Chapter 1

Example problem: The spatially-adaptive solution of the azimuthally Fourier-decomposed 3D Helmholtz equation

In this document we discuss the spatially-adaptive finite-element-based solution of the 3D Helmholtz equation in cylindrical polar coordinates, using a Fourier-decomposition of the solution in the azimuthal direction.

The driver code is very similar to the one discussed in another tutorial – the main purpose of the current tutorial is to demonstrate the use of spatial adaptivity on unstructured meshes.

1.1 A specific example

We will solve the azimuthally Fourier-decomposed Helmholtz equation

$$\nabla^2 u_N(r,z) + \left(k^2 - \frac{N^2}{r^2}\right) u_N(r,z) = 0, \qquad (1)$$

where N is the azimuthal wavenumber, in the finite domain $1 < \sqrt{r^2+z^2} < 3$. We impose the Sommerfeld radiation condition at the outer boundary of the computational domain at $\sqrt{r^2+z^2} = 3$, using a Dirichlet-to-↩ Neumann mapping, and apply flux boundary condition on the surface of the unit-sphere (where $\sqrt{r^2+z^2} = 1$) such that the exact solution is given by

$$u_N(r,z) = u_N^{[exact]}(r,z) = \sum_{l=N}^{N_{\rm terms}} h_l^{(1)}(k\sqrt{r^2+z^2})\, P_l^N\left(\frac{z}{\sqrt{r^2+z^2}}\right).$$

This solution corresponds to the superposition of several outgoing waves that emerge from the unit sphere.
The two plots below show a comparison between the exact and computed solutions for $N_{\rm terms} = 6$, a Fourier wavenumber of $N = 1$, and a (squared) Helmholtz wavenumber of $k^2 = 10$.

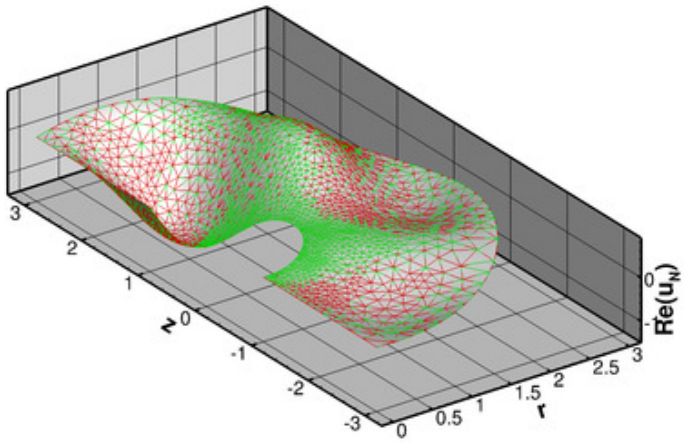

Figure 1.1 Plot of the exact (green) and computed (red) real parts of the solution of the Fourier-decomposed Helmholtz equation for N=1 and a wavenumber of k^2 = 10.

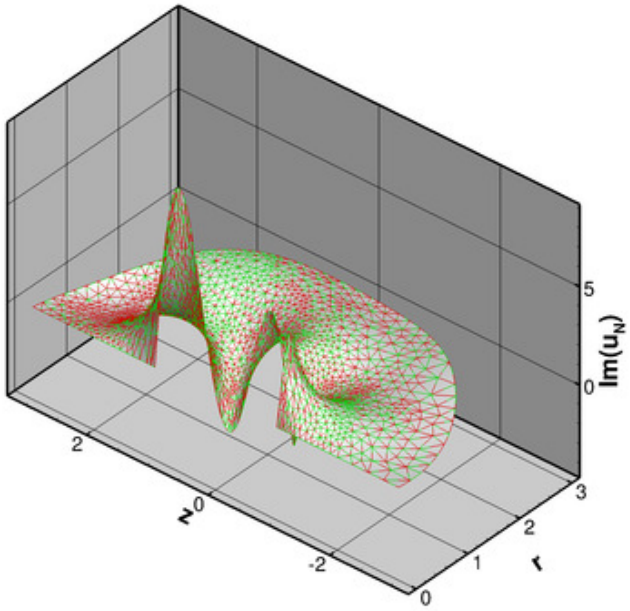

Figure 1.2 Plot of the exact (green) and computed (red) imaginary parts of the solution of the Fourier-decomposed Helmholtz equation for N=1 and a wavenumber of k^2 = 10.

1.2 The numerical solution

The driver code for this problem is very similar to the one discussed in another tutorial. Running `sdiff` on the driver codes

demo_drivers/fourier_decomposed_helmholtz/sphere_scattering/sphere_↩ scattering.cc

and

demo_drivers/fourier_decomposed_helmholtz/sphere_scattering/unstructured_sphere_scattering.cc

shows the main differences required to discretise the computational domain with an adaptive, unstructured mesh:

- The provision of the functions actions_before/after_adapt() to detach/re-attach the FaceElements that are used to enforce the Neumann boundary conditions before and after every spatial adaptation, and to pass the physical parameters to the newly created bulk elements.

