# Chapter 1

# The Finite Element Method

## 1.1 Introduction

This document provides a brief introduction to the finite element method and illustrates how the method is implemented in `oomph-lib`. The first few sections present a brief, self-contained derivation of the method. The exposition is "constructive" and avoids a number of subtleties, which can be found in standard textbooks [e.g. E.B. Becker, G.F. Carey, and J.T. Oden. Finite Elements: An Introduction. Prentice-Hall, Englewood Cliffs, New Jersey, (1981)] . We generally assume that all functions are sufficiently "well behaved" so that all mathematical manipulations "make sense". Function spaces are only used where they help to make the notation more compact.

Readers who are familiar with the theory of finite elements may skip the introductory sections, but should consult the section An object-oriented implementation which explains the particular implementation used in `oomph-lib`.

Initially, we develop the method for scalar second-order (elliptic) PDEs with Dirichlet boundary conditions, using classical 1D and 2D Poisson problems as model problems. Specifically, we consider the 1D problem

$$\frac{\mathrm{d}^2 u(x)}{\mathrm{d}x^2} = f(x) \quad \text{for } x \in [0,1] \quad \text{subject to} \quad u(x=0) = g_0 \quad \text{and} \quad u(x=1) = g_1,$$

where $f(x)$ and the constants $g_0$ and $g_1$ are given. The 2D equivalent is given by

$$\frac{\partial^2 u(x_1,x_2)}{\partial x_1^2} + \frac{\partial^2 u(x_1,x_2)}{\partial x_2^2} = f(x_1,x_2) \quad \text{for } (x_1,x_2) \in D \quad \text{subject to} \quad u|_{\partial D} = g,$$

where $f(x_1,x_2)$ and $g(x_1,x_2)$ are given.

**A note on notation**

Throughout this document, we use index notation and write, e.g., $u(x_1,x_2)$ as $u(x_i)$. We do not explicitly state the range of free indices where it is clear from the context – in the above example it would be equal to the spatial dimension of the problem.
All mathematical derivations are presented using the "natural" numbering of the indices: e.g., the components of the 3D vector $\mathbf{f}$ are $f_1, f_2$ and $f_3$. Unfortunately, this notation is inconsistent with the implementation of vectors (and most other standard "containers") in C++ where the indices start from 0 and the components of `vector<double> f(3)` are `f`[0], `f`[1] and `f`[2]. There is no elegant way to resolve this conflict. Adopting C++-style numbering in the theoretical development would make the mathematics look very odd (try it!); conversely, adopting the "natural" numbering when discussing the C++-implementation would make this documentation inconsistent with the actual implementation. We therefore use both numbering systems, each within their appropriate context.

Physically, the Poisson equation describes steady diffusion processes. For instance, the 2D Poisson problem describes the temperature distribution $u(x_i)$ within a 2D body, $D$, whose boundary $\partial D$ is maintained at a prescribed temperature, $g(x_i)$. The function $f(x_i)$ describes the strength of (distributed) heat sources in the body. In practical applications, the strength of these heat sources is bounded because no physical process can release infinite amounts of energy in a finite domain. Hence, we assume that

$$\int_D f(x_i)\, dx_1 dx_2 < \infty.$$

We usually write all PDEs in "residual form", obtained by moving all terms to the left-hand-side of the equation, so that the general second-order scalar PDE problem is

**Problem P**

$$\mathcal{R}\left(x_i; u(x_i), \frac{\partial u}{\partial x_i}, \frac{\partial^2 u}{\partial x_i\, \partial x_j}\right) = 0 \quad \text{in} \quad D,$$

with Dirichlet (essential) boundary conditions on $\partial D$

$$u|_{\partial D} = g,$$

where the function $g$ is given

To keep the notation compact, we suppress the explicit dependence of $\mathcal{R}$ on the derivatives and write the residual as
$\mathcal{R}(x_i; u(x_i))$. For example, the residual forms of the two Poisson problems are given by:

**Problem P1**

$$\mathcal{R}(x; u(x)) = \frac{\mathrm{d}^2 u(x)}{\mathrm{d}x^2} - f(x) = 0 \quad \text{for } x \in [0,1] \quad \text{subject to} \quad u(x=0) = g_0 \quad \text{and} \quad u(x=1) = g_1,$$

where $f(x)$ and the constants $g_0$ and $g_1$ are given.

and

**Problem P2**

$$\mathcal{R}(x_i; u(x_i)) = \sum_{j=1}^{2} \frac{\partial^2 u(x_i)}{\partial x_j^2} - f(x_i) = 0 \quad \text{for } x_i \in D \quad \text{subject to} \quad u|_{\partial D} = g,$$

where $f(x_i)$ and $g(x_i)$ are given, and $i = 1, 2$.

We stress that neither the finite element method, nor `oomph-lib` are restricted to scalar second-order PDEs. Documentation for the example drivers discusses generalisations to:

- non-Dirichlet boundary conditions
- systems of PDEs
- mixed interpolation
- discontinuous interpolation
- timestepping
- higher-order PDEs
- solid mechanics and Lagrangian coordinates.

## 1.2 Mathematical background

### 1.2.1 The weak solution

A classical (or strong) solution of the problem P is any function $u(x_i)$ that satisfies the PDE and boundary condition at every point in $D$,

$$\mathcal{R}(x_i; u(x_i)) \equiv 0 \quad \forall x_i \in D \quad \text{and} \quad u|_{\partial D} = g.$$

The concept of a "weak" solution is based on a slight relaxation of this criterion. A weak solution, $u_w(x_i)$, of problem P is any function that satisfies the essential boundary condition,

$$u_w|_{\partial D} = g,$$

and for which the so-called "weighted residual"

$$r = \int_D \mathcal{R}(x_i; u_w(x_i))\ \phi^{(test)}(x_i)\ \mathrm{d}x_1 \mathrm{d}x_2 \qquad (1)$$

vanishes for *any* "test function" $\phi^{(test)}(x_i)$ which satisfies homogeneous boundary conditions so that

$$\phi^{(test)}|_{\partial D} = 0.$$

At this point it might appear that we have fatally weakened the concept of a solution. If we only require the PDE to be satisfied in an average sense, couldn't any function
be a "solution"? In fact, this is not the case and we shall now demonstrate that, for all practical purposes [we refer to the standard literature for a rigorous derivation], the statement

"weak solutions are strong solutions"

is true. The crucial observation is that the weak solution requires the weighted residual to vanish for *any* test function. To show that this is equivalent to demanding that $\mathcal{R} \equiv 0 \ \ \forall x_i \in D$ (as in the definition of the strong solution), let us try to construct a counter-example for which $\mathcal{R} \not\equiv 0$ in some part of the domain (implying that the candidate solution is not a classical solution) while $r = 0$ (so that it qualifies as a weak solution). For simplicity we illustrate the impossibility of this in a 1D example. First consider a candidate solution $u_c(x)$ which satisfies the essential boundary condition but does not satisfy the PDE anywhere, and that $\mathcal{R}(x; u_c(x)) > 0$ throughout the domain, as indicated in this sketch:

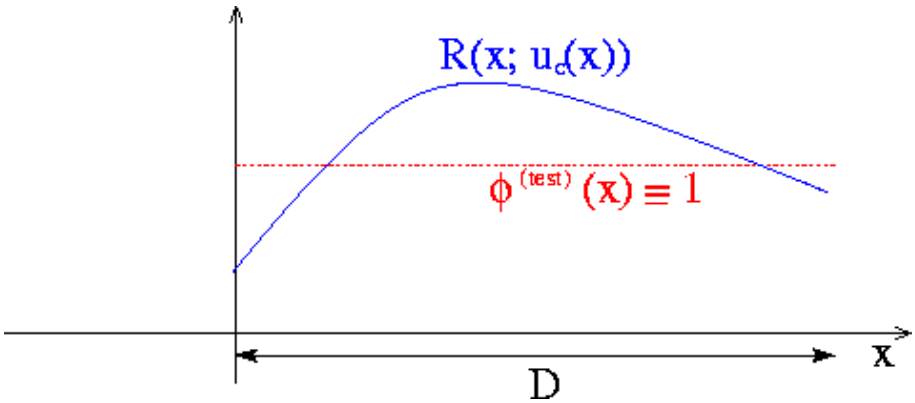


**Figure 1.1 Residual (blue/solid) is nonzero (and positive) throughout the domain. A constant test function (red/dotted) is sufficient to show that the candidate solution is not a weak solution.**

Could this candidate solution possibly qualify as a weak solution? No, using the trivial test function $\phi^{(test)} \equiv 1$ gives a nonzero weighted residual, and it follows that $\mathcal{R}(x, u_c(x))$ must have zero average if $u_c(x)$ is to qualify as a weak solution.
The figure below shows the residual for a more sophisticated candidate solution which satisfies the PDE over most of the domain.

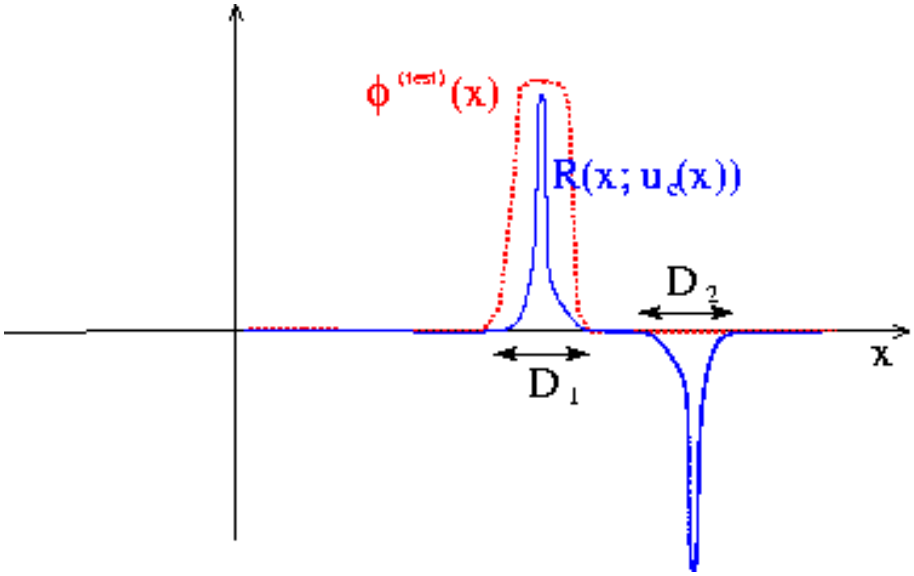


**Figure 1.2 Residual (blue/solid) is nonzero only in two small sub-domains. A suitably constructed test function with finite support (red/dotted) is sufficient to show that the candidate solution is not a weak solution.**

The residual is nonzero only in two small subdomains, $D_1$ and $D_2$. The candidate solution is such that the residual has different signs in $D_1$ and $D_2$ so that its average over the domain is zero. Could this solution qualify as a weak solution? Again the answer is no because we can choose a test function that is nonzero in only one of the two subdomains (e.g. the function shown by the red/dotted line), which gives a nonzero weighted residual.
It is clear that such a procedure may be used to obtain a nonzero weighted residual whenever the residual $\mathcal{R}$ is nonzero *anywhere* in the domain. In other words, a weak solution is a strong solution, as claimed. [To make this argument mathematically rigorous, we would have to re-assess the argument for (pathological) cases in which the residual is nonzero only at finite number of points, etc.].

### 1.2.2 A useful trick: Integration by parts

Consider now the weak form of the 2D Poisson problem P2,

$$\int_D \left( \sum_{j=1}^{2} \frac{\partial^2 u(x_i)}{\partial x_j^2} - f(x_i) \right) \phi^{(test)}(x_i)\, \mathrm{d}x_1 \mathrm{d}x_2 = 0 \quad \text{subject to} \quad u|_{\partial D} = g. \qquad (2)$$

After integration by parts and use of the divergence theorem, we obtain

$$\int_D \sum_{j=1}^{2} \frac{\partial u(x_i)}{\partial x_j} \frac{\partial \phi^{(test)}(x_i)}{\partial x_j}\, \mathrm{d}x_1 \mathrm{d}x_2 + \int_D f(x_i)\, \phi^{(test)}(x_i)\, \mathrm{d}x_1 \mathrm{d}x_2 = \oint_{\partial D} \frac{\partial u}{\partial n}\, \phi^{(test)}\, \mathrm{d}s, \qquad (3)$$

where $s$ is the arclength along the domain boundary $\partial D$ and $\partial/\partial n$ the outward normal derivative. Since the test functions satisfy homogeneous boundary conditions, $\phi^{(test)}\big|_{\partial D} = 0$, the line integral on the RHS of equation (3) vanishes. Therefore, an alternative version of the weak form of problem P2 is given by

$$\int_D \sum_{j=1}^{2} \frac{\partial u(x_i)}{\partial x_j} \frac{\partial \phi^{(test)}(x_i)}{\partial x_j}\, \mathrm{d}x_1 \mathrm{d}x_2 + \int_D f(x_i)\, \phi^{(test)}(x_i)\, \mathrm{d}x_1 \mathrm{d}x_2 = 0. \qquad (4)$$

We note that (4) involves first derivatives of the unknown function $u$ and the test function $\phi^{(test)}$, whereas (2) involves second derivatives of $u$ and the zero-th derivatives of $\phi^{(test)}$. The advantages of using the "symmetric", integrated-by-parts version of the weak form will become apparent in the subsequent sections.

