Chapter 1

A two-dimensional Biharmonic problem with the C1-curved triangular finite element

In the previous tutorial, the triangular Bell element has been successfully used to solve the Biharmonic equation which is a fourth-order problem in two dimensions. The domains of interest considered in the particular problem contained only straight boundaries so that this representation was exact. However, in many engineering applications, the geometric boundary of a problem is not straight. Solving such a problem with the straight-sided C^1-finite elements limits the convergence rate and accuracy as presented in a publication of P. Fischer(2010) .

In order to deal with a fourth-order problem, which requires C^1-continuity, on a curvilinear domain, the C^1-curved triangular element is introduced in this document. Numerical implementations of the Biharmonic equation on a circular domain will be presented. This is to compare the results obtained between the Bell and the C^1-curved triangular elements. Both rate of convergence and the obtained accuracy will be determined together with the computational time. This is to show that representing a curved boundary by a series of straight-sided elements exhibits a limitation in convergence rate and accuracy. Also, we would like to illustrate that the C^1-curved triangular element can retain the rate of convergence and accuracy when a curvilinear boundary is concerned.

The reader is referred to a two-dimensional Biharmonic problem with the Bell triangular finite element tutorial for more detailed descriptions of the variational principle of the Biharmonic equation and the Bell element.

In addition, in this document we demonstrate

- general descriptions of the `C1CurvedElement` which is our defined C^1-triangular finite element dealing with a 2D curvilinear domain
- numerical results of solving the Biharmonic equation with the `C1CurvedElement`

and

- how to implement the `C1CurvedElement` in `oomph-lib` using the Biharmonic equation as a case study.

1.1 Overview of the C1CurvedElement

The constructed `C1CurvedElement` provides a discretisation of a problem with two-dimensional, subparametric triangular finite elements. With this element, the discretisation of the domain of interest in this document will be the union of two sets of triangles. The first set will constitute of straight-sided triangles which is the triangulation of interior elements. The other will constitute of curve-sided triangles which are elements at a curved boundary. Note that each of two distinct triangles of the triangulations is either disjoint, or have a common vertex or a common edge. Consequently, we have to define mappings associated the reference and the physical triangles for each straight and curved triangles.

Regarding the approximation of variables on the straight boundary domain, the Bell shape functions will be employed. On the curved boundary domain, the shape functions will be re-constructed compatible with the Bell shape functions. These new functions will be defined over the approximated domain associated with the mappings approximating the curved boundary. Finally, all of these shape functions will be used to define interpolations of variables on the typical elements.

Specifically, we shall provide

- The mappings associated the reference triangle and the physical triangles
- The shape functions defined on the `C1CurvedElement`
- The interface of accessible functions to the shape functions

1.1.1 The mappings associated the reference triangle with the physical triangles

In order to approximate the geometry in the `C1CurvedElement`, the subparametric mapping which associates the reference and the physical triangles is considered in this study. Also, we will define mapping by using polynomials to approximate boundaries.

Since there are two sets of triangular elements in the mesh, different mappings have to be considered for typical elements. Dealing with a straight-sided triangle in a physical domain, an affine mapping will be taken into account. Nonetheless, a nonlinear mapping has to be considered in order to deal with a curved-edge triangle.

Regarding a straight-sided triangle, we define the affine mapping F_{K_I} which is parametrised by the reference coordinates $\hat{x}_\alpha, \alpha = 1, 2$, defined on the reference triangle $\hat{K} = \{(\hat{x}_1, \hat{x}_2) | 0 \leq \hat{x}_1, \hat{x}_2 \leq 1, \hat{x}_1 + \hat{x}_2 = 1\}$ and can be constructed as follow

$$x_\alpha = F_{K_{I\alpha}}(\hat{x}_1, \hat{x}_2) = x_{\alpha 3} + (x_{\alpha 1} - x_{\alpha 3})\hat{x}_1 + (x_{\alpha 2} - x_{\alpha 3})\hat{x}_2, \qquad (1)$$

where $x_{\alpha i}$ denote the global coordinates $\alpha = 1, 2$ of the vertices $\mathbf{a}_i, i = 1, 2, 3$, of the physical triangle K_I as depicted in the following figure. Also, $F_{K_{I\alpha}}$ is a α^{th} component of the mapping F_{K_I}. This mapping will help us to associate a position $(\hat{x}_1, \hat{x}_2)$ on the reference triangle with a position (x_1, x_2) on the physical triangle whose sides are straight.

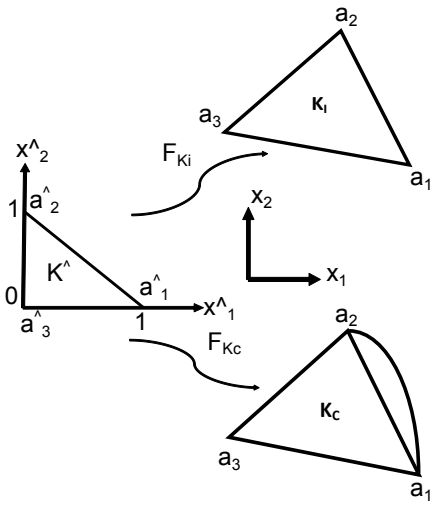

Figure 1.1 A graphical description of the mappings associated between the reference coordinates and the physical coordinates.

Regarding the curved boundary, the mapping F_{K_C} is defined to associate between the reference and the physical curved triangles. These curved triangles are considered to have two straight sides and one curved side approximating an arc of the boundary. In this case, the mapping F_{K_C} is nonlinear and can be constructed as in the following descriptions.

Let Ω be a given bounded domain on the plane whose boundary is curve in two-dimensional space. We assume that the curved boundary Γ can be subdivided into a finite number of arcs and each of them can be described as

$$x_1 = \chi_1(s), \quad x_2 = \chi_2(s), \quad s_m \leq s \leq s_M,$$

where $x_\alpha, \alpha = 1, 2$, denote coordinates in two-dimensional space defined on the curved boundary Γ and approximated by the polynomials $\chi_1(s), \chi_2(s)$ which are continuous on the arc $[s_m, s_M]$.

