne fanen ser ut.

Til slutt er det bare å trykke *Simulate*-knappen i *Regime Transition*-grupperingsboksen.

7 Demonstrasjon

7.1 Simuleringsbakgrunn

Som en demonstrasjon på hva programmet kan brukes til presenteres her et kort eksempel. Vi skal se på hvilken innvirkning valget av veggfriksjonsmodeller for gass og væske har på væskefraksjonen og trykktapet i et rør ved forskjellige gassviskositeter. I simuleringene vil gassviskositeten derfor bli brukt som variabel mens alt annet blir holdt konstant med unntak av bytte av friksjonsmodeller.

Alle simuleringene ble foretatt som om strømmen var stratifisert. Som basis for væske og gassparametrene ble vann og luft brukt, strømningsparametrene er som følger.

Parameter	Verdi	Enhet
Rørvinkel	0	°
Rørdiameter	2.2e-2	m
Rørruhet	1.9e-5	m
U_{sg}	4	m/s
U_{sl}	0.6	m/s
Gasstetthet	1.275	kg/m^3
Gassviskositet	Variabel	
Væsketetthet	1000	kg/m^3
Væskeviskositet	8.901e-4	Ns/m^2
Overflatespenning	7.28e-2	N/m
Minimumsverdi gassviskositet	1e-5	Ns/m^2
Maksimumsverdi gassviskositet	1e-2	Ns/m^2

Tabell 3: Strømningsparametre

Det ble foretatt simuleringer med to forskellige veggfriksjonsmodeller for gass («gassmodeller» fra nå av) og to forskjellige veggfriksjonsmodeller for væske («væskemodeller» fra nå av). Totalt gir dette fire mulige kombinasjoner, som også er antallet simuleringer som ble utført.

Simulerings-ID	Gassmodell	Væskemodell	Kurvefarge
101	Blasius	Blasius	Svart
102	Haaland	Blasius	Rød
103	Blasius	Haaland	Grønn
104	Haaland	Haaland	Blå

Tabell 4: Gjennomførte simuleringer

Mens gass- og væskemodellene ble endret mellom simuleringene ble de resterende tre friksjonsmodelltypene ikke endret. Det samme gjelder selvfølgelig beregningsmetoden for stratifisert strømning ettersom programmet bare kommer med én.

Modelltype	Modell
Overflatefriksjon	Haaland Extended
Boblefriksjon	Blasius
Slipp-forhold	Malnes
Beregningsmodell stratifisert	«Default»

Tabell 5: Faste programtillegg

Figurene presentert i resultatavsnittet er begge hentet direkte fra *MultiPhasePlot* ved hjelp av muligheten til å lagre plottene som bildefiler.

7.2 Modellbakgrunn

Her er formlene som er brukt av modellene.

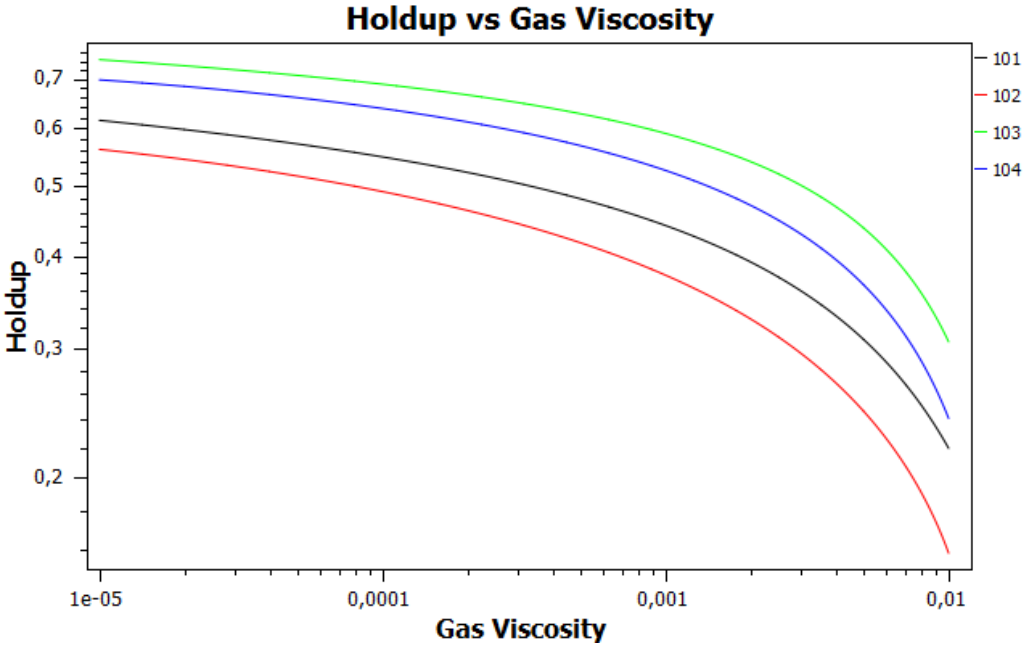
Blasius [5] $f = 0.046Re^{-0.2}$

Haaland [8] $\frac{1}{\sqrt{f}} = -1.8 \log_{10}\left[\left(\frac{\varepsilon/D}{3.7}\right)^{1.11} + \frac{6.9}{\mathrm{Re}}\right]$

7.3 Resultater

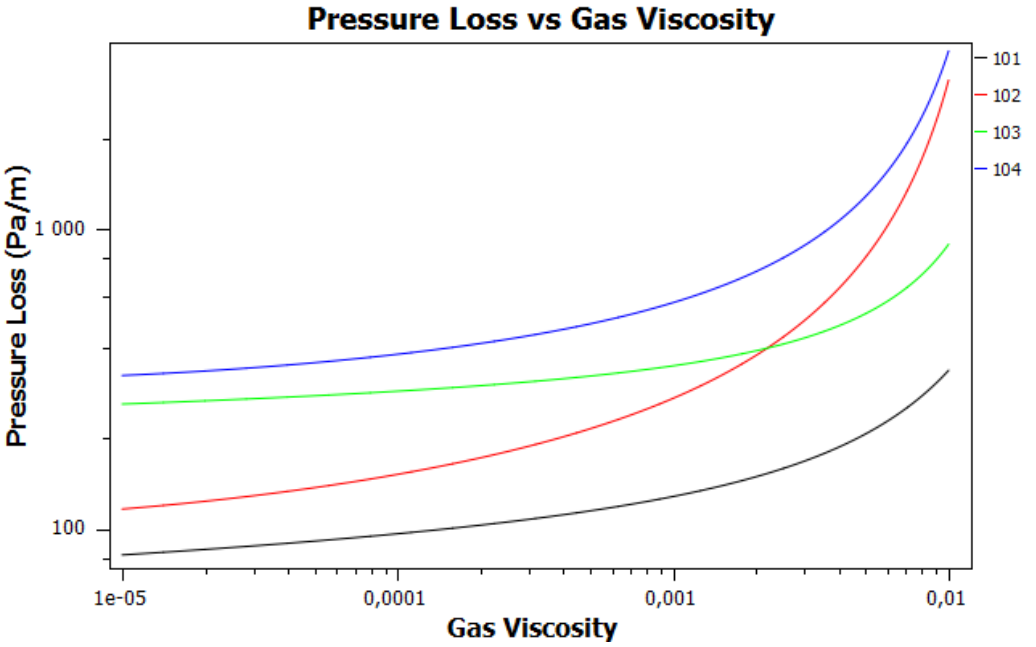
Figur 13 viser væskefraksjonen for de fire simuleringene ved forskjellige gassviskositetsverdier i et log-log-plott. Vi ser at alle kurvene har samme form men at kurvene med Blasius' væskemodell (svart og rød) jevnt over gir en lavere væskefraksjon. For gassmodellene er det motsatt, der er det kurvene med Haalands gassmodell (rød og blå) som gir lavest væskefraksjon.

Forskjellen mellom modellene er ganske stor, spesielt ved en gassviskositet på rundt 0,003 hvor kombinasjonen Blasius - Haaland som henholdsvis gass- og væskemodell gir en omtrent dobbelt så høy væskefraksjon som Haaland - Blasius gir. Det skal sies at reynoldstallet ved disse simuleringene ikke alltid er i friksjonmodellenes optimale område, så noe av avvikene skyldes nok dette.



Figur 13: Væskefraksjon plottet mot gassviskositet

Figur 14 viser trykktapet for de fire simuleringene ved forskjellige gassviskositetsverdier. Ved lave gassviskositeter er det ikke overraskende væskemodellen som har mest betydning og vi ser at kurvene grupperes etter dette. På **figur 13** så vi at Blasius' væskemodell gav lavere væskefraksjoner og her ser vi at den også gir lavere trykktap. Når gassviskositeten stiger til et nivå hvor den blir dominerende gir Haalands gassmodell et mye høyere trykktap enn Blasius'.



Figur 14: Trykktap plottet mot gassviskositet

8 Vedlikehold

8.1 Produksjon av programtillegg

Det finnes to forskjellige framgangsmåter når et nytt programtillegg skal lages. Den måten som krever minst innsikt er å kopiere kildekodemappen til et allerede eksisterende programtillegg, for så å gå gjennom de fire prosjekt- og kildekodefilene og endre på navnene brukt samt selve utregningsfunksjonen. Det førstnevnte kan gjøres unna på noen få minutter med hjelp av søk-og-erstatt-funksjonen som ofte er innebygd i de fleste tekst-redigeringsverktøy brukt i programmeringssammenheng.

Den andre framgangsmåter er å lage programtillegget fra grunnen av, et kort sammendrag av hvordan dette gjøres vil bli presentert her, i **tillegg A** er det en mer grundig gjennomgang av prosessen.

For å lage et programtillegg må funksjonene som i det aktuelle programtilleggrensesnittet er definert, men ikke implementert (de som er «pure virtual»), implementeres. En titt på definisjonsfilene til en av de aktuelle grensesnittene vil vise alle disse funksjonene. Her er definisjonen til *InterfaceFrictionInterface*, valgt fordi den er den enkleste. Alle funksjonen med «= 0» på enden må implementeres. Grensesnittene er alle en del av *MultiPhaseSupport*.

```
class InterfaceFrictionInterface : public PluginInterface
{
public:
        virtual ~InterfaceFrictionInterface() {};

        virtual double calculateFrictionFactor(Data const &data, double alpha,
                        double reynoldsGas,double hdiameterGas,double thicknessFilm,
                        double frictionFactorGas) const = 0;
        virtual bool isInterfaceFriction() const = 0;
};
```

Figur 15: InterfaceFrictionInterface header fil

Vi ser at denne klassen arver *PluginInterface* så vi må åpne denne filen også for å se om også den har funksjoner som må implementeres.

Alle disse funksjonene må defineres (uten «= 0») i header-filen i prosjektet og så implementeres. En enkel implementering kan se slik ut:

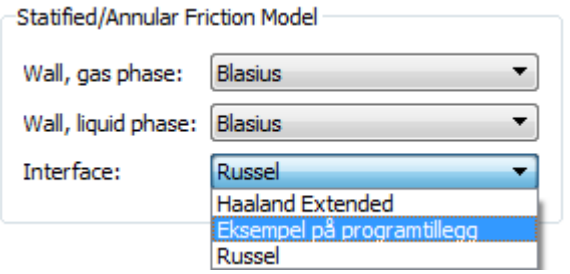
```
double ProgramTilleggEksempel::calculateFrictionFactor(Data const &data,
                double alpha, double reynoldsGas, double hdiameterGas,
                double thicknessFilm, double frictionFactorGas) const
{
    return frictionFactorGas*20;
}

QString ProgramTilleggEksempel::pluginName() const {
    return QString("Eksempel på programtillegg");
}

bool ProgramTilleggEksempel::isInterfaceFriction() const {
    return true;
}
Q_EXPORT_PLUGIN2(../../../application/MultiPhase/debug/lib/
                 interfacefriction/programtilleggeksempel,
                 ProgramTilleggEksempel);
```

Figur 16: Implementering av InterfaceFrictionInterface i programtilleggeksempel.cpp

Etter noen andre endringer i prosjektet for å endre kompileringen til bibliotek-modus vil resultatet til slutt være en dll-fil som automatisk blir lastet av *MultiPhasePlot* hvis lagringsadressen er riktig i prosjektetfilen. Programtillegget vil dukke opp med navnet som er definert i *pluginName()*-funksjonen.



Figur 17: Eksempelet lastet inn i *MultiPhasePlot*

Komplett kildekode til eksempelet følger med kildekoden til programmet, mens en mer omfattende gjennomgang av prosessen som sagt er tilgjengelig i **tillegg A**.

8.2 Kildekodedokumentasjon og vedlikehold

Etter å ha brukt tid på å forstå kildekoden til *MapIT* i starten av prosjektet var behovet for en godt forståelig og dokumentert kildekode åpenbart. Når det gjelder forståeligheten til koden kommer noe gratis som følge av objektorienteringen men det er likevel viktig

å ha det i bakhodet under hele utviklingsprosessen. I tillegg er det nødvendig å ha en kildekodevurdering med jevne mellomrom for å se om designvalgene har vært riktige og for å se om alt som en gang gav mening, fortsatt gjør det. Dette ble gjort mange ganger under utviklingsfasen av prosjektet, spesielt på programtilleggrensesnittene som ikke kan endres i ettertid uten at programtilleggene også må endres og rekompileres.