### 1.2.3 [Nearly an aside:] Function spaces

We stated in the introduction that all functions are assumed to be *sufficiently "well-behaved" so that all mathematical manipulations "make sense"*. It is important to realise that we have already (tacitly) used this assumption in the derivation of the weak form. The weak form does not "make sense" if we allow candidate solutions and test functions for which the integral in (1) does not exist. This imposes restrictions on the types of functions that are "admissible" in our problem. The precise form of these restrictions depends on the form of the residual $\mathcal{R}(x_i, u(x_i))$, and, typically, the restrictions are related to the functions' differentiability. It is convenient to employ the concept of a "function space" to collectively refer to all functions that satisfy the required restrictions. [In this introduction, none of the additional (heavy) machinery from functional analysis is required].
For instance, we can ensure that the integrated-by-parts version of problem P2 in equation (4) "makes sense" if we restrict $u(x_i)$ and $\phi^{(test)}(x_i)$ to all functions whose zeroth and first derivatives are square integrable over the domain $D$. These functions are members of a (well-known) function space that is usually denoted by $H^1(D)$. In fact, $H^1(D)$ is a particular instance of a family of function spaces – the Sobolev spaces $H^i(D)$ where $i = 0, 1, 2, ...$ which contain all functions whose zeroth, first, ..., i-th derivatives are square-integrable over the domain $D$. The members of these function spaces have the property that

$$u(x_i) \in H^0(D) \quad \Longleftrightarrow \quad \int_D u^2(x_i)\, dx_1 dx_2 < \infty,$$

$$u(x_i) \in H^1(D) \quad \Longleftrightarrow \quad \int_D \left( u^2(x_i) + \sum_{j=1}^{2} \left( \frac{\partial u(x_i)}{\partial x_j} \right)^2 \right) dx_1 dx_2 < \infty,$$

etc. We use the subscript "0" to restrict a given function space to the subset of its members which vanish on the domain boundary $\partial D$,

$$u(x_i) \in H_0^i(D) \quad \Longleftrightarrow \quad u(x_i) \in H^i(D) \;\text{ and }\; u|\partial D = 0.$$

Using these function spaces, we can provide a concise definition of the weak form of problem P2:

**Problem P2$_{\text{weak}}$**

Find the function $u(x_i) \in H^1(D)$ that satisfies the essential boundary conditions

$$u|_{\partial D} = g,$$

and for which

$$\int_D \sum_{j=1}^{2} \frac{\partial u(x_i)}{\partial x_j} \frac{\partial \phi^{(test)}(x_i)}{\partial x_j} \, \mathrm{d}x_1 \mathrm{d}x_2 + \int_D f(x_i)\, \phi^{(test)}(x_i) \, \mathrm{d}x_1 \mathrm{d}x_2 = 0$$

for *all* test functions $\phi^{(test)}(x_i) \in H_0^1(D)$.

It is important to realise that the choice of suitable function spaces for $u(x_i)$ and $\phi^{(test)}(x_i)$ is problem-dependent, guided by the inspection of the weak form for a specific problem. The (pragmatic) procedure is straightforward: write down the weak form and determine the (minimal) constraints that must be imposed on $u(x_i)$ and $\phi(x_i)$ for the weak form to "make sense". All functions that satisfy these constraints, are "admissible" and, collectively, they form a function space $H(D)$, say. The weak form of the general problem P can then be written as

**Problem $P_{weak}$**

Find the function $u(x_i) \in H(D)$ that satisfies the essential boundary conditions

$$u|_{\partial D} = g,$$

and for which

$$\int_D \mathcal{R}(x_i, u(x_i))\ \phi^{(test)}(x_i)\ \mathrm{d}x_1 \mathrm{d}x_2 = 0$$

for *all* test functions $\phi^{(test)}(x_i) \in H_0(D)$.

[If you followed the above argument carefully you will have realised that our strategy for ensuring that the weak form "makes sense" uses a sufficient rather than a necessary condition. For instance, it is not necessary for $u(x_i)$ and $\phi^{(test)}(x_i)$ to be members of the same function space. Alternative formulations are possible but we shall not pursue such ideas any further in this introduction.]

### 1.2.4 The Galerkin method

We now exploit the definition of the weak solution to develop a numerical method that can be used to determine approximate solutions to problem $P_{weak}$. We start by splitting the solution into two parts,

$$u(x_i) = u_h(x_i) + u_p(x_i),$$

where $u_p(x_i)$ is an (arbitrary) function that satisfies the Dirichlet boundary conditions,

$$u_p|_{\partial D} = g.$$

The unknown function $u_h(x_i)$ then has to satisfy the homogeneous boundary conditions

$$u_h|_{\partial D} = 0.$$

We expand $u_h(x_i)$ in terms of a (given) infinite set of basis functions $\psi_j(x_i) \in H_0(D) \quad (j = 1, ..., \infty)$,

$$u(x_i) = u_p(x_i) + \sum_{j=1}^{\infty} U_j \psi_j(x_i), \qquad (5)$$

which discretises the problem because the solution is now determined by the (as yet unknown) discrete coefficients $U_j \quad (j = 1, ..., \infty)$. There are many possible sets of basis functions: polynomials, trigonometric functions, systems of eigenfunctions; mathematically speaking, the only requirement is that the basis functions are sufficiently general that the solution can be represented by the expansion (5). In other words, the functions must be a complete basis for $H_0(D)$.
How do we determine the discrete coefficients $U_j$? Inserting the expansion for $u(x)$ into the definition of the weighted residual yields

$$r = \int_D \mathcal{R}\left(x_i;\ u_p(x_i) + \sum_{j=1}^{\infty} U_j \psi_j(x_i)\right)\ \phi^{(test)}(x_i)\ \mathrm{d}x_1 \mathrm{d}x_2 = 0, \qquad (6)$$

and we recall that this equation must be satisfied for *any* test function $\phi^{(test)} \in H_0(D)$. The functions $\psi_j(x_i)$ form a complete basis for $H_0(D)$, and so all possible test functions $\phi^{(test)}(x_i)$ may be represented as

$$\phi^{(test)}(x_i) = \sum_{k=1}^{\infty} \Phi_k \psi_k(x_i). \qquad (7)$$

Thus, the condition

...for any basis function $\phi^{(test)}(x_i)$...

becomes

...for any values of the coefficients $\Phi_k$...

Inserting the expansion (7) into the definition of the weak solution (6) yields

$$r = \sum_{k=1}^{\infty} \Phi_k \; r_k(U_1, U_2, ...) = 0, \quad (8)$$

where

$$r_k(U_1, U_2, ...) = \int_D \mathcal{R} \left( x_i; \; u_p(x_i) + \sum_{j=1}^{\infty} U_j \psi_j(x_i) \right) \; \psi_k(x_i) \; \mathrm{d}x_1 \mathrm{d}x_2. \quad (9)$$

Equation (8) must hold for *any* value of the coefficients $\Phi_k$, so the coefficients $U_j$ must satisfy the equations

$$r_k(U_1, U_2, ...) = 0, \quad \text{for } k = 1, 2, ...$$

In practice, we truncate the expansions (5) and (7) after a finite number of terms to obtain the approximations (indicated by tildes)

$$\widetilde{u}(x_i) = u_p(x_i) + \sum_{j=1}^{M} U_j \psi_j(x_i) \quad \text{and} \quad \widetilde{\phi^{(test)}}(x_i) = \sum_{k=1}^{M} \Phi_k \psi_k(x_i), \quad (10)$$

and we determine the $M$ unknown coefficients, $U_1, ..., U_M$, from the $M$ algebraic equations

$$r_k(U_1, ..., U_M) = 0, \quad \text{where} \quad k = 1, ..., M. \quad (11)$$

The number of terms in each truncated expansion must be the same, so that we obtain $M$ equations for $M$ unknowns.
The truncation of the expansions (5) and (7) introduces two approximations:

- The approximate solution $\widetilde{u}(x_i)$ is a member of the finite-dimensional function space $\widetilde{H}(D) \subset H(D)$ spanned by the basis functions included in the expansion (10).
- We "test" the solution with functions from $\widetilde{H_0}(D)$ rather than with "all" functions $\phi^{(test)} \in H_0(D)$.

$\widetilde{H}(D) \to H(D)$ as $M \to \infty$, however, so the approximate solution $\widetilde{u}(x_i)$ converges to the exact solution $u(x_i)$ as we include more and more terms in the expansion. [The precise definition of "convergence" requires the introduction of a norm, which allows us to measure the "difference" between two functions. We refer to the standard literature for a more detailed discussion of this issue.]
In general, the equations $r_k(U_1, ..., U_M)$ are nonlinear and must be solved by an iterative method such as Newton's method. Consult your favourite numerical analysis textbook (if you can't think of one, have a look through chapter 9 in Press, W. H.; Flannery, B. P.; Teukolsky, S. A.; and Vetterling, W. T. "Numerical Recipes in C++. The Art of Scientific Computing", Cambridge University Press) for a reminder of how (and why) Newton's method works. The following algorithm shows the method applied to our equations:

**Algorithm 1: Newton's method**

1. Set the iteration counter $i = 0$ and provide an initial approximation for the unknowns, $U_j^{(0)} \;\; (j = 1, .., M)$.
2. Evaluate the residuals
$$r_k^{(i)} = r_k \left( U_1^{(i)}, ..., U_M^{(i)} \right) \quad \text{for } k = 1, ..., M.$$
3. Compute a suitable norm of the residual vector (e.g. the maximum norm). If the norm is less than some pre-assigned tolerance, stop and and accept $U_j^{(i)} (j = 1, ..., M)$ as the solution.
4. Compute the Jacobian matrix
$$J_{kj} = \left. \frac{\partial r_k}{\partial U_j} \right|_{\left( U_1^{(i)}, ..., U_M^{(i)} \right)} \quad \text{for } j, k = 1, ..., M.$$

5. Solve the linear system

$$\sum_{j=1}^{M} J_{kj}\ \delta U_j = -r_k^{(i)} \qquad \text{where } k = 1, ..., M$$

for $\delta U_j$ (j=1,...,M).

6. Compute an improved approximation via

$$U_j^{(i+1)} = U_j^{(i)} + \delta U_j \quad \text{for } j = 1, ..., M.$$

7. Set $i = i + 1$ and go to 2.

For a "good" initial approximation, $U_j^{(0)}$, Newton's method converges quadratically towards the exact solution. Furthermore, for linear problems, Newton's method provides the exact solution (modulo any roundoff errors that might be introduced during the solution of the linear system) in a single iteration. Newton's method can, therefore, be used as a robust, general-purpose solver, if (!) a good initial guess for the solution can be provided. In practice, this is not a serious restriction, because good initial guesses can often be generated by continuation methods. In `oomph-lib`, Newton's method is the default nonlinear solver.
Let us, briefly, examine the cost of the non-trivial steps involved in Newton's method:

- Step 2 requires the evaluation of $M$ integrals over the domain to determine the discrete residuals $r_k^{(i)}$ from (9). (We note that, in general, the integrals must be evaluated numerically.)
- Step 3 requires the computation of $M^2$ entries in the Jacobian matrix, each an integral of the form

$$J_{kj} = \int_D \frac{\partial}{\partial U_j} \mathcal{R}\left( x_i;\ \sum_{j=1}^{M} U_j \psi_j(x_i) \right)\ \psi_k(x_i)\ \mathrm{d}x_1 \mathrm{d}x_2 \quad \text{for } j, k = 1, ..., M.$$

- Step 4 requires the solution of a $M \times M$ linear system.

In general, steps 3 and 4 will be very costly if $M$ is large. However, if the domain has a simple shape and the differential operator has a sufficiently simple structure, it is often possible to choose basis functions with suitable orthogonality properties that render the Jacobian matrix $J_{kj}$ sparse. As an example, we consider the application of Galerkin's method in the 1D Poisson problem P1:

**Example: Galerkin method applied to the model problem P1**

We perform the usual integration by parts to derive the symmetric weak form of the problem:

**Problem P1$_{\text{weak}}$**

Find the function $u(x) \in H(D)$ that satisfies the essential boundary conditions

$$u(0) = g_0 \quad \text{and} \quad u(1) = g_1,$$

and for which

$$\int_0^1 \left( \frac{\mathrm{d}u(x)}{\mathrm{d}x}\ \frac{\mathrm{d}\phi^{(test)}(x)}{\mathrm{d}x}\ + f(x)\ \phi^{(test)}(x) \right)\ \mathrm{d}x = 0$$

for *all* test functions $\phi^{(test)}(x) \in H_0(D)$.

Inspection of the weak form shows that the choice $H(D) = H^1(D)$ is sufficient to ensure the existence of the integral. Of course, the "higher" Sobolev spaces $H^2(D), H^3(D), ...$ would also ensure the existence of the integral but would impose unnecessary additional restrictions on our functions.
Next, we need to construct a function $u_p(x)$ that satisfies the Dirichlet boundary conditions. In 1D this is trivial, and the simplest option is the function $u_p(x) = g_0 + (g_1 - g_0)x$, which interpolates linearly between the two boundary values. Since $\mathsf{d}u_p(x)/\mathsf{d}x = g_1 - g_0$, the discrete residuals are given by

$$r_k(U_1, U_2, ..., U_M) = \int_0^1 \left[ \left( (g_1 - g_0) + \sum_{j=1}^{M} U_j \frac{\mathsf{d}\psi_j(x)}{\mathsf{d}x} \right) \frac{\mathsf{d}\psi_k(x)}{\mathsf{d}x} + f(x)\, \psi_k(x) \right] \mathsf{d}x \quad \text{for } k = 1, .., M, \qquad (12)$$

and the Jacobian matrix has the form

$$J_{kj} = \int_0^1 \frac{\mathsf{d}\psi_j(x)}{\mathsf{d}x} \frac{\mathsf{d}\psi_k(x)}{\mathsf{d}x} \mathsf{d}x \quad \text{for } j, k = 1, .., M. \qquad (13)$$

The (Fourier) basis functions

$$\psi_j(x) = \sin(\pi j x)$$

are a suitable basis because

- they satisfy the homogeneous boundary conditions,
- they and their derivatives are square integrable, allowing all integrals to be evaluated, and
- they are a complete basis for $H_0^1(D)$.

Furthermore, the orthogonality relation

$$\int_0^1 \cos(\pi k x) \cos(\pi j x)\, dx = 0 \quad \text{for } j \neq k$$

implies that the Jacobian matrix is a diagonal matrix, which is cheap to assemble and invert. Indeed, the assembly of the Jacobian matrix in step 4, and the solution of the linear system in step 5 have an "optimal" computational complexity: their cost increases linearly with the number of unknowns in the problem.
Unfortunately, the application of the method becomes difficult, if not impossible, in cases where the differential operators have a more complicated structure, and/or the domain has a more complicated shape. The task of finding a complete set of basis functions that vanish on the domain boundary in an arbitrarily-shaped, higher-dimensional domain is nontrivial. Furthermore, for a complicated differential operator, it will be extremely difficult to find a system of basis functions for which the Jacobian matrix has a sparse structure. If the matrix is dense, the assembly and solution of the linear system in steps 4 and 5 of Newton's method can become prohibitively expensive.

