Chapter 1

Refineable Boussinesq Convection using a multi-domain approach: Combining (Refineable) Navier–Stokes and Advection-Diffusion domains

In this tutorial we present an alternative approach to the solution of multi-physics problems, using what we call a "multi-domain" approach. We illustrate the methodology by re-considering the `previously-described,` two-dimensional Boussinesq convection problem in which an incompressible, Newtonian fluid is heated from below.

The tutorial is quite long but comprises four relatively self-contained parts. Feel free to skip the theory if you want to jump straight in...

- **Part I** provides an overview of `oomph-lib's` overall framework for setting up multi-domain interactions.

- **Part II** illustrates the general procedure for the specific example of a Boussinesq convection problem. We demonstrate how to implement a two-way, multi-domain interaction between the Navier-Stokes equations in which the body force is affected by thermal buoyancy effects, and the advection-diffusion equations for the temperature, in which the "wind" is given by fluid velocity.

- **Part III** discusses the driver code to solve the Boussinesq problem, using the elements developed in Part II.

- **Part IV** discusses how to optimise multi-domain interactions.

1.1 Part I: The overall framework for handling multi-domain interactions

When two or more physical processes interact within the same spatial domain there are three different approaches to setting up the interaction within `oomph-lib`. One approach is to write a completely new element that discretises all the PDEs involved in the problem. The second approach, described in the tutorials discussing the `the non-refineable` and `refineable` solution of the Boussinesq convection problem, is to create a combined multi-physics element using inheritance from two, or more, existing elements. In both these cases, the physical processes and their associated fields interact within a single element and we refer to this methodology as a **multi-field** approach.

In this tutorial we describe a third approach to multi-physics problems, in which we solve the problem using two different types of elements on two different meshes (domains), each occupying the same physical space. Rather than interacting locally within elements, the different physical processes interact directly between the two domains, so that, in the Boussinesq convection problem considered here, each advection-diffusion element obtains the "wind" from the (separate) Navier-Stokes element that occupies the same position, while the Navier-Stokes elements obtain the temperature required for the computation of the buoyancy force from the corresponding advection-diffusion element.

One benefit of this **multi-domain** approach, compared to a **multi-field** approach, is that different error estimators can be used on each domain and there is no longer a need to construct a combined multi-physics error estimator; see the `tutorial for the adaptive solution of the Boussinesq convection problem` for more details on this issue. Moreover, the meshes do not have to have the same refinement pattern, which can be advantageous if the different physical processes act over different spatial scales. The figure below shows that in the Boussinesq convection problem the Navier–Stokes mesh (panel (a)) requires much more refinement than the advection-diffusion mesh (panel (b)).

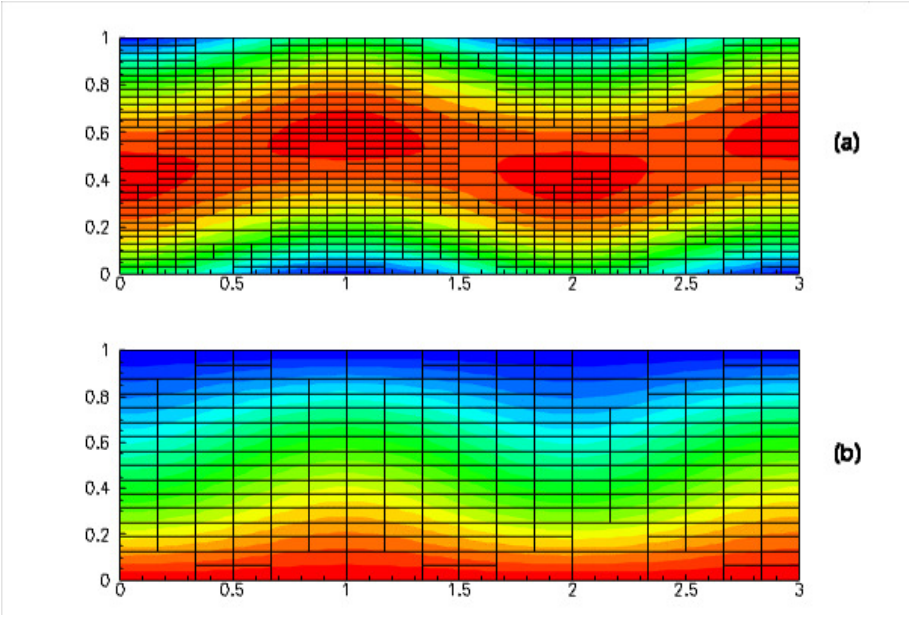

Figure 1.1 Steady Convection Rolls: (a) contours of x-velocity and corresponding element boundaries in Navier--Stokes mesh, (b) contours of temperature and corresponding element boundaries in advection-diffusion mesh.

1.1.1 The ElementWithExternalElement class

Interaction between different elements in different domains is a fundamental feature of many multi-physics problems and the generic functionality to deal with such interactions is provided by the `ElementWithExternalElement` class. Any element that requires information from an "external element" should therefore inherit from the base class `ElementWithExternalElement`.

