

RheoPlast: Marrying Phase Field and Fluid-Structure Interactions

Adam Powell Bo Zhou Jorge Vieyra Wanida Pongsaksawad

Version 0.5.0 released August 23, 2004
Built August 23, 2004 for a pc i686 running linux-gnu

Abstract

RheoPlast is a code framework for phase field solidification modeling with fluid flow and elastic solid behavior using a fluid-structure interactions formulation. It is designed to be modular and flexible, such that one can select on the command line between various phase field energy functions, transport coupling terms, initial and boundary conditions, in addition to the various parameters of the model. It is based on the data management objects and solvers of the **PETSc** suite, with small additions to aid in timestepping. The package also comes with a simple diffusion code using the **RheoPlast** infrastructure, and for historical reasons, a Cahn-Hilliard code based on the Master's thesis of David Dussault which was used to produce displays for several talks and a poster.

1 Introduction

There are lots of phase field codes out there, but the best and most open is this one. It's also the most modular, the most flexible, the highest-performance, well, what can I say, it's just the best! And its authors are the most modest...

Almost all of the documentation is generated from the source code comments by **cxref**; if you don't have **cxref** you're missing most of the documentation. Also, the use of postscript specials in some of the graphics result in omission of figures from the PDF documentation.

As for the name, no it's not rhinoplasty, but **RheoPlast**! This name comes from a combination of "rheology" and "plastic" since the mixed fluid-solid systems we hope to model are something like pseudoplastic fluids, and of course it rhymes with "rheocast", which is one of the processes we'd like to simulate.

Note that this manual assumes a basic familiarity with **PETSc**, the Portable Extensible Toolkit for Scientific Computing [1]. **PETSc** is an object-oriented library of data objects and solvers, and is available from Argonne National Laboratories at <http://www-unix.mcs.anl.gov/petsc/>; **PETSc** documentation can also be browsed at that site. **RheoPlast** also uses the **Illuminator** distributed visualization library, both for real-time display of simulation results and also for data storage; that library is at <http://lyre.mit.edu/~powell/illuminator.html>.

Share and enjoy.

2 Included Software

There are three programs shipped in the **rheoplast** package: **cahnhill**, **diffuse** and **rheoplast**. The big deal is **rheoplast**, so feel free to skip to that if you like.

2.1 The original: **cahnhill**

cahnhill is the original phase field/fluid-structure interactions code, included here largely for historical purposes. It was mostly written by David Dussault in 2001-2002 for a two-fluid Cahn-Hilliard system, and adapted by Adam Powell to do fluid-structure interactions as well. Note that it includes its own solver written by David Dussault, and its visualization format is plain text, because I developed and ran it on my laptop, which at the time didn't have enough memory to run X windows and the program's memory-hungry solver at the same time!

Also, note that it gets the pressure wrong. I don't know why this is, but it's a problem. It's here largely for historical purposes as its output was used in talks and posters for about a year. It will probably be removed after the release of version 0.5.

Sources:

1. `cahnhill-old.c` and `cahnhill-old.h` (appendices A and B, pages 7 and 11)
2. `dussolve.c` (section C, page 13)
3. `cahnout.c` (section D, page 13)

2.2 The testbed: diffuse

`diffuse` is a simple diffusion test case which uses the timestepping infrastructure in `timestep.c`, including the looping required for two and three dimensions, intelligent symmetry boundary conditions, and limited variable semi-implicit timestep size options. It does not use the temporary fields machinery, nor the ability to mix time derivatives and constraint equations in implicit timestepping. The `timestep.c` infrastructure is designed to be used in a variety of projects involving timestepping finite differences using PETSc distributed arrays.

Future features may involve Adams-Bashforth and Adams-Moulton timestepping for higher accuracy.

Sources:

1. `diffuse.c` (section E, page 14)
2. `timestep.c` and `timestep.h` (appendices H and I, pages 23 and 31): timestepping infrastructure.

2.3 The big enchilada: rheoplast

`rheoplast` is the big enchilada here, the thing worth downloading the package for.

Sources:

1. `rheoplast.c` and `rheoplast.h` (appendices F and G, pages 16 and 21): `main()` function, and timestep callbacks which evaluate temporary fields, functions and (eventually) Jacobians with the help of modules.
2. `timestep.c` and `timestep.h` (appendices H and I, pages 23 and 31): timestepping infrastructure described above.
3. `cahnhill.c` and `cahnhill.h` (appendices J and K, pages 33 and 38): a straightforward implementation of Cahn-Hilliard, this is a good “example module” which uses the features of `rheoplast` well, such as a temporary field for the chemical potential, and convective transport.
4. `vortflow.c` and `vortflow.h` (appendices L and M, pages 39 and 39): velocity-vorticity formulation fluid dynamics in 2-D. Documentation includes a complete mathematical description of the formulation.
5. `membrane.c` and `membrane.h` (appendices N and O, pages 44 and 44): Ternary polymer solution module, used to simulate immersion precipitation of polymer membranes.

Items 3 through 5 are `RheoPlast`’s *modules* which can be mixed and matched at the command line to do multi-physics simulations. For example, one can combine Cahn-Hilliard and velocity-vorticity flow to simulate two fluids, and optionally add shear strain in the mixed-stress model to do particle fluid-structure interactions. Some examples of such combinations are illustrated in table 1.

2.3.1 Adding a rheoplast module

Adding a module to `rheoplast` involves writing new source and header files for the module in the pattern of the existing modules, then modifying `rheoplast.c` and `rheoplast.h`. In broad terms, here are the steps to take:

- Add your module’s header file to *both* of the `#include` lists in `rheoplast.h`. It is included in two steps because the `AppCtx` structure typedefs need the module-specific structure typedefs, and the module function prototypes need the `AppCtx` structure typedef.

RheoPlast module	cahnhill	membrane	vortflow	pressflow	shearstrain	vectorphase	heatcond	electra
Field variables	C	ϕ_2, ϕ_3	u, v, ω	u, v, p	γ_{xx}, γ_{xy}	p, q	T	V
Temporary fields	μ	μ_2, μ_3		(ω)		$\phi, \epsilon, \tau, \partial\phi/\partial t$		σ_{eff}
Cahn-Hill. two-fluid	✓		✓					
Polym-soln demixing		✓	(✓)					
Cahn-Hilliard FSI	✓			✓	✓			
Polycrystal freezing						✓		
Dendritic freezing						✓	✓	
Polycrystal FSI				✓	✓	✓		
Dendrite FSI				✓	✓	✓	✓	
Cahn-Hill. e-chem	✓		(✓)	(✓)				✓

Table 1: Example simulations and their usage of present (first three) and future RheoPlast modules.

- Add your module’s parameter structure, and a `PetscTruth` flag indicating whether your module is used in a given simulation, to the main `AppCtx` structure also in `rheoplast.h`.
- Add your module’s `HELP` entry to the `help[]` string at the top of `rheoplast.c`.
- Search `rheoplast.c` for the word “modules” and look for where to add your module’s function calls, as described in appendix F. You should find places to add calls to your `first_setup`, `labels_initcond`, `temp_parameters_line` and `func_interior_line` functions, and a test for the command-line option activating your module.
- Add your module’s `.c` source file(s) to `rheoplast_SOURCES` in `Makefile.am`. Also add its `.h` header file(s) to `noinst_HEADERS`, and add all files to `BUILT_TEXFILES`. Making this change will require you to run the `autogen.sh` script again to rebuild `Makefile.in`, and then `configure` to rebuild the new `Makefile`.

A flowchart (not yet complete) showing the interactions among functions in `rheoplast.c`, `timestep.c`, the modules, and PETSc is shown in figure 1.

2.3.2 Future prospects

RheoPlast is a state-of-the-art uniform-grid finite difference code for doing these things. It will move in stages to version 0.9, with the addition of the `pressflow`, `shearstrain`, `vectorphase`, `heatcond` and `electra` modules mentioned above and a graphical user interface (GUI) based on `libglade`, then with a bit more testing, to 1.0.

But in late 2004, we will start to develop a new finite element code for phase field modeling, which will concentrate calculation power where it is needed and run considerably more efficiently. This might be pushed back by issues such as shapefunctions for higher-order PDEs (*e.g.* the biharmonic operator in Cahn-Hilliard requires continuous derivatives and doesn’t work with conventional finite element shapefunctions). But the point is, all of this finite difference RheoPlast code may be obsolete by mid-2005 or so, so while it’s a nice code for getting up to speed and obtaining good results quickly, don’t count on it to meet all of your needs for the indefinite future.

In the meantime, we are interested in feedback and ideas for Rheoplast2. The current feature wishlist includes:

- Finite elements, probably based on PETSc version 3.
- Dynamic mesh Lagrangian treatment of fluid flow following the methodology of Antaki *et al.* [4], so nodes move with the fluid, eliminating numerical diffusion and the need for upwinding.
- A real module system (likely based on `libgmodule`) which does not require modification of main source code to add a module. All of the functions, callbacks and data will be presented by the module at load-time.

2.4 Timestepping Infrastructure

This deserves a section to describe the nature of time-derivative vs. constraint equations, the symmetry boundary conditions, etc. I’ll write it later.

Figure 1: RheoPlast detailed flowchart.

3 RheoPlast and Literature

3.1 Phase Field and Fluid-Structure Interactions

The fluid-structure interactions methodology and preliminary results are described in a paper entitled “Floating Solids: Combining Phase Field and Fluid-Structure Interactions” [5], which was submitted to *J. Appl. Numer. Anal.* in May, 2004 and currently available at <http://lyre.mit.edu/~powell/papers/>. This paper used the old `cahnhill` program (to be removed after RheoPlast version 0.5 is released) with the default 3/8 cm square domain and properties of water (matching Jacqmin 1997). Dimensionless C is used with fourth-order polynomial double-well free energy.

When `cahnhill` is built, the solid impinging particle simulations can be replicated using the command line:

```
./cahnhill -nx 21 -ny 21 -dt 1.E-4 -time_factor 1.1 -dt_max 1.E-3
```

To run the oscillating solid simulation, open `cahnhill-old.c`, uncomment lines 1786-1787 (square initial condition) and comment lines 1801-1804 (to remove the impinging particles initial condition), then run with the same command line as above. To run the oscillating liquid droplet simulation, use this same square initial condition and change the “6” on line 104 of `cahnhill-old.c` to a “4”.

These results were also presented at the TMS Annual Meetings in February 2002 and March 2003, in a poster at the Physical Metallurgy Gordon Conference in July 2002, at MIT and Purdue Universities in December 2002, at Northwestern University in April 2003, at the University of Tokyo in January and July 2004, and at Boston University in July 2004.

The oscillating liquid droplet simulation can be approximately reproduced by rheoplast using the `vortflow` module and command line:

```
./rheoplast -da_grid_x 25 -da_grid_y 25 -width_x 0.00375 -width_y 0.00375  
-symmetry_x -symmetry_y -with_cahnhill -ch_surftens 0.03 -with_vortflow -density  
1000 -viscosity 0.0004 -ts_dt 0.00002 -ts_dt_max 0.001
```

In the future, the `shearstrain` module will enable the fluid-structure interactions simulations described by the paper and talks mentioned above.

3.2 Polymer Membranes

Bo Zhou presented a poster at the 2003 MRS Fall Meeting [6], a talk at the 2004 TMS Annual Meeting, and another paper at the 2004 Gordon Research Conference on Polymer Membranes, all based on the `membrane` and `vortflow` modules of `rheoplast`. These results are described in her paper entitled “Phase Field Simulations of Liquid-Liquid Demixing During Immersion Precipitation of Polymeric Membranes in 2D and 3D” [7]. The following command lines will duplicate the results of her simulations:

- Ternary spinodal decomposition with periodic boundary conditions:

```
rheoplast -with_membrane -da_grid_x 150 -da_grid_y 150 -width_x 1 -width_y 1  
-explicit_timesteps 4000 -explicit_deltat 2.5e-11 -explicit_monsteps 1000  
-ts_max_steps 20000 -ts_dt 1e-7 -monsteps 100 -K_ss 2e-5 -K_pp 2e-5  
-mobility_ss 2 -mobility_pp 2 -m_random -m_random_center_phi_s 0.2  
-m_random_center_phi_p 0.2 -m_random_fluct 0.005
```

Variations demonstrated the effect of K_{ss} and K_{pp} on lenscale using the `-Kss` and `-Kpp` parameters.

- 2-D CA-acetone-water decomposition without flow:¹

```
rheoplast -with_membrane -da_grid_x 150 -da_grid_y 300 -width_x 1 -width_y 2  
-explicit_timesteps 4000 -explicit_deltat 2.5e-11 -explicit_monsteps 1000  
-m_layers 0.3 -symmetry_y -ts_max_steps 20000 -ts_dt 1e-7 -monsteps 100 -Kss  
1e-4 -Kpp 1e-4
```

¹Requires modification to functions `psiprime2()` and `psiprime3()` (appendices N.1.8 and N.1.9, pages 48 and 48) which are not part of RheoPlast version 0.5.

This illustrated the effect of a somewhat different free energy function on membrane morphology, in this case the size and timescale of initial decomposition fluctuations are both larger than for PVDF (below).

- 2-D PVDF-DMF-water polymer solution-coagulant bath membrane decomposition without flow:

```
rheoplast -with_membrane -da_grid_x 150 -width_x 1 -width_y 2 -da_grid_y 300
-explicit_timesteps 2000 -explicit_deltat 2.5e-11 -explicit_monsteps 1000
-m_layers 0.3 -symmetry_y -K_ss 1e-4 -K_pp 1e-4 -ts_max_steps 20000 -ts_dt
1e-7 -monsteps 100
```

The base case simulations, illustrated effect of initial coagulant bath and polymer solution composition on membrane morphology.

- 3-D PVDF-DMF-water polymer solution-coagulant bath membrane decomposition without flow:

```
rheoplast -with_membrane -threedee -width_x 0.45 -da_grid_x 90 -width_y 1.0
-da_grid_y 200 -width_z 0.45 -da_grid_z 90 -ts_max_steps 40000 -ts_dt 1e-7
-monsteps 100 -m_layers 0.3 -symmetry_y -explicit_timesteps 2000
-explicit_deltat 1e-11 -explicit_monsteps 400
```

Three-dimensional version of the previous case, illustrates the mechanism of pore formation in the membrane surface.

- 2-D PVDF-DMF-water polymer solution-coagulant bath membrane decomposition with flow:

```
rheoplast -with_membrane -da_grid_x 150 -width_x 1 -width_y 2 -da_grid_y 300
-explicit_timesteps 4000 -explicit_deltat 2.5e-11 -explicit_monsteps 1000
-m_layers 0.3 -symmetry_y -ts_max_steps 8000 -ts_dt 1e-7 -monsteps 50 -K_ss
1e-4 -K_pp 1e-4 -with_vortflow -Sc 1000 -Fp 1e9 -snes_monitor
```

Demonstrates the effects of fluid flow on membrane morphology. In particular, the ratio of the Sc and Fp parameters (Schmidt number and force parameter) determines the stability of the membrane top layer.

