

Chapter 1

A simple fluid-structure interaction problem revisited: Finite Reynolds number flow, driven by an oscillating ring – mesh update done by AlgebraicElements.

We re-visit the simple fluid-structure interaction problem considered in the [earlier example](#). This time we perform the update of the nodal positions with AlgebraicElements.

```
//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) 2006-2026 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//=====
// Driver for 2D Navier Stokes flow, driven by oscillating ring
// with pseudo-elasticity: The mean radius of ring is adjusted to
// allow conservation of volume (area).

// Oomph-lib includes
#include "generic.h"
#include "navier_stokes.h"

//Need to instantiate templated mesh
#include "meshes/quarter_circle_sector_mesh.h"

//Include namespace containing Sarah's asymptotics
#include "osc_ring_sarah_asymptotics.h"

using namespace std;
using namespace oomph;
using namespace MathematicalConstants;
```

```

//=====
/// Namespace for physical parameters
//=====
namespace Global_Physical_Variables
{
    /// Reynolds number
    double Re=100.0; // ADJUST_PRIORITY

    /// Reynolds x Strouhal number
    double ReSt=100.0; // ADJUST_PRIORITY
}

//=====
/// Driver for oscillating ring problem: Wall performs oscillations
/// that resemble eigenmodes of freely oscillating ring and drives
/// viscous fluid flow. Mean radius of wall is adjustable and
/// responds to a pressure value in the fluid to allow for
/// mass conservation.
//=====

template<class ELEMENT>
class OscRingNStProblem : public Problem
{
public:
    /// Constructor: Pass timestep and function pointer to the
    /// solution that provides the initial conditions for the fluid
    OscRingNStProblem(const double& dt,
                      FiniteElement::UnsteadyExactSolutionFctPt IC_fct_pt);

    /// Destructor (empty)
    ~OscRingNStProblem() {}

    /// Get pointer to wall as geometric object
    GeomObject* wall_pt()
    {
        return Wall_pt;
    }

    /// Update after solve (empty)
    void actions_after_newton_solve() {}

    /// Update the problem specs before solve (empty)
    void actions_before_newton_solve() {}

    /// Update the problem specs before checking Newton
    /// convergence: Update the fluid mesh and re-set velocity
    /// boundary conditions -- no slip velocity on the wall means
    /// that the velocity on the wall is dependent.
    void actions_before_newton_convergence_check()
    {
        // Update the fluid mesh -- auxiliary update function for algebraic
        // nodes automatically updates no slip condition.
        fluid_mesh_pt() ->node_update();
    }

    /// Update the problem specs after adaptation:
    /// Set auxiliary update function that applies no slip on all
    /// boundary nodes and choose fluid pressure dof that drives
    /// the wall deformation
    void actions_after_adapt()
    {
        // Ring boundary: No slip; this also implies that the velocity needs
        // to be updated in response to wall motion. This needs to be reset
        // every time the mesh is changed -- there's no mechanism by which
        // auxiliary update functions are copied to newly created nodes.
        // (that's because unlike boundary conditions, they don't
        // occur exclusively at boundaries)
        unsigned ibound=1;
    }
}

