

# Chapter 1

## Demo problem: Boundary-driven elastic deformation of a fish-shaped domain

Detailed documentation to be written. Here's a plot of the result and the already fairly well documented driver code...

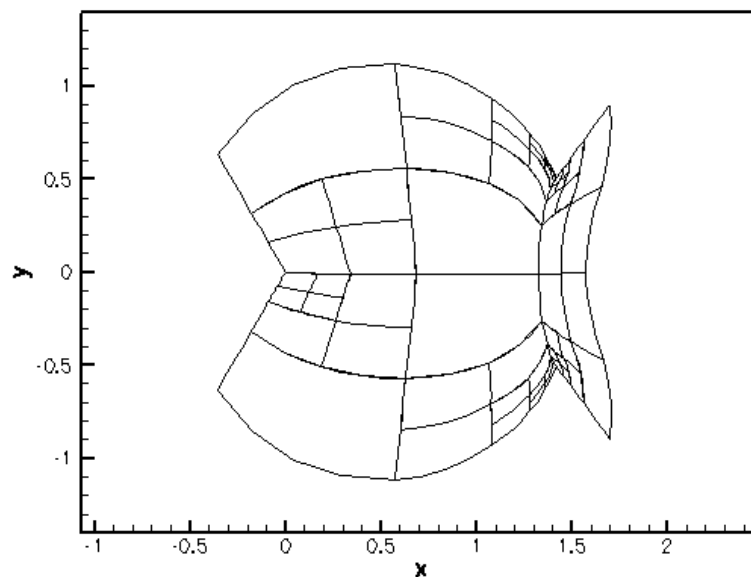


Figure 1.1 Boundary-driven elastic deformation of a fish-shaped domain.

```
//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) 2006-2025 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 elastic deformation of a fish-shaped domain with
// adaptivity -- deformation is driven by displacement of
// GeomObject-based boundary!

// Generic oomph-lib headers
#include "generic.h"

// Solid mechanics
#include "solid.h"

// The fish mesh
#include "meshes/fish_mesh.h"

using namespace std;

using namespace oomph;

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

//=====
/// Refineable fish mesh upgraded to become a solid mesh
//=====
template<class ELEMENT>
class ElasticFishMesh : public virtual RefineableFishMesh<ELEMENT>,
                        public virtual SolidMesh
{
public:

    /// Constructor: Build underlying adaptive fish mesh and then
    /// set current Eulerian coordinates to be the Lagrangian ones.
    /// Pass pointer to geometric objects that specify the
    /// fish's back in the "current" and "undeformed" configurations,
    /// and pointer to timestepper (defaults to Static)
    /// Note: FishMesh is virtual base and its constructor is automatically
    /// called first! --> this is where we need to build the mesh;
    /// the constructors of the derived meshes don't call the
    /// base constructor again and simply add the extra functionality.
    ElasticFishMesh(GeomObject* back_pt, GeomObject* undeformed_back_pt,
                    TimeStepper* time_stepper_pt=&Mesh::Default_TimeStepper) :
        FishMesh<ELEMENT>(back_pt,time_stepper_pt),
        RefineableFishMesh<ELEMENT>(back_pt,time_stepper_pt)
    {
        // Mesh has been built, adaptivity etc has been set up -->
        // assign the Lagrangian coordinates so that the current
        // configuration becomes the stress-free initial configuration
        set_lagrangian_nodal_coordinates();

        // Build "undeformed" domain: This is a "deep" copy of the
        // Domain that we used to create set the Eulerian coordinates
        // in the initial mesh -- the original domain (accessible via
        // the private member data Domain_pt) will be used to update
        // the position of the boundary nodes; the copy that we're
        // creating here will be used to determine the Lagrangian coordinates
        // of any newly created SolidNodes during mesh refinement
        double xi_nose = this->Domain_pt->xi_nose();
        double xi_tail = this->Domain_pt->xi_tail();
        Undeformed_domain_pt=new FishDomain(undeformed_back_pt,xi_nose,xi_tail);

        // Loop over all elements and set the undeformed macro element pointer
        unsigned n_element=this->nelement();
        for (unsigned e=0;e<n_element;e++)
        {
            // Get pointer to full element type
            ELEMENT* el_pt=dynamic_cast<ELEMENT*>(this->element_pt(e));

            // Set pointer to macro element so the curvilinear boundaries
            // of the undeformed mesh/domain get picked up during adaptive
            // mesh refinement
            el_pt->set_undeformed_macro_elem_pt(
                Undeformed_domain_pt->macro_element_pt(e));

            // Use MacroElement representation for
            // Lagrangian coordinates of newly created
            // nodes
            el_pt->enable_use_of_undeformed_macro_element_for_new_lagrangian_coords();
        }
    }
}