References

- [1] S. Balay, W.D. Gropp, L. Curfman McInnes, B.F. Smith, “Efficient Management of Parallelism in Object Oriented Numerical Software Libraries”, *Modern Software Tools in Scientific Computing*, ed. E. Arge, A. M. Bruaset, H. P. Langtangen, Birkhauser Press (1997), 163–202.
- [2] R. Kobayashi, J.A. Warren, W.C. Carter, “Vector-valued phase field model for crystallization and grain boundary formation,” *Physica D* **119** (1998) 415–423.
- [3] R. Kobayashi, J.A. Warren, W.C. Carter, “A continuum model of grain boundaries,” *Physica D* **140** (2000) 141–150.
- [4] J. Antaki, G. Belloch, O. Ghattas, I. Malcevic, G. Miller, N. Walkington, “A Parallel Dynamic-Mesh Lagrangian Method for Simulation of Flows with Dynamic Interfaces,” *Proc. Sci. Comp.* **2000**.
- [5] A. Powell, “Floating Solids: Mixing Phase Field and Fluid-Structure Interactions”, submitted to *J. Appl. Numer. Anal.* May, 2004.
- [6] B. Zhou and A. Powell, “Simulations of Polymeric Membrane Formation by Immersion Precipitation: Liquid-Liquid Demixing,” *MRS Symp. Proc.* Fall Meeting, 2003.
- [7] Bo Zhou and Adam Powell, “Phase Field Simulations of Liquid-Liquid Demixing During Immersion Precipitation of Polymeric Membranes in 2D and 3D,” submitted to *J. Membrane Sci.* May, 2004.
- [8] M.-H. Giga, Y. Giga, *Arch. Rational Mech. Anal.* **141** (1998) 117.

- [9] R. Kobayashi, Y. Giga, *J. Stat. Phys.* **95** (1999) 1187.
- [10] D. Jacqmin, “Calculation of Two-Phase Navier-Stokes Flows Using Phase-Field Modeling,” *J. Comp. Phys.* **155** (1999) 96–127.

4 Copying

RheoPlast Phase Field Multi-Physics Code

Copyright (C) 2002, 2003, 2004 Adam Powell, David Dussault, Bo Zhou, Jorge Vieyra, Wanida Pongsaksawad

This code is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This code is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this code; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

You may contact the authors by email at hazelsct@mit.edu, dussault@aerodyne.com, bozhou@mit.edu, el.oso@mit.edu and wanida@mit.edu respectively.

5 Version History

5.1 RheoPlast 0.5

Planned first release including features:

- Legacy program `cahnhill`, to be removed after this release.
- Timestep infrastructure user `diffuse`, tenure uncertain.
- `rheoplast` with modules `cahnhill`, `vortflow` and `membrane`, including sufficient information to duplicate publications and presentations to date of Adam Powell and Bo Zhou.
- Links against Illuminator version 0.8.9 and PETSc 2.2.0.

A File `cahnhill-old.c`

Included Files

```
#include "cahnhill-old.h" (Section B)
#include </usr/lib/petsc/include/petscda.h>
#include <stdlib.h>
```

Preprocessor definitions

```
#define __FUNCT__ "main"
#define __FUNCT__ "get_properties"
#define __FUNCT__ "get_F"
#define __FUNCT__ "get_f"
#define __FUNCT__ "get_initial_conditions"
```

A.1 Variables

A.1.1 Variable `Lx`

```
double Lx
```

A.1.2 Variable Ly

double Ly

A.1.3 Variable eps

double eps

A.1.4 Variable alpha

double alpha

A.1.5 Variable beta

double beta

A.1.6 Variable timestep

int timestep

A.1.7 Variable last_timestep

int last_timestep

A.1.8 Variable Term_gravity

int Term_gravity

A.1.9 Variable Term_migration

int Term_migration

A.1.10 Variable Term_surface_tension

int Term_surface_tension

A.1.11 Variable Term_vis_var

int Term_vis_var

A.1.12 Variable Term_vis_comp

int Term_vis_comp

A.1.13 Variable Term_diffusion

int Term_diffusion

A.1.14 Variable Term_elastic_shear

int Term_elastic_shear

A.1.15 Variable compressible

int compressible

A.1.16 Variable run_steady

int run_steady

A.1.17 Variable dt

double dt

A.1.18 Variable sigma

double sigma

A.1.19 Variable xxprev

Vec xxprev

A.1.20 Variable fprev

Vec fprev

A.1.21 Variable rhoprev

Vec rhoprev

A.1.22 Variable rho

Vec rho

A.1.23 Variable vis

Vec vis

A.1.24 Variable div_V

Vec div_V

A.1.25 Variable kappa

double kappa

A.1.26 Variable D

double D

A.1.27 Variable fs

double fs

A.1.28 Variable Rconst

double Rconst

A.1.29 Variable Fconst

double Fconst

A.1.30 Variable M_Fe

double M_Fe

A.1.31 Variable M_O

double M_O

A.1.32 Variable T

double T

A.1.33 Variable rho_const

double rho_const

A.1.34 Variable vis_const

double vis_const

A.1.35 Variable rho_metal

double rho_metal

A.1.36 Variable rho_slag

double rho_slag

A.1.37 Variable vis_metal

double vis_metal

A.1.38 Variable vis_slag

double vis_slag

A.1.39 Variable shear_modulus

double shear_modulus

A.1.40 Variable gx

double gx

A.1.41 Variable gy

double gy

A.1.42 Variable kappa_scale

double kappa_scale

A.1.43 Variable D_slag

double D_slag

A.1.44 Variable c_metal

double c_metal

A.1.45 Variable c_slag

double c_slag

A.1.46 Variable omega_metal

double omega_metal

A.1.47 Variable omega_slag

```
double omega_slag
```

A.1.48 Variable const1

```
double const1
```

A.1.49 Variable couette

```
int couette
```

A.1.50 Variable couette_velocity

```
double couette_velocity
```

A.1.51 Variable stagnation

```
int stagnation
```

A.1.52 Variable stagnation_umax

```
double stagnation_umax
```

A.1.53 Local Variables

```
help
```

```
static char help[]
```

A.2 Functions

A.2.1 Global Function get_F()

```
int get_F ( Vec xx, Vec* F, AppCtx* user )
```

A.2.2 Global Function get_f()

```
int get_f ( Vec xx, Vec* F, AppCtx* user )
```

A.2.3 Global Function get_initial_conditions()

```
int get_initial_conditions ( Vec* xx, AppCtx* user )
```

A.2.4 Global Function get_properties()

```
int get_properties ( Vec* xx, AppCtx* user )
```

A.2.5 Global Function main()

```
int main ( int argc, char** args )
```

B File cahnhill-old.h

Included Files

```
#include </usr/lib/petsc/include/petscda.h>
```

```
#include <stdlib.h>
```

Preprocessor definitions

```
#define g2ng( i, j, nvar )
    #define g2nl( i, j, nvar )
    #define g2n1g( i, j )
    #define g2n1l( i, j )
    #define g2npg( i, j, nvar )
    #define g2npl( i, j, nvar )
    #define harmonic_mean( A, B )
    #define linear_mean( A, B )
    #define cubinterp_zero( phi )
    #define cubinterp_one( phi )
    #define C( i, j )
    #define u( i, j )
    #define v( i, j )
    #define p( i, j )
    #define gxx( i, j )
    #define gxy( i, j )
    #define Cprev( i, j )
    #define uprev( i, j )
    #define vprev( i, j )
    #define pprev( i, j )
    #define gxxprev( i, j )
    #define gxyprev( i, j )
    #define rho( i, j )
    #define rhoprev( i, j )
    #define vis( i, j )
    #define mu( i, j )
    #define psiprime( i, j )
    #define div_V( i, j )
```

B.1 Type definitions

B.1.1 Typedef AppCtx

```
typedef struct {...} AppCtx
struct
{
    double param;
    DA da;
    DA da1;
```

```

    DA daplot;
    PetscViewer binviewer;
    PetscViewer viewer;
    PetscDraw draw;
    int nx;
    int ny;
    int nodes;
    int unknowns;
    double dx;
    double dy;
    int nu;
    int nv;
    int np;
    int nC;
    int ngxx;
    int ngxy;
    int nvars;
    int nvarsplot;
    int view_gcr;
}

```

C File dussolve.c

Included Files

```

#include "cahnhill-old.h"
#include </usr/lib/petsc/include/petscda.h>
#include <stdlib.h>

```

(Section B)

Preprocessor definitions

```

#define __FUNCT__ "gcr_Mfree"
#define __FUNCT__ "gcr"

```

C.1 Functions

C.1.1 Global Function gcr()

```
int gcr ( Mat M, Vec xx, Vec F, Vec* dxx, double epsilon, double tol, AppCtx* user )
```

C.1.2 Global Function gcr_Mfree()

```
int gcr_Mfree ( Vec xx, Vec F, Vec* dxx, double epsilon, double tol, AppCtx* user )
```

D File cahnout.c

Included Files

```

#include "cahnhill-old.h"
#include </usr/lib/petsc/include/petscda.h>
#include <stdlib.h>
#include <string.h>

```

(Section B)

Preprocessor definitions

```
#define __FUNCT__ "get_drop_radius"
#define __FUNCT__ "get_Vplot"
#define __FUNCT__ "myDAviewer"
```

D.1 Functions

D.1.1 Global Function get_Vplot()

```
int get_Vplot ( Vec xx, Vec rho, Vec* Vplot, AppCtx* user )
```

D.1.2 Global Function get_drop_radius()

```
int get_drop_radius ( Vec xx, AppCtx* user, double* drop_xradius, double* drop_yradius, double*
drop_dradius, double* drop_mass )
```

D.1.3 Global Function myDAviewer()

```
int myDAviewer ( DA theda, Vec X, int nlevels, PetscScalar* levels, char* symbols )
```

E File diffuse.c

RCS Header: /cvsroot/rheoplast/diffuse.c,v 1.26 2004/08/18 21:19:52 hazelsct Exp

This uses the new explicit timestepping framework to solve the diffusion equation in finite differences really, really fast. (Yes, diffusion is boring, but you gotta start somewhere. :-)

Included Files

```
#include "timestep.h" (Section I)
#include </usr/lib/petsc/include/petscsnes.h>
#include </usr/lib/petsc/include/petscda.h>
#include </usr/lib/petsc/include/petscblaslapack.h>
#include <stdlib.h>
#include <illuminator.h>
#include </usr/lib/petsc/include/petscblaslapack.h>
```

Preprocessor definitions

```
#define DPRINTF( fmt, args... )
```

E.1 Type definitions

E.1.1 Typedef AppCtx

```
typedef struct {...} AppCtx
struct
{
    DA theda;
    Vec global;
    Vec local;
```

```

    PetscScalar mesh_fourier_x;
    PetscScalar mesh_fourier_y;
    PetscScalar mesh_fourier_z;
    int plotsteps;
    PetscTruth contours;
    PetscTruth twodee;
    PetscViewer theviewer;
    CommStyle style;
}

```

E.2 Variables

E.2.1 Local Variables

help

PETSc help string, printed when run with -help.

```
static char help[]
```

E.3 Functions

E.3.1 Global Function `func_interior_line()`

```
void func_interior_line ( PetscScalar* x, PetscScalar* func, PetscScalar* temp, PetscTruth**
mixed_constraints, PetscScalar time, int points, int gxm, int gym, int gzm, int xs, int ys, int zs,
int xm, int nx, int ny, int nz, void* user )
```

E.3.2 Global Function `jack_interior_line()`

```
void jack_interior_line ( PetscScalar* x, PetscScalar* temp, Mat jack, PetscScalar time, int
points, int gxm, int gym, int gzm, int xs, int ys, int zs, int xm, int nx, int ny, int nz, int
firstrow, void* user )
```

E.3.3 Global Function `main()`

```
int main ( int argc, char* argv[] )
```

E.3.4 Global Function `step_interior_line()`

This performs one explicit timestep for a line of data in the 2-D or 3-D array. In this simple example, it uses BLAS functions (defined in `petscblaslapack.h`) to do this extremely fast. Fastest explicit diffusion solver you'll find!

```
void step_interior_line ( PetscScalar* old, PetscScalar* new, PetscScalar* temps, PetscTruth**
mixed_constraints, PetscScalar time, int points, int gxm, int gym, int gzm, int xs, int ys, int zs,
int xm, int nx, int ny, int nz, void* user )
```

- `PetscScalar* old` Field variable array in previous timestep.
- `PetscScalar* new` Field variable array to fill in new timestep.
- `PetscScalar* temps` Temporary parameters (empty here) calculated by `temp_parameters_line`.
- `PetscTruth** mixed_constraints` Arrays of boolean variables indicating constraint equations in mixed timestep-constraint fields.
- `PetscScalar time` Current simulation time.
- `int points` Number of points to calculate at.
- `int gxm` Overall x -width of the array (for y increment).
- `int gym` Overall y -width of the array (for z increment).

- `int gzm` Overall z -width of the array (zero if 2-D).
- `int xs` Starting x -coordinate of the line.
- `int ys` Starting y -coordinate of the line.
- `int zs` Starting z -coordinate of the line.
- `int xm` Width of the interior part of the line.
- `int nx` Overall x -width of the entire global array.
- `int ny` Overall y -width of the entire global array.
- `int nz` Overall z -width of the entire global array (zero if 2-D).
- `void* user` User data structure pointer.

E.3.5 Global Function `temp_parameters_line()`

This is not necessary, so it does nothing.

```
void temp_parameters_line ( PetscScalar* x, PetscScalar* temp, PetscScalar time, int points, int
gxm, int gym, int gzm, int xs, int ys, int zs, int xm, int nx, int ny, int nz, void* user )
```

- `PetscScalar* x` Array of unknowns.
- `PetscScalar* temp` Array of temporary parameters to calculate.
- `PetscScalar time` Current simulation time.
- `int points` Number of points to calculate at.
- `int gxm` Overall x -width of the array (for y increment).
- `int gym` Overall y -width of the array (for z increment).
- `int gzm` Overall z -width of the array (zero if 2-D).
- `int xs` Starting x -coordinate of the line.
- `int ys` Starting y -coordinate of the line.
- `int zs` Starting z -coordinate of the line.
- `int xm` Width of the interior part of the line.
- `int nx` Overall x -width of the entire global array.
- `int ny` Overall y -width of the entire global array.
- `int nz` Overall z -width of the entire global array (zero if 2-D).
- `void* user` User data structure pointer.

