Petr Krysl

A Pragmatic Introduction to Finite Element Analysis for Structural Engineers

With the Matlab toolbox `SOFEA`

November 2005

Pressure Cooker Press
San Diego

Contents

Part I Introducing the Galerkin method

1 Model of a Taut Wire 3
- 1.1 Deriving the PDE model 3
- 1.2 Balance equation 3
- 1.3 Boundary conditions 4
- 1.4 Boundary conditions (in space) 4
- 1.5 Initial conditions 5
- 1.6 Anything else? 6

2 The method of Mr. Galerkin 7
- 2.1 Residual of the balance equation 7
- 2.2 Integral test of the residual 8
- 2.3 Test function 8
- 2.4 Trial function 9
- 2.5 Manipulation of the residuals 10
- 2.6 Stiffness and mass matrix 11
- 2.7 Piecewise linear basis functions 12
- 2.8 Numerical quadrature 14
- 2.9 Putting it together: system of ODE's 17

3 Introducing the Matlab code 19
- 3.1 Statics 19
- 3.2 Statics: uniform load 19
- 3.3 Free vibration 22
- 3.4 Virtual work principle 22

Part II Thermal analysis

4 Model of Heat Diffusion 25
- 4.1 Balance equation 25
- 4.2 Constitutive equation 27
- 4.3 Boundary conditions 28
- 4.4 Initial condition 29
- 4.5 Summary of the PDE model of heat conduction 29

5 Galerkin method for the model of heat conduction 31
5.1 Weighted residual formulation 31
5.2 Reducing the model dimension 32
5.3 Test and trial functions: basis functions on triangulations 34
5.4 Basis functions on the standard triangle 35
5.5 Discretizing the weighted residual equation 37
5.6 Derivatives of the basis functions; Jacobian 40
5.7 Numerical integration 43
5.8 Conductivity matrix 44
5.9 Surface heat transfer matrix and load 46

6 Steady-state heat diffusion solutions 51
6.1 Steady-state diffusion equation 51
6.2 Thick-walled tube 51
6.3 Orthotropic insert 53
6.4 The T4 NAFEMS Benchmark 56

7 Transient heat diffusion solutions 61
7.1 Discretization in time for transient heat diffusion 61
7.2 Transient diffusion: The T3 NAFEMS Benchmark 63
7.3 Transient cooling in a shrink-fitting application 65

8 Expanding the library of element types 69
8.1 Quadratic triangle T6 69
8.2 Quadratic 1-D element L3 71
8.3 Point element P1 71
8.4 Measuring (integrating) over domains 72
8.5 On the simplex elements 74
8.6 Quadrilateral Q4 75
8.7 Tetrahedron T4 75

9 Convergence and error control 77
9.1 First look at errors 77
9.2 Richardson extrapolation 78
9.3 The T4 NAFEMS Benchmark revisited 78
9.4 Shrink fitting revisited 79

Part III Stress analysis

---

10 Model of elastodynamics 83
10.1 Balance equation 83

References 85

Index 87

Part I

Introducing the Galerkin method

1

Model of a Taut Wire

This chapter will formulate a relatively simple model of a taut string. In the next chapter, we will seek approximate solutions to this model that are obtained with the Galerkin method.

1.1 Deriving the PDE model

Figure 1.1 illustrates an idealization of a taught wire. The wire is under prestressing force, P, assumed to be uniform along the length of the wire. The left hand end is immovably fixed, while the right hand side end is held in a fixture which can slide perpendicularly to the axis of the wire. A transverse force F_L is applied at the movable end. In addition, there may be some distributed force q acting along the length (but we shall ignore gravity). The transverse displacement is a function of both the axial coordinate x and the time t, $w = w(x,t)$. The transverse displacement is assumed to be very small compared to the length of the wire.

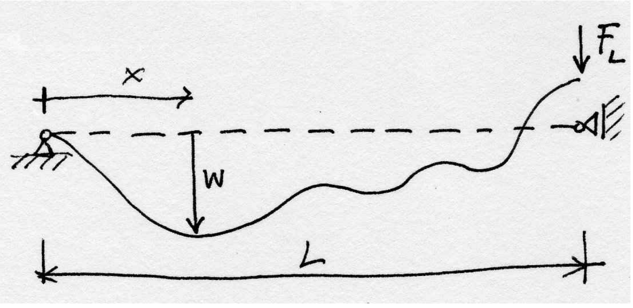


Fig. 1.1. Schematic of taut wire

1.2 Balance equation

Taking a section of length Δx of the wire (see Figure 1.2, collecting all the forces, and equating them to the inertial force (Newton's law), leads to a ***balance equation*** for the taut wire

$$P\frac{\partial^2 w}{\partial x^2} + q = \mu\ddot{w} \ , \tag{1.1}$$

where $\ddot{w} = \frac{\partial^2 w}{\partial t^2}$ is the acceleration.

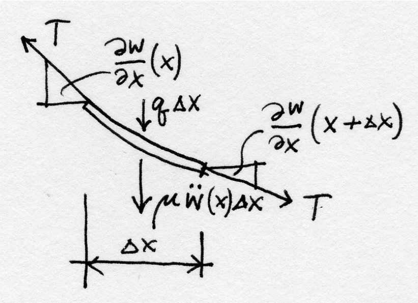

T
$\frac{\partial w}{\partial x}(x)$
$q\Delta x$
$\frac{\partial w}{\partial x}(x+\Delta x)$
$\mu \ddot{w}(x)\Delta x$
T
Δx

Fig. 1.2. The forces acting on a segment of the taut wire

1.3 Boundary conditions

The function w that describes the transverse deflection takes two arguments, x, and t. It is defined on a rectangle shown in Figure 1.3: $0 \leq x \leq L$, and $0 \leq t \leq \bar{t}$. It needs to be determined to satisfy the balance equation (1.1), but that would not completely nail the answer down. We also know something about the solution, namely at the ***boundaries*** of the domain rectangle.

How many pieces of information do we need to know? A reasonable answer is, 'Enough to make the solution unique.' To find the deflection w is going to involve integration, because the balance equation refers to space and time derivatives of w. Using the definitions

$$\begin{aligned} v &= \frac{\partial w}{\partial t} \\ \theta &= \frac{\partial w}{\partial x} \end{aligned}$$

we may rewrite all the balance equation that involves the second derivatives of the function w as a system of first order differential equations

$$\begin{aligned} \frac{\partial \theta}{\partial t} &= \frac{\partial v}{\partial x} \\ T\frac{\partial \theta}{\partial x} + q - \mu\frac{\partial v}{\partial t} &= 0 \end{aligned}$$

For each derivative $\frac{\partial v}{\partial x}$, $\frac{\partial \theta}{\partial x}$, one boundary condition (integration constant) will be needed. Similarly, for each of the time derivatives $\frac{\partial v}{\partial t}$, and $\frac{\partial \theta}{\partial t}$ one boundary condition along the time axis will be required.

1.4 Boundary conditions (in space)

The conditions on w along the edges of the domain rectangle parallel to the time axis are known (for historical reasons) as *the* ***boundary conditions***. (Perhaps also because they are applied along the *physical boundaries* of the structure.)

At the left-hand side end of the wire we are prescribing in general nonzero displacement,

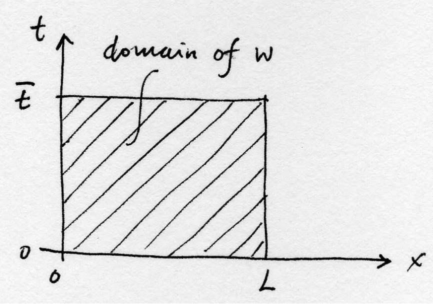


Fig. 1.3. The domain of the deflection function w

$$w(0,t) = \bar{w}_0 . \tag{1.2}$$

As we shall find out, there is a good reason why this kind of condition is commonly called the ***essential boundary condition***.

At the other end the boundary condition is of a different nature. It is also a bit more interesting, as we have to derive it. Again, we take a short section of the wire of length Δx (see Figure 1.4). This time there are terms that are multiplied by Δx, but there are also others which are not. Only the latter survive when we make Δx go to zero.

$$-T\frac{\partial w}{\partial x}(L,0) + F_L = 0 \ . \tag{1.3}$$

This boundary condition is simply the balance of forces at the end of the wire. Boundary conditions of this kind are called ***natural boundary conditions***, and we'll find out presently why.

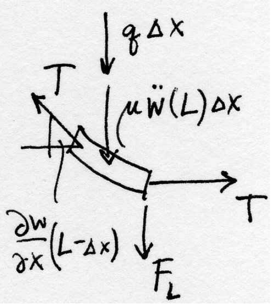


Fig. 1.4. The forces acting on a right hand side segment of the taut wire

1.5 Initial conditions

Along the edges of the domain rectangle that are parallel to the space axis we also apply two conditions. However, as we all are aware, the time direction is special. Therefore, it will probably come to us naturally to expect to know something about the deflection at one point in time, typically at $t = 0$. Because this is the initial point along the time axis, these conditions are known as the ***initial conditions*** (and we need two of them):

$$w(x,0)=\bar{W}(x), \quad \frac{\partial w}{\partial t}(x,0)=\bar{V}(x)\ , \tag{1.4}$$

where $\bar{W}(x)$ (the initial deflection) and $\bar{V}(x)$ (the initial velocity) are known functions.

1.6 Anything else?

The balance equation (1.1), the boundary conditions (1.2) and (1.3), and the initial conditions (1.4) are all we need to fully define what model it is we are trying to find solutions to. It is an ***initial boundary value problem***, and as such it is quite typical of the models with which structural engineers have to deal. In what follows, we shall find out how to formulate an algorithm, the so-called Galerkin finite element method, which will supply an approximate solution to this problem.

2

The method of Mr. Galerkin

We will have to come to grips with the impossibility of satisfying the governing equation exactly with an approximate method. There's going to be an error in the balance equation (which we shall call a residual; another appropriate label might be imbalance). Similarly, the natural (force) boundary condition may not be satisfied exactly, and there is going to be a residual there too.[1]

2.1 Residual of the balance equation

The balance equation (1.1) may be written in the residual form as

$$P\frac{\partial^2 w}{\partial x^2} + q - \mu \ddot{w} = r_B(x,t) \ , \tag{2.1}$$

by simply moving the inertial force on the other side of the equals sign. The residual r_B is identically zero if w is the exact solution. For an approximate solution, the residual r_B varies from point to point, and from time to time, and is in general nonzero.

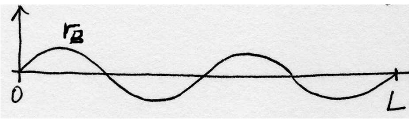


Fig. 2.1. Residual that integrates to zero, but is not identically zero

Checking that the balance residual is identically zero at each point x and each time t does not provide us with anything we can use to talk about approximate solutions: the residual is either zero or it isn't, but how do we measure whether the approximate solution for which the residual is not zero is good?

---

[1] Boris Grigoryevich Galerkin became a teacher of structural mechanics in St. Petersburg Polytechnical Institute in in 1908. Among his contemporaries, also active in St. Petersburg, were I. G. Bubnov, A. N. Krylov, and S. P. Timoshenko, well-known names in the profession. In 1915 Galerkin published an article, in which he put forward an idea of an approximate method to solve differential boundary value problems (he was working on plate and shell models at that time). Around that time Bubnov developed similar variational approach, hence this method is also known as the Bubnov-Galerkin method.

2.2 Integral test of the residual

One possible choice of a quality measure is to integrate the residual over the domain (length of the wire). We could think of the integral

$$\int_0^L r_B(x,t)\,\mathrm{d}x\ . \tag{2.2}$$

as a test: if the residual is identically zero, this integral will also come out zero. However, (2.2) may be zero even when the residual is not identically zero. In other words, if we wanted to prove that the residual corresponds to an exact solution, this would be an incomplete and flawed test. Consider Figure 2.1: the integral (2.2) is zero, but the residual itself may be very large (for instance, when $r_B = A\sin(2\pi nx/L)$, with $n = 1, 2, ...$).

2.3 Test function

A remedy that addresses this blindness of (2.2) to the shape of the residual may be to use a "window" (test) function $\eta(x)$

$$\int_0^L \eta(x) r_B(x,t)\,\mathrm{d}x\ . \tag{2.3}$$

Note that $\eta(x)$ is an arbitrary function. In particular, it could be a function of the shape shown in Figure 2.2, which is certainly going to give a nonzero value for (2.3) (the hatched area at the bottom). Therefore, it correctly indicates that the residual does not correspond to the exact solution. Equation (2.3) is known as the ***weighted residual statement***. Approximate approaches that start from the weighted residual statement are known as weighted residual methods.

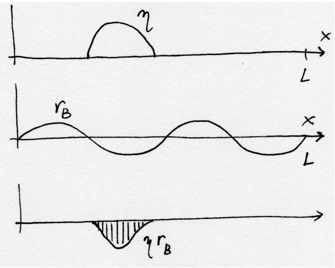


Fig. 2.2. Nonzero a residual which is detected in the integral (2.3)

Equation (2.3) is a reliable way of testing the residual, but computationally it seems hardly less difficult than testing the residual at each point of the domain: equation (2.3) needs to be evaluated for an infinite number of functions η in order to make sure there are no bumps in the residual. The job will still take an infinite time.

Let us contemplate a tangible analogy of what we're trying to do in equation (2.3). Imagine our job is to hold an inflatable balloon in a box, so that it does not jut out anywhere. Use the fingers of one hand to press down on the balloon, so that the balloon is at the top of the box in the spot where it is being held by

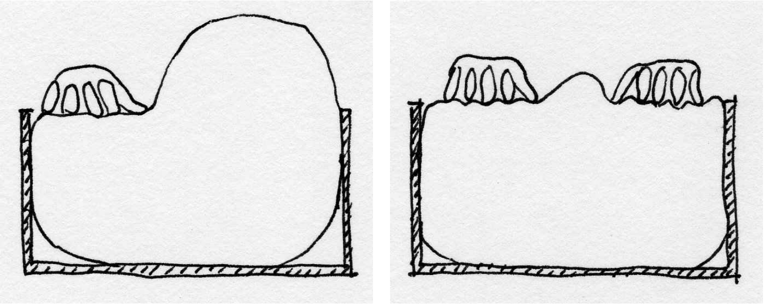


Fig. 2.3. Stuffing a balloon into a box

the finger. If we put down all five fingers, the situation is as shown on the left in Figure 2.3. Each of the fingers may be thought of as a single test function η that pushes down the residual in some spots.

Evidently, the balloon bulges out a little bit in between the fingers, and a lot everywhere else. However, we have the option of pressing down on the balloon with the fingers of our other hand, and if we enrol our friends and relatives, and the chance passersby, we will manage to do a better and better job of stuffing the balloon into the box and holding it so that it does not protrude very much. Indeed, with an infinite number of fingers, we can hold the balloon so that it does not protrude at all.

In this way, we may begin to see how an trial-and-test approximate method may be formulated. Selecting a finite number of suitable functions η_j (fingers), we may be able to keep the residual small (but in general nonzero). By applying larger numbers of test functions, we may hope to be able to reduce the error in the residual. Also, for each η_j, $j = 1, ..., N$, we will make the integral (2.3) vanish

$$\int_0^L \eta_j(x) r_B(x,t)\,\mathrm{d}x = 0 \ , \tag{2.4}$$

which provides us with the means of calculating N coefficients (numbers) from these N equations.

2.4 Trial function

The task of formulating the approximate solution consists really of describing the shape of the deflection w. This can be done in a variety of ways, but for reasons that we shall give later, a piecewise linear representation is a good choice. Figure 2.4 illustrates this concept by showing how the shape may be defined by the N coefficients w_j. The attentive reader will at this point fidget: the piecewise linear shape of the deflection curve is not going to allow us to express the second order derivatives $\partial^2 w/\partial x^2$. At the corners, the first derivatives will be discontinuous, and hence the second derivative will be a spike (so-called Dirac delta function). We can choose either to abandon the piecewise linear shape, and pass a smooth curve through the filled-circle points, or, we could change the rules of the game by getting rid of the second-order derivatives. As we shall presently see, the latter choice is commonly preferred.

In any case, the equations (2.4) may be used to calculate the values of w_j, $j = 1, ..., N$. The function that describes the shape of the approximate solution (with the N free parameters) is known as the ***trial function***. It describes a possible (candidate, trial) shape of the approximate solution; which becomes *the* solution once the values of the free parameters are known.

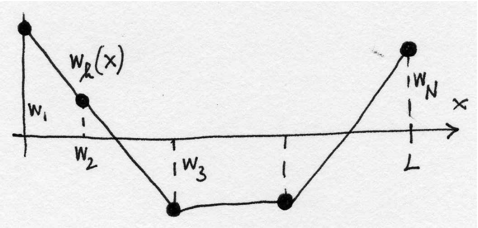


Fig. 2.4. Piecewise linear trial function

2.5 Manipulation of the residuals

We will seek the approximate solution w to satisfy the balance equation in the residual form (2.4), and we'll incorporate the boundary conditions and residual form too. The displacement boundary condition (1.2) will be included in the form of the residual

$$r_w(t) = w(0,t) - \bar{w}_0(t). \tag{2.5}$$

and the natural boundary condition (1.3) will be included as the residual

$$r_F(t) = -P\frac{\partial w}{\partial x}(L,0) + F_L \ . \tag{2.6}$$

Therefore, the approximate solution will be sought from the conditions

$$\begin{aligned} \textstyle\int_0^L \eta_j(x) r_B(x,t)\,\mathrm{d}x &= 0 \\ \xi_w r_w(t) &= 0 \\ \xi_F r_F(t) &= 0 \end{aligned} \tag{2.7}$$

There are no conditions at this point on the trial function w other than smoothness that will guarantee the existence of the integrals in the first equation (2.7).

To reduce the complexity of (2.7), we may immediately realize that at the left-hand side end of the wire, we can quite simply design the trial function to make the residual (2.5) identically zero. This will put another condition on w, and (2.7) may be cast as

$$\begin{aligned} \textstyle\int_0^L \eta_j(x) r_B(x,t)\,\mathrm{d}x &= 0 \\ \xi_F r_F(t) &= 0 \end{aligned} \tag{2.8}$$

where $w(0,t) = \bar{w}_0(t)$, and $w(x,t)$ sufficiently smooth in x.

In this way we managed to reduce the number of residuals, but we will do even better now. By applying integration by parts to the first equation in (2.8), we will be able to reduce the number of residuals further, and furthermore, we will be able to make it much easier to design a trial function by allowing for less smooth functions.

Substituting for the balance residual, we get three terms

$$\begin{aligned}&\int_0^L \eta_j(x) r_B(x,t)\,\mathrm{d}x = \\ &\int_0^L \eta_j(x) P \frac{\partial^2 w}{\partial x^2}(x)\,\mathrm{d}x + \int_0^L \eta_j(x) q(x)\,\mathrm{d}x - \int_0^L \eta_j(x)\mu(x)\ddot{w}(x,t)\,\mathrm{d}x\end{aligned} \tag{2.9}$$

Integration per partes will not affect the second and third term on the right hand side, but for the first term we obtain

$$\int_0^L \eta_j P \frac{\partial^2 w}{\partial x^2}\,\mathrm{d}x = \left[\eta_j P \frac{\partial w}{\partial x}\right]_0^L - \int_0^L \frac{\partial \eta_j}{\partial x} P \frac{\partial w}{\partial x}\,\mathrm{d}x \tag{2.10}$$

The bracketed term is fraught with possibilities. Number one, we may recognize part of the bracket in equation (2.6). In fact, if we propose to satisfy $r_F = 0$ at the right hand side end of the wire ($x = L$) identically, we may simply replace $P\frac{\partial w}{\partial x}$ (which is known there) with F_L. That takes care of the force residual (2.6). Number two, at the left-hand side end of the wire the value of $P\frac{\partial w}{\partial x}$ is unknown, but we have the option of making η_j vanish at $x = 0$. This will burden all the η_j's with a condition, $\eta_j(x = 0) = 0$, but that is something we can afford.

We are in a position to summarize: We have been able to avoid the need to carry the displacement residual (2.5) [eliminated by design of the trial function] and the force residual (2.6) [incorporated into the balance residual– hence, "natural" boundary condition]. Therefore, we will try to find the approximate solution w to satisfy the balance equation in the residual form

$$\eta_j(L)F_L - \int_0^L \frac{\partial \eta_j}{\partial x} P \frac{\partial w}{\partial x}\,\mathrm{d}x + \int_0^L \eta_j q\,\mathrm{d}x - \int_0^L \eta_j \mu \ddot{w}\,\mathrm{d}x = 0, \quad j = 1, ..., N \tag{2.11}$$

where

$$\begin{aligned}&\eta_j(x=0) = 0, \quad \eta_j \in C^0, \quad j = 1, ..., N\\ &w(x=0,t) = \bar{w}_0(t), \quad w \in C^0,\\ &w(x,t=0) \approx \bar{W}(x), \quad \frac{\partial w}{\partial t}(x,t=0) \approx \bar{V}(x).\end{aligned} \tag{2.12}$$

We write for the trial function $w \in C^0$ and similarly for the test functions. This literally means that the functions are continuous, which is a substitute here for a more precise mathematical statement, but which nevertheless ensures that the integrals in (2.11) exist.

The initial conditions need to be suitably approximated, in general we will not be able to satisfy them exactly (which is why we write ≈). Typically, interpolation is used.

2.6 Stiffness and mass matrix

It is time to come back to the choice of the test and trial functions. As advertised in Section 2.4, we have been able to change the requirements on the test and trial function: Their derivatives are now balanced– only the first-order derivatives are needed for both. Therefore, the piecewise linear interpolation function of Figure 2.4 is now an possibility. However, we can still forge ahead while keeping our options open.

Let us make the assumption that the time is fixed $t = \bar{t}$ ($\bar{t}$ some given number). To describe the trial function, we will resort to a common technique in interpolation which is to write the interpolant as a linear combination of basis functions. Therefore, let us assume that the trial function is written as

$$w(x,\bar{t}) = \sum_{i=1}^{N} N_i(x) w_i(\bar{t}) \tag{2.13}$$

where by $w_i(\bar{t})$ we simply mean that the coefficients of the linear combination w_i are actually functions of time, evaluated at the particular time $\bar{t}$. Substituting into (2.12), we obtain

$$\begin{aligned} &\eta_j(L)F_L - \int_0^L \frac{\partial \eta_j}{\partial x} P \sum_{i=1}^{N} \frac{\partial N_i}{\partial x} w_i(\bar{t})\ \mathrm{d}x + \\ &\int_0^L \eta_j q\ \mathrm{d}x - \int_0^L \eta_j \mu \sum_{i=1}^{N} N_i \ddot{w}_i(\bar{t})\ \mathrm{d}x = 0, \quad j = 1, ..., N\ , \end{aligned} \tag{2.14}$$

which may be simplified to

$$\begin{aligned} &\eta_j(L)F_L - \sum_{i=1}^{N} \left(\int_0^L \frac{\partial \eta_j}{\partial x} P \frac{\partial N_i}{\partial x}\ \mathrm{d}x \right) w_i(\bar{t}) + \\ &\int_0^L \eta_j q\ \mathrm{d}x - \sum_{i=1}^{N} \left(\int_0^L \eta_j \mu N_i\ \mathrm{d}x \right) \ddot{w}_i(\bar{t}) = 0, \quad j = 1, ..., N\ , \end{aligned} \tag{2.15}$$

With the definitions

$$K_{ji} = \int_0^L \frac{\partial \eta_j}{\partial x} P \frac{\partial N_i}{\partial x}\ \mathrm{d}x\ , \tag{2.16}$$

where K_{ji} is usually referred to as the ***stiffness matrix***, and

$$M_{ji} = \int_0^L \eta_j \mu N_i\ \mathrm{d}x\ , \tag{2.17}$$

where M_{ji} is the ***mass matrix***, we may write (2.15) as

$$\eta_j(L)F_L - \sum_{i=1}^{N} K_{ji} w_i(\bar{t}) + \int_0^L \eta_j q\ \mathrm{d}x - \sum_{i=1}^{N} M_{ji} \ddot{w}_i(\bar{t}) = 0, \quad j = 1, ..., N\ , \tag{2.18}$$

The matrix equation (2.29) is a system of coupled ordinary differential equations (evaluated at time $\bar{t}$), where the coupling is effected by the matrices K_{ji} and M_{ji}. The linear algebra is going to be much more efficient if the two matrices are *symmetric* and *sparse*.

