Chapter 3

Concepts

The objective of this work is to create a framework to implement multi-disciplinary finite element applications. Before starting, it is necessary to explain some basic concepts of the finite element method itself, multi-disciplinary problems and their solutions, and programming concepts related to its design and implementation.

In this chapter a brief introduction to finite element concepts is given first. Then some basic concepts of coupled systems are described. Finally some programming concepts and techniques are explained.

3.1 Numerical Analysis

In this section a brief introduction to numerical analysis in general will be given and a short description to different numerical methods, their similarities and their differences will be presented.

3.1.1 Numerical Analysis Scheme

There are several numerical analysis methods which are different in their approaches and type of applications. Beside their differences, they rely on a global scheme which make them similar in their overall methodologies and to some extent mixable or interchangeable. This overall scheme consists of three main steps as follows:

Idealization Defining a mathematical model which reflects a physical system. For this reason this step is also referred as *mathematical modeling*. In this step the governing equations of a physical system and its conditions are transformed into some general forms which can be solved numerically. Usually different assumptions are necessary to make this modeling possible. These assumptions make the model different from the physical problem and introduce the *modeling error* in our solutions.

Discretization Converting the mathematical model with infinite number of unknowns, called *degrees of freedom (dof)*, to a finite number of them. While the original model with infinite number of unknown cannot be solved numerically, the resulting *discrete model* with finite number of unknowns can be solve using a numerical approach. What is important to mention

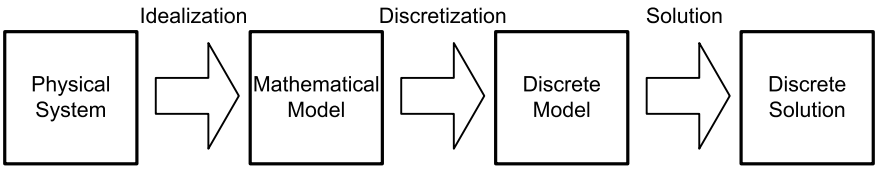

Figure 3.1: Three steps of Numerical analysis process.

here is the approximation involved in this step. This process introduces the *discretization error* to the solution which highly depends on the quality of discretization and the methodology used.

Solution Solving the discrete model and obtaining the dof and other related results. This step introduces the *solution error* coming from the inexactness of using algorithms, numerical precision of machine, or other sources.

Figure 3.1 Shows this global scheme. An important observation here is the existence of different errors and approximations introduced by different concepts. The accumulation of these errors can affect the validity of the results obtained by these methods. For this reason the validity of each step and reduction of the error in each process is one of the main challenges in using a numerical method.

3.1.2 Idealization

The idealization is the mathematical modeling of certain physical phenomena. The main task of this step is finding a proper mathematical model for a given physical problem.

Mathematical models are usually based on different assumptions. This dependency makes them useful for problems in which these assumptions are correct or near to reality. For this reason finding a good mathematical model requires a good knowledge not only about the problem but also about the assumptions of the model.

For example in solving a fluid problem the idealization consists of finding the best fluid model for the certain fluid in the problem. In this case the main questions are: Is this fluid Newtonian? Is it compressible or not? Is it a laminar flow? etc. Depending on these conditions one model can be considered more suitable than others for a certain problem. However the selected model may still introduce certain modeling errors to our solution.

There are different form of mathematical models. Some common ones are:

Strong Form Defines the mathematical model as a system of ordinary or partial differential equations and some corresponding boundary conditions.

Weak Form It expresses the mathematical equations in a particular modified form using a weighted residual approximation.

Variational Form In this form the mathematical model is presented as a functional whose stationary conditions generates the weak form.

Strong Form

As mentioned before the strong form defines the mathematical model as a system of ordinary or partial differential equations and some corresponding boundary conditions. Considering the

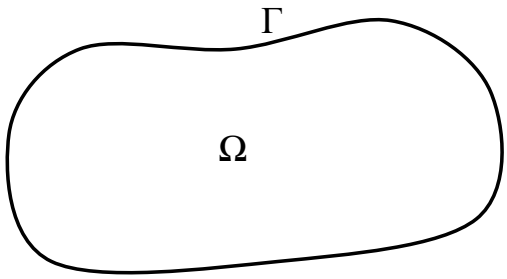

Figure 3.2: The problem of finding u over a domain Ω.

domain Ω with boundary Γ shown in figure 3.2, this form defines the model by a set of equations over the domain and the boundary as follows:

$$\begin{cases} \mathcal{L}(u(x)) = p & x \in \Omega \\ \mathcal{S}(u(x)) = q & x \in \Gamma \end{cases} \tag{3.1}$$

where $u(x)$ is the unknown and $\mathcal{L}$ is the operator applied over the domain Ω and $\mathcal{S}$ represents the operator applied over the boundary Γ. The first equation represents the governing equation over the domain and the second one represents the boundary conditions of this problem. For example, a thermal problem over the domain of figure 3.3 can be modeled with the following strong form:

$$\begin{cases} \nabla^T \mathbf{k} \nabla\theta(x) = Q & x \in \Omega \\ \theta(x) = \theta_\Gamma & x \in \Gamma_\theta \\ q(x) = q_\Gamma & x \in \Gamma_q \end{cases} \tag{3.2}$$

where $\theta(x)$ is the temperature in Ω, q is the boundary flux, Q is the internal heat source, Γ_θ is the boundary with fixed temperature θ_Γ, and Γ_q is the boundary with fixed flux q_Γ.

Weak Form

Let V be a Banach space, Considering the problem of finding the solution $u \in V$ of equation:

$$\mathcal{L}(u) = p \qquad u \in V \tag{3.3}$$

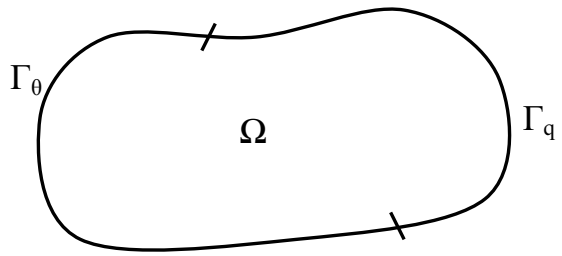

Figure 3.3: A Thermal domain Ω with fixed temperature boundary Γ_θ and fixed flux boundary Γ_q.

It can be verified that this problem is equivalent to finding the solution $u \in V$ such that for all $v \in V$ holds:

$$(\mathcal{L}(u), v) = (p, v) \qquad u \in V, \forall v \in V \tag{3.4}$$

Which is known as the *weak formulation* of the problem. The weak form defines the mathematical model using the weak formulation of the strong form. Now using the following scalar product:

$$(u, v) = \int_\Omega uv d\Omega \tag{3.5}$$

results in the *integral form*, which is the weighted integral representation of the model:

$$\int_\Omega \mathcal{L}(u)v d\Omega = \int_\Omega pv d\Omega \qquad u \in V, \forall v \in V \tag{3.6}$$

This formulation can be applied also to involve the boundary condition. For example, the same problem represented by equation 3.1 can be rewritten as follows:

$$\int_\Omega r(u)w d\Omega + \int_\Gamma \bar{r}(u)\bar{w} d\Gamma = 0 \qquad u \in V, \forall w, \bar{w} \in V \tag{3.7}$$

where r and $\bar{r}$ are the residual functions defined over the domain and the boundary respectively:

$$r(u) = \mathcal{L}(u) - p \tag{3.8}$$
$$\bar{r}(u) = S(u) - q \tag{3.9}$$

and w and $\bar{w}$ are arbitrary weighting functions over the domain and boundary. Usually it is convenient to use integration by parts in order to reduce the maximum order of derivatives in the equations and balance it by applying some derivatives to the weighting functions. Performing integration by parts on equation 3.7 yields:

$$\int_\Omega A(u)B(w) d\Omega + \int_\Gamma C(u)D(\bar{w}) d\Gamma = 0 \qquad u \in V, \forall w, \bar{w} \in V \tag{3.10}$$

Reducing the order of derivatives in A and C respect to r and $\bar{r}$, allows for a lower order of continuity requirement in the choice of the u function. However, now higher continuity for w and $\bar{w}$ is necessary.

Variational Form

The variational form usually comes from some fundamental quantities of the problem like mass, momentum, or energy whose stationary states are of interest. This form defines the mathematical model by a functional in the following form:

$$\Pi(u) = \int_\Omega F(u) d\Omega \tag{3.11}$$

The stationary state of this quantity is required, hence its variation is made equal to zero, which results in the following equation:

$$\delta\Pi(u) = \int_\Omega \delta(F(u)) d\Omega = 0 \tag{3.12}$$

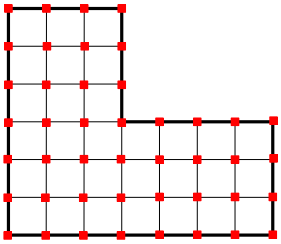

Figure 3.4: A regular domain discretized with a finite difference grid.

Deriving the variational equations from conservation laws is attractive for scientists as presents the same fundamental characteristics of the problem. Finally it is important to mention that the weak form can also be derived from the stationary state of the variational form.

3.1.3 Discretization

The first step of numerical analysis was the definition of a mathematical or *continuous* model corresponding to the real physical problem. In practice the continuous model cannot be solved analytically except for certain domains and conditions, which is the reason for numerical solution. The continuous model has an infinite number of unknowns corresponding to points in the domain and the boundary, and cannot be solved directly using numerical methods. So this second step is necessary for converting the continuous model to a discrete one with a finite number of unknowns which can be solved numerically.

There are several ways to perform this conversion which results in different numerical methods. The appropriate discretization method depends not only on the type of problem but also on the type of mathematical model describing it. A brief description on proper discretization for different models is given as follow.

Discretization of the Strong Form

Discretization of strong form is typically performed using the *finite difference method*. The idea here comes from the numerical calculation of derivatives by replacing them with differences. For example the first derivative of function $f(x)$ can be changed to its discrete form as follows:

$$\frac{df(x)}{dx} \approx \Delta_h^1(f,x) = \frac{1}{h}(f(x+h) - f(x)) \tag{3.13}$$

Where the discretization parameter h is the distance of grid points. This method also can be applied to calculate higher derivatives of a function:

$$\frac{d^n f(x)}{dx^n} \approx \Delta_h^n(f,x) = \sum_{i=0}^{n} (-1)^{n-k} \left(\frac{1}{h}\right)^n \begin{pmatrix} n \\ i \end{pmatrix} f(x+ih) \tag{3.14}$$

The discretization is simply a cartesian grid over domain. Figure 3.4 shows a sample of grid in two dimensional space.

This method has been used practically in many fields and implemented in many applications. Its methodology is simple and also is easy to program. These made it one of the favorite methods in numerical analysis. However this method has its shortcomings. First, it works well for regular

domains, but for arbitrary geometries and boundary conditions encounters difficulties. For example the irregular domain of figure 3.5 (a) can be approximated by the discrete domain shown in figure 3.5 (b). It can be seen easily that this discretization changes the domain boundary for an arbitrary geometry.

(a) (b)

Figure 3.5: An arbitrary geometry and its finite difference discrete model.

Another disadvantage is its approximated solution, which can be obtained only in the grid points hence no information is provided on other points within the grid.

Discretization of the Weak Form

Discretization of the continuous mathematical model consists of transforming the working space to some selected discrete one. Considering the following weak form in the continuous space V:

$$(\mathcal{L}(u), v) = (p, v) \qquad u \in V, \forall v \in V \tag{3.15}$$

This form can be transformed to the discrete space V_h to approximate the solution by $u_h \in V$:

$$(\mathcal{L}(u_h), v_h) = (p, v_h) \qquad u_h \in V, \forall v_h \in V \tag{3.16}$$

As mentioned before, the weak form of the mathematical model can be represented by a weighted integral as:

$$\int_\Omega r(u) w d\Omega + \int_\Gamma \bar{r}(u) \bar{w} d\Gamma = 0 \qquad u \in V, \forall w, \bar{w} \in V \tag{3.17}$$

where r and $\bar{r}$ are the residual functions defined over the domain and the boundary respectively. w and $\bar{w}$ are arbitrary weighting functions over the domain and the boundary. Here selecting a discrete space V_h as our working space results in the following discrete model:

$$\int_\Omega r(u_h) w_h d\Omega + \int_\Gamma \bar{r}(u_h) \bar{w}_h d\Gamma = 0 \qquad u_h \in V_h, \forall w_h, \bar{w}_h \in V_h \tag{3.18}$$

where r and $\bar{r}$ are the residual functions. Equation 3.18 is a *weighted integral of residuals*. So this class of approximations is referred as the *weighted residual methods*. Several well known methods like the *Finite Element Method (FEM)*, *Finite Volume (FV)*, and *Least squares* fitting are subclasses of this method.