E.3.6 Global Function `tsmonitor()`

This plots the current data, if the step number is right (according to the `plotsteps` and `explicit_plotsteps` command line options).

```
void tsmmonitor ( int time, PetscScalar realtime, PetscScalar deltat, void* user )
```

- `int time` Current timestep number.
- `PetscScalar realtime` Current simulation time.
- `PetscScalar deltat` Current timestep size.
- `void* user` User data structure pointer.

F File `rheoplast.c`

RCS Header: `/cvsroot/rheoplast/rheoplast.c,v 1.90 2004/08/23 14:51:52 hazelsct Exp`

This is the main RheoPlast file. If you want to add a module, search this file for the word "modules" to help you know where to insert your function calls, HELP definition, etc. Also see section \ref{newmodule} for other files which need to be modified in the process of adding a module.

Included Files

```
#include "rheoplast.h" (Section G)
#include </usr/lib/petsc/include/petsc.h>
#include <stdlib.h>
#include <illuminator.h>
#include </usr/lib/petsc/include/petscts.h>
#include </usr/lib/petsc/include/petscblaslapack.h>
#include <stdlib.h>
#include <time.h>
#include <sys/times.h>
#include "timestep.h" (Section I)
#include </usr/lib/petsc/include/petscsnes.h>
#include </usr/lib/petsc/include/petscda.h>
#include </usr/lib/petsc/include/petscblaslapack.h>
#include <stdlib.h>
#include "cahnhill.h" (Section K)
#include </usr/lib/petsc/include/petsc.h>
#include "vortflow.h" (Section M)
#include </usr/lib/petsc/include/petsc.h>
#include "membrane.h" (Section O)
#include </usr/lib/petsc/include/petsc.h>
#include "cahnhill.h" (Section K)
#include "vortflow.h" (Section M)
#include "membrane.h" (Section O)
```

Preprocessor definitions

```
#define __FUNCT__ "temp_parameters_line"
#define __FUNCT__ "func_interior_line"
#define __FUNCT__ "step_interior_line"
#define __FUNCT__ "jack_interior_line"
#define __FUNCT__ "thets_rhs"
```

This provides a right hand side vector for PETSc's (semi-)implicit timestepping solvers using the function `func_interior_line`.

This should probably go into `timestep.h` since it is generic. In the future, it will calculate and store temporary parameters only on an "as-needed" basis, which is to say, it will calculate and store them only at the line of calculation and its neighbor lines (neighbor planes needed in 3-D). This should save quite a bit of memory.

This is somewhat deprecated, now that `timestep.c` has constrained (semi-)implicit timestepping which bypasses it (the `implicit_steptime` function). But if such capability is up-ported into PETSc, then this will be useful again.

```
int thets_rhs It returns zero (or an error code).
TS thets Timestepping context from PETSc.
PetscScalar time Current time.
Vec unk Vector of unknowns from which to calculate functions.
Vec func Vector into which to put function values.
void *user User data structure pointer.
```

```

#define __FUNCT__ "tsmonitor"
#define DPRINTF( fmt, args... )
#define __FUNCT__ "main"

```

F.1 Variables

F.1.1 Local Variables

help

PETSc help string, printed when run with `-help`. Note that it includes entries for each of the modules' header files.

```
static char help[]
```

F.2 Functions

F.2.1 Global Function `func_interior_line()`

Evaluate the functions which make up this system on one line of interior points. The theory is that a line is big enough that the function call and other overheads are really small and we can do acceleration using level 1 BLAS if appropriate, but small enough to be really simple.

```
void func_interior_line ( PetscScalar* x, PetscScalar* func, PetscScalar* temp, PetscTruth**
mixed_constraints, PetscScalar time, int points, int gxm, int gym, int gzm, int xs, int ys, int zs,
int xm, int nx, int ny, int nz, void* user )
```

- `PetscScalar* x` The data to evaluate the function for.
- `PetscScalar* func` Where to put the evaluated function.
- `PetscScalar* temp` Array of temporary field variables.
- `PetscTruth** mixed_constraints` Arrays of boolean variables indicating constraint equations in mixed timestep-constraint fields.
- `PetscScalar time` Current simulation time.
- `int points` Number of points to evaluate at.
- `int gxm` The x -width of the “local” vector’s array, including shadow nodes, for the y -increment.
- `int gym` The y -width of the “local” vector’s array, including shadow nodes, for the z -increment.
- `int gzm` Overall z -width of the array (zero if 2-D).
- `int xs` Starting x -coordinate of the line.
- `int ys` Starting y -coordinate of the line.
- `int zs` Starting z -coordinate of the line.
- `int xm` Width of the interior part of the line.
- `int nx` Overall x -width of the entire global array.
- `int ny` Overall y -width of the entire global array.
- `int nz` Overall z -width of the entire global array (zero if 2-D).
- `void* user` User data structure pointer.

This function simply calls the `interior_line_function` function from each of the modules.

F.2.2 Global Function `jack_interior_line()`

Evaluate the Jacobian for this system on one line of interior points. The theory is that a line is big enough that the function call and other overheads are really small and we can do acceleration using level 1 BLAS if appropriate, but small enough to be really simple.

```
void jack_interior_line ( PetscScalar* x, PetscScalar* temp, Mat jack, PetscScalar time, int
points, int gxm, int gym, int gzm, int xs, int ys, int zs, int xm, int nx, int ny, int nz, int
firstrow, void* user )
```

- `PetscScalar* x` The data to evaluate the function for.
- `PetscScalar* temp` Array of temporary field variables.
- `Mat jack` Matrix into which to insert Jacobian values.
- `PetscScalar time` Current simulation time.
- `int points` Number of points to evaluate at.
- `int gxm` The x -width of the “local” vector’s array, including shadow nodes, for the y -increment.
- `int gym` The y -width of the “local” vector’s array, including shadow nodes, for the z -increment.
- `int gzm` Overall z -width of the array (zero if 2-D).
- `int xs` Starting x -coordinate of the line.
- `int ys` Starting y -coordinate of the line.
- `int zs` Starting z -coordinate of the line.
- `int xm` Width of the interior part of the line.
- `int nx` Overall x -width of the entire global array.
- `int ny` Overall y -width of the entire global array.
- `int nz` Overall z -width of the entire global array (zero if 2-D).
- `int firstrow`
- `void* user` User data structure pointer.

This function simply calls the `interior_line_jacobian` function from each of the modules (if it exists).

F.2.3 Global Function `main()`

This is `main()`.

```
int main ( int argc, char* argv[] )
```

- `int main` It returns an int to the OS.
- `int argc` Argument count.
- `char* argv[]` Arguments.

After PETSc and log file initialization, it sets the variable indices and labels based on the user-chosen equations.

We then determine which of the modules will be included in the simulation, and call those modules’ initialization functions to determine the number of field variables and temporary fields.

It then sets the basic timestepping parameters, from the command line if appropriate.

Next it creates the PETSc distributed arrays, gets the sizes of the local and global boxes, and sets the inverse square grid spacings.

Next it sets the initial condition, by getting the global array and calling the modules’ `labels_initcond` functions.

It then initializes the graphics and runs the explicit timestepping solver from `timestep.h`.

If directed to do so with a nonzero `-ts_max_steps` command line setting, it then initializes and runs semi-implicit timesteps.

An alternative semi-implicit section uses PETSc’s semi-implicit timestepping solver. It is semi-deprecated due to the constrained semi-implicit solver in `implicit_steptime`, but may come back if that functionality is up-ported into PETSc.

Finally, it cleans up, closing the graphics displays and freeing all memory.

F.2.4 Global Function `step_interior_line()`

This calculates the dynamic functions using the function `func_interior_line` and then uses them and the timestep size to determine the explicit timestepping changes to the field variables, on one line.

```
void step_interior_line ( PetscScalar* old, PetscScalar* new, PetscScalar* temp, PetscTruth**
mixed_constraints, PetscScalar time, int points, int gxm, int gym, int gzm, int xs, int ys, int zs,
int xm, int nx, int ny, int nz, void* user )
```

- `PetscScalar* old` Field variable array in previous timestep.
- `PetscScalar* new` Field variable array to fill in new timestep.
- `PetscScalar* temp` Temporary parameters calculated by `temp_parameters_line`.
- `PetscTruth** mixed_constraints` Arrays of boolean variables indicating constraint equations in mixed timestep-constraint fields.
- `PetscScalar time` Current simulation time.
- `int points` Number of points to calculate at.
- `int gxm` Overall x -width of the array (for y increment).
- `int gym` Overall y -width of the array (for z increment).
- `int gzm` Overall z -width of the array (zero if 2-D).
- `int xs` Starting x -coordinate of the line.
- `int ys` Starting y -coordinate of the line.
- `int zs` Starting z -coordinate of the line.
- `int xm` Width of the interior part of the line.
- `int nx` Overall x -width of the entire global array.
- `int ny` Overall y -width of the entire global array.
- `int nz` Overall z -width of the entire global array (zero if 2-D).
- `void* user` User data structure pointer.

F.2.5 Global Function `temp_parameters_line()`

This function calculates the temporary parameters on which the function calculation is based (in section F.2.1). The theory is that at some point, we'll calculate only the lines in the neighborhood of the current function line, and save gobs and gobs of memory; meanwhile, calculating these once saves a good amount of time.

```
void temp_parameters_line ( PetscScalar* x, PetscScalar* temp, PetscScalar time, int points, int
gxm, int gym, int gzm, int xs, int ys, int zs, int xm, int nx, int ny, int nz, void* user )
```

- `PetscScalar* x` Array of unknowns from which temp parameters are calculated.
- `PetscScalar* temp` Storage for one line of temp parameters.
- `PetscScalar time` Current simulation time.
- `int points` Number of points to evaluate at.
- `int gxm` The x -width of the "local" vector's array, including shadow nodes, for the y -increment.
- `int gym` The y -width of the "local" vector's array, including shadow nodes, for the z -increment.
- `int gzm` Overall z -width of the array (zero if 2-D).
- `int xs` Starting x -coordinate of the line.
- `int ys` Starting y -coordinate of the line.
- `int zs` Starting z -coordinate of the line.
- `int xm` Width of the interior part of the line.

- `int nx` Overall x -width of the entire distributed array.
- `int ny` Overall y -width of the entire distributed array.
- `int nz` Overall z -width of the entire distributed array (zero if 2-D).
- `void* user` User data structure pointer.

This function simply calls the `temp_parameters_line` function from each of the modules.

F.2.6 Global Function `tsmonitor()`

This plots the current data, if the step number is right (according to the `monsteps` and `explicit_monsteps` command line options).

- ```
void tsmoitor (int step, PetscScalar time, PetscScalar deltat, void* user)
```
- `int step` Current timestep number.
  - `PetscScalar time` Current simulation time.
  - `PetscScalar deltat` Current timestep size.
  - `void* user` User data structure pointer.

In addition to printing various diagnostics ( $\Delta t$ , min/max field values, etc.), this also calculates the "time velocity" since the last call to `tsmonitor`, which is the simulated time divided by CPU time spent in this process.

## G File `rheoplast.h`

**RCS Header:** `/cvsroot/rheoplast/rheoplast.h,v 1.30 2004/08/23 14:33:28 hazelsct Exp`

This `#includes` all of the the typedefs for the various field variable structures, and their function prototypes, and has the main `AppCtx` and `DPRINTF()`.

### Included Files

```
#include </usr/lib/petsc/include/petsc.h>

#include <stdlib.h>

#include <illuminator.h>

#include </usr/lib/petsc/include/petscts.h>

#include </usr/lib/petsc/include/petscblaslapack.h>

#include <stdlib.h>

#include <time.h>

#include <sys/times.h>

#include "timestep.h" (Section I)
 #include </usr/lib/petsc/include/petscsnes.h>
 #include </usr/lib/petsc/include/petscda.h>
 #include </usr/lib/petsc/include/petscblaslapack.h>
 #include <stdlib.h>

#include "cahnhill.h" (Section K)
 #include </usr/lib/petsc/include/petsc.h>

#include "vortflow.h" (Section M)
 #include </usr/lib/petsc/include/petsc.h>
```

```

#include "membrane.h" (Section O)
 #include </usr/lib/petsc/include/petsc.h>

#include "cahnhill.h" (Section K)
#include "vortflow.h" (Section M)
#include "membrane.h" (Section O)

```

## Preprocessor definitions

```

#define RHEOPLAST_H
 #define DPRINTF(fmt, args...)
 #define APPCTX_DEFINED

```

### G.1 Type definitions

#### G.1.1 Typedef AppCtx

```

typedef struct {...} AppCtx
struct
{
 DA theda;
 Vec global;
 Vec local;
 Vec localfunc;
 PetscScalar* parameters;
 PetscScalar xwid;
 PetscScalar ywid;
 PetscScalar zwid;
 PetscScalar deltax_m2;
 PetscScalar deltay_m2;
 PetscScalar deltaz_m2;
 PetscScalar deltat;
 int expsteps;
 int impsteps;
 int current_timestep;
 int monsteps;
 int vars;
 int tempvars;
 int symmflags;
 int* symmtypes;
 char** label;
 char* save_basename;
 FILE* logfile;
 field_plot_type* plot_types;
 PetscTruth contours;
 PetscTruth jacobian;
 PetscTruth threedee;
}

```

```

PetscTruth load_data;
EqStyle* timestyle;
PetscViewer theviewer;
Mat J;
CommStyle style;
PetscTruth cahnhill;
PetscTruth vortflow;
PetscTruth membrane;
chparm thecahnhill;
vortparm thevortex;
mparm themembrane;
}

```

## H File timestep.c

**RCS Header:** /cvsroot/rheoplast/timestep.c,v 1.64 2004/08/23 17:52:37 hazelsct Exp

This file includes an explicit timestepper and a semi-implicit one. The explicit timestepper can optimize for slow or fast communication; fast is really no different than traditional, but slow does asynchronous I/O to communicate the content of the interface nodes while computing the inner ones, and should work quite a bit better for bandwidth-limited Beowulf clusters. In practice, "fast" is quite a bit faster, because the short loops involved in computing the interface nodes in the asynchronous version slow things down considerably.

It also does "constrained implicit" timestepping, which differs from PETSc's timestepping algorithms in that where the latter requires that all functions be of the form:

$$\frac{\partial u}{\partial t} = f(u),$$

it is often necessary to have some fields described by "constraint" functions of the form:

$$f(u) = 0.$$

For example, incompressible Navier-Stokes has the motion equations which can be written as the former time-derivatives, and the continuity equation which must be written in the latter constraint form. The `implicit_steptime` function allows for those two types of equations to be mixed.

Both of these steppers implement temporary fields, which are calculated from the solved field variables in order to simplify the programming, save time, and in the case of rheoplast, pass needed information between modules.

When I have some time, I'll consider implementing these within PETSc's TS timestepping object class, and then submitting a patch upstream.