**An aside: Integration by parts revisited**

Let us briefly return to the two versions of the weak form and examine the equations that we would have obtained had we applied Galerkin's method to the original form of the weak equations,

$$\int_0^1 \left( \frac{\mathsf{d}^2 u(x)}{\mathsf{d}x^2} - f(x) \right) \phi^{(test)}(x) \, \mathsf{d}x = 0.$$

The discrete residuals are then given by

$$r_k = \int_0^1 \left( \sum_{j=1}^{M} U_j \frac{\mathsf{d}^2 \psi_j(x)}{\mathsf{d}x^2} - f(x) \right) \psi_k(x) \, \mathsf{d}x = 0, \qquad \text{for } k = 1, ..., M. \qquad (14)$$

and the Jacobian matrix has the form

$$J_{kj} = \int_0^1 \frac{\mathsf{d}^2 \psi_j(x)}{\mathsf{d}x^2} \, \psi_k(x) \, \mathsf{d}x. \qquad (15)$$

- The restrictions that must be imposed on the basis functions, $\psi_j(x)$, if (14) is to "make sense" are much more awkward than for the symmetric form for the problem; the product of their zeroth and second derivatives must be integrable over the domain.
- The Jacobian matrix (13) that arises from the symmetric form of the weak problem is symmetric for *any* choice of basis functions whereas the Jacobian matrix (15) that arises from the original form is symmetric only for certain types of basis functions. This is not only advantageous for the solution of the linear system (only a fraction of the entries in the matrix need to be computed, and (more efficient) linear solvers that exploit the symmetry of the matrix can be used), but also properly reflects the symmetry in the (self-adjoint!) ODE.

## 1.3 The Finite Element Method

Galerkin's method is an efficient method for finding the approximate solution to a given problem if (and only if) we can:

1. Construct a function $u_p(x_i) \in H(D)$ that satisfies the essential boundary conditions.
2. Specify a set of basis functions that
   (a) spans the function space $H_0(D)$,
   (b) vanishes on the domain boundary, and
   (c) leads to a sparse Jacobian matrix.

.

We shall now develop the finite element method: an implementation of Galerkin's method that automatically satisfies all the above requirements.

### 1.3.1 Finite Element shape functions

The key feature of the finite element method is that the basis functions have finite support, being zero over most of the domain, and have the same functional form. We illustrate the idea and its implementation for the 1D Poisson problem P1 in its symmetric (integrated-by-parts) form:

$$r_k = \int_0^1 \left\{ \left( \frac{\mathsf{d}u_p(x)}{\mathsf{d}x} + \sum_{j=1}^{M} U_j \frac{\mathsf{d}\psi_j(x)}{\mathsf{d}x} \right) \frac{\mathsf{d}\psi_k(x)}{\mathsf{d}x} + f(x) \, \psi_k(x) \right\} \mathsf{d}x = 0, \qquad \text{for } k = 1, ..., M. \qquad (16)$$

The integral (16) exists for all basis functions $\psi_j(x)$ whose first derivatives are square integrable; a class of functions that includes piecewise linear functions.

We shall now construct a particular set of piecewise linear basis functions — the (global) linear finite-element shape functions, often known as "hat functions". For this purpose, we introduce $N$ equally-spaced "nodes" into the domain $x \in [0,1]$; node $j$ is located at $X_j = (j-1)h$, where $h = 1/(N-1)$ is the distance between the nodes. The (global) linear finite-element shape functions are defined by

$$\psi_j(x) = \begin{cases} 0 & \text{for } x < X_{j-1} \\ \frac{x - X_{j-1}}{X_j - X_{j-1}} & \text{for } X_{j-1} < x < X_j \\ \frac{X_{j+1} - x}{X_{j+1} - X_j} & \text{for } X_j < x < X_{j+1} \\ 0 & \text{for } x > X_{j+1} \end{cases} \tag{17}$$

and are illustrated below:

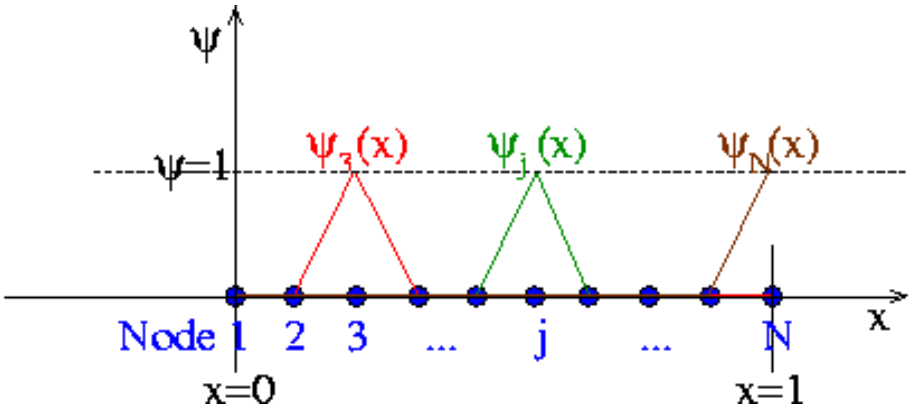


**Figure 1.3 The (global) linear finite-element shape functions in 1D.**

The finite-element shape functions have finite support; in particular, the function $\psi_j(x)$ is nonzero only in the vicinity of node j and varies linearly between one (at node j) and zero (at nodes $j-1$ and $j+1$). Furthermore, the shape functions satisfy the "interpolation condition"

$$\psi_j(X_i) = \delta_{ij} = \begin{cases} 1, \text{ if } i = j, \\ 0, \text{ if } i \neq j, \end{cases}$$

where $\delta_{ij}$ is the Kronecker delta.
The coefficients $V_j$ in an expansion of the form

$$\tilde{v}(x) = \sum_{j=1}^{N} V_j \, \psi_j(x)$$

have a straightforward interpretation: $V_j$ is the value of the function $v(x)$ at node $j$. The global shape functions vary linearly between the nodes, and so $\tilde{v}(x)$ provides piecewise linear interpolation between the 'nodal values' $V_j$.

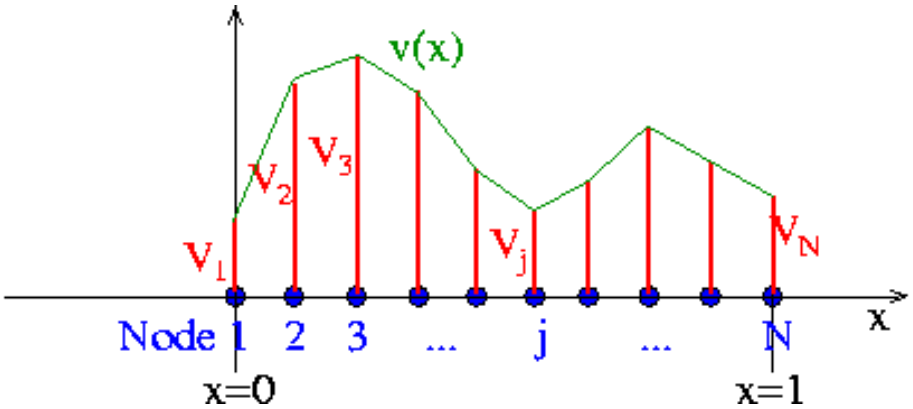


**Figure 1.4 The superposition of the (global) linear finite-element shape functions provides a piecewise linear interpolation between the `nodal values'.**

Why are these shape functions useful in the Galerkin method? Consider the requirements listed at the beginning of this section:

1. It is easy to construct a function $u_p(x)$ that satisfies the essential boundary conditions by choosing

$$u_p(x) = g_0\psi_1(x) + g_1\psi_N(x),$$

where $\psi_1(x)$ and $\psi_N(x)$ are the global finite-element shape functions associated with the two boundary nodes, 1 and $N$.

2. Regarding the requirements on the basis functions:

   (a) The global finite-element shape functions $\psi_j(x)$ and their first derivatives are square integrable. Hence, the finite-dimensional function space $H^1_{0,FEM}(D)$ spanned by the basis functions $\psi_j(x) \quad (j = 2, ..., N-1)$, associated with the internal nodes, is a subset of $H^1_0$, as required. Furthermore, it is easy to show that

$$\left| v(x) - \sum_{j=2}^{N-1} v(X_j)\psi_j(x) \right| \to 0 \quad \text{as } N \to \infty \text{ and } h = \tfrac{1}{N-1} \to 0, \qquad (18)$$

   for any $v(x) \in H^1_0(D)$. In other words, $H^1_{0,FEM}(D)$ approaches $H^1_0(D)$ as $N \to \infty$.

   (b) The global finite-element shape functions $\psi_2(x), \psi_3(x), ..., \psi_{N-1}(x)$ vanish on the domain boundary.

   (c) The Jacobian matrix is sparse because its entries

$$J_{kj} = \frac{\partial r_k}{\partial U_j} = \int_0^1 \frac{\mathrm{d}\psi_j(x)}{\mathrm{d}x} \frac{\mathrm{d}\psi_k(x)}{\mathrm{d}x} \, \mathrm{d}x$$

   are nonzero when the basis functions $\psi_j(x)$ and $\psi_k(x)$ are both non-zero. For these shape functions,

$$J_{kj} \neq 0 \quad \text{when} \quad k = j-1,\ j,\ j+1,$$

   indicating that the Jacobian matrix is tri-diagonal.

We can now formulate the finite-element-based solution of problem P1 in the following algorithm:

**Algorithm 2: Finite Element solution of problem P1 by Newton's method**

- Choose the number of nodal points, $N$, and distribute them evenly through the domain so that $X_j = (j-1)h$, where $h = 1/(N-1)$. This defines the global shape functions $\psi_j(x)$.
- Set

$$u_p(x) = g_0\psi_1(x) + g_1\psi_N(x)$$

  and

$$u_h(x) = \sum_{j=2}^{N-1} U_j \, \psi_j(x).$$

- Provide an initial guess for the unknowns $U_2, U_3, ..., U_{N-1}$. Since P1 is a linear problem, the quality of the initial guess is irrelevant and we can simply set $U_i^{(0)} = 0 \quad \text{for } i = 2, ..., N-1$.
- Determine the residuals

$$r_k^{(0)} = \int_0^1 \left\{ \left( g_0 \frac{\mathrm{d}\psi_1(x)}{\mathrm{d}x} + g_1 \frac{\mathrm{d}\psi_N(x)}{\mathrm{d}x} + \sum_{j=2}^{N-1} U_j^{(0)} \frac{\mathrm{d}\psi_j(x)}{\mathrm{d}x} \right) \frac{\mathrm{d}\psi_k(x)}{\mathrm{d}x} + f(x)\,\psi_k(x) \right\} \mathrm{d}x \qquad \text{for } k = 2, ..., N-1,$$

  and the entries in the Jacobian matrix

$$J_{kj} = \frac{\partial r_k}{\partial U_j} = \int_0^1 \frac{\mathrm{d}\psi_j(x)}{\mathrm{d}x} \frac{\mathrm{d}\psi_k(x)}{\mathrm{d}x} \, dx \quad \text{for } j, k = 2, ..., N-1.$$

- Solve the linear system

$$\sum_{j=2}^{N-1} J_{kj}\ \delta U_j = -r_k^{(0)} \quad \text{for } k = 2, ..., N-1.$$

  for $\delta U_k\ (k = 2, ..., N-1)$ .

- Correct the initial guess via

$$U_j = U_j^{(0)} + \ \delta U_j \quad \text{for } j = 2, ..., N-1.$$

  P1 is a linear problem, so $U_j\ (j = 2, ..., N-1)$ is the exact solution. For nonlinear problems, we would have to continue the Newton iteration until the residuals $r_k(U_2, ..., U_{N-1})\ (k = 2, ..., N-1)$ were sufficiently small.

- The finite-element solution is

$$u^{(FE)}(x) = g_0\psi_1(x) + g_1\psi_N(x) + \sum_{j=2}^{N-1} U_j\psi_j(x) \qquad (19)$$

### 1.3.2 Improving the quality of the solution: non-uniformly spaced nodes and higher-order shape functions

Algorithm 2 presents the simplest possible implementation of the finite element method for problem P1. We now discuss two straightforward extensions that can significantly improve the quality of the approximate solution.
The finite-element approximation in the previous section is piecewise linear between the nodal values. The accuracy to which the exact solution can be represented by a piecewise linear interpolant is limited by the number of nodal points, which is the essence of the convergence statement (18). The number of nodes required to resolve the solution to a given accuracy depends on the nature of the solution — more nodes are needed to interpolate rapidly varying functions.
If the solution is rapidly varying in a small portion of the domain it would be wasteful to use the same (fine) nodal spacing throughout the domain. A non-uniform spacing, see below,

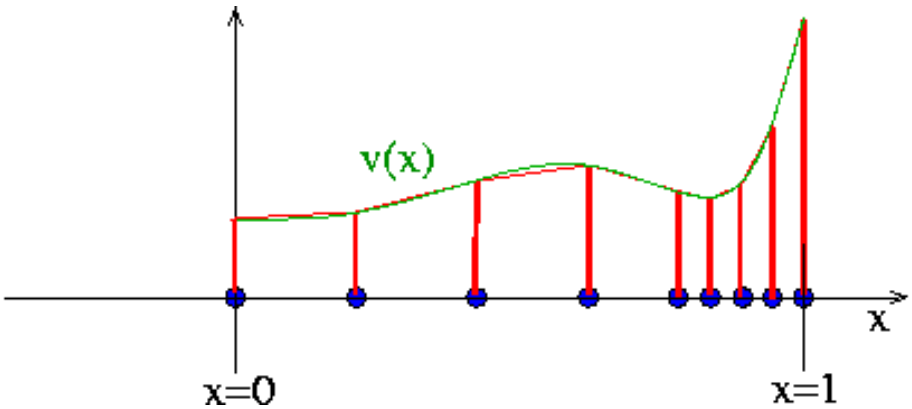


**Figure 1.5 Adaptive 1D finite element mesh: Non-uniform spacing of nodes to achieve a high resolution only where it is required.**

improves the accuracy of the solution without greatly increasing the total number of unknowns. Non-uniform spacing of the nodes is easy to implement and does not require any significant changes in Algorithm 2 – we simply choose appropriate values for the nodal positions $X_j$; an approach known as "h-refinement" because it alters the distance, $h$, between nodes.