Before we will consider the nonlinear mapping F_{K_C}, we will first determine a mapping φ_α which associates the side $\hat{\mathbf{a}}_\mathbf{1}\hat{\mathbf{a}}_\mathbf{2}$ on the reference triangle, $\hat{K}$, with the curved boundary Γ. This mapping is defined to be parametrised by the reference coordinate, say $\hat{x}_2$, and satisfies

$$\begin{aligned} \varphi_\alpha(\hat{x}_2 = 0) = \chi_\alpha(s_m) = x_{\alpha 1} = \mathbf{a}_1, \\ \varphi_\alpha(\hat{x}_2 = 1) = \chi_\alpha(s_M) = x_{\alpha 2} = \mathbf{a}_2, \end{aligned} \qquad (2)$$

where $\varphi_\alpha(\hat{x_2}) \equiv \chi_\alpha(s_m + (s_M - s_m)\hat{x}_2)$. Note that either the reference coordinate $\hat{x}_1$ or $\hat{x}_2$ can be used to parametrise the arc but re-calculation will be required to defined the mapping.

Here, the Hermite-type polynomial of degree 3 is our choice to approximate the curved boundary. The reason for this choice is that cubic is the minimum degree of polynomial to obtain the C^1-continuity (see M.Bernadou (1993)). Hence, the following four conditions have to be specified

$$\begin{aligned} x_{\alpha 1} = \varphi_\alpha(0) = \chi_\alpha(s_m), \; x_{\alpha 2} = \varphi_\alpha(1) = \chi_\alpha(s_M), \\ \varphi'_\alpha(0) = (s_M - s_m)\chi'_\alpha(s_m), \; \varphi'_\alpha(1) = (s_M - s_m)\chi'_\alpha(s_M). \end{aligned} \qquad (3)$$

These conditions are constraints in order to satisfy (2) for the coordinate values x_α in order to obtain the continuity.

By considering the cubic polynomial, $\varphi_\alpha(\hat{x}_2)$, in the form of (4) with the conditions for derivatives in (3), the mapping $\varphi_\alpha(\hat{x}_2)$ can be obtained.

$$\varphi_\alpha(\hat{x}_2) = x_{\alpha 1} + (x_{\alpha 2} - x_{\alpha 1})\hat{x}_2 + \hat{x}_2(1-\hat{x}_2)[m_\alpha \hat{x}_2 + c_\alpha]. \quad (4)$$

Regarding the nonlinear mapping F_{K_C}, it is defined to approximate the coordinates $x_\alpha, \alpha = 1,2$, so that the side $\hat{\mathbf{a}}_\mathbf{1}\hat{\mathbf{a}}_\mathbf{2}$ on the reference triangle, $\hat{K}$, associates with the curved side of the curved physical triangle. Also, the mapping F_{K_C} has to be an affine mapping along the side $\mathbf{a}_3\mathbf{a}_\alpha, \alpha = 1,2$. Therefore, the nonlinear mapping F_{K_C} that associates the reference and the curved elements with the cubic polynomial approximating a curved boundary is obtained as follow

$$x_\alpha = F_{K_{C\alpha}}(\hat{x}_1, \hat{x}_2) = x_{\alpha 3} + (x_{\alpha 1} - x_{\alpha 3})\hat{x}_1 + (x_{\alpha 2} - x_{\alpha 3})\hat{x}_2$$

$$+\frac{1}{2}\hat{x}_1\hat{x}_2\left\{[2(x_{\alpha 2} - x_{\alpha 1}) - (s_M - s_m)(\chi'_\alpha(s_m) + \chi'_\alpha(s_M))]\,(\hat{x}_2 - \hat{x}_1) + (s_M - s_m)\,[\chi'_\alpha(s_m) - \chi'_\alpha(s_M)]\right\}, \quad (5)$$

where $F_{K_{C\alpha}}$ denotes an α^{th} component of the mapping F_{K_C}. Note that the nonlinear mapping $F_{K_{C\alpha}}(1-\hat{x}_2, \hat{x}_2) = \varphi_\alpha(\hat{x}_2)$ when comes to the side $\mathbf{a}_1\mathbf{a}_2$, as decided.

The derivation of the nonlinear mapping F_{K_C} can be found in a publication of M.Bernadou (1993) .

1.1.2 The shape functions defined on the C1CurvedElement

In this section, we will elaborate the shape functions defined on the `C1CurvedElement`. Since there are two types of elements defined on the `C1CurvedElement`, two sets of shape functions will be considered.

On the straight-sided triangular element, we inherit the `BellElement` to approximate values defined on the element. Therefore, the Bell shape functions will be employed and their details can be found in the Bell triangular finite element tutorial. Note that the Bell shape functions are defined with respect to the global coordinates so that the transformation of derivatives between the local and the global coordinates is no longer need in an implementation.

Next, we will define shape functions of the C^1-curved finite element in order to interpolate any function defined over the curve-sided triangular element. Unlike the interpolation functions defined on the Bell triangular element, the interpolation functions of the C^1-curved triangle are chosen to defined over the reference triangle $\hat{K}$. Therefore, for any polynomial $p \in P_K$ defined over the curved triangle K_C, we employ the mapping defined in (5) to associate this polynomial p with the polynomial defined on the reference triangle as

$$\hat{p} = p \circ F_{K_C}.$$

Note that defining a polynomial on the reference triangle, $\hat{K}$, is a desirable condition which is convenient for the study of the approximation error and to take into account the numerical integration.

Subsequently, using a polynomial of degree 3 to approximate a curved boundary in our study, polynomials of degree 7 have to be defined as shape functions over the reference triangle in order to interpolate unknowns on a curved triangle. This is in order to achieve the required convergence and to satisfy the C^1-compatibility with the Bell finite elements. The reader may refer to a publication of M.Bernadou (1993) for more details.

