Chapter 1

The Bretherton problem: An air finger propagates into a 2D channel

This tutorial discusses the solution of the classical Bretherton problem in which an air-finger is driven steadily into a 2D fluid-filled, rigid-walled channel.

The driver code listed below was originally developed as a test-bed for the paper

Hazel, A.L. & Heil, M. (2008) The influence of gravity on the steady propagation of a semi-infinite bubble into a flexible channel. Physics of Fluids 20, 092109 (pdf reprint)

in which we studied the elastic-walled equivalent, paying particular attention to the effect of transverse gravity which plays quite an important role in this problem.

Compared to other tutorials, there is no detailed discussion of the driver code itself because the implementation is somewhat messy. However, given that the driver code is (of course!) very well documented you should be able to figure out what's going on once you've read through the discussion of the problem formulation and our brief comments on the Implementation. Get in touch if you need more information.

1.1 The problem

The sketch below shows the problem setup: An (inviscid) air finger propagates at a steady speed U into a 2D rigid-walled, fluid-filled channel of half-width H_0^*. In the process it displaces some of the viscous fluid (of density ρ, viscosity μ and surface tension σ^*) and deposits a film of thickness h_0^* on the channel walls. [Note that, for the sake of simplicity, we ignore the effect of transverse gravity in this discussion; the driver code listed below allows the specification of a gravitational body force which will break the up-down symmetry of the solution.]

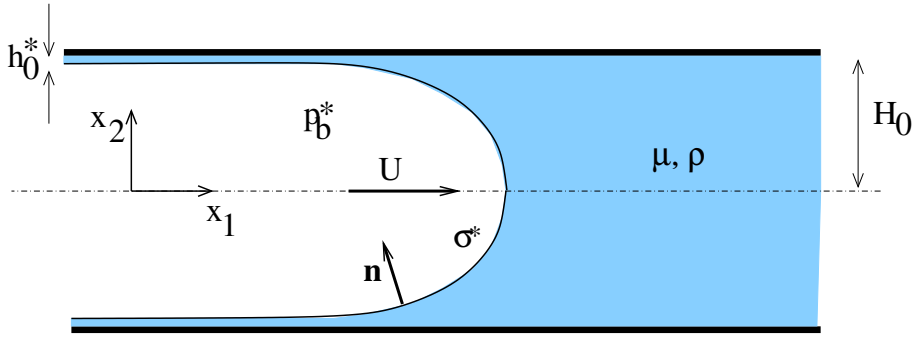

Figure 1.1 Sketch of the problem setup

We non-dimensionalise the velocities on U, the lengths on H_0^* and the pressure on the viscous scale, $\mu U/H_0^*$ and solve the problem in a frame moving with the velocity of the finger. The flow is then governed by the non-dimensional Navier-Stokes equations

$$Re\, u_j \frac{\partial u_i}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$$

and the continuity equation

$$\frac{\partial u_i}{\partial x_i} = 0,$$

where the Reynolds number is defined as $Re = UH_0\rho/\mu$. On the free fluid surface, whose outer unit normal we denote by $\mathbf{n}$, the fluid normal velocity vanishes,

$$\mathbf{u}\cdot\mathbf{n} = 0 \quad \text{on the air-liquid interface.}$$

We measure the fluid pressure p relative to the bubble pressure p_b. Then the dynamic boundary condition implies that

$$-pn_i + \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) n_j + \frac{1}{Ca}\,\kappa_f n_i = 0 \quad \text{on the air-liquid interface,}$$

where $\kappa_f = \kappa_f^* H_0$ is the non-dimensional interface curvature and the capillary number $Ca = U\mu/\sigma^*$ is a non-dimensional measure of the bubble velocity. The no-slip condition on the channel walls requires that $\mathbf{u} = (-1, 0)$ at $x_2 = \pm 1$. Far behind the bubble tip, the fluid remains stationary on the walls. In the moving frame of reference, this corresponds to the plug flow profile $\mathbf{u} = (-1, 0)$ as $x_1 \to -\infty$. The residual film thickness h_0 has to be determined as part of the solution and it determines the volume flux through the system. The Poiseuille velocity profile far ahead of the bubble tip,

$$\mathbf{u} = (-1 + \frac{3}{2}(1-h_0)(1-x_2^2), 0) \quad \text{as } x_1 \to \infty, \qquad (1)$$

is then determined by overall continuity.