**An aside: adaptive mesh refinement**

A non-uniform distribution of nodes requires a priori knowledge of the regions in which we expect the solution to undergo rapid variations. The alternative is to use adaptive mesh refinement: start with a relatively coarse, uniform mesh and compute the solution. If the solution on the coarse mesh displays rapid variations in certain parts of the domain (and is therefore likely to be poorly resolved), refine the mesh in these regions and re-compute. Such adaptive mesh refinement procedures can be automated and are implemented in `oomph-lib` for a large number of problems; see the section A 2D example for an example.

The quality of the interpolation can also be improved by using higher-order interpolation, but maintaining

- the compact support for the global shape functions, so that $\psi_j(x)$ is nonzero only in the vicinity of node $j$, and
- the interpolation condition
$$\psi_j(X_i) = \delta_{ij}$$
so that the shape function $\psi_j$ is equal to one at node $j$ and zero at all others.

For instance, we can use the (global) quadratic finite-element shape functions shown below:

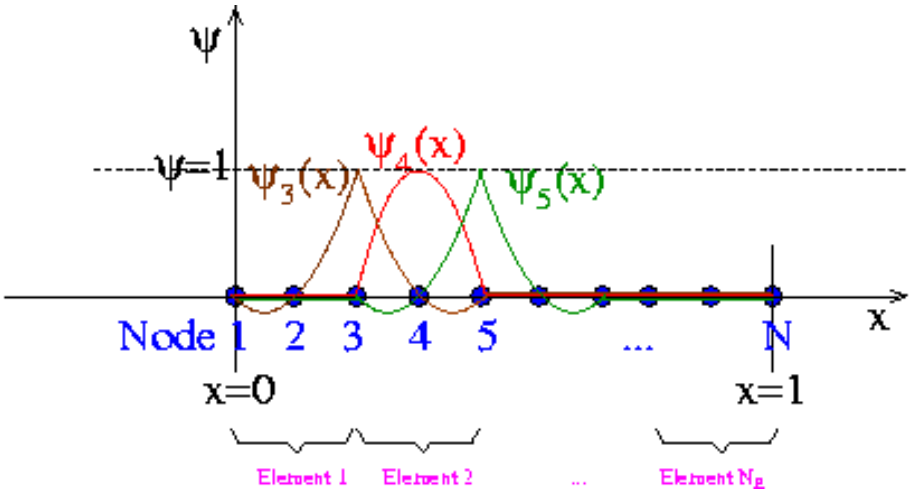


**Figure 1.6 (Global) quadratic finite-element basis functions in 1D.**

Note that we could also use shape functions with identical functional forms; instead, we have chosen two different forms for the shape functions so that the smooth sections of the different shape functions overlap within the elements. This is not necessary but facilitates the representation of the global shape functions in terms of their local counterparts within elements, see section Local coordinates.
The implementation of higher-order interpolation does not require any significant changes in Algorithm 2. We merely specify the functional form of the (global) quadratic finite-element shape functions sketched above. This approach is known as "p-refinement", because it increases order of the **p**olynomials that are used to represent the solution.

## 1.4 An element-by-element implementation for 1D problems

From a mathematical point of view, the development of the finite element method for the 1D model problem P1 is now complete, but Algorithm 2 has a number of features that would make it awkward to implement in an actual computer program. Furthermore, having been derived directly from Galerkin's method, the algorithm is based on globally-defined shape functions and does not exploit the potential subdivision of the domain into "elements". We shall now derive a mathematically equivalent scheme that can be implemented more easily.

### 1.4.1 Improved book-keeping: distinguishing between equations and nodes.

Since only a subset of the global finite-element shape functions act as basis functions for the (homogeneous) functions $\phi^{(test)}(x)$ and $u_h(x)$, algorithm 2 resulted in a slightly awkward numbering scheme for the equations and

the unknown (nodal) values. The equation numbers range from 2 to $N-1$, rather than from 1 to $N-2$ because we identified the unknowns by the node numbers. Although this is perfectly transparent in our simple 1D example, the book-keeping quickly becomes rather involved in more complicated problems. We therefore treat node and equation numbers separately. [**Note:** We shall use the terms "equation number" and "number of the unknown" interchangeably; this is possible because we must always have the same number of equations and unknowns.] We can (re-)write the finite-element solution (19) in more compact form as

$$u^{(FE)}(x) = \sum_{j=1}^{N} U_j \, \psi_j(x).$$

where the summation now includes *all* nodes in the finite element mesh. To make this representation consistent with the boundary conditions, the nodal values, $U_j$, of nodes on the boundary are set to the prescribed boundary values

$$U_j = g(X_j) \quad \text{if } X_j \in \partial D.$$

In the 1D problem P1, $U_1 = g_0$ and $U_N = g_1$.
Furthermore, we associate each unknown nodal value, $U_j$, with a distinct equation number, $\mathcal{E}(j)$, in the range from 1 to $N-2$. In the above example, the equation numbering scheme is given by

| Node number $j$ | 1 | 2 | 3 | ... | $N-1$ | $N$ |
|---|---|---|---|---|---|---|
| Equation number $\mathcal{E}(j)$ | n/a | 1 | 2 | ... | $N-2$ | n/a |

where n/a indicates a node whose value is prescribed by the boundary conditions. To facilitate the implementation in a computer program, we indicate the fact that a nodal value is determined by boundary conditions (i.e. that it is "pinned"), by setting the equation number to a negative value (-1, say), so that the equation numbering scheme becomes

| Node number $j$ | 1 | 2 | 3 | ... | $N-1$ | $N$ |
|---|---|---|---|---|---|---|
| Equation number $\mathcal{E}(j)$ | -1 | 1 | 2 | ... | $N-2$ | -1 |

We now re-formulate algorithm 2 as follows (the revised parts of the algorithm are enclosed in boxes):

**Algorithm 3: Finite Element solution of problem P1**

**Phase 1: Setup**

- Discretise the domain with $N$ nodes which are located at $X_j$, $j = 1, ..., N$, and choose the order of the global shape functions $\psi_j(x)$.

- Initialise the total number of unknowns, $M = 0$.
- Loop over all nodes $j = 1, ..., N$:
  - If node j lies on the boundary:
    - Assign its value according to the (known) boundary condition

      $$U_j = g(X_j)$$

    - Assign a negative equation number to reflect its "pinned" status:

      $$\mathcal{E}(j) = -1$$

  - Else:

* Increment the number of the unknowns

$$M = M + 1$$

* Assign the equation number

$$\mathcal{E}(j) = M$$

* Provide an initial guess for the unknown nodal value, e.g.

$$U_j = 0.$$

- Now that $\psi_j(x)$ and $U_j$ have been defined and initialised, we can determine the current FE approximations for $u(x)$ and $\mathrm{d}u(x)/\mathrm{d}x$ from

$$u^{(FE)}(x) = \sum_{k=1}^{N} U_k \psi_k(x) \quad \text{and} \quad \frac{\mathrm{d}u^{(FE)}(x)}{\mathrm{d}x} = \sum_{k=1}^{N} U_k \frac{\mathrm{d}\psi_k(x)}{\mathrm{d}x}.$$

**Phase 2: Solution**

- Loop over all nodes $k = 1, ..., N$:
  - Determine the equation number: $\mathcal{E}(k)$
  - If $\mathcal{E}(k) \neq -1$:

$$r_{\mathcal{E}(k)} = \int_0^1 \frac{\mathrm{d}u^{(FE)}(x)}{\mathrm{d}x} \frac{\mathrm{d}\psi_k(x)}{\mathrm{d}x}\, dx + \int_0^1 f(x)\, \psi_k(x)\, dx.$$

    * Loop over all nodes $j = 1, ..., N$:
    * Determine the equation number: $\mathcal{E}(j)$
    * If $\mathcal{E}(j) \neq -1$:

$$J_{\mathcal{E}(k)\mathcal{E}(j)} = \frac{\partial r_{\mathcal{E}(k)}}{\partial U_j} = \int_0^1 \frac{\mathrm{d}\psi_j(x)}{\mathrm{d}x} \frac{\mathrm{d}\psi_k(x)}{\mathrm{d}x}\, dx$$

- Solve the $M \times M$ linear system

$$\sum_{j=1}^{M} J_{kj}\, y_j = -r_k \quad \text{for } k = 1, ..., M.$$

for $y_k$ $(k = 1, ..., M)$.

- Loop over all nodes $j = 1, ..., N$:
  - Determine the equation number: $\mathcal{E}(j)$
  - If $\mathcal{E}(j) \neq -1$:
  - Correct the initial guess via

$$U_{\mathcal{E}(j)} = U_{\mathcal{E}(j)} + y_{\mathcal{E}(j)}$$

.

- P1 is a linear problem, so $U_j$ $(j = 1, ..., N)$ is the exact solution. For nonlinear problems, we would have to

continue the Newton iteration until the residuals $r_k$ $(k = 1, ..., M)$ were sufficiently small.

**Phase 3: Postprocessing (document the solution)**

- The finite-element solution is given by

$$u^{(FE)}(x) = \sum_{j=1}^{N} U_j \psi_j(x).$$

### 1.4.2 Element-by-element assembly

In its current form, our algorithm assembles the equations and the Jacobian matrix equation-by-equation and does not exploit the finite support of the global shape functions, which permits decomposition of the domain into elements. Each element consists of a given number of nodes that depends upon the order of local approximation within the element. In the case of linear interpolation each element consists of two nodes, as seen below:

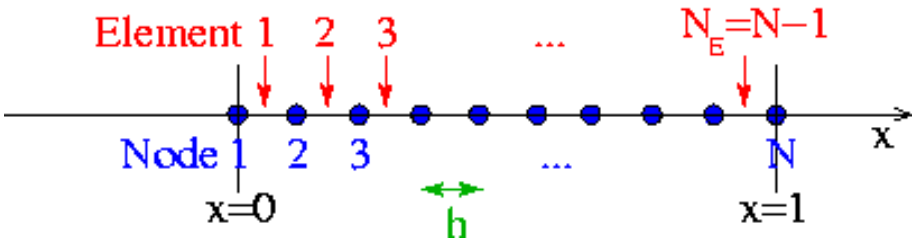


**Figure 1.7 A 1D finite element mesh.**

In the following three sections we shall develop an alternative, element-based assembly procedure, which involves the introduction of

- local and global node numbers,
- local and global equation numbers,
- local coordinates.

#### 1.4.2.1 Local and global node numbers

Consider the discretisation of the 1D problem P1 with $N_E$ two-node (linear) finite elements. The residual associated with node k is given by

$$r_{\mathcal{E}(k)} = \int_0^1 \left( \frac{\mathrm{d}u^{(FE)}(x)}{\mathrm{d}x} \frac{\mathrm{d}\psi_k(x)}{\mathrm{d}x} + f(x)\, \psi_k(x) \right) dx. \qquad (20)$$

The global finite-element shape functions have finite support, so the integrand is non-zero only in the two elements $k-1$ and $k$, adjacent to node $k$. This allows us to write

$$r_{\mathcal{E}(k)} = \int_{\text{Element } k-1} \left( \frac{\mathrm{d}u^{(FE)}(x)}{\mathrm{d}x} \frac{\mathrm{d}\psi_k(x)}{\mathrm{d}x}\, dx + f(x)\, \psi_k(x) \right) dx + \int_{\text{Element } k} \left( \frac{\mathrm{d}u^{(FE)}(x)}{\mathrm{d}x} \frac{\mathrm{d}\psi_k(x)}{\mathrm{d}x}\, dx + f(x)\, \psi_k(x) \right) dx.$$

Typically (e.g. in the Newton method) we require *all* the residuals $r_k$ $(k = 1, ..., M)$ — the entire residual vector. We could compute the entries in this vector by using the above equation to calculate each residual $r_{\mathcal{E}(k)}$ individually for all (unpinned) nodes $k$. In the process, we would visit each element twice: once for each of the two residuals that are associated with its nodes. We can, instead, re-arrange this procedure to consist of a single loop over the elements, in which the appropriate contribution is added to the global residuals associated with each element's nodes. In order to achieve this, we must introduce the concept of local and global node numbers, illustrated in this sketch:

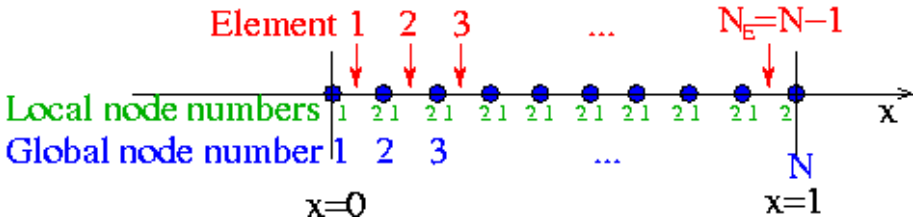


**Figure 1.8 Local and global node numbers in a 1D mesh.**

We label the nodes in each element $e$ with local node numbers so that for two-node elements, the left (right) node has the local node number 1 (2). The relation between local and global node numbers can be represented in a simple lookup scheme

$$j_{global} = \mathcal{J}(j_{local}, e),$$

which determines the global node number $j_{global}$ of local node $j_{local}$ in element $e$. The lookup scheme establishes how the nodes are connected by elements and is one of the main steps in the "mesh generation" process. For the 1D example above, the lookup scheme is given by

| Element $e$ | 1 | | 2 | | ... | $N_E$ | |
|---|---|---|---|---|---|---|---|
| Local node number $j$ | 1 | 2 | 1 | 2 | ... | 1 | 2 |
| Global node number $\mathcal{J}(j,e)$ | 1 | 2 | 2 | 3 | ... | $N-1$ | $N$ |

where $N = N_E + 1$.
If we discretise the domain with three-node (quadratic) elements, as in this sketch,

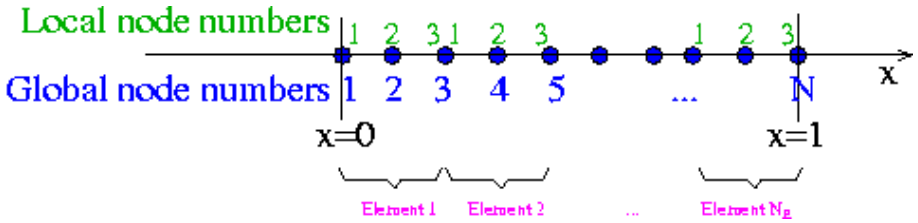


**Figure 1.9 Local and global node numbers in a 1D mesh with three-node elements.**

the lookup scheme becomes

| Element $e$ | 1 | | | 2 | | | ... | $N_E$ | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Local node number $j$ | 1 | 2 | 3 | 1 | 2 | 3 | ... | 1 | 2 | 3 |
| Global node number $\mathcal{J}(j,e)$ | 1 | 2 | 3 | 3 | 4 | 5 | ... | N-2 | N-1 | N |

where $N = 2N_E + 1$. Provided such a lookup scheme has been constructed, the global residual vector and the global Jacobian matrix for $n$-node elements can be assembled with the following algorithm

**Algorithm 4: Assembling the residual vector and the Jacobian matrix using local and global node numbers**

- Initialise the residual vector, $r_k = 0$ for $k = 1, ..., M$ and the Jacobian matrix $J_{kj} = 0$ for $j, k = 1, ..., M$.
- Loop over the elements $e = 1, ..., N_E$
  - Loop over the local nodes $k_{local} = 1, ..., n$
    - Determine the global node number $k_{global} = \mathcal{J}(k_{local}, e)$.
    - Determine the global equation number $\mathcal{E}(k_{global})$.

