

# Chapter 1

## Example problem: A rigid cylinder rotating in a viscous fluid beneath a free surface.

Detailed documentation to be written. Here's the driver code...

(This problem is solved using spatially adaptive elements with a pseudo-elastic remesh strategy)

```
//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) 2006-2022 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//=====
//A driver program to solve the problem of a cylinder rotating near a free
//surface
#include "generic.h"
#include "navier_stokes.h"
#include "solid.h"
#include "fluid_interface.h"
using namespace oomph;
//=start_of_namespace=====
// Namespace for physical parameters
//=====
namespace Global_Physical_Variables
{

    /// Pseudo-solid Poisson ratio
    double Nu=0.1;

    /// Direction of the wall normal vector
    Vector<double> Wall_normal;

    /// Function that specifies the wall unit normal
    void wall_unit_normal_fct(const Vector<double> &x,
                             Vector<double> &normal)
    {
        normal=Wall_normal;
    }
} // end_of_namespace
//My own Ellipse class
class GeneralEllipse : public GeomObject
```

```

{
private:
//Internal data to store the centre and semi-axes
double *centre_x_pt, *centre_y_pt, *a_pt, *b_pt;
public:

//Constructor
GeneralEllipse(const double &centre_x, const double &centre_y,
               const double &a, const double &b)
: GeomObject(1,2), centre_x_pt(0), centre_y_pt(0), a_pt(0), b_pt(0)
{
    centre_x_pt = new double(centre_x);
    centre_y_pt = new double(centre_y);
    a_pt = new double(a);
    b_pt = new double(b);
}
//Destructor
~GeneralEllipse()
{
    delete centre_x_pt;
    delete centre_y_pt;
    delete a_pt;
    delete b_pt;
}
//Return the position
void position(const Vector<double> &xi, Vector<double> &r) const
{
    r[0] = *centre_x_pt + *a_pt*cos(xi[0]);
    r[1] = *centre_y_pt + *b_pt*sin(xi[0]);
}
};
//A Domain
class CylinderAndInterfaceDomain : public Domain
{
public:
    double centre_x, centre_y;
private:

    double Lower_left[2], Lower_right[2], Lower_mid_left[2], Lower_mid_right[2];
    double Upper_left[2], Upper_right[2], Upper_mid_left[2], Upper_mid_right[2];
    double Lower_centre_left[2], Lower_centre_right[2];
    double Upper_centre_left[2], Upper_centre_right[2];

    // Geometric object that represents the rotating cylinder
    GeomObject* Cylinder_pt;
public:
//Constructor, pass the length and height of the domain
CylinderAndInterfaceDomain(const double &Length, const double &Height)
{
    centre_x = Length/2.0;
    centre_y = Height/2.0; //3.0*Height/4.0;
//Create a new ellipse object to represent the rotating cylinder
Cylinder_pt = new GeneralEllipse(centre_x,centre_y,0.2*Height,0.2*Height);
//Set some basic coordinates
Lower_left[0] = 0.0;
Lower_left[1] = 0.0;

Upper_left[0] = 0.0;
Upper_left[1] = Height;
Lower_right[0] = Length;
Lower_right[1] = 0.0;

Upper_right[0] = Length;
Upper_right[1] = Height;
//Let's just do some mid coordinates
Lower_mid_left[0] = Length/10.0;
Lower_mid_left[1] = 0.0;
Upper_mid_left[0] = Length/10.0;
Upper_mid_left[1] = Height;
Vector<double> xi(1), f(2);
xi[0] = -3.0*atan(1.0);
Cylinder_pt->position(xi,f);
Lower_centre_left[0] = f[0];
Lower_centre_left[1] = f[1];

xi[0] = 3.0*atan(1.0);
Cylinder_pt->position(xi,f);
Upper_centre_left[0] = f[0];
Upper_centre_left[1] = f[1];
Lower_mid_right[0] = 9.0*Length/10.0;
Lower_mid_right[1] = 0.0;
Upper_mid_right[0] = 9.0*Length/10.0;
Upper_mid_right[1] = Height;
xi[0] = -1.0*atan(1.0);
Cylinder_pt->position(xi,f);
Lower_centre_right[0] = f[0];
Lower_centre_right[1] = f[1];