"External elements" typically provide source-like terms for the `ElementWithExternalElement`. For instance, in the Boussinesq convection problem studied here, the advection-diffusion element is the "external element" for the Navier-Stokes element because it affects the body force in the Navier-Stokes equations; in a fluid-structure

interaction problem, the Navier-Stokes elements that are adjacent to the fluid-loaded elastic wall act as the "external elements" for the FSIWallElement because they determine the traction that the fluid exerts onto the wall; etc.
Within a finite-element context, the effect of the "external element" onto the ElementWithExternalElement typically has to be evaluated at the integration points of the ElementWithExternalElement. Therefore, the ElementWithExternalElement base class provides storage for a pointer to an "external element" (and the local coordinate within it) for each integration point. Access to this information is provided via the member functions

```
FiniteElement*& ElementWithExternalElement::external_element_pt(
                                const unsigned& interaction,
                                const unsigned& ipt);
```

and

```
Vector<double>& ElementWithExternalElement::external_element_local_coord(
                      const unsigned& interaction,
                      const unsigned& ipt);
```

For a given integration point, ipt, within the ElementWithExternalElement, these functions identify the corresponding point in the "external element" by returning a pointer to the "external element" and the local coordinate within it, respectively. The unsigned parameter interaction denotes the interaction index, required to allow for cases where an element interacts with more than one "external element". This occurs, for instance, in double-diffusive convection problems (discussed in a separate demo driver code), in which the body force in the Navier-Stokes equations is affected by two physical properties, *e.g.* temperature and salinity, both of which are transported by separate advection-diffusion processes; or in FSI problems where a beam or shell element is affected by a fluid load on both its "front" and "back" (see, e.g., the FSI channel with leaflet problem).
Data that affects the fields in the "external elements" must be specified so that the off-diagonal blocks in the ElementWithExternalElement's Jacobian matrix (representing the derivatives of the ElementWith↩ ExternalElement's residual vector with respect to the Data in the "external elements") can be calculated. We divide such data into two categories:

- *External field data:* all Data that affects the fields through which the "external element" interacts with the ElementWithExternalElement.
- *External geometric data:* all geometric Data that affects the shape and position of the "external element" and, therefore, spatial derivatives of its fields.

The rationale for this division is that in many cases the field data's contribution to the interaction blocks in the Jacobian matrix can be found analytically, whereas the geometric data is more easily handled by finite differencing. Both types of external data are determined (automatically) by two member functions of the ElementWith↩ ExternalElement. The first one,

```
virtual void  ElementWithExternalElement::identify_all_field_data_for_external_interaction(
    Vector<std::set<FiniteElement*> > const &external_elements_pt,
    std::set<std::pair<Data*,unsigned> > &paired_interaction_data);
```

determines the external field data that affect the (possibly multiple) interactions in the ElementWithExternal↩ Element. Given the vector external_elements_pt (within which external_elements_pt[i] contains the set of pointers to the "external elements" involved in the ElementWithExternalElement's i -th interaction), the function returns a set of pairs, each of which comprises a pointer to a Data object and an unsigned that identifies a value within it. The default implementation of this function includes **all** field data of **all** "external elements", using calls to

```
void FiniteElement::identify_field_data_for_interactions(...)
```

for each "external element". Similarly, the function

```
virtual void  ElementWithExternalElement::identify_all_geometric_data_for_external_interaction(
    Vector<std::set<FiniteElement*> >  const &external_elements_pt,
    std::set<Data*> &external_geometric_data_pt);
```

returns the set of **all** Data that affect the shape or position of any of the "external elements" specified by the vector external_elements_pt. Again, this function has a default implementation that determines **all** geometric Data associated with "external elements" by calling

```
void FiniteElement::identify_geometric_data(..)
```

for each "external element".
We note that the default implementation of these functions can (and, where possible, should) be overloaded to exclude data values that do not actually contribute to the interaction. For instance, in the Boussinesq convection problem considered here, the advection-diffusion elements are only affected by the velocity degrees of freedom in the Navier-Stokes elements but not by the pressures. Our re-implementation of ElementWithExternal↩ Element::identify_all_field_data_for_external_interaction(...) in the multi-domain advection-diffusion element for the Boussinesq problem therefore excludes the pressure degrees of freedom, see **Part IV**.

1.1.2 Computation of the Jacobian matrix

Most single-physics elements already provide functions to compute their own Jacobians analytically. These functions compute the derivatives of the element's residual vector with respect to the element's "own" degrees of freedom (e.g. the fluid velocity and pressure in a Navier-Stokes element). The interaction with the "external element" introduces additional dependencies because the element's residual vector now also depends on the unknowns associated with the "external elements". The derivatives of the element's residual vector with respect to these unknowns must therefore be included into the computation of the element's Jacobian matrix.
To maximise code re-use, we follow the approach discussed in the `comments` section of the single-domain version of the problem and re-use the underlying element's `fill_in_contribution_to_jacobian(...)` function, while employing finite-differencing to evaluate the derivatives of the element's residual vector with respect to the unknowns associated with the "external elements". This is done most easily by using the function
`ElementWithExternalElement::fill_in_jacobian_from_external_interaction_by_fd(..)`
which computes the derivatives of the element's residual vector with respect to external field and external geometric data by calls to the two functions
`ElementWithExternalElement::fill_in_jacobian_from_external_interaction_field_by_fd(..)`
and
`ElementWithExternalElement::fill_in_jacobian_from_external_interaction_geometric_by_fd(..)`
If, in a particular problem, it is known that the interaction is not affected by the position or shape of the "external elements" (*i.e.* if the interaction terms do not involve spatial derivatives of the "external element's" field variables) the unnecessary computation of the derivatives with respect to the "external element's" geometric data may be suppressed using a call to
`ElementWithExternalElement::ignore_external_geometric_data();`
Depending on the complexity of the interaction terms, it may be worthwhile to provide a function that computes the off-diagonal blocks analytically. This is discussed in more detail in **`Part IV`**.