- The generation of an unstructured mesh whose curvilinear boundaries are represented by GeomObjects – this ensures that the domain boundaries become increasingly well resolved under mesh refinement.

That's all!

1.3 Code listing

Here's a listing of the complete driver code:

```
//LIC// ====================================================================
//LIC// This file forms part of oomph-lib, the object-oriented,
//LIC// multi-physics finite-element library, available
//LIC// at http://www.oomph-lib.org.
//LIC//
//LIC// Copyright (C) 2006-2021 Matthias Heil and Andrew Hazel
//LIC//
//LIC// This library is free software; you can redistribute it and/or
//LIC// modify it under the terms of the GNU Lesser General Public
//LIC// License as published by the Free Software Foundation; either
//LIC// version 2.1 of the License, or (at your option) any later version.
//LIC//
//LIC// This library is distributed in the hope that it will be useful,
//LIC// but WITHOUT ANY WARRANTY; without even the implied warranty of
//LIC// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
//LIC// Lesser General Public License for more details.
//LIC//
//LIC// You should have received a copy of the GNU Lesser General Public
//LIC// License along with this library; if not, write to the Free Software
//LIC// Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
//LIC// 02110-1301  USA.
//LIC//
//LIC// The authors may be contacted at oomph-lib@maths.man.ac.uk.
//LIC//
//LIC//====================================================================
//Driver for Fourier-decomposed  Helmholtz problem
#include <complex>
#include <cmath>
//Generic routines
#include "generic.h"
// The Helmholtz equations
#include "fourier_decomposed_helmholtz.h"

// The mesh
#include "meshes/triangle_mesh.h"
// Get the Bessel functions
#include "oomph_crbond_bessel.h"
using namespace oomph;
using namespace std;

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

//===== start_of_namespace_planar_wave================================
/// Namespace to test representation of planar wave in spherical
/// polars
//====================================================================
namespace PlanarWave
{

 /// Number of terms in series
 unsigned N_terms=100;

 /// Wave number
 double K=3.0*MathematicalConstants::Pi;

 /// Imaginary unit
 std::complex<double> I(0.0,1.0);
```

```
 /// Exact solution as a Vector of size 2, containing real and imag parts
 void get_exact_u(const Vector<double>& x, Vector<double>& u)
 {
  // Switch to spherical coordinates
  double R=sqrt(x[0]*x[0]+x[1]*x[1]);

  double theta;
  theta=atan2(x[0],x[1]);

  // Argument for Bessel/Hankel functions
  double kr = K*R;

  // Need half-order Bessel functions
  double bessel_offset=0.5;
  // Evaluate Bessel/Hankel functions
  Vector<double> jv(N_terms);
  Vector<double> yv(N_terms);
  Vector<double> djv(N_terms);
  Vector<double> dyv(N_terms);
  double order_max_in=double(N_terms-1)+bessel_offset;
  double order_max_out=0;

  // This function returns vectors containing
  // J_k(x), Y_k(x) and their derivatives
  // up to k=order_max, with k increasing in
  // integer increments starting with smallest
  // positive value. So, e.g. for order_max=3.5
  // jv[0] contains J_{1/2}(x),
  // jv[1] contains J_{3/2}(x),
  // jv[2] contains J_{5/2}(x),
  // jv[3] contains J_{7/2}(x).
  CRBond_Bessel::bessjyv(order_max_in,
                         kr,
                         order_max_out,
                         &jv[0],&yv[0],
                         &djv[0],&dyv[0]);


  // Assemble  exact solution (actually no need to add terms
  // below i=N_fourier as Legendre polynomial would be zero anyway)
  complex<double> u_ex(0.0,0.0);
  for(unsigned i=0;i<N_terms;i++)
   {
    //Associated_legendre_functions
    double p=Legendre_functions_helper::plgndr2(i,0,cos(theta));

    // Set exact solution
    u_ex+=(2.0*i+1.0)*pow(I,i)*
     sqrt(MathematicalConstants::Pi/(2.0*kr))*jv[i]*p;
   }

  // Get the real & imaginary part of the result
  u[0]=u_ex.real();
  u[1]=u_ex.imag();

 }//end of get_exact_u

 /// Plot
 void plot()
 {
  unsigned nr=20;
  unsigned nz=100;
  unsigned nt=40;
  ofstream some_file("planar_wave.dat");
  for (unsigned i_t=0;i_t<nt;i_t++)
   {
    double t=2.0*MathematicalConstants::Pi*double(i_t)/double(nt-1);
    some_file « "ZONE I="« nz « ", J="« nr « std::endl;