```

```

        unsigned num_nod= fluid_mesh_pt ()->nboundary_node(ibound);
        for (unsigned inod=0;inod<num_nod;inod++)
        {
            fluid_mesh_pt ()->boundary_node_pt(ibound,inod)->
            set_auxiliary_node_update_fct_pt(
                FSI_functions::apply_no_slip_on_moving_wall);
        }
    }

    // Set the reference pressure as the constant pressure in element 0
    dynamic_cast<PseudoBucklingRingElement*>(Wall_pt)
        ->set_reference_pressure_pt(fluid_mesh_pt ()->element_pt(0)
                                      ->internal_data_pt(0));
}

/// Run the time integration for ntsteps steps
void unsteady_run(const unsigned &ntsteps, const bool& restarted,
                  DocInfo& doc_info);

/// Set initial condition (incl previous timesteps) according
/// to specified function.
void set_initial_condition();

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

/// Access function for the fluid mesh
AlgebraicRefineableQuarterCircleSectorMesh<ELEMENT>* fluid_mesh_pt()
{
    return Fluid_mesh_pt;
}

/// Dump problem data.
void dump_it(ofstream& dump_file, DocInfo doc_info);

/// Read problem data.
void restart(ifstream& restart_file);

private:

    /// Write header for trace file
    void write_trace_file_header();

    /// Function pointer to set the intial condition
    FiniteElement::UnsteadyExactSolutionFctPt IC_Fct_pt;

    /// Pointer to wall
    GeomObject* Wall_pt;

    /// Pointer to fluid mesh
    AlgebraicRefineableQuarterCircleSectorMesh<ELEMENT>* Fluid_mesh_pt;

    /// Pointer to wall mesh (contains only a single GeneralisedElement)
    Mesh* Wall_mesh_pt;

    /// Trace file
    ofstream Trace_file;

    /// Pointer to node on coarsest mesh on which velocity is traced
    Node* Veloc_trace_node_pt;

    /// Pointer to node in symmtry plane on coarsest mesh at
    /// which velocity is traced
    Node* Sarah_veloc_trace_node_pt;

};

//=====
// Constructor: Pass (constant) timestep and function pointer to the solution
// that provides the initial conditions for the fluid.
//=====

template<class ELEMENT>
OscRingNSTProblem<ELEMENT>::OscRingNSTProblem(const double& dt,
FiniteElement::UnsteadyExactSolutionFctPt IC_fct_pt) : IC_Fct_pt(IC_fct_pt)

{
    // Period of oscillation
    double T=1.0;

    //Allocate the timestepper
    add_time stepper_pt(new BDF<4>);

    // Initialise timestep -- also sets the weights for all timesteppers
    // in the problem.
}

```

```

initialise_dt(dt);

// Parameters for pseudo-buckling ring
double eps_buckl=0.1; // ADJUST_PRIORITY
double ampl_ratio=-0.5; // ADJUST_PRIORITY
unsigned n_buckl=2; // ADJUST_PRIORITY
double r_0=1.0;
// Build wall geometric element
Wall_pt=new PseudoBucklingRingElement(eps_buckl,ampl_ratio,n_buckl,r_0,T,
time stepper_pt());

// Fluid mesh is suspended from wall between these two Lagrangian
// coordinates:
double xi_lo=0.0;
double xi_hi=2.0*atan(1.0);

// Fractional position of dividing line for two outer blocks in fluid mesh
double fract_mid=0.5;

// Build fluid mesh
Fluid_mesh_pt=new AlgebraicRefineableQuarterCircleSectorMesh<ELEMENT>(
Wall_pt,xi_lo,fract_mid,xi_hi,time stepper_pt());

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

// Fluid mesh is first sub-mesh
add_sub_mesh(Fluid_mesh_pt);

// Build wall mesh
Wall_mesh_pt=new Mesh;

// Wall mesh is completely empty: Add Wall element in its GeneralisedElement
// incarnation
Wall_mesh_pt->add_element_pt(dynamic_cast<GeneralisedElement*>(Wall_pt));

// Wall mesh is second sub-mesh
add_sub_mesh(Wall_mesh_pt);

// Combine all submeshes into a single Mesh
build_global_mesh();

// Extract pointer to node at center of mesh (this node exists
// at all refinement levels and can be used to doc continuous timetrace
// of velocities)
unsigned nnod=fluid_mesh_pt()->finite_element_pt(0)->nnode();
Veloc_trace_node_pt=fluid_mesh_pt()->finite_element_pt(0)->node_pt(nnod-1);

// Extract pointer to node in symmetry plane (this node exists
// at all refinement levels and can be used to doc continuous timetrace
// of velocities)
unsigned nnod_1d=dynamic_cast<ELEMENT*>(
fluid_mesh_pt()->finite_element_pt(0))->nnode_1d();
Sarah_veloc_trace_node_pt=fluid_mesh_pt()->
finite_element_pt(0)->node_pt(nnod_1d-1);

// The "pseudo-elastic" wall element is "loaded" by a pressure.
// Use the "constant" pressure component in Crouzeix Raviart
// fluid element as that pressure.
dynamic_cast<PseudoBucklingRingElement*>(Wall_pt)
->set_reference_pressure_pt(fluid_mesh_pt()
->element_pt(0)->internal_data_pt(0));

// Set the boundary conditions for this problem:
//-----
// All nodes are free by default -- just pin the ones that have
// Dirichlet conditions here.
// Bottom boundary:
unsigned ibound=0;
{
    unsigned num_nod= fluid_mesh_pt()->nboundary_node(ibound);
    for (unsigned inod=0;inod<num_nod;inod++)
    {
        // Pin vertical velocity
        {
            fluid_mesh_pt()->boundary_node_pt(ibound,inod)->pin(1);
        }
    }
}

// Ring boundary: No slip; this also implies that the velocity needs
// to be updated in response to wall motion
ibound=1;
{