```

```

    }

    /// Destructor: Kill "undeformed" Domain
    virtual ~ElasticFishMesh()
    {
        delete Undeformed_domain_pt;
    }

private:

    /// Pointer to "undeformed" Domain -- used to determine the
    /// Lagrangian coordinates of any newly created SolidNodes during
    /// Mesh refinement
    Domain* Undeformed_domain_pt;

};

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

//=====
/// Global variables
//=====
namespace Global_Physical_Variables
{
    /// Pointer to strain energy function
    StrainEnergyFunction* Strain_energy_function_pt;

    /// Pointer to constitutive law
    ConstitutiveLaw* Constitutive_law_pt;

    /// Elastic modulus
    double E=1.0;

    /// Poisson's ratio
    double Nu=0.3;

    /// "Mooney Rivlin" coefficient for generalised Mooney Rivlin law
    double C1=1.3;

    /// Body force
    double Gravity=0.0;

    /// Body force vector: Vertically downwards with magnitude Gravity
    void body_force(const double& t,
                   const Vector<double>& xi,
                   Vector<double>& b)
    {
        b[0]=0.0;
        b[1]=-Gravity;
    }
}

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

//=====
/// Boundary-driven elastic deformation of fish-shaped domain.
//=====
template<class ELEMENT>
class ElasticFishProblem : public Problem
{
public:

    /// Constructor:
    ElasticFishProblem();

    /// Run simulation.
    void run();

    /// Access function for the mesh
    ElasticFishMesh<ELEMENT>* mesh_pt()

```

```

    {return dynamic_cast<ElasticFishMesh<ELEMENT>*>(Problem::mesh_pt());}

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

    /// Update function (empty)
    void actions_after_newton_solve() {}

    /// Update before solve: We're dealing with a static problem so
    /// the nodal positions before the next solve merely serve as
    /// initial conditions. For meshes that are very strongly refined
    /// near the boundary, the update of the displacement boundary
    /// conditions (which only moves the SolidNodes *on* the boundary),
    /// can lead to strongly distorted meshes. This can cause the
    /// Newton method to fail --> the overall method is actually more robust
    /// if we use the nodal positions as determined by the Domain/MacroElement-
    /// based mesh update as initial guesses.
    void actions_before_newton_solve()
    {
        bool update_all_solid_nodes=true;
        mesh_pt()->node_update(update_all_solid_nodes);
    }

    /// Update after adapt: Pin all redundant pressure nodes (if required)
    void actions_after_adapt()
    {
        // Pin the redundant solid pressures (if any)
        PVDEquationsBase<2>::pin_redundant_nodal_solid_pressures(
            mesh_pt()->element_pt());
    }

private:

    // Geometric object that represents the deformable fish back
    Circle* Fish_back_pt;

};

//=====
/// Constructor:
//=====
template<class ELEMENT>
ElasticFishProblem<ELEMENT>::ElasticFishProblem()
{
    // Set coordinates and radius for the circle that will become the fish back
    double x_c=0.5;
    double y_c=0.0;
    double r_back=1.0;

    // Build geometric object that will become the deformable fish back
    //GeomObject* fish_back_pt=new ElasticFishBackElement(x_c,y_c,r_back);
    Fish_back_pt=new Circle(x_c,y_c,r_back);

    // Build geometric object that specifies the fish back in the
    // undeformed configuration (basically a deep copy of the previous one)
    GeomObject* undeformed_fish_back_pt=new Circle(x_c,y_c,r_back);

    // Build fish mesh with geometric objects that specify the deformable
    // and undeformed fish back
    Problem::mesh_pt()=new ElasticFishMesh<ELEMENT>(Fish_back_pt,
                                                    undeformed_fish_back_pt);

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

    // Change/doc targets for mesh adaptation
    mesh_pt()->max_permitted_error()=0.05;
    mesh_pt()->min_permitted_error()=0.005;
    mesh_pt()->doc_adaptivity_targets(cout);

    // Pin all nodal positions apart from those on the tail
    unsigned num_bound = mesh_pt()->nboundary();
    for(unsigned ibound=0;ibound<num_bound;ibound++)
    {
        if (ibound!=2)
        {
            unsigned num_nod=mesh_pt()->nboundary_node(ibound);
            for (unsigned inod=0;inod<num_nod;inod++)
            {
                for (unsigned i=0;i<2;i++)
                {
                    mesh_pt()->boundary_node_pt(ibound,inod)->pin_position(i);
                }
            }
        }
    }
}