1.2 Some results

Here are some pretty pictures of the computed flow field,

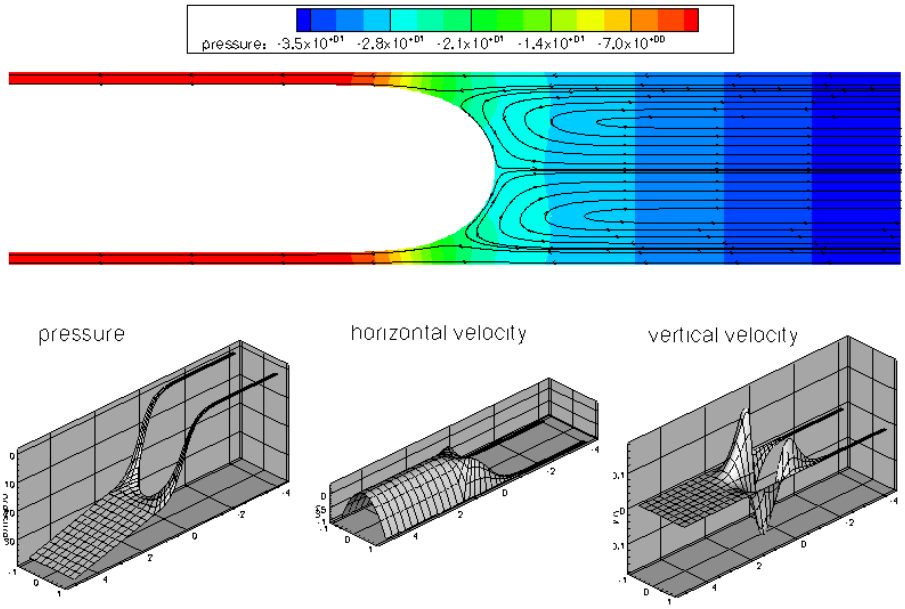

Figure 1.2 Velocity and pressure fields

and here is a comparison of the computational predictions for the bubble pressure and film thickness against Bretherton's famous asymptotic predictions (valid only for small capillary numbers!):

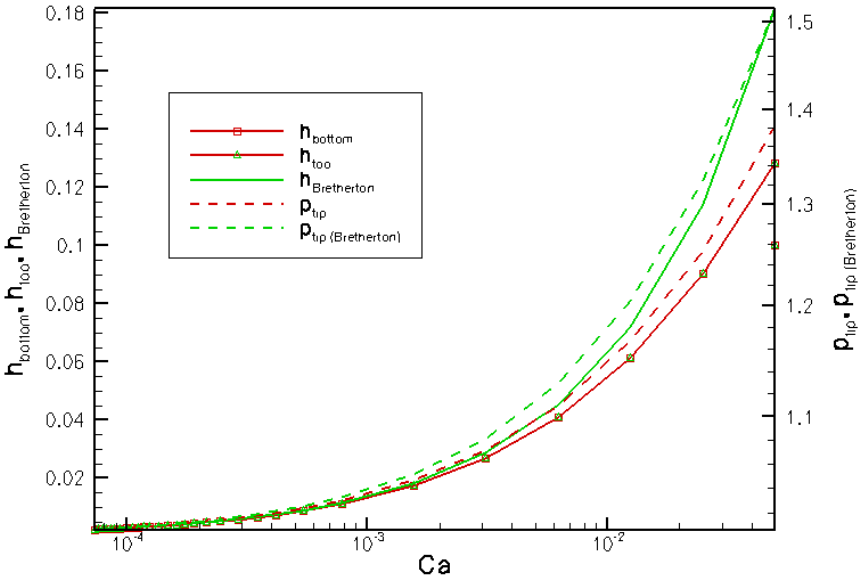

Figure 1.3 Bubble pressure and film thickness far behind the finger tip vs. capillary number; computed solution and Bretherton's predictions for small Ca.

1.3 Implementation

The discretisation of the problem is reasonably straightforward, the most (human-)time-consuming part being the creation of the spine mesh, discussed in another tutorial. The real complication arises from the fact that the application of the "inflow condition" (1) at the right end of the domain – superficially a Dirichlet condition for the velocity – depends on the non-dimensional film thickness h_0 at the left end of the domain. This introduces a non-local interaction between these degrees of freedom. We handle this by providing a templated wrapper class BrethertonElement (templated by the type of the "wrapped" fluid element) which allows the specification of the film thickness as external Data (i.e. Data whose values affect the element's residuals but are not determined by the element; see the discussion of oomph-lib's various types of elemental Data in the "bottom up" discussion of the library's data structure). Read the driver code listed below to see how it works!