It is common to choose a discrete space V_h made by a set of known *trial functions* N_i and define the discrete solution as follows:

$$u_h \approx \sum_{j=1}^{n} a_j N_j \tag{3.19}$$

where a_j are unknown coefficients, N_j are known trial-functions, and n is the number of unknowns. Substituting this in equation 3.18 results:

$$\int_\Omega r(\sum_{j=1}^{n} a_j N_j) w_h d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} a_j N_j) \bar{w}_h d\Gamma = 0 \qquad \forall w_h, \bar{w}_h \in V_h \tag{3.20}$$

with:

$$w_h = \sum_{i=1}^{n} \alpha_i w_i \quad , \quad \bar{w}_h = \sum_{i=1}^{n} \alpha_i \bar{w}_i \tag{3.21}$$

where α_i are arbitrary coefficients, w_i and $\bar{w}_i$ are arbitrary functions, and n is the number of unknowns. Expanding equation 3.20 gives:

$$\sum_{i=1}^{n} \alpha_i \left[\int_\Omega r(\sum_{j=1}^{n} a_j N_j) w_i d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} a_j N_j) \bar{w}_i d\Gamma \right] = 0 \qquad \forall \alpha_i, w_i, \bar{w}_i \in V_h \tag{3.22}$$

As α_i are arbitrary, all components of above sum must be zero in order to satisfy the equation. This results in the following set of equations:

$$\int_\Omega r(\sum_{j=1}^{n} a_j N_j) w_i d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} a_j N_j) \bar{w}_i d\Gamma = 0 \qquad \forall w_i, \bar{w}_i \in V_h \quad , \quad i = 1, 2, 3, ..., n \tag{3.23}$$

There are several set of functions that can be used as weighting functions. Here is a list of some common choices:

Collocation Method Using *Dirac's delta* δ_i as the weighting function:

$$w_i = \delta_i \quad , \quad \bar{w}_i = \delta_i \tag{3.24}$$

where δ_i is a function such that:

$$\int_\Omega f \delta_i d\Omega = f_i \tag{3.25}$$

Substituting this weighting function in our reference equation 3.23 results in the following discrete model:

$$r_i(\sum_{j=1}^{n} a_j N_j) + \bar{r}_i(\sum_{j=1}^{n} a_j N_j) = 0 \quad , \quad i = 1, 2, 3, ..., n \tag{3.26}$$

This method satisfies the equation just in the set of collocation points and gives a set of discrete equations similar to those obtained by the finite difference method.

Subdomain Method It is an extension of the previous method. It uses a weighting function w_i which is identity in a subdomain Ω_i and zero elsewhere:

$$w_i = \begin{cases} I & x \in \Omega_i \\ 0 & x \notin \Omega_i \end{cases} \quad , \quad \bar{w}_i = \begin{cases} I & x \in \Gamma_i \\ 0 & x \notin \Gamma_i \end{cases} \tag{3.27}$$

The discrete model can be obtained by substituting this weighting function in the general weak form of equation 3.23:

$$\int_{\Omega_i} r(\sum_{j=1}^{n} a_j N_j) d\Omega_i + \int_{\Gamma_i} \bar{r}(\sum_{j=1}^{n} a_j N_j) d\Gamma_i = 0 \quad , \quad i = 1, 2, 3, ..., n \tag{3.28}$$

This method provides a uniform approximation in each subdomain and establishes a way to divide the domain into subdomains for solving the problem.

Least Square Method This method uses the governing operator applied to the trial functions as its weighting functions:

$$w_i = \delta \mathcal{L}(N_i) \quad , \quad \bar{w}_i = \delta S(N_i) \tag{3.29}$$

Here is the resulted discrete model by substituting the above weighting function in equation 3.23:

$$\int_{\Omega} r(\sum_{j=1}^{n} a_j N_j) \mathcal{L}(N_i) d\Omega + \int_{\Gamma} \bar{r}(\sum_{j=1}^{n} a_j N_j) S(N_i) d\Gamma = 0 \quad , \quad i = 1, 2, 3, ..., n \tag{3.30}$$

One can verify that the resulting equation is equivalent to minimizing the square of the global residual $\mathcal{R}$ over the domain:

$$\delta \mathcal{R} = 0 \tag{3.31}$$

where:

$$\mathcal{R} = \int_{\Omega} r^2(u_h) d\Omega + \int_{\Gamma} \bar{r}^2(u_h) d\Gamma = 0 \tag{3.32}$$

Galerkin Method It uses the trial functions as weighting functions:

$$w_i = N_i \quad , \quad \bar{w}_i = N_i \tag{3.33}$$

Substituting equation 3.33 in equation 3.23 results in the following Galerkin discrete model:

$$\int_{\Omega} r(\sum_{j=1}^{n} a_j N_j) N_i d\Omega + \int_{\Gamma} \bar{r}(\sum_{j=1}^{n} a_j N_j) N_i d\Gamma = 0 \quad , \quad i = 1, 2, 3, ..., n \tag{3.34}$$

This method usually improves the solution process because frequently, but not always, leads to symmetric matrices with some other useful features which makes it a favorite methodology and a usual base for the finite element solution.

Discretization of the Variational Form

The *Rayleigh-Ritz* method is the classical discretization method for the variational form of the continuous model. It also was the first trial-function method. The idea is to approximate the solution u by $\tilde{u}$ defined by a set of trial functions as follows:

$$\tilde{u} = \sum_{i=1}^{n} \alpha_i N_i \tag{3.35}$$

where α_i are unknown coefficients, N_i are known trial-functions, and n is the number of unknowns. Now considering the following continuous model in its variational form:

$$\delta\Pi(u) = \int_{\Omega} \delta(F(u)) d\Omega = 0 \tag{3.36}$$

Inserting the trial function expansion of equation 3.35 into equation 3.36 gives:

$$\delta\Pi(\tilde{u}) = \sum_{i=1}^{n} \frac{\partial \Pi}{\partial \alpha_i} \delta\alpha_i = 0 \tag{3.37}$$

As this equation must be true for any variation $\delta\alpha$, all its component must be equal to zero. This results into the following set of equations:

$$\begin{aligned} \frac{\partial \Pi(\tilde{u})}{\partial \alpha_1} &= \int_{\Omega} \frac{\partial (F(\tilde{u}))}{\partial \alpha_1} d\Omega = 0 \\ \frac{\partial \Pi(\tilde{u})}{\partial \alpha_2} &= \int_{\Omega} \frac{\partial (F(\tilde{u}))}{\partial \alpha_2} d\Omega = 0 \\ &\vdots \\ \frac{\partial \Pi(\tilde{u})}{\partial \alpha_n} &= \int_{\Omega} \frac{\partial (F(\tilde{u}))}{\partial \alpha_n} d\Omega = 0 \end{aligned} \tag{3.38}$$

Mixed Discretization

Sometimes it is useful to mix two or more types of discretization in order to describe complex phenomena or just for simplifying some approximations while keeping a more detailed approach in other aspects of the problem. A typical case of mixing different forms is the modeling of time dependent problems, where one can use a finite difference discretization in time while using a weighted residual discretization in space.

3.1.4 Solution

The last step in the numerical methodology is the solution. This step consists of solving the discrete model using proper algorithms in order to find the main unknowns and also to calculate other additional unknowns of the problem. This process includes:

Calculating Components All components of a discrete model (like derivatives in the finite difference method, integrals in weighted residual or variational methods, etc.) are calculated.

Creating the Global System The discrete model's components are put together in order to create the global system of equations representing the discrete model.

Solving the Global System The global system must be solved to calculate the unknowns of the problem. For some models this leads to a system of linear equations which can be solved using linear solvers. Some algorithms create a diagonal global system and made the solving part completely trivial.

Calculating Additional Results In many problems not only the principal unknown, i.e. displacement in structural problems, are of interest, but also some additional results, like stresses and strains in structural problems, must be calculated.

Iterating In many algorithms some iterations are also needed to determine the unknown or calculate different sets of unknowns. Solving nonlinear problems, calculating time dependent unknowns, and optimization problems are examples of algorithms where iterations are needed.

3.2 Finite Element Method

In the previous section a brief introduction to numerical methods was given. As this thesis work is based on finite element methodology, a brief description of this method and its basic steps are presented.

The finite element method (FEM) in general takes the integral form of the problem and uses piecewise polynomials as its trial functions. There are a wide range of formulations which lead to the FEM but the most used one is the Galerkin method, which is used here to describe the FEM and its basic steps.

3.2.1 Discretization

Considering a continuous problem:

$$\begin{cases} \mathcal{L}(u(x)) = p & x \in \Omega \\ \mathcal{S}(u(x)) = q & x \in \Gamma \end{cases} \tag{3.39}$$

and its integral form as follows:

$$\int_{\Omega} r(u) w d\Omega + \int_{\Gamma} \bar{r}(u) \bar{w} d\Gamma = 0 \qquad u \in V, \forall w, \bar{w} \in V \tag{3.40}$$

where r and $\bar{r}$ are the residual functions defined over the domain and the boundary respectively:

$$r(u) = \mathcal{L}(u) - p \tag{3.41}$$

$$\bar{r}(u) = \mathcal{S}(u) - q \tag{3.42}$$

Defining the discrete finite element space V_h as composition of polynomial functions N_i:

$$x = \sum_{i=1}^{n} \alpha_i^x N_i \tag{3.43}$$

and transforming the equation 3.40 to this space results in:

$$\int_\Omega r(u_h) w_h d\Omega + \int_\Gamma \bar{r}(u_h) \bar{w}_h d\Gamma = 0 \qquad u_h \in V_h, \forall w_h, \bar{w}_h \in V_h \tag{3.44}$$

where:

$$u_h = \sum_{i=1}^{n} \alpha_i N_i \tag{3.45}$$

$$w = \sum_{i=1}^{n} \beta_i N_i \tag{3.46}$$

$$\bar{w} = \sum_{i=1}^{n} \beta_i N_i \tag{3.47}$$

Expanding of equation 3.44 with these definitions results in:

$$\int_\Omega r(\sum_{j=1}^{n} \alpha_j N_j) \sum_{i=1}^{n} \beta_i N_i d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} \alpha_j N_j) \sum_{i=1}^{n} \beta_i N_i d\Gamma = 0 \qquad \alpha, N \in V_h \quad , \quad \forall \beta \in V_h \tag{3.48}$$

The following alternative form is obtained by taking out the β_i from integrals:

$$\sum_{i=1}^{n} \beta_i \left[\int_\Omega r(\sum_{j=1}^{n} \alpha_j N_j) N_i d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} \alpha_j N_j) N_i d\Gamma \right] = 0 \qquad \alpha, N \in V_h \quad , \quad \forall \beta \in V_h \tag{3.49}$$

As β_i are arbitrary, all components of above sum must be zero in order to satisfy the equation. This results in the following set of equations:

$$\int_\Omega r(\sum_{j=1}^{n} \alpha_j N_j) N_i d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} \alpha_j N_j) N_i d\Gamma = 0 \qquad \alpha, N \in V_h \quad , \quad i = 1, 2, 3, ..., n \tag{3.50}$$

An observation here is the equivalence of the discrete form in equation 3.50 with the one in equation 3.34 obtained by the Galerkin method in section 3.1.3.

Node and Degree of Freedom

In the previous section, the general process of converting the continuous integral form in equation 3.40 to its discrete form in equation 3.50 was explained. In the finite element method, each unknown value is referred as a *degree of freedom (dof)* and it is considered to be the finite element solution u_h at a domain point called *node*.

$$\alpha_i = a_i \tag{3.51}$$

where a_i is the approximate solution u_h at node i. Using this assumption the set of equations 3.50 can be rewritten as follows:

$$\int_\Omega r(\sum_{j=1}^{n} a_j N_j)N_i d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} a_j N_j)N_i d\Gamma = 0 \qquad a, N \in V_h \quad , \quad i = 1, 2, 3, ..., n \tag{3.52}$$

This set of equations can be solved to obtain directly the unknowns at each node of the domain. Substituting equation 3.51 into 3.45 results:

$$u_h = \sum_{i=1}^{n} a_i N_i \tag{3.53}$$

which relates the approximated solution u_h over the domain with the nodal values obtained from solving the previous set of equations 3.52. In this way, the approximate solution can be obtained not only at all nodes but also at any other points of the domain.