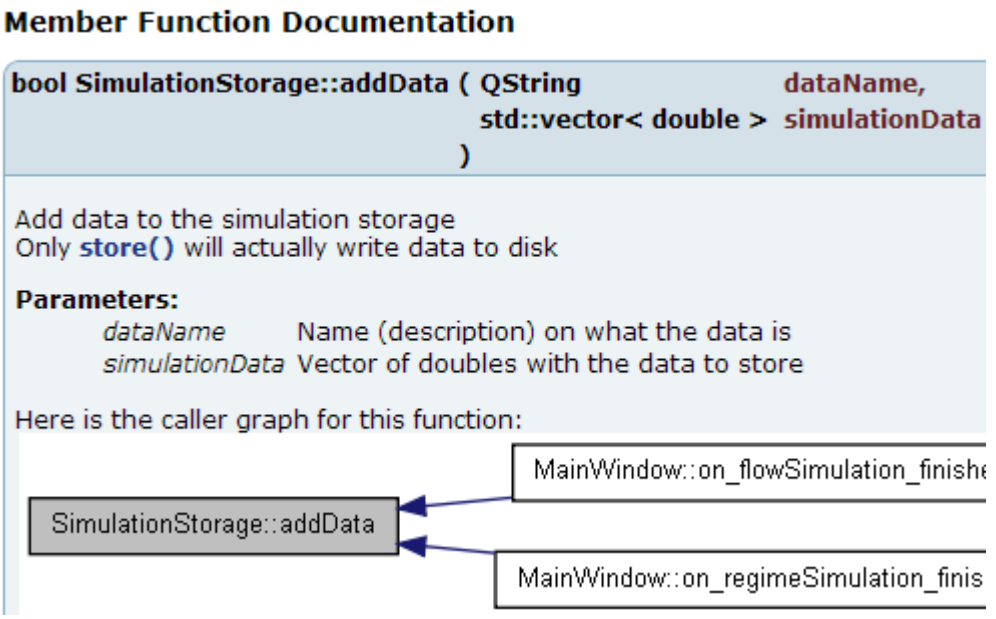
Funksjoner, klasser og argumenter ble grundig dokumentert inne i koden ved hjelp av kildekodekommentering slik som på eksempelet under. Dette gir alle som endrer kildekoden en rask og lett tilgjengelig dokumentasjon som forteller i grove trekk hva hver funksjon gjør og ofte også hvorfor og hvordan.

```
/** Add data to the simulation storage
*   Only store() will actually write data to disk
*   @param dataName Name (description) on what the data is
*   @param simulationData Vector of doubles with the data to store
*/
bool SimulationStorage::addData(QString dataName,
                                std::vector<double> simulationData)
```

Figur 18: Eksempel på kildekodekommentering

Ved å skrive denne kildekodekommenteringen i en bestemt syntax åpner det også for prosessering av koden med Doxygen. Doxygen er et fritt tilgjengelig verktøy som analyserer kildekoden og så genererer vevsider der kildekoden og den tilhørende kildekodekommenteringen presenteres på en fin måte og som binder det hele sammen ved å lage lenker mellom de forskjellige funksjonene slik at man lett kan slå de opp. Den kan også lage grafer som viser hvordan klassene forholder seg til hverandre og hvor funksjonskall kommer fra. Det hele blir en grundig kildekodedokumentasjon. En veldig nedkuttet versjon omgjort til å passe i papirform er vedlagt som **tillegg C**, mens på **figur 19** ser vi et skjermbilde fra vevsideversjonen som er tilgjengelig i programmet ved å trykke **Help -> Source code documentation** og på denne nettadressen.[3]

[3] http://ibar.sf.net



Figur 19: Skjermbilde, kildekodedokumentasjon generert av *Doxygen*

Et annet aspekt av kildekodevedlikeholdet er eventuelle endringer i framtidige versjoner av bibliotekene programmet er basert på, det vil si Qt og Qwt. Nokia har strenge regler som sier at hele 4.x-serien vil være bakover- og framoverkompatibel både hva gjelder kildekode og kompilerte programmer. Dette sikrer at programmet kan benytte oppdaterte versjoner av Qt i mange år framover uten endringer og at programtillegg kompilert med andre versjoner av Qt fungerer knirkefritt. Qwt har ikke en slik garanti men det er ikke noe som tilsier at en oppdatering til nye versjoner av Qwt skulle kreve noen stor endring av kildekoden, Qwt har heller ingen direkte kontakt med programtilleggene.

Det er selvfølgelig ikke nødvendig å holde programmet oppdatert mot siste Qt-utgave hele tiden så lenge det fungerer fint som det er.

9 Videre arbeid

9.1 Fleksibel regimeovergangberegning

Til simulering av regimeovergangene, i motsetning til strømningssimuleringene, støttes for øyeblikket kun beregninger med U_{sg} som variabel plottet mot U_{sl}. Å tilby mer fleksibilitet til å for eksempel benytte U_{sl} som variabel, eller i det minste tillate plotting av grafene med U_{sg} som variabel men med aksene byttet om vil gjøre at programmet blir mer brukervennlig ettersom det varierer i litteraturen om det er U_{sl} eller U_{sg} som benyttes på X-aksen. Den sistnevnte funksjonen er enkel å implementere og ble faktisk implementert men så fjernet igjen siden det ikke var nok tid til å også eksponere muligheten på en tilfredsstillende måte i brukergrensesnittet.

Det går selvfølgelig allerede an å eksportere verdiene for å lage et slikt plott i et annet program.

9.2 Dimensjonsløse grupper

En annen funksjon som det ikke ble tid til å implementere er plotting mot valgfrie dimensjonsløse grupper. Det var heller ikke et av målene fra starten av men noe som dukket opp underveis som en attraktiv funksjon. Det er ingenting i veien for at dette kan implementeres senere. Måten dataene er lagret på i programmet legger fint til rette for at dette skal kunne gjøres. Å plotte mot dimensjonsløse grupper er en meget hjelpsom funksjon å ha i analyse av flerfasestrømning.

9.3 Automatisk regimevalg

En alternativ strømningssimuleringsmetode hvor regimet velges automatisk basert på regimeinformasjon hentet fra valgte regimeovergangprogramtillegg ville vært en fin ekstrafunksjon til programmet. Simuleringene vil automatisk skifte regime for beregningene når resultatene skulle tilsi det. Resultatene vil ikke nødvendigvis være en kontinuerlig kurve, men det er ikke et problem.

Det aller meste som er nødvendig for en slik funksjon er allerede implementert i programmet, det vil bare være snakk om litt ekstra programlogikk i den biten av programmet som utfører simuleringskallene for å legge til funksjonen, men det er ikke sikkert funksjonen vil kunne fungere like bra i alle omstendigheter. Problemer kan oppstå fordi hvert regimeovergangprogramtillegg kun gir overgangskriterier for én overgang mellom to regimer. Hadde hvert regimeovergangprogramtillegg gitt regimeoverganger mellom alle regimene hadde funksjonen blitt mer pålitelig enn om den er avhengig av en nærmest tilfeldig sammensetning av forskjellige regimeoverganger for å få dekket alle regimene. Disse programtilleggene er ikke nødvendigvis enige om regimeovergangene.

En endring i regimeovergangprogramtilleggene til at hver av de skal dekke alle regimene vil ikke være en god idé siden det ikke eksisterer så mange gode komplette regimeovergangtilnærminger. En slik løsning kan føre til at programtilleggene blir tvunget til å bare finne på noe tull for de regimeovergangene de egentlig ikke er beregnet på. Å legge til en alternativ type regimeovergangprogramtillegg som gjør akkurat dette i stedet for å erstatte den nåværende kunne derimot vært gjennomført uten negative konsekvenser for resten av programmet.

9.4 Gasstrykk og temperatur

I brukergrensesnittet er muligheten til å bruke gasstrykk og temperatur som variabel allerede eksponert, men deaktiviert fordi det ikke er implementert. En slik funksjon vil være avhengig av et stoffbibliotek og vil medføre en mye større og annerledes kompleksitet i programmet. Utfordringen i å legge til en slik funksjon vil altså ikke være i det å koble den mot programmet, men i å lage den i det hele tatt.

9.5 Annet

Ved omfattende bruk av programmet vil det sannsynligvis dukke opp flere større eller små funksjoner som kan øke programmets bruksområde. Programmets arkitektur er slik at mange av disse bør kunne implementeres uten for store omveltninger i kildekoden.

10 Konklusjon

Det resulterende programmet fungerer svært bra og utfører arbeidsoppgavene som var forespeilet på en effektiv og god måte i tråd med målene som ble satt ved oppgavestart. Mange beregningsmodeller har blitt implementert og dette gir programmet et allerede stort bruksområde.

Erfaringen fra opplæring av en doktorgradstudent i utvikling av programtillegg er at det er lett forståelig for noen som kan C++. Programmet er allerede tiltenkt et bruksområde som en rask måte å identifisere initialbetingelser på for bruk i mer avanserte flerfase-strømningssimuleringer.

Programmet har blitt lastet opp på en prosjektside på Internett hvor det kan lastes ned gratis av alle.[4] Der er også kildekoden tilgjengelig under en lisens (GNU GPL) som kort forklart gir alle rett til å bruke kildekoden til hva de vil forutsatt at kildekoden blir frigitt dersom det kompilerte programmet i endret form distribueres.

Den totale kildekodemengden for alle delene av programmet ble til slutt på litt over 9000 linjer. For programtilleggene, som det er 24 av, stammer en vesentlig del av linjene fra strukturen som hvert programtillegg må følge, definert av programtilleggrensesnittene. Hovedprogrammet og støttebiblioteket derimot består nærmest utelukkende av unik kode og krevde det meste av utviklingstiden.

- Hovedprogrammet - 4200 linjer
- Støttebiblioteket - 1300 linjer
- Programtilleggene - 3700 linjer

Det at stukturdelen av programtilleggene av samme type er ganske lik kan gi inntrykk av å være i strid med målet om å unngå kildekodedupliseringen som var så utbredt i MapIT. Det er imidlertid to elementer som gjør at dette ikke er tilfellet. For det første består den dupliserte koden som sagt av struktur nødvendig for å etterleve kravene stilt av grense-snittene, ikke av programlogikk slik tilfellet var i MapIT. For det andre er kildekodeduplisering i hovedprogrammet noe annet enn kildekodeduplisering i uavhengige programtillegg.

[4] http://multiphaseplot.googlecode.com/

Referanser

[1] Schlumberger oilfield glossary, entry id 11049. http://www.glossary.oilfield.slb.com/Display.cfm?Term=mist%20flow.

[2] Utfordrende flerfasestrømninger. http://forskningsradet.ravn.no/bibliotek/forskning/200604/200604019.html, 2006.

[3] B. J. Azzopardi. Drops in annular two-phase flow. International Journal of Multiphase Flow, 23(7):1 – 53, 1997.

[4] D. Barnea and Y. Taitel. A model for slug length distribution in gas-liquid slug flow. International Journal of Multiphase Flow, 19(5):829 – 838, 1993.

[5] H. Blasius. Mitt. forschungsard. 131, 1913.

[6] Nokia Corporation. Supported platforms. http://doc.qt.nokia.com/4.6/supported-platforms.html, June 2010.

[7] Luca Galbiati and Pierangelo Andreini. The transition between stratified and annular regimes for horizontal two-phase flow in small diameter tubes. International Communications in Heat and Mass Transfer, 19(2):185 – 190, 1992.

[8] S. Haaland. Simple and explicit formulas for the friction factor in turbulent flow. Journal of Fluids Engineering ASME, 103(5):89 – 90, 1983.

[9] E. T. Hurlburt and T. J. Hanratty. Prediction of the transition from stratified to slug and plug flow for long pipes. International Journal of Multiphase Flow, 28(5):707 – 729, 2002.

[10] Kristian Kjæstad. Flow patterns in two-phase flow. Master Thesis at Norwegian University of Science and Technology, NTNU, 2003.

[11] Keyla S. et al. MARRUAZ. Horizontal slug flow in a large-size pipeline: Experimentation and modeling. Journal of the Brazilian Society of Mechanical Sciences, 23:481 – 490, 2001.

[12] Ole Jørgen Nydal. Lecture Notes TEP12 Multiphase Flow. Norwegian University of Science and Technology, 2009.

[13] Weisman and Kang. Flow pattern transitions in vertical and upwardly inclined lines. Int. J. Multiphase Flow, 7:271–291, 1981.

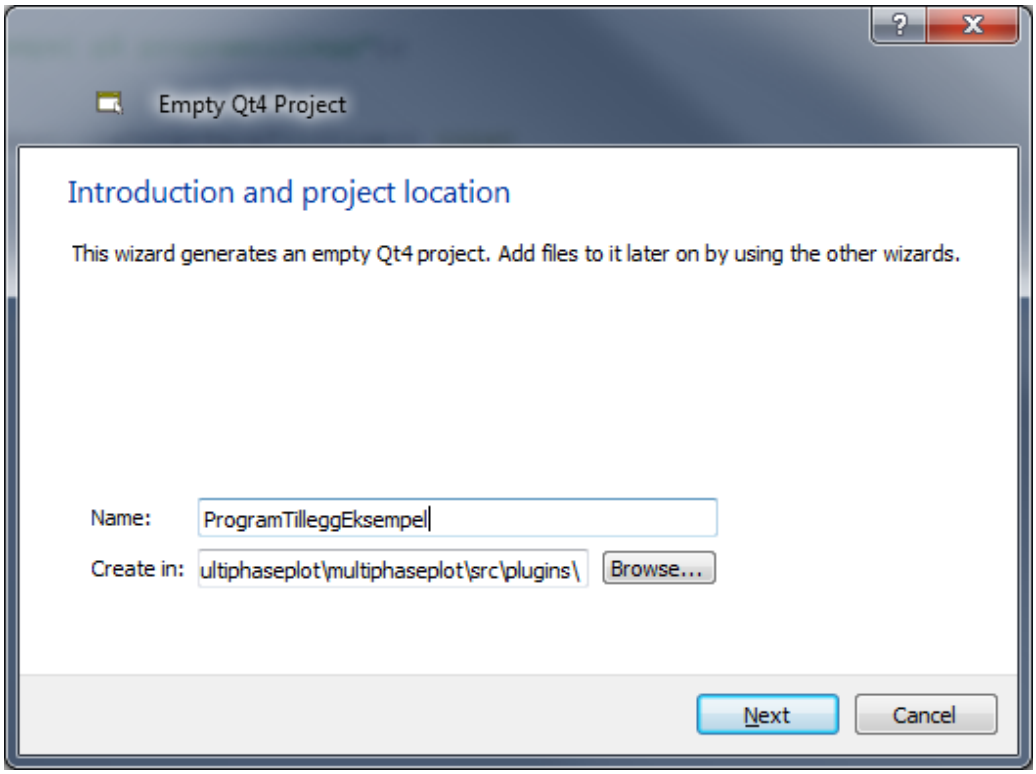
[14] R. H. S Winterton and J. S. Munaweera. Bubble size in two-phase gas-liquid bubbly flow in ducts. Chemical Engineering and Processing, 40(5):437 – 447, 2001.

Tillegg

Tillegg A Brukerveiledning - Produksjon av programtillegg

For å lage et programtillegg fra grunnen av må QtSDK[5] med QtCreator installeres og kildekoden til *MultiPhasePlot* må lastes ned[6] og pakkes ut.

Deretter er det bare å starte QtCreator og lage et tomt Qt prosjekt.



Figur 20: Skjermbilde, nytt prosjekt

Hvis *multiphaseplot/src/plugins/*-mappen velges som grunnmappe for prosjektet så vil de relative adressene i dette eksempelet være korrekte, dette vil være den enkleste løsningen å velge.