1.4 The driver code

```
//LIC// ====================================================================
//LIC// This file forms part of oomph-lib, the object-oriented,
//LIC// multi-physics finite-element library, available
//LIC// at http://www.oomph-lib.org.
//LIC//
//LIC// Copyright (C) 2006-2021 Matthias Heil and Andrew Hazel
//LIC//
//LIC// This library is free software; you can redistribute it and/or
//LIC// modify it under the terms of the GNU Lesser General Public
//LIC// License as published by the Free Software Foundation; either
//LIC// version 2.1 of the License, or (at your option) any later version.
//LIC//
//LIC// This library is distributed in the hope that it will be useful,
//LIC// but WITHOUT ANY WARRANTY; without even the implied warranty of
//LIC// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
//LIC// Lesser General Public License for more details.
//LIC//
//LIC// You should have received a copy of the GNU Lesser General Public
//LIC// License along with this library; if not, write to the Free Software
//LIC// Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
```

```
//LIC// 02110-1301  USA.
//LIC//
//LIC// The authors may be contacted at oomph-lib@maths.man.ac.uk.
//LIC//
//LIC//====================================================================
// The 2D Bretherton problem
#include<algorithm>

// The oomphlib headers
#include "generic.h"
#include "navier_stokes.h"
#include "fluid_interface.h"
// The mesh
#include "meshes/bretherton_spine_mesh.h"
using namespace std;

using namespace oomph;
//====================================================================
/// Namepspace for global parameters
//====================================================================
namespace Global_Physical_Variables
{

 /// Reynolds number
 double Re;

 /// Womersley = Reynolds times Strouhal
 double ReSt;

 /// Product of Reynolds and Froude number
 double ReInvFr;

 /// Capillary number
 double Ca;

 /// Direction of gravity
 Vector<double> G(2);

 /// Pointer to film thickness at outflow on the lower wall
 double* H_lo_pt;

 /// Pointer to film thickness at outflow on the upper wall
 double* H_up_pt;

 /// Pointer to y-position at inflow on the lower wall
 double* Y_lo_pt;

 /// Pointer to y-position at inflow on the upper wall
 double* Y_up_pt;

 /// Set inflow velocity, based on spine heights at outflow
 /// and channel width at inflow
 void inflow(const Vector<double>& x, Vector<double>& veloc)
 {
#ifdef PARANOID
  std::ostringstream error_stream;
  bool throw_error=false;
  if (H_lo_pt==0)
   {
    error_stream << "You must set H_lo_pt\n";
    throw_error = true;
   }
  if (H_up_pt==0)
   {
    error_stream << "You must set H_up_pt\n";
    throw_error = true;
   }
  if (Y_lo_pt==0)
   {
    error_stream << "You must set Y_lo_pt\n";
    throw_error = true;
   }
  if (Y_up_pt==0)
   {
    error_stream << "You must set Y_up_pt\n";
    throw_error = true;
   }
  //If one of the errors has occured
  if(throw_error)
   {
    throw OomphLibError(error_stream.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif
  // Get average film thickness far ahead
  double h_av=0.5*(*H_lo_pt+*H_up_pt);
```

```
  // Get position of upper and lower wall at inflow
  double y_up=*Y_up_pt;
  double y_lo=*Y_lo_pt;
  // Constant in velocity profile
  double C =6.0*(2.0*h_av+y_lo-y_up)/
   (y_up*y_up*y_up-y_lo*y_lo*y_lo-h_av*y_up*
    y_up*y_up+h_av*y_lo*y_lo*y_lo-3.0*y_lo*y_up*y_up+
    3.0*y_lo*y_lo*y_up+3.0*y_lo*y_up*y_up*h_av-3.0*y_lo*y_lo*y_up*h_av);
  // y coordinate
  double y=x[1];
  // Parallel, parabolic inflow
  veloc[0]=-1.0+C*(1.0-h_av)*((y_lo-y)*(y_up-y));
  veloc[1]=0.0;
 }
}
//========================================================================
/// "Bretherton element" is a fluid element (of type ELEMENT) for which
/// the (inflow) velocity  at those nodes that are located on a
/// specified Mesh boundary is prescribed by Dirichlet boundary
/// conditions. The key is that prescribed velocity profile can be
/// a function of some external Data -- this dependency must
/// be taken into account when computing the element's Jacobian
/// matrix.
///
/// This element type is useful, for instance, in the Bretherton
/// problem, where the parabolic "inflow" profile is a function
/// of the film thickness (represented by Spine heights) at the
/// "outflow".
//========================================================================
template<class ELEMENT>
class BrethertonElement : public ELEMENT
{
public:

 /// \short Typedef for pointer (global) function that specifies the
 /// the inflow
 typedef void (*InflowFctPt)(const Vector<double>& x, Vector<double>& veloc);

