

RheoPlast: Marrying Phase Field and Fluid-Structure Interactions

Adam Powell Bo Zhou Jorge Vieyra Wanida Pongsaksawad

Version 0.8.9 released March 27, 2006
Built January 25, 2007 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** timestepping infrastructure.

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 results 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 two programs shipped in the `rheoplast` package: `diffuse` and `rheoplast`. The big deal is `rheoplast`, so feel free to skip to that if you like.

2.1 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 A, page 8)
2. `timestep.c` and `timestep.h` (appendices D and E, pages 20 and 29): timestepping infrastructure.

2.2 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 B and C, pages 11 and 17): `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 D and E, pages 20 and 29): timestepping infrastructure described above.
3. `cahnhill.c` and `cahnhill.h` (appendices F and G, pages 31 and 37): 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 H and I, pages 38 and 45): velocity-vorticity formulation fluid dynamics in 2-D. Documentation includes a complete mathematical description of the formulation.
5. `membrane.c` and `membrane.h` (appendices J and K, pages 46 and 52): Ternary polymer solution module, used to simulate immersion precipitation of polymer membranes.
6. `vectorphase.c` and `vectorphase.h` (appendices L and M, pages 53 and 58): the vector-valued phase field model of Warren, Kobayashi and Carter as described in their 1998 paper [2]. Documentation includes a complete mathematical description of the model. Note it is superseded by their 2000 model [3], as described in the introduction to appendix L.
7. `heatcond.c` and `heatcond.h` (appendices N and O, pages 59 and 62): heat conduction, also convection.
8. `pressflow.c` and `pressflow.h` (appendices P and Q, pages 63 and 69): velocity-pressure formulation Navier-Stokes fluid dynamics in 2-D or 3-D. This uses the conventional velocity staggered mesh.
9. `shearstrain.c` and `shearstrain.h` (appendices R and S, pages 70 and 74): shear strain field in the mixed-stress model described in that appendix. This is the model used to do fluid-structure interactions with phase field.
10. `electra.c` and `electra.h` (appendices T and U, pages 75 and 79):

Items 3 through 10 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.

2.2.1 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.
- 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`.

| RheoPlast module | cahnhill | ternary | flow | 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 | ✓ | | (✓) | (✓) | | | | ✓ |
| Ternary e-chem, EMR | | ✓ | (✓) | (✓) | | | | ✓ |

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

- Search `rheoplast.c` for the word “modules” and look for where to add your module’s function calls, as described in appendix B. 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.

2.2.2 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 unwinding.
- 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.

Figure 1: RheoPlast detailed flowchart.

2.3 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.

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 (removed after `RheoPlast` version 0.5 was 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 the old `cahnhill` was built, the solid impinging particle simulations could 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" (field variables).

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 now 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 -ts_max_steps  
1000
```

The corresponding simulation using `pressflow` is considerably slower:

```
rheoplast -da_grid_x 25 -da_grid_y 25 -width_x 0.00375 -width_y 0.00375  
-symmetry_xmin -symmetry_ymin -boundary_xmax -boundary_ymax -with_cahnhill  
-ch_surftens 0.03 -with_pressflow -density 1000 -viscosity 0.0004 -ts_dt  
0.00002 -ts_dt_max 0.001 -ts_max_steps 1000 -tsstyle implicit
```

Unfortunately, the `vortflow` module does not interact well with `shearstrain`, so we replace it with the standard velocity-pressure Navier-Stokes formulation. Again unfortunately, this formulation makes symmetry boundary conditions difficult, so one must include the full geometry in the simulation domain. Running the oscillating solid droplet simulation can then be done with command line:

```
rheoplast -da_grid_x 50 -da_grid_y 50 -width_x 0.00375 -width_y 0.00375  
-with_cahnhill -ch_surftens 0.03 -with_vortflow -density 1000 -viscosity 0.0004  
-with_shearstrain -shear_modulus -ts_dt 0.00002 -ts_dt_max 0.001 -ts_max_steps  
1000
```

In the future, the 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, a poster at the 2004 Gordon Research Conference on Polymer Membranes, and a talk at the 2004 MRS Fall Meeting, 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
-K_ss 1e-4 -K_pp 1e-4
```

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 without flow but with variable polymer mobility:

Get these from Bo!

The enhanced polymer mobility at low polymer concentrations has a dramatic effect on membrane morphology.

¹Requires modification to functions `psiprime2()` and `psiprime3()` (appendices J.1.13 and J.1.14, pages 51 and 51) which are not part of `RheoPlast` version 0.5.

- 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
```

This demonstrates the effects of fluid flow on membrane morphology. In particular, the ratio of the Sc and Fp parameters (Schmidt number and force parameter) appears to determine 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, 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

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.

5.2 RheoPlast 0.8.9

This is a pre-0.9 snapshot with expanded capabilities but relatively little documentation.

A File `diffuse.c`

RCS Header: `/cvsroot/rheoplast/diffuse.c,v 1.31 2006/02/10 01:02:46 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 E)
#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... )
```

A.1 Type definitions

A.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;
}
```

A.2 Variables

A.2.1 Variable Surf

ISurface Surf

A.2.2 Variable Disp

IDisplay Disp

A.2.3 Local Variables

help

PETSc help string, printed when run with -help.

```
static char help[]
```

A.3 Functions

A.3.1 Global Function func_boundary_line()

```
void func_boundary_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, int side )
```

A.3.2 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 )
```

A.3.3 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 )
```

A.3.4 Global Function main()

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

A.3.5 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.

A.3.6 Global Function `temp_parameters_boundary_line()`

```
void temp_parameters_boundary_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, int side )
```

A.3.7 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.

A.3.8 Global Function `tsmonitor()`

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

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

- `int tsmmonitor` Returns zero or an error code.
- `int time` Current timestep number.

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

B File rheoplast.c

RCS Header: /cvsroot/rheoplast/rheoplast.c,v 1.113 2006/03/06 19:37:36 wanida 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 C)
#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 E)
#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 G)
#include </usr/lib/petsc/include/petsc.h>
#include "vortflow.h" (Section I)
#include </usr/lib/petsc/include/petsc.h>
#include "membrane.h" (Section K)
#include </usr/lib/petsc/include/petsc.h>
#include "vectorphase.h" (Section M)
#include </usr/lib/petsc/include/petsc.h>
#include "heatcond.h" (Section O)
#include </usr/lib/petsc/include/petsc.h>
#include "pressflow.h" (Section Q)
#include </usr/lib/petsc/include/petsc.h>
#include "shearstrain.h" (Section S)
#include </usr/lib/petsc/include/petsc.h>
#include "electra.h" (Section U)
#include </usr/lib/petsc/include/petsc.h>
#include "chternary.h" (Section W)
#include </usr/lib/petsc/include/petsc.h>
#include "cahnhill.h" (Section G)
#include "vortflow.h" (Section I)
#include "membrane.h" (Section K)
#include "vectorphase.h" (Section M)
```

```

#include "heatcond.h" (Section O)
#include "pressflow.h" (Section Q)
#include "shearstrain.h" (Section S)
#include "electra.h" (Section U)
#include "chternary.h" (Section W)

```

```
#include <string.h>
```

Preprocessor definitions

```

#define __FUNCT__ "temp_parameters_line"
#define __FUNCT__ "temp_parameters_boundary_line"
#define __FUNCT__ "func_interior_line"
#define __FUNCT__ "func_boundary_line"
#define __FUNCT__ "step_interior_line"
#define __FUNCT__ "jack_interior_line"
#define __FUNCT__ "calculate_integration_variables"
#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"

```

B.1 Variables

B.1.1 Variable Surf

```
ISurface Surf
```

B.1.2 Variable Disp

```
IDisplay Disp
```

B.1.3 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[]
```

B.2 Functions

B.2.1 Global Function `calculate_integration_variables()`

Loop over y and z , calling `*_integrate_interior_line`, to calculate the integration variable field.

```
int calculate_integration_variables ( AppCtx* data, PetscScalar time )
```

· `int calculate_integration_variables` returns zero (or an error code).

- `AppCtx* data` The data structure to use.
- `PetscScalar time` Current simulation time.

B.2.2 Global Function `func_boundary_line()`

Evaluate the functions which make up this system on one line of boundary 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_boundary_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, int side )
```

- `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.
- `int side` Side on which to calculate the function values.

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

B.2.3 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 )
```

| | |
|---|--|
| · <code>PetscScalar* x</code> | The data to evaluate the function for. |
| · <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>PetscScalar time</code> | Current simulation time. |
| · <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 gzm</code> | Overall z -width of the array (zero if 2-D). |
| · <code>int xs</code> | Starting x -coordinate of the line. |
| · <code>int ys</code> | Starting y -coordinate of the line. |
| · <code>int zs</code> | Starting z -coordinate of the line. |
| · <code>int xm</code> | Width of the interior part of the line. |
| · <code>int nx</code> | Overall x -width of the entire global array. |
| · <code>int ny</code> | Overall y -width of the entire global array. |
| · <code>int nz</code> | Overall z -width of the entire global array (zero if 2-D). |
| · <code>void* user</code> | User data structure pointer. |

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

B.2.4 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 )
```

| | |
|----------------------------------|---|
| · <code>PetscScalar* x</code> | The data to evaluate the function for. |
| · <code>PetscScalar* temp</code> | Array of temporary field variables. |
| · <code>Mat jack</code> | Matrix into which to insert Jacobian values. |
| · <code>PetscScalar time</code> | Current simulation time. |
| · <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 gzm</code> | Overall z -width of the array (zero if 2-D). |
| · <code>int xs</code> | Starting x -coordinate of the line. |
| · <code>int ys</code> | Starting y -coordinate of the line. |
| · <code>int zs</code> | Starting z -coordinate of the line. |
| · <code>int xm</code> | Width of the interior part of the line. |
| · <code>int nx</code> | Overall x -width of the entire global array. |
| · <code>int ny</code> | 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).

B.2.5 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.

RheoPlast begins its run with PETSc and log file initialization.

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

It then determines which modules will be included in the simulation, and calls those modules' initialization functions to determine the number of field variables and temporary fields.

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

We next create the integration vector if necessary

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.

B.2.6 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.