- If $\mathcal{E}(k_{global}) \neq -1$
  - Add the element's contribution to the residual

$$r_{\mathcal{E}(k_{global})} = r_{\mathcal{E}(k_{global})} + \int_e \left( \frac{\mathrm{d}u^{(FE)}(x)}{\mathrm{d}x} \frac{\mathrm{d}\psi_{k_{global}}(x)}{\mathrm{d}x} + f(x)\, \psi_{k_{global}}(x) \right) dx$$

  - Loop over the local nodes $j_{local} = 1, ..., n$
  - Determine the global node number $j_{global} = \mathcal{J}(j_{local}, e)$
  - Determine the global equation number $\mathcal{E}(j_{global})$
  - If $\mathcal{E}(j_{global}) \neq -1$: Add the element's contribution to the Jacobian matrix

$$J_{\mathcal{E}(k_{global})\mathcal{E}(j_{global})} = J_{\mathcal{E}(k_{global})\mathcal{E}(j_{global})} + \int_e \left( \frac{\mathrm{d}\psi_{j_{global}}(x)}{\mathrm{d}x} \frac{\mathrm{d}\psi_{k_{global}}(x)}{\mathrm{d}x} \right) dx$$

#### 1.4.2.2 Local and global equation numbers

Each element makes a contribution to only a small number of entries in the global residual vector and the Jacobian matrix; namely, those entries associated with the unknowns stored at the nodes in the element. In general, each element is associated with $\mathcal{N}_{dof}(e)$ unknowns, say. Element $e$ contributes to $\mathcal{N}_{dof}(e)$ entries in the *global* residual vector and $\mathcal{N}^2_{dof}(e)$ entries in the *global* Jacobian matrix. In fact, the finite support of the shape functions leads to sparse *global* Jacobian matrices and it would be extremely wasteful to allocate storage for all its entries, and use Algorithm 4 to calculate those that are non-zero. Instead, we combine each element's contributions into an $\mathcal{N}_{dof}(e) \times \mathcal{N}_{dof}(e)$ "element Jacobian matrix" $J^{(e)}_{ij}$, $i, j = 1, ..., \mathcal{N}_{dof}(e)$ and "element residual vector" $r^{(e)}_i$, $i = 1, ..., \mathcal{N}_{dof}(e)$. These (dense) matrices and vectors are then assembled into the global matrix and residuals vector. The entries in the element's Jacobian matrix and its residual vector are labelled by the "local equation numbers", which range from 1 to $\mathcal{N}_{dof}(e)$ and are illustrated in this sketch:

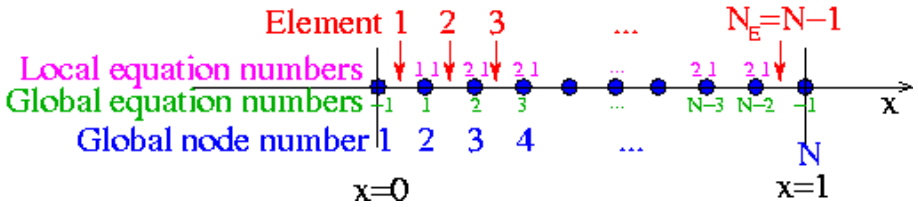


**Figure 1.10 Local and global node and equation numbers.**

In order to add the elemental contributions to the correct *global* entries in the residual vector and Jacobian matrix, it is necessary to translate between the local and global equation numbers; and we introduce another lookup scheme $\widehat{\mathcal{E}}(i_{local}, e)$ that stores the global equation number corresponding to local equation $i_{local}$ in element $e$. The lookup scheme can be generated by the following algorithm

**Algorithm 5: Establishing the relation between local and global equation numbers**

- Construct the global equation numbering scheme, using the algorithm detailed in Phase 1 of Algorithm 3.
- Loop over the elements $e = 1, ..., N_e$
  - Initialise the counter for the number of degrees of freedom in the element, $j_{dof} = 0$.
  - Loop over the element's local nodes $j_{local} = 1, ..., n$
    - Determine the global node number $j_{global} = \mathcal{J}(j_{local}, e)$
    - Determine the global equation number $\mathcal{E}(j_{global})$
    - If $\mathcal{E}(j_{global}) \neq -1$:

· Increment the number of degrees of freedom in the element, $j_{dof} = j_{dof} + 1$

· Add the entry to the lookup scheme that relates local and global equation numbers, $\widehat{\mathcal{E}}(j_{dof}, e) = \mathcal{E}(j_{global})$

– Assign the number of degrees of freedom in the element, $\mathcal{N}_{dof}(e) = j_{dof}$

For the 1D problem P1 with two-node elements, the lookup scheme has the following form:

| Element $e$ | 1 | 2 | | 3 | | ... | $N_E - 1$ | | $N_E$ |
|---|---|---|---|---|---|---|---|---|---|
| Number of degrees of freedom $\mathcal{N}_{dof}(e)$ | 1 | 2 | | 2 | | ... | 2 | | 1 |
| Local equation number $i_{local}$ | 1 | 1 | 2 | 1 | 2 | ... | 1 | 2 | 1 |
| Global equation number $\widehat{\mathcal{E}}(i_{local}, e)$ | 1 | 1 | 2 | 2 | 3 | ... | $N-3$ | $N-2$ | $N-2$ |

Using this lookup scheme, we can re-arrange the computation of the residual vector and the Jacobian matrix as follows:

**Algorithm 6: Element-based assembly of the residual vector and the Jacobian matrix**

- Initialise the global residual vector, $r_j = 0$ for $j = 1, ..., M$ and the global Jacobian matrix $J_{jk} = 0$ for $j, k = 1, ..., M$
- Loop over the elements $e = 1, ..., N_E$

**Compute the element's residual vector and Jacobian matrix**

• Determine the number of degrees of freedom in this element, $n_{dof} = \mathcal{N}_{dof}(e)$.
• Initialise the counter for the local degrees of freedom $i_{dof} = 0$ (counting the entries in the element's residual vector and the rows of the element's Jacobian matrix)

– Loop over the local nodes $j_{local} = 1, ..., n$
• Determine the global node number $j_{global} = \mathcal{J}(j_{local}, e)$
• Determine the global equation number $\mathcal{E}(j_{global})$
• If $\mathcal{E}(j_{global}) \neq -1$:

• Increment the counter for the local degrees of freedom $i_{dof} = i_{dof} + 1$

• Determine the entry in the element's residual vector

$$r^{(e)}_{i_{dof}} = \int_e \left( \frac{\mathsf{d}u^{(FE)}(x)}{\mathsf{d}x} \, \frac{\mathsf{d}\psi_{j_{global}}(x)}{\mathsf{d}x} + f(x) \, \psi_{j_{global}}(x) \right) dx$$

• Initialise the second counter for the local degrees of freedom $j_{dof} = 0$ (counting the columns in the element's Jacobian matrix)

• Loop over the local nodes $k_{local} = 1, ..., n$
* Determine the global node number $k_{global} = \mathcal{J}(k_{local}, e)$
* Determine the global equation number $\mathcal{E}(k_{global})$

* If $\mathcal{E}(k_{global}) \neq -1$:

• Increment the counter for the local degrees of freedom $j_{dof} = j_{dof} + 1$

• Determine the entry in the element's Jacobian matrix

$$J^{(e)}_{i_{dof} j_{dof}} = \int_e \left( \frac{\mathsf{d}\psi_{k_{global}}(x)}{\mathsf{d}x} \; \frac{\mathsf{d}\psi_{j_{global}}(x)}{\mathsf{d}x} \right) dx$$

**Add the element's contribution to the global residual vector and Jacobian matrix**

– Loop over the local degrees of freedom $i_{dof} = 1, ..., n_{dof}$

* Add the element's contribution to the global residual vector

$$r_{\widehat{\mathcal{E}}(i_{dof},e)} = r_{\widehat{\mathcal{E}}(i_{dof},e)} + r^{(e)}_{i_{dof}}$$

· Loop over the local degrees of freedom $j_{dof} = 1, ..., n_{dof}$

• Add the element's contribution to the global Jacobian matrix

$$J_{\widehat{\mathcal{E}}(i_{dof},e)\widehat{\mathcal{E}}(j_{dof},e)} = J_{\widehat{\mathcal{E}}(i_{dof},e)\widehat{\mathcal{E}}(j_{dof},e)} + J^{(e)}_{i_{dof} j_{dof}}$$

Note that the order in which we loop over the local degrees of freedom within each element *must* be the same as the order used when constructing the local equation numbering scheme of Algorithm

1. [**Exercise:** What would happen if we reversed the order in which we loop over the element's nodes in Algorithm 5 while retaining Algorithm 6 in its present form? Hint: Note that the local equation numbers are computed "on the fly" by the highlighted sections of the two algorithms.] In an actual implementation of the above procedure, Algorithms 5 and 6 are likely to be contained in separate functions. When the functions are first implemented (in the form described above), they will obviously be consistent with each other. However, there is a danger that in subsequent code revisions changes might only be introduced in one of the two functions. To avoid the potential for such disasters, it is preferable to create an explicit storage scheme for the local equation numbers that is constructed during the execution of Algorithm 5 and used in Algorithm 6. For this purpose, we introduce yet another lookup table, $\mathcal{L}(j_{local}, e)$, which stores the local equation number associated with the nodal value stored at local node $j_{local}$ in element $e$. Again we set the equation number to -1 if the nodal value is pinned. The revised form of Algorithm 5 is then given by (as before, only the sections that are highlighted have been changed):

**Algorithm 7: Establishing the relation between local and global equation numbers (revised)**

- Set up the global equation numbering scheme, using the algorithm detailed in Phase 1 of Algorithm 3.
- Loop over the elements $e = 1, ..., N_e$
  - Initialise the counter for the number of degrees of freedom in the element, $j_{dof} = 0$.
  - Loop over the element's local nodes $j_{local} = 1, ..., n$

* Determine the global node number $j_{global} = \mathcal{J}(j_{local}, e)$
* Determine the global equation number $\mathcal{E}(j_{global})$
* If $\mathcal{E}(j_{global}) \neq -1$:
  - Increment the number of degrees of freedom in the element, $j_{dof} = j_{dof} + 1$
  - Add the entry to the lookup scheme that relates local and global equation numbers, $\widehat{\mathcal{E}}(j_{dof}, e) = \mathcal{E}(j_{global})$

Store the local equation number associated with the current local node: $\mathcal{L}(j_{local}, e) = j_{dof}$.

.

* Else:

- Set the local equation number associated with the current local node to -1 to indicate that it is pinned: $\mathcal{L}(j_{local}, e) = -1$

– Assign the number of degrees of freedom in the element, $\mathcal{N}_{dof}(e) = j_{dof}$

For the 1D problem P1 with two-node elements the elements, the lookup table $\mathcal{L}(j_{local}, e)$ has the following entries:

| Element $e$ | 1 | | 2 | | 3 | | ... | $N_E - 1$ | | $N_E$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Local node number $j_{local}$ | 1 | 2 | 1 | 2 | 1 | 2 | ... | 1 | 2 | 1 | 2 |
| Local equation number $\mathcal{L}(j_{local}, e)$ | -1 | 1 | 1 | 2 | 1 | 2 | ... | 1 | 2 | 1 | -1 |

Using this lookup scheme, we revise Algorithm 6 as follows (only the highlighted regions have changed; we have removed the initialisation of the "counters" for the equation numbers since they are no longer computed "on the fly"):

**Algorithm 9: Element-based assembly of the residual vector and the Jacobian matrix (revised)**

- Initialise the global residual vector, $r_j = 0$ for $j = 1, ..., M$ and the global Jacobian matrix $J_{jk} = 0$ for $j, k = 1, ..., M$
- Loop over the elements $e = 1, ..., N_E$

**Compute the element's residual vector and Jacobian matrix**

- Determine the number of degrees of freedom in this element, $n_{dof} = \mathcal{N}_{dof}(e)$.
- Loop over the local nodes $j_{local} = 1, ..., n$

– Determine the global node number $j_{global} = \mathcal{J}(j_{local}, e)$

– Determine the global equation number $\mathcal{E}(j_{global})$

- If $\mathcal{E}(j_{global}) \neq -1$:

- Determine the local equation number from the element's lookup scheme $i_{dof} = \mathcal{L}(j_{local}, e)$.