```

---

```

    xi[0] = 1.0*atan(1.0);
    Cylinder_pt->position(xi,f);
    Upper_centre_right[0] = f[0];
    Upper_centre_right[1] = f[1];

    //There are six macro elements
    Macro_element_pt.resize(6);
    // Build the macro elements
    for (unsigned i=0;i<6;i++)
    {Macro_element_pt[i]= new QMacroElement<2>(this,i);}
}
// Destructor: Empty; cleanup done in base class
~CylinderAndInterfaceDomain() {}
//Private little interpolation problem
void linear_interpolate(double Left[2], double Right[2],
                        const double &s, Vector<double> &f)
{
    for(unsigned i=0;i<2;i++)
    {
        f[i] = Left[i] + (Right[i] - Left[i])*0.5*(s+1.0);
    }
}

// Sort out the vector representation of the i-th macro element
void macro_element_boundary(const unsigned &time,
                           const unsigned &m,
                           const unsigned &direction,
                           const Vector<double> &s,
                           Vector<double> &f)
{
    using namespace QuadTreeNames;
#ifdef WARN_ABOUT_SUBTLY_CHANGED_OOMPH_INTERFACES
    // Warn about time argument being moved to the front
    OomphLibWarning(
        "Order of function arguments has changed between versions 0.8 and 0.85",
        "CylinderAndInterfaceDomain::macro_element_boundary(...)",
        OOMPH_EXCEPTION_LOCATION);
#endif
    Vector<double> xi(1);
    //Switch on the macro element
    switch(m)
    {
        //Macro element 0, is the left-hand film
        case 0:

            switch(direction)
            {
                case N:
                    linear_interpolate(Upper_left,Upper_mid_left,s[0],f);
                    break;
                case S:
                    linear_interpolate(Lower_left,Lower_mid_left,s[0],f);
                    break;
                case W:
                    linear_interpolate(Lower_left,Upper_left,s[0],f);
                    break;
                case E:
                    linear_interpolate(Lower_mid_left,Upper_mid_left,s[0],f);
                    break;
                default:
                    std::ostringstream error_stream;
                    error_stream << "Direction is incorrect: " << direction << std::endl;
                    throw OomphLibError(error_stream.str(),
                                         OOMPH_CURRENT_FUNCTION,
                                         OOMPH_EXCEPTION_LOCATION);
            }

            break;

        //Macro element 1, is immediately left of the cylinder
        case 1:

            switch(direction)
            {
                case N:
                    linear_interpolate(Upper_mid_left,Upper_centre_left,s[0],f);
                    break;
                case S:
                    linear_interpolate(Lower_mid_left,Lower_centre_left,s[0],f);
                    break;
                case W:
                    linear_interpolate(Lower_mid_left,Upper_mid_left,s[0],f);
                    break;
                case E:
                    xi[0] = 5.0*atan(1.0) - 2.0*atan(1.0)*0.5*(1.0+s[0]);
            }
    }
}

```

```

    Cylinder_pt->position(xi,f);
    break;
default:
    std::ostringstream error_stream;
    error_stream << "Direction is incorrect: " << direction << std::endl;

    throw OomphLibError(error_stream.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
}

break;
//Macro element 2, is immediately above the cylinder
case 2:

switch(direction)
{
case N:
    linear_interpolate(Upper_mid_left,Upper_mid_right,s[0],f);
    break;

case S:
    xi[0] = 3.0*atan(1.0) - 2.0*atan(1.0)*0.5*(1.0+s[0]);
    Cylinder_pt->position(xi,f);
    break;
case W:
    linear_interpolate(Upper_centre_left,Upper_mid_left,s[0],f);
    break;
case E:
    linear_interpolate(Upper_centre_right,Upper_mid_right,s[0],f);
    break;
default:
    std::ostringstream error_stream;
    error_stream << "Direction is incorrect: " << direction << std::endl;

    throw OomphLibError(error_stream.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
}

break;
//Macro element 3, is immediately right of the cylinder
case 3:
switch(direction)
{
case N:
    linear_interpolate(Upper_centre_right,Upper_mid_right,s[0],f);
    break;

case S:
    linear_interpolate(Lower_centre_right,Lower_mid_right,s[0],f);
    break;
case W:
    xi[0] = -atan(1.0) + 2.0*atan(1.0)*0.5*(1.0+s[0]);
    Cylinder_pt->position(xi,f);
    break;
case E:
    linear_interpolate(Lower_mid_right,Upper_mid_right,s[0],f);
    break;
default:
    std::ostringstream error_stream;
    error_stream << "Direction is incorrect: " << direction << std::endl;
    throw OomphLibError(error_stream.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
}

break;

//Macro element 4, is immediately below cylinder
case 4:

switch(direction)
{
case N:
    //linear_interpolate(Lower_centre_left,Lower_centre_right,s[0],f);
    xi[0] = -3.0*atan(1.0) + 2.0*atan(1.0)*0.5*(1.0+s[0]);
    Cylinder_pt->position(xi,f);
    break;

case S:
    linear_interpolate(Lower_mid_left,Lower_mid_right,s[0],f);
    break;
case W:
    linear_interpolate(Lower_mid_left,Lower_centre_left,s[0],f);
    break;
case E:
    linear_interpolate(Lower_mid_right,Lower_centre_right,s[0],f);