    Vector<double> x(2);
    Vector<double> u(2);
    for (unsigned i=0;i<nr;i++)
     {
      x[0]=0.001+double(i)/double(nr-1);
      for (unsigned j=0;j<nz;j++)
       {
        x[1]=double(j)/double(nz-1);
        get_exact_u(x,u);
        complex<double> uu=complex<double>(u[0],u[1])*exp(-I*t);
        some_file « x[0] « " " « x[1] « " "
                  « uu.real() « " " « uu.imag() « "\n";
       }
     }
   }
 }

}
```

```
/// //////////////////////////////////////////////////////////////////
/// //////////////////////////////////////////////////////////////////
/// //////////////////////////////////////////////////////////////////

//===== start_of_namespace=============================================
/// Namespace for the Fourier decomposed Helmholtz problem parameters
//======================================================================
namespace ProblemParameters
{
 /// Square of the wavenumber
 double K_squared=10.0;

 /// Fourier wave number
 int N_fourier=3;

 /// Number of terms in computation of DtN boundary condition
 unsigned Nterms_for_DtN=6;

 /// Number of terms in the exact solution
 unsigned N_terms=6;

 /// Coefficients in the exact solution
 Vector<double> Coeff(N_terms,1.0);

 /// Imaginary unit
 std::complex<double> I(0.0,1.0);

 /// Exact solution as a Vector of size 2, containing real and imag parts
 void get_exact_u(const Vector<double>& x, Vector<double>& u)
 {
  // Switch to spherical coordinates
  double R=sqrt(x[0]*x[0]+x[1]*x[1]);

  double theta;
  theta=atan2(x[0],x[1]);

  // Argument for Bessel/Hankel functions
  double kr = sqrt(K_squared)*R;

  // Need half-order Bessel functions
  double bessel_offset=0.5;
  // Evaluate Bessel/Hankel functions
  Vector<double> jv(N_terms);
  Vector<double> yv(N_terms);
  Vector<double> djv(N_terms);
  Vector<double> dyv(N_terms);
  double order_max_in=double(N_terms-1)+bessel_offset;
  double order_max_out=0;

  // This function returns vectors containing
  // J_k(x), Y_k(x) and their derivatives
  // up to k=order_max, with k increasing in
  // integer increments starting with smallest
  // positive value. So, e.g. for order_max=3.5
  // jv[0] contains J_{1/2}(x),
  // jv[1] contains J_{3/2}(x),
  // jv[2] contains J_{5/2}(x),
  // jv[3] contains J_{7/2}(x).
  CRBond_Bessel::bessjyv(order_max_in,
                         kr,
                         order_max_out,
                         &jv[0],&yv[0],
                         &djv[0],&dyv[0]);


  // Assemble  exact solution (actually no need to add terms
  // below i=N_fourier as Legendre polynomial would be zero anyway)
  complex<double> u_ex(0.0,0.0);
  for(unsigned i=N_fourier;i<N_terms;i++)
   {
    //Associated_legendre_functions
    double p=Legendre_functions_helper::plgndr2(i,N_fourier,
                                                cos(theta));
    // Set exact solution
    u_ex+=Coeff[i]*sqrt(MathematicalConstants::Pi/(2.0*kr))*(jv[i]+I*yv[i])*p;
   }

  // Get the real & imaginary part of the result
  u[0]=u_ex.real();
  u[1]=u_ex.imag();

 }//end of get_exact_u

 /// Get -du/dr (spherical r) for exact solution. Equal to prescribed
 /// flux on inner boundary.
 void exact_minus_dudr(const Vector<double>& x, std::complex<double>& flux)
```

```
 {
  // Initialise flux
  flux=std::complex<double>(0.0,0.0);

  // Switch to spherical coordinates
  double R=sqrt(x[0]*x[0]+x[1]*x[1]);

  double theta;
  theta=atan2(x[0],x[1]);

  // Argument for Bessel/Hankel functions
  double kr=sqrt(K_squared)*R;
  // Helmholtz wavenumber
  double k=sqrt(K_squared);
  // Need half-order Bessel functions
  double bessel_offset=0.5;
  // Evaluate Bessel/Hankel functions
  Vector<double> jv(N_terms);
  Vector<double> yv(N_terms);
  Vector<double> djv(N_terms);
  Vector<double> dyv(N_terms);
  double order_max_in=double(N_terms-1)+bessel_offset;
  double order_max_out=0;

  // This function returns vectors containing
  // J_k(x), Y_k(x) and their derivatives
  // up to k=order_max, with k increasing in
  // integer increments starting with smallest
  // positive value. So, e.g. for order_max=3.5
  // jv[0] contains J_{1/2}(x),
  // jv[1] contains J_{3/2}(x),
  // jv[2] contains J_{5/2}(x),
  // jv[3] contains J_{7/2}(x).
  CRBond_Bessel::bessjyv(order_max_in,
                         kr,
                         order_max_out,
                         &jv[0],&yv[0],