Shape Functions and Elements

Returning to the equation 3.52 a set of trial functions N_i are necessary to define the problem. In the finite element method these functions are called *shape functions*. The correct definition of the shape functions plays an important role in the correct approximation of the solution and its many important properties.

The discretization introduced in the previous section reduced the infinite number of unknowns in equation 3.40 to a finite number n in the set of equations 3.52. This is a big step in making the model numerically solvable but still is not complete for solving it in practice. The problem comes from the fact that each equation in set of equations 3.52 involves a complete integration over the domain and a complete relation between the unknowns in the equations which makes the solution very costly. To avoid these problems, let us divide the domain Ω into several sub-domains Ω^e as follows:

$$\Omega^1 \cup \Omega^2 \cup ... \cup \Omega^e \cup ... \cup \Omega^m = \Omega \quad , \quad \Omega^1 \cap \Omega^2 \cap ... \cap \Omega^e \cap ... \cap \Omega^m = \emptyset \tag{3.54}$$

where each partition Ω^e is called an *element*. Dividing the integrals in equations 3.52 yields:

$$\sum_{e=1}^{m} \left[\int_{\Omega^e} r(\sum_{j=1}^{n} a_j N_j)N_i d\Omega^e + \int_{\Gamma^e} \bar{r}(\sum_{j=1}^{n} a_j N_j)N_i d\Gamma^e \right] = 0 \quad i = 1, 2, 3, ..., n \tag{3.55}$$

where Γ^e is the part of boundary Γ related to element Ω_e as shown in figure 3.6. Now let us define the shape function N_i as follows:

$$N_i = \begin{cases} N_i^e & x \in \Omega^e \cup \Gamma^e \\ 0 & x \notin \Omega^e \cup \Gamma^e \end{cases} \tag{3.56}$$

where Ω^e is a partition of domain called *element*. Substituting this shape function into equation 3.52 results in the following equation:

$$\sum_{e=1}^{m_i} \left[\int_{\Omega^e} r(\sum_{j=1}^{m_i} a_j N_j^e)N_i^e d\Omega^e + \int_{\Gamma^e} \bar{r}(\sum_{j=1}^{m_i} a_j N_j^e)N_i^e d\Gamma^e \right] = 0 \quad i = 1, 2, 3, ..., n \tag{3.57}$$

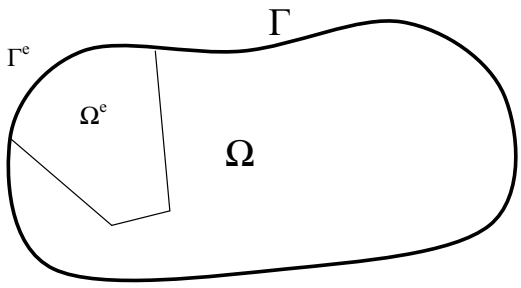

Figure 3.6: An element Ω^e and its boundary Γ^e.

Where m_i is the number of elements containing node i. In this manner the relation between unknowns is reduced to those between the neighbors and in each equation the integration must be performed only over some elements.

Boundary Conditions

Considering the following problem:

$$\begin{cases} \mathcal{L}(u(x)) = p & x \in \Omega \\ \mathcal{S}(u(x)) = q & x \in \Gamma \end{cases} \tag{3.58}$$

and assuming that $\mathcal{L}$ contains at most mth-order derivatives. The boundary condition of such a problem can be divided in two categories: *essential* and *natural* boundary conditions.

The essential boundary conditions $\mathcal{S}_D$ are conditions that contain derivatives with order less equal to $m-1$. These conditions are also called *Dirichlet* conditions (named after Peter Dirichlet). For example the prescribed displacement in structural problems is a Dirichlet condition.

The rest of boundary conditions are considered to be natural boundary conditions $\mathcal{S}_N$. These conditions are also referred as *Neumann* boundary conditions (named after Carl Neumann). For example in a structural problem the boundary tractions are the Neumann condition of the problem.

Applying this division to equation 3.58 results in:

$$\begin{cases} \mathcal{L}(u(x)) = p & x \in \Omega \\ \mathcal{S}_D(u(x)) = q_D & x \in \Gamma_D \\ \mathcal{S}_N(u(x)) = q_N & x \in \Gamma_N \end{cases} \tag{3.59}$$

where $\mathcal{S}_D$ is the Dirichlet condition applied to boundary Γ_D and $\mathcal{S}_N$ is the Neumann condition applied to boundary Γ_N as can be seen in Figure 3.7. Transforming equation 3.59 to its integral form results:

$$\int_\Omega r(u)w d\Omega + \int_{\Gamma_D} \bar{r}_D(u)\bar{w} d\Gamma_D + \int_{\Gamma_N} \bar{r}_N(u)\bar{w} d\Gamma_N = 0 \qquad u \in V, \forall w, \bar{w} \in V \tag{3.60}$$

where:

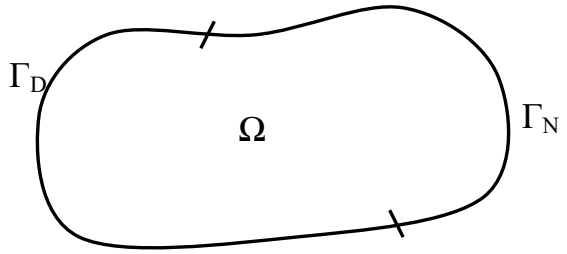

Figure 3.7: A domain and its Dirichlet and Neumann boundaries.

$$r(u) = \mathcal{L}(u) - p \tag{3.61}$$
$$\bar{r}_D(u) = \mathcal{S}_D(u) - q_D \tag{3.62}$$
$$\bar{r}_N(u) = \mathcal{S}_N(u) - q_N \tag{3.63}$$

If the choice of solution u is restricted to functions that satisfy the Dirichlet condition on Γ_D, the integral over the Dirichlet boundary Γ_D can be omitted by restricting the choice of $\bar{w}$ to functions which are zero on Γ_D. Using these restrictions results in:

$$\int_\Omega r(u) w d\Omega + \int_{\Gamma_N} \bar{r}_N(u)\bar{w} d\Gamma_N = 0 \qquad u \in V, \forall w, \bar{w} \in V \tag{3.64}$$
$$u = \bar{u} \qquad x \in \Gamma_D \tag{3.65}$$

where $\bar{u}$ is the solution over the Dirichlet boundary. Converting the above model to its discrete form using the same process described before results in the following discrete equations:

$$\int_\Omega r(\sum_{j=1}^{n} a_j N_j) N_i d\Omega + \int_{\Gamma_N} \bar{r}_N(\sum_{j=1}^{n} a_j N_j) N_i d\Gamma_N = 0 \quad a, N \in V_h \quad , \quad i = 1, 2, ..., n \tag{3.66}$$
$$u = \bar{u} \quad x \in \Gamma_D \tag{3.67}$$

3.2.2 Solution

In this section the global flow of a general finite element solution process will be described and some techniques used to improve the efficiency in practice will be explained.

Calculating Components

Applying the essential condition to equation 3.57 results in:

$$\sum_{e=1}^{m_i} \left[\int_{\Omega^e} r(\sum_{j=1}^{m_i} a_j N_j^e) N_i^e d\Omega^e + \int_{\Gamma_N^e} \bar{r}(\sum_{j=1}^{m_i} a_j N_j^e) N_i^e d\Gamma_N^e \right] = 0 \quad i = 1, 2, ..., n \tag{3.68}$$
$$u = \bar{u} \quad x \in \Gamma_D \tag{3.69}$$

If the differential equations are linear we can write the equation above as follows:

$$\mathbf{Ka} + \mathbf{f} = 0 \tag{3.70}$$

where:

$$\mathbf{K}_{ij} = \sum_{e=1}^{m} \mathbf{K}_{ij}^{e} \tag{3.71}$$

$$\mathbf{f}_{i} = \sum_{e=1}^{m} \mathbf{f}_{i}^{e} \tag{3.72}$$

This step consist of calculating the shape functions and their derivatives in each element and then perform the integration for each element.

The usual technique here is to calculate these components in local coordinates of the element and transform the result to global coordinates. Usually the shape functions are defined in terms of local coordinates and their values and gradients with respect to local coordinates are known. However the elemental matrices contain gradients of shape functions with respect to global coordinates. These gradients can be calculated using the local ones and the inverse of some matrix **J** known as the *jacobian matrix*. Considering the global coordinate x,y,z and elemental local coordinates ξ,η,ζ, it can be seen that the gradients of the shape functions with respect to the local coordinates can be written in terms of the global ones as follows:

$$\begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} = \mathbf{J} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix}$$

where **J** is the jacobian matrix. Now the gradients of the shape functions with respect to the global coordinates can be calculated as follows:

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{bmatrix}$$

After calculating the gradients of the shape functions respect to the global coordinates, we can integrate them over the elements. This can be done by transforming the integration domain from global to local coordinates as follows:

$$\int_{\Omega^e} f d\Omega^e = \int_{\Omega^e} f \; det\mathbf{J} \; d\xi \, d\eta \, d\zeta \tag{3.73}$$

Now all elemental matrices can be calculated using local coordinates and transformed to global coordinates. The usual way to calculate the integrals over the elements is using a *Gaussian Quadrature* method. This method converts the integration of a function over the domain to a weighted sum of function values at certain sample points as follows:

$$\int_{a}^{b} f(x) \, dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{3.74}$$

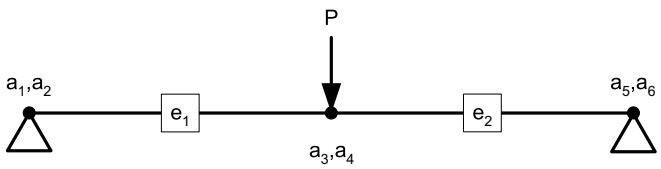

Figure 3.8: A simple beam example with two elements and three nodes.

This method uses nearly half the sample points to achieve the same level of accuracy of other classical quadratures. Thus it is an effective method for calculating elemental integrals in FEM. This method as well as some other integration methods will be explained later in section 5.1.

Creating the Global System

As mentioned earlier for linear differential equations the set of equations 3.68 can be written as a global system of equations in the form of:

$$\mathbf{K}\mathbf{a} = \mathbf{f} \tag{3.75}$$

where:

$$\mathbf{K}_{ij} = \sum_{e=1}^{m} \mathbf{K}_{ij}^{e} \tag{3.76}$$

$$\mathbf{f}_{i} = \sum_{e=1}^{m} \mathbf{f}_{i}^{e} \tag{3.77}$$

However the sums in the equation above must be applied to the corresponding coordinates. Having the elemental matrices and vectors $\mathbf{K}^e$ and $\mathbf{f}^e$, the procedure of putting them together in order to create the global system of equation 3.75 is called *assembly* and consists of finding the position of each elemental component in the global equations system and adding it to the value in its position.

