

## Chapter 1

# Demo problem: Solution of a "free-boundary" Poisson problem in an "elastic" domain revisited – this time with AlgebraicElements

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



Figure 1.1 Adaptive solution of Poisson's equation in a fish-shaped domain for various 'widths' of the domain.

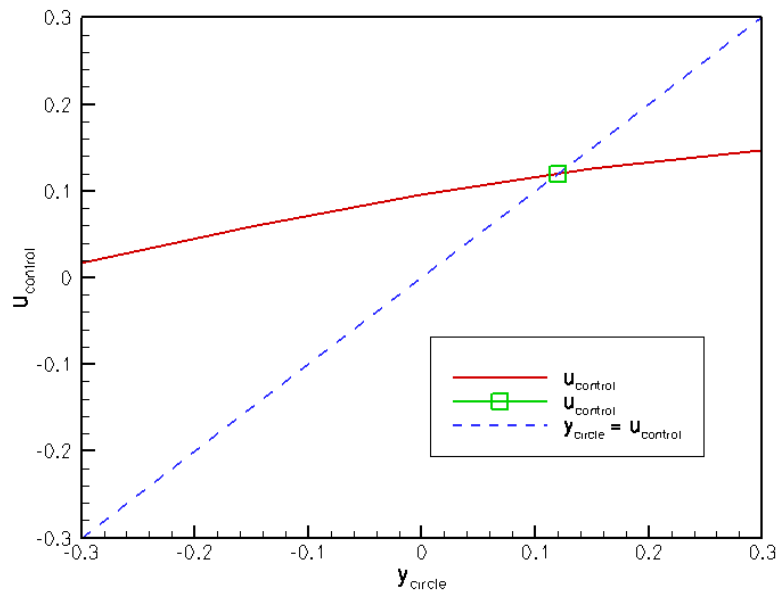


Figure 1.2 Solution of Poisson's equation at a control node as a function of the 'width' of the domain.

```
//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) 2006-2021 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//=====
// Driver for solution of "free boundary" 2D Poisson equation in
// fish-shaped domain with adaptivity

// Generic oomph-lib headers
#include "generic.h"
// The Poisson equations
#include "poisson.h"
// The fish mesh
#include "meshes/fish_mesh.h"
// Circle as generalised element:
#include "circle_as_generalised_element.h"
using namespace std;
using namespace oomph;

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

//=====
// Namespace for const source term in Poisson equation
//=====
namespace ConstSourceForPoisson
{
    // Strength of source function: default value 1.0
    double Strength=1.0;
}
```

```

/// Const source function
void get_source(const Vector<double>& x, double& source)
{
    source = -Strength*(1.0+x[0]*x[1]);
}

}

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

//=====
/// Refineable Poisson problem in deformable fish-shaped domain.
/// Template parameter identify the elements.
//=====
template<class ELEMENT>
class RefineableFishPoissonProblem : public Problem
{
public:

    /// \short Constructor: Bool flag specifies if position of fish back is
    /// prescribed or computed from the coupled problem. String specifies
    /// output directory.
    RefineableFishPoissonProblem(
        const bool& fix_position, const string& directory_name,
        const unsigned& i_case);

    /// Destructor
    virtual ~RefineableFishPoissonProblem();

    /// \short Update after Newton step: Update mesh in response to
    /// possible changes in the wall shape
    void actions_before_newton_convergence_check()
    {
        fish_mesh_pt()->node_update();
    }

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

    /// Update the problem specs before solve: Update mesh
    void actions_before_newton_solve()
    {
        fish_mesh_pt()->node_update();
    }
    //Access function for the fish mesh
    AlgebraicRefineableFishMesh<ELEMENT>* fish_mesh_pt()
    {
        return Fish_mesh_pt;
    }

    /// Return value of the "load" on the elastically supported ring
    double& load()
    {
        return *Load_pt->value_pt(0);
    }

    /// \short Return value of the vertical displacement of the ring that
    /// represents the fish's back
    double& y_c()
    {
        return static_cast<ElasticallySupportedRingElement*>(fish_mesh_pt()->
            fish_back_pt()->y_c());
    }

    /// Doc the solution
    void doc_solution();

    /// Access to DocInfo object
    DocInfo& doc_info() {return Doc_info;}
private:

    /// Helper fct to set method for evaluation of shape derivs
    void set_shape_deriv_method()
    {
        bool done=false;
        //Loop over elements and set pointers to source function
        unsigned n_element = fish_mesh_pt()->nelement();
        for(unsigned i=0;i<n_element;i++)
        {
            // Upcast from FiniteElement to the present element
            ELEMENT *el_pt = dynamic_cast<ELEMENT*>(fish_mesh_pt()->element_pt(i));

            // Direct FD
            if (Case_id==0)