```

```

unsigned num_nod=fluid_mesh_pt()->nboundary_node(ibound);
for (unsigned inod=0;inod<num_nod;inod++)
{
    // Which node are we dealing with?
    Node* node_pt=fluid_mesh_pt()->boundary_node_pt(ibound,inod);

    // Set auxiliary update function pointer to apply no-slip condition
    // to velocities whenever nodal position is updated
    node_pt->set_auxiliary_node_update_fct_pt(
        FSI_functions::apply_no_slip_on_moving_wall);

    // Pin both velocities
    for (unsigned i=0;i<2;i++)
    {
        node_pt->pin(i);
    }
}

// Left boundary:
ibound=2;
{
    unsigned num_nod=fluid_mesh_pt()->nboundary_node(ibound);
    for (unsigned inod=0;inod<num_nod;inod++)
    {
        // Pin horizontal velocity
        {
            fluid_mesh_pt()->boundary_node_pt(ibound,inod)->pin(0);
        }
    }
}

// Complete the build of all elements so they are fully functional
//-----

//Find number of elements in mesh
unsigned n_element = fluid_mesh_pt()->nelement();

// Loop over the elements to set up element-specific
// things that cannot be handled by constructor
for(unsigned i=0;i<n_element;i++)
{
    // Upcast from FiniteElement to the present element
    ELEMENT *el_pt = dynamic_cast<ELEMENT*>(fluid_mesh_pt()->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;
}

//Attach the boundary conditions to the mesh
cout << "Number of equations: " << assign_eqn_numbers() << std::endl;

// Set parameters for Sarah's asymptotic solution
//-----

// Amplitude of the oscillation
SarahBL::epsilon=static_cast<PseudoBucklingRingElement*>(Wall_pt)->
eps_buckl();

// Womersley number is the same as square root of Reynolds number
SarahBL::alpha=sqrt(Global_Physical_Variables::Re);

// Amplitude ratio
SarahBL::A=static_cast<PseudoBucklingRingElement*>(Wall_pt)->ampl_ratio();

// Buckling wavenumber
SarahBL::N=static_cast<PseudoBucklingRingElement*>(Wall_pt)->n_buckl_float();

// Frequency of oscillation (period is one)
SarahBL::Omega=2.0*MathematicalConstants::Pi;

}

//=====
/// Set initial condition: Assign previous and current values
/// of the velocity from the velocity field specified via
/// the function pointer.
///
/// Values are assigned so that the velocities and accelerations
/// are correct for current time.