 /// Constructor: Call the constructor of the underlying element
 BrethertonElement() : ELEMENT() {}

 /// \short Activate the dependency of the "inflow" on the external
 /// data. Pass the vector of pointers to the external Data that affects
 /// the inflow, the id of the boundary on which the inflow
 /// condition is to be applied and the function pointer to
 /// to the global function that defines the inflow
 void activate_inflow_dependency_on_external_data(
  const Vector<Data*>& inflow_ext_data,
  const unsigned& inflow_boundary,
  InflowFctPt inflow_fct_pt)
  {
   // Copy data that affects the inflow
   unsigned n_ext=inflow_ext_data.size();
   Inflow_ext_data.resize(n_ext);
   for (unsigned i=0;i<n_ext;i++)
    {
     Inflow_ext_data[i]=inflow_ext_data[i];
    }
   // Set inflow boundary
   Inflow_boundary=inflow_boundary;
   // Set fct pointer to inflow condition
   Inflow_fct_pt=inflow_fct_pt;
  }


 /// short Overload assign local equation numbers: Add the dependency
 /// on the external Data that affects the inflow profile
 void assign_local_eqn_numbers(const bool  &store_local_dof_pt)
  {
   // Call the element's local equation numbering procedure first
   ELEMENT::assign_local_eqn_numbers(store_local_dof_pt);

   // Now add the equation numbers for the Data that affects the inflow
   // profile

   // Number of local equations so far
   unsigned local_eqn_count = this->ndof();

   // Find out max. number of values stored at all Data values
   // that affect the inflow
   unsigned max_nvalue=0;
   unsigned n_ext=Inflow_ext_data.size();
   for (unsigned i=0;i<n_ext;i++)
    {
     // The external Data:
     Data* data_pt=Inflow_ext_data[i];
     // Number of values
```

```
    unsigned n_val=data_pt->nvalue();
    if (n_val>max_nvalue) max_nvalue=n_val;
   }
  // Allocate sufficient storage
  Inflow_ext_data_eqn.resize(n_ext,max_nvalue);

  //A local queue to store the global equation numbers
  std::deque<unsigned long> global_eqn_number_queue;
  // Loop over external Data that affect the "inflow"
  for (unsigned i=0;i<n_ext;i++)
   {
    // The external Data:
    Data* data_pt=Inflow_ext_data[i];

    // Loop over number of values:
    unsigned n_val=data_pt->nvalue();
    for (unsigned ival=0;ival<n_val;ival++)
     {
      // Is it free or pinned?
      long eqn_number=data_pt->eqn_number(ival);
      if (eqn_number>=0)
       {
        // Add global equation number to local storage
        global_eqn_number_queue.push_back(eqn_number);
        // Store local equation number associated with this external dof
        Inflow_ext_data_eqn(i,ival)=local_eqn_count;
        // Increment counter for local dofs
        local_eqn_count++;
       }
      else
       {
        Inflow_ext_data_eqn(i,ival)=-1;
       }
     }
   }

  //Now add our global equations numbers to the internal element storage
  this->add_global_eqn_numbers(global_eqn_number_queue,
                               GeneralisedElement::Dof_pt_deque);
 }

/// Overloaded Jacobian computation: Computes the Jacobian
/// of the underlying element and then adds the FD
/// operations to evaluate the derivatives w.r.t. the Data values
/// that affect the inflow.
void get_jacobian(Vector<double>& residuals,
                  DenseMatrix<double>& jacobian)
 {
  // Loop over Data values that affect the inflow
  unsigned n_ext=Inflow_ext_data.size();
  // Call "normal" jacobian first.
  ELEMENT::get_jacobian(residuals,jacobian);

  if (n_ext==0) return;

  // Get ready for FD operations
  Vector<double> residuals_plus(residuals.size());
  double fd_step=1.0e-8;
  // Loop over Data values that affect the inflow
  for (unsigned i=0;i<n_ext;i++)
   {
    // Get Data item
    Data* data_pt=Inflow_ext_data[i];

    // Loop over values
    unsigned n_val=data_pt->nvalue();
    for (unsigned ival=0;ival<n_val;ival++)
     {
      // Local equation number
      int local_eqn=Inflow_ext_data_eqn(i,ival);

      // Dof or pinned?
      if (local_eqn>=0)
       {
        //get pointer to the data value
        double *value_pt = data_pt->value_pt(ival);
        //Backup Data value
        double backup = *value_pt;

        // Do FD step
        *value_pt += fd_step;

        // Re-assign the inflow velocities for nodes in this element
        reassign_inflow();


        // Fill in the relevant column in the Jacobian matrix
```

```
         unsigned n_dof = this->ndof();
         //Zero the residuals
         for(unsigned idof=0;idof<n_dof;idof++) {residuals_plus[idof] = 0.0;}
         // Re-compute the element residual
         this->get_residuals(residuals_plus);