B.2.7 Global Function `temp_parameters_boundary_line()`

This function calculates the temporary parameters on which the function calculation is based on the boundary (in section B.2.3). 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_boundary_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, int side )
```

- `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.
- `int side` Side on which to calculate the temp parameters.

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

B.2.8 Global Function `temp_parameters_line()`

This function calculates the temporary parameters on which the function calculation is based (in section B.2.3). 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.

B.2.9 Global Function `tsmonitor()`

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

```
int tsmonitor ( int step, PetscScalar time, PetscScalar deltat, void* user )
```

- `int tsmonitor` Returns zero or an error code.
- `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 (Δt , 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.

C File `rheoplast.h`

RCS Header: `/cvsroot/rheoplast/rheoplast.h,v 1.36 2006/03/06 19:37:36 wanida Exp`

This `#include`s 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 E)
    #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 G)
    #include </usr/lib/petsc/include/petsc.h>

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

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

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

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

#include "pressflow.h" (Section Q)
    #include </usr/lib/petsc/include/petsc.h>

#include "shearstrain.h" (Section S)
    #include </usr/lib/petsc/include/petsc.h>

#include "electra.h" (Section U)
    #include </usr/lib/petsc/include/petsc.h>

#include "chternary.h" (Section W)
    #include </usr/lib/petsc/include/petsc.h>

#include "cahnhill.h" (Section G)
#include "vortflow.h" (Section I)
#include "membrane.h" (Section K)
#include "vectorphase.h" (Section M)
#include "heatcond.h" (Section O)
#include "pressflow.h" (Section Q)
#include "shearstrain.h" (Section S)
#include "electra.h" (Section U)
#include "chternary.h" (Section W)

```

Preprocessor definitions

```

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

```

C.1 Type definitions

C.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 intvars;
    int bcflags;
    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;
    Vec intvec;
    CommStyle style;
    PetscTruth cahnhill;
    PetscTruth vortflow;
    PetscTruth membrane;
    PetscTruth vectorphase;
    PetscTruth heatcond;
    PetscTruth pressflow;
    PetscTruth shearstrain;
    PetscTruth electra;
    PetscTruth chternary;
```

```

    chparam thecahnhill;
    vortparam thevortex;
    mparam themembrane;
    vectorphasers thephasers;
    heatparam theheater;
    pressparam thepressure;
    strainparam thestrain;
    echemparam thepotential;
    chtparam thechternary;
}

```

D File timestep.c

RCS Header: /cvsroot/rheoplast/timestep.c,v 1.98 2006/03/08 03:21:15 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 E)
    #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"

```

D.1 Variables

D.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
```

unm1

```
static Vec unm1
```

unm2

```
static Vec unm2
```

unm3

```
static Vec unm3
```

unm4

```
static Vec unm4
```

unm5

```
static Vec unm5
```

current_time

```
static PetscScalar* current_time
```

temp_fields

```
static PetscScalar* temp_fields
```

implicit_deltat

```
static PetscScalar implicit_deltat
```

timestyle_coefficient

```
static PetscScalar timestyle_coefficient
```

```

implicit_timestyle
static EqStyle* implicit_timestyle

timestep_style
static TSStyle timestep_style

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_bcflags
static int implicit_bcflags

implicit_symmtypes
static int* implicit_symmtypes

implicit_dadims
static int implicit_dadims

```

D.2 Functions

D.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 bcflags, int* symmtypes,
void* user, FILE* logfile )
```

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

D.2.2 Global Function `implicit_steptime()`

This function implements constrained implicit timestepping as described above.

```
int implicit_steptime ( DA theda, Vec unip1, Vec unip1_local, Mat jack, int* currentstep, int
timesteps, PetscScalar* currenttime, PetscScalar deltat, int tps, EqStyle* timestyle, int bcflags,
int* symmtypes, void* user, FILE* logfile )
```

- `int implicit_steptime` Returns 0 or error code: -1 for malloc error.
- `DA theda` The DA object we're working on.
- `Vec unip1` Global vector of all fields.
- `Vec unip1_local` Local vector of all fields.
- `Mat jack` Matrix to use as Jacobian.
- `int* currentstep` Entering: initial timestep number, return: final timestep number.
- `int timesteps` Number of timesteps to run.
- `PetscScalar* currenttime` Entering: initial time, return: final time.
- `PetscScalar deltat` Timestep size.
- `int tps` Number of temporary field variables to store in array.
- `EqStyle* timestyle` Constraint, time derivative, or blend nature of each field equation.
- `int bcflags` Flags controlling boundary conditions.

- `int* symmtypes` Array of symmetry types for each dof (if NULL then assumes all are MIRROR_PLANE).
- `void* user` User parameters' structure.
- `FILE* logfile` 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.

Timestep style is Crank-Nicholson by default, or you can specify otherwise using: `-tsstyle implicit` for fully-implicit or `-tsstyle gear_2` for M=2 Gear style (see Jorge's reference).

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 `tymestyle` be either `CONSTRAINT_ONLY` or `TIME_CONST_BLEND` so the residual can be directly set at those nodes (done in function `FuncEvaluate`, section D.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/utis/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, with the correct `timestyle` coefficient for multiplying timestep equation functions and Jacobian rows (Δt for fully implicit, $\frac{\Delta t}{2}$ for Crank-Nicholson, $\frac{2\Delta t}{3}$ for M=2 Gear, etc.).
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 the timestep was successful, so produce the new next timestep guess vector and copy solution and function vectors into the old, update the time, and call the monitor.

The command-line option `-guess_explicit` turns on "explicit initial guess" for the next timestep, which extrapolates the difference between the previous and current timesteps to estimate the next one.

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

#### D.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 unpr1, 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 unpr1`
- `Mat* jack` Jacobian matrix.
- `Mat* preck` Preconditioner matrix.
- `MatStructure* flag` Flag indicating preconditioner structure.
- `void* user` User parameters' structure.

Vec unpr1 Field variable values.

#### D.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.

#### D.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.

### D.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.

### D.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.

|                               |                                                                                   |
|-------------------------------|-----------------------------------------------------------------------------------|
| · <code>int zs</code>         | Starting global $z$ -coordinate of the interior region.                           |
| · <code>int xm</code>         | Width of the interior region in the $x$ -direction.                               |
| · <code>int ym</code>         | Width of the interior region in the $y$ -direction.                               |
| · <code>int zm</code>         | Width of the interior region in the $z$ -direction.                               |
| · <code>int gxs</code>        | Starting global $x$ -coordinate of the local array.                               |
| · <code>int gys</code>        | Starting global $y$ -coordinate of the local array.                               |
| · <code>int gzs</code>        | Starting global $z$ -coordinate of the local array.                               |
| · <code>int gxm</code>        | Full width of the local array in the $x$ -direction.                              |
| · <code>int gym</code>        | Full width of the local array in the $y$ -direction.                              |
| · <code>int gzm</code>        | Full width of the local array in the $z$ -direction.                              |
| · <code>int ny</code>         | Overall width of the entire distributed array in the $y$ -direction.              |
| · <code>int* symmtypes</code> | Array of symmetry types for each dof (if NULL then assumes all are MIRROR_PLANE). |
| · <code>int dof</code>        | Number of degrees of freedom per node.                                            |

### D.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)
```

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

### D.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.

### D.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.

## E 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 ALL_PERIODIC 0x000
#define XMIN_SYMMETRY 0x001
#define XMAX_SYMMETRY 0x002
#define YMIN_SYMMETRY 0x004
#define YMAX_SYMMETRY 0x008
#define ZMIN_SYMMETRY 0x010
#define ZMAX_SYMMETRY 0x020
#define XMIN_BOUNDARY 0x040
#define XMAX_BOUNDARY 0x080
#define YMIN_BOUNDARY 0x100
#define YMAX_BOUNDARY 0x200
#define ZMIN_BOUNDARY 0x400
#define ZMAX_BOUNDARY 0x800
#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

#define TIMESTEP_HELP "This includes a timestepping infrastructure with explicit and
semi-implicit\ntimestepping. If both are used, explicit timestepping comes first.\n\nExplicit
parameters are set using flags:\n -explicit_timesteps <timesteps> Number of explicit timesteps\n
-explicit_monsteps <monsteps> Timesteps between tsmonitor() calls\n -explicit_deltat <dt>
(set automatically if not provided)\nFor explicit finite differencing, it's also possible to
perform some\ncalculations during communication between processors, which should be faster\nin
network-limited situations (but isn't, I'll have to investigate why).\nEnable this with
\n -slowcomm\n\nControl implicit timestepping as follows:\n -ts_max_steps <ts> Number of
semi-implicit timesteps\n -monsteps <m> Timesteps between tsmonitor() calls\n -ts_dt <dt>
Initial semi-implicit timestep size\n -ts_dt_max <dt_max> Maximum semi-implicit timestep size\n
-ts_dt_factor <dt_f> Multiply dt by this each timestep until max reached\nThe default implicit
style is semi-implicit Crank-Nicholson, or one can\nspecify fully-implicit or M=2 Gear style
using, respectively:\n -tsstyle implicit\n -tsstyle gear_2\nThe initial guess for a given timestep
can either be set to the previous\ntimestep (the default), or extrapolated from the previous
two timesteps if\nenabled using the flag:\n -guess_explicit\nTo halve the timestep in case of
non-convergence of the implicit solver:\n -noconv_cutttime\n\nTo use symmetry boundary conditions
(instead of the default periodic), use:\n -symmetry_x\n -symmetry_y\n -symmetry_z\n\n"

```

## E.1 Type definitions

### E.1.1 Typedef CommStyle

```

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

```

### E.1.2 Typedef TSStyle

```

typedef enum {...} TSStyle
enum
{
 FULLY_IMPLICIT;
 CRANK_NICHOLSON;
 GEAR_2;
 GEAR_3;
 GEAR_4;
 GEAR_5;
 GEAR_6;
}

```

### E.1.3 Typedef EqStyle

```
typedef int EqStyle
```

## F File cahnhill.c

RCS Header: /cvsroot/rheoplast/cahnhill.c,v 1.55 2006/02/28 21:00:07 wanida 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 C)
#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 E)
#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 G)
#include </usr/lib/petsc/include/petsc.h>
#include "vortflow.h" (Section I)
#include </usr/lib/petsc/include/petsc.h>
#include "membrane.h" (Section K)
#include </usr/lib/petsc/include/petsc.h>
#include "vectorphase.h" (Section M)
#include </usr/lib/petsc/include/petsc.h>
#include "heatcond.h" (Section O)
#include </usr/lib/petsc/include/petsc.h>
#include "pressflow.h" (Section Q)
#include </usr/lib/petsc/include/petsc.h>
#include "shearstrain.h" (Section S)
#include </usr/lib/petsc/include/petsc.h>
```

|                                           |             |
|-------------------------------------------|-------------|
| #include "electra.h"                      | (Section U) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "chternary.h"                    | (Section W) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "cahnhill.h"                     | (Section G) |
| #include "vortflow.h"                     | (Section I) |
| #include "membrane.h"                     | (Section K) |
| #include "vectorphase.h"                  | (Section M) |
| #include "heatcond.h"                     | (Section O) |
| #include "pressflow.h"                    | (Section Q) |
| #include "shearstrain.h"                  | (Section S) |
| #include "electra.h"                      | (Section U) |
| #include "chternary.h"                    | (Section W) |

## 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_temp_parameters_boundary_line"

#define __FUNCT__ "cahnhill_interior_line_function"

#define __FUNCT__ "cahnhill_boundary_line_function"

#define __FUNCT__ "cahnhill_interior_line_jacobian"

```

## F.1 Functions

### F.1.1 Global Function `cahnhill_boundary_line_function()`

This calculates the time derivatives and constraint functions for the Cahn-Hilliard equations for a boundary line. Note the `shearstrain` module interaction programmed here;

```
void cahnhill_boundary_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, int side)
```

- `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.
- `int side`

`void cahnhill_interior_line_function` It returns nothing.