```

```

//=====
template<class ELEMENT>
void OscRingNSTProblem<ELEMENT>::set_initial_condition()
{
    // Elastic wall: We're starting from a given initial state in which
    // the wall is undeformed. If set_initial_condition() is called again
    // after mesh refinement for first timestep, this needs to be reset.
    dynamic_cast<PseudoBucklingRingElement*>(Wall_pt)->set_R_0(1.0);

    // Backup time in global timestepper
    double backed_up_time=time_pt()->time();

    // Past history for velocities needs to be established for t=time0-deltat, ...
    // Then provide current values (at t=time0) which will also form
    // the initial guess for first solve at t=time0+deltat

    // Vector of exact solution values (includes pressure)
    Vector<double> soln(3);
    Vector<double> x(2);

    //Find number of nodes in mesh
    unsigned num_nod = fluid_mesh_pt()->nnode();

    // Get continuous times at previous timesteps
    int nprev_steps=time stepper_pt()->nprev_values();
    Vector<double> prev_time(nprev_steps+1);
    for (int itime=nprev_steps;itime>=0;itime--)
    {
        prev_time[itime]=time_pt()->time(unsigned(itime));
    }

    // Loop over current & previous timesteps (in outer loop because
    // the mesh might also move)
    for (int itime=nprev_steps;itime>=0;itime--)
    {
        double time=prev_time[itime];

        // Set global time (because this is how the geometric object refers
        // to continous time
        time_pt()->time()=time;

        cout << "setting IC at time =" << time << std::endl;

        // Update the fluid mesh for this value of the continuous time
        // (The wall object reads the continous time from the same
        // global time object)
        fluid_mesh_pt()->node_update();

        // Loop over the nodes to set initial guess everywhere
        for (unsigned jnod=0;jnod<num_nod;jnod++)
        {
            // Get nodal coordinates
            x[0]=fluid_mesh_pt()->node_pt(jnod)->x(0);
            x[1]=fluid_mesh_pt()->node_pt(jnod)->x(1);

            // Get intial solution
            (*IC_Fct_pt)(time,x,soln);

            // Loop over velocity directions (velocities are in soln[0] and soln[1]).
            for (unsigned i=0;i<2;i++)
            {
                fluid_mesh_pt()->node_pt(jnod)->set_value(itime,i,soln[i]);
            }

            // Loop over coordinate directions
            for (unsigned i=0;i<2;i++)
            {
                fluid_mesh_pt()->node_pt(jnod)->x(itime,i)=x[i];
            }
        }
    }

    // Reset backed up time for global timestepper
    time_pt()->time()=backed_up_time;
}

//=====
/// Doc the solution
///
//=====

```

```

template<class ELEMENT>
void OscRingNStProblem<ELEMENT>::doc_solution(DocInfo& doc_info)
{
    cout << "Doc-ing step " << doc_info.number()
        << " for time " << time stepper_pt()>time_pt()>time() << std::endl;

    ofstream some_file;
    char filename[100];

    // Number of plot points
    unsigned npts;
    npts=5;

    // Output solution on fluid mesh
    //-----
    snprintf(filename, sizeof(filename), "%s/soln%i.dat", doc_info.directory().c_str(),
             doc_info.number());
    //some_file.precision(20);
    some_file.open(filename);
    unsigned nelem=fluid_mesh_pt()>nelement();
    for (unsigned ielem=0;ielem<nelem;ielem++)
    {
        dynamic_cast<ELEMENT*>(fluid_mesh_pt()>element_pt(ielem))>
            full_output(some_file,npts);
    }
    some_file.close();

    // Plot wall posn
    //-----
    snprintf(filename, sizeof(filename), "%s/Wall%i.dat", doc_info.directory().c_str(),
             doc_info.number());
    some_file.open(filename);
    unsigned nplot=100;
    Vector<double> xi_wall(1);
    Vector<double> r_wall(2);
    for (unsigned iplot=0;iplot<nplot;iplot++)
    {
        xi_wall[0]=0.5*Pi*double(iplot)/double(nplot-1);
        Wall_pt->position(xi_wall,r_wall);
        some_file << r_wall[0] << " " << r_wall[1] << std::endl;
    }
    some_file.close();

    // Doc Sarah's asymptotic solution
    //-----
    snprintf(filename, sizeof(filename), "%s/exact_soln%i.dat", doc_info.directory().c_str(),
             doc_info.number());
    some_file.open(filename);
    fluid_mesh_pt()->output_fct(some_file,npts,
                                  time stepper_pt()>time_pt()>time(),
                                  SarahBL::full_exact_soln);
    some_file.close();

    // Get position of control point
    //-----
    Vector<double> r(2);
    Vector<double> xi(1);
    xi[0]=MathematicalConstants::Pi/2.0;
    wall_pt()>position(xi,r);

    // Get total volume (area) of computational domain, energies and average
    //-----
    // pressure
    //-----
    double area=0.0;
    double press_int=0.0;
    double diss=0.0;
    double kin_en=0.0;
    nelem=fluid_mesh_pt()>nelement();

    for (unsigned ielem=0;ielem<nelem;ielem++)
    {
        area+=fluid_mesh_pt()>finite_element_pt(ielem)>size();
        press_int+=dynamic_cast<ELEMENT*>(fluid_mesh_pt()>element_pt(ielem))
            ->pressure_integral();
        diss+=dynamic_cast<ELEMENT*>(fluid_mesh_pt()>element_pt(ielem))>
            dissipation();
        kin_en+=dynamic_cast<ELEMENT*>(fluid_mesh_pt()>element_pt(ielem))>
    }
}