```

```

    {
        el_pt->evaluate_shape_derivs_by_direct_fd();
        if (!done) std::cout << "\n\n [CR residuals] Direct FD" << std::endl;
    }
    // Chain rule with/without FD
    else if ( (Case_id==1) || (Case_id==2) )
    {
        // It's broken but let's call it anyway to keep self-test alive
        bool i_know_what_i_am_doing=true;
        el_pt->evaluate_shape_derivs_by_chain_rule(i_know_what_i_am_doing);
        if (Case_id==1)
        {
            el_pt->enable_always_evaluate_dresidual_dnodal_coordinates_by_fd();
            if (!done) std::cout << "\n\n [CR residuals] Chain rule and FD"
                                << std::endl;
        }
        else
        {
            el_pt->disable_always_evaluate_dresidual_dnodal_coordinates_by_fd();
            if (!done) std::cout << "\n\n [CR residuals] Chain rule and analytic"
                                << std::endl;
        }
    }
    // Fastest with/without FD
    else if ( (Case_id==3) || (Case_id==4) )
    {
        // It's broken but let's call it anyway to keep self-test alive
        bool i_know_what_i_am_doing=true;
        el_pt->evaluate_shape_derivs_by_fastest_method(i_know_what_i_am_doing);
        if (Case_id==3)
        {
            el_pt->enable_always_evaluate_dresidual_dnodal_coordinates_by_fd();
            if (!done) std::cout << "\n\n [CR residuals] Fastest and FD"
                                << std::endl;
        }
        else
        {
            el_pt->disable_always_evaluate_dresidual_dnodal_coordinates_by_fd();
            if (!done) std::cout << "\n\n [CR residuals] Fastest and analytic"
                                << std::endl;
        }
    }
    }
    done=true;
}

}

/// Node at which the solution of the Poisson equation is documented
Node* Doc_node_pt;

/// Trace file
ofstream Trace_file;

/// Pointer to fish mesh
AlgebraicRefineableFishMesh<ELEMENT>* Fish_mesh_pt;

/// Pointer to single-element mesh that stores the GeneralisedElement
/// that represents the fish back
Mesh* Fish_back_mesh_pt;

/// \short Pointer to data item that stores the "load" on the fish back
Data* Load_pt;

/// \short Is the position of the fish back prescribed?
bool Fix_position;

/// Doc info object
DocInfo Doc_info;

/// Case id
unsigned Case_id;
};
//=====
/// Constructor for adaptive Poisson problem in deformable fish-shaped
/// domain. Pass flag if position of fish back is fixed, and the output
/// directory.
//=====
template<class ELEMENT>
RefineableFishPoissonProblem<ELEMENT>::RefineableFishPoissonProblem(
    const bool& fix_position, const string& directory_name,
    const unsigned& i_case) : Fix_position(fix_position), Case_id(i_case)
{
    // Set output directory
    Doc_info.set_directory(directory_name);

    // Initialise step number