Deretter må det legges til en ny klasse via **File -> Add New... -> New Class**. Det enkleste er bare å kalle klassen det samme som prosjektet. Under *Base Class* må navnet på programtilleggrensesnittet som skal arves skrives. Alternativene er:

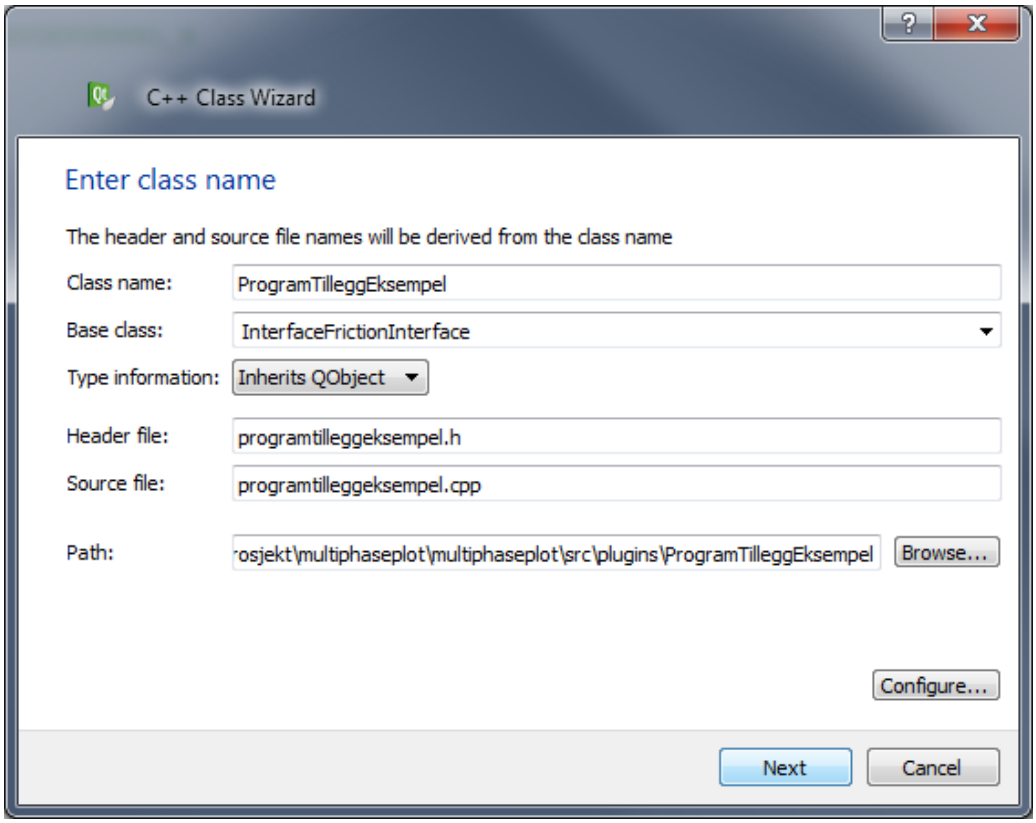
[5] http://qt.nokia.com/downloads/sdk-windows-cpp/
[6] http://multiphaseplot.googlecode.com/

- FrictionInterface
- InterfaceFrictionInterface
- SlipRelationInterface
- CalculateRegimeInterface
- RegimeTransitionInterface

I dette tilfellet velges *InterfaceFrictionInterface*. (Navnet kan virke noe forvirrende, *Interface* på starten av ordet refererer til overflaten mellom fasene, mens *Interface* på slutten av ordet refererer til at dette er et av grensesnittene som programtilleggene bruker, alle programtilleggklassene har navn som slutter med *Interface*.

De tre første programtilleggrensesnittene er friksjonsprogramtillegg, *CalculateRegimeInterface* er beregningsmetoder for et spesifikt regime, for eksempel unit cell-modellen for slug-strøm. *RegimeTransitionInterface* brukes av programtillegg som finner regimeoverganger.

Det er viktig at *Inherits QObject* blir valgt under *Type Information*. Resten fylles ut av seg selv.



Figur 21: Skjermbilde, ny klasse

I header-filen til den nye klassen må de riktige header-filene fra Qt og *MultiPhaseSupport* inkluderes.

```
#include <QObject>
#include <QtPlugin>
#include "../../application/Support/interfacefrictioninterface.h"
#include "../../application/Support/data.h"
```

Figur 22: Inkludering av riktige filer

Ved å klikke F2 når *interfacefrictioninterface.h* er fokusert av tekstmarkøren åpnes denne filen. Alle funksjonene som her er definert med «= 0» på slutten, såkalte «pure virtual»-funksjoner, må reimplementeres i vårt programtillegg.

```
class InterfaceFrictionInterface : public PluginInterface
{
public:
        virtual ~InterfaceFrictionInterface() {};

        virtual double calculateFrictionFactor(Data const &data, double alpha,
                        double reynoldsGas,double hdiameterGas,double thicknessFilm,
                        double frictionFactorGas) const = 0;
        virtual bool isInterfaceFriction() const = 0;
};
```

Figur 23: InterfaceFrictionInterface header-filen

Vi ser at denne klassen arver *PluginInterface* så vi må åpne denne filen for å se om også denne har funksjoner som må implementeres. Dette kan også gjøres ved å trykke F2 når *PluginInterface*-teksten er markert.

```
class PluginInterface
{
public:
        virtual ~PluginInterface() {};
        virtual QString pluginName() const = 0;
};
```

Figur 24: PluginInterface header-filen

I tillegg til å legge til disse funksjonene må det i header-filen til prosjektet spesifiseres at vi bruker *InterfaceFrictionInterface*-interfacet ved hjelp av *Q_INTERFACE()*-makroen. Klassedefinisjonen vår vil da bli seende slik ut.

```
class ProgramTilleggEksempel
            : public QObject, public InterfaceFrictionInterface
{
    Q_OBJECT
    Q_INTERFACES(InterfaceFrictionInterface)
public:
    ProgramTilleggEksempel();
    double calculateFrictionFactor(Data const &data, double alpha,
                double reynoldsGas, double hdiameterGas,
                double thicknessFilm, double frictionFactorGas) const;
    QString pluginName() const;
    bool isInterfaceFriction() const;
};
```

Figur 25: ProgramTilleggEksempel header-filen

Bortsett fra implementeringen av konstruktoren vil en enkel implementering av disse funksjonene se som følgende ut. Også her ser vi at vi må legge til en makro for at Qt skal forstå at dette skal være et programtillegg *(Q_EXPORT_PLUGIN2())*. Det kan være verdt å legge merke til at det som blir returnert fra *pluginName()*-funksjonen vil være det som dukker opp som programtilleggets navn i *MultiPhasePlot* senere.

```
double ProgramTilleggEksempel::calculateFrictionFactor(Data const &data,
               double alpha, double reynoldsGas, double hdiameterGas,
               double thicknessFilm, double frictionFactorGas) const
{
    return frictionFactorGas*20;
}

QString ProgramTilleggEksempel::pluginName() const {
    return QString("Eksempel på programtillegg");
}

bool ProgramTilleggEksempel::isInterfaceFriction() const {
    return true;
}
Q_EXPORT_PLUGIN2(../../../application/MultiPhase/debug/lib/
                 interfacefriction/programtilleggeksempel,
                 ProgramTilleggEksempel);
```

Figur 26: Funksjonene implementert

Det aller siste som må gjøres før programtillegget kan kompileres er å endre prosjektfilen. Der må vi fortelle kompilatoren at det er en dll-fil vi vil ha, ikke en exe-fil som er det som er standard, i tillegg må vi spesifisere at programtillegget skal linkes mot *MultiPhaseSupport*. Dette gjøres ved å legge noen linjer til prosjektfilen *ProsjektTilleggEksempel.pro*. Først inkluderer vi konfigurasjonsfilen *config.pri*, for å vite om vi er i debug eller release-modus, deretter inkluderes kildekodefilene før vi til slutt spesifiserer at vi skal ha et bibliotek (library), sier hvor filen skal lagres og at vi skal linke mot *MultiPhaseSupport*.

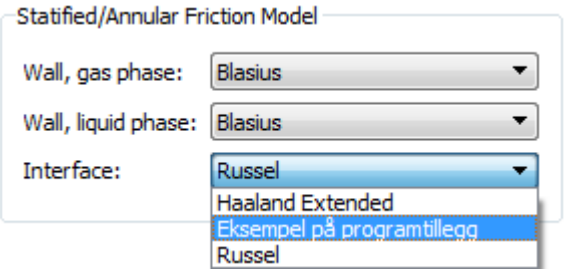
```
MPP_ROOT = ../..
include( $${MPP_ROOT}/config.pri )
HEADERS += programtilleggeksempel.h
SOURCES += programtilleggeksempel.cpp
TEMPLATE = lib
TARGET  = ../../../application/MultiPhase/debug/
           lib/interfacefriction/programtilleggeksempel
LIBS += -L../../application/MultiPhase/$${NAME_STR}/
LIBS += -lMultiPhaseSupport
```

Figur 27: Kompileringen må endres til bibliotekmodus

Den nest siste linjen forteller prosjektet hvor den skal søke etter *MultiPhaseSupport*-filen. Det er viktig at hvis kompileringen foregår i debug-modus må man linke mot debug-versjonen av *MultiPhaseSupport* og vice-versa for release. Dette oppnås ved at variabelen *$${NAME_STR}*, hentet fra *config.pri*, inneholder enten «debug» eller «release» slik at prosjektet leter i den riktige mappen etter filen.

Deretter er det bare å trykke F5 for å kompilere programtillegget. Hvis filadressene er riktige bør programtillegget automatisk havne på riktig sted og ved å starte *MultiPhasePlot* kan vi verifisere dette.

MultiPhasePlot er ikke ferdigkompilert i kildekodedistribusjonen av programmet, så for å teste programtillegget må dette gjøres en gang først, se **tillegg B**. Et annet alternativ er å kompilere programtillegget i release-modus for så å kopiere det inn i riktig mappe i den kompilerte versjonen av programmet.



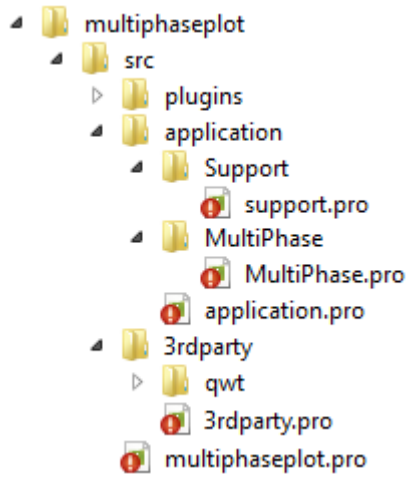
Figur 28: Skjermbilde, programtillegget dukker opp i *MultiPhasePlot*

Den komplette kildekoden til dette eksempelet er tilgjengelig i *multiphaseplot/src/plugins/ProgramTilleggEksempel*-mappen.

Tillegg B Brukerveiledning - Kompilering av programpakken

Til kompilering av programmet er QtSDK[7] (Qt Software Development Kit) for Windows nødvendig. Denne inneholder GCC (GNU Compiler Collection) og MinGW, et åpen kildekode-kompileringsverktøy for kompilering av programmer i Windows. Selve kompileringen skjer automatisk når *compile*-kommendoen blir gitt i utviklingsverktøyet som følger med QtSDK, QtCreator. QtCreator er et «alt-i-ett» prosjektbehandling- og kildekoderedigeringsverktøy.

Programpakken som ble laget i dette prosjektet består av mange deler, hovedprogrammet *MultiPhasePlot*, støttebiblioteket *MultiPhaseSupport*, Qwt og alle programtilleggene. Hver av disse har en egen prosjektfil som kan åpnes i QtCreator for å kompilere denne delen. Hvis det i stedet for å kompilere programpakken bit for bit, er ønskelig å kompilere hele programpakken i ett så kan den overordnede prosjektfilen åpnes. Denne ligger på toppen av kildekodetreet i *src*-mappen og heter *multiphaseplot.pro*. Ved å åpne denne og så trykke **Build -> Build All** vil QtCreator jobbe seg gjennom undermappene i riktig rekkefølge slik at resultatet til slutt blir en ferdig programpakke som havner i *src/application/MultiPhase/debug* eller *release* avhengig av om kompileringen er satt til debug eller release-modus. På samme måte kan alle programtilleggene kompileres i ett men uten resten av programmet ved å bruke *plugins.pro* i *src/plugins*-mappen.



Figur 29: Mappestruktur med prosjektfiler

[7] http://qt.nokia.com/downloads/sdk-windows-cpp/

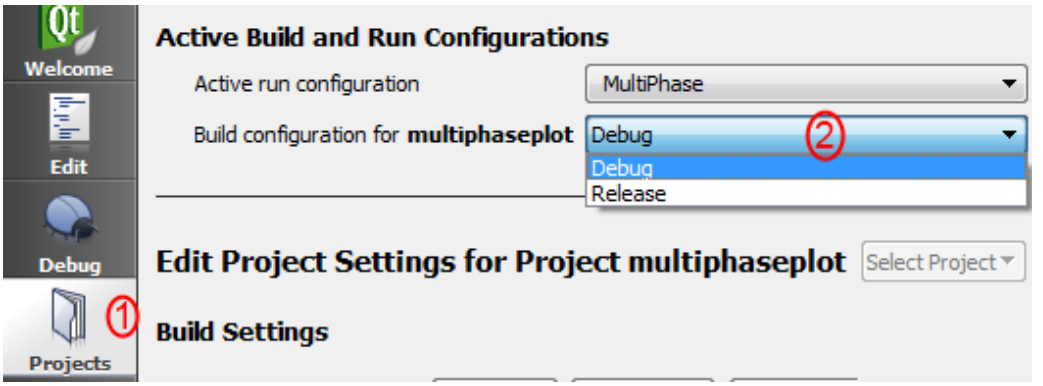
Dersom det er ønskelig å distribuere denne versjonen av programmet til noen som ikke har Qt installert bør Qt-bibliotekene som programmet er avhengig av legges ved. Disse er:

- QtCore4
- QtGui4
- QtNetwork4
- QtSvg4
- QtWebKit4
- QtXmlPatterns4
- mingwm10
- libgcc_s_dw2-1
- phonon4

B.1 Debug- og release-modus

Under utviklingen av programvaren er det praktisk å kompilere programdelene i debug-modus. Dette gir mange ekstra feilsøkingsmuligheter underveis, men de resulterende programfilene er noe tregere og omtrent 10 ganger så store. Derfor er det å anbefale å bytte til release-modus hvis programmet skal distribueres videre til andre. Å bytte mellom de to modusene er fort gjort.

- I *projects*-fanen i QtCreator må debug eller release velges
- I *config.pri*-filen i *src*-mappen må debug og release velges og avvelges.



Figur 30: Under *projects*-fanen i QtCreator kan debug eller release velges

```
CONFIG += debug # release/debug
CONFIG -= release # release/debug
```

Figur 31: I *config.pri* må aktiv modus legges til som «+ =» og den andre som «− =»

Dersom kompileringen feiler etter et bytte mellom debug og release kan en tvungen rekompilering av all koden være nødvendig. Dette gjøres fra **Build -> Rebuild All**

Tillegg C Kildekodedokumentasjon fra Doxygen

MultiPhasePlot
Source Code Documentation
0.6.20100530

Contents

1 Class Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

2 Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

3 Class Documentation

3.1 CalculateFlowThread Class Reference

Class for doing the actual calculations, inherits QThread.