```

```

    }
}

//Loop over the elements in the mesh to set parameters/function pointers
unsigned n_element = mesh_pt()->nelement();
for(unsigned i=0;i<n_element;i++)
{
    //Cast to a solid element
    ELEMENT *el_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(i));

    // Set the constitutive law
    el_pt->constitutive_law_pt() =
        Global_Physical_Variables::Constitutive_law_pt;

    // Set the body force
    el_pt->body_force_fct_pt()=Global_Physical_Variables::body_force;
}

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

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

// Refine the problem uniformly (this automatically passes the
// function pointers/parameters to the finer elements
refine_uniformly();

// The non-pinned positions of the newly SolidNodes will have been
// determined by interpolation. Update all solid nodes based on
// the Mesh's Domain/MacroElement representation.
bool update_all_solid_nodes=true;
mesh_pt()->node_update(update_all_solid_nodes);

// Now set the Eulerian equal to the Lagrangian coordinates
mesh_pt()->set_lagrangian_nodal_coordinates();
}

//=====
/// Doc the solution
//=====
template<class ELEMENT>
void ElasticFishProblem<ELEMENT>::doc_solution(DocInfo& doc_info)
{
    ofstream some_file;
    char filename[100];

    // Number of plot points
    unsigned npts = 5;

    // Output shape of deformed body
    snprintf(filename, sizeof(filename), "%s/soln%i.dat", doc_info.directory().c_str(),
        doc_info.number());
    some_file.open(filename);
    mesh_pt()->output(some_file, npts);
    some_file.close();

    // removed until Jacobi eigensolver is re-instated
    // // Output principal stress vectors at the centre of all elements
    // SolidHelpers::doc_2D_principal_stress<ELEMENT>(doc_info, mesh_pt());
}

//=====
/// Run the problem
//=====
template<class ELEMENT>
void ElasticFishProblem<ELEMENT>::run()
{
    // Output
    DocInfo doc_info;

    // Set output directory
    doc_info.set_directory("RESULT");

    // Step number
    doc_info.number()=0;
    // Initial parameter values
    // Gravity:
    Global_Physical_Variables::Gravity=0.1;
    //Parameter incrementation
    unsigned nstep=5;

```

```

for(unsigned i=0;i<nstep;i++)
{
    //Solve the problem with Newton's method, allowing for up to 5
    //rounds of adaptation
    newton_solve(5);

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

    // Increment width
    Fish_back_pt->y_c()+=0.03;
}

}

//=====
/// Driver for simple elastic problem
//=====
int main()
{
    //Initialise physical parameters
    Global_Physical_Variables::E = 2.1;
    Global_Physical_Variables::Nu = 0.4;
    Global_Physical_Variables::Cl = 1.3;
    // Define a strain energy function: Generalised Mooney Rivlin
    Global_Physical_Variables::Strain_energy_function_pt =
        new GeneralisedMooneyRivlin(&Global_Physical_Variables::Nu,
                                    &Global_Physical_Variables::Cl,
                                    &Global_Physical_Variables::E);
    // Define a constitutive law (based on strain energy function)
    Global_Physical_Variables::Constitutive_law_pt =
        new IsotropicStrainEnergyFunctionConstitutiveLaw(
            Global_Physical_Variables::Strain_energy_function_pt);
    //Set up the problem with pure displacement formulation
    ElasticFishProblem<RefineableQPVElement<2,3> > problem;
    problem.run();
}

```

---

## 1.1 PDF file

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