The desired P_7- C^1 reference element will constitute of $\hat{K}$ which is a unit right-angled triangle, the set of degrees of freedom $\hat{\Sigma}(\hat{w})$, and $\hat{P}$ which is a space of complete polynomials of degree 7 with its dimension equals to 36. The set of $\hat{\Sigma}(\hat{w})$ composes of values and their derivatives defined on vertices, $\hat{\mathbf{a}}_i, i = 1,2,3$, and along edges, $\hat{\mathbf{b}}_i, i = 1,2,3$, $\hat{\mathbf{d}}_i, i = 1,...,6$, and the internal nodes, $\hat{\mathbf{e}}_i, i = 1,2,3$, of the reference triangle illustrated in the following figure and are defined as

$$\hat{\Sigma}(\hat{w}) = \left\{ \hat{w}(\hat{\mathbf{a}}_i), \frac{\partial \hat{w}}{\partial \hat{x}_1}(\hat{\mathbf{a}}_i), \frac{\partial \hat{w}}{\partial \hat{x}_2}(\hat{\mathbf{a}}_i), \frac{\partial^2 \hat{w}}{\partial \hat{x}_1^2}(\hat{\mathbf{a}}_\mathbf{i}), \frac{\partial^2 \hat{w}}{\partial \hat{x}_1 \hat{x}_2}(\hat{\mathbf{a}}_i), \frac{\partial^2 \hat{w}}{\partial \hat{x}_2^2}(\hat{\mathbf{a}}_i), i = 1, 2, 3 \right\}$$

$$\cup \left\{ -\frac{\partial \hat{w}}{\partial \hat{x}_1}(\hat{\mathbf{b}}_1); -\frac{\partial \hat{w}}{\partial \hat{x}_2}(\hat{\mathbf{b}}_2); \frac{\sqrt{2}}{2} \left(\frac{\partial \hat{w}}{\partial \hat{x}_1} + \frac{\partial \hat{w}}{\partial \hat{x}_2} \right) (\hat{\mathbf{b}}_3) \right\} \cup$$

$$\left\{ \hat{w}(\hat{\mathbf{d}}_i), i = 1, ..., 6; -\frac{\partial \hat{w}}{\partial \hat{x}_1}(\hat{\mathbf{d}}_i), i = 1, 2; -\frac{\partial \hat{w}}{\partial \hat{x}_2}(\hat{\mathbf{d}}_i), i = 3, 4; \frac{\sqrt{2}}{2} \left(\frac{\partial \hat{w}}{\partial \hat{x}_1} + \frac{\partial \hat{w}}{\partial \hat{x}_2} \right) (\hat{\mathbf{d}}_i), i = 5, 6 \right\}$$

$$\cup \{ \hat{w}(\hat{\mathbf{e}}_i), i = 1, 2, 3 \},$$

where $\hat{w}$ denotes a function defined over the reference element $\hat{K}$.

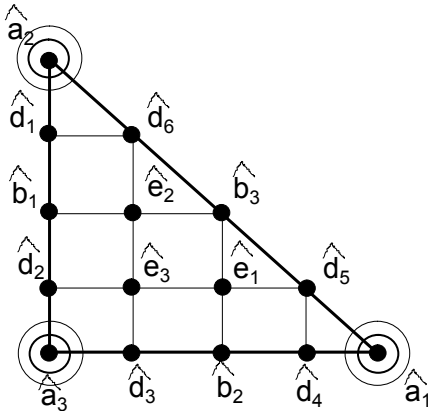

Figure 1.2 The reference element of the C1-curved finite element compatible with the Bell triangle where the degree of polynomial approximating curved boundaries is cubic.

Since the $P_7 - C^1$-shape functions of the C^1-curved finite element are defined on the reference triangle $\hat{K}$, its 36 degrees of freedom have to be associated with those defined on the curved physical triangle. To consider the degrees of freedom defined on the curved physical triangle, they have to ensure the connection of class C^1-compatible with the Bell triangles as the Bell triangles are its adjacent elements.

To define such a connection, the following conditions have to be satisfied to assure a compatibility between a curved finite element and the Bell triangle.

- One-variable polynomials of any function p defined over the curved triangle and their normal derivatives $\frac{\partial p}{\partial n}$ along the connected sides have to coincide with those of the Bell elements. That is the degrees of the polynomial and its normal derivatives along the connecting sides have to be of degree 5 and 3, respectively.
- the degrees of freedom of the curved finite element relative to the connecting sides have to be identical to those of the Bell elements and have to be entirely determined on those sides.

Therefore, we have that our C^1-curved triangular finite element has to define so that the set of degrees of freedom on the vertices of the curved physical triangle is the same as those of the Bell element.

Furthermore, there will be three additional degrees of freedom defined inside the curved triangle K_C as seen in Fig. 1.3. These nodes come from three internal nodes defined in order to ensure the C^1-continuity in determining the polynomial of degree 7 on the reference element. Consequently, there are 21 degrees of freedom in total defined over the curved triangular element.

The set $\Sigma_K(v)$ of values of degrees of freedom of v defined on the curved physical triangle is then given by

$$\Sigma_K(v) = \{(D^\alpha v(\mathbf{a}_i), \alpha = 0, 1, 2), i = 1, 2, 3; v(\mathbf{e}_i), i = 1, 2, 3\}. \quad (6)$$

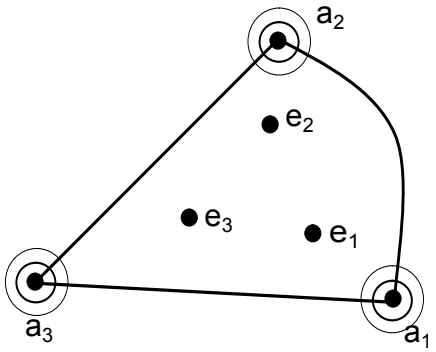

Figure 1.3 The set of degrees of freedom for the C1-curved finite element compatible with the Bell element.

In order to interpolate a function v defined on the curved physical triangle, the $P_7 - C^1$-shape functions and the 36 associated values of degrees of freedom have to be taken into account. Since the 36 nodal degrees of freedom depend on 21 nodal degrees of freedom defined on K_C illustrated in (6), the association between those values of degrees of freedom have to be defined.

Since the number of degrees of freedom defined on the curved and the reference triangles are different, the derivation of the associations is not straightforward. Detailed descriptions of the derivation of the association $\tilde{M}$ can be found in a publication of M.Bernadou (1993).

1.1.3 The interpolation of a function, v, defined on the domain

The interpolation of a function v defined on the physical triangle can be determined associated with the straight and the curved element.