The first property will follow if we take as the test functions η_j the basis functions themselves, $\eta_j \equiv N_i$. The second property may be achieved if the basis functions N_i are nonzero only on a small subset of the interval $0 \leq x \leq L$.

2.7 Piecewise linear basis functions

Lets us get back to the piecewise linear approximation we advertised for the trial function in Section 2.4. The broken line cannot be represented as a linear combination of linear functions that are all defined on the whole interval $0 \leq x \leq L$ (only two such functions are linearly independent, and these functions cannot represent the corners in the broken line). Therefore, we have to describe the piecewise linear curve interval-by-interval.

A common technique in interpolation is to write the interpolant as a linear combination of basis functions. In one dimension, the piecewise linear basis function is called the ***hat functions***. The six functions that are shown in Figure 2.5, all are examples of hat functions. For reasons that will be discussed later, we would want the hat functions as the constituent parts of a linear combination to be able to reproduce an arbitrary linear function over the whole interval. Because of the way in which we construct the hat functions in Figure 2.5, this property is automatically available.

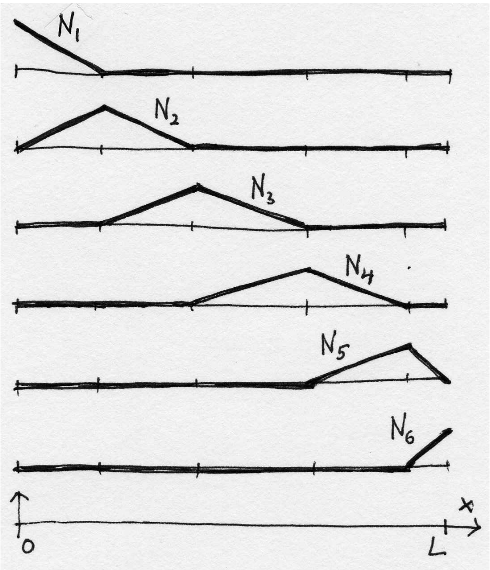


Fig. 2.5. Piecewise linear basis functions

Let us describe the construction of the piecewise linear basis functions. (In this book, the one-dimensional elements with two nodes at the end points are going to be referred to as L2.) First, the length of the wire is divided into disjoint subintervals. These subintervals are the ***finite elements*** for the one-dimensional problem. The end-points of the finite elements are called ***nodes***. Together, the finite elements and the nodes are known as the ***finite element mesh***: see Figure 2.6 (the element numbers are in the boxes; nodes are indicated by filled circles). Since all basis functions are constructed in the same way, we show the procedure for basis function N_3: as shown in the Figure 2.5, it is nonzero over two elements, 2 and 3; zero everywhere else. To be able to write it down over the two adjacent elements, we have to agree on the value of N_3 at node 3 (i.e. $N_3(x_3)$), which is shared by elements 2 and 3. Choosing $N_3(x_3) = 1$ has certain advantages, which will be introduced momentarily. Using the concept of Lagrange interpolation polynomials, we may write the function N_3 within element 2 as

$$N_3(x) = \frac{x - x_2}{x_3 - x_2}, \quad x_2 \leq x \leq x_3$$

and within element 3 as

$$N_3(x) = \frac{x - x_4}{x_3 - x_4}, \quad x_3 \leq x \leq x_4 \ .$$

All the other functions are expressed analogously. Putting them together in a linear combination for the trial function, we write

$$w(x) = \sum_{i=1}^{N} N_i(x) w_i$$

(for simplicity, we omit the time argument). Evaluating $w(x)$ at node k, we obtain

$$w(x_k) = \sum_{i=1}^{N} N_i(x_k) w_i$$

where the crucial expression is $N_i(x_k)$: by definition, the basis function N_k has value $+1$ at x_k, while all other functions $N_i, i \neq k$ are zero at x_k. This property is usually expressed mathematically as

$$N_i(x_k) = \delta_{ik} \ , \tag{2.19}$$

where the symbol δ_{ik} is known as the Kronecker delta

$$\delta_{ik} = \begin{cases} 1, \text{ if } i = k; \\ 0, \text{ otherwise.} \end{cases}$$

Because of this property, the value of $w(x_k)$ is

$$w(x_k) = \sum_{i=1}^{N} N_i(x_k) w_i = \sum_{i=1}^{N} \delta_{ik} w_i = w_k \ ,$$

and the parameters w_i have the physical meaning of the value of the interpolated function at the node i. The w_i's are usually called the ***degrees of freedom***, since being the control parameters of the trial function, they determine the shape of the actual solution from all the possible shapes of the trial function. They are the objects that our numerical method solves for.

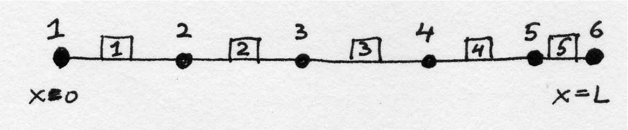


Fig. 2.6. The finite element mesh

2.8 Numerical quadrature

While in the preceding section we described how to compute the basis function N_3 by visiting the adjacent finite elements on which the function was nonzero, the task in an actual finite element program is different. The algorithm there is designed to facilitate numerical evaluation of the integrals in the residual equations. The integrals are calculated element-by-element (the integrands are in general discontinuous from element to element). Therefore, instead of being interested in a single basis function at any point within the mesh, we will rather be striving to calculate

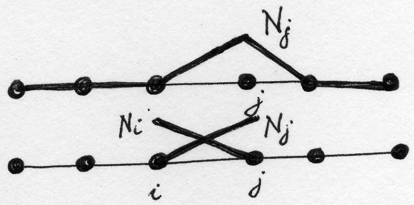


Fig. 2.7. Two different views on how to evaluate basis functions in the finite element mesh: top – compute a single basis function over the whole mesh; bottom – compute all nonzero basis functions over a single element.

the values (and their derivatives) of *all the nonzero basis functions* at a particular point (the quadrature point) within a *single finite element*; see Figure 2.7. In the element-centric view, we would evaluate the functions associated with the nodes at the endpoints of the element. Say for element connecting nodes i, and j, the functions N_i and N_j would be expressed as

$$N_i(x) = \frac{x - x_j}{x_i - x_j}, \quad N_j(x) = \frac{x - x_i}{x_j - x_i} \tag{2.20}$$

It is common practice to develop numerical integration rules on *standard intervals*. Often that will be $-1 \le \xi \le +1$ (line elements in one dimension, quadrilaterals in two dimensions, and bricks in three dimensions all use this interval definition in so-called tensor-product forms). For instance, Simpson's 1/3 rule is given on this interval as

$$\int_{-1}^{+1} f(\xi)\mathrm{d}\xi \approx \frac{1}{3}f(\xi=-1) + \frac{4}{3}f(\xi=0) + \frac{1}{3}f(\xi=+1)$$

In general, a numerical quadrature rule would be written on the standard interval $-1 \le \xi \le +1$ as

$$\int_{-1}^{+1} f(\xi)\mathrm{d}\xi \approx \sum_{k=1}^{M} f(\xi_k)W_k \tag{2.21}$$

where ξ_k are the locations of the integration points, and W_k are their weights. Integrating numerically arbitrary functions over arbitrary intervals is then made possible by a map from the standard interval $-1 \le \xi \le +1$ to the arbitrary interval $a \le x \le b$

$$x = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\xi \tag{2.22}$$

(the first part is the midpoint of the interval $a \le x \le b$, the second part is the departure from the midpoint to either side). Because this map is linear, the relationship between the differentials is constant,

$$\mathrm{d}x = \frac{1}{2}(b-a)\mathrm{d}\xi$$

where the factor $\frac{1}{2}(b-a)$ is called the ***Jacobian determinant***. The Simpson's 1/3 rule is for an arbitrary interval $a \le x \le b$ expressed as

$$\int_{a}^{b} f(x)\mathrm{d}x \approx \frac{1}{2}(b-a)\left[\frac{1}{3}f(\xi=-1) + \frac{4}{3}f(\xi=0) + \frac{1}{3}f(\xi=+1)\right]$$

In general, if the map is

$$x = g(\xi) \tag{2.23}$$

and the numerical quadrature over an arbitrary interval may be written as

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{k=1}^{M} f(\xi_k)\frac{\partial g}{\partial \xi}(\xi_k)W_k \tag{2.24}$$

where $\frac{\partial g}{\partial \xi}(\xi_k)$ is the Jacobian determinant evaluated at the quadrature point ξ_k.

Lets us now look at the integrals in the mass matrix (2.17). The integral will be evaluated over each element individually, and these contributions will be summed together. Therefore, let us consider the integral (2.17)over a single element, connecting nodes i and j

$$\int_{x_i}^{x_j} N_j(x)\mu(x)N_i(x)\ \mathrm{d}x\ .$$

Notice that we are not getting the indexes mixed up: N_j and N_i are the only two basis functions which are nonzero over the interval $x_i \leq x \leq x_j$. Clearly, the function $f(x)$ in equation (2.24) is

$$N_j(x)\mu(x)N_i(x)$$

which means that we have to express the basis functions in terms of ξ ($\mu(x)$ is typically constant over an element). However, we have the map (2.22), which upon substitution into (2.20) yields

$$N_i(\xi) = \frac{\xi - 1}{-2}, \quad N_j(\xi) = \frac{\xi + 1}{+2} \tag{2.25}$$

Basis functions expressed on the standard interval (2.25) are sometimes referred to as being expressed in the parametric coordinates. It is noteworthy that (2.25) are just the Lagrange interpolation polynomials on the standard interval. As we shall see later, writing down the basis functions over a standard shape – a square for general quadrilaterals, a cube for general brick elements, standards triangles or standard tetrahedra for general triangles or general tetrahedra, and so on– is not only convenient, but also highly advisable from the point of view of computer implementation: most of the code for different element shapes and types is then shared, and does not have to be repeated. However, it does mean that we have to express the derivative of the basis functions using a chain rule:

$$\frac{\partial N_i(\xi)}{\partial x} = \frac{\partial N_i(\xi)}{\partial \xi}\frac{\partial \xi}{\partial x} \tag{2.26}$$

The partial derivative $\frac{\partial \xi}{\partial x}$ is readily available from (2.22), and may be identified as the inverse of the Jacobian. To state the integral of the term corresponding to the mass matrix

$$\int_{x_i}^{x_j} N_j(x)\mu(x)N_i(x)\ \mathrm{d}x \approx \frac{1}{2}(x_j - x_i)\sum_{k=1}^{M}\frac{\partial N_j}{\partial x}(\xi_k)\frac{\partial N_i}{\partial x}(\xi_k)W_k.$$

where (2.26) is used to evaluate the derivatives of the basis functions.

At a first sight, the integrals in the stiffness matrix (2.16), and in the mass matrix (2.17), the latter will seem to require a more accurate numerical quadrature rule: the stiffness matrix involves products of the derivatives of the basis functions, which for linear basis functions are constants; the mass matrix, on the other hand, requires products of the basis functions themselves, which are linear functions of x. Therefore, the stiffness matrix will result from integrals of constants, while the mass matrix is the result of integrals of quadratic functions. However, at times increased efficiency and even accuracy may be achieved if the mass matrix is not integrated exactly, in particular diagonal mass matrices are often used to achieve both benefits.

The numerical quadratures that are in common use with polynomial finite elements are the Gaussian rules. They are well described in a number of textbooks, see for instance Reference [CC2005], and we are going to introduce them later.

2.9 Putting it together: system of ODE's

Applying the piecewise linear basis functions derived in Section 2.7 to equation (2.29) is a straightforward. The unknown degrees of freedom are $w_2(t)$, $w_3(t)$, ..., $w_N(t)$; the function $w_1(t)$ is given by the boundary conditions: the condition $w(x=0,t)=\bar{w}_0(t)$ becomes, as a result of the Kronecker delta property, $w_1(t)=\bar{w}_0(t)$.

The stiffness and mass matrices will be symmetric, tri-diagonal, i.e.

$$K_{ji}=\int_0^L \frac{\partial N_j}{\partial x}P\frac{\partial N_i}{\partial x}\,\mathrm{d}x \begin{cases}\neq 0, \text{ if } |i-j|\leq 1;\\ =0, \text{ otherwise.}\end{cases} , \tag{2.27}$$

and

$$M_{ji}=\int_0^L N_j \mu N_i\,\mathrm{d}x \begin{cases}\neq 0, \text{ if } |i-j|\leq 1;\\ =0, \text{ otherwise.}\end{cases} , \tag{2.28}$$

Equation (2.15) is trivially modified to read

$$N_j(L)F_L-\sum_{i=1}^{N}K_{ji}w_i(t)+\int_0^L N_j q\,\mathrm{d}x-\sum_{i=1}^{N}M_{ji}\ddot{w}_i(t)=0, \quad j=2,...,N \ , \tag{2.29}$$

Note that j runs from 2 (as explained above), but i ranges over all nodes, i.e. also the first degree of freedom is included, even though it is determined from the boundary condition. In effect, nonzero displacement $w_1(t)$ generates an external force with the jth component

$$-K_{j1}w_1(t)-M_{j1}\ddot{w}_1(t) \ .$$

The second order differential equations (2.29) may be integrated for instance by converting them to first order form and using an off-the-shelf Matlab integrator. However, because of their special form, there are excellent custom-tailored algorithms for this purpose: for example the Newmark explicit algorithm.

3

Introducing the Matlab code

In this chapter we will introduce a finite element library (or toolbox, if you prefer), `SOFEA`[1]. It will be used to produce finite element solutions using the results of the previous chapter for the Galerkin method. `SOFEA` implementation is in Matlab, and its design is based on the object oriented support in Matlab (release 14 and later). In particular, all the methods and algorithms are present in the library as classes, or methods defined for classes.

3.1 Statics

When the inertial forces may be neglected in the balance equation, we have the case of statics (static equilibrium). The Galerkin formulation simply drops the terms with the accelerations, and reads

$$N_j(L)F_L - \sum_{i=1}^{N} K_{ji} w_i + \int_0^L N_j q \,\mathrm{d}x = 0, \quad j = 2, ..., N \ , \tag{3.1}$$

which may be arranged in matrix form as

$$\boldsymbol{K}\boldsymbol{d} = \boldsymbol{L} \tag{3.2}$$

where $\boldsymbol{K}$ is a square $(N-1) \times (N-1)$ matrix collecting K_{ji}, $i, j = 2, ..., N$. The column matrix $\boldsymbol{d}$ collects the degrees of freedom $d_k = w_{k+1}$, $k = 1, ..., N-1$. The column matrix $\boldsymbol{L}$ is the load vector, with components

$$L_k = N_{k+1}(L)F_L - K_{k+1,1} w_1 + \int_0^L N_{k+1} q \,\mathrm{d}x = 0, \quad k = 1, ..., N-1 \ , \tag{3.3}$$

3.2 Statics: uniform load

Furthermore, we assume the transverse load q is uniform, and the transverse force is absent, $F_L = 0$. The Matlab script implementing the solution is `taut_wire/w1`. It starts with the definition of the variables.

```
disp('Taut wire: example 1-- statics, uniform load');
L=6;
P=4;
q =-0.1;
```

---

[1] `SOFEA` is ©2005, Petr Krysl

Next, the mesh is defined: an array of nodes is created, with node 1 at $x = 0$ and so on. The function `fenode` is the constructor of the class `fenode`, and the attributes are being passed as fields of a `struct`, as pairs "name, value" (for instance, `'id',j`): this approach is uniformly adopted for all constructors. The array `gcells` collects finite elements of the type `gcell_l2`. The attribute `conn` is the connectivity: the numbers of nodes that are connected by the element. The finite elements are referred to as ***geometric cells***. The main reason is that the finite elements have rather limited responsibilities in `SOFEA`, namely calculation of the basis functions (and their derivatives), and drawing of the shape of the cell are essentially all that is required.

```
0005 n=2; % number of elements
0006 % Mesh
0007 x=0;
0008 fens=[];
0009 for j= 1:n+1
0010     fens=[fens fenode(struct ('id',j,'xyz',[x]));];
0011     x = x+(L/n);
0012 end
0013 gcells = [];
0014 for j= 1:n
0015     gcells = [gcells gcell_l2(struct('id',j,'conn',[j j+1]))];
0016 end
```

The operations on the mesh that reflect the particular problem that is being solved are encapsulated in the class descended from the finite element block, `feblock`. In particular, the prestressed wire stiffness and mass matrix, the effect of the distributed load, q, and the effect of the nonzero displacement at $x = 0$, are computed by the methods of the class `feblock_defor_taut_wire`. Note that Simpson's 1/3 rule is being used for the numerical quadrature. The finite element block consists of all the finite elements.

```
0017 % Finite element block
0018 feb = feblock_defor_taut_wire(struct ('mater',mater_defor,...
0019     'gcells',gcells,...
0020     'integration_rule',simpson_1_3_rule,...
0021     'P',P));
```

The quantities that are interpolated on the finite element mesh, such as the transverse displacement of the wire, w, or the geometry of the mesh, are represented in `SOFEA` as instances of the class `field`. The field `geom` records the geometry of the mesh. The constructor on line `0023` retrieves this information from the array of the nodes. The dimension of the field is 1 because each degree of freedom is just a single displacement. On line `0025` we define the field of the transverse displacements, `w`. For convenience, it is defined by cloning the `geom` field, and then zeroing out all the degrees of freedom.

```
0022 % Geometry
0023 geom = field(struct ('name',['geom'], 'dim', 1, 'fens',fens));
0024 % Define the displacement field
0025 w   = 0*clone(geom,'w');
```

Next, the displacement (essential) boundary conditions are defined, and applied to the displacement field. The method `set_ebc` simply records which components of the degrees of freedom, at which node, are being prescribed (or released), and to which value are they being prescribed. The method `apply_ebc` is then used to transfer this information to the actual degrees of freedom. Finally, the method `numbereqns` numbers the degrees of freedom that are not being prescribed, effectively assigning each one a global equation number.

An important remark should be made here: Lines `0028`, `0029`, and `0030` illustrate a design feature of Matlab, where all arguments are passed by value. Therefore, no matter what we do with the arguments inside the functions, the values that were passed by the caller into those functions do not change at all. The functions work with copies, not the actual variables that the caller passed. If the caller wishes to change the variables, the method must return the changed value, and the caller must assign this value. Example: on line `0031` the method `numbereqns` will number the equations in a copy of the field `w`, and then will return the copy. Since we assign back to the field `w`, the computed numbering of the equations will be now available in `w`; if we did not sign back to `w`, all the work done by the method `numbereqns` would be forgotten.

```
0026 % Apply EBC's
0027 fenids=[1]; prescribed=[1]; component=[1]; val=0;
0028 w   = set_ebc(w, fenids, prescribed, component, val);
0029 w   = apply_ebc (w);
0030 % Number equations
0031 w   = numbereqns (w);
```

Next we create the global system equations. The stiffness matrix is an object, `K`, which gets created in line `0033`, and initialized to represent a dense `neqns`×`neqns` matrix. In line `0034`, the stiffness matrices calculated for each finite element by the block `feb` are assembled into the global matrix `K` (class `dense_sysmat`). The number of equations is being retrieved from the displacement field (the globally equation numbers have been placed there above) using the `get` method. The `get` method is available for all `SOFEA` objects, and just typing `w` at the command line produces a list of all the properties that can be obtained from the object. A great way in which objects may be explored is the graphical user interface of the object browser, `OBgui` (part of `SOFEA`). Its operation is supported by the output of `get(w)` (notice that only the object itself is passed as argument): using `get` in this way returns a cell array, with the name and the description of each attribute.

```
0032 % Assemble the system matrix
0033 K = start (dense_sysmat, get(w, 'neqns'));
0034 K = assemble (K, stiffness(feb, geom, w));
```

The load q is in this case represented by the `body_load` class. The global load object `sysvec` is assembled from element load vectors computed by the finite element block.

```
0035 % Load
0036 bl = body_load(struct ('magn',inline(num2str(q))));
0037 F = start (sysvec, get(w, 'neqns'));
0038 F = assemble (F, body_loads(feb, geom, w, bl));
```

Finally, the global stiffness object is asked to produce the actual stiffness array, and the global load object is asked to supply the actual load vector. The standard backslash Matlab operator than produces the solution, which is then stored in the proper places in the displacement field `w`. The method `scatter_sysvec` distributes the system vector (the solution of the system of linear equations) to the proper degrees of freedom, and we should note that again the result is assigned to `w`.

```
0039 % Solve
0040 w = scatter_sysvec(w, get(K,'mat')\get(F,'vec'));
```

A graphical representation is generated by plotting the linear coordinate (`gather`ed from the geometry field `geom`) versus the linear interpolation of the approximate displacements (`gather`ed from `w`). The analytical solution is also plotted (line `0043`). It is noteworthy that the approximate solution interpolates the analytical solution.

```
0041 % Plot
0042 xs= (0:0.01:L);
0043 plot (xs, -q/P*xs.*(xs/2-L),'r-','linewidth', 3);
0044 hold on
0045 plot (gather (geom, (1:n+ 1),'values'), ...
0046     gather (w, (1:n+ 1),'values'),'bo-','linewidth', 3);
0047 figure (gcf)
```

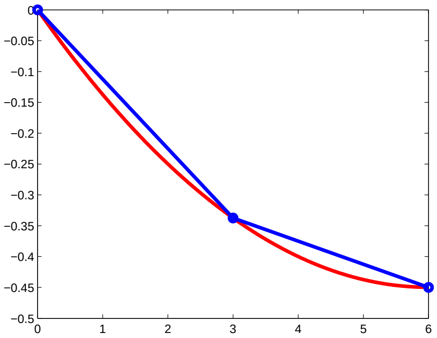


Fig. 3.1. The displacements of the taut wire

3.3 Free vibration

If we remove all external loads, and prescribe homogeneous (zero) displacements, the Galerkin formulation reads

$$-\sum_{i=2}^{N} K_{ji} w_i(t) - \sum_{i=2}^{N} M_{ji} \ddot{w}_i(t) = 0, \quad j = 2, ..., N \;, \tag{3.4}$$

which may be arranged in matrix form as

$$\boldsymbol{K}\boldsymbol{w} + \boldsymbol{M}\ddot{\boldsymbol{w}} = \boldsymbol{0} \tag{3.5}$$

where $\boldsymbol{K}$ is a square $(N-1) \times (N-1)$ matrix collecting K_{ji}, $i, j = 2, ..., N$, and analogously for the mass matrix. The column matrix $\boldsymbol{w}$ collects the degrees of freedom $w_i(t)$. Equations (3.5) represent the so-called free vibration response. The solution is sought in the form $\boldsymbol{w}(t) = \boldsymbol{\phi} \exp(\omega t)$, which leads to the generalized eigenvalue problem

$$\boldsymbol{K}\boldsymbol{\phi} - \omega^2 \boldsymbol{M}\boldsymbol{\phi} = \boldsymbol{0} \tag{3.6}$$

where ω is the circular frequency, and ϕ is the eigenmode.