### Included Files

```

#include "timestep.h" (Section I)
#include </usr/lib/petsc/include/petscsnes.h>
#include </usr/lib/petsc/include/petscda.h>
#include </usr/lib/petsc/include/petscblaslapack.h>
#include <stdlib.h>

```

### Preprocessor definitions

```

#define __FUNCT__ "xmin.symm"
#define __FUNCT__ "xmax.symm"
#define __FUNCT__ "ymin.symm"
#define __FUNCT__ "ymax.symm"

```

```

#define __FUNCT__ "zmin_symm"
#define __FUNCT__ "zmax_symm"
#define __FUNCT__ "explicit_steptime"
#define __FUNCT__ "FuncEvaluate"
#define __FUNCT__ "ts_implicit_function"
#define __FUNCT__ "ts_implicit_jacobian"
#define __FUNCT__ "implicit_steptime"

```

## H.1 Variables

### H.1.1 Local Variables

#### **implicit\_da**

```
static DA implicit_da
```

#### **implicit\_jack**

```
static Mat implicit_jack
```

#### **un**

```
static Vec un
```

#### **temp\_local**

```
static Vec temp_local
```

#### **Fun**

```
static Vec Fun
```

#### **Fun\_local**

```
static Vec Fun_local
```

#### **Funp1**

```
static Vec Funp1
```

#### **Funp1\_local**

```
static Vec Funp1_local
```

#### **current\_time**

```
static PetscScalar* current_time
```

#### **temp\_fields**

```
static PetscScalar* temp_fields
```

#### **implicit\_deltat**

```
static PetscScalar implicit_deltat
```

#### **implicit\_timestyle**

```
static EqStyle* implicit_timestyle
```

#### **mixed\_constraints**

```
static PetscTruth** mixed_constraints
```

#### **dof**

```
static int dof
```

#### **nx**

```
static int nx
```

#### **ny**

```
static int ny
```

```

nz
static int nz

xs
static int xs

ys
static int ys

zs
static int zs

xm
static int xm

ym
static int ym

zm
static int zm

gxs
static int gxs

gys
static int gys

gzs
static int gzs

gxm
static int gxm

gym
static int gym

gzm
static int gzm

implicit_tps
static int implicit.tps

implicit_symmflags
static int implicit.symmflags

implicit_symmtypes
static int* implicit.symmtypes

implicit_dadims
static int implicit.dadims

```

## H.2 Functions

### H.2.1 Global Function `explicit_steptime()`

This is the explicit function, it does the timestepping with the slow- or fast-communication optimization. Note that the slow-communication optimization assumes that calculations can be done between the start and end of global to local scattering, that is during communication; performance testing seems to be showing that this is not the case, or else the gain from doing this is minimal... Also note: neither temporary field variables nor symmetry boundary conditions are implemented in the slow communication section!

```

int explicit_steptime (DA theda, Vec globalreal, Vec localreal, int* currentstep, int timesteps,
PetscScalar* currenttime, PetscScalar deltat, int tps, CommStyle comm, int symmflags, int*
symmtypes, void* user, FILE* logfile)

```

|                                         |                                                                                                                 |
|-----------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| · <code>int explicit_steptime</code>    | Returns 0 or error code: -1 for malloc error, -2 for non-periodic DA (only can use fully-periodic DAs for now). |
| · <code>DA theda</code>                 | The DA object we're working on.                                                                                 |
| · <code>Vec globalreal</code>           | Global vector of all fields.                                                                                    |
| · <code>Vec localreal</code>            | Local vector of all fields.                                                                                     |
| · <code>int* currentstep</code>         | Entering: initial timestep number, return: final timestep number.                                               |
| · <code>int timesteps</code>            | Number of timesteps to run.                                                                                     |
| · <code>PetscScalar* currenttime</code> | Entering: initial time, return: final time.                                                                     |
| · <code>PetscScalar deltat</code>       | Timestep size.                                                                                                  |
| · <code>int tps</code>                  | Number of temporary parameters to store in array.                                                               |
| · <code>CommStyle comm</code>           | Communication optimization to use (see typedef enum above).                                                     |
| · <code>int symmflags</code>            | Flags controlling symmetry boundary conditions.                                                                 |
| · <code>int* symmtypes</code>           | Array of symmetry types for each dof (if NULL then assumes all are MIRROR_PLANE).                               |
| · <code>void* user</code>               | User parameters' structure.                                                                                     |
| · <code>FILE* logfile</code>            | Log file to fprintf debugging information to (or NULL).                                                         |

## H.2.2 Global Function `implicit_steptime()`

This function implements constrained implicit timestepping as described above.

```
int implicit_steptime (DA theda, Vec unp1, Vec unp1_local, Mat jack, int* currentstep, int
timesteps, PetscScalar* currenttime, PetscScalar deltat, int tps, EqStyle* timestyle, int
symmflags, int* symmtypes, void* user, FILE* logfile)
```

|                                         |                                                                                   |
|-----------------------------------------|-----------------------------------------------------------------------------------|
| · <code>int implicit_steptime</code>    | Returns 0 or error code: -1 for malloc error.                                     |
| · <code>DA theda</code>                 | The DA object we're working on.                                                   |
| · <code>Vec unp1</code>                 | Global vector of all fields.                                                      |
| · <code>Vec unp1_local</code>           | Local vector of all fields.                                                       |
| · <code>Mat jack</code>                 | Matrix to use as Jacobian.                                                        |
| · <code>int* currentstep</code>         | Entering: initial timestep number, return: final timestep number.                 |
| · <code>int timesteps</code>            | Number of timesteps to run.                                                       |
| · <code>PetscScalar* currenttime</code> | Entering: initial time, return: final time.                                       |
| · <code>PetscScalar deltat</code>       | Timestep size.                                                                    |
| · <code>int tps</code>                  | Number of temporary field variables to store in array.                            |
| · <code>EqStyle* timestyle</code>       | Constraint, time derivative, or blend nature of each field equation.              |
| · <code>int symmflags</code>            | Flags controlling symmetry boundary conditions.                                   |
| · <code>int* symmtypes</code>           | Array of symmetry types for each dof (if NULL then assumes all are MIRROR_PLANE). |
| · <code>void* user</code>               | User parameters' structure.                                                       |
| · <code>FILE* logfile</code>            | Log file to fprintf debugging information to (or NULL).                           |

First we copy arguments into static variables, get DA info, allocate temporary field variable storage, set up the vectors to store variable and function data.

Variable timesteps are of just one variety for now: exponential growth of `deltat` with timestep number up to a maximum value. This is controlled by the options `-ts.dt_max`, which sets the maximum value, and `-ts.dt_factor`, which is the amount by which to multiply `dt` at the beginning of each timestep (default 1.1).

The symmetry flag `SYMMETRY_ZERO_INSIDE` requires that the residual at the symmetry plane of nodes be set to the value of the field variable, so the solver will set that field variable to zero there. This in

turns require that the field variable's timestyle be either `CONSTRAINT_ONLY` or `TIME_CONST_BLEND` so the residual can be directly set at those nodes (done in function `FuncEvaluate`, section H.2.5). Equations of type `TIME_CONST_BLEND` are given `PetscTruth` arrays indicating constraint status for all mixed fields over all points in the local part of the global array (i.e. not including ghost points). Temporary fields are given one large `PetscScalar` array for all temporary fields over all points (including ghost points) ordered by z, y, x, then (temporary or mixed) field variable. Both of these arrays are ordered by z, y, x, then in the latter case temporary field variable, which is similar to PETSc vector array orderings.

Now we're ready to create the various function vectors, and evaluate the time derivatives in this initial state.

Next we initialize the PETSc SNES object used to solve the equations in each timestep. If `-snes_mf` is specified, then we run matrix-free. Otherwise, we construct a Jacobian, either analytical if `jack` was passed non-null, or using PETSc's finite difference coloring method as used in SNES tutorial example 14. The finite difference part was put together based on PETSc example `src/snes/utlis/damgsnes.c` and I really don't understand it...

Then we run the timestep loop, which has three parts:

1. Use PETSc's SNES solver to calculate the next timestep.
2. Check convergence and, if it failed and `-noconv_cutttime` is set, then back up with a smaller timestep and restore previous timestep as first guess for next.
3. Otherwise copy the new result vector and function into the old, update the time, and call the monitor.

### H.2.3 Global Function `ts_implicit_function()`

This is the SNES callback for the constrained implicit timestepper `implicit_steptime()`.

```
int ts_implicit_function (SNES thesnes, Vec unpr, Vec snesF, void* user)
```

- `int ts_implicit_function`            It returns zero or error code.
- `SNES thesnes`                        SNES object in question.
- `Vec unpr`                              Field variable values.
- `Vec snesF`                             Where to return the function value.
- `void* user`                            User parameters' structure.

### H.2.4 Global Function `ts_implicit_jacobian()`

This is the SNES Jacobian callback for the constrained implicit timestepper `implicit_steptime()`.

```
int ts_implicit_jacobian (SNES thesnes, Vec unpr, Mat* jack, Mat* preck, MatStructure* flag, void* user)
```

- `int ts_implicit_jacobian`            It returns zero or error code.
- `SNES thesnes`                        SNES object in question.
- `Vec unpr`                              Field variable values.
- `Mat* jack`                             Jacobian matrix.
- `Mat* preck`                           Preconditioner matrix.
- `MatStructure* flag`                  Flag indicating preconditioner structure.
- `void* user`                            User parameters' structure.

`Vec unpr` Field variable values.

### H.2.5 Local Function FuncEvaluate()

This evaluates the set of functions which make up the time derivatives and constraints to evaluate. It does its own localization of the global vector, etc. It assumes that temp\_fields is already set up, as are all of the corner parameters (xs, ys, xm, gxs, etc.).

```
static int FuncEvaluate (Vec u, Vec u_local, Vec Fu, Vec Fu_local, PetscScalar time, void* user)
```

- `int FuncEvaluate` It returns zero or error code.
- `Vec u` Field variable values.
- `Vec u_local` Local vector to copy the field variables into.
- `Vec Fu` Global vector to put the function values into.
- `Vec Fu_local` Local vector for temporary function value storage.
- `PetscScalar time` Current simulation time.
- `void* user` User parameters' structure.

### H.2.6 Local Function xmax\_symm()

This takes a local array which is oversized (since we told PETSc we're using periodic BCs) and replaces the outer edges with mirror reflections across the lines just past the outer columns. It has per-dof options to put the symmetry lines at the outer columns/planes and set certain dofs to zero at or just beyond the outer columns (in rheoplast these options are set in the modules).

```
static void xmax_symm (PetscScalar* localarray, int xs, int ys, int zs, int xm, int ym, int zm, int gxs, int gys, int gzs, int gxm, int gym, int gzm, int nx, int* symmtypes, int dof)
```

- `PetscScalar* localarray` "Local" array with shadow nodes provided by the PETSc distributed array.
- `int xs` Starting global  $x$ -coordinate of the interior region.
- `int ys` Starting global  $y$ -coordinate of the interior region.
- `int zs` Starting global  $z$ -coordinate of the interior region.
- `int xm` Width of the interior region in the  $x$ -direction.
- `int ym` Width of the interior region in the  $y$ -direction.
- `int zm` Width of the interior region in the  $z$ -direction.
- `int gxs` Starting global  $x$ -coordinate of the local array.
- `int gys` Starting global  $y$ -coordinate of the local array.
- `int gzs` Starting global  $z$ -coordinate of the local array.
- `int gxm` Full width of the local array in the  $x$ -direction.
- `int gym` Full width of the local array in the  $y$ -direction.
- `int gzm` Full width of the local array in the  $z$ -direction.
- `int nx` Overall width of the entire distributed array in the  $x$ -direction.
- `int* symmtypes` Array of symmetry types for each dof (if NULL then assumes all are MIRROR\_PLANE).
- `int dof` Number of degrees of freedom per node.

### H.2.7 Local Function xmin\_symm()

This takes a local array which is oversized (since we told PETSc we're using periodic BCs) and replaces the outer edges with mirror reflections across the lines just past the outer columns. It has per-dof options to put the symmetry lines at the outer columns/planes and set certain dofs to zero at or just beyond the outer columns (in rheoplast these options are set in the modules).

```
static void xmin_symm (PetscScalar* localarray, int xs, int ys, int zs, int xm, int ym, int zm,
int gxs, int gys, int gzs, int gxm, int gym, int gzm, int* symmtypes, int dof)
```

- `PetscScalar* localarray`            “Local” array with shadow nodes provided by the PETSc distributed array.
- `int xs`                                Starting global  $x$ -coordinate of the interior region.
- `int ys`                                Starting global  $y$ -coordinate of the interior region.
- `int zs`                                Starting global  $z$ -coordinate of the interior region.
- `int xm`                                Width of the interior region in the  $x$ -direction.
- `int ym`                                Width of the interior region in the  $y$ -direction.
- `int zm`                                Width of the interior region in the  $z$ -direction.
- `int gxs`                               Starting global  $x$ -coordinate of the local array.
- `int gys`                               Starting global  $y$ -coordinate of the local array.
- `int gzs`                               Starting global  $z$ -coordinate of the local array.
- `int gxm`                               Full width of the local array in the  $x$ -direction.
- `int gym`                               Full width of the local array in the  $y$ -direction.
- `int gzm`                               Full width of the local array in the  $z$ -direction.
- `int* symmtypes`                       Array of symmetry types for each dof (if NULL then assumes all are MIRROR.PLANE).
- `int dof`                                Number of degrees of freedom per node.

## H.2.8 Local Function `yymax_symm()`

This takes a local array which is oversized (since we told PETSc we’re using periodic BCs) and replaces the outer edges with mirror reflections across the lines just past the outer columns. It has per-dof options to put the symmetry lines at the outer columns/planes and set certain dofs to zero at or just beyond the outer columns (in rheoplast these options are set in the modules).

```
static void yymax_symm (PetscScalar* localarray, int xs, int ys, int zs, int xm, int ym, int zm,
int gxs, int gys, int gzs, int gxm, int gym, int gzm, int ny, int* symmtypes, int dof)
```

- `PetscScalar* localarray`            “Local” array with shadow nodes provided by the PETSc distributed array.
- `int xs`                                Starting global  $x$ -coordinate of the interior region.
- `int ys`                                Starting global  $y$ -coordinate of the interior region.
- `int zs`                                Starting global  $z$ -coordinate of the interior region.
- `int xm`                                Width of the interior region in the  $x$ -direction.
- `int ym`                                Width of the interior region in the  $y$ -direction.
- `int zm`                                Width of the interior region in the  $z$ -direction.
- `int gxs`                               Starting global  $x$ -coordinate of the local array.
- `int gys`                               Starting global  $y$ -coordinate of the local array.
- `int gzs`                               Starting global  $z$ -coordinate of the local array.
- `int gxm`                               Full width of the local array in the  $x$ -direction.
- `int gym`                               Full width of the local array in the  $y$ -direction.
- `int gzm`                               Full width of the local array in the  $z$ -direction.
- `int ny`                                Overall width of the entire distributed array in the  $y$ -direction.
- `int* symmtypes`                       Array of symmetry types for each dof (if NULL then assumes all are MIRROR.PLANE).
- `int dof`                                Number of degrees of freedom per node.