If we want to interpolate a function v defined on the straight physical triangle, the Bell shape functions have to be taken into account. Therefore, the interpolation can be defined as

$$u(x_1, x_2) = \sum_{l=1}^{3} \sum_{k=1}^{6} u_{lk} \psi_{lk}^B(x_1, x_2), \quad (7)$$

where u_{lk} and $\psi_{lk}^B, l = 1, 2, 3, k = 1, ..., 6$, are the nodal values and the Bell shape functions defined on the physical element K_I, respectively.

Otherwise, the interpolation of a function v defined on the curved physical triangle can be determined by quantities defined on the reference triangle as

$$u(x_1(\xi), x_2(\xi)) = \sum_{j=1}^{36} \hat{u}_j \hat{\psi}_j(\xi_1, \xi_2), \quad (8)$$

where $\hat{u}_j$ are the nodal values defined at node j on the reference triangle $\hat{K}$. Also, $\hat{\psi}_j, j = 1, ..., 36$, are the C^1-shape functions defined by complete polynomials of degree 7 on the reference element $\hat{K}$.

Since all these nodal values $\hat{u}_j$ are defined to depend only on 21 nodal values, u_k, defined on the curved element K_C, the set of nodal values defined on the reference element can be associated with those defined on the curved element through the association $\tilde{M}$ by

$$\hat{u}_j = \sum_{k=1}^{21} u_k \tilde{M}_{kj}, \forall j = 1, ..., 36, \qquad (9)$$

Substituting (9) into (8), we have that the approximation of the unknown u can be expressed as the linear combination between the shape functions defined on the reference triangle $\hat{K}$ and the values defined on the physical curved triangle as follow

$$u(x_1(\xi), x_2(\xi)) = \sum_{j=1}^{36} \sum_{k=1}^{21} u_k \tilde{M}_{kj} \hat{\psi}_j(\xi_1, \xi_2).$$

Note that the shape functions of the C^1-curved element can be determined from the δ_{ij} property of the shape function. Also, they are defined with respect to the local coordinates. Hence, the transformation of derivatives between the local and the global coordinates is needed in an implementation in order to ensure C^1-continuity.

1.1.4 The interface of accessible functions to the shape functions of the C1CurvedElement

Regarding the curvilinear boundary domain study in this tutorial, its triangulation constitutes of both straight-sided and curve-sided triangle. Different number of nodes and shape functions have to be considers in `C1Curved↩ Element` in order to associate with both the straight and curved triangular elements in the triangulation.

Now, we will consider the face-element of the `C1CurvedElement`. According to section 1.1.2, we have that 21 degrees of freedom defined on the C^1-curved triangular element are associated with three vertices and three interior node of the curved physical triangle. Unlike the C^1-curved triangular elements, all 18 degrees of freedom of the Bell element are defined on only three vertices.

Therefore, the face element of the `C1CurvedElement` have to define generally in order to associate with both straight-sided and curve-sided elements. Consequencely, there are six nodes in total in a `C1CurvedElement` with `NNODE_1D` equals to 2. The following figure illustrates the number of nodes defined in both the Bell and the C^1-curved triangular elements.

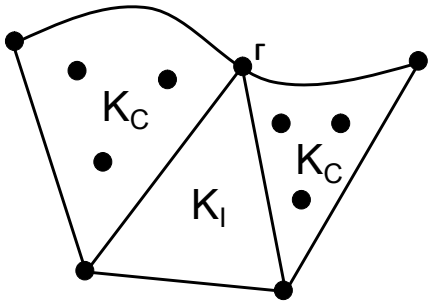

Figure 1.4 The graphical description of the face-element defined in the `C1CurvedElement`.

Apparently, our constructed `C1CurvedElement` are only available for three interior nodes for all cases of `NN↩ ODE_1D`. The fixed number of interior nodes is a consequence of using degree 3 of polynomial to approximate a curvilinear boundary as mentioned in subsection 1.1.1.

In order to approximate the geometry, the linear mapping defined in (1) is employed to associate with the straight-sided triangles while the nonlinear mapping defined in (5) is employed to associate with the curved triangles. Those mapping will be inherited from GeometricCurvedTElementShape<2> and can be accessible in `C1Curved↩ Elemet<>` by

```
C1CurvedElement<>::shape(s,x),
```

Note that, unlike other elements in `oomph-lib`, the geometric shape functions in `C1CurvedElement` rather computed the global position of the local coordinate s.

Regarding shape functions defined on the `C1CurvedElement`, different definitions of shape functions are defined in order to associate with both straight and curved triangular elements. On the set of straight-sided triangles, the Bell shape functions will be employed to approximate the unknowns. When the curve-sided triangles are concerned, the C^1-curved triangular shape functions defined in subsection 1.1.2 will be used to approximate the unknowns. Hence, the `C1CurvedElement` is a C^1-triangular element carries two families of C^1-shape functions defined on both straight and curved triangles.

The shape functions of both Bell and C^1-curved triangular elements can be accessible via `BellElement↩ Shape::Bshape(...)` and `C1CurvedElementShape::Cshape(...)`, respectively. These functions will compute the shape functions at local coordinate s.

As the global coordinates are required in the derivation of the shape functions of the `BellElementShape<>`, the physical coordinates of vertices have to be passed as an argument when shape functions are overloaded. The interface of the Bell shape functions obtained from `BellElementShape<>` at local coordinate s is `Bell↩ ElementShape<>::Bshape(s,psi,position)`. Similarly, the first and the second-order derivatives can be obtained at local coordinate s as `BellElementShape<>::dBshape(s,psi,dpsi,position)` and `BellElementShape<>::d2Bshape(s,psi,dpsi,d2psi,position)`, respectively. The vector `position` is the collection of coordinates on vertices of straight-edged triangles.

Now, to obtained the basis functions that employed to approximate variables on the straight-sided triangles in `C1↩ CurvedElement`, they can be accessible via

```
C1CurvedElement<>::basis_straight(s,psi),
```

Furthermore, the first- and second-order derivatives of the Bell shape functions in `C1CurvedElement<>` can be accessible via

```
C1CurvedElement<>::dbasis_straight(s,psi,dpsi),
```

and

```
C1CurvedElement<>::d2basis_straight(s,psi,dpsi,d2psi),
```

respectively.