         for(unsigned idof=0;idof<n_dof;idof++)
          {
           jacobian(idof,local_eqn)=
            (residuals_plus[idof]-residuals[idof])/fd_step;
          }

         //Reset spine height
         *value_pt = backup;

        }
       // Note: Re-assignment of inflow is done on the fly during next loop
      }
    }

   // Final re-assign for the inflow velocities for nodes in this element
   reassign_inflow();

  }


private:


 /// \short For all nodes that are located on specified boundary
 /// re-assign the inflow velocity, using the function pointed to
 /// by the function pointer
 void reassign_inflow()
  {
   // Loop over all nodes in element -- if they are
   // on inflow boundary, re-assign their velocities
   Vector<double> x(2);
   Vector<double> veloc(2);
   unsigned n_nod = this->nnode();
   for (unsigned j=0;j<n_nod;j++)
    {
     Node* nod_pt = this->node_pt(j);
     if(nod_pt->is_on_boundary(Inflow_boundary))
      {
#ifdef PARANOID
           for (unsigned i=0;i<2;i++)
            {
             if (nod_pt->eqn_number(0)>=0)
              {
               std::ostringstream error_stream;
               error_stream
                << "We're assigning a Dirichlet condition for the "
                << i << "-th "
                << "velocity, even though it is not pinned!\n"
                << "This can't be right! I'm bailing out..."
                << std::endl;
               throw OomphLibError(error_stream.str(),
                                   OOMPH_CURRENT_FUNCTION,
                                   OOMPH_EXCEPTION_LOCATION);
              }
            }
#endif
           // Get inflow profile
           x[0]=nod_pt->x(0);
           x[1]=nod_pt->x(1);
           Inflow_fct_pt(x,veloc);
           nod_pt->set_value(0,veloc[0]);
           nod_pt->set_value(1,veloc[1]);
      }
    }
  }


 /// Storage for the external Data that affects the inflow
 Vector<Data*> Inflow_ext_data;

 /// \short Storage for the local equation numbers associated the Data
 /// values that affect the inflow
 DenseMatrix<int> Inflow_ext_data_eqn;

 /// Number of the inflow boundary in the global mesh
 unsigned Inflow_boundary;

 /// \short Function pointer to the global function that specifies the
 /// inflow velocity profile on the global mesh boundary Inflow_boundary
 InflowFctPt Inflow_fct_pt;
};
// Note: Specialisation must go into namespace!
```

```
namespace oomph
{
//======================================================================
/// Face geometry of the Bretherton 2D Crouzeix_Raviart spine elements
//======================================================================
template<>
class FaceGeometry<BrethertonElement<SpineElement<QCrouzeixRaviartElement<2> > > >: public virtual
       QElement<1,3>
{
  public:
 FaceGeometry() : QElement<1,3>() {}
};
//======================================================================
/// Face geometry of the Bretherton 2D Taylor Hood spine elements
//======================================================================
template<>
class FaceGeometry<BrethertonElement<SpineElement<QTaylorHoodElement<2> > > >: public virtual QElement<1,3>
{
  public:
 FaceGeometry() : QElement<1,3>() {}
};
//======================================================================
/// Face geometry of the Face geometry of the
///  the Bretherton 2D Crouzeix_Raviart spine elements
//======================================================================
template<>
class FaceGeometry<FaceGeometry<BrethertonElement<
 SpineElement<QCrouzeixRaviartElement<2> > > > >: public virtual PointElement
{
  public:
 FaceGeometry() : PointElement() {}
};
//======================================================================
/// Face geometry of face geometry of
/// the Bretherton 2D Taylor Hood spine elements
//======================================================================
template<>
class FaceGeometry<FaceGeometry<BrethertonElement<SpineElement<QTaylorHoodElement<2> > > > >: public virtual
       PointElement
{
  public:
 FaceGeometry() : PointElement() {}
};
}

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

//======================================================================
/// Bretherton problem
//======================================================================
template<class ELEMENT>
class BrethertonProblem : public Problem
{
public:

 /// Constructor:
 BrethertonProblem();

 /// \short Spine heights/lengths are unknowns in the problem so their
 /// values get corrected during each Newton step. However,
 /// changing their value does not automatically change the
 /// nodal positions, so we need to update all of them