```

```

        break;
    default:
        std::ostringstream error_stream;
        error_stream << "Direction is incorrect: " << direction << std::endl;
        throw OomphLibError(error_stream.str(),
                             OOMPH_CURRENT_FUNCTION,
                             OOMPH_EXCEPTION_LOCATION);
    }
    break;
    //Macro element 5, is right film
    case 5:

    switch(direction)
    {
    case N:
        linear_interpolate(Upper_mid_right,Upper_right,s[0],f);
        break;

    case S:
        linear_interpolate(Lower_mid_right,Lower_right,s[0],f);
        break;
    case W:
        linear_interpolate(Lower_mid_right,Upper_mid_right,s[0],f);
        break;
    case E:
        linear_interpolate(Lower_right,Upper_right,s[0],f);
        break;
    default:
        std::ostringstream error_stream;
        error_stream << "Direction is incorrect: " << direction << std::endl;
        throw OomphLibError(error_stream.str(),
                             OOMPH_CURRENT_FUNCTION,
                             OOMPH_EXCEPTION_LOCATION);
    }
    break;

    default:
        std::ostringstream error_stream;
        error_stream << "Wrong domain number: " << m << std::endl;
        throw OomphLibError(error_stream.str(),
                             OOMPH_CURRENT_FUNCTION,
                             OOMPH_EXCEPTION_LOCATION);
    }
}
};
//Now I need to actually create a Mesh
template<class ELEMENT>
class CylinderAndInterfaceMesh : public virtual SolidMesh
{
    double Height;
protected:
    //Pointer to the domain
    CylinderAndInterfaceDomain* Domain_pt;
public:
    //Access function to the domain
    CylinderAndInterfaceDomain* domain_pt() {return Domain_pt;}
    //Constructor,
    CylinderAndInterfaceMesh(const double &length, const double &height,
                             TimeStepper* time_stepper_pt) : Height(height)
    {
        //Create the domain
        Domain_pt = new CylinderAndInterfaceDomain(length,height);
        //Initialise the node counter
        unsigned node_count=0;
        //Vectors Used to get data from domains
        Vector<double> s(2), r(2);

        //Setup temporary storage for the Node
        Vector<Node *> Tmp_node_pt;
        //Now blindly loop over the macro elements and associate and finite
        //element with each
        unsigned Nmacro_element = Domain_pt->nmacro_element();
        for(unsigned e=0;e<Nmacro_element;e++)
        {
            //Create the FiniteElement and add to the Element_pt Vector
            Element_pt.push_back(new ELEMENT);
            //Read out the number of linear points in the element
            unsigned Np =
                dynamic_cast<ELEMENT*>(finite_element_pt(e))->nnode_ld();

            //Loop over nodes in the column
            for(unsigned l1=0;l1<Np;l1++)
            {
                //Loop over the nodes in the row
                for(unsigned l2=0;l2<Np;l2++)
                {
                    //Allocate the memory for the node

```

```

    Tmp_node_pt.push_back(finite_element_pt(e)->
        construct_node(l1*Np+l2,time_stepper_pt));

    //Read out the position of the node from the macro element
    s[0] = -1.0 + 2.0*(double)l2/(double)(Np-1);
    s[1] = -1.0 + 2.0*(double)l1/(double)(Np-1);
    Domain_pt->macro_element_pt(e)->macro_map(s,r);

    //Set the position of the node
    Tmp_node_pt[node_count]->x(0) = r[0];
    Tmp_node_pt[node_count]->x(1) = r[1];

    //Increment the node number
    node_count++;
}
} //End of loop over macro elements

//Now the elements have been created, but there will be nodes in
//common, need to loop over the common edges and sort it, by reassigning
//pointers and the deleting excess nodes

//Read out the number of linear points in the element
unsigned Np =
    dynamic_cast<ELEMENT*>(finite_element_pt(0))->nnode_ld();
//DelaunayEdge between Elements 0 and 1
for(unsigned n=0;n<Np;n++)
{
    //Set the nodes in element 1 to be the same as in element 0
    finite_element_pt(1)->node_pt(Np*n)
        = finite_element_pt(0)->node_pt(n*Np+Np-1);
    //Remove the nodes in element 1 from the temporary node list
    delete Tmp_node_pt[Np*Np + Np*n];
    Tmp_node_pt[Np*Np + Np*n] = 0;
}
//DelaunayEdge between Elements 1 and 2
for(unsigned n=0;n<Np;n++)
{
    //Set the nodes in element 2 to be the same as in element 1
    finite_element_pt(2)->node_pt(n*Np)
        = finite_element_pt(1)->node_pt((Np-1)*Np+Np-1-n);
    //Remove the nodes in element 2 from the temporary node list
    delete Tmp_node_pt[2*Np*Np + n*Np];
    Tmp_node_pt[2*Np*Np + n*Np] = 0;
}
//DelaunayEdge between Elements 1 and 4
for(unsigned n=0;n<Np;n++)
{
    //Set the nodes in element 4 to be the same as in element 1
    finite_element_pt(4)->node_pt(n*Np)
        = finite_element_pt(1)->node_pt(n);
    //Remove the nodes in element 2 from the temporary node list
    delete Tmp_node_pt[4*Np*Np + n*Np];
    Tmp_node_pt[4*Np*Np + n*Np] = 0;
}
//DelaunayEdge between Element 2 and 3
for(unsigned n=0;n<Np;n++)
{
    //Set the nodes in element 3 to be the same as in element 2
    finite_element_pt(3)->node_pt(Np*(Np-1)+n)
        = finite_element_pt(2)->node_pt(Np*n+Np-1);
    //Remove the nodes in element 3 from the temporary node list
    delete Tmp_node_pt[3*Np*Np + Np*(Np-1)+n];
    Tmp_node_pt[3*Np*Np + Np*(Np-1)+n] = 0;
}
//DelaunayEdge between Element 4 and 3
for(unsigned n=0;n<Np;n++)
{
    //Set the nodes in element 3 to be the same as in element 4
    finite_element_pt(3)->node_pt(n)
        = finite_element_pt(4)->node_pt(Np*(Np-1)+Np-1);
    //Remove the nodes in element 3 from the temporary node list
    delete Tmp_node_pt[3*Np*Np + n];
    Tmp_node_pt[3*Np*Np + n] = 0;
}
//DelaunayEdge between Element 3 and 5
for(unsigned n=0;n<Np;n++)
{
    //Set the nodes in element 5 to be the same as in element 3
    finite_element_pt(5)->node_pt(n*Np)
        = finite_element_pt(3)->node_pt(Np*n+Np-1);
    //Remove the nodes in element 5 from the temporary node list
    delete Tmp_node_pt[5*Np*Np + n*Np];
    Tmp_node_pt[5*Np*Np + n*Np] = 0;
}
}
//Now set the actual true nodes