3.4 Virtual work principle

Part II

Thermal analysis

4

Model of Heat Diffusion

4.1 Balance equation

In this section, our goal is to derive the balance equation that governs heat conduction in solids as a partial differential expression. It will be converted to a residual form, which will then be treated with the Galerkin method.

To begin with we pick a control volume, and we keep track of the heat energy within that volume. The control volume may be the whole structure, part of the structure, or just a very small chunk of material surrounding a given point in space (Figure 4.1). The amount of heat energy in the control volume U is expressed in terms of volume density of heat energy, u

$$U = \int_V u \, \mathrm{d}V \tag{4.1}$$

As the means of change of the heat energy within the control volume we consider *outflow* (inflow) of heat energy via the boundaries, and *heat generation* (or loss) within the volume. These quantities will be expressed in terms of rates. Therefore, the amount of energy flowing out of the control volume through its bounding surface S per unit time is

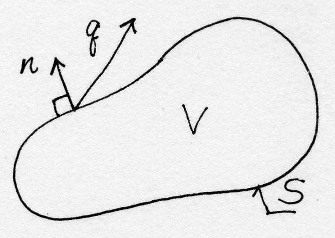


Fig. 4.1. The domain for the heat conduction problem

$$\int_S \boldsymbol{n} \cdot \boldsymbol{q} \, \mathrm{d}S \,, \tag{4.2}$$

where $\boldsymbol{n}$ is the outer normal to the surface S, and $\boldsymbol{q}$ is the heat flux (amount of heat flowing through a unit area per unit time). The amount of energy generated within the control volume per unit time is

$$\int_V Q \, \mathrm{d}V \,. \tag{4.3}$$

where Q is the rate of heat generation per unit volume; for example, heat is released or consumed by various deformation and chemical processes (as work of viscous stresses, reaction product of curing concrete or polymer resins, and so on).

Collecting the terms, we can write for the change of the heat energy within the control volume the rate equation

$$\frac{\mathrm{d}U}{\mathrm{d}t} = -\int_S \boldsymbol{n}\cdot\boldsymbol{q}\,\mathrm{d}S + \int_V Q\,\mathrm{d}V\ . \tag{4.4}$$

Finally, differentiating U with respect to time will be possible if we assume that $U = U(T)$, i.e. if U is a function of the absolute temperature T. Holding the control volume fixed in time, the time differentiation may be taken inside the integral over the volume

$$\frac{\mathrm{d}U}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\int_V u\,\mathrm{d}V = \int_V \frac{\mathrm{d}u}{\mathrm{d}t}\,\mathrm{d}V\ , \tag{4.5}$$

and with the application of the chain rule, the relationship (4.5) is expressed as

$$\frac{\mathrm{d}U}{\mathrm{d}t} = \int_V \frac{\mathrm{d}u}{\mathrm{d}t}\,\mathrm{d}V = \int_V \frac{\mathrm{d}u}{\mathrm{d}T}\frac{\partial T}{\partial t}\,\mathrm{d}V\ , \tag{4.6}$$

The quantity $c_V = \frac{\mathrm{d}u}{\mathrm{d}T}$ is a characteristic property of a solid material (called specific heat at constant volume), and needs to be measured. It is typically dependent on temperature, but we will assume that it is a constant; otherwise it leads to nonlinear models.

Substituting, we write

$$\int_V c_V\frac{\partial T}{\partial t}\,\mathrm{d}V = -\int_S \boldsymbol{n}\cdot\boldsymbol{q}\,\mathrm{d}S + \int_V Q\,\mathrm{d}V\ . \tag{4.7}$$

This equation consists of volume integrals and a surface integral. If all the integrals were volume integrals, over the same volume of course, we could proclaim that the integral statement (sometimes called a global balance equation) would hold provided the integrands satisfied a so-called local balance equation (recall that to get the local balance equation is our goal). For instance, from the integral statement

$$\int_V \alpha\frac{\partial M}{\partial t}\,\mathrm{d}V = \int_V \mu\,\mathrm{d}V\ , \tag{4.8}$$

where α, M, and μ are some functions, one could conclude that

$$\alpha\frac{\partial M}{\partial t} = \mu\ , \tag{4.9}$$

which is a local version of (4.8). An argument along these lines could for instance invoke the assumption that the volume V was arbitrary, and that it could be shrunk around a given point, which in the limit would allow the volume to be canceled on both sides of the equation.

To execute this program for equation (4.7), we have to convert the surface integral to a volume integral. We have the needed tool in the celebrated ***divergence theorem***.

$$\int_V \mathrm{div}\boldsymbol{q}\,\mathrm{d}V = \int_S \boldsymbol{n}\cdot\boldsymbol{q}\,\mathrm{d}S\ , \tag{4.10}$$

where the divergence of the flux vector is defined in Cartesian coordinates as

$$\mathrm{div}\boldsymbol{q} = \frac{\partial q_x}{\partial x} + \frac{\partial q_y}{\partial y} + \frac{\partial q_z}{\partial z}\ .$$

Consequently, equation (4.7) may be rewritten

$$\int_V c_V \frac{\partial T}{\partial t}\,\mathrm{d}V = -\int_V \mathrm{div}\boldsymbol{q}\,\mathrm{d}V + \int_V Q\,\mathrm{d}V\ . \tag{4.11}$$

and grouping the terms as

$$\int_V \left[c_V \frac{\partial T}{\partial t} + \mathrm{div}\boldsymbol{q} - Q\right]\,\mathrm{d}V = 0\ . \tag{4.12}$$

we may conclude that the inside of the bracket has to vanish since the volume could be entirely arbitrary. Therefore, we arrive at the ***local balance equation***

$$c_V \frac{\partial T}{\partial t} + \mathrm{div}\boldsymbol{q} - Q = 0\ . \tag{4.13}$$

4.2 Constitutive equation

Equation (4.13) contains too many variables: both temperature and heat flux. Since it is a scalar equation, the logical next step is to express the heat flux in terms of temperature. That is the contents of the Fourier model: heat flows opposite to the gradient of the temperature (downhill). In matrix form

$$\boldsymbol{q} = -\boldsymbol{\kappa}(\mathrm{grad}T)^T\ . \tag{4.14}$$

The matrix $\boldsymbol{\kappa}$ is the conductivity matrix of the material. The most common forms of $\boldsymbol{\kappa}$ are

$$\boldsymbol{\kappa} = \kappa\mathbf{1} \tag{4.15}$$

for the so-called thermally isotropic material, and

$$\boldsymbol{\kappa} = \begin{pmatrix} \kappa_x & 0 & 0 \\ 0 & \kappa_y & 0 \\ 0 & 0 & \kappa_z \end{pmatrix} \tag{4.16}$$

for materials that have three orthogonal directions of different thermal conductivities (orthotropic material); κ is the isotropic thermal conductivity coefficient, $\mathbf{1}$ is the identity matrix, and κ_x, κ_y, and κ_z are the orthotropic thermal conductivities. (Some materials have preferred directions in which heat would like to flow, for instance along the fibers in a composite. Visually, we can imagine a corrugated steel roof, with the channels running not directly downhill, but tilted away from the slope – the water would run preferentially in the channels, but generally downhill.)

The funny looking transpose of the temperature gradient follows from the definition: the gradient of the scalar is a row matrix

$$\mathrm{grad}T = \begin{pmatrix} \frac{\partial T}{\partial x} & \frac{\partial T}{\partial y} & \frac{\partial T}{\partial z} \end{pmatrix} \tag{4.17}$$

With the constitutive equation, the balance equation (4.13) is now purely in terms of the absolute temperature,

$$c_V \frac{\partial T}{\partial t} - \mathrm{div}\left[\boldsymbol{\kappa}(\mathrm{grad}T)^T\right] - Q = 0\ . \tag{4.18}$$

4.3 Boundary conditions

From now on, V is going to be the volume of the whole solid of interest. The most important fact about the boundary conditions is that we need to have a boundary condition *at each point* of the surface S. As we expect by now, the model is about temperature. Correspondingly, the boundary conditions are an expression of our knowledge of the temperature distribution in the solid.

The simplest boundary condition results if we know the surface temperature along one part of S at all times. This part of the surface will be called S_1 (see Figure 4.2).Therefore,

$$T(\boldsymbol{x}, t) - \overline{T}(\boldsymbol{x}, t) = 0, \quad \boldsymbol{x} \text{ on } \; S_1 \; . \tag{4.19}$$

where by $\boldsymbol{x}$ we mean the position vector. This type of condition is known as the primary, or essential, boundary condition.

The heat flux entering or leaving the solid may also be known (measured by a heat flux gauge). Generally, we do not know the heat flux along the surface, only the normal component, which is obtainable from the normal and the heat flux as $q_n = \boldsymbol{n} \cdot \boldsymbol{q}$. Therefore, along the part of the surface S_2 the normal component of the heat flux may be prescribed

$$\boldsymbol{n} \cdot \boldsymbol{q} - \overline{q}_n = 0, \quad \text{on } \; S_2 \; . \tag{4.20}$$

All quantities are given at a particular point on the boundary as functions of time, similarly to the first boundary condition. This type of condition is known as the natural (or flux) boundary condition.

In the last example of a boundary condition, we will mention the heat transfer driven by a temperature difference at surface. The normal component of the heat flux is given as

$$\boldsymbol{n} \cdot \boldsymbol{q} - h(T - T_a) = 0, \quad \text{on } \; S_3 \; . \tag{4.21}$$

where T_a is the known temperature of the surrounding medium (ambient temperature), and h is the heat transfer coefficient.

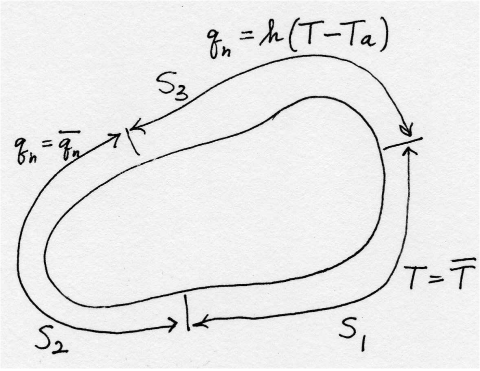


Fig. 4.2. The subdivision of the surface for the purpose of the boundary condition application

4.4 Initial condition

The primary variable in our problem is the temperature, T, and it is present in the balance equation (4.18) with the first order time derivative. Therefore, we will need one initial condition,

$$T(\boldsymbol{x}, 0) = \overline{T}_0(\boldsymbol{x}) \quad \text{in} \quad V \ . \tag{4.22}$$

The initial condition must be match with boundary conditions on S_1 at time $t = 0$:

$$\overline{T}_0(\boldsymbol{x}) = \overline{T}(\boldsymbol{x}, 0), \quad \boldsymbol{x} \text{ on } \ S_1 \ . \tag{4.23}$$

4.5 Summary of the PDE model of heat conduction

Figure 4.3 gives a diagrammatic overview of the terminology and the various equations of the model of heat conduction.

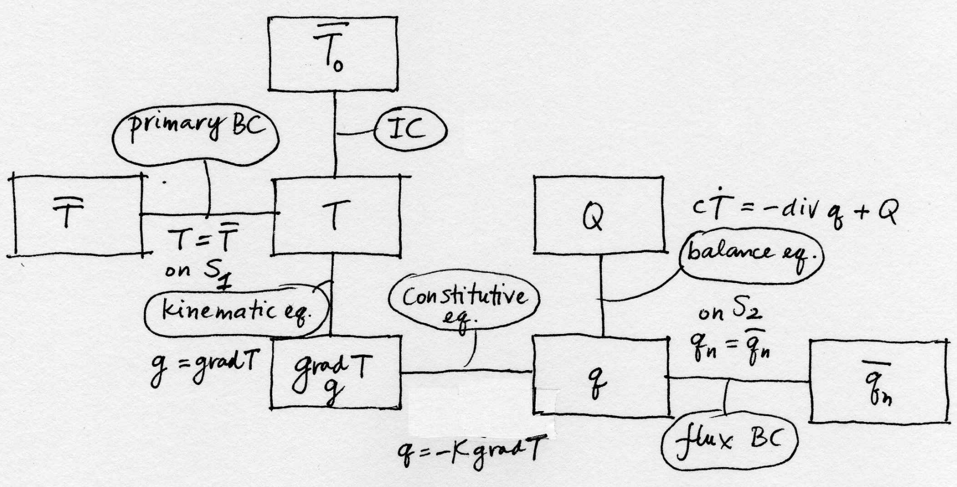

$\overline{T}_0$
primary BC
IC
$\overline{T}$
T
$T = \overline{T}$
on S_1
kinematic eq.
$g = \text{grad}T$
grad T
g
Constitutive eq.
$q = -K\,\text{grad}T$
Q
$c\dot{T} = -\text{div}\, q + Q$
balance eq.
on S_2
$q_n = \overline{q}_n$
q
$\overline{q}_n$
flux BC

Fig. 4.3. Diagram of the heat conduction model

5

Galerkin method for the model of heat conduction

5.1 Weighted residual formulation

The balance equation (4.18) defines the balance residual as

$$r_B = c_V \frac{\partial T}{\partial t} - \operatorname{div}\left[\boldsymbol{\kappa}(\operatorname{grad}T)^T\right] - Q \ . \tag{5.1}$$

As explained in Chapter 2, the first step is to write the weighted residual equation

$$\int_V \eta(\boldsymbol{x}) r_B(\boldsymbol{x}, t)\ \mathrm{d}V \ . \tag{5.2}$$

The first and the third term are kept as they are, but the second term reminds us of a similar term in equation (2.3): the test function η multiplies an expression that contains the second derivatives of temperature (the $\operatorname{div}\left[\boldsymbol{\kappa}(\operatorname{grad}T)^T\right]$ term), which was why Section 2.5 was needed. Balancing the order of differentiation by shifting one derivative from the temperature to the test function η will be beneficial here too: similarly to Section 2.5, we will be able to satisfy the natural boundary conditions without having to include them as a residual (naturally!). As before, the price to pay is the need to place some restrictions on the test function.

Integration by parts was used in Section 2.5, and just a little bit more general tool will work here too. For the moment, it will be convenient to work with the expression

$$-\eta \operatorname{div}\left[\boldsymbol{\kappa}(\operatorname{grad}T)^T\right] = \eta \operatorname{div}\boldsymbol{q} \ ,$$

that is, with the flux variable replacing $\boldsymbol{\kappa}(\operatorname{grad}T)^T$.

The integration by parts in the case of a multidimensional integral is generalized in the divergence theorem (4.10). We may anticipate that $\eta \operatorname{div}\boldsymbol{q}$ is the result of the Leibniz rule applied to the vector $\eta\ \boldsymbol{q}$. That is indeed the case

$$\operatorname{div}(\eta\ \boldsymbol{q}) = \eta \operatorname{div}\boldsymbol{q} + (\operatorname{grad}\eta)\ \boldsymbol{q} \tag{5.3}$$

which is easily verified in components.

Therefore, we may start working on the integral

$$\int_V \eta \operatorname{div}\boldsymbol{q}\ \mathrm{d}V$$

where we substitute from (5.3)

$$\int_V \eta \operatorname{div}\boldsymbol{q}\ \mathrm{d}V = \int_V \operatorname{div}(\eta\ \boldsymbol{q})\ \mathrm{d}V - \int_V (\operatorname{grad}\eta)\ \boldsymbol{q}\ \mathrm{d}V \tag{5.4}$$

The divergence theorem may be applied to the first integral on the right

$$\int_V \eta \, \mathrm{div}\boldsymbol{q} \, \mathrm{d}V = \int_S \eta \, \boldsymbol{q} \cdot \boldsymbol{n} \, \mathrm{d}S - \int_V (\mathrm{grad}\eta) \, \boldsymbol{q} \, \mathrm{d}V \tag{5.5}$$

But $\boldsymbol{q} \cdot \boldsymbol{n}$ is known on parts of the boundary – see equations (4.20) and (4.21). Therefore, we may split the surface integral into one for each sub-surface,

$$\int_V \eta \, \mathrm{div}\boldsymbol{q} \, \mathrm{d}V = \int_{S_1} \eta \, \boldsymbol{q}\cdot\boldsymbol{n} \, \mathrm{d}S + \int_{S_2} \eta \, \overline{q}_n \, \mathrm{d}S + \int_{S_3} \eta \, h(T - T_a) \, \mathrm{d}S - \int_V (\mathrm{grad}\eta) \, \boldsymbol{q} \, \mathrm{d}V \tag{5.6}$$

We see that the situation is analogous to the one discussed below equation (2.10): The integral over the part of the surface S_1 is troublesome, because $\boldsymbol{q}\cdot\boldsymbol{n}$ is unknown there. However, we have the option of making η vanish along S_1. In this way, we obtain

$$\int_V \eta \, \mathrm{div}\boldsymbol{q} \, \mathrm{d}V = \int_{S_2} \eta \, \overline{q}_n \, \mathrm{d}S + \int_{S_3} \eta \, h(T - T_a) \, \mathrm{d}S - \int_V (\mathrm{grad}\eta) \, \boldsymbol{q} \, \mathrm{d}V \tag{5.7}$$

where $\eta(\boldsymbol{x}) = 0$ for $\boldsymbol{x} \in S_1$.

Expanding the weighted residual equation (5.2) yields

$$\begin{aligned} &\int_V \eta \, r_B \, \mathrm{d}V = \\ &\quad \int_V \eta c_V \frac{\partial T}{\partial t} \, \mathrm{d}V + \int_V (\mathrm{grad}\eta) \, \boldsymbol{\kappa} (\mathrm{grad}T)^T \, \mathrm{d}V - \int_V \eta Q \, \mathrm{d}V \\ &\quad + \int_{S_2} \eta \, \overline{q}_n \, \mathrm{d}S + \int_{S_3} \eta \, h(T - T_a) \, \mathrm{d}S = 0, \qquad \eta(\boldsymbol{x}) = 0 \text{ for } \boldsymbol{x} \in S_1 \end{aligned} \tag{5.8}$$

5.2 Reducing the model dimension

In this Section we show how the original three-dimensional model can be reduced to just two active dimensions. For some structures we can make the observation that the temperature does not vary along one coordinate direction, say along the z direction. Figure 5.1 shows a disk of thickness t. It is a slice of a structure of an unchanging cross-section which is very along in the z direction compared to the transverse dimensions (some authors call this “infinitely long”, evidently with a bagfull of grains of salt). If the temperature distribution does not depend on the z direction, and if we can neglect what is happening near the end sections, the component of the temperature gradient along the z direction will be negligible, $\partial T/\partial z = 0$. This does not necessarily mean that the z component of the heat flux is also zero: the partial derivatives $\partial T/\partial x$, and $\partial T/\partial y$ multiply the first two columns in row three of (4.14) to yield

$$q_z = \kappa_{zx} \partial T/\partial x + \kappa_{zy} \partial T/\partial y \; .$$

However, for the two classes of materials (4.15) and (4.16) the two coefficients κ_{zx} and κ_{zy} are identically zero, which means that if the temperature gradient $\partial T/\partial z$ is zero, the heat flux in that direction also vanishes.

Going back to the Figure 5.1: the heat flux through the cross sections is zero, and the temperature through the thickness of the disk is uniform (i.e. the temperature does not vary with z). The surface of the three-dimensional solid consists of the two cross sections, and of the cylindrical surfaces, the inner and the outer. The two

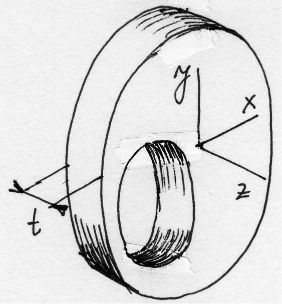


Fig. 5.1. Diagram of the heat conduction model

cylindrical surfaces may be associated with boundary condition of any type. The two cross sections are associated with the boundary condition of zero heat flux, $\overline{q}_n = 0$ (type S_2, equation (4.20))

$$\boldsymbol{n} \cdot \boldsymbol{q} = \pm q_z = 0, \quad \text{on the cross sections} \ . \tag{5.9}$$

Since the temperature does not vary with z, the integrals (5.7) may be simplified by pre-integrating in the thickness direction. The volume integrals then result in integrals over the cross-sectional area, S_c, (see Figure 5.2); provided $\overline{q}_n$ and h are independent of z, the surface integrals result in curve integrals over the contour of the cross-section, C_c.

$$\begin{aligned} & t\int_{S_c} \eta c_V \frac{\partial T}{\partial t} \, \mathrm{d}S + t\int_{S_c} (\mathrm{grad}\eta) \, \boldsymbol{\kappa} (\mathrm{grad}T)^T \, \mathrm{d}S - t\int_{S_c} \eta Q \, \mathrm{d}S \\ & + t\int_{C_{c,2}} \eta \, \overline{q}_n \, \mathrm{d}C + t\int_{C_{c,3}} \eta \, h(T - T_a) \, \mathrm{d}C = 0, \qquad \eta(\boldsymbol{x}) = 0 \text{ for } \boldsymbol{x} \in C_{c,1} \end{aligned} \tag{5.10}$$

Note that the thickness t will cancel, and consequently does not play a role at all. Nevertheless, equation (5.10) still applies to a fully three-dimensional body. Note that (5.10) does not refer to z, except in the term $\partial./\partial z$. We know that the temperature does not depend on z, and concerning the gradient of η: we simply assume that η does not depend on z: $\eta = \eta(x, y)$. The last assumption completes the reduction of the problem to two dimensions: all the functions depend on x and y only.