                         &djv[0],&dyv[0]);


  // Assemble  exact solution (actually no need to add terms
  // below i=N_fourier as Legendre polynomial would be zero anyway)
  complex<double> u_ex(0.0,0.0);
  for(unsigned i=N_fourier;i<N_terms;i++)
   {
    //Associated_legendre_functions
    double p=Legendre_functions_helper::plgndr2(i,N_fourier,
                                                cos(theta));
    // Set flux of exact solution
    flux-=Coeff[i]*sqrt(MathematicalConstants::Pi/(2.0*kr))*p*
     ( k*(djv[i]+I*dyv[i]) - (0.5*(jv[i]+I*yv[i])/R) );
   }

 }// end of exact_normal_derivative


} // end of namespace

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

//========= start_of_problem_class====================================
/// Problem class
//====================================================================
template<class ELEMENT>
class FourierDecomposedHelmholtzProblem : public Problem
{

public:

 /// Constructor
 FourierDecomposedHelmholtzProblem();

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

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

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

 /// Doc the solution. DocInfo object stores flags/labels for where the
 /// output gets written to
 void doc_solution(DocInfo& doc_info);
```

```
 /// Recompute gamma integral before checking Newton residuals
 void actions_before_newton_convergence_check()
  {
   if (!CommandLineArgs::command_line_flag_has_been_set("--square_domain"))
    {
     Helmholtz_outer_boundary_mesh_pt->setup_gamma();
    }
  }

 /// Actions before adapt: Wipe the mesh of prescribed flux elements
 void actions_before_adapt();

 /// Actions after adapt: Rebuild the mesh of prescribed flux elements
 void actions_after_adapt();

 /// Check gamma computation
 void check_gamma(DocInfo& doc_info);

private:

 /// Create BC elements on outer boundary
 void create_outer_bc_elements();

 /// Create flux elements on inner boundary
 void create_flux_elements_on_inner_boundary();


 /// Delete boundary face elements and wipe the surface mesh
 void delete_face_elements( Mesh* const & boundary_mesh_pt)
  {
   // Loop over the surface elements
   unsigned n_element = boundary_mesh_pt->nelement();
   for(unsigned e=0;e<n_element;e++)
    {
     // Kill surface element
     delete  boundary_mesh_pt->element_pt(e);
    }

   // Wipe the mesh
   boundary_mesh_pt->flush_element_and_node_storage();

  }
#ifdef ADAPTIVE

 /// Pointer to the "bulk" mesh
 RefineableTriangleMesh<ELEMENT>* Bulk_mesh_pt;
#else

 /// Pointer to the "bulk" mesh
 TriangleMesh<ELEMENT>* Bulk_mesh_pt;
#endif

 /// Pointer to mesh containing the DtN boundary
 /// condition elements
 FourierDecomposedHelmholtzDtNMesh<ELEMENT>* Helmholtz_outer_boundary_mesh_pt;

 /// on the inner boundary
 Mesh* Helmholtz_inner_boundary_mesh_pt;

 /// Trace file
 ofstream Trace_file;
}; // end of problem class
//======================start_of_actions_before_adapt=======================
/// Actions before adapt: Wipe the mesh of face elements
//========================================================================
template<class ELEMENT>
void FourierDecomposedHelmholtzProblem<ELEMENT>::actions_before_adapt()
{
 // Kill the flux elements and wipe the boundary meshs
 if (!CommandLineArgs::command_line_flag_has_been_set("--square_domain"))
  {
   delete_face_elements(Helmholtz_outer_boundary_mesh_pt);
  }
 delete_face_elements(Helmholtz_inner_boundary_mesh_pt);
 // Rebuild the Problem's global mesh from its various sub-meshes
 rebuild_global_mesh();
}// end of actions_before_adapt
//======================start_of_actions_after_adapt=======================
///  Actions after adapt: Rebuild the face element meshes
//========================================================================
template<class ELEMENT>
void FourierDecomposedHelmholtzProblem<ELEMENT>::actions_after_adapt()
{
 // Complete the build of all elements so they are fully functional

 // Loop over the Helmholtz bulk elements to set up element-specific
 // things that cannot be handled by constructor: Pass pointer to
```

```
 // wave number squared
 unsigned n_element = Bulk_mesh_pt->nelement();
 for(unsigned e=0;e<n_element;e++)
  {
   // Upcast from GeneralisedElement to Helmholtz bulk element
   ELEMENT *el_pt = dynamic_cast<ELEMENT*>(Bulk_mesh_pt->element_pt(e));