 void actions_before_newton_convergence_check()
  {
   // Update
   Bulk_mesh_pt->node_update();
   // Apply inflow on boundary 1
   unsigned ibound=1;
   unsigned num_nod=mesh_pt()->nboundary_node(ibound);

   // Coordinate and velocity
   Vector<double> x(2);
   Vector<double> veloc(2);
   // Loop over all nodes
   for (unsigned inod=0;inod<num_nod;inod++)
    {
     // Get nodal position
     x[0]=mesh_pt()->boundary_node_pt(ibound,inod)->x(0);
     x[1]=mesh_pt()->boundary_node_pt(ibound,inod)->x(1);
     // Get inflow profile
     Global_Physical_Variables::inflow(x,veloc);
     mesh_pt()->boundary_node_pt(ibound,inod)->set_value(0,veloc[0]);
     mesh_pt()->boundary_node_pt(ibound,inod)->set_value(1,veloc[1]);
    }
  }
```

```
 /// \short Update before solve: empty
 void actions_before_newton_solve() {}

 /// \short Update after solve can remain empty, because the update
 /// is performed automatically after every Newton step.
 void actions_after_newton_solve() {}

 ///Fix pressure value l in element e to value p_value
 void fix_pressure(const unsigned &e, const unsigned &l,
                   const double &pvalue)
  {
   //Fix the pressure at that element
   dynamic_cast<ELEMENT *>(Bulk_mesh_pt->element_pt(e))->
    fix_pressure(l,pvalue);
  }

 /// \short Activate the dependency of the inflow velocity on the
 /// spine heights at the outflow
 void activate_inflow_dependency();

 /// Run a parameter study; perform specified number of steps
 void parameter_study(const unsigned& nsteps);

 /// Doc the solution
 void doc_solution(DocInfo& doc_info);
private:

 /// Pointer to control element
 ELEMENT* Control_element_pt;

 /// Trace file
 ofstream Trace_file;

 /// Pointer to bulk mesh
 BrethertonSpineMesh<ELEMENT,SpineLineFluidInterfaceElement<ELEMENT> >*
 Bulk_mesh_pt;
};
//==========================================================================
/// Problem constructor
//==========================================================================
template<class ELEMENT>
BrethertonProblem<ELEMENT>::BrethertonProblem()
{
 // Number of elements in the deposited film region
 unsigned nx1=24;
 // Number of elements in the bottom part of the transition region
 unsigned nx2=6;
 // Number of elements in channel region
 unsigned nx3=12;
 // Number of elements in the vertical part of the transition region
 // (=half the number of elements in the liquid filled region ahead
 // of the finger tip)
 unsigned nhalf=4;
 // Number of elements through thickness of deposited film
 unsigned nh=3;

 // Thickness of deposited film
 double h=0.1;
 // Create wall geom objects
 GeomObject* lower_wall_pt=new StraightLine(-1.0);
 GeomObject* upper_wall_pt=new StraightLine( 1.0);
 // Start coordinate on wall
 double xi0=-4.0;
 // End of transition region on wall
 double xi1=1.5;
 // End of liquid filled region (inflow) on wall
 double xi2=5.0;
 //Now create the mesh
 Bulk_mesh_pt = new  BrethertonSpineMesh<ELEMENT,
  SpineLineFluidInterfaceElement<ELEMENT> >
  (nx1,nx2,nx3,nh,nhalf,h,lower_wall_pt,upper_wall_pt,xi0,0.0,xi1,xi2);
 // Make bulk mesh the global mesh
 mesh_pt()=Bulk_mesh_pt;
 // Store the control element
 Control_element_pt=Bulk_mesh_pt->control_element_pt();
 // Set pointers to quantities that determine the inflow profile
 // Film thickness at outflow on lower wall:
 Global_Physical_Variables::H_lo_pt=
  Bulk_mesh_pt->spine_pt(0)->spine_height_pt()->value_pt(0);
 // Film thickness at outflow on upper wall:
 unsigned last_spine=Bulk_mesh_pt->nfree_surface_spines()-1;
 Global_Physical_Variables::H_up_pt=
  Bulk_mesh_pt->spine_pt(last_spine)->spine_height_pt()->value_pt(0);
 // Current y-position on lower wall at inflow
 unsigned ibound=1;
 Global_Physical_Variables::Y_lo_pt=
```

```
 Bulk_mesh_pt->boundary_node_pt(ibound,0)->x_pt(0,1);
// Current y-position on upper wall at inflow
unsigned nnod=Bulk_mesh_pt->nboundary_node(ibound);
Global_Physical_Variables::Y_up_pt=
 Bulk_mesh_pt->boundary_node_pt(ibound,nnod-1)->x_pt(0,1);
// Activate dependency of inflow on spine heights at outflow
activate_inflow_dependency();
// Set the boundary conditions for this problem: All nodes are
// free by default -- just pin the ones that have Dirichlet conditions
// here
// No slip on boundaries 0 1 and 2
for(unsigned ibound=0;ibound<=2;ibound++)
 {
  unsigned num_nod=mesh_pt()->nboundary_node(ibound);
  for (unsigned inod=0;inod<num_nod;inod++)
   {
    mesh_pt()->boundary_node_pt(ibound,inod)->pin(0);
    mesh_pt()->boundary_node_pt(ibound,inod)->pin(1);
   }
 }
// Uniform, parallel outflow on boundaries 3 and 5
for(unsigned ibound=3;ibound<=5;ibound+=2)