- Determine the entry in the element's residual vector

$$r^{(e)}_{i_{dof}} = \int_e \left( \frac{\mathsf{d}u^{(FE)}(x)}{\mathsf{d}x} \, \frac{\mathsf{d}\psi_{j_{global}}(x)}{\mathsf{d}x} + f(x) \, \psi_{j_{global}}(x) \right) dx$$

- Loop over the local nodes $k_{local} = 1, ..., n$
  - Determine the global node number $k_{global} = \mathcal{J}(k_{local}, e)$
  - Determine the global equation number $\mathcal{E}(k_{global})$
  - If $\mathcal{E}(k_{global}) \neq -1$:

    · Determine the local equation number from the element's lookup scheme $j_{dof} = \mathcal{L}(k_{local}, e)$.

    · Determine the entry in the element's Jacobian matrix

$$J^{(e)}_{i_{dof} j_{dof}} = \int_e \left( \frac{\mathsf{d}\psi_{k_{global}}(x)}{\mathsf{d}x} \, \frac{\mathsf{d}\psi_{j_{global}}(x)}{\mathsf{d}x} \right) dx$$

**Add the element's contribution to the global residual vector and Jacobian matrix**

– Loop over the local degrees of freedom $i_{dof} = 1, ..., n_{dof}$
  - Add the element's contribution to the global residual vector

$$r_{\widehat{\mathcal{E}}(i_{dof}, e)} = r_{\widehat{\mathcal{E}}(i_{dof}, e)} + r^{(e)}_{i_{dof}}$$

  - Loop over the local degrees of freedom $j_{dof} = 1, ..., n_{dof}$
    · Add the element's contribution to the global Jacobian matrix

$$J_{\widehat{\mathcal{E}}(i_{dof}, e)\widehat{\mathcal{E}}(j_{dof}, e)} = J_{\widehat{\mathcal{E}}(i_{dof}, e)\widehat{\mathcal{E}}(j_{dof}, e)} + J^{(e)}_{i_{dof} j_{dof}}$$

#### 1.4.2.3 Local coordinates

Algorithm 9 computes the residual vector and the Jacobian matrix using an element-by-element assembly process. The basis functions are still based on a global definition (17) that involves unnecessary references to quantities external to the element. For example, in element $j$ the tests for $x < X_j$ and $x > X_{j+1}$ in (17) are unnecessary because these coordinate ranges are always outside the element. We shall now develop an alternative, local representation of the shape functions that involves only quantities that are intrinsic to each element.
For this purpose, we introduce a local coordinate $s \in [-1, 1]$ that parametrises the position $x(s)$ within an element so that (for two-node elements) the local coordinates $s = \pm 1$ correspond to local nodes 1 and 2, respectively. The local linear shape functions

$$\psi_1(s) = \frac{1}{2}(1 - s) \quad \text{and} \quad \psi_2(s) = \frac{1}{2}(1 + s)$$

are the natural generalisations of the global shape functions:

- $\psi_j(s)$ is equal to 1 at local node j and zero at the element's other node,
- $\psi_j(s)$ varies linearly between the nodes.

These local shape functions are easily generalised to elements with a larger number of nodes. For instance, the local shape functions for a three-node element whose nodes are distributed uniformly along the element, are given by

$$\psi_1(s) = \frac{1}{2}s(s-1), \quad \psi_2(s) = (s+1)(1-s) \quad \text{and} \quad \psi_3(s) = \frac{1}{2}s(s+1).$$

We represent the solution within element $e$ as

$$u^{(FE)}(s) = \sum_{j=1}^{n} U_{\mathcal{J}(j,e)}\ \psi_j(s),$$

where $n$ is the number of nodes in the element. $u$ is now represented exclusively in terms of quantities that are intrinsic to the element: the element's nodal values and the local coordinate. The evaluation of the integrals in algorithm 2 requires the evaluation of $\mathrm{d}u/\mathrm{d}x$, rather than $\mathrm{d}u/\mathrm{d}s$, and $f(x)$ rather than $f(s)$. In order to evaluate these terms, we must specify the mapping $x(s)$ between the local and global coordinates. The mapping $x(s)$ should be one-to-one and it must interpolate the nodal positions so that in a two-node element $e$

$$x(s=-1) = X_{\mathcal{J}(1,e)} \quad \text{and} \quad x(s=1) = X_{\mathcal{J}(2,e)}.$$

There are many mappings that satisfy these conditions but, within the finite-element context, the simplest choice is to use the local shape functions themselves by writing

$$x(s) = \sum_{j=1}^{n} X_{\mathcal{J}(j,e)}\ \psi_j(s).$$

This is known as an "isoparametric mapping" because the same ("iso") functions are used to interpolate the unknown function *and* the global coordinates. Derivatives with respect to the global coordinates can now be evaluated via the chain rule

$$\frac{\mathrm{d}u}{\mathrm{d}x} = \frac{\mathrm{d}u}{\mathrm{d}s}\left(\frac{\mathrm{d}x}{\mathrm{d}s}\right)^{-1}.$$

In element $e$,

$$\frac{\mathrm{d}u}{\mathrm{d}x} = \left(\sum_{j=1}^{n} U_{\mathcal{J}(j,e)}\ \frac{\mathrm{d}\psi_j(s)}{\mathrm{d}s}\right)\left(\sum_{j=1}^{n} X_{\mathcal{J}(j,e)}\ \frac{\mathrm{d}\psi_j(s)}{\mathrm{d}s}\right)^{-1}.$$

Finally, integration over the element can be performed in local coordinates via

$$\int_e (...)\ dx = \int_{-1}^{1} (...)\ \widehat{\mathcal{J}}\ ds,$$

where

$$\widehat{\mathcal{J}} = \frac{\mathrm{d}x}{\mathrm{d}s}$$

is the Jacobian of the mapping between $x$ and $s$.
Typically the integrands are too complicated to be evaluated analytically and we use Gauss quadrature rules to evaluate them numerically. Gauss rules (or any other quadrature rules) are defined by

- the number of integration points $N_{int}$,
- the position of the integration points in the element $S_i,\ i = 1, ..., N_{int}$,
- the weights $W_i,\ i = 1, ..., N_{int}$,

and approximate the integral over the range $s \in [-1, 1]$ by the sum

$$\int_{-1}^{1} \mathcal{F}(s)\ ds \approx \sum_{i=1}^{N_{int}} W_i\ \mathcal{F}(S_i).$$

As an example, the integration points and weights for a three-point Gauss rule are given by

| Integration point $i$ | 1 | 2 | 3 |
|---|---|---|---|
| Location $S_i$ | $-\sqrt{3/5}$ | $0$ | $\sqrt{3/5}$ |
| Weight $W_i$ | 5/9 | 8/9 | 5/9 |

#### 1.4.2.4 Putting it all together

We have now developed the necessary tools to formulate the final version of the finite element solution of problem P1 which we summarise in the following algorithm:

**Algorithm 10: The final version of the finite-element solution of problem P1**

**Phase 1: Setup**
**Phase 1a: Problem specification**

- Choose the number of elements, $N_E$, and the number of nodes per element, $n$. This defines the total number of nodes, $N$, and the local shape functions
  $\psi_j(s)\ \ (j = 1, ..., n)$ for all elements.

  **Phase 1b: Mesh generation**

- Discretise the domain by specifying the positions $X_j\ (j = 1, .., N)$, of the $N$ nodes.
- Generate the lookup scheme $\mathcal{J}(j, e)$ that establishes the relation between global and local node numbers.
- Identify which nodes are located on which domain boundaries.

  **Phase 1c: "Pin" nodes with essential (Dirichlet) boundary conditions**

- Loop over all global nodes $j$ that are located on Dirichlet boundaries:
  - Assign a negative equation number to reflect the node's "pinned" status:

$$\mathcal{E}(j) = -1$$

  **Phase 1d: Apply boundary conditions and provide initial guesses for all unknowns**

- Loop over all global nodes $j = 1, ..., N$:
  - Provide an initial guess for the unknown nodal value (e.g. $U_j = 0$), while ensuring that the values assigned to nodes on the boundary are consistent with the boundary conditions so that $U_j = g(X_j)$.

  **Phase 1e: Set up the global equation numbering scheme**

- Initialise the total number of unknowns, $M = 0$.
- Loop over all global nodes $j = 1, ..., N$:
  - If global node $j$ is not pinned (i.e. if $\mathcal{E}(j) \neq -1$)
    - Increment the number of unknowns

$$M = M + 1$$

    - Assign the global equation number

$$\mathcal{E}(j) = M$$

  **Phase 1f: Set up the local equation numbering scheme**

- Loop over the elements $e = 1, ..., N_e$
  - Initialise the counter for the number of degrees of freedom in this element, $j_{dof} = 0$.
  - Loop over the element's local nodes $j_{local} = 1, ..., n$
    - Determine the global node number $j_{global} = \mathcal{J}(j_{local}, e)$
    - Determine the global equation number $\mathcal{E}(j_{global})$
    - If $\mathcal{E}(j_{global}) \neq -1$:
      - Increment the number of degrees of freedom in this element $j_{dof} = j_{dof} + 1$
      - Add the entry to the lookup scheme that relates global and local equation numbers, $\widehat{\mathcal{E}}(j_{dof}, e) = \mathcal{E}(j_{global})$
      - Store the local equation number associated with the current local node: $\mathcal{L}(j_{local}, e) = j_{dof}$.
    - Else:

· Set the local equation number associated with the current local node to -1 to indicate that it is pinned: $\mathcal{L}(j_{local}, e) = -1$

– Assign the number of degrees of freedom in the element $\mathcal{N}_{dof}(e) = j_{dof}$

**End of setup:**

The setup phase is now complete and we can determine the current FE representation of $u^{(FE)}(x)$ and $\mathrm{d}u^{(FE)}(x)/\mathrm{d}x$ in any element $e$ $(e = 1, .., N_E)$ from

$$u = \sum_{j=1}^{n} U_{\mathcal{J}(j,e)} \psi_j(s) \qquad \text{and} \qquad \frac{\mathrm{d}u}{\mathrm{d}x} = \left( \sum_{j=1}^{n} U_{\mathcal{J}(j,e)} \frac{\mathrm{d}\psi_j(s)}{\mathrm{d}s} \right) \left( \sum_{j=1}^{n} X_{\mathcal{J}(j,e)} \frac{\mathrm{d}\psi_j(s)}{\mathrm{d}s} \right)^{-1}$$

Similarly, the global coordinates and their derivatives with respect to the local coordinates are given by

$$x = \sum_{j=1}^{n} X_{\mathcal{J}(j,e)} \psi_j(s) \qquad \text{and} \qquad \widehat{\mathcal{J}} = \frac{\mathrm{d}x}{\mathrm{d}s} = \sum_{j=1}^{n} X_{\mathcal{J}(j,e)} \frac{\mathrm{d}\psi_j(s)}{\mathrm{d}s}.$$

The derivatives of the local shape functions, $\psi_j$, with respect to the global coordinate $x$ are

$$\frac{\mathrm{d}\psi_j}{\mathrm{d}x} = \frac{\mathrm{d}\psi_j(s)}{\mathrm{d}s} \left( \sum_{j=1}^{n} X_{\mathcal{J}(j,e)} \frac{\mathrm{d}\psi_j(s)}{\mathrm{d}s} \right)^{-1} \quad \text{for } j = 1, .., n$$

**Phase 2: Solution**

**Phase 2a: Set up the linear system**

- Initialise the global residual vector, $r_j = 0$ for $j = 1, ..., M$ and the global Jacobian matrix $J_{jk} = 0$ for $j, k = 1, ..., M$
- Loop over the elements $e = 1, ..., N_E$

**Compute the element's residual vector and Jacobian matrix**

• Determine the number of degrees of freedom in this element, $n_{dof} = \mathcal{N}_{dof}(e)$.
• Initialise the element residual vector, $r_j^{(e)} = 0$ for $j = 1, ..., n_{dof}$ and the element Jacobian matrix $J_{jk}^{(e)} = 0$ for $j, k = 1, ..., n_{dof}$
• Loop over the Gauss points $i_{int} = 1, ..., N_{int}$

– Determine the local coordinate of the integration point $s = S_{i_{int}}$ and the associated weight $W_{i_{int}}$.
– Compute
  • the global coordinate $x = x(s)$
  • the source function $f = f(x) = f(x(s))$
  • the derivative $\mathrm{d}u/\mathrm{d}x$
  • the shape functions $\psi_j$ and their derivatives $\mathrm{d}\psi_j/\mathrm{d}x$
  • the Jacobian of the mapping between local and global coordinates, $\widehat{\mathcal{J}} = \mathrm{d}x/\mathrm{d}s$

– Loop over the local nodes $j_{local} = 1, ..., n$
  * Determine the global node number $j_{global} = \mathcal{J}(j_{local}, e)$
  * Determine the global equation number $\mathcal{E}(j_{global})$

* If $\mathcal{E}(j_{global}) \neq -1$
  - Determine the local equation number from the element's lookup scheme $i_{dof} = \mathcal{L}(j_{local}, e)$.
  - Add the contribution to the element's residual vector

$$r^{(e)}_{i_{dof}} = r^{(e)}_{i_{dof}} + \left( \frac{\mathsf{d}u}{\mathsf{d}x} \frac{\mathsf{d}\psi_{j_{local}}}{\mathsf{d}x} + f\, \psi_{j_{local}} \right) \widehat{\mathcal{J}}\, W_{i_{int}}$$

  - Loop over the local nodes $k_{local} = 1, ..., n$
    · Determine the global node number $k_{global} = \mathcal{J}(k_{local}, e)$
    · Determine the global equation number $\mathcal{E}(k_{global})$
    · If $\mathcal{E}(k_{global}) \neq -1$
      - Determine the local equation number from the element's lookup scheme $j_{dof} = \mathcal{L}(k_{local}, e)$.
      - Add the contribution to the element's Jacobian matrix

$$J^{(e)}_{i_{dof} j_{dof}} = J^{(e)}_{i_{dof} j_{dof}} + \left( \frac{\mathsf{d}\psi_{k_{local}}}{\mathsf{d}x} \frac{\mathsf{d}\psi_{j_{local}}}{\mathsf{d}x} \right) \widehat{\mathcal{J}}\, W_{i_{int}}$$

**Add the element's contribution to the global residual vector and Jacobian matrix**

– Loop over the local degrees of freedom $i_{dof} = 1, ..., n_{dof}$
  * Add the element's contribution to the global residual vector

$$r_{\widehat{\mathcal{E}}(i_{dof}, e)} = r_{\widehat{\mathcal{E}}(i_{dof}, e)} + r^{(e)}_{i_{dof}}$$

  * Loop over the local degrees of freedom $j_{dof} = 1, ..., n_{dof}$
    - Add the element's contribution to the global Jacobian matrix

$$J_{\widehat{\mathcal{E}}(i_{dof}, e)\widehat{\mathcal{E}}(j_{dof}, e)} = J_{\widehat{\mathcal{E}}(i_{dof}, e)\widehat{\mathcal{E}}(j_{dof}, e)} + J^{(e)}_{i_{dof} j_{dof}}$$

**Phase 2b: Solve the linear system**

- Solve the $M \times M$ linear system

$$\sum_{j=1}^{M} J_{jk}\, y_k = -r_j \quad \text{for } j = 1, ..., M.$$

for $y_k \;\; (k = 1, ..., M)$.