Unlike the Bell shape functions, the C^1-curved shape functions has a usual interface as no global coordinate is required. They can be overloaded from `C1CurvedElementShape<>::Cshape(s,psi)` while their first- and

second-order derivatives can be obtained from `C1CurvedElementShape<>::dCshape(s,psi,dpsi)` and `C1CurvedElementShape<>::d2Cshape(s,psi,dpsi,d2psi)`, respectively.

Now, to obtained the basis functions that employed to approximate variables on the curved triangles in `C1CurvedElement`, they can be accessible via

```
C1CurvedElement<>::basis_curve(s,psi),
```

Furthermore, the first- and second-order derivatives of the C^1-curved shape functions in `C1CurvedElement` can be accessible via

```
C1CurvedElement<>::dbasis_curve(s,psi,dpsi),
```

and

```
C1CurvedElement<>::d2basis_curve(s,psi,dpsi,d2psi),
```

respectively.

Regarding the association, which associated the values of degrees of freedom between the reference and physical curved triangles, mentioned in (9), it can be inherited from the functions `C1CurvedElementShape<2>::set_of_value(D,B,node_position,bd_element,bd_node_position,x)` and the following are required as its arguments

- coordinates of three vertices of a curved triangle passed as `node_position`,
- coordinates of the vertices of a curved triangle that situate on a curved boundary passed as `bd_node_position`,
- coordinates of the vertices of a curved triangle that is off-boundary passed as `x`.

Note that the matrices D, B denote the submatrices of value transformation described in a publication of M.Bernadou (1993) .

1.2 Solving the Biharmonic equation with the C1CurvedElement

The sections below provide an example of solving the fourth-order problem using `C1CurvedElement`. The fourth-order problem consider here is the same Biharmonic equation described in another tutorial which is expressed as follow:

$$\frac{\partial^4 u}{\partial x^4} + 2\frac{\partial^4 u}{\partial x^2 \partial y^2} + \frac{\partial^4 u}{\partial y^4} = 0, \quad (10)$$

with the exact solution $u(x,y) = \cos(x)e^y$. However, in this document, the problem is implemented in the quarter of the unit circular domain $\Omega = \{(x,y)|x,y \in [0,1], x^2+y^2 = 1\}$ which is a curvilinear boundary domain rather than a straight-sided boundary domain. Similarly, we also apply Dirichlet boundary conditions such that the exact solution is satisfied. Also, the conditions that have to be specified on the boundaries have to correspond with the degrees of freedom defined for the `C1CurvedElement` in the previous section.

Since 36 values of degrees of freedom defined on the reference triangle depend on the 21 values of degrees of freedom defined on the physical curved triangle, those 21 degrees of freedom on the physical curved triangle have to be imposed. These degrees of freedom constitute of 6 degrees of freedom at 3 vertex nodes and the values of degrees of freedom at 3 internal nodes. The 6 degrees of freedom defined on vertices incorporate the value of unknown field, the first derivatives with respect to the first and second coordinates, and the second derivatives with respect to the first, second, and mix derivatives have to be determined at the boundaries. The boundary specifications can be determined the same as described in a two-dimensional Biharmonic problem with the Bell triangular finite element tutorial.

Here are two tables comparing the performance with the associated computational time between the `Bell↩Element` and the `C1CurvedElement`. The accuracy of the solutions will base on the L^2-norm error which is mathematically described as

$$|u_{exact} - u_{FE}| = \left(\int_\Omega |u_{exact} - u_{FE}|^2 d\Omega \right)^{1/2},$$

where u_{exact}, u_{FE} denote the exact and the finite element solutions, respectively.

Table 1: L^2-norm error of the solution obtained from the Biharmonic implementations using the BellElement with various numbers of elements.

Number of elements	Number of dofs	Error	Time (sec)
29	48	5.67891×10^{-4}	0.23
84	198	1.77260×10^{-4}	0.66
250	636	9.42737×10^{-5}	2.02
2034	5838	3.49306×10^{-5}	17.48
4833	14022	5.12128×10^{-6}	43.94

Table 2: L^2-norm error of the solution obtained from the Biharmonic implementations using the C1CurvedElement with various numbers of elements.

Number of elements	Number of dofs	Error	Time (sec)
29	63	1.41802×10^{-4}	5.73
84	228	3.70896×10^{-5}	16.07
250	696	9.52325×10^{-6}	54.92
2034	5958	1.0478×10^{-8}	436.17
4833	14262	3.6392×10^{-10}	1749.63

Tables 1 and 2 show that the computational time for the `C1CurvedElement` is obviously very expensive compared to that of the `BellElement` when the same number of element is concerned. However, for the same

number of elements, accuracies obtained from the curved elements are superior to those obtained from the straight elements.

Furthermore, if the same error is considered, say $O(10^{-5})$, it can be seen from Tables 1 and 2 that the number of the `BellElement` has to be 2034 to obtain such an accuracy. On the other hands, the number of the `C1Curved↩Element` needed to obtain such accuracy is 84 which is a lot less. Also, the computational time to obtained the accuracy of order $O(10^{-5})$ for the `BellElement` is 17.48 and for the `C1CurvedElement` is 16.07 which is smaller.

In order to compare the convergence rate between the `BellElement` and the `C1CurvedElement`, Fig. 1.5 illustrates the comparison between the L^2-norm error and the element size obtained from those elements. It can be seen that the `C1CurvedElement` converges faster and, also, gives smaller error than the `BellElement`.

When performing the log-plot between the L^2-norm error and the element size, Fig. 1.6 depicts that the convergence rate of the `C1CurvedElement` are greater than the `BellElement` as its slope is greater. The obtained rate of convergence for the `C1CurvedElement` and `the` BellElement is quintic and quadratic, respectively.

It is noteworthy that using the `BellElement` to solve the C^1-problem with a curved boundary domain decreases its rate of convergence from its potential when solving an exact representation domain (see another tutorial). This limitation of convergence rate in the `BellElement` is due to the representation of the curved boundaries with straight-edge elements.