Signals

- void simulationError (QString errorMsg)

Public Member Functions

- CalculateFlowThread (QWidget *parent, Data *data, FrictionFunctions *frictionFunctions, CalculateFunctions *calculateFunctions, bool useMinimumHoldupCriterion, unsigned int id, double tolerance=0.0001)
- CalculateFlowThread (QWidget *parent, Data *data, FrictionFunctions *frictionFunctions, CalculateFunctions *calculateFunctions, RegimeTransitionInterface *regimeTransitionPlugin, unsigned int id, double tolerance=0.0001)
- ~CalculateFlowThread ()
- double diameterGas ()
- double x ()
- double y ()
- double holdup ()
- double alpha ()
- double dpdx ()

Protected Member Functions

- void run ()

3.1.1 Detailed Description

Class for doing the actual calculations, inherits QThread.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 CalculateFlowThread::CalculateFlowThread (QWidget * *parent*, Data * *data*, FrictionFunctions * *frictionFunctions*, CalculateFunctions * *calculateFunctions*, bool *useMinimumHoldupCriterion*, unsigned int *id*, double *tolerance* = `0.0001`)

Constructor that creates the thread and readies for simulation

Parameters:

parent Pointer to parent (owner)

data Flow data

frictionFunctions Pointer to class with pointer to functions

calculateFunctions Pointer to class with pointers to calculate functions

useMinimumHoldupCriterion Should the calculation use the minimum holdup criterion or simply calculate for the selected regime

id Thread identification, used to keep track of it in CalculateManager

tolerance Numerical tolerance in calculation answer

3.1.2.2 CalculateFlowThread::CalculateFlowThread (QWidget * *parent*, Data * *data*, FrictionFunctions * *frictionFunctions*, CalculateFunctions * *calculateFunctions*, RegimeTransitionInterface * *regimeTransitionPlugin*, unsigned int *id*, double *tolerance* = `0.0001`)

3.1.2.3 CalculateFlowThread::~CalculateFlowThread ()

Destructor, delete data and so on

3.1.3 Member Function Documentation

3.1.3.1 double CalculateFlowThread::alpha ()

Returns alpha

3.1.3.2 double CalculateFlowThread::diameterGas ()

Returns hydraulic diameters for all variables

3.1.3.3 double CalculateFlowThread::dpdx ()

Returns pressure loss

3.1.3.4 double CalculateFlowThread::holdup ()

Returns holdup

3.1.3.5 void CalculateFlowThread::run () [protected]

Run thread, gets called from start()

3.1.3.6 void CalculateFlowThread::simulationError (QString *errorMsg*) [signal]

Here is the caller graph for this function:

3.1.3.7 double CalculateFlowThread::x ()

Returns x value

3.1.3.8 double CalculateFlowThread::y ()

Returns y value

3.2 CalculateFunctions Class Reference

Class which handles pointers to regime calculation plugins designed for being passing as argument to calculation functions.

Public Member Functions

- CalculateFunctions ()
- void setPluginAnnular (CalculateRegimeInterface *pluginAnnular)
- void setPluginDispersed (CalculateRegimeInterface *pluginDispersed)
- void setPluginStratified (CalculateRegimeInterface *pluginStratified)
- void setPluginSlug (CalculateRegimeInterface *pluginSlug)
- void setEnabledAnnular (bool enabled)
- void setEnabledDispersed (bool enabled)
- void setEnabledStratified (bool enabled)
- void setEnabledSlug (bool enabled)
- CalculateRegimeInterface * pluginAnnular ()
- CalculateRegimeInterface * pluginDispersed ()
- CalculateRegimeInterface * pluginStratified ()
- CalculateRegimeInterface * pluginSlug ()
- bool isEnabledAnnular ()
- bool isEnabledDispersed ()
- bool isEnabledStratified ()
- bool isEnabledSlug ()
- bool isDefined ()

3.2.1 Detailed Description

Class which handles pointers to regime calculation plugins designed for being passing as argument to calculation functions.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 CalculateFunctions::CalculateFunctions () [inline]

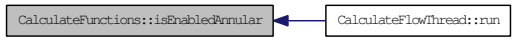
3.2.3 Member Function Documentation

3.2.3.1 bool CalculateFunctions::isDefined () [inline]

Here is the caller graph for this function:

3.2.3.2 bool CalculateFunctions::isEnabledAnnular () [inline]

Here is the caller graph for this function:



3.2.3.3 bool CalculateFunctions::isEnabledDispersed () [inline]

Here is the caller graph for this function:

3.2.3.4 bool CalculateFunctions::isEnabledSlug () [inline]

Here is the caller graph for this function:

3.2.3.5 bool CalculateFunctions::isEnabledStratified () [inline]

Here is the caller graph for this function:

3.2.3.6 CalculateRegimeInterface* CalculateFunctions::pluginAnnular () [inline]

Returns CalculateRegimeInterface pointer to plugin calculating annular flow

Here is the caller graph for this function:

3.2.3.7 CalculateRegimeInterface* CalculateFunctions::pluginDispersed () [inline]

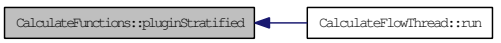
Here is the caller graph for this function:

3.2.3.8 CalculateRegimeInterface* CalculateFunctions::pluginSlug () [inline]

Here is the caller graph for this function:

3.2.3.9 CalculateRegimeInterface* CalculateFunctions::pluginStratified () [inline]

Here is the caller graph for this function:



3.2.3.10 void CalculateFunctions::setEnabledAnnular (bool *enabled*) [inline]

3.2.3.11 void CalculateFunctions::setEnabledDispersed (bool *enabled*) [inline]

3.2.3.12 void CalculateFunctions::setEnabledSlug (bool *enabled*) [inline]

3.2.3.13 void CalculateFunctions::setEnabledStratified (bool *enabled*) [inline]

3.2.3.14 void CalculateFunctions::setPluginAnnular (CalculateRegimeInterface * *pluginAnnular*) [inline]

Set the plugin used to calculate annular flow

3.2.3.15 void CalculateFunctions::setPluginDispersed (CalculateRegimeInterface * *pluginDispersed*) [inline]

3.2.3.16 void CalculateFunctions::setPluginSlug (CalculateRegimeInterface * *pluginSlug*) [inline]

3.2.3.17 void CalculateFunctions::setPluginStratified (CalculateRegimeInterface * *pluginStratified*) [inline]

3.3 CalculateManager Class Reference

Class which splits calculation tasks over several threads for processing and handles the gathering of results.

Signals

- void simulationProgressed (int percent)
- void simulationError (QString errorMsg)
- void finished ()
- void terminated ()

Public Member Functions

- CalculateManager (QWidget *parent, int simulationId, Data *data, FrictionFunctions *frictionFunctions, CalculateFunctions *calculateFunctions, bool useMinimumHoldupCriterion, unsigned int resolution, unsigned int threads=1, scale_t scaleType=SCALE_LIN, double tolerance=0.0000001, QString calculationName="")
- CalculateManager (QWidget *parent, int simulationId, Data *data, FrictionFunctions *frictionFunctions, RegimeTransitionInterface *regimeTransitionPlugin, CalculateFunctions *calculateFunctions, unsigned int resolution, unsigned int threads=1, scale_t scaleType=SCALE_LIN, double tolerance=0.0000001, QString calculationName="")
- ~CalculateManager ()
- void run ()
- void quit ()
- bool isRunning ()
- bool isFinished ()
- bool isTerminated ()
- int simulationID ()
- const Data * data ()
- std::vector< double > diameterGasList ()
- std::vector< double > xList ()
- std::vector< double > yList ()
- std::vector< double > holdupList ()
- std::vector< double > alphaList ()
- std::vector< double > dpdxList ()
- int progress ()
- QString calculationName ()
- QString pluginName ()
- QString xVariableName ()
- double xVariableMin ()
- double xVariableMax ()

3.3.1 Detailed Description

Class which splits calculation tasks over several threads for processing and handles the gathering of results.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 CalculateManager::CalculateManager (QWidget * *parent*, int *simulationId*, Data * *data*, FrictionFunctions * *frictionFunctions*, CalculateFunctions * *calculateFunctions*, bool *useMinimumHoldupCriterion*, unsigned int *resolution*, unsigned int *threads* = `1`, scale_t *scaleType* = `SCALE_LIN`, double *tolerance* = `0.0000001`, QString *calculationName* = `""`)

Constructor that creates the all calculation thread and readies for simulation

Parameters:

parent Pointer to parent (owner)

simulationId Identification number of simulation

data Flow data

frictionFunctions Pointer to class with pointer to friction functions

calculateFunctions Pointer to class with pointers to calculate functions

useMinimumHoldupCriterion Should the calculation use the minimum holdup criterion or simply calculate for the selected regime

resolution How many points to calculate

threads Number of threads to run simultaneously

scaleType What type of scale_t for x value intervals to use

tolerance Numeric tolerance

calculationName Name of calculation, used for plotting

3.3.2.2 CalculateManager::CalculateManager (QWidget * *parent*, int *simulationId*, Data * *data*, FrictionFunctions * *frictionFunctions*, RegimeTransitionInterface * *regimeTransitionPlugin*, CalculateFunctions * *calculateFunctions*, unsigned int *resolution*, unsigned int *threads* = `1`, scale_t *scaleType* = `SCALE_LIN`, double *tolerance* = `0.0000001`, QString *calculationName* = `""`)

3.3.2.3 CalculateManager::~CalculateManager ()

Destructor, delete CalculateFlowThread content

3.3.3 Member Function Documentation

3.3.3.1 std::vector< double > CalculateManager::alphaList ()

Returns vector of doubles containing alpha values for all variables

3.3.3.2 QString CalculateManager::calculationName ()

3.3.3.3 const Data* CalculateManager::data () `[inline]`

Returns const pointer to data used in simulation

3.3.3.4 std::vector< double > CalculateManager::diameterGasList ()

Returns vector of doubles containing hydraulic diameters for all variables

3.3.3.5 std::vector< double > CalculateManager::dpdxList ()

Returns vector of doubles containing pressure loss values for all variables

3.3.3.6 void CalculateManager::finished () `[signal]`

Here is the caller graph for this function:

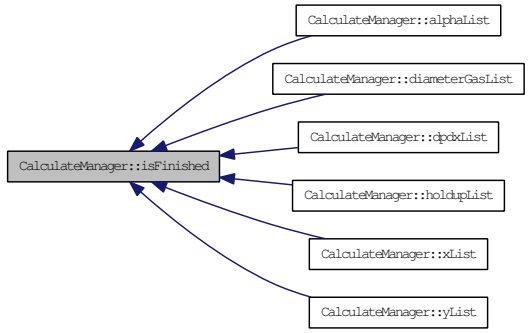
3.3.3.7 std::vector< double > CalculateManager::holdupList ()

Returns vector of doubles containing holdup for all variables

3.3.3.8 bool CalculateManager::isFinished () [inline]

Returns true if simulation is finished

Here is the caller graph for this function:



3.3.3.9 bool CalculateManager::isRunning () [inline]

Returns true if simulation is currently running

3.3.3.10 bool CalculateManager::isTerminated () [inline]

Returns true if simulation is/was terminated

3.3.3.11 QString CalculateManager::pluginName ()

3.3.3.12 int CalculateManager::progress ()

3.3.3.13 void CalculateManager::quit ()

Quit

3.3.3.14 void CalculateManager::run ()

Run calculation threads, gets called by start()

3.3.3.15 void CalculateManager::simulationError (QString *errorMsg*) [signal]

Here is the caller graph for this function:

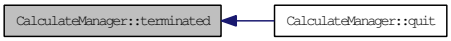
3.3.3.16 int CalculateManager::simulationID () [inline]

Returns simulation ID

3.3.3.17 void CalculateManager::simulationProgressed (int *percent*) [signal]

3.3.3.18 void CalculateManager::terminated () [signal]

Here is the caller graph for this function:



3.3.3.19 std::vector< double > CalculateManager::xList ()

Returns vector of doubles containing superficial velocities

3.3.3.20 double CalculateManager::xVariableMax ()

Returns the maximum x variable value

3.3.3.21 double CalculateManager::xVariableMin ()

Returns the minimum x variable value

3.3.3.22 QString CalculateManager::xVariableName ()

Returns QString with name of the x variable, first letter capitalized

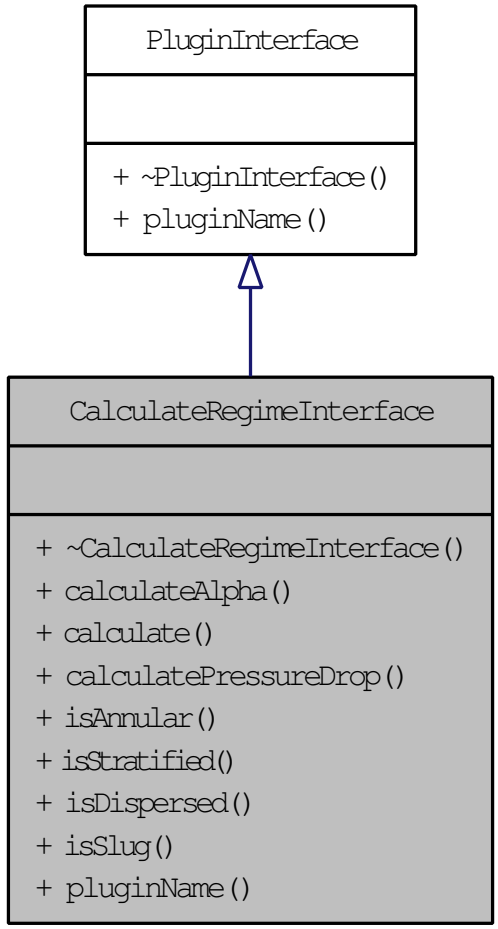
3.3.3.23 std::vector< double > CalculateManager::yList ()

Returns vector of doubles containing y values

3.4 CalculateRegimeInterface Class Reference

Plugin interface for the creation of regime calculation plugins. These plugins are to be used as the base calculation method for a specific regime. E.g. slug. Inheritance diagram for

CalculateRegimeInterface:



Public Member Functions

- virtual ~CalculateRegimeInterface ()
- virtual double calculateAlpha (Data *data, FrictionFunctions *frictionFunctions, double tolerance)=0
- virtual QList< QPair< QString, double > > calculate (Data *data, FrictionFunctions *frictionFunctions, double tolerance)=0
- virtual double calculatePressureDrop (Data *data, FrictionFunctions *frictionFunctions, double tolerance)=0
- virtual bool isAnnular () const =0
- virtual bool isStratified () const =0
- virtual bool isDispersed () const =0
- virtual bool isSlug () const =0
- virtual QString pluginName () const =0

3.4.1 Detailed Description

Plugin interface for the creation of regime calculation plugins. These plugins are to be used as the base calculation method for a specific regime. E.g. slug.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 virtual CalculateRegimeInterface::~CalculateRegimeInterface () [inline, virtual]

3.4.3 Member Function Documentation

3.4.3.1 virtual QList<QPair<QString, double> > CalculateRegimeInterface::calculate (Data * *data*, FrictionFunctions * *frictionFunctions*, double *tolerance*) [pure virtual]

3.4.3.2 virtual double CalculateRegimeInterface::calculateAlpha (Data * *data*, FrictionFunctions * *frictionFunctions*, double *tolerance*) [pure virtual]

Here is the caller graph for this function:

3.4.3.3 virtual double CalculateRegimeInterface::calculatePressureDrop (Data * *data*, FrictionFunctions * *frictionFunctions*, double *tolerance*) [pure virtual]

Here is the caller graph for this function:

3.4.3.4 virtual bool CalculateRegimeInterface::isAnnular () const [pure virtual]

3.4.3.5 virtual bool CalculateRegimeInterface::isDispersed () const [pure virtual]

3.4.3.6 virtual bool CalculateRegimeInterface::isSlug () const [pure virtual]

3.4.3.7 virtual bool CalculateRegimeInterface::isStratified () const [pure virtual]

3.4.3.8 virtual QString CalculateRegimeInterface::pluginName () const [pure virtual]

Implements PluginInterface.