 {
  unsigned num_nod=mesh_pt()->nboundary_node(ibound);
  for (unsigned inod=0;inod<num_nod;inod++)
   {
    mesh_pt()->boundary_node_pt(ibound,inod)->pin(0);
    mesh_pt()->boundary_node_pt(ibound,inod)->pin(1);
   }
 }
// Pin central spine
unsigned central_spine=(Bulk_mesh_pt->nfree_surface_spines()-1)/2;
Bulk_mesh_pt->spine_pt(central_spine)->spine_height_pt()->pin(0);
// No slip in moving frame on boundaries 0 and 2
for (unsigned ibound=0;ibound<=2;ibound+=2)
 {
  unsigned num_nod=mesh_pt()->nboundary_node(ibound);
  for (unsigned inod=0;inod<num_nod;inod++)
   {
    // Parallel flow
    mesh_pt()->boundary_node_pt(ibound,inod)->set_value(0,-1.0);
    mesh_pt()->boundary_node_pt(ibound,inod)->set_value(1, 0.0);
   }
 }

// Parallel, uniform outflow on boundaries 3 and 5
for (unsigned ibound=3;ibound<=5;ibound+=2)
 {
  unsigned num_nod=mesh_pt()->nboundary_node(ibound);
  for (unsigned inod=0;inod<num_nod;inod++)
   {
    // Parallel inflow
    mesh_pt()->boundary_node_pt(ibound,inod)->set_value(0,-1.0);
    mesh_pt()->boundary_node_pt(ibound,inod)->set_value(1, 0.0);
   }
 }

//Complete the problem setup to make the elements fully functional
//Loop over the elements in the layer
unsigned n_bulk=Bulk_mesh_pt->nbulk();
for(unsigned i=0;i<n_bulk;i++)
 {
  //Cast to a fluid element
  ELEMENT *el_pt = dynamic_cast<ELEMENT*>(Bulk_mesh_pt->
                                          bulk_element_pt(i));
  //Set the Reynolds number, etc
  el_pt->re_pt() = &Global_Physical_Variables::Re;
  el_pt->re_st_pt() = &Global_Physical_Variables::ReSt;
  el_pt->re_invfr_pt() = &Global_Physical_Variables::ReInvFr;
  el_pt->g_pt() = &Global_Physical_Variables::G;
 }
//Loop over 1D interface elements and set capillary number
unsigned interface_element_pt_range = Bulk_mesh_pt->ninterface_element();
for(unsigned i=0;i<interface_element_pt_range;i++)
 {
  //Cast to a interface element
  SpineLineFluidInterfaceElement<ELEMENT>* el_pt =
   dynamic_cast<SpineLineFluidInterfaceElement<ELEMENT>*>
   (Bulk_mesh_pt->interface_element_pt(i));
  //Set the Capillary number
  el_pt->ca_pt() = &Global_Physical_Variables::Ca;
 }
//Update the nodal positions, so that the domain is correct
//(and therefore the repeated node test passes)
Bulk_mesh_pt->node_update();
//Do equation numbering
cout « "Number of unknowns: " « assign_eqn_numbers() « std::endl;
```

```
}
//========================================================================
/// Activate the dependency of the inflow velocity on the
/// spine heights at the outflow
//========================================================================
template<class ELEMENT>
void BrethertonProblem<ELEMENT>::activate_inflow_dependency()
{
 // Spine heights that affect the inflow
 Vector<Data*> outflow_spines(2);
 // First spine
 outflow_spines[0]=Bulk_mesh_pt->spine_pt(0)->spine_height_pt();
 // Last proper spine
 unsigned last_spine=Bulk_mesh_pt->nfree_surface_spines()-1;
 outflow_spines[1]=Bulk_mesh_pt->spine_pt(last_spine)->spine_height_pt();;

 /// Loop over elements on inflow boundary (1)
 unsigned ibound=1;
 unsigned nel=Bulk_mesh_pt->nboundary_element(ibound);
 for (unsigned e=0;e<nel;e++)
  {
   // Get pointer to element
   ELEMENT* el_pt=dynamic_cast<ELEMENT*>(Bulk_mesh_pt->
                                         boundary_element_pt(ibound,e));
   // Activate depency on inflow
   el_pt->activate_inflow_dependency_on_external_data(
    outflow_spines,ibound,&Global_Physical_Variables::inflow);
  }
}
//========================================================================
/// Doc the solution
//========================================================================
template<class ELEMENT>
void BrethertonProblem<ELEMENT>::doc_solution(DocInfo& doc_info)
{
 ofstream some_file;
 char filename[100];
 // Number of plot points
 unsigned npts=5;
 // Control coordinate: At bubble tip
 Vector<double> s(2);
 s[0]=1.0;
 s[1]=1.0;

 // Last proper spine
 unsigned last_spine=Bulk_mesh_pt->nfree_surface_spines()-1;
 // Doc
 Trace_file << " " << Global_Physical_Variables::Ca;
 Trace_file << " " << Bulk_mesh_pt->spine_pt(0)->height();
 Trace_file << " " << Bulk_mesh_pt->spine_pt(last_spine)->height();
 Trace_file << " " << 1.3375*pow(Global_Physical_Variables::Ca,2.0/3.0);
 Trace_file << " " << -Control_element_pt->interpolated_p_nst(s)*
                      Global_Physical_Variables::Ca;
 Trace_file << " " << 1.0+3.8*pow(Global_Physical_Variables::Ca,2.0/3.0);
 Trace_file << " " << Control_element_pt->interpolated_u_nst(s,0);
 Trace_file << " " << Control_element_pt->interpolated_u_nst(s,1);
 Trace_file << std::endl;
 // Output solution