1.1.3 Setting up the interaction

The namespace `Multi_domain_functions` provides numerous helper functions that facilitate the setup of multi-domain interactions. Specifically, the function `Multi_domain_functions::setup_multi_↩ domain_interactions(...)` can be used to identify the "external elements" in a two-way interaction between two meshes. The function has the following interface:

```
template<class ELEMENT_0,class ELEMENT_1>
void setup_multi_domain_interactions(Problem* problem_pt,
                                     Mesh* const &first_mesh_pt,
                                     Mesh* const &second_mesh_pt,
                                     const unsigned& first_interaction=0,
                                     const unsigned& second_interaction=0);
```

Here `first_mesh_pt` and `second_mesh_pt` point to the two interacting meshes whose elements (of type `ELEMENT_0` and `ELEMENT_1`, respectively), must be derived from the `ElementWithExternalElement` class. The optional interaction parameters may be used to specify which interaction is set up in each mesh. If the parameter is not specified it defaults to zero, appropriate if there is only a single interaction.
The function must be called prior to the assignment of the equation numbers, and then again whenever either of the two meshes has changed, *e.g.* after a mesh adaptation.

1.2 Part II: Implementing multi-domain interaction elements for the Boussinesq convection problem

We illustrate the general procedures discussed above by demonstrating how to upgrade existing single-physics Navier-Stokes and advection-diffusion elements to `ElementWithExternalElements` that can be used for the multi-domain-based solution of the Boussinesq convection problem.

1.2.1 Upgrading the Navier-Stokes element to an ElementWithExternalElement

We use multiple inheritance to upgrade an existing refineable Navier-Stokes element to a `RefineableNavier↩ StokesBoussinesqElement` in which the temperature that affects the body force is given by an "external" advection-diffusion element. To facilitate code reuse we employ templating to specify the types of the Navier-Stokes and advection diffusion elements.

```
//======================nst_bous_class================================
/// Build a refineable Navier Stokes element that inherits from
/// ElementWithExternalElement so that it can "communicate" with
/// an advection diffusion element that provides the temperature
/// in the body force term.
//====================================================================
```

```
template<class NST_ELEMENT, class AD_ELEMENT>
class RefineableNavierStokesBoussinesqElement
  : public virtual NST_ELEMENT,
    public virtual ElementWithExternalElement
{
```

The constructor calls the constructors of the underlying elements, initialises the pointer to the Rayleigh number (stored as private member data in the class) and sets the number of interactions to one, indicating that the residuals of the Navier-Stokes element are only affected by a single type of "external element" – the advection-diffusion element that determines the temperature distribution.

```
public:
  /// Constructor: call the underlying constructors and
  /// initialise the pointer to the Rayleigh number to point
  /// to the default value of 0.0.
  RefineableNavierStokesBoussinesqElement()
    : NST_ELEMENT(), ElementWithExternalElement()
  {
    Ra_pt = &MultiDomainBoussinesqHelper::Default_Physical_Constant_Value;

    // There is one interaction: The effect of the advection-diffusion
    // element onto the buoyancy term
    this->set_ninteraction(1);
  }
```

We provide access functions to the Rayleigh number

```
  /// Access function for the Rayleigh number (const version)
  const double& ra() const
  {
    return *Ra_pt;
  }
```

and, given that we are dealing with a refineable element, make sure that the pointer to the Rayleigh number is passed to the "son" elements when the element is refined. Furthermore, if the external geometric data could safely be ignored in the "father" elements we assume that the same is true for the "sons":

```
  /// Access function for the pointer to the Rayleigh number
  double*& ra_pt()
  {
    return Ra_pt;
  }
```

The most important step is to overload the function that computes the body force in the Navier-Stokes equations so that it depends on the temperature at the `ipt` -th integration point, as computed by the "external element":

```
  /// Overload get_body_force_nst() to return the temperature-dependent
  /// buoyancy force, using the temperature computed by the
  /// "external" advection diffusion element associated with
  /// integration point \c ipt.
  void get_body_force_nst(const double& time,
                          const unsigned& ipt,
                          const Vector<double>& s,
                          const Vector<double>& x,
                          Vector<double>& body_force)
  {
    // Set interaction index -- there's only one interaction...
    const unsigned interaction = 0;

    // Get a pointer to the external element that computes the
    // the temperature -- we know it's an advection diffusion element.
    const AD_ELEMENT* adv_diff_el_pt =
      dynamic_cast<AD_ELEMENT*>(external_element_pt(interaction, ipt));

    // Get the temperature interpolated from the external element
    const double interpolated_t = adv_diff_el_pt->interpolated_u_adv_diff(
      external_element_local_coord(interaction, ipt));

    // Get vector that indicates the direction of gravity from
    // the Navier-Stokes equations
    Vector<double> gravity(NST_ELEMENT::g());

    // Set the temperature-dependent body force:
    const unsigned n_dim = this->dim();
    for (unsigned i = 0; i < n_dim; i++)
    {
      body_force[i] = -gravity[i] * interpolated_t * ra();
    }

  } // end overloaded body force
```