3.5 CalculateRegimeShared Class Reference

Public Member Functions

- CalculateRegimeShared ()
- ~CalculateRegimeShared ()

Static Public Member Functions

- static double bisectvoid (Data *data, FrictionFunctions *frictionFunctions, double(*evalFunction)(Data *, FrictionFunctions *, double variable, double *args, int nArg), double *args, int nArg, double tolerance, double x)
- static double reynolds (double density, double hdiameter, double velocity, double viscosity)
- static double hydraulicDiameter (double alpha, double area, double surface)
- static double relativeVelocityRisingBubbles (double alpha, double surfaceTension, double densityLiquid, double densityGas, double angle)

3.5.1 Constructor & Destructor Documentation

3.5.1.1 CalculateRegimeShared::CalculateRegimeShared ()

3.5.1.2 CalculateRegimeShared::~CalculateRegimeShared ()

3.5.2 Member Function Documentation

3.5.2.1 double CalculateRegimeShared::bisectvoid (Data * *data*, FrictionFunctions * *frictionFunctions*, double(*)(Data *, FrictionFunctions *, double variable, double *args, int nArg) *evalFunction*, double * *args*, int *nArg*, double *tolerance*, double *x*) [static]

Workhorse function which takes in pointer to function in with which a correct variable should be found, used by all four regimes

Parameters:

data Flow data

frictionFunctions Pointer to class with pointer to functions

evalFunction Pointer to function with which to evaluate the variable, takes an array of doubles as argument allowing it to pass as many arguments as it wants to

See also:

voidAnnular Function to calculate void fraction for annular flow, can be passed as argument as evalFunction
alphaDispersed Function to calculate alpha for dispersed flow, can be passed as argument as evalFunction
alphaStatified Function to calculate alpha for stratified flow, can be passed as argument as evalFunction
voidBubble Function to calculate void fraction for bubble in slug flow, can be passed as argument as evalFunction

Parameters:

args pointer to double array of arguments to pass to evalFunction

nArg number of arguments in args array

tolerance numerical error tolerance

x initial value, should be center value

3.5.2.2 double CalculateRegimeShared::hydraulicDiameter (double *alpha*, double *area*, double *surface*) [static]

Calculate hydraulic diameter

Parameters:

alpha Alpha value to use

area Area

surface Surface length

3.5.2.3 double CalculateRegimeShared::relativeVelocityRisingBubbles (double *alpha*, double *surfaceTension*, double *densityLiquid*, double *densityGas*, double *angle*) [static]

Calculates the rising velocity of gas bubbles in dispersed/slug flow

Parameters:

alpha Alpha value

surfaceTension Surface Tension

densityLiquid Density of liquid phase

densityGas Density of gas phase

angle Pipe angle

3.5.2.4 double CalculateRegimeShared::reynolds (double *density*, double *hdiameter*, double *velocity*, double *viscosity*) [static]

Calculates reynolds number

Parameters:

density Density

hdiameter Hydraulic diameter

velocity Speed of transport

viscosity Viscosity (Pa∗s)

3.6 Data Class Reference

Class which handles the data storage.

Public Member Functions

- Data ()
- Data (const Data ∗copyData)
- bool equals (Data ∗compareWith) const
- double alpha () const
- double angle () const
- double angleRad () const
- double viscosityLiquid () const
- double viscosityGas () const
- double densityLiquid () const
- double densityGas () const
- double pressureGas () const
- double temperatureGas () const
- double surfaceTension () const
- double Usl () const
- double Usg () const
- double diameter () const
- double roughness () const
- double variable () const
- double variableMax () const
- double variableMin () const
- regimetype_t regimeType () const
- variable_t variableType () const
- double slipRelationParameter () const
- QString alphaName () const
- QString angleName () const
- QString viscosityLiquidName () const
- QString viscosityGasName () const
- QString densityLiquidName () const
- QString densityGasName () const
- QString pressureGasName () const
- QString temperatureGasName () const
- QString surfaceTensionName () const
- QString UslName () const
- QString UsgName () const

- QString diameterName () const
- QString roughnessName () const
- QString regimeName () const
- QString variableName () const
- QString regimeNameShort (int length) const
- QString regimeTransitionName () const
- double Ul () const
- double Ug () const
- double area () const
- double reynoldsLiquid ()
- double reynoldsGas ()
- double hdiameterLiquid ()
- double hdiameterGas ()
- double hdiameter ()
- void setAlpha (double alpha)
- void setVariable (double variable)
- void setAngle (double angle)
- void setViscosityLiquid (double viscosityLiquid)
- void setViscosityGas (double viscosityGas)
- void setDensityLiquid (double densityLiquid)
- void setDensityGas (double densityGas)
- void setPressureGas (double pressureGas)
- void setTemperatureGas (double temperatureGas)
- void setSurfaceTension (double surfaceTension)
- void setUsl (double Usl)
- void setUsg (double Usg)
- void setDiameter (double diameter)
- void setRoughness (double roughness)
- void setVariableMin (double variableMin)
- void setVariableMax (double variableMax)
- void setVariableType (variable_t variableType)
- void setRegimeType (regimetype_t regimeType)
- void setRegimeTransitionName (QString regimeTransitionName)
- void writeData (QTextStream &out, bool flow=true) const
- QString dataInformation () const

Static Public Member Functions

- static Data * readData (QString dataStream, bool flow=true)

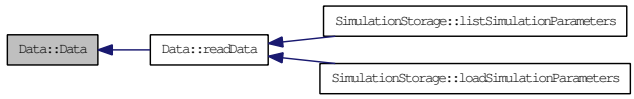
3.6.1 Detailed Description

Class which handles the data storage.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 Data::Data ()

Here is the caller graph for this function:

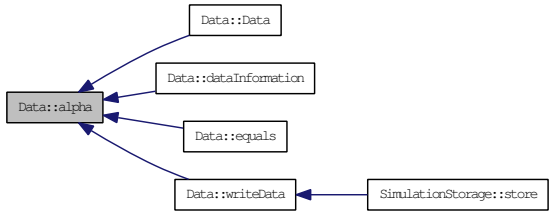


3.6.2.2 Data::Data (const Data * *copyData*)

3.6.3 Member Function Documentation

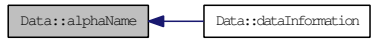
3.6.3.1 double Data::alpha () const `[inline]`

Here is the caller graph for this function:



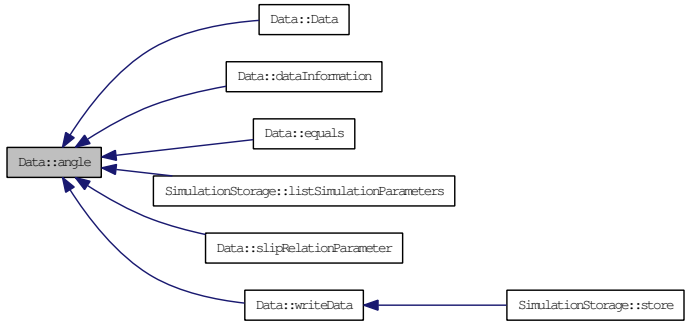
3.6.3.2 QString Data::alphaName () const [inline]

Here is the caller graph for this function:



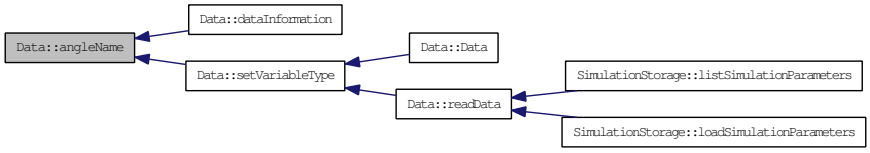
3.6.3.3 double Data::angle () const [inline]

Here is the caller graph for this function:



3.6.3.4 QString Data::angleName () const [inline]

Here is the caller graph for this function:



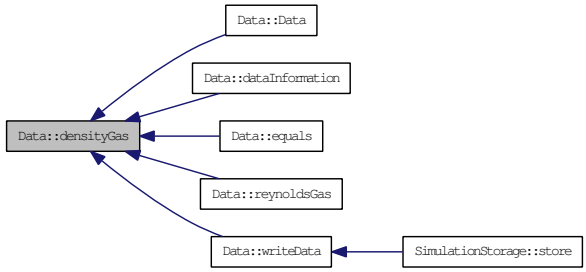
3.6.3.5 double Data::angleRad () const [inline]

3.6.3.6 double Data::area () const [inline]

3.6.3.7 QString Data::dataInformation () const

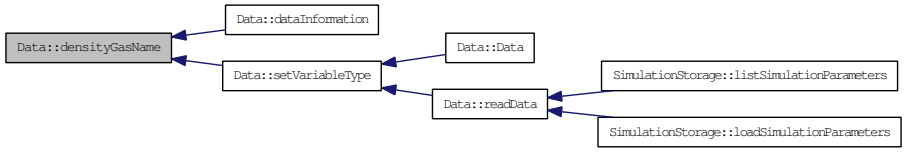
3.6.3.8 double Data::densityGas () const [inline]

Here is the caller graph for this function:



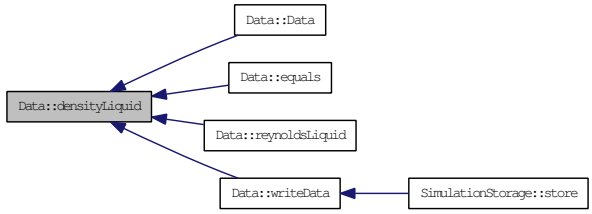
3.6.3.9 QString Data::densityGasName () const [inline]

Here is the caller graph for this function:



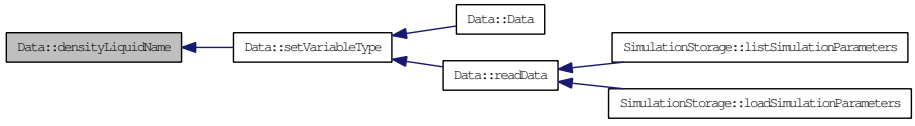
3.6.3.10 double Data::densityLiquid () const [inline]

Here is the caller graph for this function:



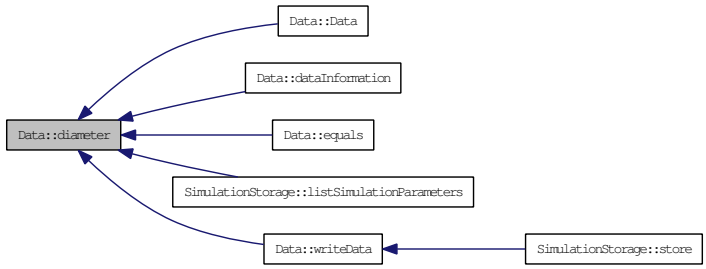
3.6.3.11 QString Data::densityLiquidName () const [inline]

Here is the caller graph for this function:



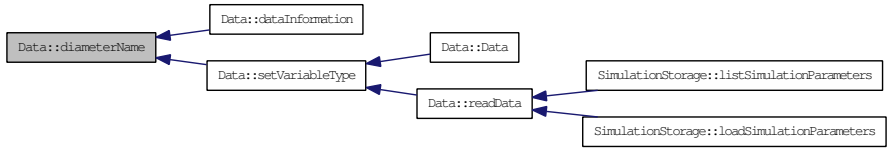
3.6.3.12 double Data::diameter () const [inline]

Here is the caller graph for this function:



3.6.3.13 QString Data::diameterName () const [inline]

Here is the caller graph for this function:



3.6.3.14 bool Data::equals (Data * *compareWith*) const

3.6.3.15 double Data::hdiameter () [inline]

3.6.3.16 double Data::hdiameterGas () [inline]

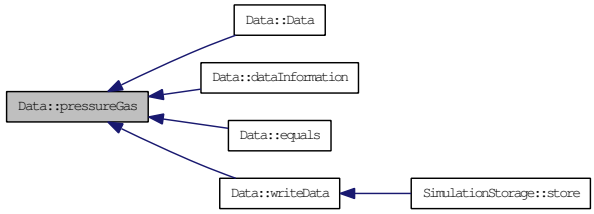
Here is the caller graph for this function:

3.6.3.17 double Data::hdiameterLiquid () [inline]

Here is the caller graph for this function:

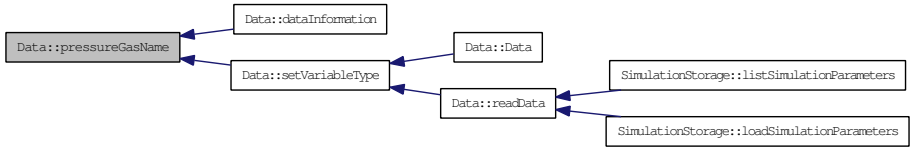
3.6.3.18 double Data::pressureGas () const [inline]

Here is the caller graph for this function:



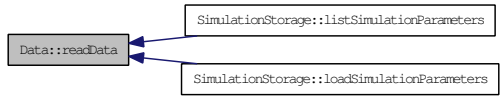
3.6.3.19 QString Data::pressureGasName () const [inline]

Here is the caller graph for this function:



3.6.3.20 Data * Data::readData (QString *dataStream*, bool *flow* = true) [static]

Here is the caller graph for this function:

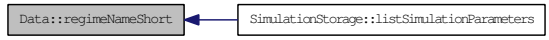


3.6.3.21 QString Data::regimeName () const [inline]

Here is the caller graph for this function:

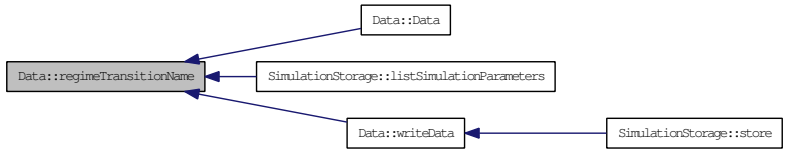
3.6.3.22 QString Data::regimeNameShort (int *length*) const

Here is the caller graph for this function:



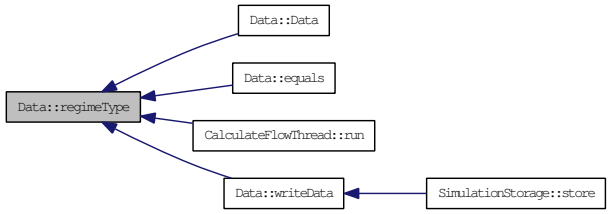
3.6.3.23 QString Data::regimeTransitionName () const [inline]

Here is the caller graph for this function:



3.6.3.24 regimetype_t Data::regimeType () const [inline]

Here is the caller graph for this function:

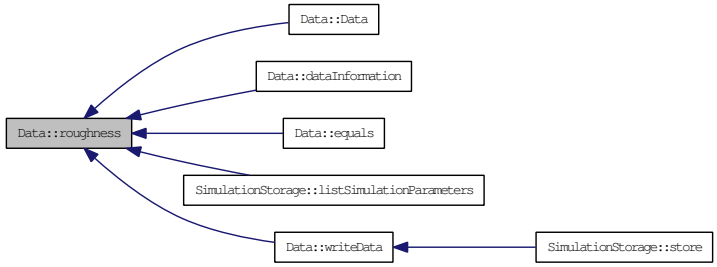


3.6.3.25 double Data::reynoldsGas () [inline]

3.6.3.26 double Data::reynoldsLiquid ()

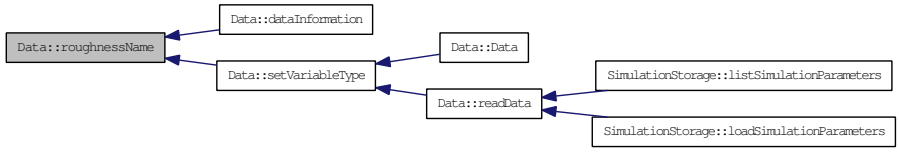
3.6.3.27 double Data::roughness () const [inline]

Here is the caller graph for this function:



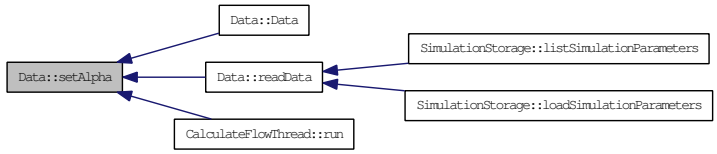
3.6.3.28 QString Data::roughnessName () const [inline]

Here is the caller graph for this function:



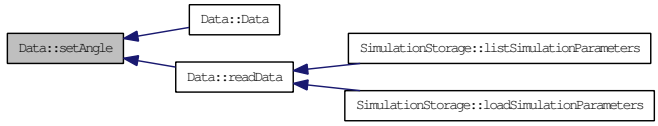
3.6.3.29 void Data::setAlpha (double *alpha*)

Here is the caller graph for this function:



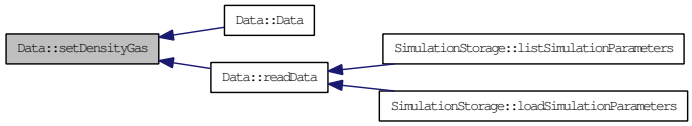
3.6.3.30 void Data::setAngle (double *angle*)

Here is the caller graph for this function:



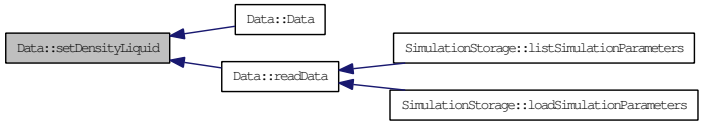
3.6.3.31 void Data::setDensityGas (double *densityGas*)

Here is the caller graph for this function:



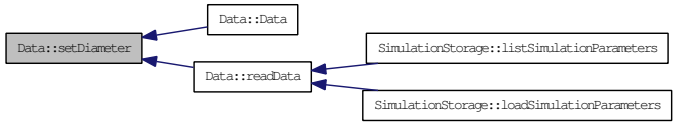
3.6.3.32 void Data::setDensityLiquid (double *densityLiquid*)

Here is the caller graph for this function:



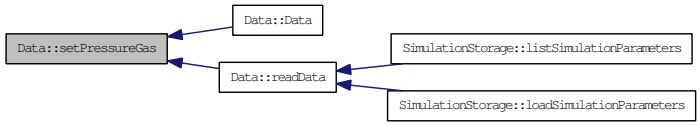
3.6.3.33 void Data::setDiameter (double *diameter*)

Here is the caller graph for this function:



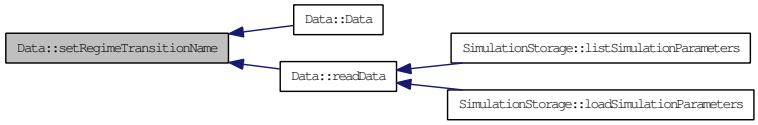
3.6.3.34 void Data::setPressureGas (double *pressureGas*)

Here is the caller graph for this function:



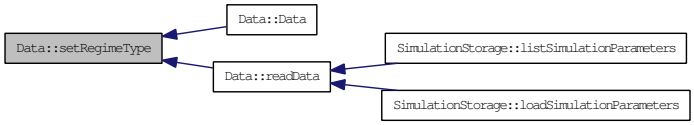
3.6.3.35 void Data::setRegimeTransitionName (QString *regimeTransitionName*)

Here is the caller graph for this function:



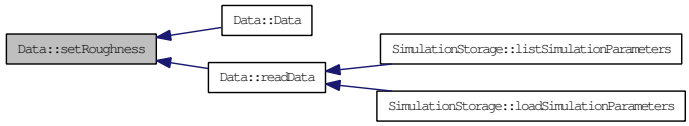
3.6.3.36 void Data::setRegimeType (regimetype_t *regimeType*)

Here is the caller graph for this function:



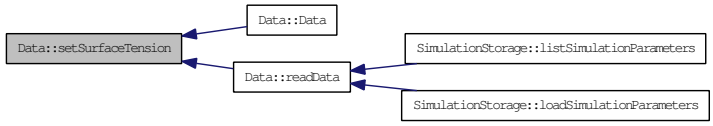
3.6.3.37 void Data::setRoughness (double *roughness*)

Here is the caller graph for this function:



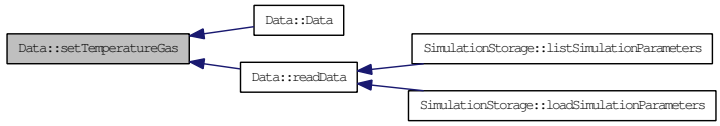
3.6.3.38 void Data::setSurfaceTension (double *surfaceTension*)

Here is the caller graph for this function:



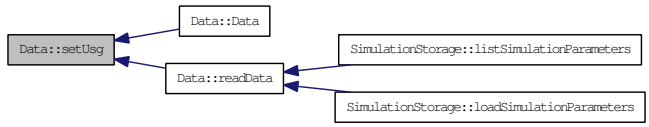
3.6.3.39 void Data::setTemperatureGas (double *temperatureGas*)

Here is the caller graph for this function:



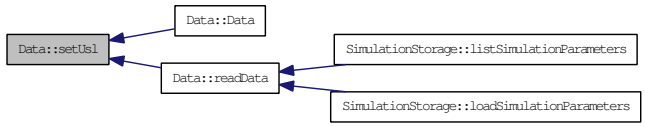
3.6.3.40 void Data::setUsg (double *Usg*)

Here is the caller graph for this function:



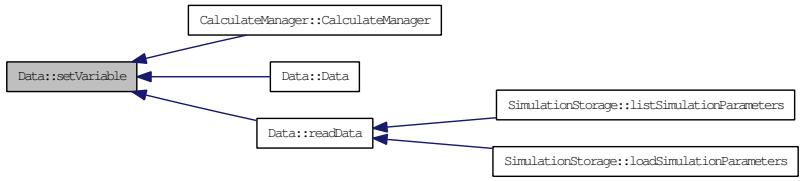
3.6.3.41 void Data::setUsl (double *Usl*) [inline]

Here is the caller graph for this function:



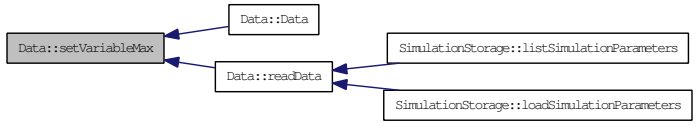
3.6.3.42 void Data::setVariable (double *variable*)

Here is the caller graph for this function:



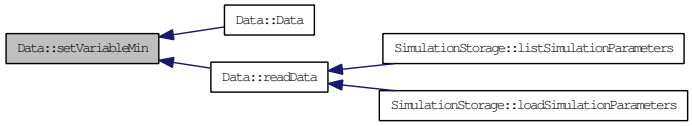
3.6.3.43 void Data::setVariableMax (double *variableMax*)

Here is the caller graph for this function:



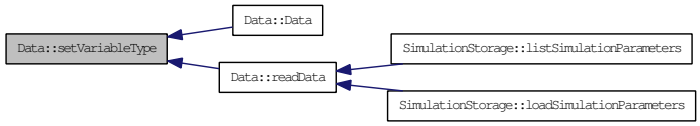
3.6.3.44 void Data::setVariableMin (double *variableMin*)

Here is the caller graph for this function:



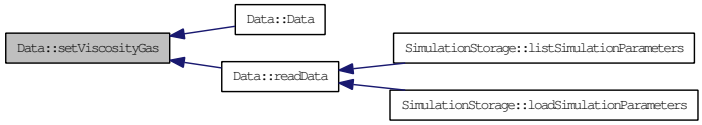
3.6.3.45 void Data::setVariableType (variable_t *variableType*)

Here is the caller graph for this function:



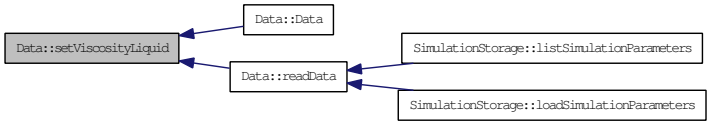
3.6.3.46 void Data::setViscosityGas (double *viscosityGas*)

Here is the caller graph for this function:



3.6.3.47 void Data::setViscosityLiquid (double *viscosityLiquid*)

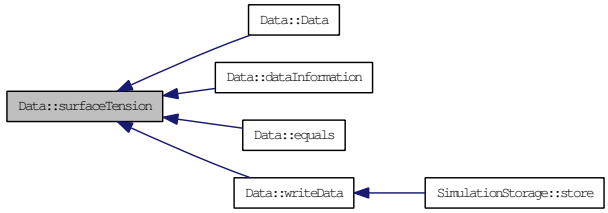
Here is the caller graph for this function:



3.6.3.48 double Data::slipRelationParameter () const [inline]

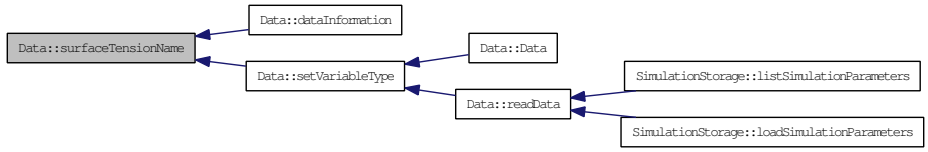
3.6.3.49 double Data::surfaceTension () const [inline]

Here is the caller graph for this function:



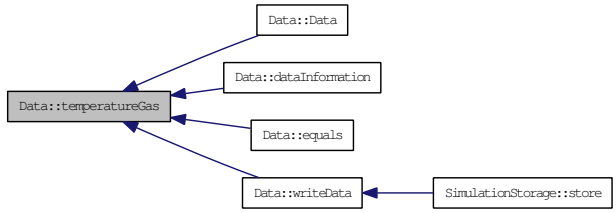
3.6.3.50 QString Data::surfaceTensionName () const [inline]

Here is the caller graph for this function:



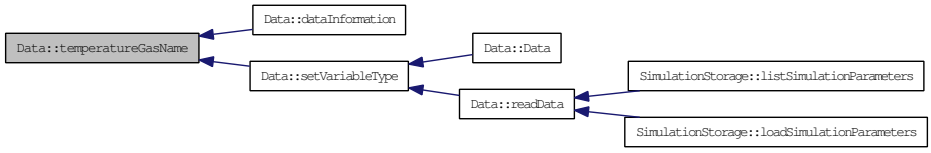
3.6.3.51 double Data::temperatureGas () const [inline]

Here is the caller graph for this function:



3.6.3.52 QString Data::temperatureGasName () const [inline]

Here is the caller graph for this function:



3.6.3.53 double Data::Ug () const [inline]

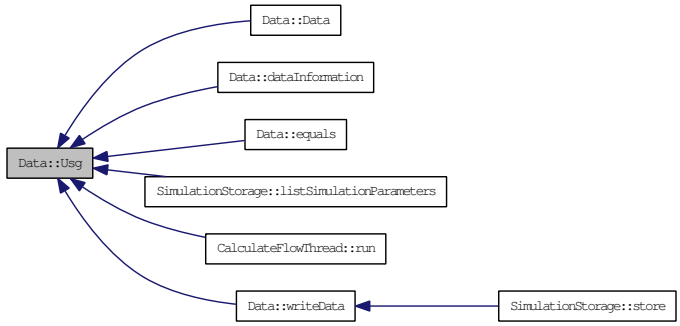
Here is the caller graph for this function:

3.6.3.54 double Data::Ul () const [inline]

Here is the caller graph for this function:

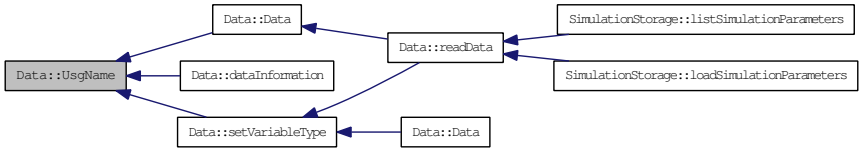
3.6.3.55 double Data::Usg () const [inline]

Here is the caller graph for this function:



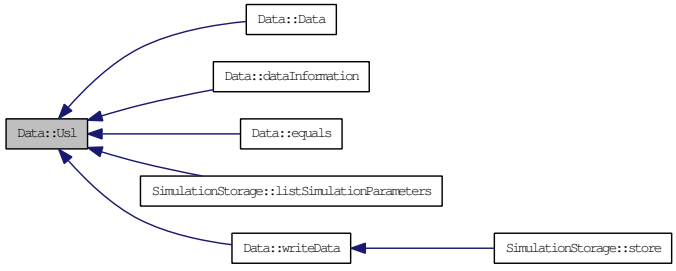
3.6.3.56 QString Data::UsgName () const [inline]

Here is the caller graph for this function:



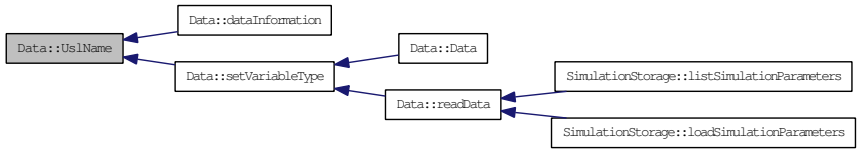
3.6.3.57 double Data::Usl () const [inline]

Here is the caller graph for this function:



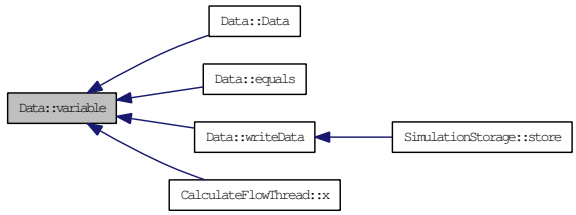
3.6.3.58 QString Data::UslName () const [inline]

Here is the caller graph for this function:



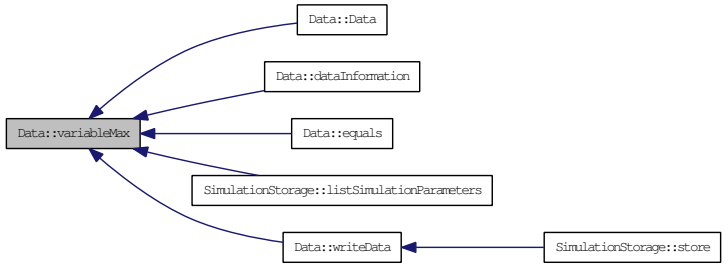
3.6.3.59 double Data::variable () const [inline]

Here is the caller graph for this function:



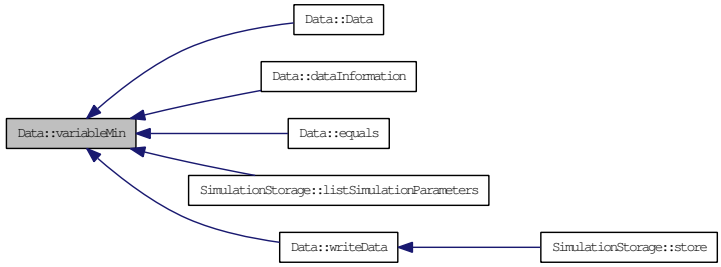
3.6.3.60 double Data::variableMax () const [inline]

Here is the caller graph for this function:



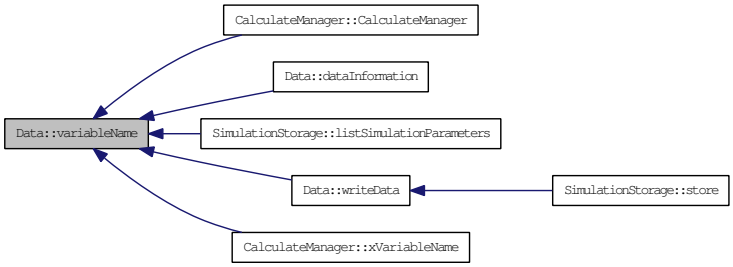
3.6.3.61 double Data::variableMin () const [inline]

Here is the caller graph for this function:



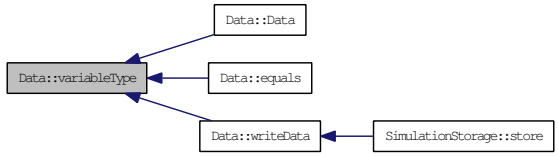
3.6.3.62 QString Data::variableName () const [inline]

Here is the caller graph for this function:



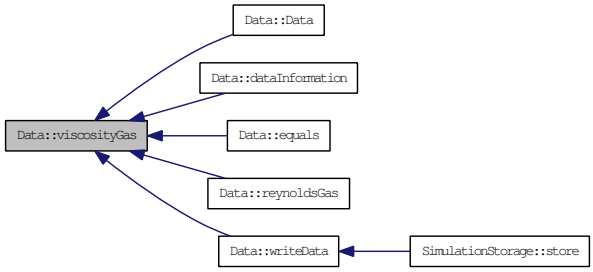
3.6.3.63 variable_t Data::variableType () const [inline]

Here is the caller graph for this function:



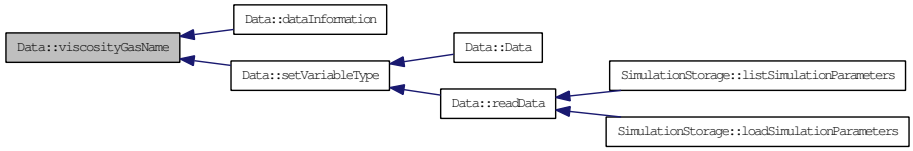
3.6.3.64 double Data::viscosityGas () const [inline]

Here is the caller graph for this function:



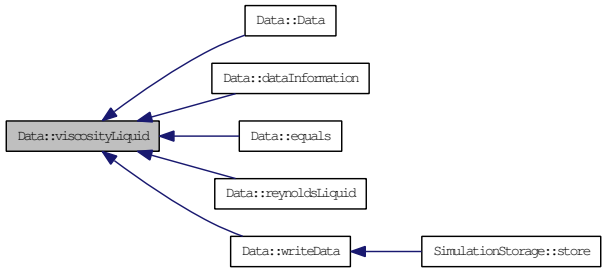
3.6.3.65 QString Data::viscosityGasName () const [inline]

Here is the caller graph for this function:



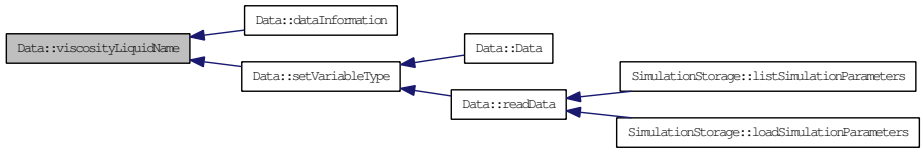
3.6.3.66 double Data::viscosityLiquid () const [inline]

Here is the caller graph for this function:



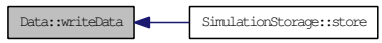
3.6.3.67 QString Data::viscosityLiquidName () const [inline]

Here is the caller graph for this function:



3.6.3.68 void Data::writeData (QTextStream & *out*, bool *flow* = `true`) const

Here is the caller graph for this function:



3.7 FrictionFunctions Class Reference

Class which handles pointers to friction and slip relation plugins designed for being passing as argument to calculation functions.

Public Member Functions

- FrictionFunctions ()
- bool equals (FrictionFunctions *frictionFunctions)
- void setPluginWallGas (FrictionInterface *pluginWallGas)
- void setPluginWallLiquid (FrictionInterface *pluginWallLiquid)
- void setPluginInterface (InterfaceFrictionInterface *pluginInterface)
- void setPluginDispersed (FrictionInterface *pluginDispersed)
- void setPluginSlipRelation (SlipRelationInterface *pluginSlipRelation)
- FrictionInterface * pluginWallGas ()
- FrictionInterface * pluginWallLiquid ()
- InterfaceFrictionInterface * pluginInterface ()
- FrictionInterface * pluginDispersed ()
- SlipRelationInterface * pluginSlipRelation ()

3.7.1 Detailed Description

Class which handles pointers to friction and slip relation plugins designed for being passing as argument to calculation functions.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 FrictionFunctions::FrictionFunctions ()

Constructor, sets pointers to NULL

3.7.3 Member Function Documentation

3.7.3.1 bool FrictionFunctions::equals (FrictionFunctions * *frictionFunctions*)

3.7.3.2 FrictionInterface* FrictionFunctions::pluginDispersed () `[inline]`

Returns FrictionInterface pointer to plugin calculating dispersed friction
Here is the caller graph for this function:

3.7.3.3 InterfaceFrictionInterface* FrictionFunctions::pluginInterface () `[inline]`

Returns InterfaceFrictionInterface pointer to plugin calculating interface friction
Here is the caller graph for this function:

3.7.3.4 SlipRelationInterface* FrictionFunctions::pluginSlipRelation () `[inline]`

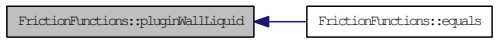
Returns SlipRelationInterface pointer to plugin calculating slip relation
Here is the caller graph for this function:

3.7.3.5 FrictionInterface* FrictionFunctions::pluginWallGas () `[inline]`

Returns FrictionInterface pointer to plugin calculating wall friction for gas phase
Here is the caller graph for this function:

3.7.3.6 FrictionInterface* FrictionFunctions::pluginWallLiquid () `[inline]`

Returns FrictionInterface pointer to plugin calculating wall friction for liquid phase
Here is the caller graph for this function:



3.7.3.7 void FrictionFunctions::setPluginDispersed (FrictionInterface * *pluginDispersed*) `[inline]`

Set the plugin used to calculate dispersed

3.7.3.8 void FrictionFunctions::setPluginInterface (InterfaceFrictionInterface * *pluginInterface*) `[inline]`

Set the plugin used to calculate interface friction

3.7.3.9 void FrictionFunctions::setPluginSlipRelation (SlipRelationInterface * *pluginSlipRelation*) `[inline]`

Set the plugin used to calculate slip relation

3.7.3.10 void FrictionFunctions::setPluginWallGas (FrictionInterface * *pluginWallGas*) [inline]

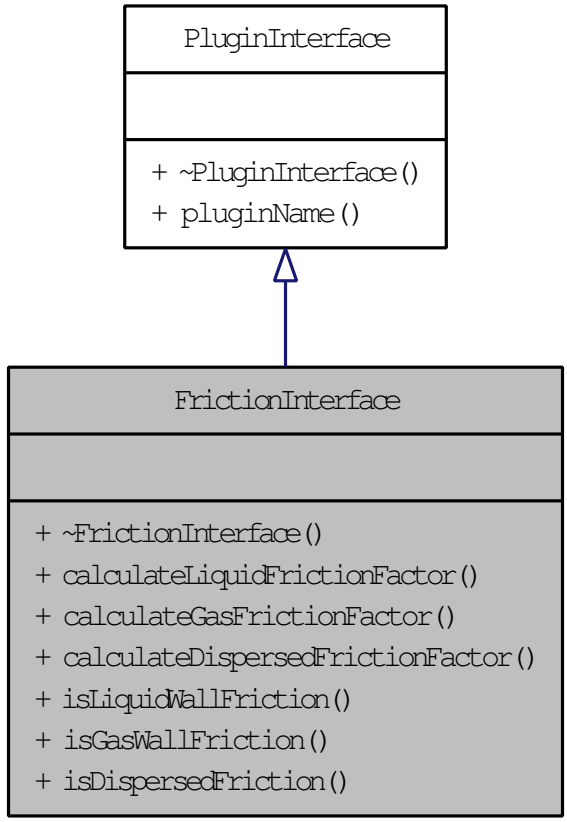
Set the plugin used to calculate wall friction for gas phase

3.7.3.11 void FrictionFunctions::setPluginWallLiquid (FrictionInterface * *pluginWallLiquid*) [inline]

Set the plugin used to calculate wall friction for liquid phase

3.8 FrictionInterface Class Reference

Plugin interface for the creation of friction plugins. Inheritance diagram for FrictionInterface:



Public Member Functions

- virtual ~FrictionInterface ()
- virtual double calculateLiquidFrictionFactor (Data const &data, double reynolds, double hdiameter) const =0
- virtual double calculateGasFrictionFactor (Data const &data, double reynolds, double hdiameter) const =0
- virtual double calculateDispersedFrictionFactor (Data const &data, double alpha, double reynolds, double hdiameter, double UDisp, double viscosity) const =0
- virtual bool isLiquidWallFriction () const =0
- virtual bool isGasWallFriction () const =0
- virtual bool isDispersedFriction () const =0

3.8.1 Detailed Description

Plugin interface for the creation of friction plugins.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 virtual FrictionInterface::~FrictionInterface () [inline, virtual]

3.8.3 Member Function Documentation

3.8.3.1 virtual double FrictionInterface::calculateDispersedFrictionFactor (Data const & *data*, double *alpha*, double *reynolds*, double *hdiameter*, double *UDisp*, double *viscosity*) const [pure virtual]

3.8.3.2 virtual double FrictionInterface::calculateGasFrictionFactor (Data const & *data*, double *reynolds*, double *hdiameter*) const [pure virtual]

3.8.3.3 virtual double FrictionInterface::calculateLiquidFrictionFactor (Data const & *data*, double *reynolds*, double *hdiameter*) const [pure virtual]

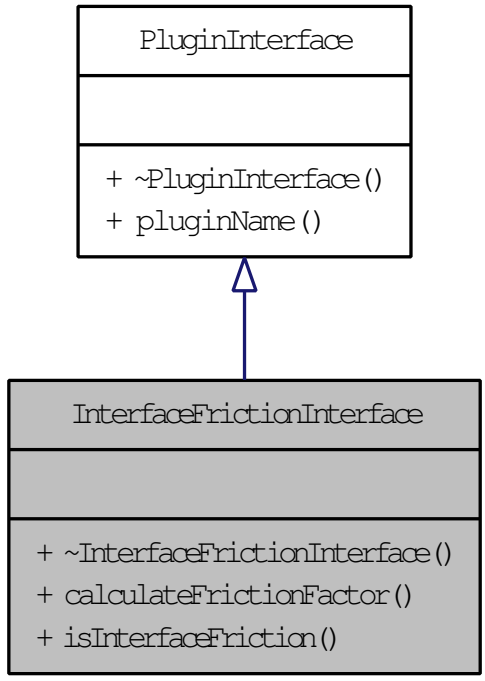
3.8.3.4 virtual bool FrictionInterface::isDispersedFriction () const [pure virtual]

3.8.3.5 virtual bool FrictionInterface::isGasWallFriction () const [pure virtual]

3.8.3.6 virtual bool FrictionInterface::isLiquidWallFriction () const [pure virtual]

3.9 InterfaceFrictionInterface Class Reference

Plugin interface for the creation of interface friction plugins. Inheritance diagram for InterfaceFrictionInterface:



Public Member Functions

- virtual ~InterfaceFrictionInterface ()
- virtual double calculateFrictionFactor (Data const &data, double alpha, double reynoldsGas, double hdiameterGas, double thicknessFilm, double frictionFactorGas) const =0
- virtual bool isInterfaceFriction () const =0

3.9.1 Detailed Description

Plugin interface for the creation of interface friction plugins.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 virtual InterfaceFrictionInterface::~InterfaceFrictionInterface () [inline, virtual]

3.9.3 Member Function Documentation

3.9.3.1 virtual double InterfaceFrictionInterface::calculateFrictionFactor (Data const & *data*, double *alpha*, double *reynoldsGas*, double *hdiameterGas*, double *thicknessFilm*, double *frictionFactorGas*) const [pure virtual]

3.9.3.2 virtual bool InterfaceFrictionInterface::isInterfaceFriction () const [pure virtual]

3.10 MainWindow Class Reference

Window class for main window.

Public Member Functions

- MainWindow (QWidget *parent=0)
- ~MainWindow ()

Protected Member Functions

- void changeEvent (QEvent *e)

3.10.1 Detailed Description

Window class for main window.

3.10.2 Constructor & Destructor Documentation

3.10.2.1 MainWindow::MainWindow (QWidget * *parent* = 0)

Constructor for MainWindow

Parameters:

parent Parent Widget

3.10.2.2 MainWindow::~MainWindow ()

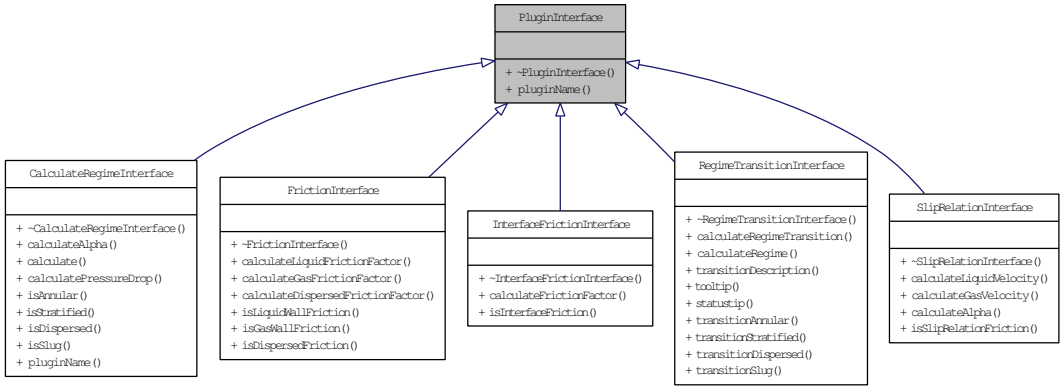
Destructor, calles function to save widget values before closing window

3.10.3 Member Function Documentation

3.10.3.1 void MainWindow::changeEvent (QEvent * *e*) [protected]

3.11 PluginInterface Class Reference

Base class to be derived by real plugininterfaces. Inheritance diagram for PluginInterface:



Public Member Functions

- virtual ~PluginInterface ()
- virtual QString pluginName () const =0

3.11.1 Detailed Description

Base class to be derived by real plugininterfaces.

3.11.2 Constructor & Destructor Documentation

3.11.2.1 virtual PluginInterface::~PluginInterface () [inline, virtual]

3.11.3 Member Function Documentation

3.11.3.1 virtual QString PluginInterface::pluginName () const [pure virtual]

Implemented in CalculateRegimeInterface.
Here is the caller graph for this function:

3.12 PluginLoader< T > Class Template Reference

Template class for loading plugins.