### H.2.9 Local Function `ymin_symm()`

This takes a local array which is oversized (since we told PETSc we're using periodic BCs) and replaces the outer edges with mirror reflections across the lines just past the outer columns. It has per-dof options to put the symmetry lines at the outer columns/planes and set certain dofs to zero at or just beyond the outer columns (in rheoplast these options are set in the modules).

```
static void ymin_symm (PetscScalar* localarray, int xs, int ys, int zs, int xm, int ym, int zm,
int gxs, int gys, int gzs, int gxm, int gym, int gzm, int* symmtypes, int dof)
```

- `PetscScalar* localarray`            “Local” array with shadow nodes provided by the PETSc distributed array.
- `int xs`                                Starting global  $x$ -coordinate of the interior region.
- `int ys`                                Starting global  $y$ -coordinate of the interior region.
- `int zs`                                Starting global  $z$ -coordinate of the interior region.
- `int xm`                                Width of the interior region in the  $x$ -direction.
- `int ym`                                Width of the interior region in the  $y$ -direction.
- `int zm`                                Width of the interior region in the  $z$ -direction.
- `int gxs`                               Starting global  $x$ -coordinate of the local array.
- `int gys`                               Starting global  $y$ -coordinate of the local array.
- `int gzs`                               Starting global  $z$ -coordinate of the local array.
- `int gxm`                               Full width of the local array in the  $x$ -direction.
- `int gym`                               Full width of the local array in the  $y$ -direction.
- `int gzm`                               Full width of the local array in the  $z$ -direction.
- `int* symmtypes`                       Array of symmetry types for each dof (if NULL then assumes all are MIRROR\_PLANE).
- `int dof`                               Number of degrees of freedom per node.

### H.2.10 Local Function `zmax_symm()`

This takes a local array which is oversized (since we told PETSc we're using periodic BCs) and replaces the outer edges with mirror reflections across the lines just past the outer columns. It has per-dof options to put the symmetry lines at the outer columns/planes and set certain dofs to zero at or just beyond the outer columns (in rheoplast these options are set in the modules).

```
static void zmax_symm (PetscScalar* localarray, int xs, int ys, int zs, int xm, int ym, int zm,
int gxs, int gys, int gzs, int gxm, int gym, int gzm, int nz, int* symmtypes, int dof)
```

- `PetscScalar* localarray`            “Local” array with shadow nodes provided by the PETSc distributed array.
- `int xs`                                Starting global  $x$ -coordinate of the interior region.
- `int ys`                                Starting global  $y$ -coordinate of the interior region.
- `int zs`                                Starting global  $z$ -coordinate of the interior region.
- `int xm`                                Width of the interior region in the  $x$ -direction.
- `int ym`                                Width of the interior region in the  $y$ -direction.
- `int zm`                                Width of the interior region in the  $z$ -direction.
- `int gxs`                               Starting global  $x$ -coordinate of the local array.
- `int gys`                               Starting global  $y$ -coordinate of the local array.
- `int gzs`                               Starting global  $z$ -coordinate of the local array.
- `int gxm`                               Full width of the local array in the  $x$ -direction.
- `int gym`                               Full width of the local array in the  $y$ -direction.

- `int gzm` Full width of the local array in the  $z$ -direction.
- `int nz` Overall width of the entire distributed array in the  $z$ -direction.
- `int* symmtypes` Array of symmetry types for each dof (if NULL then assumes all are MIRROR\_PLANE).
- `int dof` Number of degrees of freedom per node.

### H.2.11 Local Function `zmin_symm()`

This takes a local array which is oversized (since we told PETSc we're using periodic BCs) and replaces the outer edges with mirror reflections across the lines just past the outer columns. It has per-dof options to put the symmetry lines at the outer columns/planes and set certain dofs to zero at or just beyond the outer columns (in rheoplast these options are set in the modules).

```
static void zmin_symm (PetscScalar* localarray, int xs, int ys, int zs, int xm, int ym, int zm,
int gxs, int gys, int gzs, int gxm, int gym, int gzm, int* symmtypes, int dof)
```

- `PetscScalar* localarray` “Local” array with shadow nodes provided by the PETSc distributed array.
- `int xs` Starting global  $x$ -coordinate of the interior region.
- `int ys` Starting global  $y$ -coordinate of the interior region.
- `int zs` Starting global  $z$ -coordinate of the interior region.
- `int xm` Width of the interior region in the  $x$ -direction.
- `int ym` Width of the interior region in the  $y$ -direction.
- `int zm` Width of the interior region in the  $z$ -direction.
- `int gxs` Starting global  $x$ -coordinate of the local array.
- `int gys` Starting global  $y$ -coordinate of the local array.
- `int gzs` Starting global  $z$ -coordinate of the local array.
- `int gxm` Full width of the local array in the  $x$ -direction.
- `int gym` Full width of the local array in the  $y$ -direction.
- `int gzm` Full width of the local array in the  $z$ -direction.
- `int* symmtypes` Array of symmetry types for each dof (if NULL then assumes all are MIRROR\_PLANE).
- `int dof` Number of degrees of freedom per node.

## I File `timestep.h`

### Included Files

```
#include </usr/lib/petsc/include/petscsnes.h>
#include </usr/lib/petsc/include/petscda.h>
#include </usr/lib/petsc/include/petscblaslapack.h>
#include <stdlib.h>
```

### Preprocessor definitions

```
#define Timestep_H
#define CONSTRAINT_ONLY 0
#define Timestep_ONLY_VAR 0x100
#define TIME_CONST_BLEND_VAR 0x200
```

```

#define DPRINTF(fmt, args...)
#define TSPRINTF(fmt, args...)
#define NO_SYMMETRY 0x00
#define XMIN_SYMMETRY 0x01
#define XMAX_SYMMETRY 0x02
#define YMIN_SYMMETRY 0x04
#define YMAX_SYMMETRY 0x08
#define ZMIN_SYMMETRY 0x10
#define ZMAX_SYMMETRY 0x20
#define SYMMETRY_MIRROR_INSIDE 0
#define SYMMETRY_MIRROR_PLANE 1
#define SYMMETRY_MIRROR_OUTSIDE 2
#define SYMMETRY_ZERO_INSIDE 4
#define SYMMETRY_ZERO_PLANE 5
#define SYMMETRY_ZERO_OUTSIDE 6
#define SYMMETRY_XMIN_START 0x000001
#define SYMMETRY_XMAX_START 0x000010
#define SYMMETRY_YMIN_START 0x000100
#define SYMMETRY_YMAX_START 0x001000
#define SYMMETRY_ZMIN_START 0x010000
#define SYMMETRY_ZMAX_START 0x100000
#define SYMMETRY_XMIN_MASK 0x00000F
#define SYMMETRY_XMAX_MASK 0x0000F0
#define SYMMETRY_YMIN_MASK 0x000F00
#define SYMMETRY_YMAX_MASK 0x00F000
#define SYMMETRY_ZMIN_MASK 0x0F0000
#define SYMMETRY_ZMAX_MASK 0xF00000

```

## I.1 Type definitions

### I.1.1 Typedef CommStyle

```

typedef enum {...} CommStyle
enum
{
 SLOW_COMMUNICATION;
 FAST_COMMUNICATION;
}

```

### I.1.2 Typedef EqStyle

```

typedef int EqStyle

```

## J File cahnhill.c

RCS Header: /cvsroot/rheoplast/cahnhill.c,v 1.37 2004/08/18 21:19:52 hazelsct Exp

This provides a simple Cahn-Hilliard module for Rheoplast. The free energy goes as

$$\mathcal{F} = \int \left( \beta \Psi(C) + \frac{\alpha}{2} |\nabla C|^2 \right) dV, \quad (1)$$

where  $\Psi(C)$  is the homogeneous free energy, given here as  $C^2(1-C)^2$ , or with the `-ch_polymer` option, a Flory-Huggins polymer solution free energy given by

$$\beta \Psi(C) = \frac{C}{m} \ln C + (1-C) \ln(1-C) + \chi C(1-C). \quad (2)$$

The chemical potential is then the variation of this free energy, given by

$$\mu = \frac{\delta \mathcal{F}}{\delta C} = \beta \Psi'(C) - \alpha \nabla^2 C. \quad (3)$$

### Included Files

```
#include "rheoplast.h" (Section G)
#include </usr/lib/petsc/include/petsc.h>
#include <stdlib.h>
#include <illuminator.h>
#include </usr/lib/petsc/include/petscts.h>
#include </usr/lib/petsc/include/petscblaslapack.h>
#include <stdlib.h>
#include <time.h>
#include <sys/times.h>
#include "timestep.h" (Section I)
#include </usr/lib/petsc/include/petscsnes.h>
#include </usr/lib/petsc/include/petscda.h>
#include </usr/lib/petsc/include/petscblaslapack.h>
#include <stdlib.h>
#include "cahnhill.h" (Section K)
#include </usr/lib/petsc/include/petsc.h>
#include "vortflow.h" (Section M)
#include </usr/lib/petsc/include/petsc.h>
#include "membrane.h" (Section O)
#include </usr/lib/petsc/include/petsc.h>
#include "cahnhill.h" (Section K)
#include "vortflow.h" (Section M)
#include "membrane.h" (Section O)
```

### Preprocessor definitions

```
#define _FUNCT_ "psiprime"
#define _FUNCT_ "psidoubleprime"
#define _FUNCT_ "cahnhill_first_setup"
#define _FUNCT_ "cahnhill_labels_initcond"
#define SMALL_PRIME 1571
#define MEDIUM_PRIME 524287
#define LARGE_PRIME 2147483647
```

```

#define MY_RANDOM(pseudo_random)
#define C(point)
#define Cfunc(point)
#define mu(point)
#define vu(point)
#define vv(point)
#define pu(point)
#define pv(point)
#define pw(point)
#define V(point)
#define sigma(point)
#define sigmaprime(point)
#define sigma_eff(point)
#define __FUNCT__ "cahnhill_temp_parameters_line"
#define __FUNCT__ "cahnhill_interior_line_function"
#define __FUNCT__ "cahnhill_interior_line_jacobian"

```

## J.1 Functions

### J.1.1 Global Function `cahnhill_first_setup()`

The basic setup, assigning the number of solved and temporary field variables, the stencil width, and using options to set the parameters in the `chparm` struct typedef.

```

void cahnhill_first_setup (PetscTruth threedee, int* vars, int* tempvars, int* stencilwid,
AppCtx* data)

```

- `PetscTruth threedee` Request support for 3-D.
- `int* vars` Pointer to the number of solved field variables.
- `int* tempvars` Pointer to the number of temporary field variables.
- `int* stencilwid` Pointer to the stencil width.
- `AppCtx* data` Pointer to the `AppCtx` struct typedef, into whose `chparm` structure this inserts parameters from the command line.

The standard model temporarily sets  $\alpha$  and  $\beta$  to  $\epsilon/\delta x$  and  $\sigma$ , and mobility to the dummy value -1, so that if not overridden by the user parameter setting, its default value of  $\epsilon^2$  can be set in `cahnhill_labels_initcond()`.

### J.1.2 Global Function `cahnhill_interior_line_function()`

This calculates the time derivative of  $C$  using the divergence of flux, which goes down the gradient of chemical potential given by equation 3. That time derivative is thus given by

$$\frac{\partial C}{\partial t} = \nabla \cdot (\kappa \nabla \mu). \quad (4)$$

where  $\kappa$  is the mobility.

```

void cahnhill_interior_line_function (PetscScalar* x, PetscScalar* func, PetscScalar* temp,
PetscTruth** mixed.constraints, int points, int gxm, int gym, PetscScalar xmin, PetscScalar xmax,
PetscScalar ycoord, PetscScalar zcoord, PetscScalar time, AppCtx* data)

```

|                                               |                                                                                                                                                     |
|-----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| · <code>PetscScalar* x</code>                 | The field variables from which to evaluate the function.                                                                                            |
| · <code>PetscScalar* func</code>              | Where to put the evaluated function.                                                                                                                |
| · <code>PetscScalar* temp</code>              | Array of temporary field variables.                                                                                                                 |
| · <code>PetscTruth** mixed_constraints</code> | Arrays of boolean variables indicating constraint equations in mixed timestep-constraint fields.                                                    |
| · <code>int points</code>                     | Number of points to evaluate at.                                                                                                                    |
| · <code>int gxm</code>                        | The $x$ -width of the “local” vector’s array, including shadow nodes, for the $y$ -increment.                                                       |
| · <code>int gym</code>                        | The $y$ -width of the “local” vector’s array, including shadow nodes, for the $z$ -increment.                                                       |
| · <code>PetscScalar xmin</code>               | First node $x$ -coordinate.                                                                                                                         |
| · <code>PetscScalar xmax</code>               | Last node plus one $x$ -coordinate.                                                                                                                 |
| · <code>PetscScalar ycoord</code>             | This line $y$ -coordinate.                                                                                                                          |
| · <code>PetscScalar zcoord</code>             | This line $z$ -coordinate.                                                                                                                          |
| · <code>PetscScalar time</code>               | Current simulation time.                                                                                                                            |
| · <code>AppCtx* data</code>                   | Pointer to the main simulation parameter structure, which includes the <code>chparam</code> struct typedef, from which this gets needed parameters. |

For now this assumes uniform  $\kappa$ .

It includes a convective term with first-order upwinding for velocity-vorticity flow.

### J.1.3 Global Function `cahnhill_interior_line_jacobian()`

This calculates the Jacobian of the equations corresponding to the Cahn-Hilliard variables.

```
void cahnhill_interior_line_jacobian (PetscScalar* x, PetscScalar* temp, Mat J, int points, int
gxm, int gym, int firstrow, PetscScalar xmin, PetscScalar xmax, PetscScalar ycoord, PetscScalar
zcoord, PetscScalar time, AppCtx* data)
```

|                                   |                                                                                                                                                     |
|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| · <code>PetscScalar* x</code>     | The field variables from which to evaluate the Jacobian.                                                                                            |
| · <code>PetscScalar* temp</code>  | Array of temporary field variables.                                                                                                                 |
| · <code>Mat J</code>              | Where to put the evaluated Jacobian.                                                                                                                |
| · <code>int points</code>         | Number of points to evaluate at.                                                                                                                    |
| · <code>int gxm</code>            | The $x$ -width of the “local” vector’s array, including shadow nodes, for the $y$ -increment.                                                       |
| · <code>int gym</code>            | The $y$ -width of the “local” vector’s array, including shadow nodes, for the $z$ -increment.                                                       |
| · <code>int firstrow</code>       | The matrix row number corresponding to the first point in the line.                                                                                 |
| · <code>PetscScalar xmin</code>   | First node $x$ -coordinate.                                                                                                                         |
| · <code>PetscScalar xmax</code>   | Last node plus one $x$ -coordinate.                                                                                                                 |
| · <code>PetscScalar ycoord</code> | This line $y$ -coordinate.                                                                                                                          |
| · <code>PetscScalar zcoord</code> | This line $z$ -coordinate.                                                                                                                          |
| · <code>PetscScalar time</code>   | Current simulation time.                                                                                                                            |
| · <code>AppCtx* data</code>       | Pointer to the main simulation parameter structure, which includes the <code>chparam</code> struct typedef, from which this gets needed parameters. |

First this calculates the fixed coefficients in the  $-\kappa\alpha\nabla^2\nabla^2C$  term of the transport equation. The variable `jvalue` will hold the Jacobian values for insertion into the matrix. These are fixed in eight of the thirteen non-zeroes (18 of the 25 in 3-D); the other five (seven in 3-D) depend on  $\Psi''(C)$ , as discussed in section J.1.6 (page 37). The fixed Jacobian values are stored right away in the `jvalue` array, and fixed parts of the

$\Psi''(C)$ -dependent Jacobian values are temporarily stored in the `fvalue` array for subsequent insertion into `jvalue` and addition to the variable parts.