   //Set the k_squared  pointer
   el_pt->k_squared_pt() = &ProblemParameters::K_squared;
   // Set pointer to Fourier wave number
   el_pt->fourier_wavenumber_pt()=&ProblemParameters::N_fourier;
  }
 // Create prescribed-flux elements and BC elements
 // from all elements that are adjacent to the boundaries and add them to
 // Helmholtz_boundary_meshes
 create_flux_elements_on_inner_boundary();
 if (!CommandLineArgs::command_line_flag_has_been_set("--square_domain"))
  {
   create_outer_bc_elements();
  }
 // Rebuild the Problem's global mesh from its various sub-meshes
 rebuild_global_mesh();

}// end of actions_after_adapt
//=======start_of_constructor==============================================
/// Constructor for Fourier-decomposed Helmholtz problem
//==========================================================================
template<class ELEMENT>
FourierDecomposedHelmholtzProblem<ELEMENT>::
FourierDecomposedHelmholtzProblem()
{
 // Open trace file
 Trace_file.open("RESLT/trace.dat");

 // Create circles representing inner and outer boundary
 double x_c=0.0;
 double y_c=0.0;
 double r_min=1.0;
 double r_max=3.0;
 Circle* inner_circle_pt=new Circle(x_c,y_c,r_min);
 Circle* outer_circle_pt=new Circle(x_c,y_c,r_max);

 // Edges/boundary segments making up outer boundary
 //-------------------------------------------------
 Vector<TriangleMeshCurveSection*> outer_boundary_line_pt(4);

 // Number of segments used for representing the curvilinear boundaries
 unsigned n_segments = 20;

 // All poly boundaries are defined by two vertices
 Vector<Vector<double> > boundary_vertices(2);

 // Bottom straight boundary on symmetry line
 //------------------------------------------
 boundary_vertices[0].resize(2);
 boundary_vertices[0][0]=0.0;
 boundary_vertices[0][1]=-r_min;
 boundary_vertices[1].resize(2);
 boundary_vertices[1][0]=0.0;
 boundary_vertices[1][1]=-r_max;
 unsigned boundary_id=0;
 outer_boundary_line_pt[0]=
  new TriangleMeshPolyLine(boundary_vertices,boundary_id);
 if (CommandLineArgs::command_line_flag_has_been_set("--square_domain"))
  {
   // Square outer boundary:
   //-----------------------
   Vector<Vector<double> > boundary_vertices(4);
   boundary_vertices[0].resize(2);
   boundary_vertices[0][0]=0.0;
   boundary_vertices[0][1]=-r_max;
   boundary_vertices[1].resize(2);
   boundary_vertices[1][0]=r_max;
   boundary_vertices[1][1]=-r_max;
   boundary_vertices[2].resize(2);
   boundary_vertices[2][0]=r_max;
   boundary_vertices[2][1]=r_max;
   boundary_vertices[3].resize(2);
   boundary_vertices[3][0]=0.0;
   boundary_vertices[3][1]=r_max;
   boundary_id=1;
   outer_boundary_line_pt[1]=
    new TriangleMeshPolyLine(boundary_vertices,boundary_id);
  }
 else
  {
   // Outer circular boundary:
```

```
   //-------------------------
   // The intrinsic coordinates for the beginning and end of the curve
   double s_start = -0.5*MathematicalConstants::Pi;
   double s_end   =  0.5*MathematicalConstants::Pi;

   boundary_id = 1;
   outer_boundary_line_pt[1]=
    new TriangleMeshCurviLine(outer_circle_pt,
                              s_start,
                              s_end,
                              n_segments,
                              boundary_id);
  }
 // Top straight boundary on symmetry line
 //----------------------------------------
 boundary_vertices[0][0]=0.0;
 boundary_vertices[0][1]=r_max;
 boundary_vertices[1][0]=0.0;
 boundary_vertices[1][1]=r_min;
 boundary_id=2;
 outer_boundary_line_pt[2]=
  new TriangleMeshPolyLine(boundary_vertices,boundary_id);

 // Inner circular boundary:
 //-------------------------

 // The intrinsic coordinates for the beginning and end of the curve
 double s_start =  0.5*MathematicalConstants::Pi;
 double s_end   =  -0.5*MathematicalConstants::Pi;

 boundary_id = 3;
 outer_boundary_line_pt[3]=
  new TriangleMeshCurviLine(inner_circle_pt,
                            s_start,
                            s_end,
                            n_segments,
                            boundary_id);


 // Create closed curve that defines outer boundary
 //------------------------------------------------
 TriangleMeshClosedCurve *outer_boundary_pt =
  new TriangleMeshClosedCurve(outer_boundary_line_pt);

 // Use the TriangleMeshParameters object for helping on the manage of the
 // TriangleMesh parameters. The only parameter that needs to take is the