Static Public Member Functions

- static void loadPlugins (QDir pluginDir, std::vector< T > &retVector, QStringList filter=(QStringList("*")))
- static QObject * loadPlugin (QFileInfo filePath)

3.12.1 Detailed Description

template<class T> class PluginLoader< T >

Template class for loading plugins.

3.12.2 Member Function Documentation

3.12.2.1 template<class T > QObject * PluginLoader< T >::loadPlugin (QFileInfo *filePath*) [inline, static]

Here is the caller graph for this function:

3.12.2.2 template<class T > void PluginLoader< T >::loadPlugins (QDir *pluginDir*, std::vector< T > & *retVector*, QStringList *filter* = (QStringList("*"))) [inline, static]

Static function that loads the plugins in pluginDir to retVector

Parameters:

pluginDir QDir with path to directory where all files with names that if in filter List will be loaded

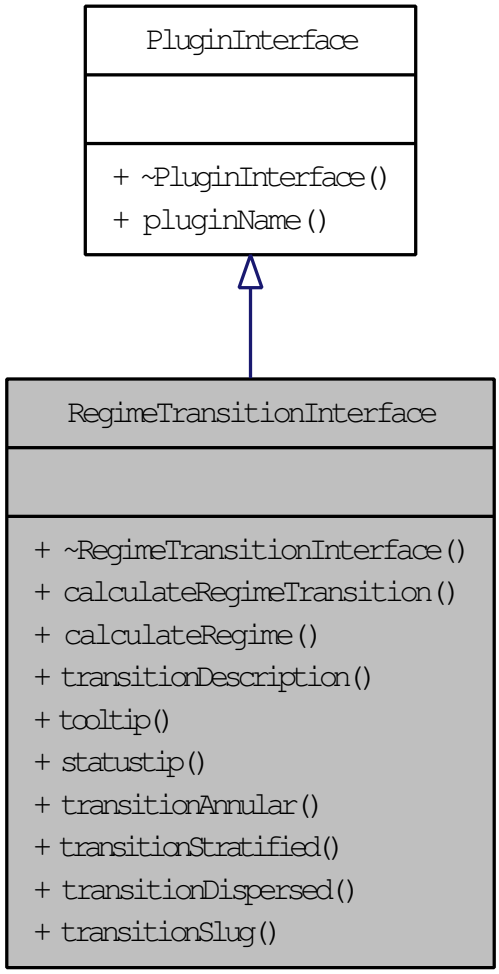
retVector Vector of pointers to plugin class inherited from PluginInterface, e.g. <FrictionInterface *>

filter QStringList of filters for what files to load

3.13 RegimeTransitionInterface Class Reference

Plugin interface for the creation of regime transition plugins. These plugins can be used to find the transition point in a Usg/Usl graph between different multiphase flow regimes. Inheritance

diagram for RegimeTransitionInterface:



Public Member Functions

- virtual ~RegimeTransitionInterface ()
- virtual double calculateRegimeTransition (Data *data, FrictionFunctions *frictionFunctions, CalculateFunctions *calculateFunctions, double tolerance, double Usg)=0
- virtual regimetype_t calculateRegime (Data *data, FrictionFunctions *frictionFunctions, CalculateFunctions *calculateFunctions, double tolerance, double Usg, double Usl)=0
- virtual QString transitionDescription () const =0
- virtual QString tooltip () const =0
- virtual QString statustip () const =0
- virtual bool transitionAnnular () const =0
- virtual bool transitionStratified () const =0
- virtual bool transitionDispersed () const =0
- virtual bool transitionSlug () const =0

3.13.1 Detailed Description

Plugin interface for the creation of regime transition plugins. These plugins can be used to find the transition point in a Usg/Usl graph between different multiphase flow regimes.

3.13.2 Constructor & Destructor Documentation

3.13.2.1 virtual RegimeTransitionInterface::~RegimeTransitionInterface () [inline, virtual]

3.13.3 Member Function Documentation

3.13.3.1 virtual regimetype_t RegimeTransitionInterface::calculateRegime (Data * *data*, FrictionFunctions * *frictionFunctions*, CalculateFunctions * *calculateFunctions*, double *tolerance*, double *Usg*, double *Usl*) [pure virtual]

3.13.3.2 virtual double RegimeTransitionInterface::calculateRegimeTransition (Data * *data*, FrictionFunctions * *frictionFunctions*, CalculateFunctions * *calculateFunctions*, double *tolerance*, double *Usg*) [pure virtual]

Here is the caller graph for this function:

3.13.3.3 virtual QString RegimeTransitionInterface::statustip () const [pure virtual]

3.13.3.4 virtual QString RegimeTransitionInterface::tooltip () const [pure virtual]

3.13.3.5 virtual bool RegimeTransitionInterface::transitionAnnular () const [pure virtual]

3.13.3.6 virtual QString RegimeTransitionInterface::transitionDescription () const [pure virtual]

3.13.3.7 virtual bool RegimeTransitionInterface::transitionDispersed () const [pure virtual]

3.13.3.8 virtual bool RegimeTransitionInterface::transitionSlug () const [pure virtual]

3.13.3.9 virtual bool RegimeTransitionInterface::transitionStratified () const [pure virtual]

3.14 SimulationStorage Class Reference

Class for storing and retrieving simulation data as csv files.

Public Member Functions

- SimulationStorage (simulationtype_t simulationType, int simulationID, QDir folderPath, const Data *data)
- ~SimulationStorage ()
- bool addData (QString dataName, std::vector< double > simulationData)
- void store (bool flow=true)

Static Public Member Functions

- static void listSimulationParameters (simulationtype_t simulationType, int idStart, int idEnd, QDir folderPath, std::vector< int > &ids, std::vector< QString > &regimeNames, std::vector< QString > &variableNames, std::vector< double > &variablesMin, std::vector< double > &variablesMax, std::vector< double > &Usgs, std::vector< double > &Usls, std::vector< double > &angles, std::vector< double > &diameters, std::vector< double > &roughnesses)
- static bool loadSimulation (simulationtype_t simulationType, int simulationID, QDir folderPath, std::vector< QString > &dataNames, std::vector< std::vector< double > > &simulationData)
- static bool deleteSimulation (simulationtype_t simulationType, int simulationID, QDir folderPath)
- static Data * loadSimulationParameters (simulationtype_t simulationType, int simulationID, QDir folderPath)
- static QString fileNamePrefix (simulationtype_t simulationType)

3.14.1 Detailed Description

Class for storing and retrieving simulation data as csv files.

3.14.2 Constructor & Destructor Documentation

3.14.2.1 SimulationStorage::SimulationStorage (simulationtype_t *simulationType*, int *simulationID*, QDir *folderPath*, const Data * *data*)

Constructor for storing simulations

Parameters:

simulationType What type of simulation has been run, of type simulationtype_t (SIMULATION_FLOW or SIMULATION_REGIMETRANSITION)

simulationID Unique identification number of simulation

folderPath Folder path to the file where the simulation will be stored

data Pointer to Data class with the parameters used to make simulation, const

3.14.2.2 SimulationStorage::~SimulationStorage ()

Clean up, delete Data

3.14.3 Member Function Documentation

3.14.3.1 bool SimulationStorage::addData (QString *dataName*, std::vector< double > *simulationData*)

Add data to the simulation storage Only store() will actually write data to disk

Parameters:

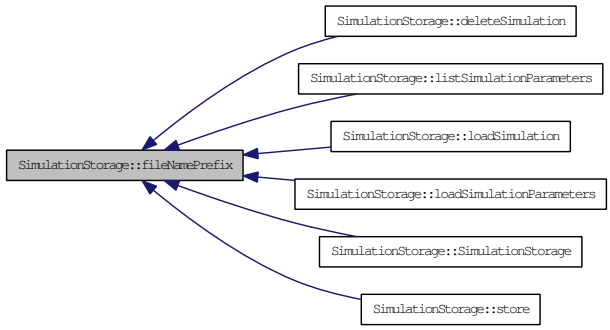
dataName Name (description) on what the data is

simulationData Vector of doubles with the data to store

3.14.3.2 bool SimulationStorage::deleteSimulation (simulationtype_t *simulationType*, int *simulationID*, QDir *folderPath*) [static]

3.14.3.3 QString SimulationStorage::fileNamePrefix (simulationtype_t *simulationType*) [static]

Here is the caller graph for this function:



3.14.3.4 void SimulationStorage::listSimulationParameters (simulationtype_t *simulationType*, int *idStart*, int *idEnd*, QDir *folderPath*, std::vector< int > & *ids*, std::vector< QString > & *regimeNames*, std::vector< QString > & *variableNames*, std::vector< double > & *variablesMin*, std::vector< double > & *variablesMax*, std::vector< double > & *Usgs*, std::vector< double > & *Usls*, std::vector< double > & *angles*, std::vector< double > & *diameters*, std::vector< double > & *roughnesses*) [static]

Read all simulations saved with identification numbers between idStart and idEnd and puts the data into std::vectors of double and QString This function is intended to be used for loading data into listview in MainWindow

Parameters:

simulationType What type of simulation has been run, of type simulationtype_t (SIMULATION_FLOW or SIMULATION_REGIMETRANSITION)

idStart Integer id of first simulation to load (list)

idEnd Integer id of last simulation to load (list)

folderPath Folder path to the file where the simulation will be stored

ids Vector of integers to where all ids will be written, data already in vector will be removed (Applies to all following vectors as well)

regimeNames Vector of QString to where all regime names will be written (shortened to 4 chars)

variableNames Vector of QString to where all variable names will be written

variablesMin Vector of double to where all minimum variables values will be written

variablesMax Vector of double to where all maximum variables values will be written

Usgs Vector of double to where all Usg values will be written

Usls Vector of double to where all Usl values will be written

angles Vector of double to where all angles values will be written

diameters Vector of double to where all diameters values will be written

roughnesses Vector of double to where all roughnesses values will be written

3.14.3.5 bool SimulationStorage::loadSimulation (simulationtype_t *simulationType*, int *simulationID*, QDir *folderPath*, std::vector< QString > & *dataNames*, std::vector< std::vector< double > > & *simulationData*) [static]

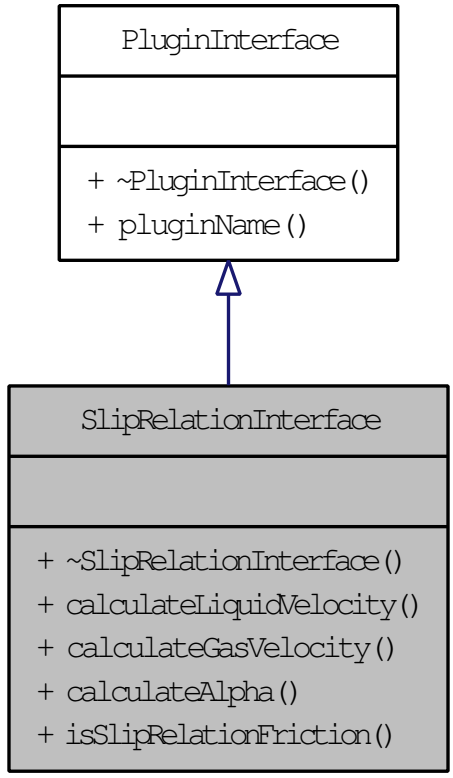
3.14.3.6 Data * SimulationStorage::loadSimulationParameters (simulationtype_t *simulationType*, int *simulationID*, QDir *folderPath*) [static]

3.14.3.7 void SimulationStorage::store (bool *flow* = true)

Write the data to disk Disk format is a semicolon separated csv file

3.15 SlipRelationInterface Class Reference

Inheritance diagram for SlipRelationInterface:



Public Member Functions

- virtual ~SlipRelationInterface ()
- virtual double calculateLiquidVelocity (Data const &data, double Usl, double Usg, double alpha) const =0
- virtual double calculateGasVelocity (Data const &data, double UlDisp, double alpha) const =0
- virtual double calculateAlpha (Data const &data, double alpha, double Usl, double Usg) const =0
- virtual bool isSlipRelationFriction () const =0

3.15.1 Constructor & Destructor Documentation

3.15.1.1 virtual SlipRelationInterface::~SlipRelationInterface () [inline, virtual]

3.15.2 Member Function Documentation

3.15.2.1 virtual double SlipRelationInterface::calculateAlpha (Data const & *data*, double *alpha*, double *Usl*, double *Usg*) const [pure virtual]

3.15.2.2 virtual double SlipRelationInterface::calculateGasVelocity (Data const & *data*, double *UlDisp*, double *alpha*) const [pure virtual]

3.15.2.3 virtual double SlipRelationInterface::calculateLiquidVelocity (Data const & *data*, double *Usl*, double *Usg*, double *alpha*) const [pure virtual]

3.15.2.4 virtual bool SlipRelationInterface::isSlipRelationFriction () const [pure virtual]

3.16 XYPlot Class Reference

Class for storing and retrieving simulation data as csv files.

Public Member Functions

- XYPlot (QWidget *parent, QString title, QString xAxisLabel="", QString yAxisLabel="")
- void addCurve (std::vector< double > xValuesList, std::vector< double > yValuesList, QString legend="")
- virtual void clear ()
- QwtPlot * Plot ()

Public Attributes

- QwtPlotZoomer * zoomer

3.16.1 Detailed Description

Class for storing and retrieving simulation data as csv files. Class for creating simple plots, based on QwtPlot.

3.16.2 Constructor & Destructor Documentation

3.16.2.1 XYPlot::XYPlot (QWidget * *parent*, QString *title*, QString *xAxisLabel* = "", QString *yAxisLabel* = "")

Constructor that creates plot base with title and labels

Parameters:

parent Parent Widget (parent window)

title Title of plot

xAxisLabel Label of x axis

yAxisLabel Title of y axis

3.16.3 Member Function Documentation

3.16.3.1 void XYPlot::addCurve (std::vector< double > *xValuesList*, std::vector< double > *yValuesList*, QString *legend* = "")

Adds a curve to plot, a plot can have an unlimited number of curves

Parameters:

xValuesList A vector of doubles of x values for the curve

yValuesList A vector of doubles of y values for the curve

legend Curve legend (

3.16.3.2 void XYPlot::clear () [virtual]

Remove all curves plotted. Reimplemented to reset _curves to zero so that curve colours are the same when starting a new plot

3.16.3.3 QwtPlot * XYPlot::Plot ()

Returns a pointer to the QwtPlot

3.16.4 Member Data Documentation

3.16.4.1 QwtPlotZoomer* XYPlot::zoomer

3.17 XYPlotArea Class Reference

Widget class which adds a XYPlot together with buttons for manipulating plot.

Public Member Functions

- XYPlotArea (QWidget *parent, QString title, QString xAxisLabel="", QString yAxisLabel="")
- XYPlot * Plot ()

3.17.1 Detailed Description

Widget class which adds a XYPlot together with buttons for manipulating plot.

3.17.2 Constructor & Destructor Documentation

3.17.2.1 XYPlotArea::XYPlotArea (QWidget * *parent*, QString *title*, QString *xAxisLabel* = "", QString *yAxisLabel* = "")

Constructor that creates plot area with title and labels

Parameters:

parent Parent Widget (parent window)

title Title of plot

xAxisLabel Label of x axis

yAxisLabel Title of y axis

3.17.3 Member Function Documentation

3.17.3.1 XYPlot * XYPlotArea::Plot ()

Returns pointer to XYPlot

Index