```

```

    kin_energy();
}

// Total kinetic energy in entire domain
double global_kin=4.0*kin_en;

// Max/min refinement level
unsigned min_level;
unsigned max_level;
fluid_mesh_pt()->get_refinement_levels(min_level,max_level);

// Total dissipation for quarter domain
double time=time_pt()->time();
double diss_asympt=SarahBL::Total_Diss_sarah(time)/4.0;

// Asymptotic predictions for velocities at control point
Vector<double> x_sarah(2);
Vector<double> soln_sarah(3);
x_sarah[0]=Sarah_veloc_trace_node_pt->x(0);
x_sarah[1]=Sarah_veloc_trace_node_pt->x(1);
SarahBL::exact_soln(time,x_sarah,soln_sarah);

// Doc
Trace_file << time_pt()->time()
    << " " << r[1]
    << " " << global_kin
    << " " << SarahBL::Kin_energy_sarah(time_pt()->time())
    << " " << static_cast<PseudoBucklingRingElement*>(Wall_pt)->r_0()
    << " " << area
    << " " << press_int/area
    << " " << diss
    << " " << diss_asympt
    << " " << Veloc_trace_node_pt->x(0)
    << " " << Veloc_trace_node_pt->x(1)
    << " " << Veloc_trace_node_pt->value(0)
    << " " << Veloc_trace_node_pt->value(1)
    << " " << fluid_mesh_pt()->n_element()
    << " " << ndof()
    << " " << min_level
    << " " << max_level
    << " " << fluid_mesh_pt()->nrefinement_overruled()
    << " " << fluid_mesh_pt()->max_error()
    << " " << fluid_mesh_pt()->min_error()
    << " " << fluid_mesh_pt()->max_permitted_error()
    << " " << fluid_mesh_pt()->min_permitted_error()
    << " " << fluid_mesh_pt()->max_keep_unrefined()
    << " " << doc_info.number()
    << " " << Sarah_veloc_trace_node_pt->x(0)
    << " " << Sarah_veloc_trace_node_pt->x(1)
    << " " << Sarah_veloc_trace_node_pt->value(0)
    << " " << Sarah_veloc_trace_node_pt->value(1)
    << " " << x_sarah[0]
    << " " << x_sarah[1]
    << " " << soln_sarah[0]
    << " " << soln_sarah[1]
    << " "
    << static_cast<PseudoBucklingRingElement*>(Wall_pt)->r_0()-1.0
    << std::endl;

// Output fluid solution on coarse-ish mesh
//-----
// Extract all elements from quadtree representation
Vector<Tree*> all_element_pt;
fluid_mesh_pt()->forest_pt()->
stick_all_tree_nodes_into_vector(all_element_pt);

// Build a coarse mesh
Mesh* coarse_mesh_pt = new Mesh();

//Loop over all elements and check if their refinement level is
//equal to the mesh's minimum refinement level
nelem=all_element_pt.size();
for (unsigned ielem=0;ielem<nelem;ielem++)
{
    Tree* el_pt=all_element_pt[ielem];
    if (el_pt->level()==min_level)
    {
        coarse_mesh_pt->add_element_pt(el_pt->object_pt());
    }
}

// Output fluid solution on coarse mesh
snprintf(filename, sizeof(filename), "%s/coarse_soln%u.dat", doc_info.directory().c_str(),
doc_info.number());