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

And a convective term with first-order upwinding for velocity-pressure flow.

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

And a convective term with first-order upwinding for velocity-pressure flow.

### F.1.2 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()`.

### F.1.3 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 44. 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)
```

- `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 `chparam` 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.

And a convective term with first-order upwinding for velocity-pressure flow.

### F.1.4 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)
```

- `PetscScalar* x`                    The field variables from which to evaluate the Jacobian.
- `PetscScalar* temp`                Array of temporary field variables.
- `Mat J`                                Where to put the evaluated Jacobian.
- `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 firstrow`                      The matrix row number corresponding to the first point in the line.
- `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 `chparm` 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 F.1.8 (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.

### F.1.5 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 F.1.9, page 37.

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 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.

#### F.1.6 Global Function `cahnhill_temp_parameters_boundary_line()`

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

#### F.1.7 Global Function `cahnhill_temp_parameters_line()`

This calculates the Cahn-Hilliard temporary parameter  $\mu$ , which is the chemical potential given by equation 44.

```
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.                                                                   |

- 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 `chparm` struct typedef, from which this gets needed parameters.

### F.1.8 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, chparm* thecahnhill)
```

- PetscScalar psidoubleprime            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.

### F.1.9 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.

## G File `cahnhill.h`

RCS Header: `/cvsroot/rheoplast/cahnhill.h,v 1.9 2006/02/13 05:32:23 wanida 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.\n\nTo use it, add option:\n -with-cahnhill\n\nand control it with the properties:\n\n-ch_intwidth <epsilon> interface thickness/dx [3.0]\n\n-ch_surftens <sigma> interface energy [1.0]\n\n-ch_mobility <kappa> mobility [epsilon^2]\n\nOne can also use a polymer solution model by selecting\n\n-ch_polymer and setting properties:\n\n-ch_polymer_chi <chi> interaction parameter [0.68]\n\n-ch_polymer_m <m> extent of polymerization [64.0]\n\nThe default initial condition is a centered square. If -symmetry_x and\n\n-symmetry_y are specified, this is a square centered at the
```

origin. An alternate initial condition with a square (cube) centered on the middle of the x-axis can be used (automatically turning on all symmetries) by specifying: `-ch_particles` Or one can specify a two-layer system in the y-direction using: `-ch_layers <thickness>` where thickness is the fraction in the bottom C=1 layer, or `-ch_trilayer` for a C=1, C=0, C=1 three-layer initial condition. One can also use a random initial distribution to simulate spinodal decomposition with: `-ch_random_center <center>` center of random distribution (required) `-ch_random_fluct <fluct>` half-width of uniform distribution [0.01]

## G.1 Type definitions

### G.1.1 Typedef AppCtx

```
typedef void AppCtx
```

### G.1.2 Typedef chparm

Structure typedef for Cahn-Hilliard species transport.

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

## H File vortflow.c

**RCS Header: /cvsroot/rheoplast/vortflow.c,v 1.85 2006/03/06 18:15:03 wanida 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 C)
#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 E)
#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 G)
#include </usr/lib/petsc/include/petsc.h>
#include "vortflow.h" (Section I)
#include </usr/lib/petsc/include/petsc.h>
#include "membrane.h" (Section K)
#include </usr/lib/petsc/include/petsc.h>
#include "vectorphase.h" (Section M)
#include </usr/lib/petsc/include/petsc.h>
#include "heatcond.h" (Section O)
#include </usr/lib/petsc/include/petsc.h>
#include "pressflow.h" (Section Q)
#include </usr/lib/petsc/include/petsc.h>
#include "shearstrain.h" (Section S)
#include </usr/lib/petsc/include/petsc.h>
#include "electra.h" (Section U)
#include </usr/lib/petsc/include/petsc.h>
#include "chternary.h" (Section W)
#include </usr/lib/petsc/include/petsc.h>
#include "cahnhill.h" (Section G)
#include "vortflow.h" (Section I)
#include "membrane.h" (Section K)
#include "vectorphase.h" (Section M)
#include "heatcond.h" (Section O)
#include "pressflow.h" (Section Q)
#include "shearstrain.h" (Section S)
#include "electra.h" (Section U)
#include "chternary.h" (Section W)
```

## Preprocessor definitions

```
#define __FUNCT__ "vortflow_first_setup"
```

```

#define __FUNCT__ "vortflow_labels_initcond"
#define u(point)
#define v(point)
#define omega(point)
#define vis(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 C2(point)
#define C3(point)
#define Mu2(point)
#define Mu3(point)
#define __FUNCT__ "vortflow_temp_parameters_line"
#define __FUNCT__ "vortflow_temp_parameters_boundary_line"
#define __FUNCT__ "vortflow_interior_line_function"
#define interp_function(x)

```

This calculates the time derivatives and constraint functions for the velocity-vorticity form of the Navier-Stokes equations for a boundary line.

void vortflow\_boundary\_line\_function It returns nothing.

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.

int side Side on which to calculate the function values.

```

#define __FUNCT__ "vortflow_boundary_line_function"

```

## H.1 Functions

### H.1.1 Global Function `vortflow__boundary__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_boundary_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, int side)
```

- `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.
- `int side`

`void vortflow__interior__line__function` It returns nothing.

### H.1.2 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.

### H.1.3 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)
```

|                                               |                                                                                                                                                      |
|-----------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| · <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>vortparm</code> 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.

Dynamics of the elastic shear strain term are documented in `shearstrain.c` (appendix R, page 70). The curl of the shear stress divergence, which is necessary here, is given by

$$\nabla \times (-\nabla \cdot \tau) = G \nabla \times (\nabla \cdot \gamma) = G \left( \frac{\partial^2 \gamma_{xy}}{\partial x^2} + \frac{\partial^2 \gamma_{yy}}{\partial x \partial y} - \frac{\partial^2 \gamma_{xx}}{\partial x \partial y} - \frac{\partial^2 \gamma_{yx}}{\partial y^2} \right). \quad (18)$$

The symmetry of the shear strain tensor, and the incompressibility condition ( $\gamma_{xx} + \gamma_{yy} = 0$ ) reduce this easily to a function of just  $\gamma_{xx}$  and  $\gamma_{xy}$ . Because the shear strain is on a staggered mesh, this becomes a somewhat complex thing whose second derivatives are calculated on a stencil four nodes across, looking like:

$$y''|_{x_{1.5}} = \frac{y_0 - y_1 - y_2 + y_3}{2(\Delta x)^2}. \quad (19)$$

With shear strain and phase field, we use an interpolation function of the order parameter to weight the elastic and viscous stresses according to

$$\tau = p(\phi)\tau_{el} + (1 - p(\phi))\tau_{visc}. \quad (20)$$

This interpolation function is most simply expressed as

$$p(C) = \int_{-\infty}^{\phi} \begin{cases} 0, \phi' \geq 1 \\ a(\phi' - \phi_0)(1 - \phi'), \phi_0 \leq \phi' \leq 1, \\ 0, \phi' \leq \phi_0 \end{cases} d\phi', \quad (21)$$

where  $\phi_0$  is the threshold  $\phi$  value below which solid behavior goes away and  $a$  is a scaling constant such that  $p(\phi > 1) = 1$ . For  $\phi_0 = 0.1$ , this evaluates to

$$p(\phi) = \frac{1}{0.1215} \left[ -\frac{\phi^3}{3} + \frac{1.1\phi^2}{2} - 0.1\phi + \frac{0.029}{6} \right]. \quad (22)$$

This is also applied for Cahn-Hilliard with  $C$  taking the place of  $\phi$ .

With shear strain and no phase field, we can use this for incompressible viscoelastic mechanics by applying both the elastic and viscous stresses.

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 H.1.4, page 44).

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

#### H.1.4 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.

#### H.1.5 Global Function `vortflow_temp_parameters_boundary_line()`

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

#### H.1.6 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 <code>vortparm</code> struct typedef, from which this gets needed parameters. |

## I File vortflow.h

**RCS Header:** /cvsroot/rheoplast/vortflow.h,v 1.25 2006/03/03 00:52:57 wanida 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"
```

### I.1 Type definitions

#### I.1.1 Typedef AppCtx

```
typedef void AppCtx
```

#### I.1.2 Typedef vortparm

Structure typedef for velocity-vorticity fluid flow.

```
typedef struct {...} vortparm
struct
{
 PetscScalar viscosity;
 PetscScalar density;
 PetscScalar F;
```

```

PetscScalar sineFt0;
PetscScalar sineFmax;
PetscScalar uniFmax;
PetscScalar stagnation;
PetscScalar u_init;
PetscScalar omega_init;
PetscTruth solid;
PetscTruth uniform;
int uvar;
int vvar;
int omegavar;
int visvar;
}

```

## J File membrane.c

RCS Header: /cvsroot/rheoplast/membrane.c,v 1.47 2005/12/27 00:56:02 zhou Exp

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

$$\mathcal{F} = \int \left( f(\phi_2, \phi_3) + \frac{1}{2} \sum_{i,j=2,3} K_{ij} \nabla \phi_i \cdot \nabla \phi_j \right) dV, \quad (23)$$

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 C)
#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 E)
#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 G)
#include </usr/lib/petsc/include/petsc.h>
#include "vortflow.h" (Section I)
#include </usr/lib/petsc/include/petsc.h>
#include "membrane.h" (Section K)
#include </usr/lib/petsc/include/petsc.h>

```

|                                           |             |
|-------------------------------------------|-------------|
| #include "vectorphase.h"                  | (Section M) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "heatcond.h"                     | (Section O) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "pressflow.h"                    | (Section Q) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "shearstrain.h"                  | (Section S) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "electra.h"                      | (Section U) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "chternary.h"                    | (Section W) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "cahnhill.h"                     | (Section G) |
| #include "vortflow.h"                     | (Section I) |
| #include "membrane.h"                     | (Section K) |
| #include "vectorphase.h"                  | (Section M) |
| #include "heatcond.h"                     | (Section O) |
| #include "pressflow.h"                    | (Section Q) |
| #include "shearstrain.h"                  | (Section S) |
| #include "electra.h"                      | (Section U) |
| #include "chternary.h"                    | (Section W) |

## Preprocessor definitions

```

#define __FUNCT__ "psiprime2"

#define __FUNCT__ "psiprime3"

#define __FUNCT__ "psidoubleprime2"

#define __FUNCT__ "psidoubleprime3"

#define __FUNCT__ "M22"

#define __FUNCT__ "M33"

#define __FUNCT__ "M23"

#define __FUNCT__ "M32"

#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_temp_parameters_boundary_line"
#define __FUNCT__ "membrane_interior_line_function"
#define u(point)
#define v(point)
#define u(point)
#define v(point)
#define w(point)
#define __FUNCT__ "membrane_boundary_line_function"
#define u(point)
#define v(point)
#define u(point)
#define v(point)
#define w(point)
#define u(point)
#define v(point)
#define u(point)
#define v(point)
#define w(point)

```

## J.1 Functions

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

```

void membrane_boundary_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, int side)

```

### J.1.2 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()`.

### J.1.3 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 47. That time derivative is thus given by

$$\frac{\partial \phi_i}{\partial t} = \nabla \cdot (M_{ij} \nabla \mu_j). \quad (24)$$

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)
```

|                                               |                                                                                                                                                   |
|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| · <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>mparm</code> struct typedef, from which this gets needed parameters. |

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

### J.1.4 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)
```

|                                         |                                                                            |
|-----------------------------------------|----------------------------------------------------------------------------|
| · <code>PetscScalar* globalarray</code> | The global field array.                                                    |
| · <code>int nx</code>                   | Overall $x$ -width of the global array.                                    |
| · <code>int ny</code>                   | Overall $y$ -width of the global array.                                    |
| · <code>int nz</code>                   | Overall $z$ -width of the global array.                                    |
| · <code>int xm</code>                   | The $x$ -width of the local part of the array.                             |
| · <code>int ym</code>                   | The $y$ -width of the local part of the array.                             |
| · <code>int zm</code>                   | The $z$ -width of the local part of the array.                             |
| · <code>int xs</code>                   | The (integer) $x$ -coordinate of the start of the local part of the array. |
| · <code>int ys</code>                   | The (integer) $y$ -coordinate of the start of the local part of the array. |
| · <code>int zs</code>                   | The (integer) $z$ -coordinate of the start of the local part of the array. |
| · <code>int vars</code>                 | 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.