```

```

for(unsigned n=0;n<node_count;n++)
{
    if(Tmp_node_pt[n]!=0) {Node_pt.push_back(Tmp_node_pt[n]);}
}

//Finally set the nodes on the boundaries
set_nboundary(5);

for(unsigned n=0;n<Np;n++)
{
    //Left hand side
    Node* temp_node_pt = finite_element_pt(0)->node_pt(n*Np);
    this->convert_to_boundary_node(temp_node_pt);
    add_boundary_node(3,temp_node_pt);

    //Right hand side
    temp_node_pt = finite_element_pt(5)->node_pt(n*Np+Np-1);
    this->convert_to_boundary_node(temp_node_pt);
    add_boundary_node(1,temp_node_pt);

    //LH part of lower boundary
    temp_node_pt = finite_element_pt(0)->node_pt(n);
    this->convert_to_boundary_node(temp_node_pt);
    add_boundary_node(0,temp_node_pt);
    //First part of upper boundary
    temp_node_pt = finite_element_pt(0)->node_pt(Np*(Np-1)+n);
    this->convert_to_boundary_node(temp_node_pt);
    add_boundary_node(2,temp_node_pt);

    //First part of hole boundary
    temp_node_pt = finite_element_pt(4)->node_pt(Np*(Np-1)+n);
    this->convert_to_boundary_node(temp_node_pt);
    add_boundary_node(4,temp_node_pt);
}
for(unsigned n=1;n<Np;n++)
{
    //Middle of lower boundary
    Node* temp_node_pt = finite_element_pt(4)->node_pt(n);
    this->convert_to_boundary_node(temp_node_pt);
    add_boundary_node(0,temp_node_pt);

    //Middle of upper boundary
    temp_node_pt = finite_element_pt(2)->node_pt(Np*(Np-1)+n);
    this->convert_to_boundary_node(temp_node_pt);
    add_boundary_node(2,temp_node_pt);
    //Next part of hole
    temp_node_pt = finite_element_pt(3)->node_pt(n*Np);
    this->convert_to_boundary_node(temp_node_pt);
    add_boundary_node(4,temp_node_pt);
}
for(unsigned n=1;n<Np;n++)
{
    //Final part of lower boundary
    Node* temp_node_pt = finite_element_pt(5)->node_pt(n);
    this->convert_to_boundary_node(temp_node_pt);
    add_boundary_node(0,temp_node_pt);

    //Middle of upper boundary
    temp_node_pt = finite_element_pt(5)->node_pt(Np*(Np-1)+n);
    this->convert_to_boundary_node(temp_node_pt);
    add_boundary_node(2,temp_node_pt);

    //Next part of hole
    temp_node_pt = finite_element_pt(2)->node_pt(Np-n-1);
    this->convert_to_boundary_node(temp_node_pt);
    add_boundary_node(4,temp_node_pt);
}
for(unsigned n=1;n<Np-1;n++)
{
    //Final part of hole
    Node* temp_node_pt = finite_element_pt(1)->node_pt(Np*(Np-n-1)+Np-1);
    this->convert_to_boundary_node(temp_node_pt);
    add_boundary_node(4,temp_node_pt);
}

//Now loop over all the nodes and set their Lagrangian coordinates
unsigned Nnode = nnode();
for(unsigned n=0;n<Nnode;n++)
{
    //Cast node to an elastic node
    SolidNode* temp_pt = static_cast<SolidNode*>(Node_pt[n]);
    for(unsigned i=0;i<2;i++)
    {temp_pt->xi(i) = temp_pt->x(i);}
}
};