```

```

some_file.open(filename);
nelem=coarse_mesh_pt->nElement();
for (unsigned ielem=0;ielem<nelem;ielem++)
{
    dynamic_cast<ELEMENT*>(coarse_mesh_pt->element_pt(ielem))->
        full_output(some_file,npts);
}
some_file.close();

// Write restart file
snprintf(filename, sizeof(filename), "%s/restart%i.dat", doc_info.directory().c_str(),
         doc_info.number());
some_file.open(filename);
dump_it(some_file,doc_info);
some_file.close();

}

//=====================================================================
/// Dump the solution
//=====================================================================
template<class ELEMENT>
void OscRingNSTProblem<ELEMENT>::dump_it(ofstream& dump_file,DocInfo doc_info)
{
    // Dump refinement status of mesh
    //fluid_mesh_pt()->dump_refinement(dump_file);

    // Call generic dump()
    Problem::dump(dump_file);

}

//=====================================================================
/// Read solution from disk
//=====================================================================
template<class ELEMENT>
void OscRingNSTProblem<ELEMENT>::restart(ifstream& restart_file)
{
    // Refine fluid mesh according to the instructions in restart file
    //fluid_mesh_pt()->refine(restart_file);

    // Rebuild the global mesh
    //rebuild_global_mesh();

    // Read generic problem data
    Problem::read(restart_file);

    //    // Complete build of all elements so they are fully functional
    //    finish_problem_setup();
    // Assign equation numbers
    //cout << "\nNumber of equations: " << assign_eqn_numbers()
    //      << std::endl << std::endl;
}

//=====================================================================
/// Driver for timestepping the problem: Fixed timestep but
/// guaranteed spatial accuracy. Beautiful, innit?
///
//=====================================================================
template<class ELEMENT>
void OscRingNSTProblem<ELEMENT>::unsteady_run(const unsigned& ntsteps,
                                                const bool& restarted,
                                                DocInfo& doc_info)
{
    // Open trace file
    char filename[100];
    sprintf(filename, sizeof(filename), "%s/trace.dat", doc_info.directory().c_str());
    Trace_file.open(filename);

    // Max. number of adaptations per solve
    unsigned max_adapt;

    // Max. number of adaptations per solve
    if (restarted)
    {
        max_adapt=0;
    }
}

```

```

    }
else
{
    max_adapt=1;
}
// Max. and min. error for adaptive refinement/unrefinement
fluid_mesh_pt() ->max_permitted_error()= 0.5e-2;
fluid_mesh_pt() ->min_permitted_error()= 0.5e-3;

// Don't allow refinement to drop under given level
fluid_mesh_pt() ->min_refinement_level()=2;

// Don't allow refinement beyond given level
fluid_mesh_pt() ->max_refinement_level()=6;

// Don't bother adapting the mesh if no refinement is required
// and if less than ... elements are to be merged.
fluid_mesh_pt() ->max_keep_unrefined()=20;

// Get max/min refinement levels in mesh
unsigned min_refinement_level;
unsigned max_refinement_level;
fluid_mesh_pt() ->get_refinement_levels(min_refinement_level,
                                         max_refinement_level);

cout << "\n Initial mesh: min/max refinement levels: "
    << min_refinement_level << " " << max_refinement_level << std::endl << std::endl;

// Doc refinement targets
fluid_mesh_pt() ->doc_adaptivity_targets(cout);

// Write header for trace file
write_trace_file_header();

// Doc initial condition
doc_solution(doc_info);
doc_info.number()++;

// Switch off doc during solve
doc_info.disable_doc();

// If we set first to true, then initial guess will be re-assigned
// after mesh adaptation. Don't want this if we've done a restart.
bool first;
bool shift;
if (restarted)
{
    first=false;
    shift=false;
    // Move time back by dt to make sure we're re-solving the read-in solution
    time_pt() ->time() -= time_pt() ->dt();
}
else
{
    first=true;
    shift=true;
}
//Take the first fixed timestep with specified tolerance for Newton solver
double dt=time_pt() ->dt();
unsteady_newton_solve(dt,max_adapt,first,shift);

// Switch doc back on
doc_info.enable_doc();

// Doc initial solution
doc_solution(doc_info);
doc_info.number()++;

// Now do normal run; allow for one mesh adaptation per timestep
max_adapt=1;

//Loop over the remaining timesteps
for(unsigned t=2;t<ntsteps;t++)
{
    // Switch off doc during solve
    doc_info.disable_doc();

    //Take fixed timestep
    first=false;
    unsteady_newton_solve(dt,max_adapt,first);

    // Switch doc back on
    doc_info.enable_doc();

    // Doc solution
}