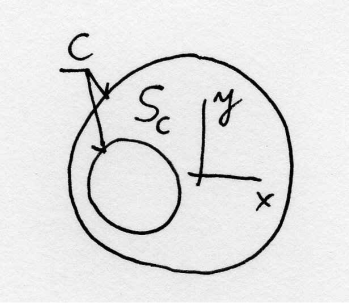


Fig. 5.2. Diagram of the heat conduction model

5.3 Test and trial functions: basis functions on triangulations

It is time to talk about the test and trial function. They are both functions of x and y only, $\eta = \eta(x, y)$ and $T = T(x, y, t)$ (and for the trial function, time). The only difference between them is the value they assume on part of the boundary (part of the cross-section contour, for our two-dimensional disk) where the temperature is being prescribed, $C_{c,1}$:

$$T(\boldsymbol{x}, t) = \overline{T}(\boldsymbol{x}, t), \quad \eta(\boldsymbol{x}) = 0 \quad \boldsymbol{x} \text{ on } C_{c,1} \ .$$

Let us consider first the test function. It needs to be defined as a function of x and y over arbitrarily shaped domains. The concept of piecewise linear functions defined over tilings of arbitrary domains into triangles is quite ancient (at least in terms of the development of computational mechanics)[1]. The domain of the disk with a hole (shown in Figure 5.2) is approximated as a collection of triangles (in other words, it is *tiled* with triangles, or *triangulated*), see Figure 5.3. The mesh consisting of triangles is typically called ***triangulation***, even though sometimes *any* mesh is called that. The vertices of the triangulation are called ***nodes*** (compare with Section 2.7), while the line segments connecting the nodes are called ***edges***. Evidently, the triangles are the finite elements.

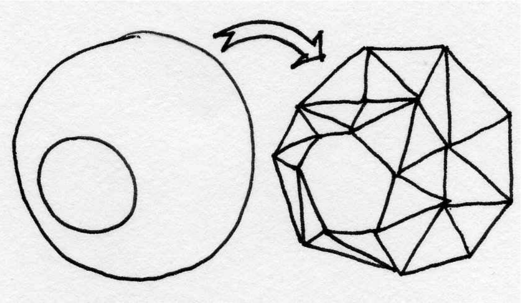


Fig. 5.3. Mesh of the disk domain

Interpolation on the triangle mesh will be treated as a linear combination of "tents". Each individual tent is formed by grabbing one of the nodes (say J) and raising it out of the plane of the triangulation (traditionally to a unit height). The tent canvas is stretched over the edges that connect at the node J, and are clamped down by the ring of the edges that surround node J. The cartoon of one particular basis function tent is shown in Figure 5.4. For those who do not like tents, the term *hat function* may be preferable.

All the triangles that are connected in the node J ***support*** the function N_J, which is another way of saying that the function N_J is nonzero in these triangles; evidently, it is defined to be zero everywhere else. (If you are inside the "tent", you are standing on the support of the function.)

---

[1] The so-called "linear triangle" made its first appearance in a lecture by Courant in 1943, applied to Poisson's equation, which is a time-independent version of the heat conduction equation of this chapter. It was then picked up as a structural element in aerospace engineering to model Delta wing skin panels, as described in the 1956 paper by Turner, Clough, Martin and Topp. Clough then applied the triangle to problems in civil engineering, and he also coined the terminology "finite element".

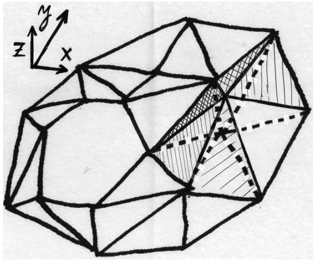


Fig. 5.4. Visual representation of one basis function on the mesh of the disk

It remains to write down the equations that define the function N_J at any point within its support. That means writing an expression for each triangle separately. As discussed in Section 2.8, the alternative viewpoint would rather express all the nonzero pieces of all the basis functions over a single triangle (element). Referring to Figure 5.4, there are only three such functions: the three basis functions associated with the nodes at the corners of the element; all the other basis functions in the mesh are identically zero over this element. Thus, we stand before the task of writing down the expressions for the three basis functions on a single triangle.

5.4 Basis functions on the standard triangle

Each of the three basis functions is zero along one edge of the triangle: again, refer to Figure 5.4. The task is accomplished most readily when the triangle is in a special position with respect to the coordinates: the ***standard triangle***; see Figure 5.5. The basis functions associated with nodes ② and ③ are simply

$$N_2(\xi,\eta) = \xi \ , \tag{5.11}$$

and

$$N_3(\xi,\eta) = \eta \ . \tag{5.12}$$

As is easily verified, N_2 is zero along the edge ①③, and assumes value $+1$ at node ②; analogous properties hold for N_3. If N_1 should be equal to $+1$ at the origin, it must be written as

$$N_1(\xi,\eta) = 1 - \xi - \eta \ . \tag{5.13}$$

Clearly, N_1 vanishes at the edge opposite node ①. Thus, we see that the three functions we just formulated satisfy the Kronecker delta property, equation (2.19). As in Section 2.7 this means the degree of freedom at each node of the triangle is the value of the interpolated function at the node.

In this way, we formulate interpolation over the standard triangle. One quantity that we can interpolate on the standard triangle are the Cartesian coordinates.

$$\boldsymbol{x} = \sum_{i=1}^{3} N_i(\xi,\eta)\boldsymbol{x}_i \ , \tag{5.14}$$

where the result of the interpolation is a point in the Cartesian coordinates

$$\boldsymbol{x} = \begin{bmatrix} x \\ y \end{bmatrix} \ ,$$

and

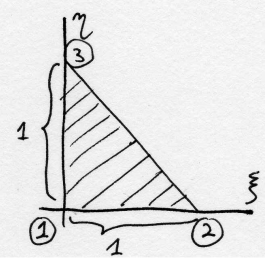


Fig. 5.5. Standard triangle

$$\boldsymbol{x}_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix} \qquad i = 1, 2, 3 \ ,$$

are the coordinates of the three points that are being interpolated. Equation (5.14) is a mapping from the pair ξ, η to the point x, y. Substituting for the basis functions, it may be written explicitly as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} (x_2 - x_1) & (x_3 - x_1) \\ (y_2 - y_1) & (y_3 - y_1) \end{bmatrix} \begin{bmatrix} \xi \\ \eta \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \ . \tag{5.15}$$

This matrix equation is accompanied by the picture in Figure 5.15. The two vectors, $\boldsymbol{v}$ and $\boldsymbol{w}$, are the two columns of the square matrix in (5.15):

$$\boldsymbol{v} = \begin{bmatrix} (x_2 - x_1) \\ (y_2 - y_1) \end{bmatrix} \ ,$$

and

$$\boldsymbol{w} = \begin{bmatrix} (x_3 - x_1) \\ (y_3 - y_1) \end{bmatrix} \ .$$

If both ξ and η vary between zero and one, equation (5.15) adds the two vectors, $\xi \boldsymbol{v}$ and $\eta \boldsymbol{w}$ to the vector $[x_1, y_1]^T$, and the result then covers the entire parallelogram; on the other hand, if ξ and η are confined to the interior of the standard triangle, equation (5.15) covers the area of the hatched triangle. To summarize, equation (5.15) is a *map* from the standard triangle to a triangle in the Cartesian coordinates with corners in given locations.

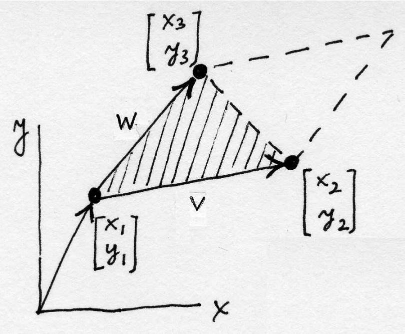

$\begin{bmatrix} x_3 \\ y_3 \end{bmatrix}$
w
y
v
$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$
$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$
x

Fig. 5.6. Interpolating Cartesian coordinates on the standard triangle

Inverting (5.15) to express ξ and η, which could then be substituted into (5.11) – (5.13) to produce basis functions in terms of x and y, looks appealing but should be resisted. The reason is that numerical quadrature is available on the standard triangle, but is much harder on general triangles. This will become especially clear with quadratic elements later in the book.

However, since equation (5.15) is a invertible map from the standard triangle to a triangle in the Cartesian coordinates (invertibility follows if the triangle does not have its corners in a single straight-line: why?), we do get an approach to *evaluating basis functions on a general triangle*. Given a point $\bar{x}, \bar{y}$ in the Cartesian coordinates, and within the bounds of a triangle, we can use the inverse of the map (5.15) to obtain point $\bar{\xi}, \bar{\eta}$ in the standard triangle (path 1 in Figure 5.7). Therefore, we may then evaluate $N_i(\bar{\xi}, \bar{\eta})$, which is the value $N_i(\bar{x}, \bar{y})$ (path 2 in Figure 5.7). That seems awkward, but normally we would want to evaluate the basis functions in order to perform numerical quadrature, that is at a particular point (quadrature point) within the triangle. In that case, $\bar{\xi}, \bar{\eta}$ would be *known* (and $\bar{x}, \bar{y}$ would be unknown), and calculation of the function value is easy. Evaluation of the *derivatives* of the basis functions is a little bit more complex, and will be discussed later in the section on numerical quadrature.

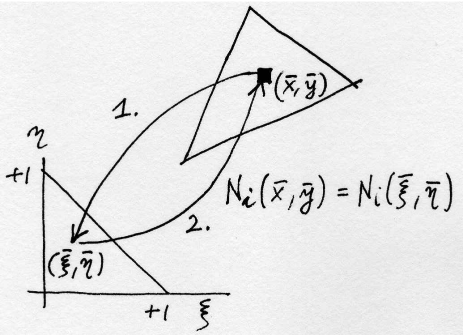

1.
$(\bar{x}, \bar{y})$
η
+1
$N_i(\bar{x}, \bar{y}) = N_i(\bar{\xi}, \bar{\eta})$
2.
$(\bar{\xi}, \bar{\eta})$
+1
ξ

Fig. 5.7. Using the map from the standard triangle to evaluate basis functions over a general triangle

We understand now that each node in the mesh is associated with a single basis function. In the following, whenever we will write

$$N_i = N_i(x, y) \ ,$$

it has to be understood that within each triangle in the mesh, $x = x(\xi, \eta)$, $y = y(\xi, \eta)$, where ξ and η are coordinates on the standard triangle.

5.5 Discretizing the weighted residual equation

The trial function will be expressed using the basis functions as (compare with Section 2.7)

$$T(x, y, t) = \sum_{i=1}^{N} N_i(x, y) T_i(t)$$

where the sum ranges over all the basis functions (i.e. over all the nodes in the mesh). Included are also nodes on the boundary where the temperature is being prescribed, $C_{c,1}$[2]. These nodes are on the other hand excluded from the set of possible test functions (which is expected to vanish along $C_{c,1}$), so that we will choose

$$\eta(x,y) = N_i(x,y), \quad i \text{ excluded when node } i \in C_{c,1}$$

The nodes whose basis functions are not part of the linear combination for the test function are shown as empty circles in Figure 5.8.

To simplify, we shall adopt the following notation:

$$\eta(x,y) = N_j(x,y), \quad \forall \text{ free } j,$$

where "free j" ranges over the nodes where the temperature is not being prescribed; and

$$T(x,y,t) = \sum_{\text{all } i} N_i(x,y)T_i(t)$$

where "all i" ranges over all the nodes, including those where the temperature is being prescribed.

As an aside, because the basis on the standard triangle satisfies the Kronecker delta property (2.19), the values of the degrees of freedom $T_i(t)$ at the nodes "prescribed i" (the nodes with the empty circles in Figure 5.8) are simply the values of the interpolated prescribed temperature at the nodes, $T_i(t) = \overline{T}(x_i, y_i, t)$.

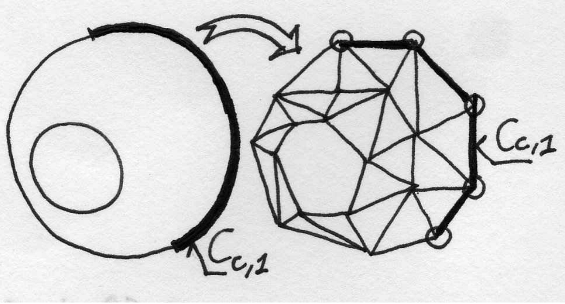


Fig. 5.8. Interpolating Cartesian coordinates on the standard triangle

The finite element expansions for the trial and test functions are now substituted into the weighted residual integral (5.10), which upon the cancellation of the thickness t reads

$$\begin{aligned}&\int_{S_c} \eta c_V \frac{\partial T}{\partial t}\, \mathrm{d}S + \int_{S_c} (\mathrm{grad}\eta)\, \boldsymbol{\kappa} (\mathrm{grad}T)^T\, \mathrm{d}S - \int_{S_c} \eta Q\, \mathrm{d}S \\ &+t \int_{C_{c,2}} \eta\, \bar{q}_n\, \mathrm{d}C + \int_{C_{c,3}} \eta\, h(T - T_a)\, \mathrm{d}C = 0, \qquad \eta(\boldsymbol{x}) = 0 \text{ for } \boldsymbol{x} \in C_{c,1}\ .\end{aligned} \tag{5.16}$$

For clarity, the substitution will be shown term-by-term (henceforth we will omit the arguments):

$$\int_{S_c} \eta c_V \frac{\partial T}{\partial t}\, \mathrm{d}S = \int_{S_c} N_j c_V \sum_{\text{all } i} N_i \frac{\partial T_i}{\partial t}\, \mathrm{d}S\ , \quad \forall \text{ free } j\ , \tag{5.17}$$

---

[2] Apropos boundaries: Figure 5.8 clearly shows that with straight edges we are only approximating any boundaries that are curved. Some error is involved, but fortunately we are able to control this error by reducing the length of the edges.

which simplifies to

$$\sum_{\text{all } i}\left[\int_{S_c} N_j c_V N_i \, \mathrm{d}S\right]\frac{\partial T_i}{\partial t} \; , \quad \forall \text{ free } j \; , \tag{5.18}$$

The term in the bracket mixes together i and j from two different sets. However, some of the degrees of freedom $\partial T_i/\partial t$ are known. Therefore, separating the known and unknown may be a good idea:

$$\begin{aligned}&\sum_{\text{all } i}\left[\int_{S_c} N_j c_V N_i \, \mathrm{d}S\right]\frac{\partial T_i}{\partial t} = \\ &\sum_{\text{free } i}\left[\int_{S_c} N_j c_V N_i \, \mathrm{d}S\right]\frac{\partial T_i}{\partial t} + \\ &\sum_{\text{prescribed } i}\left[\int_{S_c} N_j c_V N_i \, \mathrm{d}S\right]\frac{\partial \overline{T}_i(t)}{\partial t} \; , \quad \forall \text{ free } j \; ,\end{aligned} \tag{5.19}$$

The first integral on the right hand side of (5.19) suggests defining a matrix

$$C_{ji} = \int_{S_c} N_j c_V N_i \, \mathrm{d}S \; , \quad \forall \text{ free } j, i, \tag{5.20}$$

the ***capacity matrix***. The integral in the second term will be given a different symbol, since the meaning is different from the first (the latter is a load-like term)

$$\overline{C}_{ji} = \int_{S_c} N_j c_V N_i \, \mathrm{d}S \; , \quad \forall \text{ free } j, \; \forall \text{ prescribed } i. \tag{5.21}$$

Next, the second term in (5.16):

$$\begin{aligned}&\int_{S_c} (\mathrm{grad}\eta) \; \boldsymbol{\kappa}(\mathrm{grad}T)^T \, \mathrm{d}S = \int_{S_c} (\mathrm{grad}N_j) \; \boldsymbol{\kappa}(\mathrm{grad}\sum_{\text{all } i} N_i T_i)^T \, \mathrm{d}S = \\ &\sum_{\text{free } i}\left[\int_{S_c} (\mathrm{grad}N_j) \; \boldsymbol{\kappa}(\mathrm{grad}N_i)^T \, \mathrm{d}S\right] T_i + \\ &\sum_{\text{prescribed } i}\left[\int_{S_c} (\mathrm{grad}N_j) \; \boldsymbol{\kappa}(\mathrm{grad}N_i)^T \, \mathrm{d}S\right] \overline{T}_i \quad \forall \text{ free } j,\end{aligned} \tag{5.22}$$

and the ***conductivity matrix*** may be defined

$$K_{ij} = \int_{S_c} (\mathrm{grad}N_j) \; \boldsymbol{\kappa}(\mathrm{grad}N_i)^T \, \mathrm{d}S \; , \quad \forall \text{ free } j, i, \tag{5.23}$$

and the elements to go with the load-like term

$$\overline{K}_{ij} = \int_{S_c} (\mathrm{grad}N_j) \; \boldsymbol{\kappa}(\mathrm{grad}N_i)^T \, \mathrm{d}S \; , \quad \forall \text{ free } j, \; \forall \text{ prescribed } i. \tag{5.24}$$

Next, the load term corresponding to the internal heat generation:

$$L_{Q,j} = \int_{S_c} N_j Q \, \mathrm{d}S \; , \quad \forall \text{ free } j, \tag{5.25}$$

Finally, the terms corresponding to natural boundary conditions. On the $C_{c,2}$ part of the boundary, only a load term results.

$$L_{q2,j} = -\int_{C_{c,2}} N_j \, \overline{q}_n \, \mathrm{d}C \tag{5.26}$$

On the other hand, on the $C_{c,3}$ part of the boundary, where the heat flux is proportional to the difference between the ambient temperature and the surface temperature, we get a load term

$$L_{q3,j} = \int_{C_{c,3}} N_j \, hT_a \, \mathrm{d}C \ , \quad \forall \text{ free } j, \tag{5.27}$$

and a ***surface heat transfer matrix***:

$$H_{ji} = \int_{C_{c,3}} N_j \, hN_i \, \mathrm{d}C \ , \quad \forall \text{ free } j, i, \tag{5.28}$$

To summarize, using the definitions of the various matrices and load terms, the system of ordinary differential equations that results from the finite element discretization in space reads

$$\begin{aligned} &\sum_{\text{free } i} C_{ji} \frac{\partial T_i}{\partial t} + \sum_{\text{free } i} K_{ji} T_i + \sum_{\text{free } i} H_{ji} T_i \\ &+ \sum_{\text{prescribed } i} \overline{C}_{ji} \frac{\partial \overline{T}_i(t)}{\partial t} + \sum_{\text{prescribed } i} \overline{K}_{ji} \overline{T}_i - L_{Q,j} - L_{q2,j} - L_{q3,j} = 0 \end{aligned} \quad \forall \text{ free } j, \tag{5.29}$$

5.6 Derivatives of the basis functions; Jacobian

The results of this section are much more general than may be expected. While we derive the formulas from which the derivatives of basis functions may be calculated for the linear triangles, the same implementation is used for all the so-called ***isoparametric*** elements in the `SOFEA` toolbox.

To evaluate the conductivity matrix, we need to be able to calculate the derivatives of the basis functions with respect to x and y. Equations (5.11–5.13) define the functions over the standard triangle in terms of ξ and η. Therefore, to express $\partial N_i / \partial x$ we use the chain rule

$$\frac{\partial N_i}{\partial x} = \frac{\partial N_i}{\partial \xi} \frac{\partial \xi}{\partial x} + \frac{\partial N_i}{\partial \eta} \frac{\partial \eta}{\partial x} \tag{5.30}$$

$$\frac{\partial N_i}{\partial y} = \frac{\partial N_i}{\partial \xi} \frac{\partial \xi}{\partial y} + \frac{\partial N_i}{\partial \eta} \frac{\partial \eta}{\partial y} \tag{5.31}$$

For the purpose of this discussion, the function that is being differentiated does not really matter. We will replace it with a $\heartsuit$, while we arrange the above equation into a matrix expression

$$\left[\frac{\partial \heartsuit}{\partial x}, \frac{\partial \heartsuit}{\partial y} \right] = \left[\frac{\partial \heartsuit}{\partial \xi}, \frac{\partial \heartsuit}{\partial \eta} \right] \begin{bmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \xi}{\partial y} \\ \frac{\partial \eta}{\partial x} & \frac{\partial \eta}{\partial y} \end{bmatrix} . \tag{5.32}$$

The derivatives are arranged in row matrices is that these objects are *gradients* of the $\heartsuit$ function [compare with (4.17)]. The matrix

$$\left[\widetilde{J}\right] = \begin{bmatrix} \dfrac{\partial \xi}{\partial x} & \dfrac{\partial \xi}{\partial y} \\ \dfrac{\partial \eta}{\partial x} & \dfrac{\partial \eta}{\partial y} \end{bmatrix} , \tag{5.33}$$

is the ***Jacobian matrix*** of the mapping $\xi = \xi(x, y), \eta = \eta(x, y)$, which is the inverse of the map $x = x(\xi, \eta), y = y(\xi, \eta)$ of equation (5.15). The question is how to evaluate the partial derivatives of the type $\partial\xi/\partial x$, since the inverse of the map (5.15) is not known (at least not in general). If we start the chain rule from the other end (switching the role of the variables), we obtain

$$\left[\frac{\partial \heartsuit}{\partial \xi}, \frac{\partial \heartsuit}{\partial \eta}\right] = \left[\frac{\partial \heartsuit}{\partial x}, \frac{\partial \heartsuit}{\partial y}\right] \begin{bmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial x}{\partial \eta} \\ \dfrac{\partial y}{\partial \xi} & \dfrac{\partial y}{\partial \eta} \end{bmatrix} \tag{5.34}$$

and inverting the Jacobian matrix in equation (5.32) we get

$$\left[\frac{\partial \heartsuit}{\partial \xi}, \frac{\partial \heartsuit}{\partial \eta}\right] = \left[\frac{\partial \heartsuit}{\partial x}, \frac{\partial \heartsuit}{\partial y}\right] \left[\widetilde{J}\right]^{-1} . \tag{5.35}$$

Comparing (5.34) and (5.35) yields

$$[J] = \begin{bmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial x}{\partial \eta} \\ \dfrac{\partial y}{\partial \xi} & \dfrac{\partial y}{\partial \eta} \end{bmatrix} = \left[\widetilde{J}\right]^{-1} , \tag{5.36}$$

where $[J]$ is the ***Jacobian matrix*** of the map (5.15). The elements of $[J]$ are directly available from the matrix in (5.15). However, even more useful is to start from (5.14), and by definition the Jacobian matrix is then

$$[J] = \begin{bmatrix} \displaystyle\sum_{i=1}^{3} \frac{\partial N_i}{\partial \xi} x_i \, , & \displaystyle\sum_{i=1}^{3} \frac{\partial N_i}{\partial \eta} x_i \\ \displaystyle\sum_{i=1}^{3} \frac{\partial N_i}{\partial \xi} y_i \, , & \displaystyle\sum_{i=1}^{3} \frac{\partial N_i}{\partial \eta} y_i \end{bmatrix} , \tag{5.37}$$

Note that the Jacobian matrix may be expressed as the product of two matrices:

$$[J] = [\mathtt{x}]^T [\mathtt{Nder}] , \tag{5.38}$$

where $[\mathtt{x}]$ collects the coordinates of the nodes (three nodes, for the triangle)

$$[\mathtt{x}] = \begin{bmatrix} x_1 \, , \, y_1 \\ x_2 \, , \, y_2 \\ x_3 \, , \, y_3 \end{bmatrix} , \tag{5.39}$$

and $[\mathtt{Nder}]$ collects in each row the gradient of the basis function with respect to the parametric coordinates

$$[\mathtt{Nder}] = \begin{bmatrix} \dfrac{\partial N_1}{\partial \xi} \, , & \dfrac{\partial N_1}{\partial \eta} \\ \dfrac{\partial N_2}{\partial \xi} \, , & \dfrac{\partial N_2}{\partial \eta} \\ \dfrac{\partial N_3}{\partial \xi} \, , & \dfrac{\partial N_3}{\partial \eta} \end{bmatrix} , \tag{5.40}$$

The calculation of the spatial derivatives by an isoparametric geometric cell (recall that finite elements in `SOFEA` represent the calculation of basis functions and their derivatives in the `gcell` class) is a straightforward rewrite of the formulas above. The method `Ndermat_spatial` takes three arguments: a descendent of the class `gcell`, and two arrays (5.40) and (5.39). The dimensions of the two arrays are (line `0013`): `nbfuns`= number of basis functions, and `dim`= number of space dimensions (= 2 for the triangle).

```
0013 function [Nspatialder,detJ] = Ndermat_spatial[3] (self, Nder, x)
0014     [nbfuns,dim] = size(Nder);
0015     if (size(Nder) ~= size(x))
0016         error('Wrong dimensions of arguments!');
0017     end
```

The Matlab code on line `0018` reflects literally the formula (5.38).

```
0018     J = x' * Nder;% Compute the Jacobian matrix
0019     detJ = det(J);% Compute the Jacobian
```

The Jacobian (determinant of the Jacobian matrix) should be positive. An error is reported when the Jacobian is non-positive; the generic case is treated in line `0023`, which literally transcribes equation (5.32).

```
0020     if (detJ <= 0) % trouble
0021         error('Non-positive Jacobian');
0022     else % the generic case
0023         Nspatialder = Nder * inv(J);
0024     end
0025     return;
0026 end
```

Since the Jacobian is needed both to evaluate the derivatives of the basis functions, and to numerically integrate, it makes sense to compute it only once. Therefore, the method `Ndermat_spatial` discussed in Section 5.6 returns both the array of derivatives and the Jacobian.