```

```

Doc_info.number()=0;

// Open trace file
char filename[100];
sprintf(filename,"%s/trace.dat",directory_name.c_str());
Trace_file.open(filename);
Trace_file
« "VARIABLES=\load\","y<sub>circle</sub>\","u<sub>control</sub>\""
« std::endl;
// 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 element that will become the fish back
GeomObject* fish_back_pt=new ElasticallySupportedRingElement(x_c,y_c,r_back);
// Build fish mesh with geometric object that specifies the fish back
Fish_mesh_pt=new AlgebraicRefineableFishMesh<ELEMENT>(fish_back_pt);
// Add the fish mesh to the problem's collection of submeshes:
add_sub_mesh(Fish_mesh_pt);
// Build mesh that will store only the geometric wall element
Fish_back_mesh_pt=new Mesh;
// So far, the mesh is completely empty. Let's add the
// one (and only!) GeneralisedElement which represents the shape
// of the fish's back to it:
Fish_back_mesh_pt->add_element_pt(dynamic_cast<GeneralisedElement*>(
    Fish_mesh_pt->fish_back_pt()));
// Add the fish back mesh to the problem's collection of submeshes:
add_sub_mesh(Fish_back_mesh_pt);
// Now build global mesh from the submeshes
build_global_mesh();

// Create/set error estimator
fish_mesh_pt()->spatial_error_estimator_pt()=new Z2ErrorEstimator;

// Choose a node at which the solution is documented: Choose
// the central node that is shared by all four elements in
// the base mesh because it exists at all refinement levels.

// How many nodes does element 0 have?
unsigned nnod=fish_mesh_pt()->finite_element_pt(0)->nnod();
// The central node is the last node in element 0:
Doc_node_pt=fish_mesh_pt()->finite_element_pt(0)->node_pt(nnod-1);
// Doc
cout « std::endl « "Control node is located at: "
« Doc_node_pt->x(0) « " " « Doc_node_pt->x(1)
« std::endl « std::endl;
// Position of fish back is prescribed
if (Fix_position)
{
    // Create the load data object
    Load_pt=new Data(1);

    // Pin the prescribed load
    Load_pt->pin(0);
    // Pin the vertical displacement
    dynamic_cast<ElasticallySupportedRingElement*>(
        Fish_mesh_pt->fish_back_pt())->pin_yc();
}
// Coupled problem: The position of the fish back is determined
// via the solution of the Poisson equation: The solution at
// the control node acts as the load for the displacement of the
// fish back
else
{
    // Use the solution (value 0) at the control node as the load
    // that acts on the ring. [Note: Node == Data by inheritance]
    Load_pt=Doc_node_pt;
}
// Set the pointer to the Data object that specifies the
// load on the fish's back
dynamic_cast<ElasticallySupportedRingElement*>(Fish_mesh_pt->fish_back_pt())->
    set_load_pt(Load_pt);

// Set the boundary conditions for this problem: All nodes are
// free by default -- just pin the ones that have Dirichlet conditions
// here.
unsigned num_bound = fish_mesh_pt()->nboundary();
for(unsigned ibound=0;ibound<num_bound;ibound++)
{
    unsigned num_nod= fish_mesh_pt()->nboundary_node(ibound);
    for (unsigned inod=0;inod<num_nod;inod++)
    {
        fish_mesh_pt()->boundary_node_pt(ibound,inod)->pin(0);
    }
}
// Set homogeneous boundary conditions on all boundaries
for(unsigned ibound=0;ibound<num_bound;ibound++)