There is only one external interaction so the interaction index is set to zero and the value of the temperature at the integration point is obtained by casting the external element to an advection diffusion element and finding its interpolated field at the appropriate stored local coordinate. Once the temperature has been obtained, the code is identical to that discussed in the tutorial for the corresponding multi-field implementation; in that implementation the temperature could be found from a member function because the interaction is internal to

the element.
Finally, we have to compute the element's Jacobian matrix. The easiest (although potentially inefficient) way to do this is to recycle the analytical computation of the derivatives of the Navier-Stokes residuals with respect to the fluid degrees of freedom, as implemented in
`NST_ELEMENT::fill_in_contribution_to_jacobian(...)`,

```
    /// Compute the element's residual vector and the Jacobian matrix.
    void fill_in_contribution_to_jacobian(Vector<double>& residuals,
                                          DenseMatrix<double>& jacobian)
    {
      // Get the analytical contribution from the basic Navier-Stokes element
      NST_ELEMENT::fill_in_contribution_to_jacobian(residuals, jacobian);
```

and then fill in the derivatives with respect to the degrees of freedom associated with the "external elements" by finite differencing:

```
#ifdef USE_FD_FOR_DERIVATIVES_WRT_EXTERNAL_DATA_IN_MULTI_DOMAIN_BOUSSINESQ

      // Get the off-diagonal terms by finite differencing
      this->fill_in_jacobian_from_external_interaction_by_fd(residuals,
                                                             jacobian);

#else
```

If this is deemed to be too inefficient, we can provide a function that computes the required entries in the Jacobian analytically:

```
      // Get the off-diagonal terms analytically
      this->fill_in_off_diagonal_block_analytic(residuals, jacobian);

#endif
    }
```

(The code illustrates both approaches and employs the macro `USE_FD_FOR_DERIVATIVES_WRT_EXTERNAL_DATA_IN_MULTI_DOMAIN_BOUSSINESQ` to choose which one to use.)
We refer to **Part IV** for a discussion of how to implement the fully-analytic computation of the Jacobian matrix in the function `fill_in_off_diagonal_block_analytic(...)`.

1.2.2 Upgrading the advection-diffusion element to an ElementWithExternalElement

Upgrading the advection-diffusion element to an `ElementWithExternalElement` in which the wind is given by the "external" Navier Stokes element follows the same procedure. We use multiple inheritance to construct the element and set the number of interactions to one:

```
//======================ad_bous_class=================================
/// Build an AdvectionDiffusionElement that inherits from
/// ElementWithExternalElement so that it can "communicate" with the
/// a NavierStokesElement that provides its wind.
//====================================================================
template<class AD_ELEMENT, class NST_ELEMENT>
class RefineableAdvectionDiffusionBoussinesqElement
  : public virtual AD_ELEMENT,
    public virtual ElementWithExternalElement
{
public:
  /// Constructor: call the underlying constructors
  RefineableAdvectionDiffusionBoussinesqElement()
    : AD_ELEMENT(), ElementWithExternalElement()
  {
    // There is one interaction
    this->set_ninteraction(1);
  }
```

We overload the function that computes the "wind" for the advection diffusion equations so that it is given by the fluid velocity at the `ipt` -th integration point, as computed by the "external element":

```
    /// Overload the wind function in the advection-diffusion equations.
    /// This provides the coupling from the Navier--Stokes equations to the
    /// advection-diffusion equations because the wind is the fluid velocity,
    /// obtained from the "external" element
    void get_wind_adv_diff(const unsigned& ipt,
                           const Vector<double>& s,
                           const Vector<double>& x,
                           Vector<double>& wind) const
    {
      // There is only one interaction
      unsigned interaction = 0;

      // Dynamic cast "external" element to Navier Stokes element
      NST_ELEMENT* nst_el_pt =
        dynamic_cast<NST_ELEMENT*>(external_element_pt(interaction, ipt));

      // Wind is given by the velocity in the Navier Stokes element
      nst_el_pt->interpolated_u_nst(
```

```
        external_element_local_coord(interaction, ipt), wind);

    } // end of get_wind_adv_diff
```

Again, there is only one external interaction so the interaction index is set to zero and the external element must be cast to a Navier-Stokes element so that the interpolated velocity field can be found. The code is similar to that used in the multi-field implementation, which uses internal, rather than external, interaction.

The element's Jacobian matrix can be computed by the same methods discussed for the Navier-Stokes elements:

```
    /// Compute the element's residual vector and the Jacobian matrix.
    void fill_in_contribution_to_jacobian(Vector<double>& residuals,
                                          DenseMatrix<double>& jacobian)
    {
      // Get the contribution from the basic advection diffusion element
      AD_ELEMENT::fill_in_contribution_to_jacobian(residuals, jacobian);

#ifdef USE_FD_FOR_DERIVATIVES_WRT_EXTERNAL_DATA_IN_MULTI_DOMAIN_BOUSSINESQ

      // Get the off-diagonal terms by finite differencing
      this->fill_in_jacobian_from_external_interaction_by_fd(residuals,
                                                             jacobian);

#else

      // Get the off-diagonal terms analytically
      this->fill_in_off_diagonal_block_analytic(residuals, jacobian);

#endif
    }
```

1.3 Part III: The driver code for the multi-domain Boussinesq problem

Using the upgraded elements discussed above, the driver code for the multi-domain-based solution of the Boussinesq convection problem is very similar to that of the single-domain example. Indeed, the main() functions are virtually identical. The only difference is that the RefineableConvectionProblem takes two template arguments (specifying the two different element types) instead of one (specifying the type of the single, combined element).