To round off the discussion in this section, we need to present the code that evaluates the basis functions (5.11–5.13) and the derivatives of the basis functions with respect to the parametric coordinates ξ, η. For the linear triangle (class `gcell_T3`) the two methods are delightfully simple: method `Nmat` computes a column array of basis function values, N_j in row j, given the parametric coordinates $\xi \leftarrow$ `param_coords(1)`, $\eta \leftarrow$ `param_coords(2)`.

```
0008 function val = Nmat[4](self,param_coords)
0009     val = [(1 - param_coords(1) - param_coords(2));...
0010         param_coords(1); ...
0011         param_coords(2)];
0012     return;
0013 end
```

The method `Nder_param` returns an array with three rows (one for each basis function), with the gradient of the basis function j with respect to ξ, η in row j.

```
0010 function val = Nder_param[5](self, param_coords)
```

---

[3] Folder: `SOFEA/classes/gcell/@gcell`

[4] Folder: `SOFEA/classes/gcell/@gcell_T3`

[5] Folder: `SOFEA/classes/gcell/@gcell_T3`

```
    val = [-1 -1; ...
           +1  0; ...
            0 +1];
    return;
end
```

5.7 Numerical integration

Stepping closely along the lines of the discussion in Section 2.8, we formulate the numerical integration procedure for the linear triangle. We begin by highlighting the role of the Jacobian matrix.

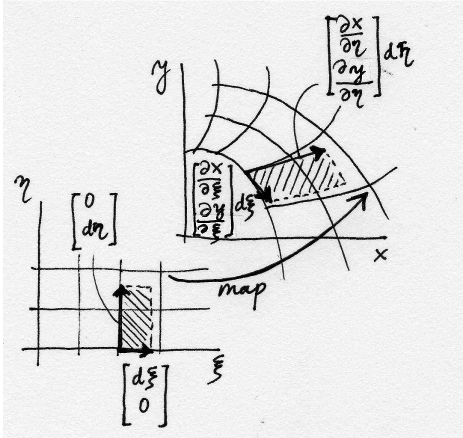
Consider a map from ξ, η to x, y: a slight generalization of (5.14) in that the map is not necessarily linear (see Figure 5.9)

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x(\xi,\eta) \\ y(\xi,\eta) \end{bmatrix} . \tag{5.41}$$

The parallelogram (rectangle) generated by the vectors $[\mathrm{d}\xi, 0]^T$ and $[0, \mathrm{d}\eta]^T$ (given in components in the Cartesian coordinate system ξ, η), has the area of (× is the cross product symbol)

$$\begin{bmatrix} \mathrm{d}\xi \\ 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ \mathrm{d}\eta \end{bmatrix} = \mathrm{d}\xi \mathrm{d}\eta .$$

Remember, this is happening in two dimensions: the cross product is a scalar.



$\begin{bmatrix} \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \eta} \end{bmatrix} d\eta$
y
$\begin{bmatrix} \frac{\partial x}{\partial \xi} \\ \frac{\partial y}{\partial \xi} \end{bmatrix} d\xi$
η
$\begin{bmatrix} 0 \\ d\eta \end{bmatrix}$
x
map
$\begin{bmatrix} d\xi \\ 0 \end{bmatrix}$
ξ

Fig. 5.9. Mapping of areas for a general map between coordinates

The two vectors $[\mathrm{d}\xi, 0]^T$ and $[0, \mathrm{d}\eta]^T$ are mapped by the map (5.41) to vectors

$$\begin{bmatrix} \mathrm{d}\xi \\ 0 \end{bmatrix} \longrightarrow \mathrm{d}\xi \begin{bmatrix} \dfrac{\partial x}{\partial \xi} \\ \dfrac{\partial y}{\partial \xi} \end{bmatrix} , \quad \begin{bmatrix} 0 \\ \mathrm{d}\eta \end{bmatrix} \longrightarrow \mathrm{d}\eta \begin{bmatrix} \dfrac{\partial x}{\partial \eta} \\ \dfrac{\partial y}{\partial \eta} \end{bmatrix} , \tag{5.42}$$

where the square brackets hold components in the standard Cartesian basis. Note that these vectors are ***tangent*** to the coordinate curves, which consist of the points in the physical space x, y that are maps of the curves $\xi = \text{const}$ and $\eta = \text{const}$. The area of the hatched parallelogram in Figure 5.9 is

$$\mathrm{d}\xi \begin{bmatrix} \dfrac{\partial x}{\partial \xi} \\ \dfrac{\partial y}{\partial \xi} \end{bmatrix} \times \mathrm{d}\eta \begin{bmatrix} \dfrac{\partial x}{\partial \eta} \\ \dfrac{\partial y}{\partial \eta} \end{bmatrix} = \mathrm{d}\xi\mathrm{d}\eta \begin{bmatrix} \dfrac{\partial x}{\partial \xi} \\ \dfrac{\partial y}{\partial \xi} \end{bmatrix} \times \begin{bmatrix} \dfrac{\partial x}{\partial \eta} \\ \dfrac{\partial y}{\partial \eta} \end{bmatrix} . \tag{5.43}$$

Compare this equation with (5.36): the two vectors in the cross product are the columns of the Jacobian matrix from (5.36). In fact, the cross product of the columns is the determinant of the Jacobian matrix (or, as the determinant is known, the Jacobian). Therefore, the map (5.41) maps areas as

$$\mathrm{d}\xi\mathrm{d}\eta \longrightarrow \mathrm{d}\xi\mathrm{d}\eta \det[J] . \tag{5.44}$$

As a consequence of (5.44), we have the following change of coordinates in integrals:

$$\int_{S_{[x,y]}} f(x,y)\mathrm{d}x\mathrm{d}y = \int_{S_{[\xi,\eta]}} f(\xi,\eta) \det[J(\xi,\eta)]\, \mathrm{d}\xi\mathrm{d}\eta . \tag{5.45}$$

Numerical quadrature rules take advantage of the relative ease with which these rules may be formulated on standard shapes, triangles, squares, cubes, etc. Thus, the integral on the left of (5.45) will be approximated as

$$\int_{S_{[x,y]}} f(x,y)\mathrm{d}x\mathrm{d}y \approx \sum_{k=1}^{M} f(\xi_k,\eta_k) \det[J(\xi_k,\eta_k)]\, W_k . \tag{5.46}$$

We will introduce two integration rules for the standard triangle, one-point and three-point quadrature, but many other rules are available: a number of authors have compiled tables, see for instance Hughes' book [Hug00]. The 1-point rule will be able to integrate linear polynomials in ξ, η exactly, and the 3-point does the job for up to quadratic polynomials in ξ, η. Table 5.1 gives the coordinates of the integration points, and their weights.

Rule	Coordinates ξ_j, η_j	Weights W_j	Integrates exactly
1-point	1/3, 1/3	1/2	linear polynomial
	2/3, 1/6	1/6	
3-point	1/6, 2/3	1/6	quadratic polynomial
	1/6, 1/6	1/6	

Table 5.1. Numerical integration rules on the standard triangle

5.8 Conductivity matrix

As already discussed in Chapter 3, all the problem dependent code is concentrated in a descendent of the `feblock` class. In particular, the two-dimensional heat diffusion model of this chapter is implemented in the `feblock_diffusion` finite element block class.

The conductivity and other matrices are computed by evaluating the contributions from each element separately, storing these contributions element-by-element in a cell array, and then finally assembling all the element contributions into the overall system matrix. Thus, the method `conductivity` returns the array `ems` of element matrix objects (class `elemat`), each of which represents the conductivity matrix of a single element.

The method begins by retrieving some information from the parent class, such as `gcells` (cell array of the geometric cells), `integration_rule`, and the material `mat`.

```
function ems = conductivity[6](self, geom, theta)
    gcells = get(self.feblock,'gcells');
    ngcells = length(gcells);
    nfens = get(gcells(1),'nfens');
    dim = get(geom,'dim');
    % Pre-allocate the element matrices
    ems(1:ngcells) = deal(elemat);
    % Integration rule
    integration_rule = get(self.feblock, 'integration_rule');
    pc = get(integration_rule, 'param_coords');
    w  = get(integration_rule, 'weights');
    npts_per_gcell = get(integration_rule, 'npts');
    % Material
    mat = get(self.feblock, 'mater');
    kappa = get(mat,'conductivity');
```

The loop over all the geometric cells starts with the retrieval of the connectivity (i.e. the numbers of the nodes which are connected together by the cell), and of the array of the node coordinates, `x` (compare with (5.39)). Then, the element conductivity matrix `Ke` is initialized to zero, and the loop over all the quadrature points may begin.

```
    % Now loop over all gcells in the block
    for i=1:ngcells
        conn = get(gcells(i), 'conn'); % connectivity
        x = gather(geom,conn,'values','noreshape');% node coord
        Ke = zeros(nfens); % element matrix
```

The loop over the integration points starts with calculation of the gradients of the basis functions with respect to the parametric coordinates, see array (5.40). Then compute the spatial derivatives of the basis functions, `Nspder`, and the Jacobian, `detJ`.

```
        % Loop over all integration points
        for j=1:npts_per_gcell
            Nder = Ndermat_param (gcells(i), pc(j,:));
            [Nspder,detJ] = Ndermat_spatial(gcells(i), Nder, x);
```

Often for orthotropic materials the axes of orthotropy vary from point-to-point. In that case it makes sense to describe the material properties in local Cartesian coordinates, and then allow the finite element block to define a transformation matrix between the local coordinate directions and the global Cartesian basis: refer to Figure 5.10. The attribute of a material parameter is thus the material conductivity in the rotated basis, $\boldsymbol{e}_{\overline{x}}$, $\boldsymbol{e}_{\overline{y}}$

---

[6] Folder: `SOFEA/classes/feblock/@feblock_diffusion`

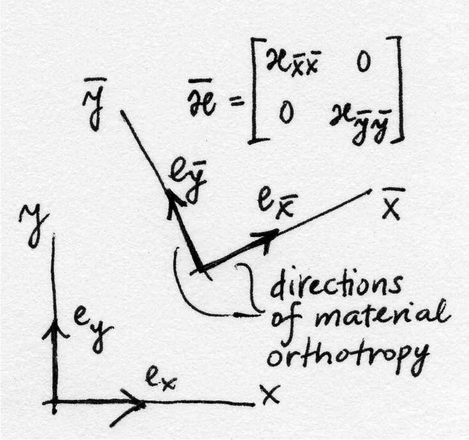

$\bar{y}$
$\bar{\varkappa} = \begin{bmatrix} \varkappa_{\bar{x}\bar{x}} & 0 \\ 0 & \varkappa_{\bar{y}\bar{y}} \end{bmatrix}$
$e_{\bar{y}}$
$e_{\bar{x}}$
$\bar{x}$
y
directions of material orthotropy
e_y
e_x
x

Fig. 5.10. Directions of material orthotropy

$$[\overline{\kappa}] = \begin{bmatrix} \kappa_{\overline{x}} & 0 \\ 0 & \kappa_{\overline{y}} \end{bmatrix} \tag{5.47}$$

which is rotated using the transformation matrix

$$[R_m] = \left[[\boldsymbol{e}_{\overline{x}}] \; [\boldsymbol{e}_{\overline{y}}] \right] \tag{5.48}$$

whose columns are the components of the basis vectors $\boldsymbol{e}_{\overline{x}}$, $\boldsymbol{e}_{\overline{y}}$ in the global Cartesian coordinates. in The material conductivity matrix in the global basis is then expressed using the ordinary transformation rule

$$[\kappa] = [R_m][\overline{\kappa}][R_m]^T \tag{5.49}$$

Compute the local material directions.

```
            Rm=material_directions(self,...
                map_two_xyz(gcells(i), pc(j,:),x),x'*Nder);
```

Now exercise the integration rule.

```
            Ke = Ke + Nspder*Rm*kappa*Rm'*Nspder' * detJ * w(j);
```

Finally, the computed element conductivity matrix `Ke` is stored in the `ems(i)` object, both the matrix itself and the equation numbers that go with each column and each row.

```
        ems(i)=set(ems(i), 'mat', Ke);
        ems(i)=set(ems(i), 'eqnums',gather(temp,conn,'eqnums'));
```

5.9 Surface heat transfer matrix and load

While in the preceding section the required integrals were over the area of the domain S_c, the surface heat transfer matrix (5.28) and the surface heats transfer load (5.27) (and also the prescribed heat flux load (5.26)) require integration over the bounding curve, $C_{c,3}$ (or $C_{c,2}$). Therefore, since the area integrals are being performed over the area of the triangles in the mesh, the curve integrals will be evaluated over the edges of these triangles.

Evaluating the basis functions (5.11–5.13) along the edges of the standard triangle, we may observe that the basis function associated with the opposite vertex is identically zero, and the other two at the end-points of the edge vary linearly along the edge. In fact, completely in agreement with the basis functions (2.20) defined on the line element L2. Therefore, integrating an expression along the edge of the triangle T3 that connects nodes i, j yields exactly the same result as integrating along the line element L2 that connects nodes i, j. However, the two approaches are not the same thing: if, for the purpose of numerical integration, we use the element L2, the design of the numerical integration code will be reusable: the same piece of code may be used to integrate quantities along a curve which is tiled with finite element edges with linear variation of basis functions – the triangle T3, the quadrilateral Q4, the hexahedron H8, the tetrahedron T4.

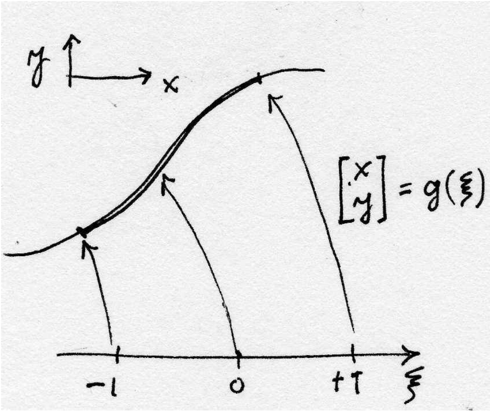


Fig. 5.11. Mapping of the standard interval to a Cartesian space

Let us take for instance equation (5.26). The goal is to evaluate an integral of the form

$$\int_C f(\boldsymbol{p})\,\mathrm{d}C \tag{5.50}$$

where we will assume that the curve C may be "embedded" in a three-dimensional, two-dimensional, or one-dimensional Euclidean space (i.e. it may be a spatial curve, plane curve, or just an interval on the real line). Correspondingly, the point on the curve $\boldsymbol{p}$ will have appropriate number of components. To perform the integral, the elementary length $\mathrm{d}C$ is needed.

The point $\boldsymbol{p}$ on the curve will be assumed to be the result of the mapping the standard interval $-1 \leq \xi \leq +1$ (compare with the 1-D map (2.23), and refer to Figure 5.11, where the map is two-dimensional)

$$\boldsymbol{p} = \boldsymbol{g}(\xi)\ . \tag{5.51}$$

For two closely spaced points on the curve, $\boldsymbol{p}(\xi)$ and $\boldsymbol{p}(\xi + \Delta\xi)$, where $\Delta\xi$ is the distance between the two points in the standard interval, the second point may be obtained from the first using the first two terms of the Taylor series as

$$\boldsymbol{p}(\xi + \Delta\xi) = \boldsymbol{p}(\xi) + \frac{\partial \boldsymbol{p}(\xi + \varepsilon\Delta\xi)}{\partial\xi}\Delta\xi \qquad 0 \leq \varepsilon \leq 1. \tag{5.52}$$

The two points may be connected with a vector approximately tracking the curve (see Figure 5.12),

$$\boldsymbol{p}(\xi+\Delta\xi)-\boldsymbol{p}(\xi)=\frac{\partial \boldsymbol{p}(\xi+\varepsilon\Delta\xi)}{\partial\xi}\Delta\xi\ ,$$

whose length (squared) is

$$(\Delta C)^2=\left(\frac{\partial \boldsymbol{p}(\xi+\varepsilon\Delta\xi)}{\partial\xi}\Delta\xi\right)\cdot\left(\frac{\partial \boldsymbol{p}(\xi+\varepsilon\Delta\xi)}{\partial\xi}\Delta\xi\right)=\left|\left|\frac{\partial \boldsymbol{p}(\xi+\varepsilon\Delta\xi)}{\partial\xi}\right|\right|^2(\Delta\xi)^2\ .$$

Skipping over the details, we may conclude that for infinitesimally short intervals, $\Delta\xi\to \mathrm{d}\xi$, the following relationship is obtained

$$\mathrm{d}C=\left|\left|\frac{\partial \boldsymbol{p}(\xi)}{\partial\xi}\right|\right|\mathrm{d}\xi\ , \tag{5.53}$$

where $\frac{\partial \boldsymbol{p}(\xi)}{\partial\xi}$ is the vector ***tangent*** to the curve at ξ, and $\left|\left|\frac{\partial \boldsymbol{p}(\xi)}{\partial\xi}\right|\right|$ is the Jacobian to be used in the change-of-variables device for the curve integral.

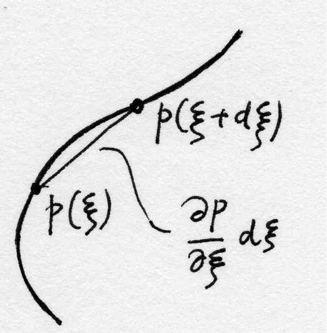


Fig. 5.12. Length of a curve

Let us now specialize these developments to the L2 element: the map (5.54) reads

$$\boldsymbol{p}=\boldsymbol{g}(\xi)=\sum_{i=1}^{2}N_i(\xi)\boldsymbol{x}_i\ . \tag{5.54}$$

where N_i are given by (2.25). Therefore, the tangent vector (see (5.53)) reads

$$\frac{\partial \boldsymbol{p}(\xi)}{\partial\xi}=\frac{\boldsymbol{x}_2-\boldsymbol{x}_1}{2}$$

and the Jacobian is $h/2$, where $h=||\boldsymbol{x}_2-\boldsymbol{x}_1||$ is the length of the element.

Of course, for general elements with n nodes, the implementation confused the tangent as

$$\frac{\partial \boldsymbol{p}(\xi)}{\partial\xi}=\texttt{x'*Nder}\ ,$$

using the following two matrices ,

$$[\mathrm{x}]=\begin{bmatrix}x_1\ ,\ y_1\\ x_2\ ,\ y_2\\ \ldots,\ \ldots\\ x_n\ ,\ y_n\end{bmatrix}\ , \tag{5.55}$$

where the number of columns is equal to the number of spatial dimensions, 1, 2 (which is assumed in (5.55)), or 3, and [`Nder`] collects in each row the gradient of the basis function with respect to the parametric coordinate

$$[\texttt{Nder}] = \begin{bmatrix} \dfrac{\partial N_1}{\partial \xi} \\ \dfrac{\partial N_2}{\partial \xi} \\ \dots \\ \dfrac{\partial N_n}{\partial \xi} \end{bmatrix}, \tag{5.56}$$

These matrices should be compared with those defined for the triangle T3, equations (5.39) and (5.40). The only difference is the number of space dimensions, the number of basis functions, and the number of parametric dimensions; all of these are taken account of by the Matlab code automatically.

The method `surface_transfer_loads` is a rewrite of the above formulas. The setup is straightforward and is omitted.

```
function evs = surface_transfer_loads[7] (self, geom, temp, amb)
...
```

The main work is done in this loop: the interpolated values of the nodal ambient temperatures are retrieved from the field `amb`, and `He` is computed as the heat surface transfer matrix for the element. Note the calculation of the Jacobian: it is in evaluated from the matrix of tangent vectors `x'*Nder`.

```
    % surface transfer coefficient
    h = self.surface_transfer;
    % Now loop over all gcells in the block
    for i=1:ngcells
        conn = get(gcells(i), 'conn'); % connectivity
        pT = gather(amb, conn, 'prescribed_values');
        if norm (pT) ~= 0
            x = gather(geom, conn, 'values', 'noreshape'); % node coord
            He = zeros(nfens); % element matrix
            % Loop over all integration points
            for j=1:npts_per_gcell
                N = Nmat(gcells(i), pc(j,:));
                Nder = Ndermat_param (gcells(i), pc(j,:));
                detJ = Jacobian(gcells(i),x'*Nder);
                He = He + h*N*N' * detJ * w(j);
            end
```

And the load vector is calculated and stored to be returned from this method.

```
            evs(i) = set(evs(i), 'vec', He*pT);
            evs(i) = set(evs(i), 'eqnums', ...
                gather(temp, conn, 'eqnums'));
```

---

[7] Folder: `SOFEA/classes/feblock/@feblock_diffusion`

6

Steady-state heat diffusion solutions

6.1 Steady-state diffusion equation

The ordinary differential equations that result from discretization in space, equations (5.29), lead to steady-state solutions when $\partial T_i(t)/\partial t = 0$, and $\partial \overline{T}_i(t)/\partial t = 0$. The latter are a sine qua non condition, while the former follows when all the transients in the solution decay (in infinite time, in general). Substituting into (5.29), we obtain

$$\sum_{\text{free } i} K_{ji}T_i + \sum_{\text{free } i} H_{ji}T_i = -\sum_{\text{prescribed } i} \overline{K}_{ji}\overline{T}_i + L_{Q,j} + L_{q2,j} + L_{q3,j} \quad \forall \text{ free } j, \tag{6.1}$$

which is a system of linear equations for the unknown nodal temperatures. The nodal temperatures are now just numbers, not functions.