```

```

//if (icount%10==0)
{
    doc_solution(doc_info);
    doc_info.number()++;
}
}

// Close trace file
Trace_file.close();
}

//=====================================================================
/// Write trace file header
//=====================================================================
template<class ELEMENT>
void OscRingNSTProblem<ELEMENT>::write_trace_file_header()
{
    // Doc parameters in trace file
    Trace_file << "# err_max " << fluid_mesh_pt()->max_permitted_error() << std::endl;
    Trace_file << "# err_min " << fluid_mesh_pt()->min_permitted_error() << std::endl;
    Trace_file << "# Re " << Global_Physical_Variables::Re << std::endl;
    Trace_file << "# St " << Global_Physical_Variables::ReSt/
        Global_Physical_Variables::Re << std::endl;
    Trace_file << "# dt " << time_stripper_pt()->time_pt()->dt() << std::endl;
    Trace_file << "# initial # elements " << mesh_pt()->nElement() << std::endl;
    Trace_file << "# min_refinement_level "
        << fluid_mesh_pt()->min_refinement_level() << std::endl;
    Trace_file << "# max_refinement_level "
        << fluid_mesh_pt()->max_refinement_level() << std::endl;

    Trace_file << "VARIABLES=time\,,"V_c_t_r_l\,,"e_k_i_n\";
    Trace_file << ",\e_k_i_n_(\a_s_y_m_p_t)\,,"R_0\,,"Area\";
    Trace_file << ",\Average pressure\,,"Total dissipation (quarter domain)\";
    Trace_file << ",\Asymptotic dissipation (quarter domain)\";
    Trace_file << ",\x<sub>1</sub><sup>(track)</sup>\";
    Trace_file << ",\x<sub>2</sub><sup>(track)</sup>\";
    Trace_file << ",\u<sub>1</sub><sup>(track)</sup>\";
    Trace_file << ",\u<sub>2</sub><sup>(track)</sup>\";
    Trace_file << ",\N<sub>element</sub>\";
    Trace_file << ",\N<sub>dof</sub>\";
    Trace_file << ",\max. refinement level\";
    Trace_file << ",\min. refinement level\";
    Trace_file << ",\# of elements whose refinement was over-ruled\";
    Trace_file << ",\max. error\";
    Trace_file << ",\min. error\";
    Trace_file << ",\max. permitted error\";
    Trace_file << ",\min. permitted error\";
    Trace_file << ",\max. permitted # of unrefined elements\";
    Trace_file << ",\doc number\";
    Trace_file << ",\x<sub>1</sub><sup>(track2 FE)</sup>\";
    Trace_file << ",\x<sub>2</sub><sup>(track2 FE)</sup>\";
    Trace_file << ",\u<sub>1</sub><sup>(track2 FE)</sup>\";
    Trace_file << ",\u<sub>2</sub><sup>(track2 FE)</sup>\";
    Trace_file << ",\x<sub>1</sub><sup>(track2 Sarah)</sup>\";
    Trace_file << ",\x<sub>2</sub><sup>(track2 Sarah)</sup>\";
    Trace_file << ",\u<sub>1</sub><sup>(track2 Sarah)</sup>\";
    Trace_file << ",\u<sub>2</sub><sup>(track2 Sarah)</sup>\";
    Trace_file << ",\R<sub>0</sub>-1\";
    Trace_file << std::endl;

}

//=====================================================================
/// Demonstrate how to solve OscRingNST problem in deformable domain
/// with mesh adaptation
//=====================================================================
int main(int argc, char* argv[])