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

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

### J.1.6 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:

$$\mu_i = \frac{\delta f}{\delta \phi_i} - K_{ii} \nabla^2 \phi_i - \frac{1}{2} (K_{ij} + K_{ji}) \nabla^2 \phi_j \quad (25)$$

```
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.
- `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.

### J.1.1.7 Local Function M22()

```
static inline PetscScalar M22 (PetscScalar phi2, PetscScalar phi3, mparm* themembrane)
```

### J.1.1.8 Local Function M23()

```
static inline PetscScalar M23 (PetscScalar phi2, PetscScalar phi3, mparm* themembrane)
```

### J.1.1.9 Local Function M32()

```
static inline PetscScalar M32 (PetscScalar phi2, PetscScalar phi3, mparm* themembrane)
```

### J.1.1.10 Local Function M33()

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

### J.1.1.11 Local Function psidoubleprime2()

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

### J.1.1.12 Local Function psidoubleprime3()

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

### J.1.1.13 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)
```

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

### J.1.14 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)
```

- PetscScalar psiprime3                    It returns the derivative of homogeneous free energy.
- PetscScalar phi2                        The  $\phi_2$  parameter it's a function of.
- PetscScalar phi3                        The  $\phi_3$  parameter it's a function of.
- mparm\* themembrane                      membrane parameter structure.

## K File membrane.h

RCS Header: /cvsroot/rheoplast/membrane.h,v 1.14 2005/04/18 20:17:14 hazelsct 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\n -m_solvent_m degree of mopolymerization of the solvent \n\n -m_polymer_m degree of mopolymerization of the polymer \n\n -mobility_ss mobility \n\n -mobility_sp mobility \n\n -mobility_ps mobility \n\n -mobility_pp mobility \n\n -K_ss K (gradient penalty coefficient)\n\n -K_pp K (gradient penalty coefficient)\n\n -K_sp K (gradient penalty coefficient)\n\n -K_ps K (gradient penalty coefficient)\n\nThe default initial condition is a centered square. If -symmetry_x and\n\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\n -m_particles\n\nOr one can specify a two-layer system in the y-direction using:\n\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\n -m_random_center_phi_s <center> center of random distribution of phi_s (required)\n\n -m_random_center_phi_p <center> center of random distribution of phi_p (required)\n\n -m_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 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 Sc;
 PetscScalar Fp;
 PetscScalar a;
 PetscScalar M0;
 PetscScalar hd_ss;
 PetscScalar hd_pp;
 int phi2var;
 int phi3var;
 int mu2var;
 int mu3var;
}

```

## L File vectorphase.c

RCS Header: /cvsroot/rheoplast/vectorphase.c,v 1.72 2005/04/19 19:52:34 el\_oso Exp

This has all of the vector phase field parts of rheoplast. It currently uses the Kobayashi, Warren and Carter 1998 formulation.

### Included Files

```
#include <rheoplast.h>
```

### Preprocessor definitions

```

#define __FUNCT__ "vectorphase_first_setup"
 #define __FUNCT__ "vectorphase_labels_initcond"
 #define p(point)
 #define q(point)
 #define pfunc(point)
 #define qfunc(point)
 #define phi(point)
 #define eps(point)
 #define tau(point)
 #define dphidt(point)
 #define u(point)
 #define v(point)

```

```

#define omega(point)
#define pomega(point)
#define pu(point)
#define pv(point)
#define pw(point)
#define T(point)
#define Tfunc(point)
#define __FUNCT__ "vectorphase_temp_parameters_line"
#define __FUNCT__ "vectorphase_interior_line_function"

```

## L.1 Functions

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

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

For vector-valued phase field, there are two field variables  $p$  and  $q$ , and three temporary fields  $\phi$ ,  $\epsilon$  and  $\tau$ . Throughout this function, 2-D is assumed, since this formulation is really only good for 2-D at this point (though that may change).

```

void vectorphase_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 `vectorphasers` structure this inserts parameters from the command line.

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

This calculates the time derivatives  $\partial p/\partial t$ ,  $\partial q/\partial t$  for an interior line. This will eventually use the free energy of Kobayashi, Warren and Carter 2000 [3] which goes as

$$\mathcal{F} = \int_{\Omega} \left[ f(\phi) + \frac{\nu^2}{2} |\nabla\phi|^2 + s\mu(\phi)|\nabla\theta| + \frac{\epsilon^2}{2} |\nabla\theta|^2 \right] dV, \quad (26)$$

and leads to the dynamic equations for  $\phi$  and  $\theta$ :

$$\tau_{\phi}\phi_t = A = \frac{\delta\mathcal{F}}{\delta\phi} = \nu^2\nabla^2\phi + f'(\phi) + s\mu'(\phi)|\nabla\theta|, \quad (27)$$

$$\tau_{\theta}\mu(\phi)\theta_t = B\mu(\phi) = \frac{\delta\mathcal{F}}{\delta\theta} = s\nabla \cdot \left[ \mu(\phi) \frac{\nabla\theta}{|\nabla\theta|} \right] + \epsilon^2\nabla^2\theta, \quad (28)$$

where  $A$  and  $B$  are used as shorthand for  $\tau_{\phi}\phi_t$  and  $\tau_{\theta}\theta_t$  etc. respectively. Unfortunately, there is a singularity in  $\theta$  diffusivity for  $|\nabla\theta| = 0$ .

Giga and Giga [8] and Kobayashi and Giga [9] have outlined a way to solve this, but we haven't yet implemented it. So for now, I'm going with KWC's original 1998 formulation [2] even though it's not quite right. That formulation begins with a slightly different free energy based on angle  $\theta$  which is the orientation times the rotational symmetry order  $N_0$ . The free energy is given by

$$\mathcal{F} = \int_{\Omega} \left[ \frac{\epsilon}{2} |\nabla\phi|^2 + F(\phi; m) + \frac{\phi^2}{2} \mu(\phi) |\nabla\theta|^2 \right] dV, \quad (29)$$

where  $F$  is the homogeneous free energy density distorted by  $m$ ,  $f$  is its derivative (given below), and  $\mu$  is a weighting function for the gradient penalty in  $\theta$ .

We can then write the variation in energy  $\delta\mathcal{F}$  as

$$\delta\mathcal{F} = -A\delta\phi - B\phi\delta\theta, \quad (30)$$

where

$$A = \nabla \cdot [\epsilon^2 \nabla \phi] + f(\phi; m) - \left[ \frac{1}{2} \frac{d\mu}{d\phi} + \frac{\mu(\phi)}{\phi} \right] |\vec{\beta}|^2, \quad (31)$$

$$B = \frac{1}{\phi} \nabla \cdot [\phi \mu(\phi) \vec{\beta}]. \quad (32)$$

The  $\vec{\beta}$  vector represents an orientation gradient given by

$$\vec{\beta} = \phi \nabla \theta = -\frac{q}{\phi} \nabla p + \frac{p}{\phi} \nabla q.$$

As is typical for phase field, these variations lead directly to equations for dynamics of  $\phi$  and  $\theta$  which are:

$$\tau \frac{\partial \phi}{\partial t} = A, \quad (33)$$

$$\tau \phi \frac{\partial \theta}{\partial t} = B, \quad (34)$$

where  $\tau$  is a kinetic timescale parameter.

The really neat thing is that we can turn these dynamics into those for the vector field using the simple transformation where  $p = \phi \cos \theta$  and  $q = \phi \sin \theta$ , so

$$\tau \frac{\partial p}{\partial t} = A \frac{p}{\phi} - B \frac{q}{\phi}, \quad (35)$$

$$\tau \frac{\partial q}{\partial t} = A \frac{q}{\phi} + B \frac{p}{\phi}. \quad (36)$$

```
void vectorphase_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 variables (phi, epsilon, tau).                                                                                                         |
| · 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 <code>vectorphasers</code> struct typedef, from which this gets needed parameters. |

The  $A$  variation has three terms: the gradient penalty, the homogeneous free energy, and a derivative related to the orientation gradient, which are labeled  $A_1$ ,  $A_2$  and  $A_3$  respectively. The first is given by

$$A_1 = \nabla \cdot (\epsilon^2 \nabla \phi).$$

The second term  $A_2$  is the derivative of the homogeneous free energy with respect to  $\phi$ :

$$A_2 = f(\phi, m) = \phi(1 - \phi) \left( \phi - \frac{1}{2} + m \right)$$

Note that it has local minima at 0 and 1, and if  $m > 0$  then the disordered  $\phi = 0$  phase is stable and the ordered phase metastable, and if  $m < 0$  then the ordered phase is stable and the disordered phase metastable.

The third term  $A_3$  is given by

$$A_3 = - \left[ \frac{1}{2} \frac{d\mu}{d\phi} + \frac{\mu\phi}{\phi} \right] |\vec{\beta}|^2,$$

and for  $\mu(\phi) = \bar{\mu}\phi^l$ , the brackets reduce and  $\vec{\beta}$  expands to

$$A_3 = -\bar{\mu} \left( l + \frac{1}{2} \right) \phi^{l-1} \left[ \left( -\frac{q}{\phi} \frac{\partial p}{\partial x} + \frac{p}{\phi} \frac{\partial q}{\partial x} \right)^2 + \left( -\frac{q}{\phi} \frac{\partial p}{\partial y} + \frac{p}{\phi} \frac{\partial q}{\partial y} \right)^2 \right].$$

Note that the vector component spatial derivatives are calculated by central differencing here. The variables `gradx` and `grady` represent  $\beta_x \phi \Delta x$  and  $\beta_y \phi \Delta y$  respectively.