 sprintf(filename,"%s/soln%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 Bulk_mesh_pt->output(some_file,npts);
 some_file.close();
 // Output boundaries
 sprintf(filename,"%s/boundaries%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 Bulk_mesh_pt->output_boundaries(some_file);
 some_file.close();
}
//========================================================================
/// Parameter study
//========================================================================
template<class ELEMENT>
void BrethertonProblem<ELEMENT>::parameter_study(const unsigned& nsteps)
{
 // Increase maximum residual
 Problem::Max_residuals=100.0;
 // Set output directory
 DocInfo doc_info;
 doc_info.set_directory("RESLT");
 doc_info.number()=0;
 // Open trace file
 char filename[100];
 sprintf(filename,"%s/trace.dat",doc_info.directory().c_str());
 Trace_file.open(filename);
 Trace_file << "VARIABLES=\"Ca\",";
```

```
 Trace_file « "\"h<sub>bottom</sub>\",\"h<sub>too</sub>\",";
 Trace_file « "\"h<sub>Bretherton</sub>\",\"p<sub>tip</sub>\",";
 Trace_file « "\"p<sub>tip (Bretherton)</sub>\",\"u<sub>stag</sub>\",";
 Trace_file « "\"v<sub>stag</sub>\"";
 Trace_file « "\"<greek>a</greek><sub>bottom</sub>\",";
 Trace_file « "\"<greek>a</greek><sub>top</sub>\"";
 Trace_file « std::endl;
 // Initial scaling factor for Ca reduction
 double factor=2.0;
 //Doc initial solution
 doc_solution(doc_info);
//Loop over the steps
 for (unsigned step=1;step<=nsteps;step++)
  {
   cout « std::endl « "STEP " « step « std::endl;
   // Newton method tends to converge is initial stays bounded below
   // a certain tolerance: This is cheaper to check than restarting
   // the Newton method after divergence (we'd also need to back up
   // the previous solution)
   double maxres=100.0;
   while (true)
    {
     cout « "Checking max. res for Ca = "
          « Global_Physical_Variables::Ca « ".   Max. residual: ";
     // Check the maximum residual
     DoubleVector residuals;
     actions_before_newton_solve();
     actions_before_newton_convergence_check();
     get_residuals(residuals);
     double max_res=residuals.max();
     cout « max_res;
     // Check what to do
     if (max_res>maxres)
      {
       cout « ". Too big!" « std::endl;
       // Reset the capillary number
       Global_Physical_Variables::Ca *= factor;
       // Reduce the factor
       factor=1.0+(factor-1.0)/1.5;
       cout « "New reduction factor: " « factor « std::endl;
       // Reduce the capillary number
       Global_Physical_Variables::Ca /= factor;
       // Try again
       continue;
      }
     // Looks promising: Let's proceed to solve
     else
      {
       cout « ". OK" « std::endl « std::endl;
       // Break out of the Ca adjustment loop
       break;
      }
    }
   //Solve
   cout « "Solving for capillary number: "
        « Global_Physical_Variables::Ca « std::endl;
   newton_solve();
   // Doc solution
   doc_info.number()++;
   doc_solution(doc_info);

   // Reduce the capillary number
   Global_Physical_Variables::Ca /= factor;
  }
}
//======================================================================
/// Driver code for unsteady two-layer fluid problem. If there are
/// any command line arguments, we regard this as a validation run
/// and perform only a single step.
//======================================================================
int main(int argc, char* argv[])
{
 // Store command line arguments
 CommandLineArgs::setup(argc,argv);
 // Set physical parameters:
 //Set direction of gravity: Vertically downwards
 Global_Physical_Variables::G[0] = 0.0;
 Global_Physical_Variables::G[1] = -1.0;
 // Womersley number = Reynolds number (St = 1)
 Global_Physical_Variables::ReSt = 0.0;
 Global_Physical_Variables::Re = Global_Physical_Variables::ReSt;
 // The Capillary number
 Global_Physical_Variables::Ca = 0.05;
 // Re/Fr -- a measure of gravity...
 Global_Physical_Variables::ReInvFr = 0.0;
 //Set up the problem
 BrethertonProblem<BrethertonElement<SpineElement<QCrouzeixRaviartElement<2> > > > problem;
```

```
 // Self test:
 problem.self_test();
 // Number of steps:
 unsigned nstep;
 if (CommandLineArgs::Argc>1)
  {
   // Validation run: Just one step
   nstep=2;
  }
 else
  {
   // Full run otherwise
   nstep=30;
  }
 // Run the parameter study: Perform nstep steps
 problem.parameter_study(nstep);
}
```

1.5 Source files for this tutorial

- The source files for this tutorial are located in the directory:

demo_drivers/navier_stokes/bretherton

- The driver code is:

demo_drivers/navier_stokes/bretherton/bretherton.cc

1.6 PDF file

A pdf version of this document is available.