#### J.1.4 Global Function `cahnhill_labels_initcond()`

This sets up the field variable labels, maximum stable explicit `deltat`, and initial condition for the Cahn-Hilliard variables.

```
void cahnhill_labels_initcond (PetscScalar* globalarray, int nx, int ny, int nz, int xm, int ym,
int zm, int xs, int ys, int zs, int vars, AppCtx* data, PetscScalar* max_explicit_deltat)
```

- `PetscScalar* globalarray`            The global field array.
- `int nx`                                Overall  $x$ -width of the global array.
- `int ny`                                Overall  $y$ -width of the global array.
- `int nz`                                Overall  $z$ -width of the global array.
- `int xm`                                The  $x$ -width of the local part of the array.
- `int ym`                                The  $y$ -width of the local part of the array.
- `int zm`                                The  $z$ -width of the local part of the array.
- `int xs`                                The (integer)  $x$ -coordinate of the start of the local part of the array.
- `int ys`                                The (integer)  $y$ -coordinate of the start of the local part of the array.
- `int zs`                                The (integer)  $z$ -coordinate of the start of the local part of the array.
- `int vars`                              Total number of field variables to be solved.
- `AppCtx* data`                        Pointer to the `AppCtx` struct typedef, whose `chparm` structure this uses for various purposes.
- `PetscScalar* max_explicit_deltat` Pointer to the largest allowable explicit timestep size for this equation, which this function can set/modify.

The standard model sets the interface thickness to thrice the minimum grid spacing and surface tension to 1 by default. These are controlled by command line options, either `-ch_intwidth` and `-ch_surftens` (`intwidth` is multiplied by  $\Delta x$ ), or by setting the homogeneous and gradient penalty coefficients themselves using `-ch_alpha` and `-ch_beta`. The constants in the code (3.1 and square root of 18) get the surface tension and interface thickness correct for the standard model; different values are needed to get the polymer model correct (though arguably, in the polymer model one should set `beta` to 1 and use `alpha` to adjust the interface thickness).

By default,  $\kappa$  is set to  $\epsilon^2/\beta$ , such that the diffusion timescale over the interface thickness is constant; this is controlled by command line option `-ch_mobility`.

Default polymer Flory-Huggins parameters are  $\chi = 0.58$  and  $m = 640$ , as discussed in appendix J.1.7, page 38.

The explicit finite difference timestep size used here is  $(\Delta x)^3/40\kappa$ , which works for the fourth-order polynomial free energy in 2-D when `nx=ny`,  $\epsilon = \Delta x$  (`intwidth=1`), and  $\sigma = 1$ .

For random fluctuations in this module, it's necessary to generate different random sequences on each process, even though `rand()` will return the same sequences. So we create a random counter which takes on values between zero and `SMALL_PRIME-1`; this is initialized to `rank` times `LARGE_PRIME` (which should make it sort of random), and incremented by `MEDIUM_PRIME` then re-moduloed to `SMALL_PRIME` for each random number generated. This counter, in turn, is divided by `SMALL_PRIME` and the result added to `rand()/RAND_MAX` then modulo 1, in order to give a "different random number" (at least at the resolution of `SMALL_PRIME`) in each process. This is not a great algorithm, but should do for the purpose needed here.

Eventually it might be good to make this resource available to the whole code.

If we're not loading in data as the initial condition, the  $C$  field is initiated here. There are several types of initial conditions here which are chosen by command-line parameters:

- Small random fluctuations are selected using the `-ch_random_center` and `-ch_random_fluct` options. These produce a uniform distribution centered at the “center” value and with width twice the fluctuation value.
- A two-layer initial condition in the  $y$ -direction is chosen using the `-ch_layers` option, whose argument is the fraction of the domain which will be in the “bottom”  $C = 1$  region.
- The `-ch_trilayer` option is used for the “metal-electrolyte-metal” simulation. This creates three layers in the  $y$ -direction with width half that of the domain, which is to say, one-quarter of that of the whole symmetric domain.
- The `-ch_particles` option is used for the “colliding particles” simulation. This creates a symmetric simulation with a square particle centered on the middle of the  $x$ -axis with width half that of the domain, which is to say, one-quarter of that of the whole symmetric domain.
- The default initial condition is a square (cube) with side length equal to half of the domain width, either in the center, or if all symmetries are on, then at the origin.

If the option `-ch_random_fluct` is selected without `-ch_random_center`, then random fluctuations with uniform distribution of width twice the fluctuation value are added to any other initial condition present.

### J.1.5 Global Function `cahnhill_temp_parameters_line()`

This calculates the Cahn-Hilliard temporary parameter  $\mu$ , which is the chemical potential given by equation 3.

```
void cahnhill_temp_parameters_line (PetscScalar* x, PetscScalar* temp, int points, int gxm, int gym, PetscScalar xmin, PetscScalar xmax, PetscScalar ycoord, PetscScalar zcoord, PetscScalar time, AppCtx* data)
```

|                                   |                                                                                                                                                     |
|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| · <code>PetscScalar* x</code>     | Array with the “real” field variables.                                                                                                              |
| · <code>PetscScalar* temp</code>  | Array with the temporary field variables.                                                                                                           |
| · <code>int points</code>         | Number of points at which to calculate the temporary variables.                                                                                     |
| · <code>int gxm</code>            | The $x$ -width of the “local” vector’s array, including shadow nodes, for the $y$ -increment.                                                       |
| · <code>int gym</code>            | The $y$ -width of the “local” vector’s array, including shadow nodes, for the $z$ -increment.                                                       |
| · <code>PetscScalar xmin</code>   | First node $x$ -coordinate.                                                                                                                         |
| · <code>PetscScalar xmax</code>   | Last node plus one $x$ -coordinate.                                                                                                                 |
| · <code>PetscScalar ycoord</code> | This line $y$ -coordinate.                                                                                                                          |
| · <code>PetscScalar zcoord</code> | This line $z$ -coordinate                                                                                                                           |
| · <code>PetscScalar time</code>   | Current simulation time.                                                                                                                            |
| · <code>AppCtx* data</code>       | Pointer to the main simulation parameter structure, which includes the <code>chparam</code> struct typedef, from which this gets needed parameters. |

### J.1.6 Local Function `psidoubleprime()`

This abstracts out the function for  $\Psi''(C)$ , the second derivative of homogeneous free energy, so it can be easily modified. Since  $\Psi(C) = C^2(1 - C)^2 = C^4 - 2C^3 + C^2$ , this returns  $12C^2 - 12C + 2$ .

There is also a polymer thermo option based on Flory-Huggins thermodynamics given in equation 2. Enable it using option `-ch_polymer` and control it using options `-ch_polymer_chi` and `-ch_polymer_m` (defaults: 0.58, 640). Note that with  $m = 1$  this reduces to the regular solution model.

```
static inline PetscScalar psidoubleprime (PetscScalar C, chparam* thecahnhill)
```

|                                           |                                                              |
|-------------------------------------------|--------------------------------------------------------------|
| · <code>PetscScalar psidoubleprime</code> | It returns the second derivative of homogeneous free energy. |
|-------------------------------------------|--------------------------------------------------------------|

- PetscScalar C                   The  $C$  parameter it's a function of.
- chparm\* thecahnhill            Cahn-Hilliard parameter structure.

### J.1.7 Local Function psiprime()

This abstracts out the function for  $\Psi'(C)$ , the derivative of homogeneous free energy, so it can be easily modified. Since  $\Psi(C) = C^2(1 - C)^2 = C^4 - 2C^3 + C^2$ , this returns  $4C^3 - 6C^2 + 2C$ .

There is also a polymer thermo option based on Flory-Huggins thermodynamics given in equation 2. Enable it using option `-ch_polymer` and control it using options `-ch_polymer_chi` and `-ch_polymer_m` (defaults: 0.58, 640). Note that with  $m = 1$  this reduces to the regular solution model.

```
static inline PetscScalar psiprime (PetscScalar C, chparm* thecahnhill)
```

- PetscScalar psiprime            It returns the derivative of homogeneous free energy.
- PetscScalar C                   The  $C$  parameter it's a function of.
- chparm\* thecahnhill            Cahn-Hilliard parameter structure.

## K File cahnhill.h

RCS Header: /cvsroot/rheoplast/cahnhill.h,v 1.5 2004/03/11 14:42:46 hazelsct Exp

The typedefs and prototypes for Cahn-Hilliard species transport.

### Included Files

```
#include </usr/lib/petsc/include/petsc.h>
```

### Preprocessor definitions

```
#define CAHNHILL.H
```

```
#define CAHNHILL.HELP "Cahn-Hilliard species transport is a basic staple of phase field modeling.\nTo use it, add option:\n -with-cahnhill\nand control it with the properties:\n -ch_intwidth <epsilon> interface thickness/dx [3.0]\n -ch_surftens <sigma> interface energy [1.0]\n -ch_mobility <kappa> mobility [epsilon^2]\nOne can also use a polymer solution model by selecting\n -ch_polymer and setting properties:\n -ch_polymer_chi <chi> interaction parameter [0.68]\n -ch_polymer_m <m> extent of polymerization [64.0]\nThe default initial condition is a centered square. If -symmetry_x and\n-symmetry_y are specified, this is a square centered at the origin. An\nalternate initial condition with a square (cube) centered on the middle of\nthe x-axis can be used (automatically turning on all symmetries) by\nspecifying:\n -ch_particles\nOr one can specify a two-layer system in the y-direction using:\n -ch_layers <thickness>\nwhere thickness is the fraction in the bottom C=1 layer, or\n -ch_trilayer\nfor a C=1, C=0, C=1 three-layer initial condition.\nOne can also use a random initial distribution to simulate spinodal\ndecomposition with:\n -ch_random_center <center> center of random distribution (required)\n -ch_random_fluct <fluct> half-width of uniform distribution [0.01]\n\n"
```

### K.1 Type definitions

#### K.1.1 Typedef AppCtx

```
typedef void AppCtx
```

### K.1.2 Typedef chparm

Structure typedef for Cahn-Hilliard species transport.

```
typedef struct {...} chparm
struct
{
 PetscTruth polymer_solution;
 PetscScalar mobility;
 PetscScalar alpha;
 PetscScalar beta;
 PetscScalar chi;
 PetscScalar m;
 int Cvar;
 int muvar;
}
```

## L File vortflow.c

RCS Header: /cvsroot/rheoplast/vortflow.c,v 1.65 2004/08/23 16:47:51 hazelsct Exp

This provides rheoplast with all of the functions for modeling fluid flow using the velocity-vorticity formulation (which seems easier than velocity-pressure).

The incompressible Navier-Stokes equations in velocity-pressure form are a pain in the neck to solve, because one must worry about spurious modes in the pressure. Sure, there are ways around it, like staggered meshes in finite difference and Taylor-Hood elements in finite elements. But there are alternate forms which don't require such tricks, including velocity-vorticity, which is particularly useful in this phase field code because the vorticity gives the rotation rate for the order parameter vector and the elastic strain tensor.

For the velocity-vorticity formulation in cartesian coordinates, our variables will be  $u$  and  $v$  for  $x$ - and  $y$ -direction velocities, and  $\omega$  for vorticity defined by

$$\omega = \nabla \times \vec{u} = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}. \quad (5)$$

The incompressible Navier-Stokes equations start with continuity:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0. \quad (6)$$

If we differentiate that with respect to  $x$ , that becomes equivalent to

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 v}{\partial x \partial y} - \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial y^2} = 0. \quad (7)$$

The two middle terms are  $\partial \omega / \partial y$ , so we can rewrite this as

$$\nabla^2 u + \frac{\partial \omega}{\partial y} = 0. \quad (8)$$

We can also differentiate equation 6 with respect to  $y$ , and through a similar manipulation end up with

$$\nabla^2 v - \frac{\partial \omega}{\partial x} = 0. \quad (9)$$

Next we turn to the incompressible equations of motion:

$$\rho \left( \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} \right) = -\frac{\partial p}{\partial x} + \frac{\partial}{\partial x} \left( \eta \frac{\partial u}{\partial x} \right) + \frac{\partial}{\partial y} \left( \eta \frac{\partial u}{\partial y} \right) + F_x, \quad (10)$$

$$\rho \left( \frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} \right) = -\frac{\partial p}{\partial y} + \frac{\partial}{\partial x} \left( \eta \frac{\partial v}{\partial x} \right) + \frac{\partial}{\partial y} \left( \eta \frac{\partial v}{\partial y} \right) + F_y. \quad (11)$$

Now we just subtract the  $y$ -derivative of equation 10 from the  $x$ -derivative of equation 11. The left side gives:

$$\rho \left( \frac{\partial^2 v}{\partial x \partial t} + \frac{\partial u}{\partial x} \frac{\partial v}{\partial x} + u \frac{\partial^2 v}{\partial x^2} + \frac{\partial v}{\partial x} \frac{\partial v}{\partial y} + v \frac{\partial^2 v}{\partial x \partial y} \right) + \frac{\partial \rho}{\partial x} \left( \frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} \right) \quad (12)$$

$$- \rho \left( \frac{\partial^2 u}{\partial y \partial t} + \frac{\partial u}{\partial y} \frac{\partial u}{\partial x} + u \frac{\partial^2 u}{\partial x \partial y} + \frac{\partial v}{\partial y} \frac{\partial u}{\partial y} + v \frac{\partial^2 u}{\partial y^2} \right) - \frac{\partial \rho}{\partial y} \left( \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} \right) \\ = \rho \left( \frac{\partial \omega}{\partial t} + \omega \frac{\partial u}{\partial x} + u \frac{\partial \omega}{\partial x} + \omega \frac{\partial v}{\partial y} + v \frac{\partial \omega}{\partial y} \right) + \nabla \rho \times \left( \frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} \right). \quad (13)$$

where  $\vec{u}$  represents the velocity vector  $(u, v)$ . The right side is a bit simpler:

$$- \frac{\partial^2 p}{\partial x \partial y} + \frac{\partial^2}{\partial x^2} \left( \eta \frac{\partial v}{\partial x} \right) + \frac{\partial^2}{\partial x \partial y} \left( \eta \frac{\partial v}{\partial y} \right) + \frac{\partial F_y}{\partial x} \quad (14)$$

$$- \frac{\partial^2 p}{\partial y \partial x} - \frac{\partial^2}{\partial y \partial x} \left( \eta \frac{\partial u}{\partial x} \right) - \frac{\partial^2}{\partial y^2} \left( \eta \frac{\partial u}{\partial y} \right) - \frac{\partial F_x}{\partial y} \\ = 0 + \eta \nabla^2 \omega + \omega \nabla^2 \eta + \frac{\partial^2 \eta}{\partial x \partial y} \left( \frac{\partial v}{\partial y} - \frac{\partial u}{\partial x} \right) + \frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y}. \quad (15)$$

So when we put equations 13 and 15 together, we get:

$$\rho \left( \frac{\partial \omega}{\partial t} + \nabla \cdot (\omega \vec{u}) \right) + \nabla \rho \times \left( \frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} \right) = \eta \nabla^2 \omega + \omega \nabla^2 \eta + \frac{\partial^2 \eta}{\partial x \partial y} \left( \frac{\partial v}{\partial y} - \frac{\partial u}{\partial x} \right) + \nabla \times \vec{F} \quad (16)$$

and for a uniform-density uniform-viscosity fluid, this simplifies to

$$\frac{\partial \omega}{\partial t} + \vec{u} \cdot \nabla \omega = \nu \nabla^2 \omega + \frac{\nabla \times \vec{F}}{\rho}. \quad (17)$$

Equations 8, 9, and either 16 or 17 comprise the velocity-vorticity form of the incompressible Navier-Stokes equations. It is interesting to note that this form consists of two equations of continuity and one of motion.