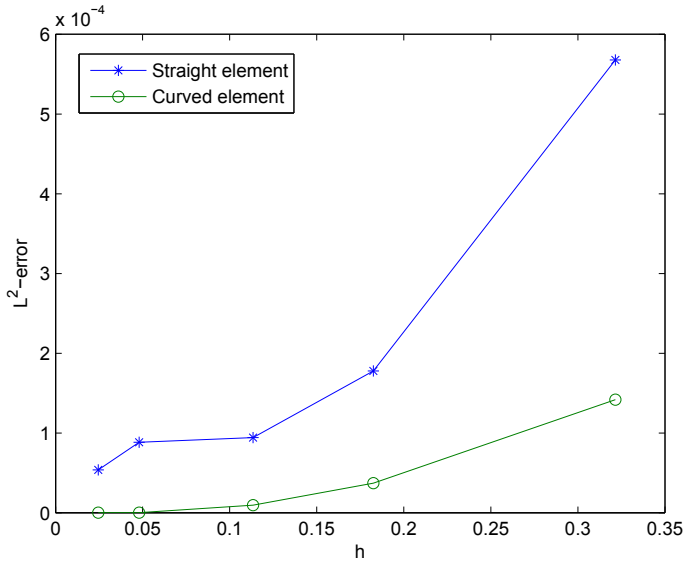

Figure 1.5 The comparison of the convergence rate between the Bell and the C1-curved triangular elements.

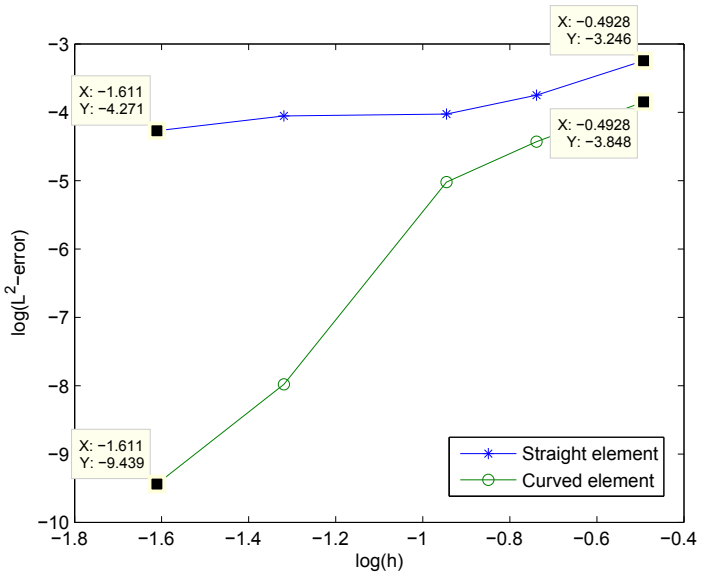

Figure 1.6 The comparison of the convergence rate between the Bell and the C1-curved triangular elements.

1.3 Implementation in oomph-lib

In order to solve the Biharmonic equation with `oomph-lib`, the Global parameters and functions can be defined the same as in `another Biharmonic tutorial`. However, we use the `BiharmonicCurved↩ Element<2,2>` to solve the Biharmonic equation when the curvilinear boundary is concerned. This can be modified in the driver code.

In sequels, we will discuss only functions that defined differently from solving the Biharmonic equation with the `BellElement`.

1.3.1 The problem class

The problem class has five member functions, illustrated as follows:

- The problem constructor
- `action_before_newton_solve()` : Update the problem specifications before solve. Boundary conditions maybe set here.
- `action_after_newton_solve()` : Update the problem specifications after solve.
- `doc_solution()` : Pass the number of the case considered, so that output files can be distinguished.
- `nodal_permutation()` : Perform a nodal permutation in a curved element.

```
//==start_of_problem_class============================================
/// 2D Biharmonic problem.
//====================================================================
template<class ELEMENT, unsigned DIM, unsigned NNODE_1D>
class MyBiharmonicProblem : public Problem
{
public:

 /// Constructor: Pass number of elements and pointer to source function
```

```
 MyBiharmonicProblem(typename MyBiharmonicEquations<DIM,NNODE_1D>::SourceFctPt
      source_fct_pt,
                          typename MyBiharmonicEquations<DIM,NNODE_1D>::ExactSolnPt exact_pt,
                          const string& node_file_name,
                          const string& element_file_name,
                          const string& poly_file_name);

 /// Destructor (empty)
 ~MyBiharmonicProblem()
  {
   delete mesh_pt();
  }


 /// Update the problem specs before solve: (Re)set boundary conditions
 void actions_before_newton_solve();

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

 /// \short Doc the solution, pass the number of the case considered,
 /// so that output files can be distinguished.
 void doc_solution(DocInfo& doc_info);

 /// Perform a nodal permutation in curved elements
 void nodal_permutation();

private:

 /// Pointer to source function
 typename MyBiharmonicEquations<DIM,NNODE_1D>::SourceFctPt Source_fct_pt;

 /// Pointer to the exact solution
 typename MyBiharmonicEquations<DIM,NNODE_1D>::ExactSolnPt Exact_soln_pt;

}; // end of problem class
```

Only function `Problem::nodal_permutation()` is newly defined to associate with a curvilinear boundary. This function is employed to swap nodes so that they are correspond with the defined shape functions when using the `C1CurvedElement`.

1.3.2 The nodal_permutation() function

Regarding the mapping associated between the reference coordinates and the physical coordinates defined on the curved triangle, it can be seen that we defined the side $\hat{\mathbf{a}}_1\hat{\mathbf{a}}_2$ of the reference to associate with the curved boundary. Also, in the definition defined in (2), the vertex node $\mathbf{a_1}$ was defined to be correspond with the position where s is minimum on each arc of the curved boundary Γ and the node $\mathbf{a_2}$ was correspond with the position where the maximum s is obtained. Furthermore, the node $\mathbf{a_3}$ was defined to be the off-boundary node of a curved triangle.

However, a discretisation of the curvilinear domain by triangles can give a triangulation where the node ordering in a triangular element incompatible with the described definitions. As illustrated in the following figure, three cases of node ordering can happen.