6.2 Thick-walled tube

The first example is a thick-walled rectangular tube, with the outside temperature being prescribed as zero, and the interior surface (perfectly) insulated. The material is isotropic. As shown in Figure 6.1, the planes of symmetry may be used to reduce the size of the problem. Therefore, only one quarter is discretized, and perfect insulation is applied at the symmetry planes (no heat flow through the symmetry planes). There is a distributed heat source in the material (for instance, heat released by curing cement paste).

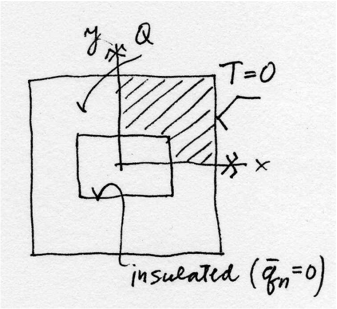


Fig. 6.1. Heat diffusion in a thick-walled rectangular tube

The Matlab script is `lshape1`[1]. The first few lines define some ancillary variables, and then the two-dimensional mesh generator of triangle meshes is invoked. The generator is thoroughly described in the user's guide `targe2_users_guide.pdf`[2], but we will describe a little bit the Matlab interface. The first argument is a cell array, each element a string (character array), with one command for the mesh generator. Thus, the first six strings define curves that bound the domain (straight-line segments), the line `0011` defines a subregion (chunk of area to be covered with triangles), and the last line defines the mesh size. The second argument to `targe2_mesher` is the thickness of the slab (of no interest here).

```
0001 kappa=[0.2 0; 0 0.2]; % conductivity matrix
0002 Q=0.01100; % uniform heat source
0003 num_integ_pts=1; % 1-point quadrature
0004 [fens,gcells] = targe2_mesher({...textcolorcomment
0005         ['curve 1 line 20 0 48 0'],...
0006         ['curve 2 line 48 0 48 48'],...
0007         ['curve 3 line 48 48 0 48'],...
0008         ['curve 4 line 0 48 0 13'],...
0009         ['curve 5 line 0 13 20 13'],...
0010         ['curve 6 line 20 13 20 0'],...
0011         'subregion 1  property 1 boundary 1 2 3 4 5 6',...
0012         ['m-ctl-point constant 3.5']
0013         }, 1.0);
```

Next, the material object appropriate for heat diffusion is created, and supplied the material conductivity matrix $\boldsymbol{\kappa}$, and the heat source Q. The finite element block of class `feblock_diffusion` is created, with attributes: the material, the array of geometric cells, and an integration rule (class `tri_rule` is used for triangles).

```
0014 mater=mater_diffusion (struct('conductivity',kappa,'source',Q));
0015 feb = feblock_diffusion (struct ('mater',mater,...
0016     'gcells',gcells,...
0017     'integration_rule',tri_rule(num_integ_pts)));
```

Two fields are created: `geom` represents the geometry (i.e. the locations of the nodes), and it is therefore initialized from the finite element node array, `fens`; and `theta` represents the temperatures at the nodes, and it is initially undefined, except for the number of nodes `nfens`.

```
0018 geom = field(struct('name',['geom'], 'dim', 2, 'fens',fens));
0019 theta=field(struct('name',['theta'], 'dim', 1, 'nfens',...
0020     get(geom,'nfens')));
```

The essential boundary conditions are next applied to the temperature field. The utility function `fenode_select` is used to select nodes from the `fens` array based on their location: nodes which fall into given bounding boxes are selected ($[x_{lo}\ x_{hi}\ y_{lo}\ y_{hi}] =$ `[48 48 0 48]` and so on for the other box); to avoid problems with precision, the boxes are for the purpose of the "in"-test inflated by 0.01. The array `prescribed` is filled with ones to indicate that all degrees of freedom are to be prescribed, the components to be prescribed are passed as an empty array (line `0026`), which simply means all components are meant. The values to which the temperatures are being prescribed are all zeros. The data defining the essential boundary conditions are set in the field, and then applied (line `0029`). The free node parameters are then assigned global equation numbers.

---

[1] Folder: `SOFEA/examples/diffusion`

[2] Folder: `SOFEA/meshes/targe2`

```
0021 fenids=[fenode_select(fens,struct('box',[48 48 0 48],...
0022     'inflate', 0.01)),...
0023     fenode_select(fens,struct('box',[0 48 48 48],...
0024     'inflate', 0.01))];
0025 prescribed=ones(length(fenids),1);
0026 comp=[];
0027 val=zeros(length(fenids),1);
0028 theta = set_ebc(theta, fenids, prescribed, comp, val);
0029 theta = apply_ebc (theta);
0030 theta = numbereqns (theta);
```

The conductivity matrix is sparse (the linear system to be solved is going to be moderately large, and the efficiency afforded by a sparse matrix is not to be sneezed at), and it is assembled from element conductivity matrices in line `0032`. The heat load vector is assembled from element load vectors, and the solution of the linear system of equations is scattered into the `theta` field.

```
0031 K = start (sparse_sysmat, get(theta, 'neqns'));
0032 K = assemble (K, conductivity(feb, geom, theta));
0033 F = start (sysvec, get(theta, 'neqns'));
0034 F = assemble (F, source_loads(feb, geom, theta));
0035 theta = scatter_sysvec(theta, get(K,'mat')\get(F,'vec'));
```

The last fragment of code takes care of the graphic presentation of the results. The field `colorfield` holds one color (a triple of floating-point numbers) per node, and those colors are obtained from the temperature field by mapping node temperatures to colors (line `0042`). The geometric cells of individual finite elements are plotted twice. Once as a raised colored surface (line `0047`), and the second time as a wireframe in the x, y plane (line `0049`). The resulting graphic is shown in Figure 6.2.

```
0038 gv=graphic_viewer;
0039 gv=reset (gv,[]);
0040 T=get(theta,'values');
0041 dcm=data_colormap(struct('range',[min(T),max(T)],'colormap',jet));
0042 colorfield=field(struct ('name', ['colorfield'], 'data',...
0043     map_data(dcm, T)));
0044 geomT=field(struct ('name', ['geomT'], ...
0045     'data',[get(geom,'values'), get(theta,'values')]));
0046 for i=1:length (gcells)
0047     draw(gcells(i), gv, struct ('x',geomT, 'u',0*geomT,...
0048         'colorfield',colorfield, 'shrink',0.9));
0049     draw(gcells(i), gv, struct ('x',geom, 'u',0*geom, ...
0050         'facecolor','none'));
0051 end
```

6.3 Orthotropic insert

The next example introduces nonzero essential boundary conditions, and orthotropic material properties. A square block of isotropic material is insulated on the vertical edges, and two different temperatures our applied on the horizontal

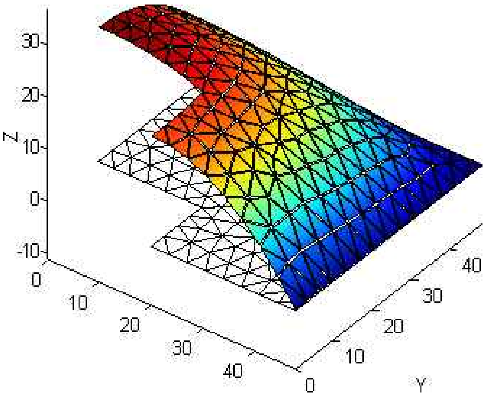


Fig. 6.2. Heat diffusion in a thick-walled rectangular tube: graphic presentation of results

edges. There is a square insert of orthotropic material within the larger square. The orientation of the material axes is indicated in Figure **??**. Physically the insert could be made of parallel fibers (for instance carbon), embedded in a polymer matrix. The fibers conduct heat well, while in the transverse direction the polymer matrix hampers heat conduction. The problem is solved with script `squareinsquare`[3]. The domain consists of two materials, and consequently we define two material conductivity matrices: the inner material has strongly orthotropic properties; the outer material is isotropic. The rotation matrix that defines the local material properties of the insert is setup in line `0005`.

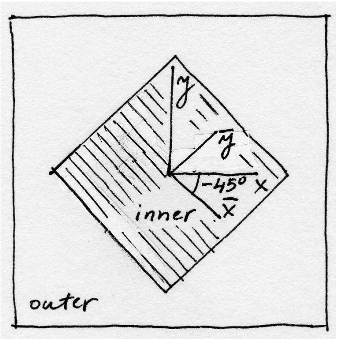


Fig. 6.3. Heat diffusion in inhomogeneous domain with orthotropic material properties

```
0001 kappainner=[2.25 0; 0 0.06]; % orthotropic conductivity matrix
0002 kappaouter=[0.25 0; 0 0.25]; % isotropic conductivity matrix
0003 alpha =-45;% local material orientation angle
0004 ca=cos(2*pi/360*alpha); sa=sin(2*pi/360*alpha);
0005 Rm = [ca, -sa;sa, ca];% local material directions
```

The mesh generator defines the eight boundary segments, and then sets up two subregions: note that the two subregions are assigned different numerical identifiers (1 and 2) to distinguish elements belonging to different subregions.

```
0007 [fens,gcells, groups] = targe2_mesher({...
```

---

[3] Folder: `SOFEA/examples/diffusion`

```
0008         ['curve 1 line -48 -48 48 -48'],...
0009         ['curve 2 line 48 -48 48 48'],...
0010         ['curve 3 line 48 48 -48 48'],...
0011         ['curve 4 line -48 48 -48 -48'],...
0012         ['curve 5 line 0 -31 31 0'],...
0013         ['curve 6 line 31 0 0 31'],...
0014         ['curve 7 line 0 31 -31 0'],...
0015         ['curve 8 line -31 0 0 -31'],...
0016         ['subregion 1  property 1 ' ...
0017         '   boundary 1 2 3 4 -8 -7 -6 -5'],...
0018         ['subregion 2  property 2 '...
0019         '   boundary 5 6 7 8'],...
0020         ['m-ctl-point constant 4.75']
0021         }, 1.0);
```

The inner subregion consists of the geometric cells `gcells(groups{2})` (`groups{2}` is a list of indexes of the cells that belong to the subregion 2). Note that the local material directions matrix is being supplied to the finite element block constructor (line `0027`).

```
0023 materinner=mater_diffusion(struct('conductivity',kappainner,...
0024     'source',0));
0025 febinner = feblock_diffusion(struct ('mater',materinner,...
0026     'gcells',gcells(groups{2}),...
0027     'integration_rule',tri_rule(num_integ_pts),'Rm',Rm));
0028 materouter=mater_diffusion(struct('conductivity',kappaouter,...
0029     'source',0));
0030 febouter = feblock_diffusion(struct ('mater',materouter,...
0031     'gcells',gcells(groups{1}),...
0032     'integration_rule',tri_rule(num_integ_pts)));
```

The boundary conditions are straightforward, but notice that the two horizontal edges are being assigned different, nonzero, temperatures.

```
0035 fenids=[fenode_select(fens,struct('box',[-48 48 -48 -48],...
0036     'inflate', 0.01))];
0037 prescribed=ones(length(fenids),1);
0038 comp=[];
0039 val=zeros(length(fenids),1)+20;% ambient temperature
0040 theta = set_ebc(theta, fenids, prescribed, comp, val);
0041 fenids=[fenode_select(fens,struct('box',[-48 48 48 48],...
0042     'inflate', 0.01))];
0043 prescribed=ones(length(fenids),1);
0044 comp=[];
0045 val=zeros(length(fenids),1)+57;% hot inner surface
0046 theta = set_ebc(theta, fenids, prescribed, comp, val);
0047 theta = apply_ebc (theta);
```

When assembling the conductivity matrix, the contributions from the two blocks are assembled separately. The thermal loads corresponding to nonzero essential boundary conditions (conductivity only: recall that this is steady-state) are assembled only for the outer subregion block, since there are no boundary conditions on the boundary of the inner block.

```
0049 K = start (sparse_sysmat, get(theta, 'neqns'));
```

```
K = assemble (K, conductivity(febinner, geom, theta));
K = assemble (K, conductivity(febouter, geom, theta));
F = start (sysvec, get(theta, 'neqns'));
F = assemble (F, nz_ebc_loads_conductivity(febouter, geom, theta));
```

The results are presented in Figure 6.4. The distorting effect of the insert is noteworthy: in one direction the insert acts as a heat sink/source, in the perpendicular direction it is an insulator.

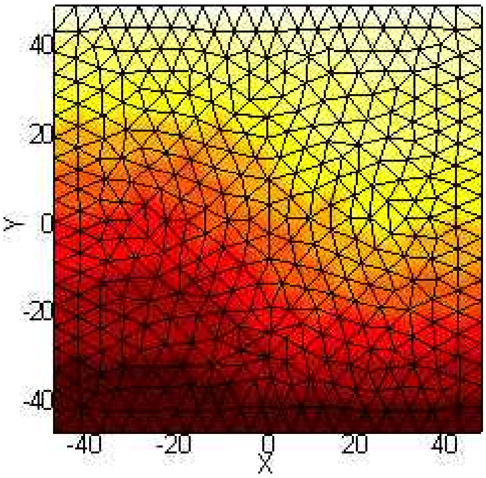


Fig. 6.4. Heat diffusion in inhomogeneous domain with orthotropic material properties: temperature distribution. Notice the distorting effect of the insert

6.4 The T4 NAFEMS Benchmark

This problem is one of the NAFEMS (National Agency for Finite Element Methods and Standards (UK)) benchmark tests for thermal analyses. It consists of 2d region 0.6 meter wide by 1 meter high, with a fixed temperature of 100°C100 on the lower boundary, perfect insulator on the left boundary, and a heat transfer at 750W/m^2°C on the other two boundaries (see Figure 6.5). The material in the region has a thermal conductivity of 52W/m°C. The problem is to calculate the steady-state temperature distribution. A complete description of this problem is given in the paper by Cameron, Casey, and Simpson [CCS].

The `SOFEA`solution is in the Matlab script `t4nafems`[4]. Note that also

```
kappa=[52 0; 0 52]; % conductivity matrix
Q=0.0; % uniform heat source
h=750;
num_integ_pts=1; % 1-point quadrature
[fens,gcells,groups,edge_gcells,edge_groups]=targe2_mesher({...
    ['curve 1 line 0 0 0.6 0'],...
    ['curve 2 line 0.6 0 0.6 0.2'],...
    ['curve 3 line 0.6 0.2 0.6 1.0'],...
    ['curve 4 line 0.6 1.0 0 1.0'],...
```

---

[4] Folder: `SOFEA/examples/diffusion`

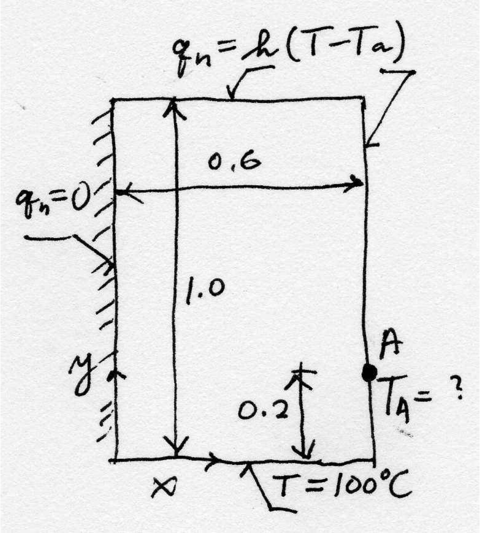


Fig. 6.5. The T4 NAFEMS benchmark geometry and boundary conditions.

```
        ['curve 5 line 0 1.0 0 0'],...
        'subregion 1  property 1 boundary 1 2 3 4 5',...
        ['m-ctl-point constant 0.05']
        }, 1.0);
...
```

Nota bene that two blocks are being created: the first for the triangular elements in the interior of the domain, and the second, `edgefeb`, for the edge elements (line segment elements with two nodes) along the two boundary edges of the domain with the convective boundary condition.

```
edgefeb = feblock_diffusion (struct ('mater',mater,...
    'gcells',edge_gcells([edge_groups{[2, 3, 4]}]),...
    'integration_rule',gauss_rule(1,num_integ_pts),...
    'surface_transfer', h));
...
```

Create a field to represent the prescribed ambient temperature along the boundary. The interior values are never used, only the ones on the boundary. They happen to be all equal to zero, but we will not ignore them in the interest of clarity.

```
amb = clone(theta, ['amb']);
fenids=[
    fenode_select(fens,struct('box',[0.6 0.6 0 1],...
    'inflate', 0.01)),...
    fenode_select(fens,struct('box',[0 1 1 1],...
    'inflate', 0.01))]   ;
prescribed=ones(length(fenids),1);
comp=[];
val=zeros(length(fenids),1)+0.0;
amb = set_ebc(amb, fenids, prescribed, comp, val);
amb = apply_ebc (amb);
```

The essential boundary condition on the temperature field is applied.

```
0036 fenids=[
0037     fenode_select(fens,struct('box',[0. 0.6 0 0],...
0038     'inflate', 0.01))]    ;
0039 prescribed=ones(length(fenids),1);
0040 comp=[];
0041 val=zeros(length(fenids),1)+100.0;
0042 theta = set_ebc(theta, fenids, prescribed, comp, val);
0043 theta = apply_ebc (theta);
0044 theta = numbereqns (theta);
```

The system matrix and the system load vector are assembled, including the surface heats transfer contribution (line `0047`), and the surface heat transfer load (line `0051`). Note that these are computed on the edge element block `edgefeb`. The contribution of the nonzero prescribed temperature is also added in.

```
0045 K = start (sparse_sysmat, get(theta, 'neqns'));
0046 K = assemble (K, conductivity(feb, geom, theta));
0047 K = assemble (K, surface_transfer(edgefeb, geom, theta));
0048 F = start (sysvec, get(theta, 'neqns'));
0049 F = assemble(F, source_loads(feb, geom, theta));
0050 F = assemble(F, nz_ebc_loads_conductivity(feb, geom, theta));
0051 F = assemble(F, surface_transfer_loads(edgefeb,geom,theta,amb));
```

After the solution, the temperature at the node $x = 0.6$, $y = 0.2$ (label A in Figure 6.6), is retrieved with `gather` and printed. The calculated value of 18.2481°C agrees well with the reference solution of 18.3°C. The singularity near the corner where the two kinds of boundary conditions meet (prescribed temperature with convective surface heat transfer) is clearly visible.

```
0052 theta = scatter_sysvec(theta, get(K,'mat')\get(F,'vec'));
0053 gather(theta,fenode_select(fens,...
0054     struct('box',[0.6 0.6 0.2 0.2],'inflate', 0.01)),'values')
```

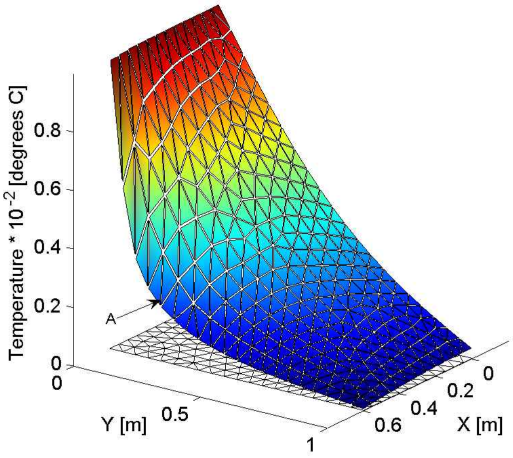


Fig. 6.6. Temperature distribution for the T4 NAFEMS benchmark.

7

Transient heat diffusion solutions

7.1 Discretization in time for transient heat diffusion

The ordinary differential equations (5.29) need to be numerically integrated in time as analytical solutions are not possible in general. Hughes (2000) describes a finite difference method, the ***generalized trapezoidal method*** , including its accuracy and stability properties (Chapter 8, Reference [Hug00]). To unclutter the equations, we will use a matrix notation, with the following definitions:

$$\widetilde{\boldsymbol{K}} = [K_{ji} + H_{ji}], \quad \text{free } j, i, \tag{7.1}$$

for the effective conductivity matrix, which bundles the bulk conductivity with the surface heat transfer matrix,

$$\overline{\boldsymbol{K}} = [\overline{K}_{ji}], \quad \text{free } j, \ \text{prescribed } i, \tag{7.2}$$

for the rectangular conductivity matrix that relates the prescribed temperatures to the heat fluxes,

$$\boldsymbol{C} = [C_{ji}], \quad \text{free } j, i, \tag{7.3}$$

for the capacity matrix,

$$\overline{\boldsymbol{C}} = [\overline{C}_{ji}], \quad \text{free } j, \ \text{prescribed } i, \tag{7.4}$$

for the rectangular capacity matrix that relates the prescribed temperature rates to the heat fluxes, and

$$\boldsymbol{L} = [L_{Q,j} + L_{q2,j} + L_{q3,j}], \quad \text{free } j \ . \tag{7.5}$$

The free temperatures and their rates are collected in column matrices

$$\boldsymbol{T} = [T_j], \quad \dot{\boldsymbol{T}} = \left[\frac{\partial T_i}{\partial t}\right], \quad \text{free } j \ . \tag{7.6}$$

and the prescribed temperatures and their rates

$$\overline{\boldsymbol{T}} = [\overline{T}_j], \quad \dot{\overline{\boldsymbol{T}}} = \left[\frac{\partial \overline{T}_i}{\partial t}\right], \quad \text{prescribed } j \ . \tag{7.7}$$

Therefore, equation (5.29) may be recast as

$$\boldsymbol{C}\dot{\boldsymbol{T}} + \widetilde{\boldsymbol{K}}\boldsymbol{T} + \overline{\boldsymbol{C}}\dot{\overline{\boldsymbol{T}}} + \overline{\boldsymbol{K}}\overline{\boldsymbol{T}} - \boldsymbol{L} = \boldsymbol{0} \ . \tag{7.8}$$

The generalized trapezoidal method proposes to express the relationship between the temperatures and the rates of temperatures at two different time instants, t_n and t_{n+1}, as

$$\theta\dot{\boldsymbol{T}}_{n+1} + (1-\theta)\dot{\boldsymbol{T}}_n = \frac{\boldsymbol{T}_{n+1} - \boldsymbol{T}_n}{\Delta t} \ , \tag{7.9}$$

where a quantity expressed at time t_n is given a subscript n, and $\Delta t = t_{n+1} - t_n$. The free parameter θ is used to control accuracy and stability of the scheme.