```

```

//Now let's do the adaptive mesh
template<class ELEMENT>
class RefineableCylinderAndInterfaceMesh :
public CylinderAndInterfaceMesh<ELEMENT>, public RefineableQuadMesh<ELEMENT>
{
public:
// Constructor
RefineableCylinderAndInterfaceMesh(const double &length, const double &height,
TimeStepper* time_stepper_pt) :
CylinderAndInterfaceMesh<ELEMENT>(length,height,time_stepper_pt)
{
// Nodal positions etc. were created in constructor for
// Cylinder...<...>. Need to setup adaptive information.
// Loop over all elements and set macro element pointer
for (unsigned e=0;e<6;e++)
{
dynamic_cast<ELEMENT*>(this->element_pt(e))->
set_macro_elem_pt(this->Domain_pt->macro_element_pt(e));
}
// Setup quadtree forest for mesh refinement
this->setup_quadtree_forest();

// Setup the boundary element info
this->setup_boundary_element_info();
}

/// Destructor: Empty
virtual ~RefineableCylinderAndInterfaceMesh() {}
};
template<class ELEMENT>
class RefineableRotatingCylinderProblem : public Problem
{
private:
double Length, Height;

//Constitutive law used to determine the mesh deformation
ConstitutiveLaw *Constitutive_law_pt;
Data* Traded_pressure_data_pt;
public:
double Re, Ca, ReInvFr, Bo;
double Omega;

double Volume;
double Angle;
Vector<double> G;

/// Constructor: Pass flag to indicate if you want
/// a constant source function or the tanh profile.
RefineableRotatingCylinderProblem(const double &length, const double &height);

/// Update the problem specs after solve (empty)
void actions_after_newton_solve() {}

/// Update the problem specs before solve:
void actions_before_newton_solve() {set_boundary_conditions();}

/// Strip off the interface before adaptation
void actions_before_adapt()
{
this->delete_volume_constraint_elements();
this->delete_free_surface_elements();
}
void actions_after_adapt() {finish_problem_setup(); this->rebuild_global_mesh();}

/// Complete problem setup: Setup element-specific things
/// (source fct pointers etc.)
void finish_problem_setup();
//Access function for the mesh
RefineableCylinderAndInterfaceMesh<ELEMENT>* Bulk_mesh_pt;
//Access function for surface mesh
Mesh* Surface_mesh_pt;
//Access function for point mesh
Mesh* Point_mesh_pt;

/// The volume constraint mesh
Mesh* Volume_constraint_mesh_pt;
void set_boundary_conditions();
void solve();

/// Create the volume constraint elements
void create_volume_constraint_elements()
{
//The single volume constraint element
VolumeConstraintElement* vol_constraint_element =
new VolumeConstraintElement(&Volume,Traded_pressure_data_pt,0);
Volume_constraint_mesh_pt->add_element_pt(vol_constraint_element);
}

```



```

//Loop over all boundaries (or just 1 and 2 why?)
for(unsigned b=0;b<4;b++)
{
    // How many bulk fluid elements are adjacent to boundary b?
    unsigned n_element = Bulk_mesh_pt->nboundary_element(b);

    // Loop over the bulk fluid elements adjacent to boundary b?
    for(unsigned e=0;e<n_element;e++)
    {
        // Get pointer to the bulk fluid element that is
        // adjacent to boundary b
        ELEMENT* bulk_elem_pt = dynamic_cast<ELEMENT*>(
            Bulk_mesh_pt->boundary_element_pt(b,e));

        //Find the index of the face of element e along boundary b
        int face_index = Bulk_mesh_pt->face_index_at_boundary(b,e);

        // Create new element
        ElasticLineVolumeConstraintBoundingElement<ELEMENT*> el_pt =
            new ElasticLineVolumeConstraintBoundingElement<ELEMENT*>(
                bulk_elem_pt,face_index);

        //Set the "master" volume control element
        el_pt->set_volume_constraint_element(vol_constraint_element);

        // Add it to the mesh
        Volume_constraint_mesh_pt->add_element_pt(el_pt);
    }
}

void delete_volume_constraint_elements()
{
    unsigned n_element = Volume_constraint_mesh_pt->nelement();
    for(unsigned e=0;e<n_element;e++)
    {
        delete Volume_constraint_mesh_pt->element_pt(e);
    }
    Volume_constraint_mesh_pt->flush_element_and_node_storage();
}

void create_free_surface_elements()
{
    //Find number of elements adjacent to upper boundary
    unsigned n_boundary_element = Bulk_mesh_pt->nboundary_element(2);
    //The boundary elements do not necessarily come in order, so we will
    //need to detect the element adjacent to boundary 1.
    //The index of that element in our array will be stored in this variable
    //(initialised to a negative and therefore invalid number)
    int final_element_index=-1;
    //Loop over the elements adjacent to the boundary
    for(unsigned e=0;e<n_boundary_element;e++)
    {
        //Create the free surface element (on face 2)
        FiniteElement *free_surface_element_pt
            = new ElasticLineFluidInterfaceElement<ELEMENT>
                (Bulk_mesh_pt->boundary_element_pt(2,e),
                 Bulk_mesh_pt->face_index_at_boundary(2,e));
        //Push it back onto the stack
        Surface_mesh_pt->add_element_pt(free_surface_element_pt);
        //Check whether the element is on the boundary 1
        unsigned n_node = free_surface_element_pt->nnode();
        //Only need to check the end nodes
        if((free_surface_element_pt->node_pt(0)->is_on_boundary(1)) ||
            (free_surface_element_pt->node_pt(n_node-1)->is_on_boundary(1)))
        {
            final_element_index=e;
        }
    }
    unsigned Nfree = Surface_mesh_pt->nelement();
    oomph_info << Nfree << " free surface elements assigned" << std::endl;
    if(final_element_index == -1)
    {
        throw OomphLibError("No Surface Element adjacent to boundary 1\n",
            OOMPH_CURRENT_FUNCTION,
            OOMPH_EXCEPTION_LOCATION);
    }

    //Make the edge point
    FiniteElement* point_element_pt=
        dynamic_cast<ElasticLineFluidInterfaceElement<ELEMENT*>>
            (Surface_mesh_pt->element_pt(final_element_index))
        ->make_bounding_element(1);

    //Add it to the stack
    Point_mesh_pt->add_element_pt(point_element_pt);
}

//Function to delete the free surface elements