Because I'm omitting the  $\epsilon'$  term, the  $B$  variation has just one term, given by

$$B = \frac{1}{\phi} \nabla \cdot (\mu(\phi) \phi \vec{\beta}) = \frac{1}{\phi} \left[ \frac{\partial}{\partial x} (\phi \mu(\phi) \beta_x) + \frac{\partial}{\partial y} (\phi \mu(\phi) \beta_y) \right].$$

These terms come together to produce the time derivatives of the vector components  $\partial p / \partial t$  and  $\partial q / \partial t$  as given in equations 35 and 36 above.

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

This sets up the parameters for this equation in the `vectorphasers` struct typedef, the field variable labels, maximum stable explicit timestep size, and initial condition for the vector phase field variables. Model parameters include the energy parameter  $m$  (`-mparam`), the ratio  $\epsilon_0 / \Delta x$  (`-epsilon`), parameters for the  $\phi$ - $\theta$  coupling function  $\mu(\phi) = \bar{\mu}\phi^l$  (`-mubar` and `-muexp`), general symmetry order  $N_0$  (`-n_general`), surface energy anisotropy parameters and kinetic anisotropy parameters which are discussed in `vectorphase_temp_parameters_line` (appendix L.1.4, page 57).

```
void vectorphase_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)
```

|                                         |                                                                            |
|-----------------------------------------|----------------------------------------------------------------------------|
| · <code>PetscScalar* globalarray</code> | The global field array.                                                    |
| · <code>int nx</code>                   | Overall $x$ -width of the global array.                                    |
| · <code>int ny</code>                   | Overall $y$ -width of the global array.                                    |
| · <code>int nz</code>                   | Overall $z$ -width of the global array.                                    |
| · <code>int xm</code>                   | The $x$ -width of the local part of the array.                             |
| · <code>int ym</code>                   | The $y$ -width of the local part of the array.                             |
| · <code>int zm</code>                   | The $z$ -width of the local part of the array.                             |
| · <code>int xs</code>                   | The (integer) $x$ -coordinate of the start of the local part of the array. |
| · <code>int ys</code>                   | The (integer) $y$ -coordinate of the start of the local part of the array. |
| · <code>int zs</code>                   | The (integer) $z$ -coordinate of the start of the local part of the array. |
| · <code>int vars</code>                 | Total number of field variables to be solved.                              |

- `AppCtx* data` Pointer to the `AppCtx` struct typedef, whose `vectorphasers` 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 explicit finite difference criterion which seems to work is one-quarter of the square of the mesh spacing.

The initial condition is generally circular nuclei, with some scattered completely randomly (controlled by parameter `-nukes`), some semi-circles on the edges, for simulations of competitive growth from edges (controlled by parameter `-edge_nukes`), and some circles limited to the interior for similar simulations (controlled by parameter `-interior_nukes`).

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

This calculates the temporary field variables in one line of the global array for vector phase field stuff. In this case, we want to calculate the order parameter  $\phi$ , which is the magnitude of the vector phase field, the orientation-dependent gradient penalty coefficient  $\epsilon$  which determines interfacial energy and thickness, and orientation-dependent kinetic parameter  $\tau$  which determines the growth rate.

```
void vectorphase_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 `vectorphasers` struct typedef, from which this gets needed parameters.

The order parameter  $\phi$  is the most straightforward, it's just the magnitude of the vector phase variable.

The gradient penalty coefficient  $\epsilon$  is quite a bit more complicated, as it's a function of the angle  $\psi$  which represents the misorientation between the gradient of  $\phi$  and the crystal orientation  $\bar{\theta} = \theta/N_0$ . As an intermediate, we'll call  $\bar{\psi}$  the angle of the order parameter gradient, as long as there's 2-fold symmetry this can be represented by

$$\bar{\psi} = \tan^{-1} \left( \frac{\partial\phi/\partial y}{\partial\phi/\partial x} \right).$$

(Actually, since we use the C math library `atan2` function, there's no 2-fold symmetry restriction in the code.)

The functional form of  $\epsilon$  given in KWC 1998 [2] is

$$\epsilon = \epsilon_0 \left[ 1 + \delta_s \left\{ 2 * \left( \frac{1 + \cos N_s \psi}{2} \right)^{n_s} - 1 \right\} \right], \quad (37)$$

where  $\delta_s$ ,  $N_s$  and  $n_s$  are set by parameters `-delta_surface`, `-n_surface` and `-exp_surface` respectively. Note that to calculate  $\psi$  we need nearest-neighbor values of  $\phi$ , but the function needs nearest neighbor

values of  $\epsilon$ . So this forces us to look to next-nearest-neighbors, expanding the required stencil width for these calculations to 2. Make sure these data exist at the edges, or you'll get some wierd behavior at the interfaces!

Note the abstraction violation here: the code calculating  $\epsilon$  assumes that  $\phi$  is already calculated in the negative  $y$  direction nearest neighbor(s), but not in the positive direction. This could therefore fail if for some reason the loops go the other way! A longer-term fix would allow two levels of temporary parameters, but it's not clear it's worth this just to save the square roots which are pretty simple. Also note that this code assumes equal spacings in both grid directions.

The kinetic timescale parameter  $\tau$  is also a function of  $\psi$ , in KWC 1998 they use

*this.*

## M File vectorphase.h

**RCS Header:** /cvsroot/rheoplast/vectorphase.h,v 1.25 2004/08/11 21:56:33 el\_oso Exp

The typedefs and function prototypes for vector-value phase field. 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 VECTORPHASE_H

#define VECTORPHASE_HELP "Vector-value phase field is currently done using the 1998 Kobayashi,
Warren\nand Carter formulation. To use it, add option:\n -with_vectorphase\nand control it
with proprerties:\n -mparam <m> KWC m parameter [0.25]\n -epsilon <eps> epsilon/dx [5.0]\n
-mubar <mb> mubar param in mu=mubar(phi)^muexp [0.01]\n -muexp <me> muexp param in same
[4.0]\n -n_general <NO> general rotation symmetry order [4]\n -delta_surface <ds> relative
variation in epsilon [0.2]\n -n_surface <Ns> surface rotation symmetry order [4]\n -exp_surface
<es> surface energy cosine exponent [3.0]\n -temp_equil <Te> equilibrium temperature [0.0]\n
-m_alpha <alpha> m T-dependance parameter [0.9]\n -m_gamma <gamma> m T-dependance parameter
[20]\nKinetic growth parameters not yet implemented. Initial condition:\n -nukes <n> number
of nucleation sites [5]\n -nuke_theta <th> Theta (degrees) for (all) nuclei [random]\n -nuke_x
<x> One nucleus: dimensionless x coord [random]\n -nuke_y <y> One nucleus: dimensionless y
coord [random]\nFor simulations with symmetry boundary conditions (instead of periodic),\nnuclei
can be planted on the edges or in the interior using:\n -edge_nukes <n> nucleation sites on the
edges [0]\n -interior_nukes <n> nucleation sites in the interior [0]\n\nA nice example with
the default four-fold symmetry and one grain is:\n\n ./rheoplast -da_grid_x 50 -da_grid_y 50
-explicit_timesteps 1000000 -explicit_monsteps 10000 -with_vectorphase -nukes 3 -contours\n\nor if
you wish to use semi-implicit timesteps (explicit is needed to get a\nworkable initial condition
for semi-implicit):\n\n ./rheoplast -da_grid_x 50 -da_grid_y 50 -explicit_timesteps 20000
-explicit_monsteps 10000 -ts_dt 0.02 -ts_max_steps 1000 -monsteps 10 -snes_mf -with_vectorphase
-nukes 1 -contours\n\n"
```

## M.1 Type definitions

### M.1.1 Typedef AppCtx

```
typedef void AppCtx
```

### M.1.2 Typedef vectorphasers

Structure typedef for vector-value phase field.

```
typedef struct {...} vectorphasers
struct
{
 PetscScalar eps_dx;
 PetscScalar mparam;
 PetscScalar mubar;
 PetscScalar muexp;
 PetscScalar NO;
 PetscScalar delta_s;
 PetscScalar N_s;
 PetscScalar n_s;
 PetscScalar T_e;
 PetscScalar C;
 PetscScalar m_alpha;
 PetscScalar m_gamma;
 int pvar;
 int qvar;
 int phivar;
 int epsvar;
 int tauvar;
 int dphidvar;
}
```

## N File heatcond.c

RCS Header: /cvsroot/rheoplast/heatcond.c,v 1.40 2005/04/28 14:55:25 el\_oso Exp

This is a heat conduction module for rheoplast. It is here mostly for doing dendrite simulations.

### Included Files

```
#include "rheoplast.h" (Section C)
#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 E)
#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 G)
#include </usr/lib/petsc/include/petsc.h>
#include "vortflow.h" (Section I)
#include </usr/lib/petsc/include/petsc.h>
#include "membrane.h" (Section K)
#include </usr/lib/petsc/include/petsc.h>
```

```

#include "vectorphase.h" (Section M)
 #include </usr/lib/petsc/include/petsc.h>
#include "heatcond.h" (Section O)
 #include </usr/lib/petsc/include/petsc.h>
#include "pressflow.h" (Section Q)
 #include </usr/lib/petsc/include/petsc.h>
#include "shearstrain.h" (Section S)
 #include </usr/lib/petsc/include/petsc.h>
#include "electra.h" (Section U)
 #include </usr/lib/petsc/include/petsc.h>
#include "chternary.h" (Section W)
 #include </usr/lib/petsc/include/petsc.h>
#include "cahnhill.h" (Section G)
#include "vortflow.h" (Section I)
#include "membrane.h" (Section K)
#include "vectorphase.h" (Section M)
#include "heatcond.h" (Section O)
#include "pressflow.h" (Section Q)
#include "shearstrain.h" (Section S)
#include "electra.h" (Section U)
#include "chternary.h" (Section W)

```

## Preprocessor definitions

```

#define T(point)
 #define Tfunc(point)
 #define dphidt(point)
 #define u(point)
 #define v(point)
 #define pu(point)
 #define pv(point)
 #define pw(point)
 #define __FUNCT__ "heatcond_first_setup"
 #define __FUNCT__ "heatcond_labels_initcond"
 #define __FUNCT__ "heatcond_temp_parameters_line"
 #define __FUNCT__ "heatcond_interior_line_function"

```

## N.1 Functions

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

```

void heatcond_boundary_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, int side)

```

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

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

```

void heatcond_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 `heatparm` structure this inserts parameters from the command line.

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

This calculates the time derivatives  $\partial T/\partial t$  for an interior line.

```
void heatcond_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 `heatparm` struct typedef, from which this gets needed parameters.