The velocity-vorticity formulation is used here in **RheoPlast** for a couple of reasons. It has the numerical advantage of no zeroes on the diagonal, unlike velocity-pressure whose "pressure equation", which is zero divergence of velocity, has no pressure in it. Also unlike velocity-pressure, it has no spurious modes in difference equations with all of the variables computed at each node, so we don't need a staggered mesh for stability (though we do need a staggered mesh for shear strain when that is included). Finally, the velocity and vorticity happen to be just the parameters needed for the convective and rotational terms in the vector-valued phase field and shear strain tensor field equations.

It remains to be seen whether these advantages will be sufficient in three dimensions to justify the additional two fields and equations required for the three components of the vorticity vector field, vs. the scalar pressure field. If memory is not a problem, that should be the case.

## Included Files

```
#include "rheoplast.h" (Section G)
#include </usr/lib/petsc/include/petsc.h>
#include <stdlib.h>
#include <illuminator.h>
#include </usr/lib/petsc/include/petscts.h>
#include </usr/lib/petsc/include/petscblaslapack.h>
#include <stdlib.h>
#include <time.h>
#include <sys/times.h>
#include "timestep.h" (Section I)
#include </usr/lib/petsc/include/petscsnes.h>
#include </usr/lib/petsc/include/petscda.h>
```

```

#include </usr/lib/petsc/include/petscblaslapack.h>
#include <stdlib.h>
#include "cahnhill.h" (Section K)
#include </usr/lib/petsc/include/petsc.h>
#include "vortflow.h" (Section M)
#include </usr/lib/petsc/include/petsc.h>
#include "membrane.h" (Section O)
#include </usr/lib/petsc/include/petsc.h>
#include "cahnhill.h" (Section K)
#include "vortflow.h" (Section M)
#include "membrane.h" (Section O)

```

## Preprocessor definitions

```

#define __FUNCT__ "vortflow_first_setup"
#define __FUNCT__ "vortflow_labels_initcond"
#define __FUNCT__ "vortflow_temp_parameters_line"
#define u(point)
#define v(point)
#define omega(point)
#define ufunc(point)
#define vfunc(point)
#define omegafunc(point)
#define phi(point)
#define gxx(point)
#define gxy(point)
#define C(point)
#define mu(point)
#define phi2(point)
#define phi3(point)
#define mu2(point)
#define mu3(point)
#define __FUNCT__ "vortflow_interior_line_function"

```

## L.1 Functions

### L.1.1 Global Function `vortflow_first_setup()`

The basic setup, setting the number of solved and temporary field variables and the stencil width.

```
void vortflow_first_setup (PetscTruth threedee, int* vars, int* tempvars, int* stencilwid,
AppCtx* data)
```

- `PetscTruth threedee` Request support for 3-D.
- `int* vars` Pointer to the number of solved field variables.
- `int* tempvars` Pointer to the number of temporary field variables.
- `int* stencilwid` Pointer to the stencil width.
- `AppCtx* data` Pointer to the `AppCtx` struct typedef, into whose `vortparm` structure this inserts parameters from the command line.

### L.1.2 Global Function `vortflow_interior_line_function()`

This calculates the time derivatives and constraint functions for the velocity-vorticity form of the Navier-Stokes equations for an interior line. Note the `shearstrain` module interaction programmed here;

```
void vortflow_interior_line_function (PetscScalar* x, PetscScalar* func, PetscScalar* temp,
PetscTruth** mixed_constraints, int points, int gxm, int gym, PetscScalar xmin, PetscScalar xmax,
PetscScalar ycoord, PetscScalar zcoord, PetscScalar time, AppCtx* data)
```

- `PetscScalar* x`                    The field variables from which to evaluate the function.
- `PetscScalar* func`                Where to put the evaluated function.
- `PetscScalar* temp`                Array of temporary field variables.
- `PetscTruth** mixed_constraints`   Arrays of boolean variables indicating constraint equations in mixed timestep-constraint fields.
- `int points`                        Number of points to evaluate at.
- `int gxm`                            The  $x$ -width of the “local” vector’s array, including shadow nodes, for the  $y$ -increment.
- `int gym`                            The  $y$ -width of the “local” vector’s array, including shadow nodes, for the  $z$ -increment.
- `PetscScalar xmin`                 First node  $x$ -coordinate.
- `PetscScalar xmax`                 Last node plus one  $x$ -coordinate.
- `PetscScalar ycoord`               This line  $y$ -coordinate.
- `PetscScalar zcoord`               This line  $z$ -coordinate.
- `PetscScalar time`                 Current simulation time.
- `AppCtx* data`                     Pointer to the main simulation parameter structure, which includes the `vortparm` struct typedef, from which this gets needed parameters.

The  $u$  equation is the constraint in equation 8 above, divided by `deltax_m2` to bring the Jacobian in line for small `deltax`.

Likewise, the  $v$  equation is the constraint in equation 9 above, divided by `deltay_m2` to bring the Jacobian in line for small `deltax`.

The vorticity equation is the time-derivative in equation 17 above. Here it is implemented in three parts: the convective terms, the stress terms (viscous and, if `shearstrain` is present, elastic), and the body force terms, including interface curvature from the Cahn-Hilliard module.

For Cahn-Hilliard, we add the body force due to interface curvature as described by Jacqmin \cite{jacqderive}:  $-C\nabla\mu$ , whose curl is  $-\nabla \times (C\nabla\mu)$ .

For the membrane, since it’s basically Cahn-Hilliard, we add a similar curvature body force given by  $-\sum \phi_i \nabla \mu_i$ , whose curl is  $-\sum \nabla \times (\phi_i \nabla \mu_i)$ .

I’m adding a  $x$ -driving force sinusoidal in  $y$  and starting time  $t = t_0$  such that it is zero during any initial explicit timesteps. A force amplitude of  $4\pi^2$  (meaning force curl amplitude of  $8\pi^3$ ) gives a velocity amplitude of one, which should be good for what we’re looking for. The `-sineforcet0` and `-sineforcemax` command-line switches control the force starting time and amplitude respectively, and the latter is normalized by density in the function `vortflow_labels_initcond()` (section L.1.3, page 42).

### L.1.3 Global Function `vortflow_labels_initcond()`

This uses options to set the parameters in the `vortparm` struct typedef, sets up the field variable labels, maximum stable explicit `deltat`, and initial condition for the velocity-vorticity variables.

```
void vortflow_labels_initcond (PetscScalar* globalarray, int nx, int ny, int nz, int xm, int ym,
int zm, int xs, int ys, int zs, int vars, AppCtx* data, PetscScalar* max_explicit_deltat)
```

- `PetscScalar* globalarray`        The global field array.
- `int nx`                             Overall  $x$ -width of the global array.

- `int ny` Overall  $y$ -width of the global array.
- `int nz` Overall  $z$ -width of the global array.
- `int xm` The  $x$ -width of the local part of the array.
- `int ym` The  $y$ -width of the local part of the array.
- `int zm` The  $z$ -width of the local part of the array.
- `int xs` The (integer)  $x$ -coordinate of the start of the local part of the array.
- `int ys` The (integer)  $y$ -coordinate of the start of the local part of the array.
- `int zs` The (integer)  $z$ -coordinate of the start of the local part of the array.
- `int vars` Total number of field variables to be solved.
- `AppCtx* data` Pointer to the `AppCtx` struct typedef, whose `vortparm` structure this uses for various purposes.
- `PetscScalar* max_explicit_deltat` Pointer to the largest allowable explicit timestep size for this equation, which this function can set/modify.

#### L.1.4 Global Function `vortflow_temp_parameters_line()`

There are no temporary field variables for velocity-vorticity flow, so this does nothing.

```
void vortflow_temp_parameters_line (PetscScalar* x, PetscScalar* temp, int points, int gxm, int
gym, PetscScalar xmin, PetscScalar xmax, PetscScalar ycoord, PetscScalar zcoord, PetscScalar time,
AppCtx* data)
```

- `PetscScalar* x` Array with the "real" field variables.
- `PetscScalar* temp` Array with the temporary field variables.
- `int points` Number of points at which to calculate the temporary variables.
- `int gxm` The  $x$ -width of the "local" vector's array, including shadow nodes, for the  $y$ -increment.
- `int gym` The  $y$ -width of the "local" vector's array, including shadow nodes, for the  $z$ -increment.
- `PetscScalar xmin` First node  $x$ -coordinate.
- `PetscScalar xmax` Last node plus one  $x$ -coordinate.
- `PetscScalar ycoord` This line  $y$ -coordinate.
- `PetscScalar zcoord` This line  $z$ -coordinate.
- `PetscScalar time` Current simulation time.
- `AppCtx* data` Pointer to the main simulation parameter structure, which includes the `vortparm` struct typedef, from which this gets needed parameters.

## M File `vortflow.h`

**RCS Header:** `/cvsroot/rheoplast/vortflow.h,v 1.21 2004/03/14 16:20:18 hazelsct Exp`

The typedefs and prototypes for velocity-vorticity fluid flow. This is not meant to be included on its own, only when someone `#includes "rheoplast.h"`.

### Included Files

```
#include </usr/lib/petsc/include/petsc.h>
```

### Preprocessor definitions

```
#define VORTFLOW_H
```

```
#define VORTFLOW_HELP "Velocity-vorticity Navier-Stokes is really cool, see the
vortflow.c\ndocumentation for a complete description. To use it, add option:\n -with-vortflow\nand
control it with properties and body force parameters:\n -viscosity <eta> viscosity [1.0]\n
-density <rho> density [1.0]\n -sineforcet0 <Ft0> Sinusoidal force onset time [2.0]\n
-sineforcemax <Fmax> Sinusoidal force amplitude [1.0]\nIf -symmetry_x and -symmetry_y are
specified, then one may also specify\ninitial and boundary conditions describing stagnation flow
using:\n -stagnation_flow <umax> Maximum stagnation velocity [1.0]\n\n"
```

## M.1 Type definitions

### M.1.1 Typedef AppCtx

```
typedef void AppCtx
```

### M.1.2 Typedef vortparm

Structure typedef for velocity-vorticity fluid flow.

```
typedef struct {...} vortparm
struct
{
 PetscScalar viscosity;
 PetscScalar density;
 PetscScalar sineFt0;
 PetscScalar sineFmax;
 int uvar;
 int vvar;
 int omegavar;
}
```

## N File membrane.c

**RCS Header: /cvsroot/rheoplast/membrane.c,v 1.38 2004/08/18 21:19:52 hazelsct Exp**

This provides a simple nonsolvent/solvent/polymer membrane module for Rheoplast. It goes as

$$\mathcal{F} = \int \left( f(\phi_2, \phi_3) + \frac{1}{2} \sum_{i,j=2,3} K_{ij} \nabla \phi_i \cdot \nabla \phi_j \right) dV, \quad (18)$$

where  $f(\phi_2, \phi_3)$  is the homogeneous free energy, given as  $\frac{\phi_1}{m_1} * \ln(\phi_1) + \frac{\phi_2}{m_2} * \ln(\phi_2) + \frac{\phi_3}{m_3} * \ln(\phi_3) + \chi_{12} * \phi_1 * \phi_2 + \chi_{23} * \phi_2 * \phi_3 + \chi_{13} * \phi_1 * \phi_3$ . Here, subscript 1 represents the nonsolvent, 2 represents the solvent and 3 represents the polymer.

### Included Files

```
#include "rheoplast.h" (Section G)
#include </usr/lib/petsc/include/petsc.h>
#include <stdlib.h>
#include <illuminator.h>
#include </usr/lib/petsc/include/petscts.h>
#include </usr/lib/petsc/include/petscblaslapack.h>
#include <stdlib.h>
#include <time.h>
#include <sys/times.h>
#include "timestep.h" (Section I)
#include </usr/lib/petsc/include/petscsnes.h>
#include </usr/lib/petsc/include/petscda.h>
```

```

#include </usr/lib/petsc/include/petscblaslapack.h>
#include <stdlib.h>
#include "cahnhill.h" (Section K)
#include </usr/lib/petsc/include/petsc.h>
#include "vortflow.h" (Section M)
#include </usr/lib/petsc/include/petsc.h>
#include "membrane.h" (Section O)
#include </usr/lib/petsc/include/petsc.h>
#include "cahnhill.h" (Section K)
#include "vortflow.h" (Section M)
#include "membrane.h" (Section O)

```

## Preprocessor definitions

```

#define __FUNCT__ "psiprime2"
#define __FUNCT__ "psiprime3"
#define __FUNCT__ "psidoubleprime2"
#define __FUNCT__ "psidoubleprime3"
#define __FUNCT__ "M33"
#define __FUNCT__ "membrane_first_setup"
#define __FUNCT__ "membrane_labels_initcond"
#define SMALL_PRIME 1571
#define MEDIUM_PRIME 524287
#define LARGE_PRIME 2147483647
#define MY_RANDOM(pseudo_random)
#define phi2(point)
#define phi3(point)
#define phi2func(point)
#define phi3func(point)
#define mu2(point)
#define mu3(point)
#define __FUNCT__ "membrane_temp_parameters_line"
#define __FUNCT__ "membrane_interior_line_function"
#define u(point)
#define v(point)

```

## N.1 Functions

### N.1.1 Global Function `membrane_first_setup()`

The basic setup, assigning the number of solved and temporary field variables, the stencil width, and using options to set the parameters in the `mparm` struct typedef.

```
void membrane_first_setup (PetscTruth threedee, int* vars, int* tempvars, int* stencilwid,
AppCtx* data)
```

· `PetscTruth threedee` Request support for 3-D.