**Phase 2c: Update the initial guess**

- Loop over all global nodes $j$:
  – Determine the equation number: $\mathcal{E}(j)$
  – If $\mathcal{E}(j) \neq -1$:

$$U_{\mathcal{E}(j)} = U_{\mathcal{E}(j)} + y_{\mathcal{E}(j)}$$

- Since P1 is a linear problem, $U_j \;\; (j = 1, ..., N)$ is the exact solution; if the problem was nonlinear, we would have to continue the Newton iteration until the residuals $r_j \;\; (j = 1, ..., M)$ are sufficiently small.

**Phase 3: Postprocessing (document the solution)**

- The finite-element solution in element $e$ is given by

$$u = \sum_{j=1}^{n} U_{\mathcal{J}(j,e)} \psi_j(s)$$

#### 1.4.2.5 A 1D example

Here is an example of the finite element solution of the problem P1,

$$\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} = \pm 30 \sin(5.5x), \quad \text{subject to } u(0) = 0 \text{ and } u(1) = \mp 1$$

This problem has the (fish-shaped) exact solution

$$u(x) = \mp \left(\sin(5.5) - 1\right) x - \sin(5.5x).$$

The figure below compares the exact solution against the finite-element solution, obtained from a discretisation wth ten two-node elements. The finite-element solution was computed with `oomph-lib`, using the driver codes discussed in the `Quick Guide` and the example code that illustrates the solution of a `1D Poisson equation` with elements from the `oomph-lib` element library.

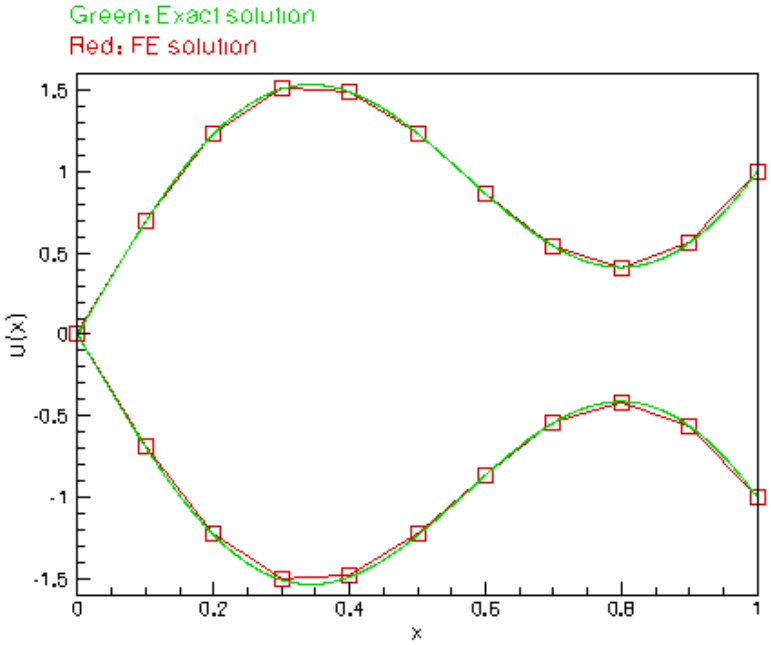


**Figure 1.11 Exact (green) and finite-element solution (red) obtained with ten two-node elements.**

## 1.5 Extension to higher-dimensional problems

Thus far, we have only discussed the implementation of the finite element method for a 1D scalar problem. For such (relatively trivial) problems, the sophisticated "machinery" of Algorithm 10 might appear to be overly complicated. The advantage of the algorithm is that it can easily be generalised to more complicated problems. We will illustrate this by extending the algorithm to higher-dimensional problems and consider the solution of the 2D Poisson problem P2

$$\mathcal{R}(x_i; u(x_i)) = \frac{\partial^2 u(x_i)}{\partial x_i^2} - f(x_i) = 0 \quad \text{subject to} \quad u|_{\partial D} = g,$$

in the non-trivial (fish-shaped) domain shown in this sketch:

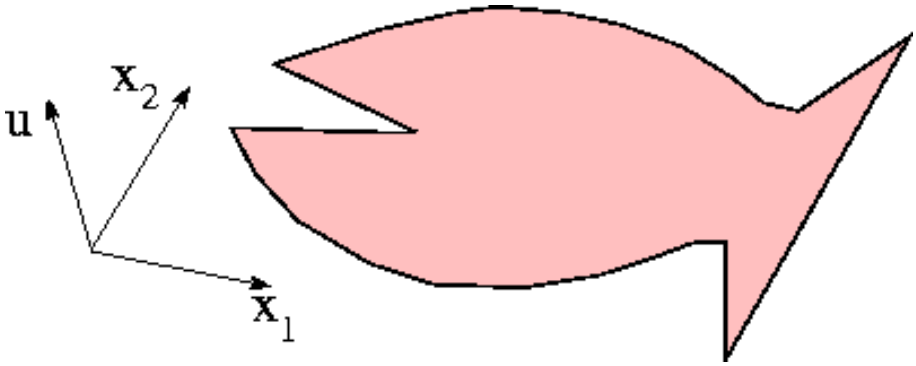


**Figure 1.12 Fish-shaped domain**

A careful inspection of Algorithm 10 shows that only very few steps need to be modified to implement this extension. Specifically, we need to

- modify the mesh generation process so that it
  - decomposes the domain into 2D elements
  - assigns two coordinates for each nodal point.
- define suitable (local) shape functions for the elements
- implement the computation of the elements' residuals and Jacobian matrices which correspond to the (integrated-by-parts) weak form of problem P2,

$$r_k = \int_D \sum_{l=1}^{2} \frac{\partial u(x_i)}{\partial x_l} \frac{\partial \psi_k(x_i)}{\partial x_l} \, dx_1 dx_2 + \int_D f(x_i) \, \psi_k(x_i) \, dx_1 dx_2 = 0.$$

There are many ways to decompose the 2D domain into finite elements. The figure below shows a discretisation using four-node quadrilaterals: a generalisation of the two-node 1D elements used in problem P1:

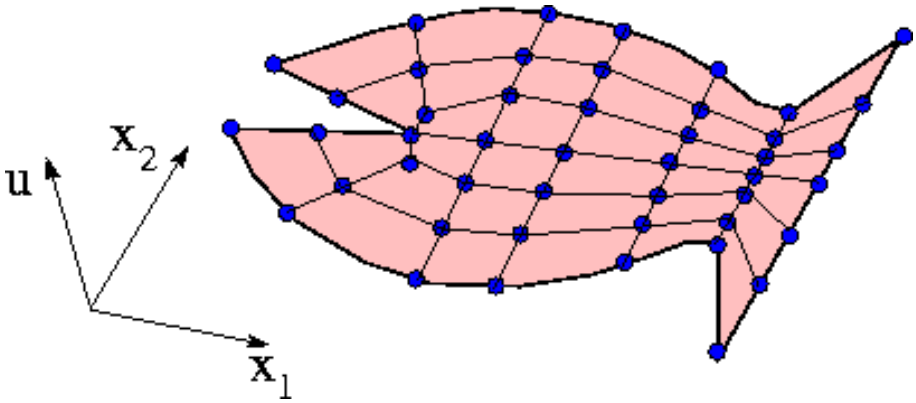


**Figure 1.13 Fish-shaped domain discretised with four-node quadrilateral**

Each node in the mesh now has two coordinates and we denote the $i$-th coordinate of nodal point $j$ by $X_{ij}$ where $i = 1, 2$. We parametrise the 2D elements by two local coordinates, $s_i \in [-1, 1]$ $(i = 1, 2)$, where the local nodes 1,2,3 and 4 are located at $(s_1, s_2) = (-1, -1), (1, -1), (-1, 1)$ and $(1, 1)$, respectively. We define the local shape functions as tensor products of the corresponding 1D linear shape functions,

$$\psi_1(s_1, s_2) = \frac{1}{4}(1 - s_1)(1 - s_2)$$

$$\psi_2(s_1, s_2) = \frac{1}{4}(1 + s_1)(1 - s_2)$$

$$\psi_3(s_1, s_2) = \frac{1}{4}(1 - s_1)(1 + s_2)$$

$$\psi_4(s_1, s_2) = \frac{1}{4}(1 + s_1)(1 + s_2),$$

and note that shape function $\psi_j(s_1, s_2)$ is equal to 1 at local node $j$ and it vanishes at the other local nodes, as required. The sketch below illustrates the corresponding global shape function associated with a node in the interior of the domain:

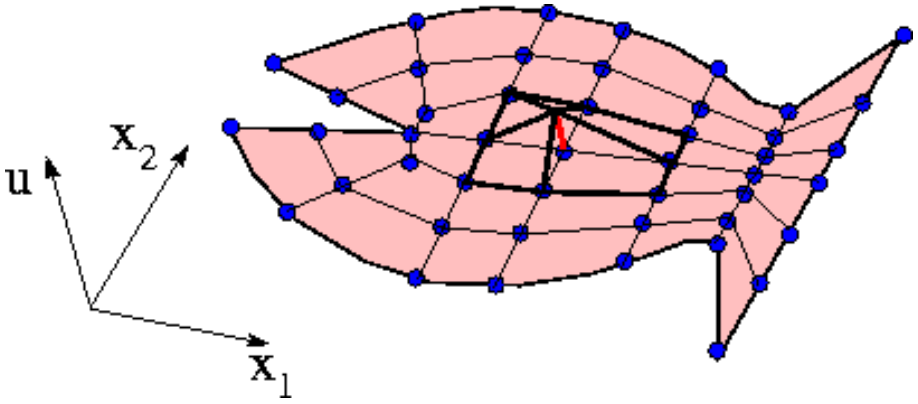


**Figure 1.14 Discretised fish-shaped domain with a single (global) shape**

The solution $u$ and the coordinates $x_i$ in an isoparametric element $e$ can be written as

$$u = \sum_{j=1}^{n} U_{\mathcal{J}(j,e)}\ \psi_j(s_1, s_2)$$

and

$$x_i = \sum_{j=1}^{n} X_{i\mathcal{J}(j,e)}\ \psi_j(s_1, s_2).$$

In 1D, the Jacobian of the mapping between $x$ and $s$ is given by $\widehat{\mathcal{J}} = dx/ds$; in 2D the derivatives transform according to the linear system

$$\begin{pmatrix} \frac{\partial}{\partial s_1} \\ \frac{\partial}{\partial s_2} \end{pmatrix} = \begin{pmatrix} \frac{\partial x_1}{\partial s_1} & \frac{\partial x_2}{\partial s_1} \\ \frac{\partial x_1}{\partial s_2} & \frac{\partial x_2}{\partial s_2} \end{pmatrix} \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \end{pmatrix},$$

and

$$\frac{\partial u}{\partial x_i} = \widehat{\mathcal{J}}_{ij}^{-1} \frac{\partial u}{\partial s_j}$$

where $\widehat{\mathcal{J}}_{ij}^{-1}$ is the inverse of the Jacobian of the mapping (not to be confused with the Jacobian matrix in the Newton method!),

$$\widehat{\mathcal{J}}_{ij} = \frac{\partial x_i}{\partial s_j} = \sum_{k=1}^{n} X_{i\mathcal{J}(k,e)} \frac{\partial \psi_k(s_1, s_2)}{\partial s_j}.$$

If

$$\widehat{\mathcal{J}} = \det \widehat{\mathcal{J}}_{ij}$$

then the integrals over the elements are

$$\int\int_e (...)\ dx_1 dx_2 = \int_{-1}^{1}\int_{-1}^{1} (...)\ \widehat{\mathcal{J}}\ ds_1 ds_2.$$

The integrals over the elements can be evaluated by 2D Gauss rules

$$\int_{-1}^{1}\int_{-1}^{1} \mathcal{F}(s_1, s_2)\ ds_1 ds_2 \approx \sum_{i=1}^{N_{int}} \mathcal{F}(S_{i1}, S_{i2})\ W_i,$$

where, for quadrilateral elements the integration points $S_{i1}, S_{i2}$ $(i = 1, ..., N_{int})$ and weights $W_i$ $(i = 1, ..., N_{int})$ are given by tensor products of the corresponding quantities in the 1D Gauss rules.

### 1.5.1 A 2D example

Here is an example of the finite element solution of problem P2,

$$\sum_{j=1}^{2} \frac{\partial^2 u(x_i)}{\partial x_j^2} = f(x_i), \quad \text{subject to } u|_{\partial D} = 0$$

for

$$f(x_i) = -1$$

in the fish-shaped domain.