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

This sets up the parameters in the `heatparm` struct typedef, field variable labels, maximum stable explicit timestep size, and initial condition for the heat conduction variables.

```
void heatcond_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 `heatparm` 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.

### N.1.5 Global Function `heatcond_temp_parameters_line()`

There are no temporary field variables for heat conduction, so this does nothing.

```
void heatcond_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 `heatparm` struct typedef, from which this gets needed parameters.

## O File `heatcond.h`

**RCS Header:** `/cvsroot/rheoplast/heatcond.h,v 1.19 2005/04/28 14:55:25 el_oso Exp`

The typedefs and prototypes for heat conduction. 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 HEATCOND_H
```

```
#define HEATCOND_HELP "Heat conduction is very simple. To use it, add option:\n -with_heat\nand control it with properties:\n -conductivity <k> thermal conductivity [1.0]\n -density <rho> density [1.0]\n -heatcap <cp> heat capacity [1.0]\n -latentheat <L> for vectorphase coupling [1.0]\n -cooling <C> for vectorphase cooling [0.0]\n -T_cool <T_cool> for vectorphase cooling [0.7]\n\n"
```

### O.1 Type definitions

#### O.1.1 Typedef `AppCtx`

```
typedef void AppCtx
```

## O.1.2 Typedef heatparm

Structure typedef for heat conduction parameters.

```
typedef struct {...} heatparm
struct
{
 PetscScalar conductivity;
 PetscScalar thermdensity;
 PetscScalar Cp;
 PetscScalar latentheat;
 PetscScalar Tinit;
 PetscScalar Tcool;
 PetscScalar C;
 int Tvar;
}
```

## P File pressflow.c

RCS Header: /cvsroot/rheoplast/pressflow.c,v 1.51 2006/03/08 11:50:00 hazelsct Exp

This provides rheoplast with all of the functions for modeling fluid flow using the velocity-pressure formulation.

The incompressible Navier-Stokes equations in velocity-pressure form are annoying because of spurious modes in the pressure. The standard finite difference way around this is to use a “staggered mesh”, in which the  $x$ -velocity mesh is offset from the pressure mesh by one-half grid spacing in the  $x$ -direction,  $y$ -velocity is offset in the  $y$ -direction, etc. This is a bit of a pain in a general parallel code like RheoPlast, particularly with regard to symmetry boundary conditions (hence the presence of the `vortflow` module), which must be specially designed for the staggered mesh. On the other hand, the `shearstrain` module also needs staggered meshes, so the infrastructure is necessary anyway; and velocity-pressure is more efficient in 3-D.

### Included Files

```
#include "rheoplast.h" (Section C)
#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 E)
 #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 G)
 #include </usr/lib/petsc/include/petsc.h>
#include "vortflow.h" (Section I)
 #include </usr/lib/petsc/include/petsc.h>
#include "membrane.h" (Section K)
 #include </usr/lib/petsc/include/petsc.h>
```

|                                           |             |
|-------------------------------------------|-------------|
| #include "vectorphase.h"                  | (Section M) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "heatcond.h"                     | (Section O) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "pressflow.h"                    | (Section Q) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "shearstrain.h"                  | (Section S) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "electra.h"                      | (Section U) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "chternary.h"                    | (Section W) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "cahnhill.h"                     | (Section G) |
| #include "vortflow.h"                     | (Section I) |
| #include "membrane.h"                     | (Section K) |
| #include "vectorphase.h"                  | (Section M) |
| #include "heatcond.h"                     | (Section O) |
| #include "pressflow.h"                    | (Section Q) |
| #include "shearstrain.h"                  | (Section S) |
| #include "electra.h"                      | (Section U) |
| #include "chternary.h"                    | (Section W) |

## Preprocessor definitions

```

#define __FUNCT__ "pressflow_first_setup"

#define __FUNCT__ "pressflow_labels_initcond"

#define u(point)
#define v(point)
#define w(point)
#define p(point)
#define omega(point)
#define ufunc(point)
#define vfunc(point)
#define wfunc(point)
#define pfunc(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 C2(point)

```

```

#define C3(point)
#define Mu2(point)
#define Mu3(point)
#define DIRAC(width, jump, ycoord)
#define __FUNCT__ "pressflow_temp_parameters_line"
#define __FUNCT__ "pressflow_temp_parameters_boundary_line"
#define __FUNCT__ "pressflow_interior_line_function"
#define interp_function(x)
#define __FUNCT__ "pressflow_boundary_line_function"
#define __FUNCT__ "pressflow_interior_line_jacobian"

```

## P.1 Variables

### P.1.1 Local Variables

**sftype**

```
static SFType sftype
```

## P.2 Functions

### P.2.1 Global Function `pressflow_boundary_line_function()`

This calculates the time derivatives and constraint functions for the velocity-pressure form of the Navier-Stokes equations for an interior line. Continuity and  $x$ -motion equations are swapped to avoid zeroes on the Jacobian diagonal.

```
void pressflow_boundary_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, int side)
```

- `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 `pressparm` struct typedef, from which this gets needed parameters.
- `int side`                           Side on which to calculate the function values.

void `pressflow_interior_line_function` It returns nothing.

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  $-\sum \nabla \times (\phi_i \nabla \mu_i)$ .

The stagnation boundary condition is identical to the initial condition, with some special handling for pressure boundary conditions.

### P.2.2 Global Function `pressflow_first_setup()`

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

```
void pressflow_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 `pressparm` structure this inserts parameters from the command line.

### P.2.3 Global Function `pressflow_interior_line_function()`

This calculates the time derivatives and constraint functions for the velocity-pressure form of the Navier-Stokes equations for an interior line. Continuity and  $x$ -motion equations are swapped to avoid zeroes on the Jacobian diagonal.

```
void pressflow_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 `pressparm` struct typedef, from which this gets needed parameters.

The equations solved are given in dimensional form with non-unit density, starting with the equations of motion:

$$\frac{\partial u}{\partial t} = -\vec{u} \cdot \nabla u - \frac{1}{\rho} \frac{\partial p}{\partial x} - \nabla \cdot \tau_x + F_x. \quad (38)$$

$$\frac{\partial v}{\partial t} = -\vec{u} \cdot \nabla v - \frac{1}{\rho} \frac{\partial p}{\partial y} - \nabla \cdot \tau_y + F_y, \quad (39)$$

$$\frac{\partial w}{\partial t} = -\vec{u} \cdot \nabla w - \frac{1}{\rho} \frac{\partial p}{\partial z} - \nabla \cdot \tau_z + F_z. \quad (40)$$

Next there are two options for the pressure equation. If `PRESSURE_MOMENTUM_DIVERGENCE` is not defined, then we use the equation of continuity:

$$\nabla \cdot \vec{u} = 0. \quad (41)$$

The viscous stress divergence for a uniform-viscosity Newtonian fluid is the Laplacian of the velocity.

The elastic shear strain term is documented in `shearstrain.c` (appendix R on page R), and its divergence is used here. The symmetry of the shear strain tensor, and the incompressibility condition ( $\gamma_{xx} + \gamma_{yy} = 0$ ) reduce this to a function of just  $\gamma_{xx}$  and  $\gamma_{xy}$ .

This is also applied for Cahn-Hilliard with  $C$  taking the place of  $\phi$ .

With shear strain and phase field, we use an interpolation function  $p(\phi) = 3\phi^2 - 2\phi^3$  of the solid strength to weight the elastic and viscous stresses according to

$$\tau = p(\phi)\tau_{el} + (1 - p(\phi))\tau_{visc}. \quad (42)$$

With shear strain and no phase field, we can use this for incompressible viscoelastic mechanics by applying both the elastic and viscous stresses.

For Cahn-Hilliard, if `statsolid` is set, then add the force function developed by Tonhardt and Amberg: (insert function here).

Otherwise, add the body force due to interface curvature as described by Jacqmin given by  $-C\nabla\mu$  [10].

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 (density-normalized)  $x$ -driving force sinusoidal in  $y$  and starting at time  $t = t_0$  such that it can be 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.

#### P.2.4 Global Function `pressflow_interior_line_jacobian()`

This calculates the Jacobian of the equations corresponding to the velocity-pressure Navier-Stokes variables.

```
void pressflow_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. |

### P.2.5 Global Function `pressflow_labels_initcond()`

This sets up the field variable labels, uses command-line options to set the parameters in the `pressparm` struct typedef including the maximum stable explicit timestep size, and builds the initial condition for the velocity-pressure variables.

```
void pressflow_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 `pressparm` 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.

Membrane formation simulations use the Schmidt number ( $Sc$ , default 1) and a dimensionless force parameter ( $Fp$ , default  $10^9$ ) to scale the viscous and interfacial curvature fore terms respectively.

The initial condition is zero velocity with (optional) stagnation flow inward in the  $x$ -direction and outward in the  $y$ -direction (and  $z$  direction in 3-D). With symmetry boundary conditions, the stagnation point is at the origin, otherwise it is at the center of the grid. Because of the staggered mesh, the extra  $\Delta x/2$  shift needed for the `vortflow` module is unnecessary here.

### P.2.6 Global Function `pressflow_temp_parameters_boundary_line()`

This calculates vorticity along the boundary.

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

- `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 `pressparm` struct typedef, from which this gets needed parameters.
- `int side` Side on which to calculate the function values.

### P.2.7 Global Function `pressflow__temp__parameters__line()`

Though not needed for this module, the vorticity is helpful for others, so it is calculated here.

```
void pressflow_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. ‘ +html+ `<i>x</i>`-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 `pressparm` struct typedef, from which this gets needed parameters.

Vorticity is calculated at the corner of the cell, where it is best defined.

## Q File `pressflow.h`

RCS Header: `/cvsroot/rheoplast/pressflow.h,v 1.8 2005/12/27 00:56:02 zhou Exp`

The typedefs and prototypes for velocity-pressure 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 PRESSFLOW_H
```

```
#define PRESSFLOW_HELP "Velocity-pressure flow is done on a staggered mesh, see the
pressflow.c\ndocumentation for a complete description. To use it, add option:\n
-with-pressflow\ndand control it with properties and body force parameters:\n
-visibility <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]\n
-sinefortype <dirac,sine>
Sinusoidal force type [sine]\n
One may also specify initial and boundary conditions describing
stagnation\n
flow inward in the x-direction and centered at the origin (which turns on\n
null
symmetries) using:\n
-flow_stagnation <umax> Maximum stagnation velocity [0.0]\n\n"
```

## Q.1 Type definitions

### Q.1.1 Typedef AppCtx

```
typedef void AppCtx
```

### Q.1.2 Typedef pressparm

Structure typedef for velocity-pressure fluid flow.