1.3.1 The problem class

The problem class is similar to that of the single-domain example and many of the functions are the same. There are now two meshes, however, one for the fluid elements and one for the advection-diffusion elements:

```
 /// Access function to the NST mesh.
 /// Casts the pointer to the base Mesh object to
 /// the actual mesh type.
 RefineableRectangularQuadMesh<NST_ELEMENT>* nst_mesh_pt()
  {
   return dynamic_cast<RefineableRectangularQuadMesh<NST_ELEMENT>*>
    (Nst_mesh_pt);
  } // end_of_nst_mesh
 /// Access function to the AD mesh.
 /// Casts the pointer to the base Mesh object to
 /// the actual mesh type.
 RefineableRectangularQuadMesh<AD_ELEMENT>* adv_diff_mesh_pt()
  {
   return dynamic_cast<RefineableRectangularQuadMesh<AD_ELEMENT>*>
    (Adv_diff_mesh_pt);
  } // end_of_ad_mesh
```

and after any mesh adaptation, the interaction between the two meshes must be set up again.

```
 /// Actions after adaptation, reset all sources, then
 /// re-pin a single pressure degree of freedom
 void actions_after_adapt()
  {
   //Unpin all the pressures in NST mesh to avoid pinning two pressures
   RefineableNavierStokesEquations<2>::
    unpin_all_pressure_dofs(nst_mesh_pt()->element_pt());

   //Pin the zero-th pressure dof in the zero-th element and set
   // its value to zero
   fix_pressure(0,0,0.0);

   // Set external elements for the multi-domain solution.
   Multi_domain_functions::
    setup_multi_domain_interactions<NST_ELEMENT,AD_ELEMENT>
    (this,nst_mesh_pt(),adv_diff_mesh_pt());
  } //end_of_actions_after_adapt
```

1.3.2 The Problem constructor

The problem constructor is slightly different from the equivalent single-domain version, mainly because there are two meshes instead of one. Firstly, we build two coarse meshes with the same number of elements (9×8, as in the original single-domain problem) and dimensions for each mesh. Error estimators and error targets are set separately for each mesh.

```
//=======start_of_constructor=============================================
/// Constructor for adaptive thermal convection problem
//========================================================================
template<class NST_ELEMENT,class AD_ELEMENT>
RefineableConvectionProblem<NST_ELEMENT,AD_ELEMENT>::
RefineableConvectionProblem()
{
 // Set output directory
 Doc_info.set_directory("RESLT");
 // # of elements in x-direction
 unsigned n_x=9;

 // # of elements in y-direction
 unsigned n_y=8;

 // Domain length in x-direction
 double l_x=3.0;

 // Domain length in y-direction
 double l_y=1.0;
 // Build the meshes
 Nst_mesh_pt =
  new RefineableRectangularQuadMesh<NST_ELEMENT>(n_x,n_y,l_x,l_y);
 Adv_diff_mesh_pt =
  new RefineableRectangularQuadMesh<AD_ELEMENT>(n_x,n_y,l_x,l_y);

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

 // Set error targets for adaptive refinement
 Nst_mesh_pt->max_permitted_error()=0.5e-3;
 Nst_mesh_pt->min_permitted_error()=0.5e-4;
 Adv_diff_mesh_pt->max_permitted_error()=0.5e-3;
 Adv_diff_mesh_pt->min_permitted_error()=0.5e-4;
```

The boundary constraints are applied to each mesh in turn; firstly on the Navier–Stokes mesh, where we must also pin a single pressure:

```
 // Set the boundary conditions for this problem: All nodes are
 // free by default -- only need to pin the ones that have Dirichlet
 // conditions here

 //Loop over the boundaries of the NST mesh
 unsigned num_bound = nst_mesh_pt()->nboundary();
 for(unsigned ibound=0;ibound<num_bound;ibound++)
  {
   //Set the maximum index to be pinned (all values by default)
   unsigned val_max;

   //Loop over the number of nodes on the boundry
   unsigned num_nod= nst_mesh_pt()->nboundary_node(ibound);
   for (unsigned inod=0;inod<num_nod;inod++)
    {

     //If we are on the side-walls, the v-velocity and temperature
     //satisfy natural boundary conditions, so we only pin the
     //first value
     if((ibound==1) || (ibound==3))
      {
       val_max=1;
      }
     else
      {
       val_max=nst_mesh_pt()->boundary_node_pt(ibound,inod)->nvalue();
      }

     //Loop over the desired values stored at the nodes and pin
     for(unsigned j=0;j<val_max;j++)
      {
       nst_mesh_pt()->boundary_node_pt(ibound,inod)->pin(j);
      }
    }
  }
 // Pin the zero-th pressure value in the zero-th element and
 // set its value to zero.
 fix_pressure(0,0,0.0);
```