```

```

void delete_free_surface_elements()
{
    //Find the number of traction elements
    unsigned Nfree_surface = Surface_mesh_pt->nelement();

    //The traction elements are ALWAYS? stored at the end
    //So delete and remove them, add one to get rid of the constraint
    for(unsigned e=0;e<Nfree_surface;e++)
    {
        delete Surface_mesh_pt->element_pt(e);
    }
    Surface_mesh_pt->flush_element_and_node_storage();
    delete Point_mesh_pt->element_pt(0);
    Point_mesh_pt->flush_element_and_node_storage();
}

};
//=====
/// Constructor for adaptive Poisson problem in deformable fish-shaped
/// domain. Pass bool to indicate if we want a constant source
/// function or the one that produces a tanh step.
//=====
template<class ELEMENT>
RefineableRotatingCylinderProblem<ELEMENT>::RefineableRotatingCylinderProblem(
    const double &length, const double &height) : Length(length), Height(height),
                                                Re(0.0), Ca(0.001),
                                                ReInvFr(0.0),
                                                Bo(0.0), Omega(1.0),
                                                Volume(12.0),
                                                Angle(1.57)
{
    Global_Physical_Variables::Wall_normal.resize(2);
    Global_Physical_Variables::Wall_normal[0] = 1.0;
    Global_Physical_Variables::Wall_normal[1] = 0.0;
    G.resize(2);
    G[0] = 0.0; G[1] = -1.0;

    // Set the initial value of the ReInvFr = Bo/Ca
    ReInvFr = Bo/Ca;

    // Build a linear solver: Use HSL's MA42 frontal solver
    //linear_solver_pt() = new HSL_MA42;
    //Set the constitutive law
    Constitutive_law_pt = new GeneralisedHookean(&Global_Physical_Variables::Nu);

    // Switch off full doc for frontal solver
    //static_cast<HSL_MA42*>(linear_solver_pt())->disable_doc_stats();
    //Allocate the timestepper (no timedependence)
    add_time_stepper_pt(new Steady<0>);

    // Build mesh
    Bulk_mesh_pt=
        new RefineableCylinderAndInterfaceMesh<ELEMENT>(length,height,
                                                         Problem::time_stepper_pt());

    // Set error estimator
    Z2ErrorEstimator* error_estimator_pt=new Z2ErrorEstimator;
    Bulk_mesh_pt->spatial_error_estimator_pt()=error_estimator_pt;

    //Refine the problem a couple of times
    bool update_all_solid_nodes=true;
    Bulk_mesh_pt->refine_uniformly();
    Bulk_mesh_pt->node_update(update_all_solid_nodes);
    Bulk_mesh_pt->refine_uniformly();
    Bulk_mesh_pt->node_update(update_all_solid_nodes);
    //Bulk_mesh_pt->refine_uniformly();
    //refine_uniformly();
    //refine_uniformly();

    // Loop over all elements and unset macro element pointer
    unsigned Nelement = Bulk_mesh_pt->nelement();
    for(unsigned e=0;e<Nelement;e++)
    {
        dynamic_cast<ELEMENT*>(Bulk_mesh_pt->element_pt(e))->
            set_macro_elem_pt(0);
    }
    //The external pressure is a piece of global data
    Traded_pressure_data_pt = new Data(1);
    this->add_global_data(Traded_pressure_data_pt);
    // Complete the build of all elements so they are fully functional
    Surface_mesh_pt = new Mesh;
    Point_mesh_pt = new Mesh;
    Volume_constraint_mesh_pt = new Mesh;
    finish_problem_setup();

    this->add_sub_mesh(Bulk_mesh_pt);
    this->add_sub_mesh(Surface_mesh_pt);
    this->add_sub_mesh(Point_mesh_pt);
}