- `int* vars` Pointer to the number of solved field variables.
- `int* tempvars` Pointer to the number of temporary field variables.
- `int* stencilwid` Pointer to the stencil width.
- `AppCtx* data` Pointer to the `AppCtx` struct typedef, into whose `mparm` structure this inserts parameters from the command line.

This sets mobility  $M_{ij}$  and gradient penalties  $K_{ij}$  for use in `membrane_labels_initcond()`.

### N.1.2 Global Function `membrane_interior_line_function()`

This calculates the time derivative of  $\phi_2, \phi_3$  using the divergence of flux, which goes down the gradient of chemical potential given by equation 20. That time derivative is thus given by

$$\frac{\partial \phi_i}{\partial t} = \nabla \cdot (M_{ij} \nabla \mu_j). \quad (19)$$

where  $M_{ij}$  is the mobility of species  $i$  due to a gradient in species  $j$ .

```
void membrane_interior_line_function (PetscScalar* x, PetscScalar* func, PetscScalar* temp,
PetscTruth** mixed_constraints, int points, int gxm, int gym, PetscScalar xmin, PetscScalar xmax,
PetscScalar ycoord, PetscScalar zcoord, PetscScalar time, AppCtx* data)
```

- `PetscScalar* x` The field variables from which to evaluate the function.
- `PetscScalar* func` Where to put the evaluated function.
- `PetscScalar* temp` Array of temporary field variables.
- `PetscTruth** mixed_constraints` Arrays of boolean variables indicating constraint equations in mixed timestep-constraint fields.
- `int points` Number of points to evaluate at.
- `int gxm` The  $x$ -width of the “local” vector’s array, including shadow nodes, for the  $y$ -increment.
- `int gym` The  $y$ -width of the “local” vector’s array, including shadow nodes, for the  $z$ -increment.
- `PetscScalar xmin` First node  $x$ -coordinate.
- `PetscScalar xmax` Last node plus one  $x$ -coordinate.
- `PetscScalar ycoord` This line  $y$ -coordinate.
- `PetscScalar zcoord` This line  $z$ -coordinate.
- `PetscScalar time` Current simulation time.
- `AppCtx* data` Pointer to the main simulation parameter structure, which includes the `mparm` struct typedef, from which this gets needed parameters.

For now this assumes uniform  $M_{ij}$ .

### N.1.3 Global Function `membrane_labels_initcond()`

This sets up the field variable labels, maximum stable explicit `deltat`, and initial condition for the Cahn-Hilliard variables.

```
void membrane_labels_initcond (PetscScalar* globalarray, int nx, int ny, int nz, int xm, int ym,
int zm, int xs, int ys, int zs, int vars, AppCtx* data, PetscScalar* max_explicit_deltat)
```

- `PetscScalar* globalarray` The global field array.
- `int nx` Overall  $x$ -width of the global array.
- `int ny` Overall  $y$ -width of the global array.
- `int nz` Overall  $z$ -width of the global array.

- `int xm` The  $x$ -width of the local part of the array.
- `int ym` The  $y$ -width of the local part of the array.
- `int zm` The  $z$ -width of the local part of the array.
- `int xs` The (integer)  $x$ -coordinate of the start of the local part of the array.
- `int ys` The (integer)  $y$ -coordinate of the start of the local part of the array.
- `int zs` The (integer)  $z$ -coordinate of the start of the local part of the array.
- `int vars` Total number of field variables to be solved.
- `AppCtx* data` Pointer to the `AppCtx` struct typedef, whose `mparm` structure this uses for various purposes.
- `PetscScalar* max_explicit_deltat` Pointer to the largest allowable explicit timestep size for this equation, which this function can set/modify.

For random fluctuations in this module, it's necessary to generate different random sequences on each process, even though `rand()` will return the same sequences. So we create a random counter which takes on values between zero and `SMALL_PRIME-1`; this is initialized to rank times `LARGE_PRIME` (which should make it sort of random), and incremented by `MEDIUM_PRIME` then re-moduloed to `SMALL_PRIME` for each random number generated. This counter, in turn, is divided by `SMALL_PRIME` and the result added to `rand()/RAND_MAX` then modulo 1, in order to give a "different random number" (at least at the resolution of `SMALL_PRIME`) in each process. This is not a great algorithm, but should do for the purpose needed here.

Eventually it might be good to make this resource available to the whole code.

If we're not loading in data as the initial condition, the  $\phi_2$  and  $\phi_3$  fields are initiated here. There are several types of initial conditions here which are chosen by command-line parameters:

- Small random fluctuations are selected using the `-m_random_center_phi_s` or `-m_random_center_phi_p` and `-m_random_fluct` options. These produce a uniform distribution centered at the "center" and with width twice the fluctuation value.
- A two-layer initial condition in the  $y$ -direction is chosen using the `-m_layers` option, whose argument is the fraction of the domain which will be in the "bottom"

The `-m_particles` option is used for the "colliding particles" simulation. This creates a symmetric simulation with a square particle centered on the middle of the  $x$ -axis with width half that of the domain, which is to say, one-quarter of that of the whole symmetric domain.

- The default initial condition is a square half of the domain width, either in the center or, if the  $x$ - and  $y$ -axes are symmetry planes, then at the origin.

If the option `-m_random_fluct` is selected without `-m_random_center`, then random fluctuations with uniform distribution of width twice the fluctuation value are added to any other initial condition present.

#### N.1.4 Global Function `membrane_temp_parameters_line()`

This calculates the membrane temporary parameters  $\mu_2$  and  $\mu_3$ , which are the chemical potentials given as the following:

$$\mu_i = \frac{\delta f}{\delta \phi_i} - K_{ii} \nabla^2 \phi_i - \frac{1}{2} (K_{ij} + K_{ji}) \nabla^2 \phi_j \quad (20)$$

```
void membrane_temp_parameters_line (PetscScalar* x, PetscScalar* temp, int points, int gxm, int
gym, PetscScalar xmin, PetscScalar xmax, PetscScalar ycoord, PetscScalar zcoord, PetscScalar time,
AppCtx* data)
```

- `PetscScalar* x` Array with the "real" field variables.
- `PetscScalar* temp` Array with the temporary field variables.
- `int points` Number of points at which to calculate the temporary variables.

|                                   |                                                                                                                                                   |
|-----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| · <code>int gxm</code>            | The $x$ -width of the “local” vector’s array, including shadow nodes, for the $y$ -increment.                                                     |
| · <code>int gym</code>            | The $y$ -width of the “local” vector’s array, including shadow nodes, for the $z$ -increment.                                                     |
| · <code>PetscScalar xmin</code>   | First node $x$ -coordinate.                                                                                                                       |
| · <code>PetscScalar xmax</code>   | Last node plus one $x$ -coordinate.                                                                                                               |
| · <code>PetscScalar ycoord</code> | This line $y$ -coordinate.                                                                                                                        |
| · <code>PetscScalar zcoord</code> | This line $z$ -coordinate.                                                                                                                        |
| · <code>PetscScalar time</code>   | Current simulation time.                                                                                                                          |
| · <code>AppCtx* data</code>       | Pointer to the main simulation parameter structure, which includes the <code>mparm</code> struct typedef, from which this gets needed parameters. |

### N.1.5 Local Function M33()

```
static inline PetscScalar M33 (PetscScalar phi2, PetscScalar phi3, mparm* themembrane)
```

### N.1.6 Local Function psidoubleprime2()

```
static inline PetscScalar psidoubleprime2 (PetscScalar phi2, PetscScalar phi3, mparm* themembrane)
```

### N.1.7 Local Function psidoubleprime3()

```
static inline PetscScalar psidoubleprime3 (PetscScalar phi2, PetscScalar phi3, mparm* themembrane)
```

### N.1.8 Local Function psiprime2()

This abstracts out the function for  $\Psi'_2(\phi_2, \phi_3)$ , the derivative of homogeneous free energy, so it can be easily modified. Since  $f(\phi_2, \phi_3) = \phi_1 * \ln(\phi_1) + \phi_2 * \ln(\phi_2) + \frac{\phi_3}{m} * \ln(\phi_3) + \chi_{12} * \phi_1 * \phi_2 + \chi_{23} * \phi_2 * \phi_3 + \chi_{13} * \phi_1 * \phi_3$  if  $m_1, m_2$  are chosen as 1 and  $m_3$  is chosen as  $m$ , this returns  $-\log(1.0 - \phi_2 - \phi_3) + \log(\phi_2) + \chi_{12} * (1.0 - 2.0 * \phi_2 - \phi_3) + \chi_{23} * \phi_3 - \chi_{13} * \phi_3$ .

```
static inline PetscScalar psiprime2 (PetscScalar phi2, PetscScalar phi3, mparm* themembrane)
```

|                                      |                                                       |
|--------------------------------------|-------------------------------------------------------|
| · <code>PetscScalar psiprime2</code> | It returns the derivative of homogeneous free energy. |
| · <code>PetscScalar phi2</code>      | The $\phi_2$ parameter it’s a function of.            |
| · <code>PetscScalar phi3</code>      | The $\phi_3$ parameter it’s a function of.            |
| · <code>mparm* themembrane</code>    | membrane parameter structure.                         |

### N.1.9 Local Function psiprime3()

This abstracts out the function for  $\Psi'_3(\phi_2, \phi_3)$ , the derivative of homogeneous free energy, so it can be easily modified. Since  $f(\phi_2, \phi_3) = \phi_1 * \ln(\phi_1) + \phi_2 * \ln(\phi_2) + \frac{\phi_3}{m} * \ln(\phi_3) + \chi_{12} * \phi_1 * \phi_2 + \chi_{23} * \phi_2 * \phi_3 + \chi_{13} * \phi_1 * \phi_3$  if  $m_1, m_2$  are chosen as 1 and  $m_3$  is chosen as  $m$ , this returns  $-1.0 - \log(1 - \phi_2 - \phi_3) + \frac{1.0}{m} + \frac{1.0}{m} * \log(\phi_3) - \chi_{12} * \phi_2 + \chi_{23} * \phi_2 - \chi_{13} * (1.0 - 2.0 * \phi_3 - \phi_2)$

```
static inline PetscScalar psiprime3 (PetscScalar phi2, PetscScalar phi3, mparm* themembrane)
```

|                                      |                                                       |
|--------------------------------------|-------------------------------------------------------|
| · <code>PetscScalar psiprime3</code> | It returns the derivative of homogeneous free energy. |
| · <code>PetscScalar phi2</code>      | The $\phi_2$ parameter it’s a function of.            |
| · <code>PetscScalar phi3</code>      | The $\phi_3$ parameter it’s a function of.            |
| · <code>mparm* themembrane</code>    | membrane parameter structure.                         |

## O File membrane.h

RCS Header: /cvsroot/rheoplast/membrane.h,v 1.12 2004/07/28 16:31:13 zhou Exp

The typedefs and prototypes for Cahn-Hilliard species transport.

### Included Files

```
#include </usr/lib/petsc/include/petsc.h>
```

### Preprocessor definitions

```
#define MEMBRANE_H
```

```
#define MEMBRANE_HELP "Cahn-Hilliard species transport is a basic staple of phase field modeling.\n\nTo use it, add option:\n -with-membrane\n\nand control it with the properties:\n\n -m.nonsolvent_m degree of mopolymerization of the nonsolvent\n -m.solvent_m degree of mopolymerization of the solvent\n -m.polymer_m degree of mopolymerization of the polymer\n -mobility_ss mobility\n -mobility_sp mobility\n -mobility_ps mobility\n -mobility_pp mobility\n -K_ss K (gradient penalty coefficient)\n -K_pp K (gradient penalty coefficient)\n -K_sp K (gradient penalty coefficient)\n -K_ps K (gradient penalty coefficient)\n\nThe default initial condition is a centered square. If -symmetry_x and\n -symmetry_y are specified, this is a square centered at the origin. With\n\nboth symmetries, an alternate initial condition with a square centered on\n\nthe middle of the x-axis can be used by specifying:\n -m.particles\n\nOr one can specify a two-layer system in the y-direction using:\n -m.layers <thickness>\n\nwhere thickness is the fraction in the bottom C=1 layer.\n\nOne can also use a random initial distribution to simulate spinodal\n\ndecomposition with:\n -m.random_center_phi_s <center> center of random distribution of phi_s (required)\n -m.random_center_phi_p <center> center of random distribution of phi_p (required)\n -m.random_fluct <fluct> half-width of uniform distribution [0.01]\n\n"
```

### O.1 Type definitions

#### O.1.1 Typedef AppCtx

```
typedef void AppCtx
```

#### O.1.2 Typedef mparm

Structure typedef for ternary Cahn-Hilliard polymer species transport.

```
typedef struct {...} mparm
struct
{
 PetscScalar mobility22;
 PetscScalar mobility23;
 PetscScalar mobility32;
 PetscScalar mobility33;
 PetscScalar K22;
 PetscScalar K33;
 PetscScalar K23;
 PetscScalar K32;
 PetscScalar m1;
 PetscScalar m2;
 PetscScalar m3;
```

```
PetscScalar Fp;
PetscScalar a;
PetscScalar M0;
int phi2var;
int phi3var;
int mu2var;
int mu3var;
}
```

## P File config.h

### Preprocessor definitions

```
#define HAVE_DLFCN_H 1

#define HAVE_INTTYPES_H 1

#define HAVE_LIBPETSC 1

#define HAVE_MEMORY_H 1

#define HAVE_STDINT_H 1

#define HAVE_STDLIB_H 1

#define HAVE_STRINGS_H 1

#define HAVE_STRING_H 1

#define HAVE_SYS_STAT_H 1

#define HAVE_SYS_TYPES_H 1

#define HAVE_UNISTD_H 1

#define PACKAGE "rheoplast"

#define PACKAGE_BUGREPORT ""

#define PACKAGE_NAME ""

#define PACKAGE_STRING ""

#define PACKAGE_TARNAME ""

#define PACKAGE_VERSION ""

#define STDC_HEADERS 1

#define VERSION "0.5.0"
```