```
typedef struct {...} pressparm
struct
{
 PetscScalar viscosity;
 PetscScalar density;
 PetscScalar sineFt0;
 PetscScalar sineFmax;
 PetscScalar uniFt0;
 PetscScalar uniFmax;
 PetscScalar stagnation;
 int uvar;
 int vvar;
 int wvar;
 int pressvar;
 int omegavar;
 int omeg2var;
 int omeg3var;
}
```

### Q.1.3 Typedef SFType

```
typedef enum {...} SFType
enum
{
 SINE_WAVE;
 DIRAC;
 UNIFORM;
}
```

## R File shearstrain.c

**RCS Header:** /cvsroot/rheoplast/shearstrain.c,v 1.52 2005/06/06 19:23:41 wanida Exp

This is an elastic shear strain module for rheoplast. It is here mostly for doing phase field/fluid-structure interaction simulations. The function comments describe the dynamics of this tensor field.

### Included Files

```
#include "rheoplast.h" (Section C)
#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 E)
 #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 G)
 #include </usr/lib/petsc/include/petsc.h>
#include "vortflow.h" (Section I)
 #include </usr/lib/petsc/include/petsc.h>
#include "membrane.h" (Section K)
 #include </usr/lib/petsc/include/petsc.h>
#include "vectorphase.h" (Section M)
 #include </usr/lib/petsc/include/petsc.h>
#include "heatcond.h" (Section O)
 #include </usr/lib/petsc/include/petsc.h>
#include "pressflow.h" (Section Q)
 #include </usr/lib/petsc/include/petsc.h>
#include "shearstrain.h" (Section S)
 #include </usr/lib/petsc/include/petsc.h>
#include "electra.h" (Section U)
 #include </usr/lib/petsc/include/petsc.h>
#include "chternary.h" (Section W)
 #include </usr/lib/petsc/include/petsc.h>
#include "cahnhill.h" (Section G)
#include "vortflow.h" (Section I)
#include "membrane.h" (Section K)
#include "vectorphase.h" (Section M)
#include "heatcond.h" (Section O)
#include "pressflow.h" (Section Q)
#include "shearstrain.h" (Section S)
#include "electra.h" (Section U)
#include "chternary.h" (Section W)

```

## Preprocessor definitions

```

#define __FUNCT__ "shearstrain_first_setup"
 #define __FUNCT__ "shearstrain_labels_initcond"
 #define gxx(point)
 #define gxy(point)
 #define gyy(point)
 #define gxxfunc(point)
 #define gxyfunc(point)
 #define gyyfunc(point)
 #define phi(point)
 #define C(point)

```

```

#define vu(point)
#define vv(point)
#define vomega(point)
#define pu(point)
#define pv(point)
#define pomega(point)
#define __FUNCT__ "shearstrain_temp_parameters_line"
#define __FUNCT__ "shearstrain_interior_line_function"

```

## R.1 Functions

### R.1.1 Global Function `shearstrain_first_setup()`

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

```

void shearstrain_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 `strainparm` structure this inserts parameters from the command line.

### R.1.2 Global Function `shearstrain_interior_line_function()`

This calculates the time derivatives  $\partial\gamma/\partial t$  for an interior line. For now it only functions in 2-D, and assumes incompressibility (then again, perhaps that is implicit in "shear strain"), such that  $\gamma_{xx} = -\gamma_{yy}$ . This module provides time derivatives to solve the equations

$$\frac{D}{Dt} \begin{pmatrix} \gamma_{xx} \\ \gamma_{xy} \end{pmatrix} = \begin{pmatrix} 2\partial u_x/\partial x \\ \partial u_x/\partial y + \partial u_y/\partial x \end{pmatrix} + \begin{pmatrix} -2\omega\gamma_{xy} \\ 2\omega\gamma_{xx} \end{pmatrix}. \quad (43)$$

This effectively gives three terms for the time derivative: one based on the non-rotational velocity gradient, one based on rotation of the material (and thus its strain rate) which uses the vorticity, and the third due to convection (which is implied by the substantial derivative).

```

void shearstrain_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 `strainparm` struct typedef, from which this gets needed parameters.

For velocity-vorticity flow, we use a staggered mesh with velocity and vorticity defined in the center of the cells and shear strain on the corners in order to prevent development of spurious modes. For  $\gamma_{xx}(= -\gamma_{yy})$ , to preserve symmetry, velocity derivatives in both directions are taken and their opposites averaged.

If vectorphase is in use, this treats the liquid equations ( $\phi < 0.1$ ) as constraint equations with the function equal to the value, to drive the shear strain there to zero.

If cahnhill is in use, this treats the liquid equations ( $C < 0.1$ ) as constraint equations with the function equal to the value, to drive the shear strain there to zero.

### R.1.3 Global Function `shearstrain_labels_initcond()`

This sets up the parameters in the `strainparm` struct typedef, field variable labels, maximum stable explicit `deltat`, and initial condition for the elastic shear strain variables.

```
void shearstrain_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 `strainparm` 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.

### R.1.4 Global Function `shearstrain_temp_parameters_line()`

There are no temporary field variables for elastic shear strain, so this does nothing.

```
void shearstrain_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 <code>strainparm</code> struct typedef, from which this gets needed parameters. |

## S File shearstrain.h

**RCS Header:** /cvsroot/rheoplast/shearstrain.h,v 1.17 2005/01/19 03:56:57 el\_oso Exp

The typedefs and prototypes for elastic shear strain. 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 SHEARSTRAIN_H

#define SHEARSTRAIN_HELP "Elastic shear strain is kinda complicated, and is used for
fluid-structure\ninteractions in phase field simulations. To use it, add option:\n
-with_shearstrain\nand control it with properties:\n -modulus <k> elastic shear modulus [1.0]\n\n"

#define SHEARSTRAIN_VORTFLOW_GXX_SHIFTX 1
#define SHEARSTRAIN_VORTFLOW_GXX_SHIFTY 0
#define SHEARSTRAIN_VORTFLOW_GXY_SHIFTX 1
#define SHEARSTRAIN_VORTFLOW_GXY_SHIFTY 1
#define SHEARSTRAIN_VORTFLOW_GYY_SHIFTX 0
#define SHEARSTRAIN_VORTFLOW_GYY_SHIFTY 0
```

## S.1 Type definitions

### S.1.1 Typedef AppCtx

```
typedef void AppCtx
```

### S.1.2 Typedef strainparm

Structure typedef for the shear strain tensor.

```
typedef struct {...} strainparm
struct
{
 PetscScalar modulus;
 int gamma_xxvar;
 int gamma_xyvar;
 int gamma_yyvar;
}
```

## T File electra.c

RCS Header: /cvsroot/rheoplast/electra.c,v 1.43 2006/03/06 18:11:48 wanida Exp

This is an electrostatic module for rheoplast. TO DO:

### Included Files

```
#include "rheoplast.h" (Section C)
#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 E)
#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 G)
#include </usr/lib/petsc/include/petsc.h>
#include "vortflow.h" (Section I)
#include </usr/lib/petsc/include/petsc.h>
#include "membrane.h" (Section K)
#include </usr/lib/petsc/include/petsc.h>
#include "vectorphase.h" (Section M)
#include </usr/lib/petsc/include/petsc.h>
#include "heatcond.h" (Section O)
#include </usr/lib/petsc/include/petsc.h>
#include "pressflow.h" (Section Q)
#include </usr/lib/petsc/include/petsc.h>
#include "shearstrain.h" (Section S)
#include </usr/lib/petsc/include/petsc.h>
#include "electra.h" (Section U)
#include </usr/lib/petsc/include/petsc.h>
#include "chternary.h" (Section W)
#include </usr/lib/petsc/include/petsc.h>
#include "cahnhill.h" (Section G)
#include "vortflow.h" (Section I)
#include "membrane.h" (Section K)
#include "vectorphase.h" (Section M)
#include "heatcond.h" (Section O)
#include "pressflow.h" (Section Q)
#include "shearstrain.h" (Section S)
#include "electra.h" (Section U)
#include "chternary.h" (Section W)
```

### Preprocessor definitions

```
#define __FUNCT__ "conductivity"
```

This function calculates electrical conductivity of each species.

```
#define __FUNCT__ "electra_first_setup"
```

```

#define __FUNCT__ "electra_labels_initcond"
#define V(point)
#define Vfunc(point)
#define sigma(point)
#define sigmaprime(point)
#define sigma_effprime(point)
#define zt_ti(point)
#define sigma_eff(point)
#define C(point)
#define u(point)
#define v(point)
#define C2(point)
#define C3(point)
#define Mu2(point)
#define Mu3(point)
#define __FUNCT__ "electra_temp_parameters_line"
#define __FUNCT__ "electra_temp_parameters_boundary_line"
#define __FUNCT__ "electra_interior_line_function"
#define __FUNCT__ "electra_boundary_line_function"
#define __FUNCT__ "electra_interior_line_jacobian"

```

## T.1 Functions

### T.1.1 Global Function `electra_boundary_line_function()`

This calculates the time derivatives and constraint functions for the conservation of charge equations for a boundary line. Note the `shearstrain` module interaction programmed here;

```

void electra_boundary_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, int side)

```

- `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.
  - `int side`
- `void cahnhill_interior_line_function` It returns nothing.  
Voltage is constant at `ymin` and `ymax`

### T.1.2 Global Function `electra_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 `echemparm` struct typedef.

```
void electra_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 `echemparm` structure this inserts parameters from the command line.

### T.1.3 Global Function `electra_interior_line_function()`

This evaluates the function of  $V$  Time derivative is zero and equal to

$$0 = \nabla \cdot (\sigma_{eff} \nabla V). \quad (44)$$

where  $\sigma_{eff}$  is the effective conductivity.

```
void electra_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`
- `AppCtx* data`

### T.1.4 Global Function `electra_interior_line_jacobian()`

This calculates the Jacobian of the equations corresponding to the electrical potential variables.

```
void electra_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)
```

- `PetscScalar* x` The field variables from which to evaluate the Jacobian.
- `PetscScalar* temp` Array of temporary field variables.
- `Mat J` Where to put the evaluated Jacobian.
- `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 firstrow` The matrix row number corresponding to the first point in the line.
- `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 `echemparm` struct typedef, from which this gets needed parameters.

First this calculates the coefficients in  $dF/dV$  part. The variable `jvalue` will hold the Jacobian values for insertion into the matrix. There are five non-zeroes per row (7 in 3-D). Additional `jvalue` from coupling variables are filled in each row corresponding to new column numbers.

### T.1.5 Global Function `electra_labels_initcond()`

This sets up the field variable labels, and initial condition for the potential variables.

```
void electra_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 `echemparm` structure this uses for various purposes.
- `PetscScalar* max_explicit_deltat` not used in this module

### T.1.6 Global Function `electra_temp_parameters_boundary_line()`

This just calculates conductivity on the boundary, since those values are needed for adjacent interior points.

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

### T.1.7 Global Function `electra_temp_parameters_line()`

This calculates the temporary parameter  $\sigma_{eff}(C)$ , which is the effective conductivity given by  $\sigma_{eff}(C) = z(C)/\tilde{z}(C) > \sigma_{Fe}(C) + \sigma_e(C)$ .

```
void electra_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 `chparm` struct typedef, from which this gets needed parameters.

## U File `electra.h`

RCS Header: `/cvsroot/rheoplast/electra.h,v 1.15 2006/02/13 05:34:03 wanida Exp`

The typedefs and prototypes for charge transport.