```

```

this->add_sub_mesh(Volume_constraint_mesh_pt);
this->build_global_mesh();
//Attach the boundary conditions to the mesh
oomph_info << "Number of equations: " << assign_eqn_numbers() << std::endl;
}
//=====
/// Complete build of Poisson problem:
/// Loop over elements and setup pointers to source function
///
//=====
template<class ELEMENT>
void RefineableRotatingCylinderProblem<ELEMENT>::finish_problem_setup()
{
    //Now sort out the free surface
    this->create_free_surface_elements();
    //Create the volume constraint elements
    this->create_volume_constraint_elements();

    // Set the boundary conditions for this problem: All nodes are
    // free by default -- just pin the ones that have Dirichlet conditions
    // here.
    //Pin bottom and cylinder
    unsigned num_bound = Bulk_mesh_pt->nboundary();
    for (unsigned ibound=0; ibound<num_bound; ibound+=4)
    {
        unsigned num_nod= Bulk_mesh_pt->nboundary_node(ibound);
        for (unsigned inod=0; inod<num_nod; inod++)
        {
            Bulk_mesh_pt->boundary_node_pt(ibound, inod)->pin(0);
            Bulk_mesh_pt->boundary_node_pt(ibound, inod)->pin(1);
        }
    }

    //Pin u and v on LHS
    {
        unsigned num_nod= Bulk_mesh_pt->nboundary_node(3);
        for (unsigned inod=0; inod<num_nod; inod++)
        {
            Bulk_mesh_pt->boundary_node_pt(3, inod)->pin(0);
            //Bulk_mesh_pt->boundary_node_pt(3, inod)->pin(1);
        }
    }

    //Pin u and v on RHS
    {
        unsigned num_nod= Bulk_mesh_pt->nboundary_node(1);
        for (unsigned inod=0; inod<num_nod; inod++)
        {
            Bulk_mesh_pt->boundary_node_pt(1, inod)->pin(0);
            Bulk_mesh_pt->boundary_node_pt(1, inod)->pin(1);
        }
    }

    dynamic_cast<FluidInterfaceBoundingElement*>
        (Point_mesh_pt->element_pt(0))->set_contact_angle(&Angle);
    dynamic_cast<FluidInterfaceBoundingElement*>
        (Point_mesh_pt->element_pt(0))->ca_pt() = &Ca;

    dynamic_cast<FluidInterfaceBoundingElement*>
        (Point_mesh_pt->element_pt(0))->
        wall_unit_normal_fct_pt() = &Global_Physical_Variables::wall_unit_normal_fct;
    //Pin one pressure
    dynamic_cast<ELEMENT*>(Bulk_mesh_pt->element_pt(0))->fix_pressure(0,0.0);
    //Loop over the lower boundary and pin nodal positions in both directions
    unsigned num_nod= Bulk_mesh_pt->nboundary_node(0);
    for (unsigned inod=0; inod<num_nod; inod++)
    {
        Bulk_mesh_pt->boundary_node_pt(0, inod)->pin_position(0);
        Bulk_mesh_pt->boundary_node_pt(0, inod)->pin_position(1);
    }

    //Loop over the RHS side and pin in x and y
    num_nod= Bulk_mesh_pt->nboundary_node(1);
    for (unsigned inod=0; inod<num_nod; inod++)
    {
        Bulk_mesh_pt->boundary_node_pt(1, inod)->pin_position(0);
        //Bulk_mesh_pt->boundary_node_pt(1, inod)->pin_position(1);
    }

    //Loop over the LHS side and pin in x
    num_nod= Bulk_mesh_pt->nboundary_node(3);
    for (unsigned inod=0; inod<num_nod; inod++)
    {
        Bulk_mesh_pt->boundary_node_pt(3, inod)->pin_position(0);
        //Bulk_mesh_pt->boundary_node_pt(3, inod)->pin_position(1);
    }
}

```

```

//Loop over the cylinder and pin nodal positions in both directions
num_nod= Bulk_mesh_pt->nboundary_node(4);
for (unsigned inod=0;inod<num_nod;inod++)
{
    Bulk_mesh_pt->boundary_node_pt(4,inod)->pin_position(0);
    Bulk_mesh_pt->boundary_node_pt(4,inod)->pin_position(1);
}
//Find number of elements in mesh
unsigned Nfluid = Bulk_mesh_pt->nelement();
//Find the number of free surface elements
unsigned Nfree = Surface_mesh_pt->nelement();
// Loop over the elements to set up element-specific
// things that cannot be handled by constructor
for(unsigned i=0;i<Nfluid;i++)
{
    // Upcast from FiniteElement to the present element
    ELEMENT *temp_pt = dynamic_cast<ELEMENT*>(Bulk_mesh_pt->element_pt(i));

    //Set the source function pointer
    temp_pt->re_pt() = &Re;
    temp_pt->re_invfr_pt() = &ReInvFr;
    temp_pt->g_pt() = &G;

    //Assign the mesh deformation constitutive law
    temp_pt->constitutive_law_pt() = Constitutive_law_pt;

}

// Pin the redundant solid pressures (if any)
PVDEquationsBase<2>::pin_redundant_nodal_solid_pressures(
    Bulk_mesh_pt->element_pt());

//Loop over the free surface elements
for(unsigned i=0;i<Nfree;i++)
{
    // Upcast from FiniteElement to the present element
    ElasticLineFluidInterfaceElement<ELEMENT> *temp_pt =
        dynamic_cast<ElasticLineFluidInterfaceElement<ELEMENT>*>
        (Surface_mesh_pt->element_pt(i));
    //Set the Capillary number
    temp_pt->ca_pt() = &Ca;

    //Pass the Data item that contains the external pressure
    temp_pt->set_external_pressure_data(this->global_data_pt(0));
}
}

template<class ELEMENT>
void RefineableRotatingCylinderProblem<ELEMENT>::set_boundary_conditions()
{
    //Only bother to set non-zero velocity on the cylinder
    unsigned Nnode = Bulk_mesh_pt->nboundary_node(4);
    for(unsigned n=0;n<Nnode;n++)
    {
        //Get x and y
        double x = Bulk_mesh_pt->boundary_node_pt(4,n)->x(0);
        double y = Bulk_mesh_pt->boundary_node_pt(4,n)->x(1);
        //Now find the vector distance to the centre
        double len_x = x - Bulk_mesh_pt->domain_pt()->centre_x;
        double len_y = y - Bulk_mesh_pt->domain_pt()->centre_y;
        //Calculate the angle and radius
        double r = sqrt(len_x*len_x + len_y*len_y);
        double theta = atan2(len_y,len_x);

        //Now set the velocities
        Bulk_mesh_pt->boundary_node_pt(4,n)->set_value(0,-Omega*r*sin(theta));
        Bulk_mesh_pt->boundary_node_pt(4,n)->set_value(1, Omega*r*cos(theta));
    }
}

template<class ELEMENT>
void RefineableRotatingCylinderProblem<ELEMENT>::solve()
{
    Newton_solver_tolerance = 1.0e-8;
    //Document the solution
    std::ofstream filenameee("input.dat");
    Bulk_mesh_pt->output(filenameee,5);
    Surface_mesh_pt->output(filenameee,5);
    //Point_mesh_pt->output(filenameee,5);
    filenameee.close();
    //Solve the initial value problem
    newton_solve();
    std::ofstream filename("first.dat");
    Bulk_mesh_pt->output(filename,5);
    Surface_mesh_pt->output(filename,5);
    //Point_mesh_pt->output(filename,5);
}