This procedure first assigns a sequential numbering to all dofs. Sometimes its useful to separate the *restricted* dofs, ones with Dirichlet conditions, from others. This can be done easily at the time of assigning indices to dofs. After that the procedure goes element by element and adds their local matrices and vectors to the global equations system using the following assembly operator $\bigsqcup$:

$$\mathbf{K}_{ij} \bigsqcup_{\mathbf{I}^e} \mathbf{K}_{ij}^{e} = \mathbf{K}_{\mathbf{I}_i^e \mathbf{I}_j^e} + \mathbf{K}_{ij}^{e} \tag{3.78}$$

$$\mathbf{f}_{i} \bigsqcup_{\mathbf{I}^e} \mathbf{f}_{i}^{e} = \mathbf{f}_{\mathbf{I}_i^e} + \mathbf{f}_{i}^{e} \tag{3.79}$$

where I^e is the vector containing the global position, which is the index of the corresponding dof, of each row or column. For example considering the beam problem of figure 3.8 with two elements and the following elemental matrices and vectors:

$$\mathbf{K}^{(1)} = \begin{bmatrix} K_{11}^{(1)} & K_{12}^{(1)} & K_{13}^{(1)} & K_{14}^{(1)} \\ K_{21}^{(1)} & K_{22}^{(1)} & K_{23}^{(1)} & K_{24}^{(1)} \\ K_{31}^{(1)} & K_{32}^{(1)} & K_{33}^{(1)} & K_{34}^{(1)} \\ K_{41}^{(1)} & K_{42}^{(1)} & K_{43}^{(1)} & K_{44}^{(1)} \end{bmatrix} \quad , \quad \mathbf{f}^{(1)} = \begin{bmatrix} f_1^{(1)} \\ f_2^{(1)} \\ f_3^{(1)} \\ f_4^{(1)} \end{bmatrix} \tag{3.80}$$

and:

$$\mathbf{K}^{(2)} = \begin{bmatrix} K_{11}^{(2)} & K_{12}^{(2)} & K_{13}^{(2)} & K_{14}^{(2)} \\ K_{21}^{(2)} & K_{22}^{(2)} & K_{23}^{(2)} & K_{24}^{(2)} \\ K_{31}^{(2)} & K_{32}^{(2)} & K_{33}^{(2)} & K_{34}^{(2)} \\ K_{41}^{(2)} & K_{42}^{(2)} & K_{43}^{(2)} & K_{44}^{(2)} \end{bmatrix} \quad , \quad \mathbf{f}^{(2)} = \begin{bmatrix} f_1^{(2)} \\ f_2^{(2)} \\ f_3^{(2)} \\ f_4^{(2)} \end{bmatrix} \tag{3.81}$$

Giving sequential indices to dofs a_1 to a_6, the index vectors I^1 and I^2 of elements e_1 and e_2 will be:

$$I^{(1)} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \quad , \quad I^{(2)} = \begin{bmatrix} 3 \\ 4 \\ 5 \\ 6 \end{bmatrix} \tag{3.82}$$

Finally, assembling the elemental matrices and vectors using the index vectors above results in the following system:

$$\mathbf{Ka} = \mathbf{f} \tag{3.83}$$

with:

$$\mathbf{K} = \begin{bmatrix} K_{11}^{(1)} & K_{12}^{(1)} & K_{13}^{(1)} & K_{14}^{(1)} & 0 & 0 \\ K_{21}^{(1)} & K_{22}^{(1)} & K_{23}^{(1)} & K_{24}^{(1)} & 0 & 0 \\ K_{31}^{(1)} & K_{32}^{(1)} & K_{33}^{(1)} + K_{11}^{(2)} & K_{34}^{(1)} + K_{12}^{(2)} & K_{13}^{(2)} & K_{14}^{(2)} \\ K_{41}^{(1)} & K_{42}^{(1)} & K_{43}^{(1)} + K_{21}^{(2)} & K_{44}^{(1)} + K_{22}^{(2)} & K_{23}^{(2)} & K_{24}^{(2)} \\ 0 & 0 & K_{31}^{(2)} & K_{32}^{(2)} & K_{33}^{(2)} & K_{34}^{(2)} \\ 0 & 0 & K_{41}^{(2)} & K_{42}^{(2)} & K_{43}^{(2)} & K_{44}^{(2)} \end{bmatrix} \tag{3.84}$$

and:

$$\mathbf{f} = \begin{bmatrix} f_1^{(1)} \\ f_2^{(1)} \\ f_3^{(1)} + f_1^{(2)} \\ f_4^{(1)} + f_2^{(2)} \\ f_3^{(2)} \\ f_4^{(2)} \end{bmatrix} \tag{3.85}$$

Another task to be done when building the global system of equations is the application of essential boundary conditions. This can be done easily by eliminating the rows and columns corresponding to restricted dofs from the global matrix and vector and apply their corresponding value to the right hand side. This procedure can be done without reordering of equations but it is more convenient to separate the restricted equations from others in order to simplify the process. Considering the following equation system where the components corresponding to Dirichlet degrees of freedom are separated from others:

$$\begin{bmatrix} \mathbf{K}_{NN} & \mathbf{K}_{DN} \\ \mathbf{K}_{DN} & \mathbf{K}_{DD} \end{bmatrix} \begin{bmatrix} \mathbf{a}_N \\ \mathbf{a}_D \end{bmatrix} = \begin{bmatrix} \mathbf{f}_N \\ \mathbf{f}_D \end{bmatrix} \tag{3.86}$$

where $\mathbf{a}_N$ are unknowns and $\mathbf{a}_D$ are known dofs with Dirichlet boundary condition and $\mathbf{f}_D$ their corresponding boundary unknown. Let us divide the above system in two restricted and not restricted part as follows:

$$[\mathbf{K}_{NN}]\,[\mathbf{a}_N] + [\mathbf{K}_{ND}]\,[\mathbf{a}_D] = [\mathbf{f}_N] \tag{3.87}$$
$$[\mathbf{K}_{DN}]\,[\mathbf{a}_N] + [\mathbf{K}_{DD}]\,[\mathbf{a}_D] = [\mathbf{f}_D] \tag{3.88}$$

Knowing the Dirichlet boundary condition values $\mathbf{a}_D$ let us move them to the right hand side:

$$[\mathbf{K}_{NN}][\mathbf{a}_N] = [\mathbf{f}_N] - [\mathbf{K}_{ND}][\mathbf{a}_D] \tag{3.89}$$

This system of equations can be solved to obtain the unknowns $\mathbf{a}_N$. Considering the previous beam example of figure 3.8. The dofs a_1 and a_5 are restricted. In order to partition the global system let us reorder the dofs as follows:

$$\tilde{\mathbf{a}} = \{a_2, a_3, a_4, a_6, a_1, a_5\} \tag{3.90}$$

which results in the following index vectors:

$$I^{(1)} = \begin{bmatrix} 5 \\ 1 \\ 2 \\ 3 \end{bmatrix} \quad , \quad I^{(2)} = \begin{bmatrix} 2 \\ 3 \\ 6 \\ 4 \end{bmatrix} \tag{3.91}$$

Now let us assemble the global matrix $\mathbf{K}$:

$$\mathbf{K} = \begin{bmatrix} K_{22}^{(1)} & K_{23}^{(1)} & K_{24}^{(1)} & 0 & K_{21}^{(1)} & 0 \\ K_{32}^{(1)} & K_{33}^{(1)} + K_{11}^{(2)} & K_{34}^{(1)} + K_{12}^{(2)} & K_{14}^{(2)} & K_{31}^{(1)} & K_{13}^{(2)} \\ K_{42}^{(1)} & K_{43}^{(1)} + K_{21}^{(2)} & K_{44}^{(1)} + K_{22}^{(2)} & K_{24}^{(2)} & K_{41}^{(1)} & K_{23}^{(2)} \\ 0 & K_{41}^{(2)} & K_{42}^{(2)} & K_{44}^{(2)} & 0 & K_{43}^{(2)} \\ K_{12}^{(1)} & K_{13}^{(1)} & K_{14}^{(1)} & 0 & K_{11}^{(1)} & 0 \\ 0 & K_{31}^{(2)} & K_{32}^{(2)} & K_{34}^{(2)} & 0 & K_{33}^{(2)} \end{bmatrix} \tag{3.92}$$

and vector $\mathbf{f}$:

$$\mathbf{f} = \begin{bmatrix} f_2^{(1)} \\ f_3^{(1)} + f_1^{(2)} \\ f_4^{(1)} + f_2^{(2)} \\ f_4^{(2)} \\ f_1^{(1)} \\ f_3^{(2)} \end{bmatrix} \tag{3.93}$$

Finally applying the Dirichlet boundary condition using equation 3.89:

$$\mathbf{K_{NN}} = \begin{bmatrix} K_{22}^{(1)} & K_{23}^{(1)} & K_{24}^{(1)} & 0 \\ K_{32}^{(1)} & K_{33}^{(1)} + K_{11}^{(2)} & K_{34}^{(1)} + K_{12}^{(2)} & K_{14}^{(2)} \\ K_{42}^{(1)} & K_{43}^{(1)} + K_{21}^{(2)} & K_{44}^{(1)} + K_{22}^{(2)} & K_{24}^{(2)} \\ 0 & K_{41}^{(2)} & K_{42}^{(2)} & K_{44}^{(2)} \end{bmatrix} \tag{3.94}$$

and:

$$[\mathbf{R}_N] = [\mathbf{f}_N] - [\mathbf{K}_{ND}][\mathbf{a}_D] = \begin{bmatrix} f_2^{(1)} \\ f_3^{(1)} + f_1^{(2)} \\ f_4^{(1)} + f_2^{(2)} \\ f_4^{(2)} \end{bmatrix} - \begin{bmatrix} K_{21}^{(1)} & 0 \\ K_{31}^{(1)} & K_{13}^{(2)} \\ K_{41}^{(1)} & K_{23}^{(2)} \\ 0 & K_{43}^{(2)} \end{bmatrix} \begin{bmatrix} a_1 \\ a_5 \end{bmatrix} \tag{3.95}$$

$$\begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix} = \begin{bmatrix} f_2^{(1)} - K_{21}^{(1)} a_1 \\ f_3^{(1)} + f_1^{(2)} - K_{31}^{(1)} a_1 - K_{13}^{(2)} a_5 \\ f_4^{(1)} + f_2^{(2)} - K_{41}^{(1)} a_1 - K_{23}^{(2)} a_5 \\ f_4^{(2)} - K_{43}^{(2)} a_5 \end{bmatrix} \tag{3.96}$$

results in the following equation to be solved:

$$\begin{bmatrix} K_{22}^{(1)} & K_{23}^{(1)} & K_{24}^{(1)} & 0 \\ K_{32}^{(1)} & K_{33}^{(1)} + K_{11}^{(2)} & K_{34}^{(1)} + K_{12}^{(2)} & K_{14}^{(2)} \\ K_{42}^{(1)} & K_{43}^{(1)} + K_{21}^{(2)} & K_{44}^{(1)} + K_{22}^{(2)} & K_{24}^{(2)} \\ 0 & K_{41}^{(2)} & K_{42}^{(2)} & K_{44}^{(2)} \end{bmatrix} \begin{bmatrix} a_2 \\ a_3 \\ a_4 \\ a_6 \end{bmatrix} = \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix} \tag{3.97}$$

Another way to apply the Dirichlet conditions is to use a *penalty method*. This applies a big coefficient to the diagonal elements corresponding to the restricted degrees of freedom. This method is easier to program but is less robust than the previous one. The problem is to find the correct coefficient because a high number may cause the system to be ill conditioned and a low one is less realistic.

The global system matrix obtained through the FEM typically has a lot of zeros in it. Holding all its values in a *dense* matrix structure, which stores all elements of matrix, implies a large overhead in memory due to the storage of zero elements. There are several alternative structures which hold a useful portion of matrix for solving. For example a *banded matrix* structure holds a band around the diagonal of a matrix and assumes all element outside the band are zero. There are also several *sparse* matrix structures like: *compressed sparse row (CSR)* which stores the nonzero elements of each row with their corresponding numbers of columns, or *compressed sparse column (CSC)* which is the transposed of compressed sparse row as a column major structure. Another common structure is the *symmetric matrix* structure that uses the symmetry property of the matrix to hold approximately half of the elements and can be combined with a sparse structure to store half of the nonzeros in matrix.

Solving the Global System

In the previous section the global system of equations was prepared and applying the essential conditions made it ready to be solved. This system of equations can be solved using conventional solvers. There are two categories of solvers, *direct solvers* and *iterative solvers*.

Direct solvers try to solve the equation by making the coefficient matrix upper triangular, lower triangular, diagonal, or sometimes decomposing it to upper lower form and calculating the unknowns using this form of matrices. Solvers like *Gaussian elimination* [81], *frontal solution* [23], *LU decomposition* [81] for general matrices, and *Cholesky* [81] for symmetric matrices are examples of this category.

Iterative solvers start with some initial values for the unknown and try to find the correct solution by calculating the residual and minimize it over iterations. *Conjugate gradient (CG)* [90], *Biconjugate gradient (BCG)* [90], *Generalized minimal residual method (GMRES)* [90] are examples of this category of solvers.

Direct solvers are very fast for small systems of equations and are less dependent on the conditioning of the matrix. The only exception is the existence of *pivot*, a zero in diagonal, which needs a special treatment. These solvers are very slow for large systems while the number of operations grows with order $O(N^3)$ where N is the size of system. Algorithms like *multi frontal solution* [36] are used to solve big systems in parallel machines taking advantage of using several processors in parallel. A way to reduce the number of operations needed for solving the problem is to reduce the bandwidth of the system matrix.