 // outer boundary.
 TriangleMeshParameters triangle_mesh_parameters(outer_boundary_pt);

 // Specify maximum element area
 double element_area = 0.1;
 triangle_mesh_parameters.element_area() = element_area;

#ifdef ADAPTIVE

 // Build "bulk" mesh
 Bulk_mesh_pt=new RefineableTriangleMesh<ELEMENT>(triangle_mesh_parameters);
 // Create/set error estimator
 Bulk_mesh_pt->spatial_error_estimator_pt()=new Z2ErrorEstimator;

 // Choose error tolerances to force some uniform refinement
 Bulk_mesh_pt->min_permitted_error()=0.00004;
 Bulk_mesh_pt->max_permitted_error()=0.0001;
#else
 // Pass the TriangleMeshParameters object to the TriangleMesh one
 Bulk_mesh_pt= new TriangleMesh<ELEMENT>(triangle_mesh_parameters);
#endif
 // Check what we've built so far...
 Bulk_mesh_pt->output("mesh.dat");
 Bulk_mesh_pt->output_boundaries("boundaries.dat");

 if (!CommandLineArgs::command_line_flag_has_been_set("--square_domain"))
  {
   // Create mesh for DtN elements on outer boundary
   Helmholtz_outer_boundary_mesh_pt=
    new FourierDecomposedHelmholtzDtNMesh<ELEMENT>(
     r_max,ProblemParameters::Nterms_for_DtN);

   // Populate it with elements
   create_outer_bc_elements();
  }
 // Create flux elements on inner boundary
 Helmholtz_inner_boundary_mesh_pt=new Mesh;
 create_flux_elements_on_inner_boundary();

 // Add the several sub meshes to the problem
 add_sub_mesh(Bulk_mesh_pt);
```

```
 add_sub_mesh(Helmholtz_inner_boundary_mesh_pt);
 if (!CommandLineArgs::command_line_flag_has_been_set("--square_domain"))
  {
   add_sub_mesh(Helmholtz_outer_boundary_mesh_pt);
  }
 // Build the Problem's global mesh from its various sub-meshes
 build_global_mesh();
 // Complete the build of all elements so they are fully functional
 unsigned n_element = Bulk_mesh_pt->nelement();
 for(unsigned i=0;i<n_element;i++)
  {
   // Upcast from GeneralsedElement to the present element
   ELEMENT *el_pt = dynamic_cast<ELEMENT*>(Bulk_mesh_pt->element_pt(i));

   //Set the k_squared pointer
   el_pt->k_squared_pt()=&ProblemParameters::K_squared;

   // Set pointer to Fourier wave number
   el_pt->fourier_wavenumber_pt()=&ProblemParameters::N_fourier;
  }

 // Setup equation numbering scheme
 cout <<"Number of equations: " << assign_eqn_numbers() << std::endl;
} // end of constructor
//=================================start_of_check_gamma==================
/// Check gamma computation: \f$ \gamma = -du/dn \f$
//========================================================================
template<class ELEMENT>
void FourierDecomposedHelmholtzProblem<ELEMENT>::check_gamma(DocInfo& doc_info)
{

 // Compute gamma stuff
 Helmholtz_outer_boundary_mesh_pt->setup_gamma();

 ofstream some_file;
 char filename[100];
 sprintf(filename,"%s/gamma_test%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);

 //first loop over elements e
 unsigned nel=Helmholtz_outer_boundary_mesh_pt->nelement();
 for (unsigned e=0;e<nel;e++)
  {
   // Get a pointer to element
   FourierDecomposedHelmholtzDtNBoundaryElement<ELEMENT>* el_pt=
    dynamic_cast<FourierDecomposedHelmholtzDtNBoundaryElement<ELEMENT>*>
    (Helmholtz_outer_boundary_mesh_pt->element_pt(e));

   //Set the value of n_intpt
   const unsigned n_intpt =el_pt->integral_pt()->nweight();

   // Get gamma at all gauss points in element
   Vector<std::complex<double> > gamma(
    Helmholtz_outer_boundary_mesh_pt->gamma_at_gauss_point(el_pt));

   //Loop over the integration points
   for(unsigned ipt=0;ipt<n_intpt;ipt++)
    {
     //Allocate and initialise coordiante
     Vector<double> x(el_pt->dim()+1,0.0);

     //Set the Vector to hold local coordinates
     unsigned n=el_pt->dim();
     Vector<double> s(n,0.0);
     for(unsigned i=0;i<n;i++)
      {
       s[i]=el_pt->integral_pt()->knot(ipt,i);
      }

     //Get the coordinates of the integration point
     el_pt->interpolated_x(s,x);

     complex<double> flux;
     ProblemParameters::exact_minus_dudr(x,flux);
     some_file << atan2(x[0],x[1]) << " "
               << gamma[ipt].real() << " "
               << gamma[ipt].imag() << " "
               << flux.real() << " "
               << flux.imag() << " "
               << std::endl;

    }// end of loop over integration points

  }// end of loop over elements

 some_file.close();
```

```
}//end of output_gamma
//===============start_of_doc=============================================