We then apply boundary constraints to the advection-diffusion mesh:

```
 //Loop over the boundaries of the AD mesh
 num_bound = adv_diff_mesh_pt()->nboundary();
```

```
for(unsigned ibound=0;ibound<num_bound;ibound++)
 {
  //Set the maximum index to be pinned (all values by default)
  unsigned val_max;

  //Loop over the number of nodes on the boundry
  unsigned num_nod= adv_diff_mesh_pt()->nboundary_node(ibound);
  for (unsigned inod=0;inod<num_nod;inod++)
   {
    //If we are on the side-walls, the v-velocity and temperature
    //satisfy natural boundary conditions, so we don't pin anything
    // in this mesh
    if ((ibound==1) || (ibound==3))
     {
      val_max=0;
     }
    else // pin all values
     {
      val_max=adv_diff_mesh_pt()->boundary_node_pt(ibound,inod)->nvalue();
      //Loop over the desired values stored at the nodes and pin
      for(unsigned j=0;j<val_max;j++)
       {
        adv_diff_mesh_pt()->boundary_node_pt(ibound,inod)->pin(j);
       }
     }
   }
 } // end of loop over AD mesh boundaries
```

and complete the build of all elements in each mesh by setting the pointers to the required physical parameters in each domain. Since neither of the interaction terms involves spatial derivatives of the field variables in the "external elements", we can ignore the derivatives with respect to the external geometric data when computing the element's Jacobian matrix. This is done by the calls to `ElementWithExternalElement::ignore_external_geometric_data();`

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

// Loop over the elements to set up element-specific
// things that cannot be handled by the (argument-free!) ELEMENT
// constructor.
unsigned n_nst_element = nst_mesh_pt()->nelement();
for(unsigned i=0;i<n_nst_element;i++)
 {
  // Upcast from GeneralsedElement to the present element
  NST_ELEMENT *el_pt = dynamic_cast<NST_ELEMENT*>
   (nst_mesh_pt()->element_pt(i));

  // Set the Reynolds number (1/Pr in our non-dimensionalisation)
  el_pt->re_pt() = &Global_Physical_Variables::Inverse_Prandtl;

  // Set ReSt (also 1/Pr in our non-dimensionalisation)
  el_pt->re_st_pt() = &Global_Physical_Variables::Inverse_Prandtl;

  // Set the Rayleigh number
  el_pt->ra_pt() = &Global_Physical_Variables::Rayleigh;

  //Set Gravity vector
  el_pt->g_pt() = &Global_Physical_Variables::Direction_of_gravity;

  // We can ignore the external geometric data in the "external"
  // advection diffusion element when computing the Jacobian matrix
  // because the interaction does not involve spatial gradients of
  // the temperature (and also because the mesh isn't moving!)
  el_pt->ignore_external_geometric_data();
 }

unsigned n_ad_element = adv_diff_mesh_pt()->nelement();
for(unsigned i=0;i<n_ad_element;i++)
 {
  // Upcast from GeneralsedElement to the present element
  AD_ELEMENT *el_pt = dynamic_cast<AD_ELEMENT*>
   (adv_diff_mesh_pt()->element_pt(i));

  // Set the Peclet number
  el_pt->pe_pt() = &Global_Physical_Variables::Peclet;

  // Set the Peclet number multiplied by the Strouhal number
  el_pt->pe_st_pt() =&Global_Physical_Variables::Peclet;

  // We can ignore the external geometric data in the "external"
  // Navier Stokes element when computing the Jacobian matrix
  // because the interaction does not involve spatial gradients of
  // the velocities (and also because the mesh isn't moving!)
  el_pt->ignore_external_geometric_data();

 } // end of setup for all AD elements
```

Finally we combine the submeshes, set up the interaction between the two meshes, and assign the equation

numbers.

```
 // combine the submeshes
 add_sub_mesh(Nst_mesh_pt);
 add_sub_mesh(Adv_diff_mesh_pt);
 build_global_mesh();
 // Set external elements for the multi-domain solution.
 Multi_domain_functions::
  setup_multi_domain_interactions<NST_ELEMENT,AD_ELEMENT>
  (this,nst_mesh_pt(),adv_diff_mesh_pt());
 // Setup equation numbering scheme
 cout << "Number of equations: " << assign_eqn_numbers() << endl;

} // end of constructor
```

1.3.3 The function actions_before_newton_solve(...)

In this function we update the problem specifications before a solution by re-applying the specific values of the Dirichlet boundary conditions for each mesh, and ensuring that the mass-conserving imperfection is applied to the velocity boundary conditions on the Navier–Stokes mesh if required. The boundary conditions are exactly the same as for the `single-domain version of the problem`, but they must be applied differently because there are now two meshes instead of one.