Iterative solvers on the contrary are highly dependent on the condition of the system which affects considerably their convergence. Usually for small systems the direct solvers are faster while

for medium to large systems the iterative ones are more suitable, still depending on the condition of system. These methods are also easier to implement and optimize than direct methods.

As mentioned before the solving cost of direct solvers is highly depended on the bandwidth of the system matrix. For this reason a procedure called *reordering* is recommended to reduce the bandwidth of system matrix. This procedure consists of changing the order of rows and columns of the matrix in order to reduce the bandwidth before the solution and then permuting the result back after solving. In practice these algorithms can be applied to renumber the degrees of freedom in optimum way once they are created and then solve the system as usual. The *Cuthill McKee* [90] algorithm is a classical example of these algorithms. Sometimes the reordering process is applied before using iterative solvers to reduce the cache miss produced by the large sparsity of matrix.

Sometimes it is recommended to prepare the system matrix before solving it using an iterative solver. This procedure is called *preconditioning* and consists of transforming the system of equations to and equivalent but better conditioned one for the solution with iterative solvers. *Diagonal* preconditioner [90] for diagonal dominant systems, *Incomplete LU* with tolerance and filling [90] for general nonsymmetric systems, and *Incomplete Cholesky* [90] for symmetric systems are examples of popular preconditioners. Unfortunately, finding the best combination of solver and preconditioner for a certain problem is a question of experience and there is not a single best combination for all problems.

Calculating Additional Results

In a linear problem after solving the global system of equations, the principal results are obtained and in some sense the problem is solved. But in many cases there are some additional results which are of interest and must be calculated. For example in structural analysis the nodal displacements can be obtained by solving the global system of equations. However the stress in elements is also important and has to be calculated. These values usually are calculated using the primary result, i.e. displacements for each element. For example the elemental stress in a structural problem can be calculated using the displacement values $\mathbf{a}^e$, obtained by solving the global system using the following equation [75]:

$$\sigma = \mathbf{DBa}^e - \mathbf{D}\varepsilon_0 + \sigma_0 \tag{3.98}$$

where σ is elemental stress, $\mathbf{D}$ is the elasticity matrix, $\mathbf{B}$ is the strain matrix, ε_0 is the initial strain, and σ_0 is the initial stress of element. However the results obtained by this equation are usually discontinuous over the domain. This means that the stress result for a node from different elements connected to it is different. For this reason different averaging methods are implemented to smooth the discontinuous results. An alternative is to use recovery methods which try to reproduce continuous gradient results with a better approximation [104].

Iterating

One may note that the previous sections where mainly based on linear differential equations. So what has to be done if the problem is not linear? There are several methods to deal with nonlinear problems. Unfortunately these methods cannot obtain the results as simply as before and usually need to perform iterations to find the results. Considering the following nonlinear system of equations:

$$\mathbf{K}(u)\mathbf{u} = \mathbf{f} \tag{3.99}$$

One can plan an iterative solution procedure where each set of unknowns $\mathbf{u}_n$ is used to calculate the system of equations and calculate the next set of unknowns $\mathbf{u}_{n+1}$:

$$\mathbf{K}(u_n)\mathbf{u}_{n+1} = \mathbf{f} \tag{3.100}$$

There are several methods for accelerating the convergence of this procedure. Some examples are *Newton method*, *Modified Newton method*, *Line search* [104], etc. As mentioned before all these methods need iterations over the solution.

3.3 Multi-Disciplinary Problems

The objective of this thesis is creating a framework to deal with multi-disciplinary problems. So before getting any further, it is important to give a general description of these problems and describe some important features of them briefly.

3.3.1 Definitions

There are different definitions for multi-disciplinary problems. A multi-disciplinary solutions is usually defined as solving a *coupled system* of different physical models together. A coupled system is assumed to be a collection of dependent problems put together defining the model.

In this work a multi-disciplinary problem, also called *coupled problem*, is defined as *solving a model which consists of components with different formulations and algorithms interacting together*. It is important to mention that this difference may come not only from the different physical nature of the problems but also from their different type of mathematical modeling or discretization.

A *field* is a subsystem of multi-disciplinary model representing a certain mathematical model. Typical examples are a fluid field and a structure field in a fluid-structure interaction problem. In a coupled system a *domain* is the part of a modeled space governed by a field equation, i.e. a structure domain and a fluid domain.

3.3.2 Categories

The definition given for multi-disciplinary problem includes a wide range of problems with very different characteristics. These problems can be grouped into different categories reflecting some of their aspects affecting the solution procedure. One classification can be made by how different subsystems interact with each other. Another classification can be done reflecting the type of domain interfaces.

Weak and Strong Coupling

One may classify multi-disciplinary problems by the type of coupling between the different subsystems. Consider a problem with two interacting subsystem as shown in figure 3.9.

The problem is calculating the solutions u_1 and u_2 of subsystems S_1 and S_2 under applied forces $F(t)$. There are two types of dependency between the subsystems:

Weak Coupling Also called *one-way* coupling where one domain depends on the other but this can be solved independently. A thermal-structure problem is a good example of this type of coupling. In this problem the material's property of the structure depends on the temperature while the thermal field can be solved independently, assuming the temperature change due to the structural deformation is very small. Figure 3.10 shows this type of coupling.

Strong Coupling Also referred as *two-way* coupling when each system depends on the other and hence none of them can be solved separately. The fluid-structure interaction problems for structures with large deformations fall into this category. In these problems, the structure deforms by the pressure coming from the fluid, and the fluid velocity and pressure depend on the shape of the deformed structure. Figure 3.11 shows this type of coupling.

Interaction Over Boundary and Domain

As mentioned before one classification of multi-disciplinary problems relies on how subsystems interact together. Another classification can be done by looking not on how they interact but where they interact with each other. There are two categories of multi-disciplinary problems using this criteria [104]:

Class I In this category the interaction occurs at the boundary of the domains. For example in a fluid-structure interaction problem the interaction occurs at the boundary of the structure in contact with fluid and vice versa. Figure 3.12 shows an example of this type of problems.

Class II This category include problems where domains can overlap totally or partially. A thermal-fluid problem is a good example of this class of problems where the domain of the fluid and the thermal problem overlap. Figure 3.13 shows an example of this type of problems.

3.3.3 Solution Methods

There are several different approaches for solving multi-disciplinary problems. Finding a suitable approach for each case, highly depends on the category of the problem and the different details of each field specially for time dependent problems. In this section an overview of different methodologies for solving these problems will be discussed.

Sequential Solution of Problems with Weak Coupling

The solving procedure of one-way coupled problems is trivial. Considering the problem of figure 3.10 with two subsystem S_1 and S_2 where S_2 depends on u_1 (the solution of S_1). This problem can be solved easily by solving S_1 first and using its solution u_1 for solving S_2 as shown in figure 3.14. For transient problems this can be done at each time step.

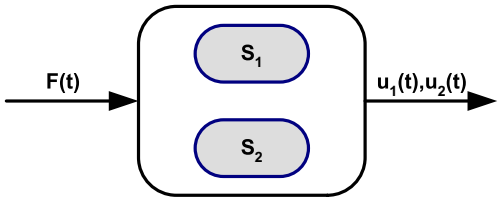

Figure 3.9: A general multi-disciplinary problem with two subsystems.

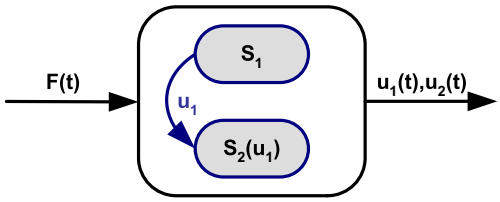

Figure 3.10: A weak coupled system where the subsystem S_2 depends on the solution of subsystem S_1.

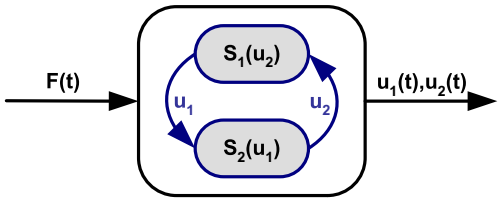

Figure 3.11: A strong coupled system where not only the subsystem S_2 depends on the solution of subsystem S_1 but also subsystem S_1 depends on S_2.

Monolithic Approach

In this approach the interacting fields are modeled together which results in a coupled continuous model and finally a multi-disciplinary element to be used directly. Consider the problem with strong coupling in figure 3.11 where not only subsystem S_2 depends on the solution of subsystem S_1, but also subsystem S_1 depends on S_2:

$$\mathcal{L}_1(u_1, u_2, t) = f_1(t) \tag{3.101}$$
$$\mathcal{L}_2(u_1, u_2, t) = f_2(t) \tag{3.102}$$

Applying the discretization over time and space one can rewrite the above equation in the following form for each time step:

$$\begin{bmatrix} \mathbf{K}_1 & \mathbf{H}_1 \\ \mathbf{H}_2 & \mathbf{K}_2 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1(t) \\ \mathbf{f}_2(t) \end{bmatrix} \tag{3.103}$$

where $\mathbf{K}_1$ and $\mathbf{K}_2$ are the field system matrices corresponding to the field variables and $\mathbf{H}_1$ and $\mathbf{H}_2$ are the field system matrices corresponding to interaction variables. These equations can be solved at each time step in order to calculate the solutions of both fields. Figure 3.15 shows this scheme.

Though this approach seems to be very easy and natural, in practice it encounters difficulties. One problem is the difficulty of the formulation. The multi-disciplinary continuous models, are usually complex by nature and this complexity makes the discretization process a tedious task. However by using computer algebra systems like *Mathematica* [103], and *Maple* [66], the symbolic derivation in this approach becomes more feasible.

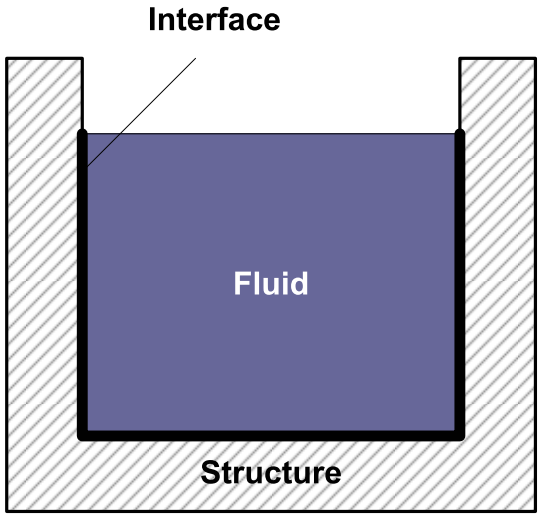

Figure 3.12: A class I fluid-structure interaction problem. The interface, shown by the thick black line, is just at the boundary of the fluid and structure domains.

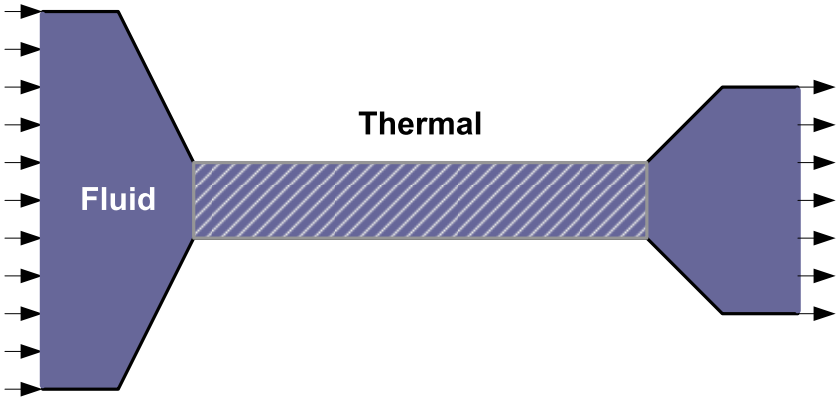

Figure 3.13: A thermal-fluid interaction problem. Here the thermal domain and the fluid domain overlap in the heating pipe part.

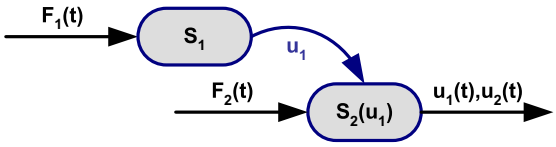

Figure 3.14: Sequential solving of one-way coupled problems.