/// Doc the solution: doc_info contains labels/output directory etc.
//========================================================================
template<class ELEMENT>
void FourierDecomposedHelmholtzProblem<ELEMENT>::doc_solution(DocInfo& doc_info)
{
 ofstream some_file;
 char filename[100];
 // Number of plot points: npts x npts
 unsigned npts=5;

 // Output solution
 //-----------------
 sprintf(filename,"%s/soln%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 Bulk_mesh_pt->output(some_file,npts);
 some_file.close();
 // Output exact solution
 //----------------------
 sprintf(filename,"%s/exact_soln%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 Bulk_mesh_pt->output_fct(some_file,npts,ProblemParameters::get_exact_u);
 some_file.close();


 // Doc error and return of the square of the L2 error
 //---------------------------------------------------
 double error,norm;
 sprintf(filename,"%s/error%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 Bulk_mesh_pt->compute_error(some_file,ProblemParameters::get_exact_u,
                             error,norm);
 some_file.close();

 // Doc L2 error and norm of solution
 cout << "\nNorm of error   : " << sqrt(error) << std::endl;
 cout << "Norm of solution: " << sqrt(norm) << std::endl << std::endl;

 // Write norm of solution to trace file
 Bulk_mesh_pt->compute_norm(norm);
 Trace_file  << norm << std::endl;
 if (!CommandLineArgs::command_line_flag_has_been_set("--square_domain"))
  {
   // Check gamma computation
   check_gamma(doc_info);
  }
} // end of doc
//============start_of_create_outer_bc_elements============================
/// Create BC elements on outer boundary
//========================================================================
template<class ELEMENT>
void FourierDecomposedHelmholtzProblem<ELEMENT>::create_outer_bc_elements()
{
 // Outer boundary is boundary 1:
 unsigned b=1;
 // Loop over the bulk elements adjacent to boundary b?
 unsigned n_element = Bulk_mesh_pt->nboundary_element(b);
 for(unsigned e=0;e<n_element;e++)
  {
   // Get pointer to the bulk 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);

   // Build the corresponding DtN element
   FourierDecomposedHelmholtzDtNBoundaryElement<ELEMENT>* flux_element_pt = new
    FourierDecomposedHelmholtzDtNBoundaryElement<ELEMENT>(bulk_elem_pt,
                                                          face_index);

   //Add the flux boundary element to the  helmholtz_outer_boundary_mesh
   Helmholtz_outer_boundary_mesh_pt->add_element_pt(flux_element_pt);
   // Set pointer to the mesh that contains all the boundary condition
   // elements on this boundary
   flux_element_pt->
    set_outer_boundary_mesh_pt(Helmholtz_outer_boundary_mesh_pt);
  }
} // end of create_outer_bc_elements
//============start_of_create_flux_elements=================
/// Create flux elements on inner boundary
//===========================================================
```

```
template<class ELEMENT>
void  FourierDecomposedHelmholtzProblem<ELEMENT>::
create_flux_elements_on_inner_boundary()
{
 // Apply flux bc on inner boundary (boundary 3)
 unsigned b=3;
// Loop over the bulk elements adjacent to boundary b
 unsigned n_element = Bulk_mesh_pt->nboundary_element(b);
 for(unsigned e=0;e<n_element;e++)
  {
   // Get pointer to the bulk 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);

   // Build the corresponding prescribed incoming-flux element
   FourierDecomposedHelmholtzFluxElement<ELEMENT>* flux_element_pt = new
    FourierDecomposedHelmholtzFluxElement<ELEMENT>(bulk_elem_pt,face_index);

   //Add the prescribed incoming-flux element to the surface mesh
   Helmholtz_inner_boundary_mesh_pt->add_element_pt(flux_element_pt);

   // Set the pointer to the prescribed flux function
   flux_element_pt->flux_fct_pt() = &ProblemParameters::exact_minus_dudr;
  } //end of loop over bulk elements adjacent to boundary b

} // end of create flux elements on inner boundary
//===== start_of_main====================================================
/// Driver code for Fourier decomposed Helmholtz problem
//========================================================================
int main(int argc, char **argv)
{
 // Store command line arguments
 CommandLineArgs::setup(argc,argv);
 // Define possible command line arguments and parse the ones that
 // were actually specified