```
//==================start_actions_before_newton_solve====================
/// Update the problem specs before solve: (Re-)set boundary conditions
/// to include an imperfection (or not) depending on the control flag.
//=======================================================================
template<class NST_ELEMENT,class AD_ELEMENT>
void RefineableConvectionProblem<NST_ELEMENT,AD_ELEMENT>::actions_before_newton_solve()
{
 // Loop over the boundaries on the NST mesh
 unsigned num_bound = nst_mesh_pt()->nboundary();
 for(unsigned ibound=0;ibound<num_bound;ibound++)
  {
   // Loop over the nodes on boundary
   unsigned num_nod=nst_mesh_pt()->nboundary_node(ibound);
   for(unsigned inod=0;inod<num_nod;inod++)
    {
     // Get pointer to node
     Node* nod_pt=nst_mesh_pt()->boundary_node_pt(ibound,inod);

     //Set the number of velocity components
     unsigned vel_max=2;
     //If we are on the side walls we only pin the x-velocity.
     if((ibound==1) || (ibound==3)) {vel_max = 1;}
     //Set the pinned velocities to zero
     for(unsigned j=0;j<vel_max;j++) {nod_pt->set_value(j,0.0);}

     //If we are on the top boundary
     if(ibound==2)
      {
       //Add small velocity imperfection if desired
       if(Imperfect)
        {
         //Read out the x position
         double x = nod_pt->x(0);
         //Set a sinusoidal perturbation in the vertical velocity
         //This perturbation is mass conserving
         double value = sin(2.0*3.141592654*x/3.0);
         nod_pt->set_value(1,value);
        }
      }

    }
  }

 // Loop over all the boundaries on the AD mesh
 num_bound=adv_diff_mesh_pt()->nboundary();
 for(unsigned ibound=0;ibound<num_bound;ibound++)
  {
   // Loop over the nodes on boundary
   unsigned num_nod=adv_diff_mesh_pt()->nboundary_node(ibound);
   for(unsigned inod=0;inod<num_nod;inod++)
    {
     // Get pointer to node
     Node* nod_pt=adv_diff_mesh_pt()->boundary_node_pt(ibound,inod);

     //If we are on the top boundary, set the temperature
     //to -0.5 (cooled)
     if(ibound==2) {nod_pt->set_value(0,-0.5);}

     //If we are on the bottom boundary, set the temperature
     //to 0.5 (heated)
     if(ibound==0) {nod_pt->set_value(0,0.5);}
    }
```

```
  }


} // end of actions before solve
```

1.3.4 The function doc_solution(...)

This function outputs all fields to the specified solution file in the directory pointed to by the `DocInfo` object.

```
//======================start_of_doc_solution==============================
/// Doc the solution
//=========================================================================
template<class NST_ELEMENT,class AD_ELEMENT>
void RefineableConvectionProblem<NST_ELEMENT,AD_ELEMENT>::doc_solution()
{
 //Declare an output stream and filename
 ofstream some_file;
 char filename[100];

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

 // Output Navier-Stokes solution
 sprintf(filename,"%s/fluid_soln%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 nst_mesh_pt()->output(some_file,npts);
 some_file.close();

 // Output advection diffusion solution
 sprintf(filename,"%s/temperature_soln%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 adv_diff_mesh_pt()->output(some_file,npts);
 some_file.close();


 Doc_info.number()++;
} // end of doc
```

1.4 Part IV: Optimising multi-domain interactions

We showed in **Part II** that only a small number of functions **must** be implemented to upgrade an existing single-physics element to an `ElementWithExternalElement` that can interact with another element in a different domain. This is because much of the required functionality is already implemented in the `ElementWithExternalElement` base class, which makes the implementation of multi-physics interactions very easy. The use of general-purpose functions may incur unnecessary computational cost, however. Hence, a better efficiency can be achieved by overloading certain functions when implementing a specific `ElementWithExternalElement`.

1.4.1 Ignoring field data that does not participate in the interaction

The function `ElementWithExternalElement::identify_all_field_data_for_external_interaction(...)` assumes that **all** data values in the external elements affect the interaction with the `ElementWithExternalElement`. In many cases this assumption is overly pessimistic (and costly). For instance in the Boussinesq convection problem only the velocity degrees of freedom in the (external) Navier-Stokes element affect the the advection-diffusion equations via the "wind". It is therefore sensible to exclude the pressure degrees of freedom from the interaction by re-implementing `identify_all_field_data_for_external_interaction(...)` as follows

```
  //================optimised_identification_of_field_data=================
  /// Overload the function that must return all field data involved
  /// in the interaction with the external (Navier Stokes) element.
  /// Only the velocity dofs in the Navier Stokes element affect the
  /// interaction with the current element.
  //=========================================================================
  template<class AD_ELEMENT, class NST_ELEMENT>
  void RefineableAdvectionDiffusionBoussinesqElement<AD_ELEMENT, NST_ELEMENT>::
    identify_all_field_data_for_external_interaction(
      Vector<std::set<FiniteElement*> const& external_elements_pt,
      std::set<std::pair<Data*, unsigned>& paired_interaction_data)
  {
    // There's only one interaction
    const unsigned interaction = 0;

    // Loop over each Navier Stokes element in the set of external elements that
    // affect the current element
    for (std::set<FiniteElement*>::iterator it =
           external_elements_pt[interaction].begin();
```

```
         it != external_elements_pt[interaction].end();
         it++)
    {
      // Cast the external element to a fluid element
      NST_ELEMENT* external_fluid_el_pt = dynamic_cast<NST_ELEMENT*>(*it);

      // Loop over the nodes
      unsigned nnod = external_fluid_el_pt->nnode();
      for (unsigned j = 0; j < nnod; j++)
      {
        // Pointer to node (in its incarnation as Data)
        Data* veloc_data_pt = external_fluid_el_pt->node_pt(j);

        // Get all velocity dofs
        const unsigned n_dim = this->dim();
        for (unsigned i = 0; i < n_dim; i++)
        {
          // Which value corresponds to the i-th velocity?
          unsigned val = external_fluid_el_pt->u_index_nst(i);