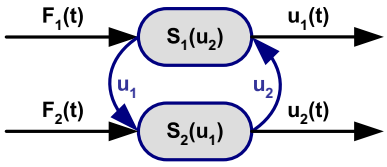

Figure 3.15: Monolithic scheme for solving multi-disciplinary problems.

Another problem is the size and bandwidth of the global system. In this method all fields have to be solved together which make it an expensive approach.

An additional disadvantage is the implementation cost. Implementing the interaction of a certain field with any new field requires the interface matrices $\mathbf{H}_1$ and $\mathbf{H}_2$ to be customized to reflect the new variables. These are not reusable for another interaction of that field. Also any existing codes, for solving each field individually, cannot be reused as they are and in many cases severe modifications are necessary for adapting them to this approach.

In spite of these problems, this approach perfectly models the interaction and results in a more robust and more stable formulation for solving coupled problems.

Staggered Methods

The intention of *staggered methods* is to solve each field separately and simulate the interaction by applying different techniques for transforming variables from one field to another. Some common techniques for staggered methods are described below:

Prediction This technique consists of predicting the value of the dependent variables in the next step. For example in the two-way coupled problem of figure 3.11 a prediction of the variable $u_2^{(n+1)}$ can be used to solve the S_1 subsystem separately. This technique is widely used to decouple different fields in problems with strong coupling. Figure 3.16 shows the prediction's scheme.

There are different methods for predicting the solution for the next step. One common method is the *last-solution predictor* which uses the actual value of the variable as the predicted value for the next step:

$$u_p^{(n+1)} = u^{(n)} \tag{3.104}$$

Another common choice is the prediction by solution gradient which applies a predicted variation to the actual value of variable to obtain the predicted value in the next step:

$$u_p^{(n+1)} = u^{(n)} + \Delta t \dot{u}^{(n)} \tag{3.105}$$

with:

$$\Delta t = t^{(n+1)} - t^{(n)} \tag{3.106}$$

$$\dot{u}^{(n)} = \left(\frac{\partial u}{\partial t}\right)^{(n)} \tag{3.107}$$

Advancing Calculating the next time step of a subsystem using the calculated or predicted solution of other subsystem. Figure 3.17 shows this technique.

Substitution Substitution is a trivial technique which uses the calculated value of one field in another field for solving it separately. Figure 3.18 shows its scheme.

Correction Considering the S_1 subsystem which is solved using the predicted solution $u_{2p}^{(n+1)}$ to obtain $u_1^{(n+1)}$. Advancing the subsystem S_2 using $u_1^{(n+1)}$ results in $u_2^{(n+1)}$. The correction step consist of substituting $u_2^{(n+1)}$ in place of the predicted value $u_{2p}^{(n+1)}$ and solve again S_1 to obtain a better result. Obviously this procedure can be repeated several times. Figure 3.19 shows this procedure.

An staggered method can be planed using the techniques above. For example, returning to the problem of figure 3.11, one can plan the following staggered method:

1. Prediction: $u_p^{(n+1)} = u_2^{(n)} + \Delta t \dot{u}_2^{(n)}$
2. Advancing: $S_1^{(n+1)}(u_p^{(n+1)}) \rightarrow u_1^{(n+1)}$
3. Substitution: $u_1^{(n+1)} = u_1^{(n+1)}$ for S_2
4. Advancing: $S_2^{(n+1)}(u_1^{(n+1)}) \rightarrow u_2^{(n+1)}$

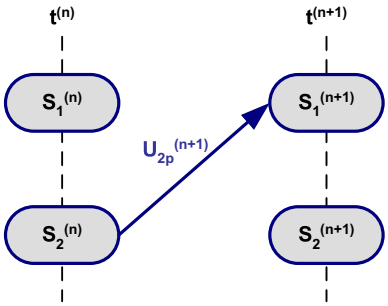

Figure 3.16: The prediction technique consists of predicting the value of the interaction variable for the next step and use it to solve the other field.

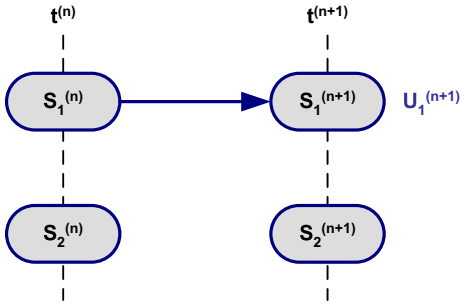

Figure 3.17: The advancing is calculating the solution of the next time step (S_1) using the calculated or predicted solution of other subsystem (S_2).

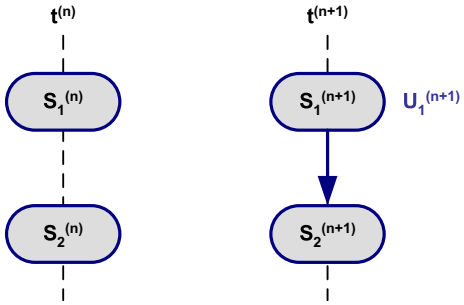

Figure 3.18: The substitution technique consists of substituting the calculated interaction variables in the field S_1 into the field S_2 to calculate it separately.

Figure 3.20 shows this procedure.

More information about staggered methods and their techniques can be found in [41, 42].

Staggered methods use less resources than the monolithic approaches because in each step solve only one part of problem. This can be a great advantage in solving large problems. Also this gives an idea to use the same process for solving large single field problems over several machines in parallel.

The other advantage is the possibility of reusing existing single field codes for solving multi-disciplinary problems almost without modification. This can be done by writing an small program to control the interaction between independent programs for each field. This strategy is referred as *master and slave method* and widely used for solving multi-disciplinary problems.

This approach also enables the use of different discretizations for each field. It also lets each field have its own mesh characteristics. This can be a great added value for solving large and complex coupled problems.

As usual, beside all advantages there are some disadvantages. Staggered methods require a careful formulation to avoid instability and obtain an accurate solution. In general the staggered method is less robust than the monolithic approach and needs more attention in time of modeling and solving.

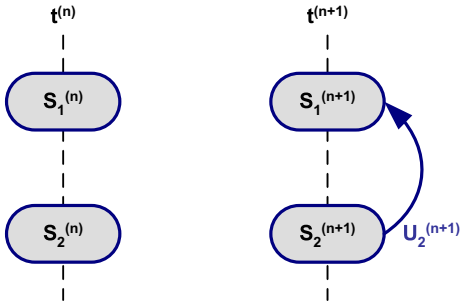

Figure 3.19: The correction consist of replacing the predicted solution with recently calculated one and resolve the subsystem to obtain a better solution.

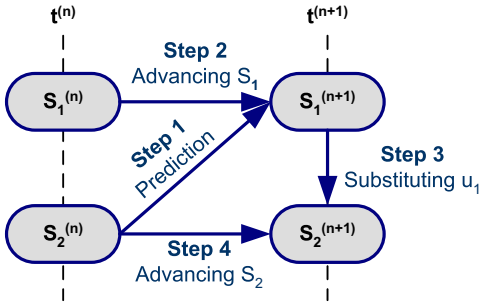

Figure 3.20: An staggered method for solving a coupled system.

3.4 Programming Concepts

Designing and implementing a new software is a hard task. Using proper software engineering solutions and also advanced programming techniques can significantly increase the quality of the program. This chapter describes different software engineering solutions and programming techniques which are useful for designing a finite element program.

3.4.1 Design Patterns

Designing usually consists of several decisions that affect the feature reusability, flexibility, and extendability of the code. However there are several classical problems that appearing during the design and can be solved easily by applying existing *Design Patterns*. Design patterns are some reusable patterns for designing a part of program representing a known problem. In this section a set of patterns that can be used in designing a finite element program are briefly explained.

Strategy Pattern

Strategy pattern defines a family of algorithms by encapsulating each algorithm in one separate class and making them interchangeable via a uniform interface established by their base class.

Figure 3.21 shows this pattern.

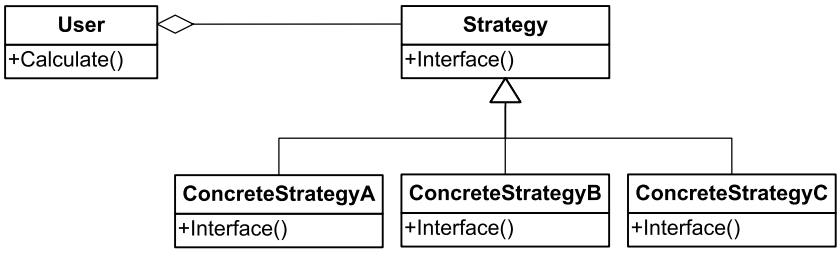

Figure 3.21: Strategy pattern's structure

In this structure `Strategy` declares the interface for all strategies. `User` has a reference to a `Strategy` object and uses the `Strategy` interface to call the algorithm. `User` also may let `Strategy` access its data via an interface. Finally each `ConcreteStrategy` implements an algorithm using the `Strategy` interface.

There are many points in finite element program design where this pattern can be used. Linear solvers, geometries, elements, condition, processes, strategies, etc. Figure 3.22 shows an example of using this pattern for designing a linear solver's structure:

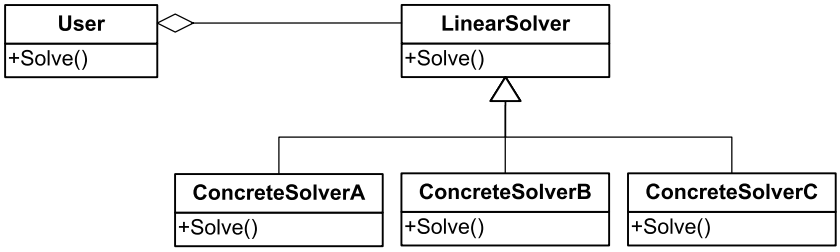

Figure 3.22: Structure designed for linear solvers using strategy pattern

Using this pattern each class derived form `LinearSolver` encapsulates one solving algorithm separately. This encapsulation makes a library easier to extend. The interface is defined by the `LinearSolver` base class and is uniform for all derived solvers. User keeps a pointer to `LinearSolver` base class which may point to any member of the solver family and use the interface of the base class to call different procedures.

Bridge pattern

The *bridge* pattern decouples the abstraction and its implementation in such a way that they can change independently. Figure 3.23 shows the structure of this pattern.

In this pattern `Abstraction` defines the interface for the user and also holds the implementor reference. `AbstractionForm` create a new concept and may also extend the Abstraction interface.

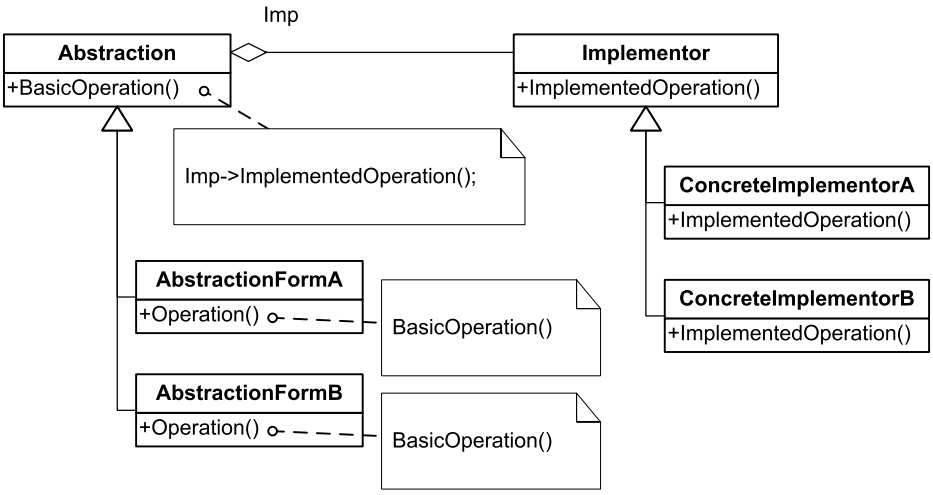

Figure 3.23: Bridge pattern's structure.

`Implementor` defines the interface for implementation part which is used by abstractions. Each `ConcreteImplementor` implements the implementor interface for a concrete case.

Bridge pattern is useful for connecting to concepts with hierarchical structure. In a finite element program this pattern can be used to connect elements to geometries, linear solvers to their reorderer, or to connect iterative solvers to preconditioners.

For example, applying a bridge pattern to an `Element`'s structure results in the structure shown in Figure 3.24.

This pattern lets each `Element` combine its formulation to any `Geometry`.