 // Square domain without DtN
 CommandLineArgs::specify_command_line_flag("--square_domain");
 // Parse command line
 CommandLineArgs::parse_and_assign();

 // Doc what has actually been specified on the command line
 CommandLineArgs::doc_specified_flags();
 // Check if the claimed representation of a planar wave in
 // the tutorial is correct -- of course it is!
 //PlanarWave::plot();
 // Test Bessel/Hankel functions
 //-----------------------------
 {
  // Number of Bessel functions to be computed
  unsigned n=3;

  // Offset of Bessel function order (less than 1!)
  double bessel_offset=0.5;

  ofstream bessely_file("besselY.dat");
  ofstream bessely_deriv_file("dbesselY.dat");

  ofstream besselj_file("besselJ.dat");
  ofstream besselj_deriv_file("dbesselJ.dat");

  // Evaluate Bessel/Hankel functions
  Vector<double> jv(n+1);
  Vector<double> yv(n+1);
  Vector<double> djv(n+1);
  Vector<double> dyv(n+1);
  double x_min=0.5;
  double x_max=5.0;
  unsigned nplot=100;
  for (unsigned i=0;i<nplot;i++)
   {
    double x=x_min+(x_max-x_min)*double(i)/double(nplot-1);
    double order_max_in=double(n)+bessel_offset;
    double order_max_out=0;

    // This function returns vectors containing
    // J_k(x), Y_k(x) and their derivatives
    // up to k=order_max, with k increasing in
    // integer increments starting with smallest
    // positive value. So, e.g. for order_max=3.5
    // jv[0] contains J_{1/2}(x),
    // jv[1] contains J_{3/2}(x),
    // jv[2] contains J_{5/2}(x),
    // jv[3] contains J_{7/2}(x).
```

```
    CRBond_Bessel::bessjyv(order_max_in,x,
                           order_max_out,
                           &jv[0],&yv[0],
                           &djv[0],&dyv[0]);
    bessely_file « x « " ";
    for (unsigned j=0;j<=n;j++)
     {
      bessely_file « yv[j] « " ";
     }
    bessely_file « std::endl;

    besselj_file « x « " ";
    for (unsigned j=0;j<=n;j++)
     {
      besselj_file « jv[j] « " ";
     }
    besselj_file « std::endl;

    bessely_deriv_file « x « " ";
    for (unsigned j=0;j<=n;j++)
     {
      bessely_deriv_file « dyv[j] « " ";
     }
    bessely_deriv_file « std::endl;

    besselj_deriv_file « x « " ";
    for (unsigned j=0;j<=n;j++)
     {
      besselj_deriv_file « djv[j] « " ";
     }
    besselj_deriv_file « std::endl;

   }
  bessely_file.close();
  besselj_file.close();
  bessely_deriv_file.close();
  besselj_deriv_file.close();
 }


 // Test Legrendre Polynomials
 //---------------------------
 {
  // Number of lower indices
  unsigned n=3;

  ofstream some_file("legendre3.dat");
  unsigned nplot=100;
  for (unsigned i=0;i<nplot;i++)
   {
    double x=double(i)/double(nplot-1);
    some_file « x « " ";
    for (unsigned j=0;j<=n;j++)
     {
      some_file «  Legendre_functions_helper::plgndr2(n,j,x) « " ";
     }
    some_file « std::endl;
   }
  some_file.close();
 }

#ifdef ADAPTIVE
 // Create the problem with 2D six-node elements from the
 // TFourierDecomposedHelmholtzElement family.
 FourierDecomposedHelmholtzProblem<ProjectableFourierDecomposedHelmholtzElement<
  TFourierDecomposedHelmholtzElement<3> > > problem;

#else

 // Create the problem with 2D six-node elements from the
 // TFourierDecomposedHelmholtzElement family.
 FourierDecomposedHelmholtzProblem<TFourierDecomposedHelmholtzElement<3> >
  problem;

#endif
 // Create label for output
 DocInfo doc_info;

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

 // Solve for a few Fourier wavenumbers
 for (ProblemParameters::N_fourier=0;ProblemParameters::N_fourier<4;
      ProblemParameters::N_fourier++)
  {
   // Step number
   doc_info.number()=ProblemParameters::N_fourier;
```

```
#ifdef ADAPTIVE
 // Max. number of adaptations
 unsigned max_adapt=1;

   // Solve the problem with Newton's method, allowing
   // up to max_adapt mesh adaptations after every solve.
   problem.newton_solve(max_adapt);
#else
   // Solve the problem
   problem.newton_solve();
#endif

   //Output the solution
   problem.doc_solution(doc_info);
  }

} //end of main
```

1.4 Source files for this tutorial

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

  demo_drivers/fourier_decomposed_helmholtz/sphere_scattering/

- The driver code is:

  demo_drivers/fourier_decomposed_helmholtz/sphere_scattering/unstructured_sphere_scattering.cc

1.5 PDF file

A pdf version of this document is available.