```

```
{
    // Store command line arguments
    CommandLineArgs::setup(argc,argv);

    //Do a certain number of timesteps per period
    unsigned nstep_per_period=40; // 80; // ADJUST_PRIORITY
    unsigned nperiod=3;

    // Work out total number of steps and timestep
    unsigned nstep=nstep_per_period*nperiod;
    double dt=1.0/double(nstep_per_period);

    // Set up the problem: Pass timestep and Sarah's asymptotic solution for
    // generation of initial condition
    OscRingNStProblem<AlgebraicElement<RefineableQCrouzeixRaviartElement<2>> >
    problem(dt,&SarahBL::full_exact_soln);

    // Restart?
    //-----
    bool restarted=false;

    // Pointer to restart file
    ifstream* restart_file_pt=0;

    // No restart
    //-----
    if (CommandLineArgs::Argc!=2)
    {
        cout << "No restart" << std::endl;
        restarted=false;

        // Refine uniformly
        problem.refine_uniformly();
        problem.refine_uniformly();
        problem.refine_uniformly();

        // Set initial condition on uniformly refined domain (if this isn't done
        // then the mesh contains the interpolation of the initial guess
        // on the original coarse mesh -- if the nodal values were zero in
        // the interior and nonzero on the boundary, then the waviness of
        // the interpolated i.g. between the nodes on the coarse mesh
        // gets transferred onto the fine mesh where we can do better
        problem.set_initial_condition();
    }

    // Restart
    //-----
    else if (CommandLineArgs::Argc==2)
    {
        restarted=true;

        // Open restart file
        restart_file_pt=new ifstream(CommandLineArgs::Argv[1],ios_base::in);
        if (restart_file_pt!=0)
        {
            cout << "Have opened " << CommandLineArgs::Argv[1] <<
                " for restart. " << std::endl;
        }
        else
        {
            std::ostringstream error_stream;
            error_stream << "ERROR while trying to open "
                << CommandLineArgs::Argv[2]
                << " for restart." << std::endl;

            throw OomphLibError(error_stream.str(),
                                OOMPH_CURRENT_FUNCTION,
                                OOMPH_EXCEPTION_LOCATION);
        }
        // Do the actual restart
        problem.restart(*restart_file_pt);
    }

    // Two command line arguments: do validation run
    if (CommandLineArgs::Argc==3)
    {
        nstep=3;
        cout << "Only doing nstep steps for validation: " << nstep << std::endl;
    }

    // Setup labels for output
}
```

```
DocInfo doc_info;
// Output directory
doc_info.set_directory("RESLT");

// Do unsteady run of the problem for nstep steps
//-----
problem.unsteady_run(nstep,restarted,doc_info);

// Validate the restart procedure
//-----
if (CommandLineArgs::Argc==3)
{
    // Build problem and restart

    // Set up the problem: Pass timestep and Sarah's asymptotic solution for
    // generation of initial condition
    OscRingNStProblem<AlgebraicElement<RefineableQCrouzeixRaviartElement<2>> >
    restarted_problem(dt,&SarahBL::full_exact_soln);

    // Setup labels for output
    DocInfo restarted_doc_info;

    // Output directory
    restarted_doc_info.set_directory("RESLT_restarted");

    // Step number
    restarted_doc_info.number()=0;

    // Copy by performing a restart from old problem
    restart_file_pt=new ifstream("RESLT/restart2.dat");

    // Do the actual restart
    restarted_problem.restart(*restart_file_pt);

    // Do unsteady run of the problem for one step
    unsigned nstep=2;
    bool restarted=true;
    restarted_problem.unsteady_run(nstep,restarted,restarted_doc_info);
}

}
```

1.1 PDF file

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