Equation (7.9) is applied to the time stepping of (7.8) by writing (7.8) at the two time instants, t_n and t_{n+1}, and then mixing together these two equations. Thus, we add together

$$\theta\left[\boldsymbol{C}\dot{\boldsymbol{T}}_{n+1} + \widetilde{\boldsymbol{K}}\boldsymbol{T}_{n+1} + \overline{\boldsymbol{C}}\dot{\overline{\boldsymbol{T}}}_{n+1} + \overline{\boldsymbol{K}}\overline{\boldsymbol{T}}_{n+1} - \boldsymbol{L}_{n+1}\right] = \boldsymbol{0} \ , \tag{7.10}$$

and

$$(1-\theta)\left[\boldsymbol{C}\dot{\boldsymbol{T}}_{n} + \widetilde{\boldsymbol{K}}\boldsymbol{T}_{n} + \overline{\boldsymbol{C}}\dot{\overline{\boldsymbol{T}}}_{n} + \overline{\boldsymbol{K}}\overline{\boldsymbol{T}}_{n} - \boldsymbol{L}_{n}\right] = \boldsymbol{0} \ , \tag{7.11}$$

and if we assume that equation (7.9) applies not only to the free temperatures, but also to the prescribed temperatures, the mixture of rates (left-hand side of (7.9)) may be replaced with the difference of the temperatures (right hand side of (7.9)). The resulting equation refers only to temperatures at two times, and may be solved to yield $\boldsymbol{T}_{n+1}$, provided $\boldsymbol{T}_n$ is known.

$$\begin{aligned}\left[\frac{1}{\Delta t}\boldsymbol{C} + \theta\widetilde{\boldsymbol{K}}\right]\boldsymbol{T}_{n+1} = {} & \left[\frac{1}{\Delta t}\boldsymbol{C} - (1-\theta)\widetilde{\boldsymbol{K}}\right]\boldsymbol{T}_n + \theta\boldsymbol{L}_{n+1} + (1-\theta)\boldsymbol{L}_n \\ & - \left[\frac{1}{\Delta t}\overline{\boldsymbol{C}} + \theta\overline{\boldsymbol{K}}\right]\overline{\boldsymbol{T}}_{n+1} + \left[\frac{1}{\Delta t}\overline{\boldsymbol{C}} - (1-\theta)\overline{\boldsymbol{K}}\right]\overline{\boldsymbol{T}}_n\end{aligned} \tag{7.12}$$

The form of equation (7.12) is pleasingly symmetric, fully reflective of the blocked nature of these equations. However, for implementation the following form is going to be more profitable:

$$\begin{aligned}\left[\frac{1}{\Delta t}\boldsymbol{C} + \theta\widetilde{\boldsymbol{K}}\right]\boldsymbol{T}_{n+1} = {} & \left[\frac{1}{\Delta t}\boldsymbol{C} - (1-\theta)\widetilde{\boldsymbol{K}}\right]\boldsymbol{T}_n + \theta\boldsymbol{L}_{n+1} + (1-\theta)\boldsymbol{L}_n \\ & -\overline{\boldsymbol{C}}\frac{\overline{\boldsymbol{T}}_{n+1} - \overline{\boldsymbol{T}}_n}{\Delta t} - \overline{\boldsymbol{K}}\left[\theta\overline{\boldsymbol{T}}_{n+1} + (1-\theta)\overline{\boldsymbol{T}}_n\right]\end{aligned} \tag{7.13}$$

The last line in this equation indicates how the contributions from prescribed temperatures (and hence also prescribed temperature rates) may be calculated: the term

$$-\overline{\boldsymbol{C}}\frac{\overline{\boldsymbol{T}}_{n+1} - \overline{\boldsymbol{T}}_n}{\Delta t}$$

introduces the contributions of the temperature rates, since the fraction on the right is an approximation of the temperature rate, and the term

$$-\overline{\boldsymbol{K}}\left[\theta\overline{\boldsymbol{T}}_{n+1} + (1-\theta)\overline{\boldsymbol{T}}_n\right]$$

contributes the effect of prescribed temperatures (in the form of a mixture of temperatures at time n and $n+1$).

Now to the question of how to choose the value of θ: upon closer inspection of equation (7.9) we may conclude that the two choices, $\theta = 0$ and $\theta = 1$, will lead to Euler methods – the ***forward*** (explicit) ***Euler*** for the former, and the ***backward*** (implicit) ***Euler*** for the latter. The value of $\theta = 1/2$ is known as the ***Crank-Nicholson*** method. The explicit Euler method has the limitation of conditional stability, which leads to severe restrictions on the time step. On the other hand, the backward Euler and the Crank-Nicholson are for equations (7.13) unconditionally stable. While the Crank-Nicholson is nominally more accurate than the backward Euler, the latter is often given preference because it tends to eliminate oscillations in the solution.

7.2 Transient diffusion: The T3 NAFEMS Benchmark

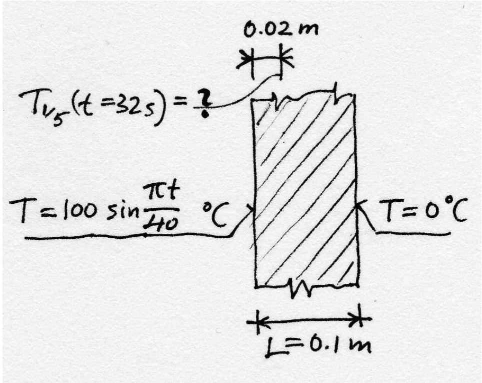


Fig. 7.1. Heat diffusion through a plate (one-dimensional problem), with time-dependent boundary conditions

This test is recommended by the National Agency for Finite Element Methods and Standards (UK), and it is surprisingly exacting. The domain shown in Figure 7.1. One face is held at 0°C, the other face experiences sinusoidal variations in temperature. The temperature at $t = 32$ seconds 0.02 m under the heated face is sought. It is assumed that the plate is very large compared to its thickness, and the problem may therefore be reduced to one dimension, along the thickness. The implementation of transient heat conduction in `SOFEA`is in fact dimension independent, and we simply take care to define the various objects properly for a 1-D problem and the rest follows.

The solution is presented in the Matlab script `t3nafems`[1]. First, the various parameters are defined. Note that the backward Euler method ($\theta = 1$) is selected for the time discretization.

```
kappa=[35.0]; % conductivity matrix
cm = 440.5;% specific heat per unit mass
rho=7200;% mass density
cv =cm* rho;% specific heat per unit volume
Q=0; % uniform heat source
Tampl=100;
Tamb=0;
Tbar =@(t)(Tampl*sin(pi*t/40)+ Tamb);%hot face temperature
num_integ_pts=2; % quadrature
L=0.1;% thickness
dt=0.05; % time step
```

---

[1] Folder: `SOFEA/examples/diffusion`

```
0012 tend= 32; % length of the time interval
0013 t=0;
0014 theta = 1.0; % generalized trapezoidal method
0015 online_graphics= ~true;% plot the solution as it is computed?
0016 n=100*5;% needs to be multiple of five
```

The mesh is created by `block1d`[2], a simple utility which produces a uniformly spaced mesh on the interval $0 \leq x \leq L$. Note that not only the essential boundary conditions are applied to the temperature field, but also the initial condition (which happens to be 0°C).

```
0017 [fens,gcells] = block1d(L,n,1.0); % Mesh
0018 mater = mater_diffusion(struct('conductivity',kappa,...
0019     'specific_heat',cv,'rho',rho,'source',Q));
0020 feb = feblock_diffusion (struct (...
0021     'mater',mater,...
0022     'gcells',gcells,...
0023     'integration_rule',gauss_rule(1,num_integ_pts)));
0024 geom = field(struct ('name',['geom'], 'dim', 1, 'fens',fens));
0025 tempn = field(struct ('name',['temp'], 'dim', 1,...
0026     'nfens',get(geom,'nfens')));
0027 tempn = set_ebc(tempn, 1, 1, 1, Tbar(t));
0028 tempn = set_ebc(tempn, n+1, 1, 1, Tamb);
0029 tempn = apply_ebc (tempn);
0030 tempn = numbereqns (tempn);
0031 tempn = scatter_sysvec(tempn,gather_sysvec(tempn)*0+Tamb);
```

The conductivity and capacity matrix are time independent; we compute them once, and henceforth work only with the arrays `Km` and `Cm`.

```
0032 K = start (dense_sysmat, get(tempn, 'neqns'));
0033 K = assemble (K, conductivity(feb, geom, tempn));
0034 Km = get(K,'mat');
0035 C = start (dense_sysmat, get(tempn, 'neqns'));
0036 C = assemble (C, capacity(feb, geom, tempn));
0037 Cm = get(C,'mat');
```

The time stepping begins. First, the temperature boundary conditions are time-dependent, which means they have to be set for each pass through the time loop (i.e. for each time instant).

```
0038 Tfifth = [];
0039 while t<tend+0.1*dt % Time stepping
...
0046     tempn1 = tempn;
0047     tempn1 = set_ebc(tempn1, 1, 1, 1, Tbar(t+dt));
0048     tempn1 = set_ebc(tempn1, n+1, 1, 1, Tamb);
0049     tempn1 = apply_ebc (tempn1);
```

The thermal loads corresponding to nonzero temperatures and temperature rates are applied next. We may compare the fields that are being passed on lines `0052` and `0054` with (7.13) and the discussion below that equation: The Matlab code is a literal transcription of the formulas. Note that we are directly working with objects of the class `field`, using operator overload (adding and multiplying fields).

---

[2] Folder: `SOFEA/meshes`

```
    F = start (sysvec, get(tempn, 'neqns'));
    F = assemble (F, nz_ebc_loads_conductivity(feb, geom, ...
        theta*tempn1 + (1-theta)*tempn));
    F = assemble (F, nz_ebc_loads_capacity(feb, geom, ...
        (tempn1-tempn)*(1/dt)));
    Tn=gather_sysvec(tempn);
    Tfifth = [Tfifth Tn(n/5+1)];
```

The individual objects in this system of linear equations for `Tn1` are again directly recognizable in formula (7.13).

```
    Tn1 = (1/dt*Cm+theta*Km) \ ((1/dt*Cm-(1-theta)*Km)*Tn+...
        get(F,'vec'));
    tempn = scatter_sysvec(tempn1,Tn1);
    t=t+dt;
end
```

The results are summarized in Figure 7.2. The reference solution is 36.6°C at the time $t = 32$ seconds, and the curve shown in the figure has been obtained with 500 elements through the thickness (yielding 36.16°C). The solution with 15 elements is seen to be in considerable error. This is somewhat surprising, but a closer look at the behavior of the solution during the time interval of interest shows significant temperature gradients near the hot surface, which perhaps explains why it is so expensive to get an accurate solution.

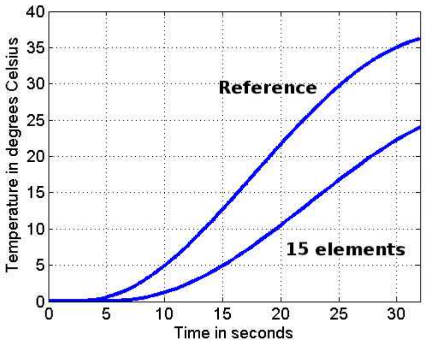


Fig. 7.2. Heat diffusion through a plate (one-dimensional problem): temperature 0.02 m under the heated face

7.3 Transient cooling in a shrink-fitting application

Shrink fitting is a common manufacturing process used to assemble two parts: Figure 7.3. In our case, the cold part is maintained at −10°C prior to the assembly, while the hot part is at 84°C. The temperature of the ambient air is 17°C. The task is to determine how long it will take before the temperature of the hot part drops below 70°C (which is given as a manufacturing constraint).

The problem is solved by the script `shrinkfit`[3]. Let us jump directly into the mesh generation: Figure 7.3 shows the two regions, and the boundary edges

---

[3] Folder: `SOFEA/examples/diffusion`

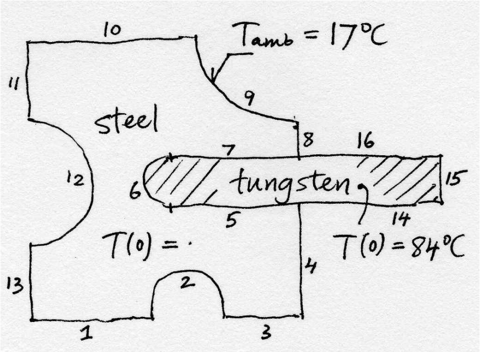


Fig. 7.3. Transient cooling of a shrink-fitted assembly: schematic

(note the numbers next to the edges). The mesh is relatively coarse considering the thickness of some of the geometry – only around three elements through the thickness of the hot part. Even so the mesh has almost 2300 triangular elements, and the transient solution takes a couple of minutes.

```
[fens,gcells,groups,edge_gcells,edge_groups]=targe2_mesher({...
    'curve 1 line 0 0 50 0',...
    'curve 2 arc 50 0 80 0 center 65 -0.001 ',...
    'curve 3 line 80 0 110 0',...
    'curve 4 line 110 0 110 50',...
    'curve 5 line 110 50 65 50 ',...
    'curve 6 arc 65 50 65 70 center 65.001 60  ',...
    'curve 7 line 65 70 110 70',...
    'curve 8 line 110 70 110 85',...
    'curve 9 arc 110 85 65 120 center 110 120 ',...
    'curve 10 line 65 120 0 120',...
    'curve 11 line 0 120 0 85',...
    'curve 12 arc 0 85 0 35 center -0.001 60 rev',...
    'curve 13 line 0 35 0 0',...
    'curve 14 line 110, 50, 160, 50',...
    'curve 15 line 160, 50, 160, 70',...
    'curve 16 line 160, 70, 110, 70',...
    ['subregion 1  property 1 boundary '...
    ' 1 2 3 4 5 6 7 8 9 10 11 12 13'],...
    ['subregion 2  property 2 boundary '...
    ' -5 -6 -7 14 15 16'],...
    ['m-ctl-point constant 3']
    }, 1.0);
```

For the edges that separate the metal from the air, elements to be used in the surface heat transfer needs to be generated (finite element block `efeb`). Note that the interior edges are omitted.

```
feb_steel = feblock_diffusion (struct ('mater',mater_steel,...
...
feb_tungsten = feblock_diffusion (struct ('mater',mater_tungsten,...
...
edge_gcells=edge_gcells([edge_groups{[(1:4) (8:16)]}]);
efeb = feblock_diffusion (struct ('mater',mater_steel,...
    'gcells',edge_gcells,...
0055      'integration_rule',gauss_rule(1,num_integ_pts),...
0056      'surface_transfer', h));
```

The ambient temperature is defined in the field `amb`. The temperature is applied at the nodes associated with the boundary edges in the loop (line `0063`).

```
0060 amb = clone(tempn, ['amb']);
0061 for i= 1:length(edge_gcells)
0062      conn = get(edge_gcells(i),'conn');
0063      amb = set_ebc(amb, conn, conn*0+1, [], conn*0+Ta);
0064 end
0065 amb = apply_ebc (amb);
```

The time stepping loop is almost identical to the example in Section 7.2, except the thermal load vector is based on the convective surface heat transfer.

```
0115      F=assemble(F, surface_transfer_loads(efeb,geom,tempn,amb));
```

The evolution of the lowest and highest temperature in the entire assembly is shown in Figure 7.4: the highest temperature drops below 70°C after around 13 seconds. Obviously, we are not addressing the issue of accuracy, neither in the resolution afforded by the mesh, nor in the selection of the time step. However, the time measurements in an actual manufacturing process are not likely to be accurate to more than half a second.

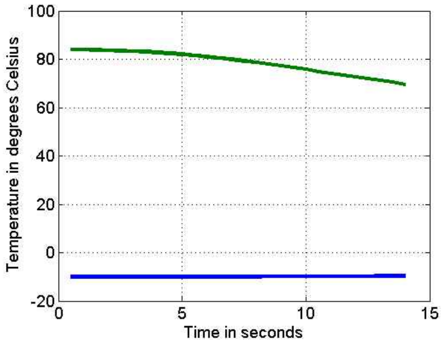


Fig. 7.4. Transient cooling of a shrink-fitted assembly: time evolution of the lowest and highest temperature in the assembly

8

Expanding the library of element types

The linear triangle T3 is not particularly accurate, but for the linear heat diffusion problem it is quite adequate. Nevertheless, we will introduce another couple of elements to expand the scope of the approximation methods discussed so far. This is desirable from a couple of different viewpoints. Firstly, with the linear triangle we have been able to construct basis functions which allow for linear variations in temperature to be represented exactly. Hence, if the exact solution is a constant gradient of temperature, the approximate solution does not involve any discretization error. Unfortunately, the usefulness of this is limited, since constant gradients of temperature are not commonly encountered in applications. If the basis functions can represent higher-order polynomials, for instance quadratic, the resulting method will be able to represent more complex gradients of temperature: linear, in the case of the quadratic variation of temperature. To oversimplify a little bit, the more complex the temperature variations that are reproduced without error, the higher the overall accuracy of the scheme.

Secondly, introducing different element types may enable us to play games with different quadrature schemes. One view of the finite element method could put the basis function above all: the elements are there only to integrate all the expressions that involved the basis functions and their derivatives as accurately as possible (exactly?). However,

8.1 Quadratic triangle T6

The triangle T6 makes it possible to design basis functions that can reproduce quadratic variations of the temperature. More precisely, it will do that in the terms of the coordinates on the standard triangle. As we shall see, the map from the standard triangle will also allow for quadratic temperature variation in the physical space, but more generally it will lead to rational expressions.

The first task will be to formulate the basis functions on the standard triangle, Figure 8.1. To be able to write down a polynomial for a particular basis function that is quadratic in ξ, η, six coefficients will be needed. To determine these coefficients, we will make use of the common device of equipping the basis functions with the Kronecker delta property (2.19). Let us start with the basis function $N_2 = a_0 + a_1\xi + a_2\eta + a_3\xi\eta + a_4\xi^2 + a_5\eta^2$: writing

$$N_2(\xi_k, \eta_k) = \delta_{2k}, \quad \text{for } k = 1, , 6$$

at all six nodes (see Table 8.1), provides us with six equations from which the six coefficients may be determined. That is however tedious and boring: let us use

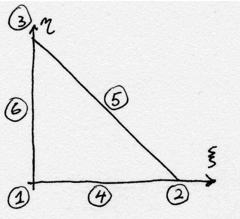


Fig. 8.1. Standard quadratic triangle.

commonsense and guesswork instead. Drawing the standard triangle plane while looking along the η axis we see that three and two nodes respectively align, which obviously makes it possible to make the function N_2 equal to zero in these two locations, and equal to one at node 2 with a Lagrange polynomial

$$N_2 = \frac{(\xi - 0)(\xi - 1/2)}{(1 - 0)(1 - 1/2)} = \xi(2\xi - 1)$$

Similarly, in the other direction we have for $N_3 = \eta(2\eta - 1)$.

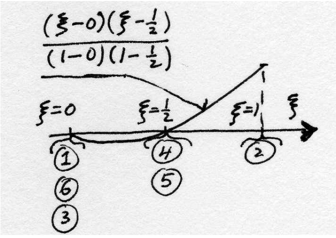


Fig. 8.2. Standard quadratic triangle: one-dimensional view of basis function N_2.

To approach the construction of the other basis functions, we note that both N_2 and N_3 may be written as the normalized product of planes: for N_2 the two planes are $\widehat{p}_2(\xi, \eta) = \xi$ and $\widetilde{p}_2(\xi, \eta) = \xi - 1/2$, and N_2 is written as

$$N_2 = \frac{\widehat{p}_2(\xi, \eta)\widetilde{p}_2(\xi, \eta)}{\widehat{p}_2(1, 0)\widetilde{p}_2(1, 0)} = \xi(2\xi - 1) \ .$$

Similarly for N_3 and N_1: the recipe is to find two planes that go through three nodes and two nodes respectively (but not through the node at which the function is supposed to be equal to one), and normalize their product. For N_1 the planes are $\widehat{p}_1(\xi, \eta) = 1 - \xi - \eta$ (this is the same N_1 as in (5.13)) and $\widetilde{p}_1(\xi, \eta) = 1 - 2\xi - 2\eta$ (compare with Figure 8.3)

$$N_1 = (1 - \xi - \eta)(1 - 2\xi - 2\eta) \ .$$

For the mid-edge nodes, 4, 5, 6, we find planes that pass through two triples of nodes. For instance, for node 6 (see Figure 8.3), the two planes are $\widehat{p}_6(\xi, \eta) = 1-\xi-\eta$ and $\widetilde{p}_6(\xi, \eta) = \eta$ (same as N_3 as in (5.12)

$$N_6 = 4(1 - \xi - \eta)\eta \ .$$

Coordinate	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
ξ	0	1	0	1/2	1/2	0
η	0	0	1	0	1/2	1/2

Table 8.1. Standard quadratic triangle: locations of the nodes

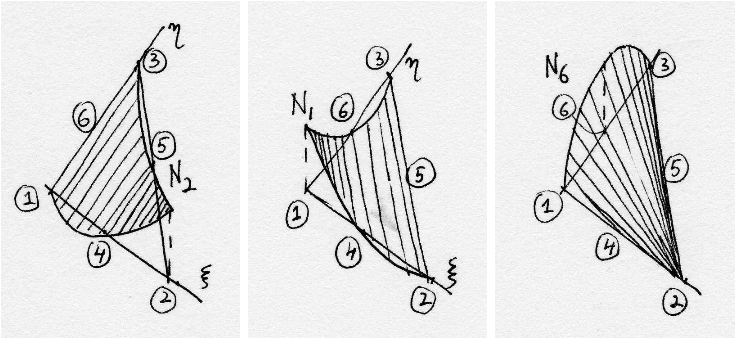


Fig. 8.3. Standard quadratic triangle: Basis functions N_2, N_1, and N_6.

8.2 Quadratic 1-D element L3

8.3 Point element P1

Looking at the `classes/gcell` folder, one may notice the `gcell_X_manifold` class folders with `X`= 0, 1, 2, 3. All `SOFEA` geometric cells are of certain so-called manifold dimension: solids are of manifold dimension 3, surfaces are of dimension 2, while curves and points are of dimensions 1 and 0. Since we commonly solve heat diffusion (and other problems) in domains that are solids, surfaces, and curves, we also have to deal with integration over the boundaries of these domains; these are, correspondingly, surfaces, curves, and points.

When the heat diffusion problem was being formulated in two-dimensional domains in Section 5, the discrete domain consisted of triangles (elements T3), and the discrete boundary consisted of line segments (elements L2). Analogously, when the heat diffusion is solved in a one-dimensional domain (interval of the real line) which is covered by elements L2, the boundary consists of two points: hence the need for elements of type P1.

Evaluating the integrals of the surface heat transfer matrix (5.28) and the surface heats transfer load (5.27) (and also the prescribed heat flux load (5.26)) over the boundary on interval on the real line simply means taking the values of the integrants at the end points. In terms of a quadrature formula applied at the boundary point a (analogous to (2.24)),

$$f(a) \approx f(\xi_1)J(\xi_1)W_1$$

which is going to give the expected results with $\xi_1 = 0$, $f(\xi_1) = f(a)$, $W_1 = 1$, and the Jacobian $J(\xi_1) = 1$. The quadrature rule with these properties is `point_rule`, and a sample script to use this type of evaluation of boundary integrals for one-dimensional heat diffusion problems is `transcool`[1].

Programming remark: With the introduction of the element P1, a closure is achieved: the same code will now work for heat diffusion problems solved on one-dimensional, two-dimensional, and three-dimensional domains. The programming principles of object-oriented design that are in action here are polymorphism (the methods operate on objects in different types uniformly), and dynamic dispatch (an appropriate method is selected based on the class of the object on which it is invoked).

---

[1] Folder: `SOFEA/examples/diffusion`

8.4 Measuring (integrating) over domains

The uniform treatment of the manifold dimension of the domain allows us to produce dimension-independent code. Therefore, integration of any scalar function over any domain or subdomain is carried out by a single method of the class `feblock`. Consider as an example the geometry of a cylinder, the volume tiled with tetrahedra, the bounding surface covered with triangles, the edges of the cylindrical faces approximated with straight two-node segments, and one node at each vertex of the mesh. We may integrate over the volume of the 3-D mesh to find and approximation of the volume of the original cylinder; or over the length of a single edge to approximate the circumference; or over the area of one circular face to find an approximation of the cross-sectional area; or to count all the nodes on the cylindrical surface when we integrate over all the vertices of the triangles on that surface.