```

```

filename.close();
//Initialise the value of the arc-length
double ds=0.001;

std::ofstream trace("trace.dat");
trace << Ca << " " << ReInvFr << " "
    << Bulk_mesh_pt->boundary_node_pt(2,0)->x(1) << std::endl;
// bool flag=true, fflag=true;
for(unsigned i=0;i<2;i++)
{
    if(i<5)
    {
        //Decrease the contact angle
        Angle -= 0.1;
        newton_solve(2);
        //newton_solve();
    }
    else
    {
        //do an arc-length continuation step in Ca
        ds = arc_length_step_solve(&Ca,ds);
    }
    // if(flag)
    // {
    //     //Do an arc-length continuation step in ReInvFr
    //     ds = arc_length_step_solve(&ReInvFr,ds);
    // }
    // else
    // {
    //     //Reset arc-length parameters
    //     if(fflag) {reset_arc_length_parameters(); fflag=false;}
    //     ds = 0.001;
    //     //Now do it in Ca
    //     ds = arc_length_step_solve(&Ca,ds);
    // }
    // if(Bulk_mesh_pt->boundary_node_pt(2,0)->x(1) < 4.0)
    // {flag=false;}
    trace << Ca << " " << ReInvFr << " " << Angle << " "
        << Bulk_mesh_pt->boundary_node_pt(2,0)->x(1) << std::endl;

    char file[100];
    sprintf(file,"step%i.dat",i);
    filename.open(file);
    Bulk_mesh_pt->output(filename,5);
    Surface_mesh_pt->output(filename,5);
    //Point_mesh_pt->output(filename,5);
    filename.close();
    //Now reset the values of the lagrange multipliers and the xi's
    //An updated lagrangian approach

    //Now loop over all the nodes and set their Lagrangian coordinates
    unsigned Nnode = Bulk_mesh_pt->nnode();
    for(unsigned n=0;n<Nnode;n++)
    {
        //Cast node to an elastic node
        SolidNode* temp_pt = static_cast<SolidNode*>(Bulk_mesh_pt->node_pt(n));
        for(unsigned j=0;j<2;j++) {temp_pt->xi(j) = temp_pt->x(j);}
    }
    //Find the number of free surface elements
    unsigned Nfree = Surface_mesh_pt->nelement();
    //Loop over the free surface elements
    for(unsigned n=0;n<Nfree;n++)
    {
        // Upcast from FiniteElement to the present element
        ElasticLineFluidInterfaceElement<ELEMENT> *temp_pt =
            dynamic_cast<ElasticLineFluidInterfaceElement<ELEMENT>*>
            (Surface_mesh_pt->element_pt(n));
        unsigned Nnode = temp_pt->nnode();
        //Reset the lagrange multipliers
        for(unsigned j=0;j<Nnode;j++) {temp_pt->lagrange(j) = 0.0;}
    }
}

//Document the solution
//filename.open("output.dat");
//Bulk_mesh_pt->output(filename,5);
//filename.close();
trace.close();
}

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

int main()
{
    RefineableRotatingCylinderProblem

```

```
<RefineablePseudoSolidNodeUpdateElement<RefineableQCrouzeixRaviartElement<2>,
RefineableQPVDElementWithContinuousPressure<2> > > problem(3.0,4.0);

//ofstream filename("mesh.dat");
//problem.Bulk_mesh_pt->output(filename,5);
problem.solve();
}
```

---

## 1.1 PDF file

A [pdf version](#) of this document is available.