Composite Pattern

The *Composite* pattern lets users group a set of object in one composite object and treat individual objects and compositions of objects uniformly. Figure 3.25 shows the structure of this pattern.

In this pattern `Component` defines the interface for object operations and also declares the interface for accessing and managing the child components. It may also define an interface for accessing the component's parent in reversible structures. `Leaf` has no children and implements just the operation. It can be used as a basic unit in composition. `Composite` stores its children and implements the child management interface. It also implements its operation by using operations of its children. Finally user can use the component interface to work with all objects in composition uniformly.

This pattern can be used to design processes which can be constructed by a set of processes, geometries with ability of grouping them in a composite one, or even elements grouping different elements in one and mixing formulations.

For example applying this pattern to process results in the structure shown in figure 3.26.

This structure lets users to combine different process in one and use it like any other process.

Template Method Pattern

The *Template Method* pattern defines the skeleton of an algorithm separately and defers some steps to subclasses. In this way template method pattern lets subclasses redefine certain steps

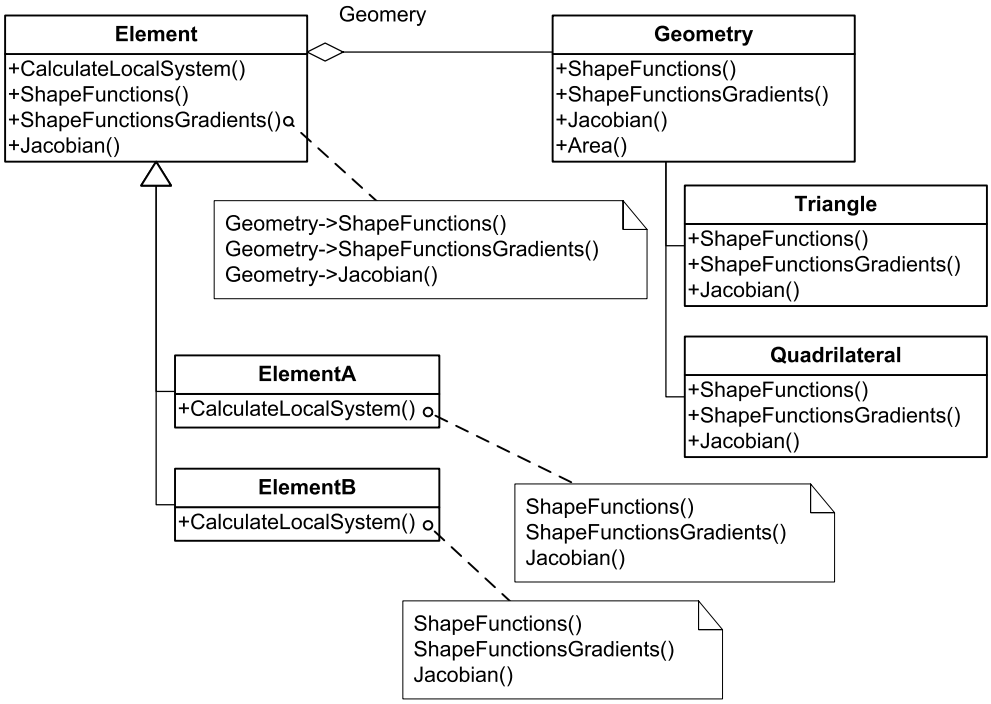

Figure 3.24: Element's structure using bridge pattern.

of an algorithm without changing the algorithm's structure. Figure 3.27 shows structure of this pattern.

In this structure `AbstractClass` defines abstract primitive operations and also implements the skeleton of an algorithm in a template method. `ConcreteClass` implements the primitive operations which will be used by the template method as changed steps of algorithm.

This method is useful in situations when various algorithms differ in some of their steps but not in global. Strategies can be designed using this pattern to provide a category of algorithms changeable by schemes or applying this pattern to linear solver design can make them independent from the matrix and vectors and their operations.

Figure 3.28 shows an example of using this pattern in designing strategies.

Prototype Pattern

The *Prototype* pattern provides a set of prototypes of objects to be created. User clones the prototype to create a new object of that type. System is extendible to any new type whose prototype is available. Figure 3.29 shows the structure of this pattern.

Prototype provides the cloning interface and is the common base class useful to create prototypes list. Each concrete prototype implements the cloning operation for itself. User creates a new object by asking its related prototype to clone itself.

Prototype pattern is very useful for designing an extendible IO. IO can use a prototype of new object and create it without problem. Figure 3.30 shows a use of this pattern in IO for creating elements.

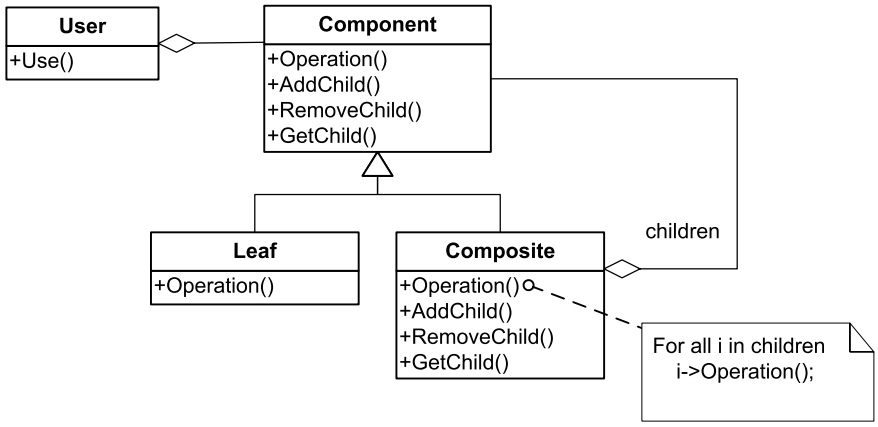

Figure 3.25: Composite pattern.

Interpreter Pattern

The *Interpreter* pattern creates a representation for given grammar and then use it to interpret the langauge. This pattern has simple structure as shown in figure 3.31.

`AbstractExpression` defines the `Interpret` interface for all nodes in syntax tree. `TerminalExpression` represents a terminal symbol in grammar and implements the Interpret method for it. For each terminal symbol in a sentence an instance of it has to be created. `NonterminalExpression` represents a nonterminal symbol in context free grammar. It holds instances of all expressions in its sentence. It also implements the `Interpret` method which usually consist of calling it members `Interpret` method.

Curiously Recursive Template Pattern

The *Curiously Recursive Template (CRT)* pattern, also called as *Barton and Nackman Trick*, consists of giving the derived class to its base class as its template argument. The idea is configuring

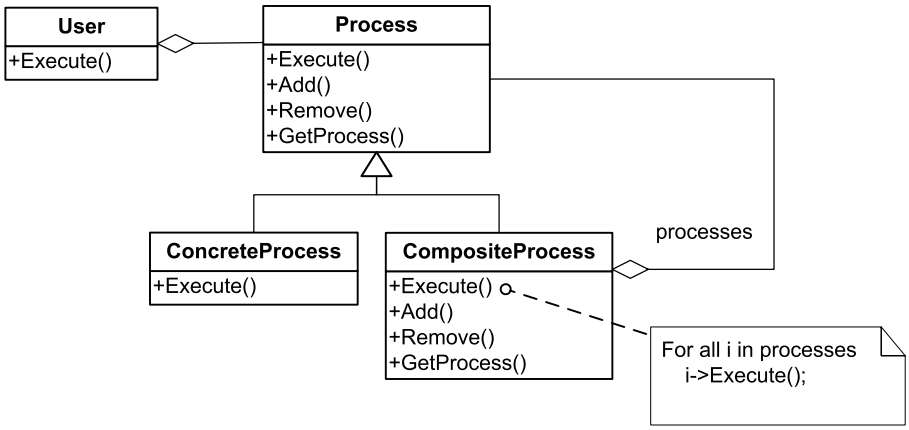

Figure 3.26: Applying composite pattern to processes' structure.

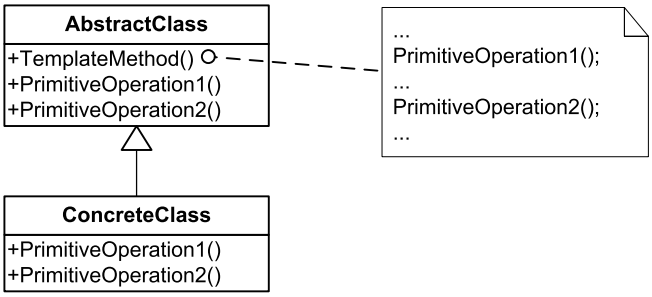

Figure 3.27: Template Method pattern structure.

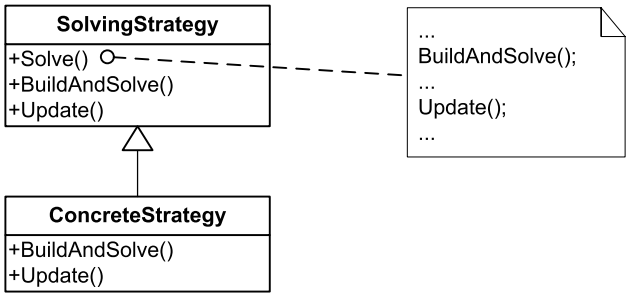

Figure 3.28: Template Method pattern applied to solving strategy.

a base class depending on its derived class and providing a type of static polymorphism which can be much more efficient than usual polymorphism by deriving and overriding the virtual functions. Figure 3.32 shows this pattern.

Using this pattern lets developer customize the base class without losing efficiency in operation. This pattern is more effective when the operation is very simple and the overhead of virtual function calling is considerable. For example a matrix library can use this pattern to let a symmetric matrix derived class change the operators of a base matrix class. In this way methods like access methods, assignments, etc. can be overridden without producing performance overhead. Figure 3.33 shows this pattern applied to the matrix example above.

The patterns described in this section are the ones used in the design of the Kratos. Description for other patterns and also more detailed explanation of patterns mentioned before can be found in [45].

3.4.2 C++ advanced techniques

Performance and memory efficiency are two crucial requirement for finite element programs. It has been shown that an optimized implementation of numerical methods in C++ can provide the same performance of Fortran implementations [100] and usually the inefficiency of the C++ codes comes from the developer's misunderstanding of the language [54]. For this reason it can be helpful to take a look at different techniques used for implementing high performance and efficient numerical algorithms in C++. These techniques are used in different parts of Kratos to improve its efficiency

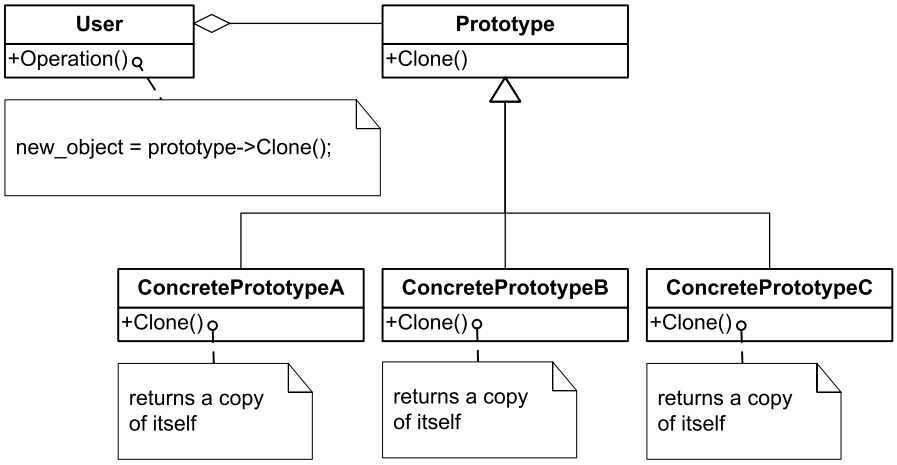

Figure 3.29: Prototype pattern.

while providing a clear and easy to use interface.

Expression Templates

Expression Templates is a technique which is used to pass expressions to a function argument in a very efficient way [97]. For example passing a function to an integration procedure to be integrated. This technique is also used in high performance linear algebra libraries to evaluate vectorial expressions [98]. In this way expressions consisting of operation over matrices and vectors can be evaluated without creating any temporary object and in a single loop.