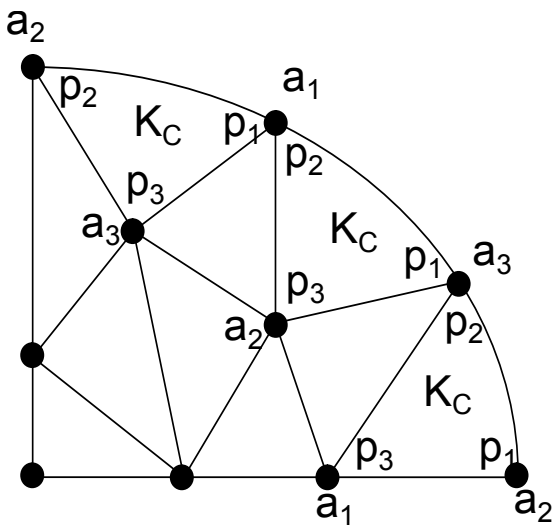

Figure 1.7 The graphical description of node ordering.

It can be seen from the definition in (2) and the figure that $F_K(\hat{\mathbf{a}}_\mathbf{i}) = \mathbf{p_i}$ rather than $\mathbf{a_i}$. Therefore, in order to derive our C^1-curved triangular shape functions, a re-position of nodes has to be performed. The function `nodal_permutation(...)` will provide a nodal permutation where the nodal pointers will be swapped so that they are associated with the shape functions defined at that node. The nodal pointers will be permuted to correspond with the conditions required in (2).

Here are codes for a nodal permutation.

```
//===start_of_nodal_permutation=============================================
/// \short Update the nodal permutation in curved elements: (Re)assign
/// coordinates so that the first and the second nodes are associated
/// with the lower and upper position on a boundary, respectively.
/// The third node is off-boundary node.
//==========================================================================
template<class ELEMENT, unsigned DIM, unsigned NNODE_1D>
 void MyBiharmonicProblem<ELEMENT,DIM,NNODE_1D>::nodal_permutation
      ()
  {
   DenseMatrix<double> position(3,2);
   DenseMatrix<double> bd_position(20,2);
   Vector<double> x(2);
   unsigned bd_element;

   // Get number of elements in the mesh
   unsigned n_element = mesh_pt()->nelement();

   // Start a classification of nodes
   // Loop over elements to get coordinates of nodes on a boundary
   for(unsigned n=0;n<n_element;n++)
    {
     // Upcast from GeneralisedElement to the present element
     ELEMENT *elem_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(n));
     // we need just three vertices
     unsigned n_node = 3;

     for(unsigned l=0;l<n_node;l++)
      {
       for(unsigned j=0;j<DIM;j++)
        {
         position(l,j) = elem_pt->node_pt(l)->x(j);
        }
      }

     int count=0;

     // check whether this element is on boundary or not
     for(unsigned i=0;i<n_node;i++)
      {
       // check whether this node is on 1st boundary (curve boundary)
       bool bd_node1 = elem_pt->node_pt(i)->is_on_boundary(1);
       if(bd_node1==1)
```

```
    {
     count += 1;
     for(unsigned j=0;j<DIM;j++)
      {
       bd_position(count-1,j) = elem_pt->node_pt(i)->x(j);
      }
    }
   else
    {
     for(unsigned j=0;j<DIM;j++)
      {
       x[j] = elem_pt->node_pt(i)->x(j);
      }
    }
  }

 if(count == 2) // this case is a boundary element
  {
   bd_element = 1;
  }
 else
  {
   bd_element = 0;
  } // end_of_classification

 /// Permutation just for the curved elements
 if(bd_element==1)
  {
   unsigned n_position_type = 6;
   DenseMatrix<double> nodal_value(3,n_position_type,0.0);
   double angle_1 = atan(bd_position(0,1)/bd_position(0,0));
   double angle_2 = atan(bd_position(1,1)/bd_position(1,0));

   if(angle_1 > angle_2)
    {
     if(position(2,0)==x[0] && position(2,1)==x[1])
      {
       Node* first_pt = elem_pt->node_pt(0);
       elem_pt->node_pt(0) = elem_pt->node_pt(1);
       elem_pt->node_pt(1) = first_pt;

       Node* second_pt = elem_pt->node_pt(3);
       elem_pt->node_pt(3) = elem_pt->node_pt(4);
       elem_pt->node_pt(5) = second_pt;
      }
     else if(position(1,0)==bd_position(0,0) && position(1,1)==bd_position(0,1))

      {
       Node* first_pt = elem_pt->node_pt(2);
       elem_pt->node_pt(0) = elem_pt->node_pt(2);
       elem_pt->node_pt(2) = first_pt;

       Node* second_pt = elem_pt->node_pt(3);
       elem_pt->node_pt(3) = elem_pt->node_pt(5);
       elem_pt->node_pt(5) = second_pt;
      }
     else
      {
       Node* first_pt = elem_pt->node_pt(0);
       elem_pt->node_pt(0) = elem_pt->node_pt(2);
       elem_pt->node_pt(2) = elem_pt->node_pt(1);
       elem_pt->node_pt(1) = first_pt;

       Node* second_pt = elem_pt->node_pt(3);
       elem_pt->node_pt(3) = elem_pt->node_pt(5);
       elem_pt->node_pt(5) = elem_pt->node_pt(4);
       elem_pt->node_pt(4) = second_pt;
      }
    }
   else if(angle_1 < angle_2)
    {
     if(position(0,0)==bd_position(0,0) && position(0,1)==bd_position(0,1))
      {
       if(position(2,0)==bd_position(1,0) && position(2,1)==bd_position(1,1))

        {
         Node* first_pt = elem_pt->node_pt(1);
         elem_pt->node_pt(1) = elem_pt->node_pt(2);
         elem_pt->node_pt(2) = first_pt;

         Node* second_pt = elem_pt->node_pt(4);
         elem_pt->node_pt(4) = elem_pt->node_pt(5);
         elem_pt->node_pt(5) = second_pt;
        }
      }
     else
      {
```

```
          Node* first_pt = elem_pt->node_pt(0);
          elem_pt->node_pt(0) = elem_pt->node_pt(1);
          elem_pt->node_pt(1) = elem_pt->node_pt(2);
          elem_pt->node_pt(2) = first_pt;

          Node* second_pt = elem_pt->node_pt(3);
          elem_pt->node_pt(3) = elem_pt->node_pt(4);
          elem_pt->node_pt(4) = elem_pt->node_pt(5);
          elem_pt->node_pt(5) = second_pt;
         }
        }
       }
     }
  }  // end of permutation
```