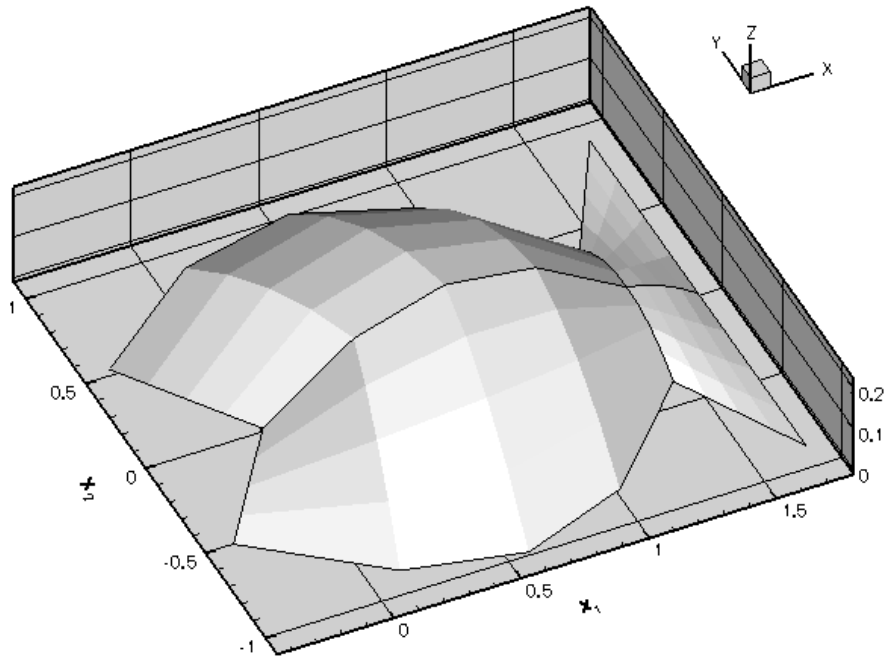


**Figure 1.15 (Adaptive) solution of Poisson's equation in a 2D fish-shaped domain.**

The solution was computed with `oomph-lib`, using a `driver code` discussed elsewhere. Initially, the solution is computed on a very coarse mesh which contains only four nine-node elements. Then the mesh is adaptively refined, guided by an automatic error estimator, and the problem is re-solved. This procedure is repeated several times. Note how the adaptation refines the mesh predominantly near the inward corners where the solution has an unbounded gradient and is therefore (always) under-resolved.

## 1.6 An object-oriented implementation

The steps described in Algorithm 10 can easily be implemented in a procedural language such as Fortran or C. The algorithm already contains a number of well-defined phases, which lend themselves to implementation as subroutines/functions. The various (global) parameters (the number of nodes and elements, etc) and lookup schemes (the relation between local and global equation numbers, the number of unknowns per element, etc.) generated during the setup phase would usually be passed to the subroutines (or functions) via common blocks (or `(bad!) global data`).
We shall now discuss `oomph-lib's` object-oriented implementation of the finite-element method. We begin by examining the "objects" that occur naturally in algorithm 10 and aim to re-distribute all global data so that it becomes member data of an appropriate object. Three "objects" already feature prominently in many parts of our algorithms:

- the Nodes,
- the Elements,
- the Mesh.

In the procedural version of the algorithm, data associated with Nodes is stored in global arrays. For instance, $X_{ij}$ stores the $i$-th coordinate of global node $j$; similarly, $U_j$ stores the nodal value at global node $j$; etc. Nodal positions and values are accessed either directly via their global node number (which identifies Nodes within the Mesh) or indirectly via their local node number within an Element. The latter requires the global lookup scheme $J(j_{local}, e)$ which determines the global node number of local Node $j_{local}$ in Element $e$.
In `oomph-lib's` object-oriented implementation, all nodal data is stored as member data of a Node object. Each Node stores

- a (vector of) coordinates, and
- a nodal value

as private member data. Access to these data is provided by member functions, e.g.

```
Node node;
double x_coord=node.x(0);
double y_coord=node.x(1);
```

[Note the C++ convention of starting indices at 0, rather than 1!]
The Node object does **not** store a node number! This is because node numbers are only used to identify Nodes in specific contexts. For instance, as discussed above, the global node number identifies Nodes within the context of the global Mesh. Instead, we provide the Mesh object with a member function `Mesh::node_pt(...)` which provides pointer-based access to its constituent Nodes:

```
unsigned j;
Mesh mesh;
Node* node_pt=mesh.node_pt(j);
```

Thus, we obtain the $x$-coordinate of global node $j$ via

```
unsigned j;
Mesh mesh;
double x=mesh.node_pt(j)->x(0);
```

Similarly, rather than determining the coordinates of local node $j_{local}$ in element $e$ via the global lookup scheme

$$x_i = X_{iJ(j_{local},e)},$$

we provide each `FiniteElement` with a member function that provides pointer-based access to its constituent Nodes

```
unsigned j_local;
FiniteElement element;
Node* node_pt=element.node_pt(j_local);
```

Similar procedures can be implemented for all other objects. We use the general design principle that all objects store only their "own" intrinsic data and we replace all global lookup schemes by pointer-based access functions. To guide the implementation, we observe the following inter-dependence between our three fundamental objects:

- **The Nodes:** Nodes have positions and values. The nodal values are either "pinned" or "free", in which case they represent unknowns in the problem. The "pinned status" of a Node is indicated by its global equation number, either a unique non-negative number, for unknowns, or -1, for values that are "pinned". Note that this convention makes "-1" a "magic number" – bad practice! In `oomph-lib`, we set the equation number of pinned Nodes to `Node::Is_pinned` – a static data member of the Node class that is, in fact, set to -1.
- **The Elements:** Elements contain Nodes and, within an Element, Nodes are identified by their local node numbers. Elements define their (local) shape functions and compute the elemental residual vector and Jacobian matrix, using data that is intrinsic to the Element. The entries in the Element's residual vector and its Jacobian matrix are labelled by local equation numbers, and each element stores a lookup scheme for the correspondence between local and global equation numbers.
- **The Mesh:** The Mesh consists of Elements and Nodes. The mesh generation process defines the nodal positions and establishes the connectivity between the Nodes. Meshes typically have a number of boundaries and the Mesh defines which Nodes are located on which boundary. Nodes are identified by a global node number within the Mesh.

An important difference between the three fundamental objects is the degree to which their member functions can be implemented in complete generality. For instance, the Node object is completely generic – there is only one type of Node, allowing the Node class to be fully implemented (as a "concrete class", in C++ terminology) in `oomph-lib`. Conversely, although all Meshes have a certain amount of common functionality (e.g. they all provide pointer-based access to their constituent nodes), the implementation of other functions are Mesh-specific. For instance, the Mesh generation process itself (typically performed in the Mesh constructor) depends on the domain geometry, the topology of its elements etc. Such functions must be implemented differently in different Meshes. `oomph-lib`

therefore provides an abstract base class `Mesh`, which defines and implements all functions that are common to all Meshes. The class also defines a number of virtual functions to establish standard interfaces for functions that can only be implemented in a concrete (derived) class, such as `FishMesh`. The same principle applies to Elements which we endow with a three-level inheritance structure:

- The abstract base class `GeneralisedElement` defines the common functionality that *all* Elements must have. All Elements must have member functions that compute the elemental Jacobian matrix and the elemental residual vector. The distinction between `GeneralisedElements` and `FiniteElements` is useful because many elements in `oomph-lib` are not finite elements, and, for instance, do not have nodes.
- The abstract base class `FiniteElement` is derived from `GeneralisedElement` and defines the common functionality that is shared by all finite elements. The `FiniteElement` class implements all functions that are generic for finite elements and specifies the interfaces for any remaining functions that cannot be implemented in complete generality. For instance, all finite elements have shape functions but their implementation depends on the geometry of the element.
- Finally, we have concrete Elements that are derived from `FiniteElement` and implement any virtual functions defined at lower levels; these Elements provide a specific discretisation of a specific PDE. The `QPoissonElement<3,2>` implements, inter alia, the computation of the element residual and the element Jacobian matrix for the 2D Poisson equation, in a nine-node quadrilateral element. [**Note:** In many cases, it is useful to sub-divide this level further by introducing an intermediate class that implements only the Element geometry; motivated by the observation that a specific (geometric) Element type (e.g. a 2D quadrilateral element) can form the basis for many different concrete Elements, e.g. Elements that solve the Poisson, Advection-Diffusion or Navier-Stokes equations. Examples of "geometric Elements" in `oomph-lib` include the `QElements`, a family of 1D line/2D quad/3D brick elements with linear (or {bi/tri}linear), quadratic (or {bi/tri}quadratic),... shape functions.]

  Algorithm 10 can easily be expressed in an object-oriented form because most steps involve operations that act exclusively on the objects' member data. Such operations are implemented as member functions of the appropriate objects.

  The steps in Algorithm 10 that do not fall into this category (the application of boundary conditions, setting of initial conditions and boundary values, etc) are usually problem-dependent and cannot be implemented in generality. However, certain steps (e.g. the solution of the nonlinear algebraic equations by Newton's method) are identical for all problems and should be implemented in a general form. For this purpose, we introduce a fourth fundamental object, **The Problem**, an abstract base class that provides member functions for generic procedures such as

  – setting up the equation numbering scheme
  – solving the equations by Newton's method

  and stores

  – a pointer to the Mesh object
  – a vector of pointers to the unknowns which must be updated during the solution of the nonlinear system by Newton's method.

  Concrete problems, such as the `FishPoissonProblem` discussed above, should be derived from the abstract `Problem` class. Any problem-specific steps can be implemented in member functions of the derived class, often the constructor.

Here is an overview of overall data structure, as required by the Poisson problems described above. The complete data structure implemented in `oomph-lib` contains numerous extensions to permit the solution of time-dependent problems, problems with many nodal values, problems in deforming domains, adaptive mesh refinement and many other features. A complete description of `the data structure` is given elsewhere.

**Data:**

- Stores and provides access to:
  - (pointers to) to values ( = double precision numbers).
  - (pointers to) the global equation numbers for all values.
- Defines and implements the functions that:
  - allow pinning and unpinning of the values.
  - assign global equation numbers to all unknown values.

**Node:**

[derived from **Data**]

- Stores and provides access to:
  - the spatial dimension of the Node.
  - the Eulerian coordinates of the Node.

**GeneralisedElement:**

- Stores and provides access to:
  - the lookup scheme that stores the correspondence between local and global equation numbers.
- Defines interfaces for the (virtual) functions that:
  - compute the element's residual vector and Jacobian matrix [**V1**].
- Defines and implements functions that:
  - return the global equation number for a given local unknown.
  - return the number of degrees of freedom (= the total number of unknowns) in the element.

**FiniteElement:**
[derived from **GeneralisedElement**]

- Stores and provides access to:
  - (pointers to) the `Nodes`.
  - (a pointer to) the spatial integration scheme.
  - the lookup scheme that stores the local equation number of the values stored at the `Nodes`.
- Defines interfaces for the (virtual) functions that:
  - specify the dimension of the element (i.e. the number of local coordinates needed to parametrise its shape – this may differ from the spatial dimension of its `Nodes!`) [**V2**]
  - specify the shape functions and their derivatives with respect to the local coordinates [**V3**]
  - specify the number of nodal values that should be stored at each node, etc. [**V4**]
- Defines and implements functions that:
  - construct `Nodes` of the required spatial dimension, with sufficient storage for the nodal values, etc. (This information is obtained by calls to the virtual functions [V4] that are defined at this level but only implemented for specific, derived elements).
  - calculate the mapping between local and global (Eulerian) coordinates, in terms of the abstract shape functions defined in the virtual functions [V3]
  - provide direct access to the nodal coordinates and the nodal values.
  - assign local equation numbers to the values stored at the `Nodes`.

**Specific Geometric Element:** [e.g. a `QElement`]
[derived from **FiniteElement**]

- Implements functions that:
  - allocate the storage for the (pointers to the) element's `Nodes`.
  - construct the spatial integration scheme and set the pointer defined in `FiniteElement`.
  - define the dimension of the element, i.e. the number of local coordinates required to parametrise its shape. This implements the virtual function [V2] defined in

**Specific Equation:** [e.g. the `PoissonEquations`]
[derived from **FiniteElement**]

- Stores and provides access to:
  - (function pointers to) any source functions, forcing terms, etc. that occur in the governing equations.
  - (pointers to) any parameters in the governing equation.
- Implements functions that:
  - compute the element's residual vector and

**Note:** The subdivision into classes that define the element geometry and the specific equations to be solved is often useful but not compulsory. It is possible (and in some cases probably preferable) to provide the complete implementation for a specific element in a single class. Furthermore, not all elements in `oomph-lib` need to be `FiniteElements`. It is possible to formulate fully functional GeneralisedElements. Such elements are often used to formulate discrete constraints or boundary conditions.

**Mesh:**

- Stores and provides access to:
  - (pointers to) its constituent `GeneralisedElements`.
  - (pointers to) its constituent `Nodes`.
  - (pointers to) the `Nodes` that are located on the `Mesh` boundaries.
- Defines and implements functions that:
  - assign the global equation numbers for all nodal values.
  - execute the `GeneralisedElement::assign_local_eqn_numbers()` function for all constituent GeneralisedElements.

**Specific Mesh:**
[derived from **Mesh**]

- Defines and implements functions that:
  - create the Mesh. This is typically done in the Mesh constructor and involves the following tasks:
    * creating the `Mesh's` constituent `Nodes` and `GeneralisedElements`
    * setting up the lookup schemes that allow direct access to `Nodes` and `FiniteElements` on the domain boundaries.

**Problem:**

- Stores and provides access to:
  - (pointer to) the global `Mesh` which provides ordered access to all `Nodes` and `GeneralisedElements` in the `Problem`.
  - (pointers to) any global `Data`.
  - (pointer to) a `LinearSolver` that will be used in the Newton method.
  - a vector of (pointers to) the `Problem's` degrees of freedom (= double precision numbers).
- Defines and implements functions that:
  - set up the global and local equation numbering schemes.
  - solve the (global) nonlinear system of equations, formally defined by the assembly of the `GeneralisedElement's` residual vectors, using Newton's method.
  - perform a self test of all fundamental objects involved in the problem to check if the `Problem` has been set up properly.
  - allow the entire Problem to be written to disk, in a format that allows restarts.

**Specific Problem:**
[derived from **Problem**]

- Implements functions that:
  - Set up the problem. This is usually done in the `Problem` constructor and involves the following tasks:
    - creation of the `Mesh` object
    - application of boundary conditions
    - completion of the build process for any `GeneralisedElements` that need to be passed some global data, such as function pointers to source functions, etc.
    - setting up the equation numbering scheme.
- Defines and implements functions that:
  - perform parameter studies
  - perform the post-processing of the results.

## 1.7 PDF file

A `pdf version` of this document is available.