### Included Files

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

### Preprocessor definitions

```
#define ELECTRA_H
#define ELECTRA_HELP "Cahn-Hilliard charge transport is a basic staple of phase field
modeling.\nTo use it, add option:\n -with-electra\nand control it with the properties:\n -density
<rho>\n -temperature\n -diffusivity\n -molarmass\n\n"
```

### U.1 Type definitions

#### U.1.1 Typedef `AppCtx`

```
typedef void AppCtx
```

### U.1.2 Typedef echemparm

Structure typedef for charge transport.

```
typedef struct {...} echemparm
struct
{
 PetscScalar rhoM;
 PetscScalar voltage;
 PetscScalar sigma_slag;
 PetscScalar echem;
 PetscScalar upperV;
 PetscScalar lowerV;
 int Vvar;
 int sigma_effvar;
 int sigmavar;
 int sigma_primevar;
 int sigma_effprimevar;
 int zt_tivar;
}
```

## V File chternary.c

RCS Header: /cvsroot/rheoplast/chtternary.c,v 1.25 2006/03/06 19:37:36 wanida Exp

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

$$\mathcal{F} = \int \left( f(C_2, C_3) + \frac{1}{2} \sum_{i,j=2,3} K_{ij} \nabla C_i \cdot \nabla C_j \right) dV, \quad (45)$$

where  $f(C_2, C_3)$  is the homogeneous free energy, given as  $\frac{C_1}{m_1} * \ln(C_1) + \frac{C_2}{m_2} * \ln(C_2) + \frac{C_3}{m_3} * \ln(C_3) \chi_{12} * C_1 * C_2 + \chi_{23} * C_2 * C_3 + \chi_{13} * C_1 * C_3$ . Here, subscript 1 represents nonsolvent, 2 represents solvent and 3 represents polymer.  $C_1 = 1 - C_2 - C_3$ .

### Included Files

```
#include "rheoplast.h" (Section C)
#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 E)
 #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 G)
 #include </usr/lib/petsc/include/petsc.h>
#include "vortflow.h" (Section I)
 #include </usr/lib/petsc/include/petsc.h>
#include "membrane.h" (Section K)
 #include </usr/lib/petsc/include/petsc.h>
```

|                                           |             |
|-------------------------------------------|-------------|
| #include "vectorphase.h"                  | (Section M) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "heatcond.h"                     | (Section O) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "pressflow.h"                    | (Section Q) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "shearstrain.h"                  | (Section S) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "electra.h"                      | (Section U) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "chternary.h"                    | (Section W) |
| #include </usr/lib/petsc/include/petsc.h> |             |
| #include "cahnhill.h"                     | (Section G) |
| #include "vortflow.h"                     | (Section I) |
| #include "membrane.h"                     | (Section K) |
| #include "vectorphase.h"                  | (Section M) |
| #include "heatcond.h"                     | (Section O) |
| #include "pressflow.h"                    | (Section Q) |
| #include "shearstrain.h"                  | (Section S) |
| #include "electra.h"                      | (Section U) |
| #include "chternary.h"                    | (Section W) |

## Preprocessor definitions

```

#define __FUNCT__ "psiprime2"
 #define __FUNCT__ "psiprime3"
 #define __FUNCT__ "chternary_first_setup"
 #define __FUNCT__ "chternary_labels_initcond"
#define SMALL_PRIME 1571
#define MEDIUM_PRIME 524287
#define LARGE_PRIME 2147483647
#define MY_RANDOM(pseudo_random)
#define C2(point)
#define C3(point)
#define C2func(point)
#define C3func(point)
#define Mu2(point)
#define Mu3(point)
#define Psi(point)
#define u(point)
#define v(point)
#define sigma(point)
#define V(point)
#define pu(point)
#define pv(point)

```

```

#define pw(point)
#define __FUNCT__ "chternary_integrate_interior_line"
#define __FUNCT__ "chternary_temp_parameters_line"
#define __FUNCT__ "chternary_temp_parameters_boundary_line"
#define __FUNCT__ "chternary_interior_line_function"
#define __FUNCT__ "chternary_boundary_line_function"

```

## V.1 Functions

### V.1.1 Global Function `chternary_boundary_line_function()`

For now this assumes uniform  $\kappa$ .

```

void chternary_boundary_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, int side)

```

### V.1.2 Global Function `chternary_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 `chtparm` struct typedef.

```

void chternary_first_setup (PetscTruth threedee, int* vars, int* tempvars, int* intvars, 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* intvars`
- `int* stencilwid` Pointer to the stencil width.
- `AppCtx* data` Pointer to the `AppCtx` struct typedef, into whose `chtparm` structure this inserts parameters from the command line.

This sets mobility and  $\text{+latex+}\$K_{\{ij\}}\$$  `chternary_labels_initcond()`.

### V.1.3 Global Function `chternary_integrate_interior_line()`

This calculates the `chternary` integration variable  $\Psi$ , which is the free energy.

```

void chternary_integrate_interior_line (PetscScalar* x, PetscScalar* integ, 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* integ` Array with the integration variables.
- `int points` Number of points at which to calculate the integration 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 `chtparm` struct typedef, from which this gets needed parameters.

#### V.1.4 Global Function `chternary_interior_line_function()`

This calculates the time derivative of  $C_2, C_3$  using the divergence of flux, which goes down the gradient of chemical potential given by equation 47. That time derivative is thus given by

$$\frac{\partial C_2}{\partial t} = \nabla \cdot (\kappa \nabla \mu). \quad (46)$$

where  $\kappa$  is the mobility.

```
void chternary_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 `chtparm` struct typedef, from which this gets needed parameters.

For now this assumes uniform  $\kappa$ .

#### V.1.5 Global Function `chternary_labels_initcond()`

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

```
void chternary_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 `chtparm` 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  $C2, C3$  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 `-cht_random_center_C_s` or `-cht_random_center_C_p` and `-cht_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 `-cht_layers` option, whose argument is the fraction of the domain which will be in the "bottom"  
The `-cht_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 `-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.

### V.1.6 Global Function `chternary_temp_parameters_boundary_line()`

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

### V.1.7 Global Function `chternary_temp_parameters_line()`

This calculates the chternary temporary parameter  $\mu_s, \mu_p$ , which is the chemical potential given as the following:

$$\mu_i = \frac{\delta f}{\delta C_i} - K_{ii} \nabla^2 C_i - \frac{1}{2} (K_{ij} + K_{ji}) \nabla^2 C_j \quad (47)$$

```
void chternary_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 <code>chtparm</code> struct typedef, from which this gets needed parameters. |

### V.1.8 Local Function psi()

This abstracts out the function for  $\Psi(C_2, C_3)$ , PetscScalar psi It returns the homogeneous free energy.

```
static inline PetscScalar psi (PetscScalar C2, PetscScalar C3, chtparm* thechternary)
```

- PetscScalar C2
- PetscScalar C3
- chtparm\* thechternary                    chternary parameter structure.

PetscScalar C The  $C_2, C_3$  parameter it's a function of.

### V.1.9 Local Function psiprime2()

This abstracts out the function for  $\Psi'_2(C_2, C_3)$ , the derivative of homogeneous free energy, so it can be easily modified. Since  $f(C_2, C_3) = C_1 * \ln(C_1) + C_2 * \ln(C_2) + \frac{C_3}{m} * \ln(C_3) + \chi_{12} * C_1 * C_2 + \chi_{23} * C_2 * C_3 + \chi_{13} * C_1 * C_3$  if  $m_1, m_2$  are chosen as 1 and  $m_3$  is chosen as  $m$ , this returns  $-\log(1.0 - C_2 - C_3) + \log(C_2) + \chi_{12} * (1.0 - 2.0 * C_2 - C_3) + \chi_{23} * C_3 - \chi_{13} * C_3$ .

```
static inline PetscScalar psiprime2 (PetscScalar C2, PetscScalar C3, chtparm* thechternary)
```

- PetscScalar C2
- PetscScalar C3
- chtparm\* thechternary                    chternary parameter structure.

PetscScalar psiprime It returns the derivative of homogeneous free energy.

PetscScalar C The  $C_2, C_3$  parameter it's a function of.

### V.1.10 Local Function psiprime3()

This abstracts out the function for  $\Psi'_3(C_2, C_3)$ , the derivative of homogeneous free energy, so it can be easily modified. Since  $f(C_2, C_3) = C_1 * \ln(C_1) + C_2 * \ln(C_2) + \frac{C_3}{m} * \ln(C_3) + \chi_{12} * C_1 * C_2 + \chi_{23} * C_2 * C_3 + \chi_{13} * C_1 * C_3$  if  $m_1, m_2$  are chosen as 1 and  $m_3$  is chosen as  $m$ , this returns  $-1.0 - \log(1 - C_2 - C_3) + \frac{1.0}{m} + \frac{1.0}{m} * \log(C_3) - \chi_{12} * C_2 + \chi_{23} * C_2 - \chi_{13} * (1.0 - 2.0 * C_3 - C_2)$

```
static inline PetscScalar psiprime3 (PetscScalar C2, PetscScalar C3, chtparm* thechternary)
```

- PetscScalar C2
- PetscScalar C3
- chtparm\* thechternary                    chternary parameter structure.

PetscScalar psiprime It returns the derivative of homogeneous free energy.  
PetscScalar C The  $C_2, C_3$  parameter it's a function of.

## W File chternary.h

RCS Header: /cvsroot/rheoplast/chternary.h,v 1.7 2006/03/06 19:37:36 wanida Exp

The typedefs and prototypes for ternary Cahn-Hilliard species transport.

### Included Files

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

### Preprocessor definitions

```
#define CHTERNARY_H
```

```
#define CHTERNARY_HELP "Ternary Cahn-Hilliard species transport is provided by the chternary module.\n\nTo use it, add option:\n -with-chternary\n\nand control it with the properties:\n -mobility_22 mobility\n -mobility_23 mobility\n -mobility_32 mobility\n -mobility_33 mobility\n -K_22 K\n -K_23 K\n -K_32 K\n -K_33 K\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 -cht_particles\n\nOr one can specify a two-layer system in the y-direction using:\n -cht_layers\n\nChoose two-layer initial condition. \n -cht_layer_thickness <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 -cht_random_center_C2 <center>\n\ncenter of random distribution of C2 (required)\n -cht_random_center_C3 <center>\n\ncenter of random distribution of C3 (required)\n -cht_random_fluct <fluct>\n\nhalf-width of uniform distribution [0.005]\n\n"
```

## W.1 Type definitions

### W.1.1 Typedef AppCtx

```
typedef void AppCtx
```

### W.1.2 Typedef chtparm

Structure typedef for ternary Cahn-Hilliard species transport.

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

```

 PetscScalar ForceTerm;
 int C2var;
 int C3var;
 int Mu2var;
 int Mu3var;
 int psivar;
 PetscTruth polymer_PVDF;
 PetscTruth metal;
}

```

## X 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 "0.8.9"

```