          // Turn pointer to Data and index of value into pair
          // and add to the set
          paired_interaction_data.insert(std::make_pair(veloc_data_pt, val));
        }
      }
    }
  } // done
```

Similar approaches can be used to ignore selected (weak) interactions when computing the element's Jacobian matrix. For instance, the `FSIWallElement` is an `ElementWithExternalElement` for which the adjacent fluid elements that apply the fluid traction to the FSI boundary act as "external elements". In high-Reynolds-number flows, the fluid traction is dominated by the pressure while shear stresses tend to be small. The `FSIWall↩Element` therefore uses the same mechanism as illustrated above to (optionally) neglect the derivatives of its residuals with respect to the fluid velocity degrees of freedom. We stress that this does not exclude the shear stress from the computation – it simply replaces the exact Jacobian by an approximate version in which the effect of the velocity degrees of freedom on the residuals of the `FSIWallElement` are neglected. This may lead to a slight degradation in the convergence rate of the Newton iteration but this may be more than compensated for by the reduction in the CPU times required to compute the Jacobian matrix.

1.4.2 Ignoring geometric data

If the interaction with the "external element" does not involve spatial derivatives of the fields represented by the "external element" or if the mesh containing the "external elements" is fixed, the external geometric data associated with the "external elements" can be ignored when computing the `ElementWithExternalElement's` Jacobian. This may be achieved by calling

```
ElementWithExternalElement::ignore_external_geometric_data();
```

1.4.3 Computing the off-diagonal blocks in the Jacobian analytically

The setup of the Jacobian matrix can be made much more efficient by computing the derivatives of the `Element↩WithExternalElements'` residual vector with respect to the field data in the associated "external elements" analytically. The multi-domain driver codes in

demo_drivers/multi_physics/boussinesq_convection

demonstrate a possible implementation. The key challenge for the implementation is that the `ElementWith↩ExternalElement` must label the entries in its elemental Jacobian matrix by its local equation numbers, whereas the "external element" can only compute the derivative of its fields with respect to its own (differently numbered) local degrees of freedom (d.o.f.s). To establish which local d.o.f. in the "external element" corresponds to a given local d.o.f. in the `ElementWithExternalElement` we exchange the (unique) global equation numbers associated with each d.o.f. Using this trick, the computation of the derivatives becomes relatively straightforward. Mathematically, it involves repeated applications of the chain rule. In the cases we considered, the fully-analytic computation of the elemental Jacobian matrix was about 3 to 4 times faster than finite-difference-based computation implemented in the `ElementWithExternalElement` base class. Whether the speedup is worth the additional (human) time required to implement the analytic computation of the off-diagonal entries depends on the application (and you!). If nothing else, the availability of a finite-difference based routine helps in the validation of any newly-developed analytic re-implementation.

1.5 Comments and Exercises

1.5.1 Comments

- The use of two separate meshes with different error estimators means that, in principle, a more accurate solution can be obtained with fewer degrees of freedom than using a combined error estimator on a single mesh. The combined error estimator implemented in the `single-domain version of the problem` will cause refinement if either the fluid error or the advection-diffusion error is above the specified tolerance. Thus, a fully-converged solution in the single-domain problem is one in which the estimated error in all field variables is below the tolerance, but this may be at the cost of some over-refinement in one or more of the field variables. In the present multi-domain approach, if the solution is converged on both meshes then the estimated error in all field variables is again below the tolerance. Hence, it is fair, in some sense, to compare the results between fully-converged solutions for the single- and multi-domain problems. As suggested by the figure above, for the Boussinesq convection problem considered here we do obtain fully-converged solutions with fewer degrees of freedom using the multi-domain approach because the temperature field is over-refined in the single-domain case. Of course, the multi-domain solution includes the extra overhead of setting up the interaction, but, in general, this cost is negligible compared to the solution of the linear systems.

- The general procedures described in this tutorial can be used to set up any interaction between different types of elements using multiple meshes.

- Before any refinements take place the combined Jacobian is exactly the same as that in the single-domain problem and so the residuals at each Newton step will be exactly the same. You can verify this by comparing the appropriate output files in the `Validation` directory.

1.5.2 Exercises

1. Investigate the difference between the solutions for the multi-domain and single-domain problems by continuing to refine until the solutions are fully-converged to a given error tolerance. What is the difference in total number of degrees of freedom? What is the difference in solution time? What is the difference between the two solutions?

2. Investigate double-diffusive convection by adding another advection-diffusion mesh to the problem that interpolates a concentration field. Examples may be found in the `double_diffusive_convection` directory.

1.6 Source files for this tutorial

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

  `demo_drivers/multi_physics/boussinesq_convection`

  which contains refineable and non-refineable multi-domain versions of the Boussinesq convection problem.

- The full driver code for the problem described in this tutorial is:

demo_drivers/multi_physics/boussinesq_convection/multi_domain_ref_b_convection.cc

- The corresponding driver code for the non-refineable version of the problem is:

demo_drivers/multi_physics/boussinesq_convection/multi_domain_boussinesq_convection.cc

- The source code for the elements is in:

src/multi_physics/multi_domain_boussinesq_elements.h

1.7 PDF file

A pdf version of this document is available.