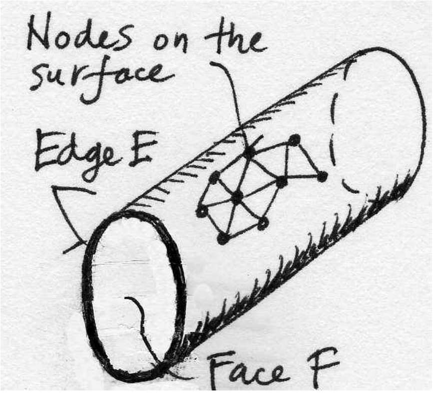


Fig. 8.4. Geometry of a cylinder.

The method `measure` of the class `feblock` evaluates the integral

$$\int_{V_n} f(\boldsymbol{x}) \, \mathrm{d}V_n \ , \tag{8.1}$$

where V_n is the volume of an n-dimensional manifold ($n = 0, 1, 2, 3$). The method takes as arguments the geometry field (evidently, the volume of any discrete manifold is going to depend on the locations of its vertices), and a function handle.

```
function result = measure[2] (self, geom, varargin)
    gcells =self.gcells;
    ngcells = length(self.gcells);
    % Integration rule
    integration_rule = get(self, 'integration_rule');
    pc = get(integration_rule, 'param_coords');
```

---

[2] Folder: `SOFEA/classes/feblock/@feblock`

```
    w  = get(integration_rule, 'weights');
    npts_per_gcell = get(integration_rule, 'npts');
```

When the function handle is not supplied, it is assumed that the function to be integrated over the manifold is $f(\boldsymbol{x}) = +1$; otherwise, it can be for instance the location-dependent mass density. There are a number of uses to which this method could be applied: as an example consider the calculation of the moments of inertia, or calculation of the centroid.

```
    if nargin >=3
        fh =varargin{1};
    else
        fh =@(x) (1);
    end
```

Loop over all geometric cells: collect the geometry from the supplied field.

```
    result = 0;
    % Now loop over all gcells in the block
    for i=1:ngcells
        conn = get(gcells(i), 'conn'); % connectivity
        x = gather(geom, conn, 'values', 'noreshape');
```

Each type of a geometric cell must provide functions for calculating the basis functions and the derivatives of the basis functions with respect to the parametric coordinates.

```
        % Loop over all integration points
        for j=1:npts_per_gcell
            N = Nmat(gcells(i), pc(j,:));
            Nder = Ndermat_param (gcells(i), pc(j,:));
```

Finally, evaluate the Jacobian, and accumulate the result using the numerical quadrature rule. It needs to be realized that again the function `Jacobian` is dispatched dynamically, to be treated differently in dependence on the dimension of the manifold. The second argument to `Jacobian` is an array of the tangent vectors to the parametric coordinates as columns.

```
            detJ = Jacobian[3](gcells(i),x'*Nder);
            result = result ...
                + fh(map_to_xyz(gcells(i),pc,x))*detJ*w(j);
        end
    end
end
```

As an example, here is the `Jacobian` method for a two-dimensional manifold (a surface). The number of space dimensions of the space `sdim` in which the manifold is embedded could be 2 (the manifold is just a piece of the Euclidean plane), or 3 (the manifold is then a piece of surface). The number of tangents must be 2 (compare with (5.42), and refer to Figure 8.5): they are

$$\frac{\partial \boldsymbol{x}(\xi,\eta)}{\partial \xi}, \quad \text{and} \quad \frac{\partial \boldsymbol{x}(\xi,\eta)}{\partial \eta}, \ .$$

The Jacobian is the length of the cross product of the two tangents (refer to the Figure 5.9). Here, the cross product is expressed through a skew-symmetric matrix.

---

[3] Folder: `SOFEA/classes/gcell/@gcell_`n`_manifold`, $n = 0, 1, 2, 3$

```
function detJ = Jacobian4(self, tangents)
    [sdim, ntan] = size(tangents);
    if      ntan==2 % 2-D gcell
        if sdim==ntan
            detJ = det(tangents);% Compute the Jacobian
        else
            detJ = norm(skewmat(tangents(1))*tangents(2));
        end
    else
        error('Got an incorrect size of tangents');
    end
end
```

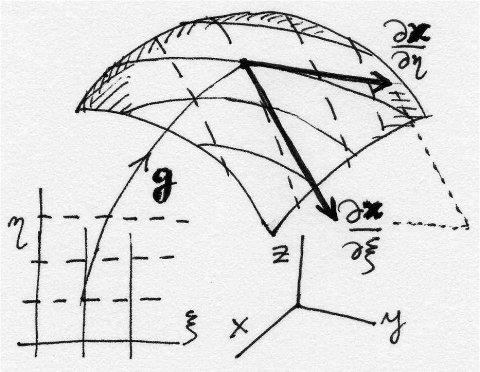

$\frac{\partial \mathbf{x}}{\partial \eta}$
$\mathbf{g}$
η
$\frac{\partial \mathbf{x}}{\partial \xi}$
z
ξ
x
y

Fig. 8.5. Surface with the coordinate curves and tangents

8.5 On the simplex elements

The point P1, the segment L2, the triangle T3, and the tetrahedron T4, are all examples of the so-called simplex elements. By definition, an n-dimensional simplex is the convex hull of $n + 1$ points (vertices) in the n-dimensional space. Tiling domains with simplex elements is attractive, because a number of mathematical properties guarantees the success of automatic tools for mesh generation. This is to be contrasted with the generation of quadrilaterals in two dimensions, and of bricks (shapes bounded by six quadrilateral faces) in three dimensions: not an easy task, where mesh generators often fail to produce good-quality meshes, or where they often just fail.

While the simplex elements perform adequately in the heat conduction models, in other types of analyses their inherent simplicity tends to work against them. For instance, as we shall see in linear elasticity the response of meshes composed of simplex elements is quite poorly represented – they are "too stiff".

---

[4] Folder: `SOFEA/classes/gcell/@gcell_2_manifold`

8.6 Quadrilateral Q4

8.7 Tetrahedron T4

The tetrahedron with four nodes at the corners (element T4) is a straightforward extension of the triangle T3. The standard tetrahedron is shown in Figure 8.6. The basis functions in the parametric coordinates are designed to be linear functions of ξ, η, ζ, and there are four corners at which to use the Kronecker delta property. It is straightforward to deduce that

$$N_1(\xi,\eta,\zeta) = 1-\xi-\eta-\zeta \;,\; N_2(\xi,\eta,\zeta) = \xi \;,\; N_3(\xi,\eta,\zeta) = \eta \;,\; N_4(\xi,\eta,\zeta) = \zeta \;. \quad (8.2)$$

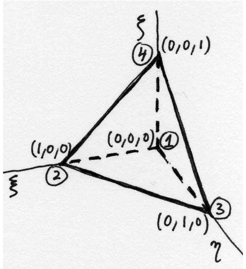


Fig. 8.6. Standard tetrahedron

Rule	Coordinates ξ_j, η_j, ζ_j	Weights W_j	Integrates exactly
1-point	1/4, 1/4, 1/4	1/6	linear polynomial
4-point	0.1381966, 0.1381966, 0.1381966	$0.041\overline{6}$	quadratic polynomial
	0.5854102, 0.1381966, 0.1381966	$0.041\overline{6}$	
	0.1381966, 0.5854102, 0.1381966	$0.041\overline{6}$	
	0.1381966, 0.1381966, 0.5854102	$0.041\overline{6}$	

Table 8.2. Numerical integration rules on the standard tetrahedron

The four basis functions of the tetrahedron each vanish along the opposite face (basis function N_i on the face opposite node i and so on). The remaining three vary along this face exactly as if it was a triangle T3. The situation is entirely analogous to the one discussed in Section 5.9 for the triangle T3 and the line segment L2. Therefore, evaluation of the surface heat transfer contributions is simply performed using geometric cells of type T3.

The script `helixcooled`[5] illustrates a solution with a full 3-D geometry discretized with the T4 tetrahedra. The problem is to determine steady state surface temperature for a helical spring, with variable cross-section– see Figure 8.7. The thick end is maintained at constant temperature, and on the rest of the surface there's convection cooling.

The mesh is a simple regular block tiled with tetrahedra, but it is then shaped by moving nodes to different locations using the utility `transform_apply`, first by changing its cross-section, and then by shifting all nodes in the y-direction. Finally, the shape is twisted into a helix using `transform_2_helix`.

---

[5] Folder: `SOFEA/examples/diffusion`

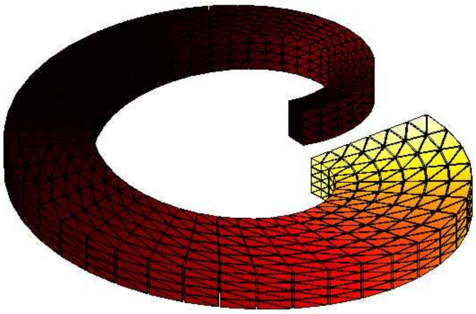


Fig. 8.7. The cooling of a helical spring.

```
[fens,gcells] = t4block(Angle,Width,Height, 50, 6, 4);
Radius = 1.2;
fens=transform_apply(fens,...
                  @(x,data)(x.*[1,(1-x(1)/Angle/1.2),1]),[]);
fens=transform_apply(fens,@(x,data)(x+ [0,Radius,0]),[]);
climbPerRevolution= 1.3;
fens = transform_2_helix(fens,climbPerRevolution);
```

The surface mesh consists of triangles T3, and is extracted from the tetrahedral mesh using the utility `simplex_mesh_bdry`. The surface mesh is immediately drawn with `drawmesh`.

```
bgcells=simplex_mesh_bdry(gcells);
drawmesh({fens,bgcells},'gcells','facecolor','red')
```

Next, the finite element blocks for the tetrahedral elements in the volume and the triangular elements on the surface are created. Note that the two blocks use different quadrature rules, `tet_rule` for the tetrahedra, and `tri_rule` for the triangles; both use just one integration point.

```
feb = feblock_diffusion (struct ('mater',mater,...
    'gcells',gcells,...
    'integration_rule',tet_rule(num_integ_pts)));
bfeb = feblock_diffusion (struct ('mater',mater,...
    'gcells',bgcells,...
    'integration_rule',tri_rule(num_integ_pts),...
    'surface_transfer', h));
```

From this point on, the script does not depend on the element types, be it the calculation of the system matrices, or graphics output.

9

Convergence and error control

9.1 First look at errors

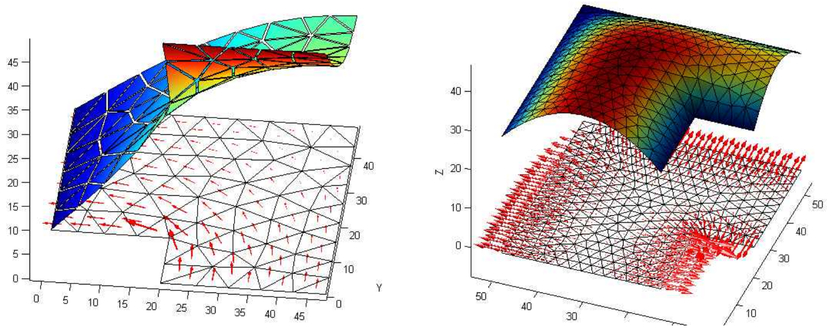


Fig. 9.1. The effect of a reentrant corner on the flux.

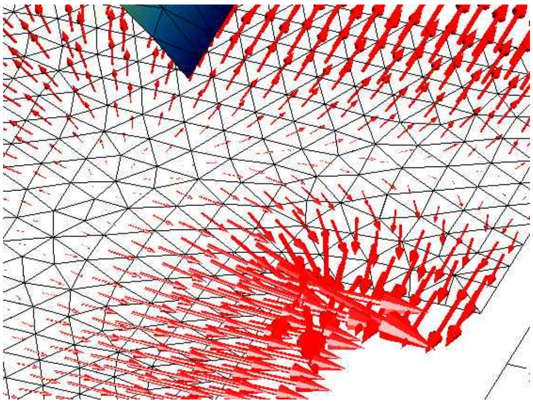


Fig. 9.2. The effect of a reentrant corner on the flux: close-up.

9.2 Richardson extrapolation

Richardson extrapolation is a way of extracting an asymptotic estimate of some quantity of interest from a series of computed values for it. If we assume that the error in the quantity q may be expanded in a Taylor series at mesh size $h = 0$, we may write the error in the remainder form as

$$E_q(h) = \tag{9.1}$$

9.3 The T4 NAFEMS Benchmark revisited

This problem has been discussed in Section 6.4. Cameron, Casey, and Simpson [CCS] cite the reference value for the temperature at the point indicated in Figure 6.5 of 18.3°C. However, more recent investigations of this benchmark indicate that value of 18.25°C should be expected [IL05]. Let us check these numbers.

Two models will be used, the first using elements T3, and the second using the more accurate quadratic elements T6.

18.25396

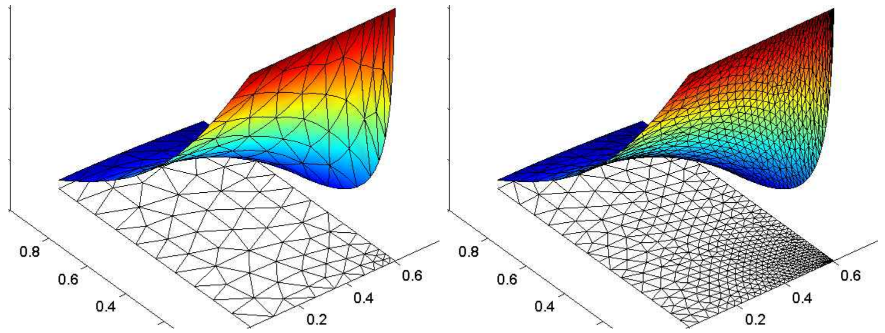


Fig. 9.3. T4 NAFEMS Benchmark: solution with quadratic elements, initial and final mesh.

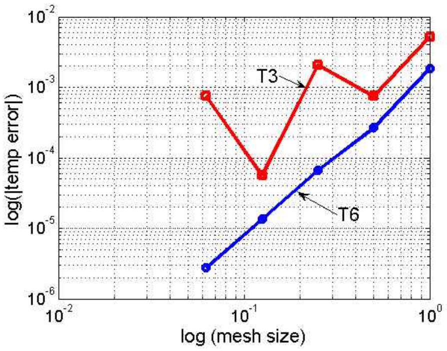


Fig. 9.4. T4 NAFEMS Benchmark: solution with quadratic elements, initial and final mesh.

9.4 Shrink fitting revisited

Figure 9.5 shows the temperature distribution at three time instants. The extremely high gradient at the beginning is evident, but in fact high temperature gradients exist even at the end of the process.

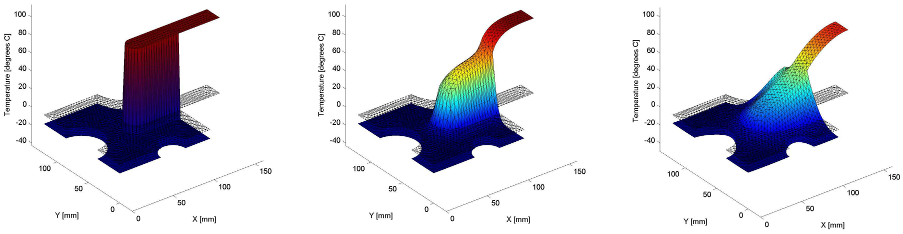


Fig. 9.5. Transient cooling of a shrink-fitted assembly; left to right: temperature distribution for time $t = 0, 2, 13$

As you recall, the heat flux is derived from the temperature (equation (4.14)). The finite element approximation with the triangles (T3) and with the line elements (L2) will be able to reproduce linearly varying temperatures, hence constant temperature gradients (i.e. heat flux). Therefore, we will conclude that where the heat flux changes, the finite element approximation will be in error. To control the error, we can reduce the element dimensions. Doing so in areas of steep changes in the heat flux, while keeping areas with approximately uniform heat flux tiled with coarse elements, is known as ***adaptive mesh control***.

Figure 9.6 shows the heat flux on two meshes as arrows centered at the barycenters of the elements (barycenter here means average of the vertex locations). The first mesh is quite coarse (script `shrinkfitad1`[1]), but it is possible to identify regions in which the gradient changes strongly (next to the tungsten inset); the adaptive mesh is generated to reflect the demand for finer (smaller) elements (script `shrinkfitad2`[2]).

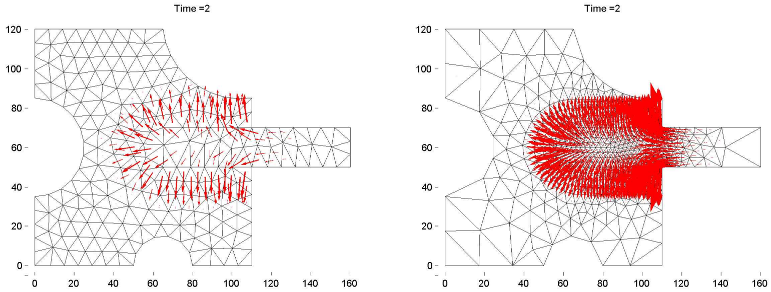


Fig. 9.6. Transient cooling of a shrink-fitted assembly; left: coarse mesh, right: adaptive mesh. Heat flux for time $t = 2$

The temperature evolution obtained with the two meshes, the coarse one, and the adaptively refined one, is illustrated in Figure 9.7, and the higher-quality of the adaptive results should be noted: especially striking is the spurious oscillation of the lowest temperature for the coarse mesh.

---

[1] Folder: `SOFEA/examples/diffusion`

[2] Folder: `SOFEA/examples/diffusion`

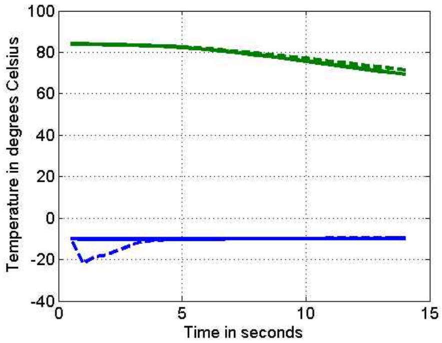


Fig. 9.7. Transient cooling of a shrink-fitted assembly: time evolution of the lowest and highest temperature in the assembly. Comparing temperatures obtained with a coarse model (dashed lines) and with an adaptively refined model (solid lines).

Part III

Stress analysis

10

Model of elastodynamics

10.1 Balance equation

References

Gra91. Graff, KF: Wave motion in elastic solids. Dover publications, New York (1991)

Hug00. Hughes, TJR: The finite element method. Linear static and dynamic finite element analysis. Dover publications, New York (2000).
This book is a must-read for anyone serious about learning finite element analysis properly. Meticulously written, and brimming with insights. Quite mathematical.

CC2005. Chapra, SC and RP Canale: Numerical Methods for Engineers. McGraw-Hill (2005)

CCS. A.D. CAMERON, J.A. CASEY, G.B. SIMPSON: Benchmark Tests for Thermal Analyses, NAFEMS Documentation.

IL05. `http://www.infolytica.com/en/coolstuff/ex0047/`

Index

`Jacobian`, 73

adaptive mesh control, 79
argument
  passed by value, 21

balance equation, 3
  global, 26
  local, 26
balance residual, 7
barycenter, 79
basis function
  derivative, 42
  triangle T3, 35
  triangle T6, 70
basis functions, 11
boundary condition
  essential, 5, 28
  natural, 5, 28
boundary conditions, 4
Bubnov-Galerkin method, 7

capacity matrix, 39
chain rule, 16
change of coordinates in integrals, 44
circular frequency, 22
class
  `body_load`, 21
  `dense_sysmat`, 21
  `elemat`, 45
  `feblock_diffusion`, 44, 52
  `feblock`, 20, 44, 72
  `field`, 20, 64
  `gcell_T3`, 42
  `gcell`, 42
  `sysvec`, 21
  `tet_rule`, 76
  `tri_rule`, 52, 76
  `fenode`, 20
Clough, 34
conductivity matrix, 39
connectivity, 20
constructor, 20
control volume, 25
convex hull, 74
Courant, 34
Crank-Nicholson method, 62
cross product, 43, 73
curved boundary
  approximation, 38

degree of freedom, 14
diagonal mass matrix, 17
Dirac delta function, 9
divergence theorem, 26
dynamic dispatch, 72

eigenmode, 22
error
  heat flux, 79
essential boundary condition, 5

finite difference
  backward Euler, 62
  forward Euler, 62
finite element, 13
  edge, 34
  isoparametric, 40
  L2, 13
  node, 13, 34
  T3, 42
  T6, 69
finite element mesh, 13
free vibration, 22

Galerkin method, 7
generalized eigenvalue problem, 22
generalized trapezoidal method, 61, 62
geometric cells, 20

hat function, 34
heat

  conduction, 25
  diffusion, 25
heat energy, 25
heat flux, 25

initial boundary value problem, 6
initial conditions, 5
integration by parts, 10
integration rule
  tetrahedron, 75
  triangle, 44, 70
interpolation, 11
isoparametric element, 40
isotropic material, 27

Jacobian, 44
Jacobian determinant, 15
Jacobian matrix, 41, 43

Kronecker delta, 14, 69

Lagrange interpolation polynomial, 13

manifold dimension, 71, 72
map
  of areas, 44
  of points, 43
  of vectors, 43
mass matrix, 12
Matlab script
  `helixcooled`, 75
  `lshape1`, 52
  `shrinkfit`, 65
  `squareinsquare`, 54
  `t3nafems`, 63
  `t4nafems`, 56
  `transcool`, 71
method
  backward Euler, 62
  Crank-Nicholson, 62
  forward Euler, 62

natural boundary conditions, 5
Newmark explicit algorithm, 17
Newton's law, 3
node, 13
numerical quadrature, 14
  point, 71
  triangles, 44

ordinary differential equations, 12
orthotropic material, 27
outer normal, 25

parametric coordinates, 16
piecewise linear, 9
piecewise linear approximation, 12
polymorphism, 72

quadrature point, 15

rate of heat generation, 26
reduction
  dimension, 32
residual, 7, 10
Richardson extrapolation, 78

simplex element, 74
Simpson's 1/3 rule, 15
skew-symmetric matrix, 73
smoothness, 10
sparse matrix, 12
specific heat, 26
standard interval, 15, 47
standard tetrahedron, 75
standard triangle, 35, 69
static equilibrium, 19
stiffness matrix, 12
surface heat transfer matrix, 40
symmetric matrix, 12

tangent vector, 43, 48, 73
taut string, 3
temperature gradient, 27
tent, 34
test function, 8
thermal conductivity, 27
transformation matrix, 46
trial function, 10, 11
trial-and-test approximate method, 9
triangulation, 34

utility
  `drawmesh`, 76
  `simplex_mesh_bdry`, 76
  `transform_apply`, 75

weighted residual method, 8