1.3.3 The Problem constructor

The problem constructor starts by overloading the function `Problem::mesh_pt()` and set to the specific mesh used in this problem. In this tutorial, we implement the problem with a 2D unstructured mesh which is externally created by `Triangle`. The generated output will be used to build the `oomph-lib` mesh. The reader may refer to another tutorial to create an unstructured triangular mesh internally.

```
//=====start_of_constructor=======================================================
/// \short Constructor for 2D Biharmonic problem.
/// Discretise the 2D domain with n_element elements of type ELEMENT.
/// Specify function pointer to source function.
//================================================================================
template<class ELEMENT, unsigned DIM, unsigned NNODE_1D>
MyBiharmonicProblem<ELEMENT,DIM,NNODE_1D>::MyBiharmonicProblem
(typename MyBiharmonicEquations<DIM,NNODE_1D>::SourceFctPt source_fct_pt,
 typename MyBiharmonicEquations<DIM,NNODE_1D>::ExactSolnPt exact_pt,
 const string& node_file_name,
 const string& element_file_name,
 const string& poly_file_name) :
 Source_fct_pt(source_fct_pt), Exact_soln_pt(exact_pt)
{
 Problem::mesh_pt() = new TriangleMesh<ELEMENT>(node_file_name,element_file_name,poly_file_name);
```

Next, the boundary conditions of the problem will be taken care. Since the Dirichlet boundary conditions are applied in this problem, we pin the nodal values and their first-order derivatives on all boundaries. In this study, we also set degrees of freedom associated with the second-order derivatives to be pinned in order to reduce the number of degrees of freedom in the problem.

Note that there are three boundaries on the quarter of the unit circular domain. The boundary identities are labelled anticlockwise where the first boundary is associated with the side $(x, y) = (0, y)$.

```
 // start_of_boundary_conditions
 // assign conditions at boundary 0
 unsigned i=0;
 unsigned  n_node1 = mesh_pt()->nboundary_node(i);
 cout  << "nnode on b0 = " << n_node1 << endl;

 for (unsigned n=0;n<n_node1;n++)
  {
   mesh_pt()->boundary_node_pt(i,n)->pin(0);
   mesh_pt()->boundary_node_pt(i,n)->pin(1);
   mesh_pt()->boundary_node_pt(i,n)->pin(2);
   mesh_pt()->boundary_node_pt(i,n)->pin(3);
   mesh_pt()->boundary_node_pt(i,n)->pin(4);
   mesh_pt()->boundary_node_pt(i,n)->pin(5);
  }
 // assign conditions at boundary 1
 i=1;
 n_node1 = mesh_pt()->nboundary_node(i);
 cout  << "nnode on b1 = " << n_node1 << endl;
 for (unsigned n=0;n<n_node1;n++)
  {
   mesh_pt()->boundary_node_pt(i,n)->pin(0);
   mesh_pt()->boundary_node_pt(i,n)->pin(1);
   mesh_pt()->boundary_node_pt(i,n)->pin(2);
```

```
   mesh_pt()->boundary_node_pt(i,n)->pin(3);
   mesh_pt()->boundary_node_pt(i,n)->pin(4);
   mesh_pt()->boundary_node_pt(i,n)->pin(5);
  }
 // assign conditions at boundary 2
 i=2;
 n_node1 = mesh_pt()->nboundary_node(2);
 cout  << "nnode on b2 = " << n_node1 << endl;
 for (unsigned n=0;n<n_node1;n++)
  {
   mesh_pt()->boundary_node_pt(i,n)->pin(0);
   mesh_pt()->boundary_node_pt(i,n)->pin(1);
   mesh_pt()->boundary_node_pt(i,n)->pin(2);
   mesh_pt()->boundary_node_pt(i,n)->pin(3);
   mesh_pt()->boundary_node_pt(i,n)->pin(4);
   mesh_pt()->boundary_node_pt(i,n)->pin(5);
  }

 // end of boundary conditions
```

According to the `C1CurvedElement`, there are two different types of triangular elements which contain different numbers of nodes in an element. The straight-sided triangular elements constitute of three vertex nodes with out any interior node while the curved elements constitute of three vertices and three interior nodes. Since there are overall six nodes in `C1CurvedElement`, all three interior nodes have to be pinned when the straight-sided elements are concerned.

```
   // pinned the unnecessary nodes in the straight-sided elements
```

We then loop over the elements and set the pointers to the physical parameters, the function pointer to the source function which is the function on the right-hand side of the Biharmonic equation, and the pointer to the exact solutions.

```
 // Loop over elements and set pointers to Physical parameters
 for(unsigned i=0;i<n_element;i++)
  {
   // Upcast from GeneralisedElement to the present element
   ELEMENT *elem_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(i));

   //Set the source function pointer and all physical variables
   elem_pt->source_fct_pt() = Source_fct_pt;

   //Set the exact solution pointer
   elem_pt->exact_pt() = Exact_soln_pt;
  } // end of pointers set up
```

Before we can implement a problem in a curved element, a nodal permutation is needed in order to satisfy (2). That is the coordinates of the first and the second nodes have to be associated with the lower and upper position on a boundary, respectively. Furthermore, the third node should be off-boundary.

```
 /// Perform a nodal permutation in curved elements
 void nodal_permutation();
```

We finish the constructor by assigning the equation numbering scheme.

```
 // Setup equation numbering scheme
 assign_eqn_numbers();
} // end of constructor
```

1.4 Source files for this tutorial

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

demo_drivers/biharmonic/unstructured_2d_biharmonic/

- The driver code for the 2D Biharmonic problem with the `BellElement` is:

demo_drivers/biharmonic/unstructured_2d_biharmonic/unstructured_2d_biharmonic_bellelement.cc

- The driver code for the 2D Biharmonic problem with the `C1CurvedElement` is:

demo_drivers/biharmonic/unstructured_2d_biharmonic/unstructured_2d_biharmonic_curvedelement.cc

1.5 PDF file

A pdf version of this document is available.