The idea is to create a template object for each operator and constructing the whole expression by combining these templates and their relative variables. For example, considering the function f as follows:

$$f(x,y) = \frac{1}{x+y}$$

Converting this function to its expression templates form can be done in three steps. First we need expressions to represent the constant and variables as follows:

```
class ConstantExpression {
  double mValue;
public:
  ConstantExpression(double Constant) : mValue(Constant){}
  double Value(){return mValue;}
};

class ReferenceExpression {
  double& mReference;
public:
  ReferenceExpression(double& Variable) : mReference(Variable){}
  double Value(){return mReference;}
};
```

Then some templates are necessary to handle the operators:

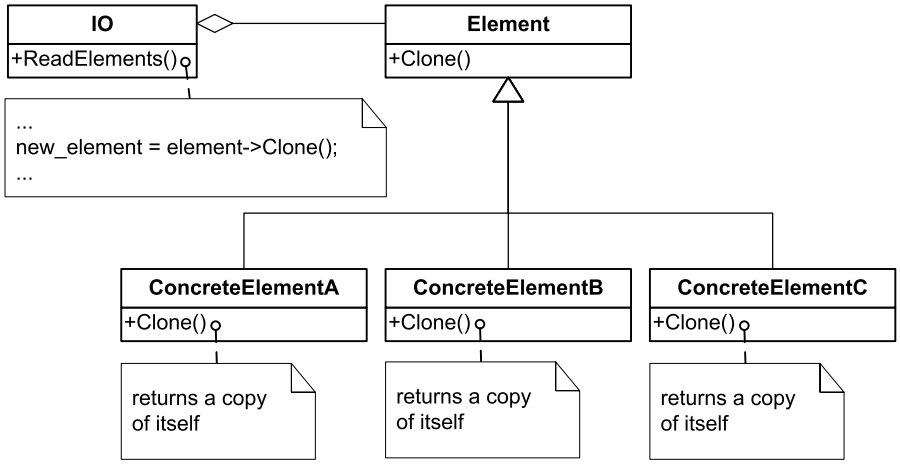

Figure 3.30: Using prototype pattern in IO for creating elements.

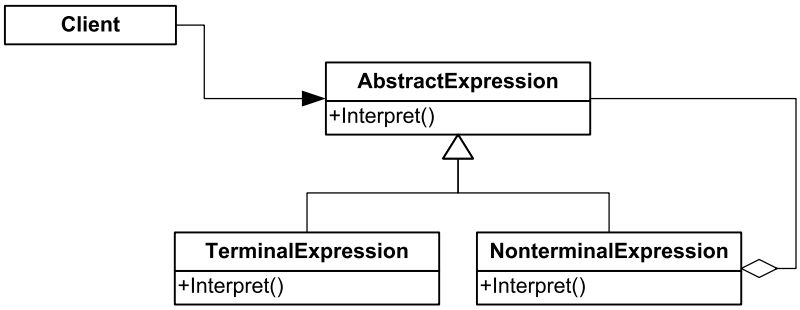

Figure 3.31: Interpreter pattern structure

```
template<class TExpression1, class TExpression2> class
SumExpression {
  TExpression1 mExpression1;
  TExpression2 mExpression2;

public:
  SumExpression(TExpression1 Expression1,
                TExpression2 Expression2) :
                mExpression1(Expression1),
                mExpression2(Expression2){}

  double Value(){return mExpression1.Value() +
                        mExpression2.Value();}

};

template<class TExpression1, class TExpression2> class
```

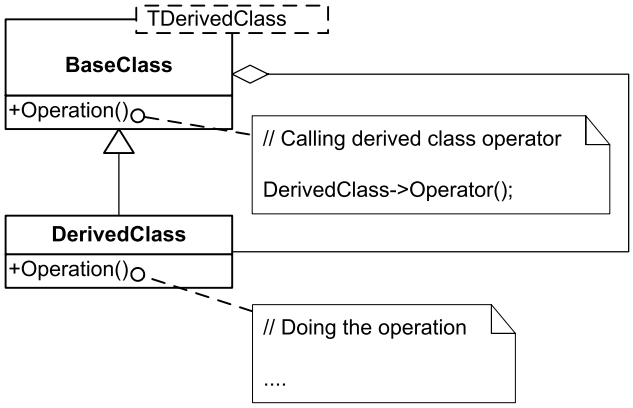

Figure 3.32: Curiously recursive template pattern.

```
DivideExpression {
  TExpression1 mExpression1;
  TExpression2 mExpression2;

public:
  DivideExpression(TExpression1 Expression1,
                   TExpression2 Expression2) :
                   mExpression1(Expression1),
                   mExpression2(Expression2){}

  double Value(){return mExpression1.Value() /
                        mExpression2.Value();}

};
```

And finally the expression template version can be written using previous components. Con-

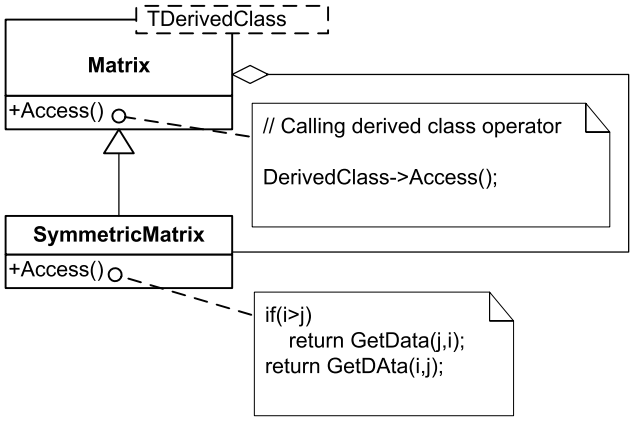

Figure 3.33: Using CRT pattern in matrix structure desing.

verting the $x + y$ expression results in:

```
SumExpression<ReferenceExpression,
              ReferenceExpression>(ReferenceExpression(x),
                                   ReferenceExpression(y));
```

Using above expression the whole function can be written as follows:

```
typedef SumExpression<ReferenceExpression,
                      ReferenceExpression> sum_expression;

typedef DivideExpression<ConstantExpression,
                         sum_expression> expression;

expression f = expression(ConstantExpression(1),
                          sum_expression(ReferenceExpression(x),
                                         ReferenceExpression(y)));
```

Writing these expressions manually is really impractical but fortunately carefully overloaded operators can do this conversion automatically. As mentioned earlier this technique can be used to evaluate matrix and vector expressions without creating temporaries and in one pass. A simple sum operation using overloaded operators over vectors or matrices can result in many redundant loops and overhead. For example considering the following innocent code:

```
// a,b,c and d are vectors
d = a + b + c;
```

This simple code to sum three vectors and assign it to another one using simple overloaded operators can generate a code equivalent to:

```
Vector t1 = b + c; Vector t2 = a + t1; d = t2;
```

In the first step the overloaded operator is used to calculate the sum of two vectors and put them in the temporary vector `t1`. Again the overloaded operator is used to calculate the sum of vector `a` and temporary vector `t1` and the result is stored in another temporary vector `t2`. Finally the second temporary vector is assigned to left hand side vector `d`. It can be seen that this operation is done by performing three loops over all vector elements and creating two temporary vectors which make it very inefficient. Using expression templates can eliminate all this overhead and make it as efficient as a hand coded procedure.

The idea is to overload operators to create the expression without evaluating it. The evaluation of the expression is postponed to the assigning time. The right hand side of this expression can be converted to the following form:

```
class ReferenceExpression {
  Vector& mReference;
public:
  ReferenceExpression(Vector& Variable) : mReference(Variable){}
  double Value(int i){return mReference[i];}
};

template<class TExpression1, class TExpression2> class
SumExpression {
  TExpression1 mExpression1;
  TExpression2 mExpression2;

public:
```

```
  SumExpression(TExpression1 Expression1,
                TExpression2 Expression2) :
                mExpression1(Expression1),
                mExpression2(Expression2){}

  double Value(int i){return mExpression1.Value(i) +
                             mExpression2.Value(i);}

};

typedef SumExpression<ReferenceExpression,
                      SumExpression<ReferenceExpression,
                                    ReferenceExpression> rhs_expression;

d = rhs_expression(ReferenceExpression(a),
                   SumExpression<ReferenceExpression,
                                 ReferenceExpression>(b,c));
```

Now an overloaded assignment operator can complete the procedure:

```
template<class TExpression>
  operator = (Vector& a, TExpression Expression)
  {
    for(int i = 0 ; i < size ; i++)
        a[i] = Expression.Value(i);
  }
```

Passing our `rhs_expression` to this assignment operator results a code equivalent to:

```
for(int i = 0 ; i < size ; i++)
  d[i] = rhs_expression.Value(i);
```

Inlining the first `Value` method and references inside it results the following code:

```
for(int i = 0 ; i < size ; i++)
  d[i] = a[i] + SumExpression<ReferenceExpression,
                              ReferenceExpression>(b,c).Value(i);
```

And inlining the second `Value` method and its all the references results the optimized code:

```
for(int i = 0 ; i < size ; i++)
  d[i] = a[i] + b[i] + c[i];
```

Template Metaprogramming

Templates were added to C++ for a better alternative to existing macros in old C. Eventually they didn't eliminate the usage of macros but gave us much more than anyone could expect. For example, template meta programming. Everything began when Erwin Unruh tricked the compiler to print a list of prime numbers at compile time. This extends the algorithm writing from standard form in C++ to a new form which makes the compiler run the algorithm and results in a new specific algorithm to run.

The Template Metaprogramming technique can be used to create an specialized algorithm at the time of compiling. This technique makes compiler interpret a subset of C++ code in order to generate this specialized algorithm. Different methods are used to simulate different programming

statements, like loops or conditional statements, at compile time. These statements are used to tell the compiler how to generate the code for our specific case.

This technique uses recursive templates to encourage the compiler into making a loop. When a template calls itself recursively the compiler has to make a loop for instantiating templates until a specialized version used as stop rule is reached.Here is an example of how template meta programming can be used to make a compile time power function. First a general template function class is needed to hold the power function class:

```
template<std::size_t TOrder> struct Pow {
  static inline
  double Value(double X)
  {
    // Calculatin result ...
    return result;
  }
};
```

This class takes order as a template argument of the class. Note that `TOrder` must be a positive integer and also known at compile time. So it cannot be used as a normal runtime power function. Now we take a recursive algorithm to compute the n-th power of a value x:

$$x_i = x_{i-1} * x, i = 2, ..., N \tag{3.108a}$$

$$x_1 = x \tag{3.108b}$$

Implementing this using template meta programming is relatively straightforward. Recursive templates can be used here to make the compiler perform a for loop and in each pass add an x multiplication to our code. Stoping the loop after repeating k times will generate a code equivalent to manually written one. First we introduce the recursive part in `Value` method:

```
template<std::size_t TOrder> struct Pow {
  static inline
  double Value(double X)
  {
    return X * Pow<TOrder - 1>::Value(X);
  }
};
```

And the stop rule in equation 3.108b as a specialized template

```
template<> struct Pow<1> {
  static inline
  double Value(double X)
  {
    return X;
  }
};
```

Now the power function is ready to use. Here is an example of calling it to calculate x^4:

```
double x = 2.00; double y = Pow<4>::Value(x); assert(y == 16.00)
```

In the time of Compiling, the compiler will try to inlining the `Value` function which results

```
double y = x * Pow<3>::Value(x)
```

And continuing the recursive calling to order 1

```
double y = x * x * x * Pow<1>::Value(x)
```

At this point the template specialization stop it from going into a infinite loop because `Value` method of `Pow<1>` is not recursive. Compiler tries to inline this `Value` method and generates the following code:

```
double y = x * x * x * x;
```

There is an advantage using this class. If `x` is known at compile time `y` also be calculated in compile time. This can be used to optimize the code for instance, loops can be unrolled when the repeat number is a known value and so on.

Template specialization is can also be used to make compiler simulate conditional statements or switch and cases. These statements can be used to generate different codes according to some conditions. For example an assignment operator for matrices may change its algorithm depending on row or column majority of a given matrix:

```
template<bool c> class Assign{};

class Assign<true> { public:
  template<class Matrix1, class Matrix2>
  Assign(Matrix1& A, Matrix2& B)
  {
    // Assigning row by row;
  }
}


class Assign<false> { public:
  template<class Matrix1, class Matrix2>
  Assign(Matrix1& A, Matrix2& B)
  {
    // Assigning column by column;
  }
}

template<class Matrix2>
  operator = (Matrix2& B)
  {
    Assign<Matrix2::IsRowMajor>(*this, B);
  }
```

In this form the compiler generates an specialized algorithm for each assigning statement.