```

```

{
    // Loop over the nodes on boundary
    unsigned num_nod=fish_mesh_pt()->nboundary_node(ibound);
    for (unsigned inod=0;inod<num_nod;inod++)
    {
        fish_mesh_pt()->boundary_node_pt(ibound,inod)->set_value(0,0.0);
    }
}

// Loop over elements and set pointers to source function
unsigned n_element = fish_mesh_pt()->nelement();
for(unsigned i=0;i<n_element;i++)
{
    // Upcast from FiniteElement to the present element
    ELEMENT *el_pt = dynamic_cast<ELEMENT*>(fish_mesh_pt()->element_pt(i));

    //Set the source function pointer
    el_pt->source_fct_pt() = &ConstSourceForPoisson::get_source;
}
// Set shape derivative method
set_shape_deriv_method();
// Do equation numbering
cout <<"Number of equations: " << assign_eqn_numbers() << std::endl;
}
//=====
// Destructeur for Poisson problem in deformable fish-shaped domain.
//=====
template<class ELEMENT>
RefineableFishPoissonProblem<ELEMENT>::~RefineableFishPoissonProblem()
{
    // Close trace file
    Trace_file.close();
}
//=====
// Doc the solution in tecplot format.
//=====
template<class ELEMENT>
void RefineableFishPoissonProblem<ELEMENT>::doc_solution()
{
    ofstream some_file;
    char filename[100];
    // Number of plot points in each coordinate direction.
    unsigned npts;
    npts=5;
    // Output solution
    if (Case_id!=0)
    {
        sprintf(filename,"%s/soln_%i_%i.dat",Doc_info.directory().c_str(),
            Case_id,Doc_info.number());
    }
    else
    {
        sprintf(filename,"%s/soln%i.dat",Doc_info.directory().c_str(),
            Doc_info.number());
    }
    some_file.open(filename);
    fish_mesh_pt()->output(some_file,npts);
    some_file.close();
    // Write "load", vertical position of the fish back, and solution at
    // control node to trace file
    Trace_file
    << static_cast<ElasticallySupportedRingElement*>(fish_mesh_pt()->
        fish_back_pt()->load()
    << " "
    << static_cast<ElasticallySupportedRingElement*>(fish_mesh_pt()->
        fish_back_pt()->y_c()
    << " " << Doc_node_pt->value(0) << std::endl;
}

//=====
// Demonstrate how to solve 2D Poisson problem in deformable
// fish-shaped domain with mesh adaptation.
//=====
template<class ELEMENT>
void demo_fish_poisson(const string& directory_name)
{
    // Set up the problem with prescribed displacement of fish back
    bool fix_position=true;
    RefineableFishPoissonProblem<ELEMENT> problem(fix_position,directory_name,0);
    // Doc refinement targets
    problem.fish_mesh_pt()->doc_adaptivity_targets(cout);
}

```

---

```

// Do some uniform mesh refinement first
//-----
problem.refine_uniformly();
problem.refine_uniformly();
// Initial value for the vertical displacement of the fish's back
problem.y_c()=-0.3;
// Loop for different fish shapes
//-----
// Number of steps
unsigned nstep=5;
// Increment in displacement
double dyc=0.6/double(nstep-1);
// Valiation: Just do one step
if (CommandLineArgs::Argc>1) nstep=1;

for (unsigned istep=0;istep<nstep;istep++)
{
    // Solve/doc
    unsigned max_solve=2;
    problem.newton_solve(max_solve);
    problem.doc_solution();

    //Increment counter for solutions
    problem.doc_info().number()++;

    // Change vertical displacement
    problem.y_c()+=dyc;
}
}
//=====
/// Demonstrate how to solve "elastic" 2D Poisson problem in deformable
/// fish-shaped domain with mesh adaptation.
//=====
template<class ELEMENT>
void demo_elastic_fish_poisson(const string& directory_name)
{
    // Loop over all cases
    for (unsigned i_case=0;i_case<5;i_case++)
    //unsigned i_case=1;
    {
        std::cout << "[CR residuals] " << std::endl;
        std::cout << "[CR residuals]===== "
                    << std::endl;
        std::cout << "[CR residuals] " << std::endl;
        //Set up the problem with "elastic" fish back
        bool fix_position=false;
        RefineableFishPoissonProblem<ELEMENT> problem(fix_position,
                                                    directory_name,
                                                    i_case);

        // Doc refinement targets
        problem.fish_mesh_pt()->doc_adaptivity_targets(cout);

        // Do some uniform mesh refinement first
        //-----
        problem.refine_uniformly();
        problem.refine_uniformly();

        // Initial value for load on fish back
        problem.load()=0.0;

        // Solve/doc
        unsigned max_solve=2;
        problem.newton_solve(max_solve);
        problem.doc_solution();
    }
}
//=====
/// Driver for "elastic" fish poisson solver with adaptation.
/// 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);
    // Shorthand for element type
    typedef AlgebraicElement<RefineableQPoissonElement<2,3> > ELEMENT;
    // Compute solution of Poisson equation in various domains
    demo_fish_poisson<ELEMENT>("RESULT");
    // Compute "elastic" coupled solution directly
    demo_elastic_fish_poisson<ELEMENT>("RESULT_coupled");
}

```

---

## 1.1 PDF file

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